Język asembler dla każdego

Bogdan Drozdowski

Jak pisać programy w języku asembler?

Wstęp.

Pewnie wiele osób spośród Was słyszało już o tym, jaki ten asembler jest straszny. Ale te opinie możecie już śmiało wyrzucić do Kosza (użytkownicy Linuksa mogą skorzystać z /dev/null), gdyż przyszła pora na przeczytanie tego oto dokumentu.

Na początek, przyjrzyjmy się niewątpliwym zaletom języka asembler:
(przeskocz zalety)

1. Mały rozmiar kodu.
   Tego nie przebije żaden kompilator żadnego innego języka. Dlaczego? Oto kilka powodów:
   - Jako programista języka asembler (asma) wiesz co i gdzie w danej chwili się znajduje. Nie musisz ciągle przeładowywać zmiennych, co zabiera i miejsce i czas. Możesz eliminować instrukcje, które są po prostu zbędne.
   - Do twojego pięknego kodu nie są dołączane żadne biblioteki z dziesiątkami procedur, podczas gdy ty używasz tylko jednej z nich. Co za marnotrawstwo!
   - Jako znawca zestawu instrukcji wiesz, które z nich są krótsze.

2. Duża szybkość działania.
   Znając sztuczki optymalizacyjne, wiedząc, które instrukcje są szybsze, eliminując zbędne instrukcje z pętli, otrzymujesz kod nierzadko dziesiątki razy szybszy od tych napisanych w językach wysokiego poziomu (high-level languages, HLLs). Nieprzebijalne.

3. Wiedza.
   Nawet jeśli nie piszesz dużo programów w asmie, zdobywasz wprost bezcenną wiedzę o tym, jak naprawdę działa komputer i możesz zobaczyć, jak marną czasem robotę wykonują kompilatory HLL-ów. Zrozumiesz, czym jest wskaźnik i często popełniane błędy z nim związane.

4. Możliwość tworzenia zmiennych o dużych rozmiarach, a nie ograniczonych do 4 czy 8 bajtów. W asemblerze zmienne mogą mieć dowolną ilość bajtów.

5. Wstawki asemblerowe.
   Jeśli mimo to nie chcesz porzucić swojego ulubionego dotąd HLLa, to w niektórych językach istnieje możliwość wstawianie kodu napisanego w asemblerze wprost do twoich programów!

Teraz przyszła kolej na rzekome argumenty przeciwko językowi asembler:
(przeskocz wady)

1. Nieprzenośność kodu między różnymi maszynami.
   No cóż, prawda. Ale i tak większość tego, co napisane dla procesorów Intela będzie działało na procesorach AMD i innych zgodnych z x86. I na odwrót.
   Nieprzenośność jest chyba najczęściej używanym argumentem przeciwko asemblerowi. Jest on

zwykle stawiany przez programistów języka C, którzy po udowodnieniu, jaki to język C jest wspaniały, wracają do pisania dokładnie w takim samym stopniu nie-przenośnych programów... Nie ukrywajmy: bez zmian kodu to tylko programy niewiele przewyższające Witaj, świecie skompilują się i uruchomią pod różnymi systemami.

2. A nowoczesne kompilatory i tak produkują najlepszy kod...

   Nieprawda, i to z kilku powodów:

   - Kompilator używa zmiennych. No i co z tego, pytacie? A to, że pamięć RAM (o dyskach itp. nie wspominając) jest wiele, wiele razy wolniejsza niż pamięć procesora (czyli rejestry). Nawet pamięć podręczna (cache) jest sporo wolniejsza.

   - Kompilator nie wie, co się znajduje np. w danym rejestrze procesora, więc pewnie wpisze tam tą samą wartość. Co innego z programistą asma.

   - Kompilator nie jest w stanie przewidzieć, co będzie w danej chwili w innych rejestrach. Więc do rejestru, który chce zmienić i tak wpisze jakąś wartość zamiast użyć innego rejestru, co prawie zawsze jest szybsze a więc lepsze. Co innego zrobiłby programista asma.

   - Kompilator może używać dłuższych lub wolniejszych instrukcji.

   - Kompilator nie zawsze może poprzestawiać instrukcje, aby uzyskać lepszy kod. Programista asma widzi, co się w jego kodzie dzieje i może wybrać inne, lepsze rozwiązanie (np. zmniejszyć rozmiary pętli czy pozbyć się zależności między instrukcjami)

   - Kompilator może nie być świadomy technologii zawartych w procesorze. Programista asma wie, jaki ma procesor i za darmo ściąga do niego pełną dokumentację.

3. Brak bibliotek standardowych.
   I znowu nieprawda. Istnieje co najmniej kilka takich. Zawierają procedury wejścia, wyjścia, alokacji pamięci i wiele, wiele innych. Nawet sam taką jedną bibliotekę napisałem...

4. Kod wygląda dziwniej. Jest bardziej abstrakcyjny.
   Dziwniej - tak, ale nie oszukujmy się. To właśnie języki wysokiego poziomu są abstrakcyjne! Asembler przecież operuje na tym, co fizycznie istnieje w procesorze - na jego własnych rejestrach przy użyciu jego własnych instrukcji.

5. Mniejsza czytelność kodu.
   Kod w języku C można tak napisać, że nie zrozumie go nawet sam autor. Kod w asmie można tak napisać, że każdy go zrozumie. Wystarczy kilka słów wstępu i komentarze. W HLLach trzeba byłoby wszystkie struktury objaśniać.
   A wygląd i czytelność kodu zależą tylko od tego, czy dany programista jest dobry, czy nie.
   Dobry programista asemblera nie będzie miał większych kłopotów z odczytaniem kodu w asmie niż dobry programista C kodu napisanego w C.

6. Brak łatwych do zrozumienia instrukcji sterujących (if, while, ...)
   Przecież w procesorze nie ma nic takiego!
   Programista asma ma 2 wyjścia: albo używać prawdziwych instrukcji albo napisać własne makra,

które takie instrukcje będą udawać (już są takie napisane). Ale nie ma nic uniwersalnego. Na jedną okazję można użyć takich instrukcji, a na inną - innych. Jednak zawsze można wybrać najszybszą wersję według własnego zdania, a nie według zdania kompilatora.
Asembler może i nie jest z początku łatwy do zrozumienia, ale wszystko przyjdzie wraz z doświadczeniem.

7. Trzeba pisać dużo kodu.
   No, tak. Jak się komuś męczą palce, to niech zostanie przy HLLach i żyje w świecie abstrakcji. Prawdziwym programistom nie będzie przecież takie coś przeszkadzać!
   Mówi się, że ZŁEJ baletnicy nawet rąbek u sukni przeszkadza.
   Poza tym, programista nad samym pisaniem kodu spędza ok 30% czasu przeznaczonego na program (reszta to plan wykonania, wdrażanie, utrzymanie, testowanie...). Nawet jeśli programiście asma zabiera to 2 razy więcej czasu niż programiście HLLa, to i tak zysk lub strata wynosi 15%.
   Dużo pisania sprawia, że umysł się uczy, zapamiętuje składnię instrukcji i nabiera doświadczenia.

8. Assmebler jest ciężki do nauki.
   Jak każdy nowy język. Nauka C lub innych podobnych przychodzi łatwo, gdy już się zna np. Pascala. Próba nauczenia się innych dziwnych języków zajmie dłużej, niż nauka asma.

9. Ciężko znajdować i usuwać błędy.
   Na początku równie ciężko, jak w innych językach. Pamiętacie jeszcze usuwanie błędów ze swoich pierwszych programów w C czy Pascalu?

10. Programy w asemblerze są ciężkie w utrzymaniu.
    Znowu powiem to samo: podobnie jest w innych językach. Najlepiej dany program zna jego autor, co wcale nie oznacza, że w przyszłości będzie dalej rozumiał swój kod (nawet napisany w jakimś HLLu). Dlatego ważne są komentarze. Zdolność do zajmowania się programem w przyszłości także przychodzi wraz z doświadczeniem.

11. Nowoczesne komputery są tak szybkie, że i tak szybkość nie robi to różnicy...
    Napiszmy program z czterema zagnieżdżonymi pętlami po 100 powtórzeń każda. Razem 100 000 000 (sto milionów) powtórzeń. Czas wykonania tego programu napisanego w jakimś HLLu liczy się w minutach, a często w dziesiątkach minut (czasem godzin - zależy od tego, co jest w pętlach). To samo zadanie napisane w asemblerze daje program, którego czas działania można liczyć w sekundach lub pojedynczych minutach!
    Po prostu najszybsze programy są pisane w asemblerze. Często otrzymuje się program 5-10 razy szybszy (lub jeszcze szybszy) niż ten w HLLu.

12. Chcesz mieć szybki program? Zmień algorytm, a nie język
    A co jeśli używasz już najszybszego algorytmu a on i tak działa za wolno?
    Każdy algorytm zawsze można zapisać w asemblerze, co poprawi jego wydajność. Nie wszystko da się zrobić w HLLu.

13. Nowoczesne komputery i tak mają dużo pamięci. Po co więc mniejsze programy?
    Wolisz mieć 1 wolno działający program o rozmiarze 1 MB, napisany w HLLu i robić 1 czynność w danej chwili, czy może wolisz wykonywać 10 czynności na raz dziesięcioma programami w asemblerze po 100kB każdy (no, przesadziłem - rzadko który program w asmie sięgnie aż tak gigantycznych rozmiarów!)?

To był tylko wstęp do bezkresnej wiedzy, jaką każdy z Was zdobędzie.

Ale nie myślcie, że całkowicie odradzam Wam języki wysokiego poziomu. Ja po prostu polecam Wam asemblera.

Najlepsze programy pisze się w czystym asemblerze, co sprawia niesamowitą radość, ale można przecież łączyć języki. Na przykład, część programu odpowiedzialną za wczytywanie danych lub wyświetlanie wyników można napisać w HLLu, a intensywne obliczeniowo pętle pisać w asmie, albo robiąc wstawki w kod, albo pisząc w ogóle oddzielne moduły i potem łączyć wszystko w całość.

Nauka tego wspaniałego języka przyjdzie Wam łatwiej, niż myślicie. Pomyślcie też, co powiedzą znajomi, gdy się dowiedzą, co umiecie!

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Informacja dla użytkowników systemów *BSD

Korzystanie z usług systemowych (przerwania 80h) w systemach klasy BSD różni się nieco od sposobu używanego w zwyczajnych Linuksach. Mianowicie:

1. numer funkcji przekazujemy w EAX
2. parametry wkładamy na stos od prawej do lewej (od końca)
3. przerwanie wywołujemy, wykonując CALL do instrukcji `int 80h`, za którą jest RET

Żeby wszystko było jasne, podam teraz przykład:
Linux:

```
; wypisywanie tekstu na ekranie:

        mov     eax, 4
        mov     ebx, 1
        mov     ecx, tekst
        mov     edx, tekst_dlugosc
        int     80h
```

BSD:

```
; wypisywanie tekstu na ekranie:

        mov     eax, 4
        push    dword tekst_dlugosc
        push    dword tekst
        push    dword 1
        call    jadro
        add     esp, 12
        ...
        ...
jadro:
        int     80h
        ret
```

Oczywiście, ta sama procedura jadro może służyć więcej niż jednemu wywołaniu przerwania systemowego.

Jeśli przy próbie uruchomienia programu dostajecie komunikat Operation not permitted (Operacja niedozwolona), to dodajcie do kodu programu nową sekcję:

```
section .note.openbsd.ident align=4
   dd 8
   dd 4
   dd 1
   db 'OpenBSD', 0
   dd 0
```

Teraz program można kompilować i linkować normalnie, tzn. linkerem LD (tak jest pokazane dalej w kursie). Podziękowania dla 'Fr3m3n' za zgłoszenie tego sposobu.

Innym wyjściem jest skorzystanie z kompilatora GCC, zamiast linkera LD: `gcc -o program program.o`. Funkcja główna programu (miejsce rozpoczęcia wykonywania się programu) musi się wtedy nazywać main, a nie _start! Wadą tego podejścia jest to, że do programu zostają dołączone pewne specjalne

pliki, co powiększa jego rozmiar.

Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Jak pisać programy w języku asembler pod Linuksem?

Część 1 - Podstawy, czyli czym to się je.

Wyobraźcie sobie, jakby to było móc programować maszynę bezpośrednio - rozmawiać z procesorem bez pośrednictwa struktur wysokiego poziomu, np. takich jak spotykamy w języku C. Bezpośrednie operowanie na procesorze umożliwia przecież pełną kontrolę jego działań! Bez zbędnych instrukcji i innych śmieci spowalniających nasze programy.

Czy już czujecie chęć pisania najkrótszych i najszybszych programów na świecie?
Programów, których czasem w ogóle NIE MOŻNA napisać w innych językach? Brzmi wspaniale, prawda?

Tylko pomyślcie o tym, co powiedzieliby znajomi, gdybyście się im pochwalili. Widzicie już te ich zdumione miny?

Miła perspektywa, prawda? No, ale dość już gadania. Zabierajmy się do rzeczy!

Zacznijmy od krótkiego wprowadzenia:

Niedziesiętne systemy liczenia.

1. Dwójkowy (binarny)
   Najprostszy dla komputera, gdzie coś jest albo włączone, albo wyłączone. System ten operuje na liczbach zwanych bitami (bit = binary digit = cyfra dwójkowa). Bit przyjmuje jedną z dwóch wartości: 0 lub 1.
   Na bajt składa się 8 bitów. Jednym bajtem można przedstawić więc 2^8=256 możliwości.

   Przeliczenie liczby zapisanej w systemie dwójkowym na dziesiętny jest proste. Podobnie jak w systemie dziesiętnym, każdą cyfrę mnożymy przez odpowiednią potęgę podstawy (podstawa wynosi 2 w systemie dwójkowym, 10 w systemie dziesiętnym).

   Oto przykład (daszek ^ oznacza potęgowanie):
   1010 1001 dwójkowo =
   1*(2^7) + 0*(2^6) + 1*(2^5) + 0*(2^4) + 1*(2^3) + 0*(2^2) + 0*(2^1) + 1*(2^0) =
   128 + 32 + 8 + 1 =
   169 dziesiętnie (lub dec, od decimal).

   Działanie odwrotne też nie jest trudne: naszą liczbę dzielimy ciągle (do chwili uzyskania ilorazu równego 0) przez 2, po czym zapisujemy reszty z dzielenia wspak:
   (przeskocz konwersję liczby dziesiętnej na dwójkową)

```
169     |
 84     | 1
 42     | 0
 21     | 0
```

```
10      | 1
 5      | 0
 2      | 1
 1      | 0
 0      | 1
```

Wspak dostajemy: 1010 1001, czyli wyjściową liczbę.

2. Szesnastkowy (heksadecymalny, w skrócie hex)
   Jako że system dwójkowy ma mniej cyfr niż dziesiętny, do przedstawienia względnie małych liczb trzeba użyć dużo zer i jedynek. Jako że bajt ma 8 bitów, podzielono go na dwie równe, 4-bitowe części. Teraz bajt można już reprezentować dwoma znakami, a nie ośmioma. Na każdy taki znak składa się 2^4=16 możliwości. Stąd wzięła się nazwa szesnastkowy.
   Powstał jednak problem: cyfr jest tylko 10, a trzeba mieć 16. Co zrobić?
   Postanowiono liczbom 10-15 przyporządkować odpowiednio znaki A-F.
   Np.
   Liczba 255 dziesiętnie = 1111 1111 binarnie = FF szesnastkowo (1111 bin = 15 dec = F hex)
   Liczba 150 dziesiętnie = 1001 0110 binarnie = 96 szesnastkowo.

   Należy zauważyć ścisły związek między systemem dwójkowym i szesnastkowym: 1 cyfra szesnastkowa to 4 bity, co umożliwia błyskawiczne przeliczanie między obydwoma systemami: wystarczy tłumaczyć po 4 bity (1 cyfrę hex) na raz i zrobione.

   Przeliczenie liczby zapisanej w systemie szesnastkowym na dziesiętny jest równie proste, jak tłumaczenie z dwójkowego na dziesiętny. Każdą cyfrę mnożymy przez odpowiednią potęgę podstawy (podstawa wynosi 16 w systemie szesnastkowym).

   Oto przykład:
   10A szesnastkowo =
   1*16^2 + 0*16^1 + A*16^0 =
   256 + 0 + 10 =
   266 dziesiętnie.

   Działanie odwrotne też nie jest trudne: naszą liczbę dzielimy ciągle (do chwili uzyskania ilorazu równego 0) przez 16, po czym zapisujemy reszty z dzielenia wspak:
   (przeskocz konwersję liczby dziesiętnej na szesnastkową)

```
266     |
16      | 10
 1      | 0
 0      | 1
```

   Wspak dostajemy kolejno: 1, 0, 10, czyli 10A, czyli wyjściową liczbę.

   Podczas pisania programów, liczby w systemie szesnastkowym oznacza się przez dodanie na końcu litery h (lub z przodu 0x), a liczby w systemie dwójkowym - przez dodanie litery b na końcu.
   Tak więc, 101 oznacza dziesiętną liczbę o wartości 101, 101b oznacza liczbę 101 w systemie dwójkowym (czyli 5 w systemie dziesiętnym), a 101h lub 0x101 oznacza liczbę 101 w systemie szesnastkowym (czyli 257 dziesiętnie).

Język asembler i rejestry procesora

Co to jest asembler?
Asembler jest to język programowania, należący do języków niskiego poziomu. Znaczy to tyle, że jednej komendzie asemblera odpowiada dokładnie jeden rozkaz procesora. Asembler operuje na rejestrach procesora.

A co to jest rejestr procesora?
Rejestr procesora to zespół układów elektronicznych, mogący przechowywać informacje (taka własna pamięć wewnętrzna procesora).
Zaraz podam Wam podstawowe rejestry, na których będziemy operować. Wiem, że ich ilość może przerazić, ale od razu mówię, abyście *NIE uczyli się tego wszystkiego na pamięć*! Najlepiej zrobicie, czytając poniższą listę tylko 2 razy, a potem wracali do niej, gdy jakikolwiek rejestr pojawi się w programach, które będę później prezentował w ramach tego kursu.
Oto lista interesujących nas rejestrów:

1. ogólnego użytku:

   - akumulator:
     RAX (64 bity) = EAX (młodsze 32 bity) + starsze 32 bity,
     EAX (32 bity) = AX (młodsze 16 bitów) + starsze 16 bitów,
     AX = AH (starsze 8 bitów) + AL (młodsze 8 bitów)
     Rejestr ten służy do wykonywania działań matematycznych, ale często w tym rejestrze będziemy mówić systemowi operacyjnemu, co od niego chcemy.

   - bazowy:
     RBX (64 bity) = EBX (młodsze 32 bity) + starsze 32 bity,
     EBX (32 bity) = BX (młodsze 16 bitów) + starsze 16 bitów,
     BX = BH (starsze 8 bitów) + BL (młodsze 8 bitów)
     Ten rejestr jest używany np. przy dostępie do tablic.

   - licznik:
     RCX (64 bity) = ECX (młodsze 32 bity) + starsze 32 bity,
     ECX (32 bity) = CX (młodsze 16 bitów) + starsze 16 bitów,
     CX = CH (starsze 8 bitów) + CL (młodsze 8 bitów)
     Tego rejestru używamy np. do określania ilości powtórzeń pętli.

   - rejestr danych:
     RDX (64 bity) = EDX (młodsze 32 bity) + starsze 32 bity,
     EDX (32 bity) = DX (młodsze 16 bitów) + starsze 16 bitów,
     DX = DH (starsze 8 bitów) + DL (młodsze 8 bitów)
     W tym rejestrze np. przechowujemy adresy różnych zmiennych.

   - rejestry dostępne tylko w trybie 64-bitowym:
     - 8 rejestrów 8-bitowych: R8B, ..., R15B
     - 8 rejestrów 16-bitowych: R8W, ..., R15W
     - 8 rejestrów 32-bitowych: R8D, ..., R15D

◊ 8 rejestrów 64-bitowych: R8, ..., R15

- rejestry indeksowe:
  - ◊ indeks źródłowy:
    RSI (64 bity) = ESI (młodsze 32 bity) + starsze 32 bity,
    ESI (32 bity) = SI (młodsze 16 bitów) + starsze 16 bitów,
    SI (16 bitów) = SIL (młodsze 8 bitów) + starsze 8 bitów (tylko tryb 64-bit)

  - ◊ indeks docelowy:
    RDI (64 bity) = EDI (młodsze 32 bity) + starsze 32 bity,
    EDI (32 bity) = DI (młodsze 16 bitów) + starsze 16 bitów,
    DI (16 bitów) = DIL (młodsze 8 bitów) + starsze 8 bitów (tylko tryb 64-bit)

  Rejestry indeksowe najczęściej służą do operacji na długich łańcuchach danych, w tym napisach i tablicach.

- rejestry wskaźnikowe:
  - ◊ wskaźnik bazowy:
    RBP (64 bity) = EBP (młodsze 32 bity) + starsze 32 bity,
    EBP (32 bity) = BP (młodsze 16 bitów) + starsze 16 bitów.
    BP (16 bitów) = BPL (młodsze 8 bitów) + starsze 8 bitów (tylko tryb 64-bit)
    Najczęściej służy do dostępu do zmiennych lokalnych danej funkcji.

  - ◊ wskaźnik stosu:
    RSP (64 bity) = ESP (młodsze 32 bity) + starsze 32 bity,
    ESP (32 bity) = SP (młodsze 16 bitów) + starsze 16 bitów.
    SP (16 bitów) = SPL (młodsze 8 bitów) + starsze 8 bitów (tylko tryb 64-bit)
    Służy do dostępu do stosu (o tym nieco później).

  - ◊ wskaźnik instrukcji:
    RIP (64 bity) = EIP (młodsze 32 bity) + starsze 32 bity,
    EIP (32 bity) = IP (młodsze 16 bitów) + starsze 16 bitów.
    Mówi procesorowi, skąd ma pobierać instrukcje do wykonywania.

2. rejestry segmentowe (wszystkie 16-bitowe) - tych najlepiej nie dotykać w Linuksie:

   - segment kodu CS - mówi procesorowi, gdzie znajdują się dla niego instrukcje.
   - segment danych DS - ten najczęściej pokazuje na miejsce, gdzie trzymamy nasze zmienne.
   - segment stosu SS - dzięki niemu wiemy, w którym segmencie jest nasz stos. O tym, czym w ogóle jest stos, powiem w następnej części.
   - segment dodatkowy ES - zazwyczaj pokazuje na to samo, co DS - na nasze zmienne.
   - FS i GS - nie mają specjalnego przeznaczenia. Są tu na wypadek, gdyby zabrakło nam innych rejestrów segmentowych.

3. rejestr stanu procesora: FLAGI (16-bitowe), E-FLAGI (32-bitowe) lub R-FLAGI (64-bitowe). Służą one przede wszystkim do badania wyniku ostatniego przekształcenia (np. czy nie wystąpiło przepełnienie, czy wynik jest zerem, itp.). Najważniejsze flagi to CF (carry flag - flaga przeniesienia), OF (overflow flag - flaga przepełnienia), SF (sign flag - flaga znaku), ZF (zero flag - flaga zera), IF (interrupt flag - flaga przerwań), PF (parity flag - flaga parzystości), DF (direction flag - flaga kierunku).

Użycie litery R przed symbolem rejestru, np. RCX, oznacza rejestr 64-bitowy, dostępny tylko na procesorach 64-bitowych.
Użycie litery E przed symbolem rejestru, np. EAX, oznacza rejestr 32-bitowy, dostępny tylko na procesorach rodziny 80386 lub lepszych. Nie dotyczy to rejestru ES.

Napisy

> RAX = EAX+starsze 32 bity; EAX=AX+ starsze 16 bitów; AX=AH+AL

oznaczają:
(przeskocz rozwinięcie rejestru RAX)

```
                              RAX (64 bity)
                                   |                  EAX (32b)
00000000000000000000000000000000   | 0000000000000000 | 00000000 | 00000000
                32b                |        16b       |        AX(16b)
                                   |                  |  AH(8b)  |   AL(8b)
```

Napisy

> RSI = ESI+starsze 32 bity; ESI = SI + starsze 16 bitów; SI = SIL+starsze 8 bitów

oznaczają:
(przeskocz rozwinięcie rejestru RSI)

```
                              RSI (64 bity)
                                   |                ESI (32b)
00000000000000000000000000000000   | 0000000000000000 | 00000000 | 00000000
                32b                |       16b        |        SI(16b)
                                   |                  |   8b     |  SIL(8b)
```

Jedna ważna uwaga - między nazwami rejestrów może pojawić się dwukropek w dwóch różnych znaczeniach:

- zapis DX : AX (lub 2 dowolne zwykłe rejestry) będzie oznaczać liczbę, której starsza część znajduje się w rejestrze po lewej stronie (DX), a młodsza - w tym z prawej (AX). Wartość liczby wynosi DX*65536 + AX.
- zapis CS : SI (rejestr segmentowy + dowolny zwykły) będzie najczęściej oznaczać wskaźnik do jakiegoś obiektu w pamięci (o pamięci opowiem następnym razem). Rejestr segmentowy zawiera oczywiście segment, w którym znajduje się ów obiekt, a rejestr zwykły - offset (przesunięcie, adres w tym segmencie) tegoż obiektu.

Na razie nie musicie się przejmować tymi dwukropkami. Mówię to tylko dlatego, żebyście nie byli zaskoczeni, gdyż w przyszłości się pojawią.

Programista może odnosić się bezpośrednio do wszystkich wymienionych rejestrów, z wyjątkiem *IP oraz flag procesora (z wyjątkami).

Jak widać po ich rozmiarach, do rejestrów 8-bitowych można wpisać liczbę z przedziału 0-255 (lub od -128 do 127, gdy najwyższy, siódmy bit służy nam jako bit oznaczający znak liczby), w 16-bitowych zmieszczą się liczby 0-65535 (od -32768 do 32767), a w 32-bitowych - liczby od 0 do 4.294.967.295 (od -2.147.483.648 do 2.147.483.647)

Dobrym, choć trudnym w odbiorze źródłem informacji są: *Intel Architecture Software Developer's Manual* (IASDM) dostępny *ZA DARMO* ze stron Intela oraz *DARMOWE* podręczniki *AMD64 Architecture Programmer's Manual* firmy AMD

Pisanie i kompilowanie (asemblowanie) swoich programów

Jak pisać programy w asemblerze?
Należy zaopatrzyć się w:

- Edytor tekstu, mogący zapisywać pliki tekstowe (bez formatowania), np. VIM, LPE, Emacs/XEmacs, Joe, Pico, Jed, Kate, KWrite.
- Kompilator języka asembler (patrz dalej)
- Odpowiedni program łączący (konsolidator, ang. linker), chyba że kompilator ma już taki wbudowany, jak np. FASM

Wtedy wystarczy napisać w edytorze tekstu plik zawierający komendy procesora (o tym później), zapisać go z rozszerzeniem .asm lub .s (GNU as), po czym użyć kompilatora, aby przetworzyć program na kod rozumiany przez procesor.

Jakiego kompilatora użyć?
Istnieje wiele kompilatorów języka asembler pod Linuksa. Do najpopularniejszych należą Netwide Asembler Project (NASM), Flat Asembler (FASM), High-Level Asembler (HLA) i Gnu As.

Można je ściągnąć z internetu:
(przeskocz adresy stron kompilatorów)

- NASM: sf.net/projects/nasm
- FASM: flatasembler.net
- HLA: webster.cs.ucr.edu
- Gnu Asembler znajduje się w pakiecie binutils (powinien być w każdej dystrybucji).

Po skompilowaniu pliku z kodem źródłowym należy użyć programu łączącego - będziemy używać standardowego LD (tego, którego używają inne kompilatory), gdyż również powinien się znajdować w każdej dystrybucji Linuksa. Mamy więc już wszystko, co potrzeba. Zaczynamy pisać. Będę tutaj używał składni NASMa i FASMa (gdyż kompilują one programy w składni Intela, która jest bardziej przejrzysta, mimo iż może się wydawać odwrotna).
Najpierw programy na systemy 32-bitowe:
(przeskocz pierwszy 32-bitowy program w składni NASM)

```
; wersja NASM na system 32-bitowy

section .text                   ; początek sekcji kodu.
global _start           ; linker ld chce mieć ten symbol globalny

_start:                         ; punkt startu programu

        mov     eax, 4          ; numer funkcji systemowej:
                                ; sys_write - zapisz do pliku
        mov     ebx, 1          ; numer pliku, do którego piszemy.
```

```
                                ; 1 = standardowe wyjście = ekran
        mov     ecx, tekst      ; ECX = adres (offset) tekstu
        mov     edx, dlugosc    ; EDX = długość tekstu
        int     80h             ; wywołujemy funkcję systemową
        mov     eax, 1          ; numer funkcji systemowej
                                ; (sys_exit - wyjdź z programu)
        int     80h             ; wywołujemy funkcję systemową

section .data                   ; początek sekcji danych.

tekst   db      "Czesc", 0ah    ; nasz napis, który wyświetlimy
dlugosc equ     $ - tekst       ; długość napisu
```

Teraz wersja dla FASMa:
(przeskocz pierwszy 32-bitowy program w składni FASM)

```
; wersja FASM na system 32-bitowy

format ELF executable           ; typ pliku
entry _start                    ; punkt startu programu

segment readable executable     ; początek sekcji kodu

_start:                         ; punkt startu programu

        mov     eax, 4          ; numer funkcji systemowej:
                                ; sys_write - zapisz do pliku
        mov     ebx, 1          ; numer pliku, do którego piszemy.
                                ; 1 = standardowe wyjście = ekran
        mov     ecx, tekst      ; ECX = adres (offset) tekstu
        mov     edx, [dlugosc]  ; EDX = długość tekstu
        int     80h             ; wywołujemy funkcję systemową
        mov     eax, 1          ; numer funkcji systemowej
                                ; (sys_exit - wyjdź z programu)
        int     80h             ; wywołujemy funkcję systemową

segment readable writeable      ; początek sekcji danych.
tekst   db      "Czesc", 0ah    ; nasz napis, który wyświetlimy
dlugosc dd      $ - tekst       ; długość napisu
```

Teraz program 64-bitowy (x86-64) dla NASMa:
(przeskocz pierwszy 64-bitowy program w składni NASM)

```
; wersja NASM na system 64-bitowy (x86-64)

section .text                   ; początek sekcji kodu.
global _start           ; linker ld chce mieć ten symbol globalny

_start:                         ; punkt startu programu

        mov     rax, 1          ; numer funkcji systemowej:
                                ; sys_write - zapisz do pliku
        mov     rdi, 1          ; numer pliku, do którego piszemy.
                                ; 1 = standardowe wyjście = ekran
        mov     rsi, tekst      ; RSI = adres (offset) tekstu
        mov     rdx, dlugosc    ; RDX = długość tekstu
        syscall                 ; wywołujemy funkcję systemową
        mov     rax, 60         ; numer funkcji systemowej
                                ; (sys_exit - wyjdź z programu)
        syscall                 ; wywołujemy funkcję systemową
```

```
section .data                   ; początek sekcji danych.

tekst   db      "Czesc", 0ah    ; nasz napis, który wyświetlimy
dlugosc equ     $ - tekst       ; długość napisu
```

I w końcu program 64-bitowy dla FASMa:
(przeskocz pierwszy 64-bitowy program w składni FASM)

```
; wersja FASM na system 64-bitowy  (x86-64)

format ELF64 executable         ; typ pliku
entry _start                    ; punkt startu programu

segment readable executable     ; początek sekcji kodu

_start:                         ; punkt startu programu
        mov     rax, 1          ; numer funkcji systemowej:
                                ; sys_write - zapisz do pliku
        mov     rdi, 1          ; numer pliku, do którego piszemy.
                                ; 1 = standardowe wyjście = ekran
        mov     rsi, tekst      ; RSI = adres (offset) tekstu
        mov     rdx, [dlugosc]  ; RDX = długość tekstu
        syscall                 ; wywołujemy funkcję systemową
        mov     rax, 60         ; numer funkcji systemowej
                                ; (sys_exit - wyjdź z programu)
        syscall                 ; wywołujemy funkcję systemową

segment readable writeable      ; początek sekcji danych.
tekst   db      "Czesc", 0ah    ; nasz napis, który wyświetlimy
dlugosc dq      $ - tekst       ; długość napisu w trybie 64-bitowym
```

Bez paniki! Teraz omówimy dokładnie, co każda linia robi.

- linie lub napisy zaczynające się średnikiem.

  Traktowane są jako komentarze i są całkowicie ignorowane przy kompilacji. Rozmiar skompilowanego programu wynikowego nie zależy od ilości komentarzy. Dlatego najlepiej wstawiać tyle komentarzy, aby inni (również my) mogli później zrozumieć nasz kod.

- (FASM) `format ELF executable` / `format ELF64 executable`

  Określa format (typ) pliku wyjściowego: wykonywalny plik ELF (format używany w Linuksie). FASM nie potrzebuje programów łączących, aby utworzyć program. Format ELF64 jest używany oczywiście pod systemem 64-bitowym.

- (FASM) `entry _start`

  Określa, gdzie program się zaczyna. Po uruchomieniu programu procesor zaczyna wykonywać komendy zaczynające się pod podaną tutaj etykietą (_start) znajdującą się w sekcji kodu.

- (FASM) `segment readable executable`

  Określa nowy segment programu - segment kodu, któremu ustawiamy odpowiednie atrybuty: do odczytu i do wykonywania. Innym atrybutem jest `writeable` (do zapisu), który powinien być

używany tylko do sekcji danych. Mimo, iż FASM zaakceptuje atrybut `writeable` dla sekcji kodu, nie powinniśmy go tam umieszczać. Zapisanie czegokolwiek do sekcji kodu może skończyć się błędem naruszenia ochrony pamięci (segmentation fault). Można jednak w tym segmencie umieszczać dane. Ale należy to robić tak, aby nie stały się one częścią programu, zwykle wpisuje się je za ostatnią komendą kończącą program. Procesor przecież nie wie, co jest pod danym adresem i z miłą chęcią potraktuje to coś jako instrukcję, co może prowadzić do przykrych konsekwencji. Swoje dane umieszczajcie tak, aby w żaden sposób strumień wykonywanych instrukcji nie wszedł na nie. Dane będziemy więc zazwyczaj umieszczać w oddzielnej sekcji.

- (NASM) `section .text`

  Wskazuje początek segmentu, gdzie znajduje się kod programu. Można jednak w tym segmencie umieszczać dane. Ale należy to robić tak, aby nie stały się one częścią programu, zwykle wpisuje się je za ostatnią komendą kończącą program. Procesor przecież nie wie, co jest pod danym adresem i z miłą chęcią potraktuje to coś jako instrukcję, co może prowadzić do przykrych konsekwencji. Swoje dane umieszczajcie tak, aby w żaden sposób strumień wykonywanych instrukcji nie wszedł na nie. Zapisanie czegokolwiek do sekcji kodu może skończyć się błędem naruszenia ochrony pamięci (segmentation fault), dlatego dane będziemy zazwyczaj umieszczać w oddzielnej sekcji.

- (NASM) `global _start`

  Sprawiamy, że nazwa _start będzie widziana poza tym programem (konkretnie to przez linker ld, który skompilowaną wersję programu przerobi na wersję wykonywalną).

- `_start:` (z dwukropkiem)

  Etykieta określająca początek programu.

- `mov eax, 4` / `mov rax, 1`

  Do rejestru EAX (32-bitowy) lub RAX (64-bitowy) wstaw (MOV = move, przesuń) wartość 4 (1 na systemach x86-64). Jest to numer funkcji systemu Linux, którą chcemy uruchomić. Jeśli chcemy skorzystać z funkcji systemowych, to zawsze EAX zawiera numer takiej funkcji.
  Numery funkcji różnią się na różnych architekturach procesorów. Poczytajcie mój spis funkcji systemowych.

  Komenda MOV ma 3 ważne ograniczenia:
  1. nie można skopiować jedną komendą MOV komórki pamięci do innej komórki pamięci, takie coś:

     ```
     mov     [a], [b]
     ```

     (gdzie a i b - dwie zmienne w pamięci) jest zabronione.
     O tym, co oznaczają nawiasy kwadratowe, czyli o adresowaniu zmiennych w pamięci - następnym razem.
  2. nie można skopiować jedną komendą MOV jednego rejestru segmentowego (cs,ds,es,ss,fs,gs) do innego rejestru segmentowego, czyli taka operacja

     ```
     mov     es, ds
     ```

jest zabroniona. W ogóle najlepiej *unikać jakichkolwiek operacji na rejestrach segmentowych*.

3. Nie można do rejestru segmentowego bezpośrednio wpisać jakieś wartości. Czyli nie można

```
mov     ds, 0
```

Ale można:

```
mov     bx, 0
mov     ds, bx
```

- `mov ebx, 1` / `mov rdi, 1`

  Do rejestru EBX (32-bitowy) lub RDI (64-bitowy) wstaw 1. Dlaczego akurat 1? Zaraz się przekonamy.

- `mov ecx,tekst` / `mov rsi, tekst`

  Do rejestru ECX (32-bitowy) lub RSI (64-bitowy) wstaw offset (adres) etykiety tekst. Można obliczać adresy nie tylko danych, ale etykiet znajdujących się w kodzie programu.

- `mov edx, dlugosc / mov edx, [dlugosc]` / `mov rdx, dlugosc / mov rdx, [dlugosc]`

  Do rejestru EDX (32-bitowy) lub RDX (64-bitowy) wstaw długość naszego tekstu. W pierwszym przypadku jest to stała, w drugim wartość pobieramy ze zmiennej.

- `int 80h` / `syscall`

  Int = interrupt = przerwanie. Nie jest to jednak znane np. z kart dźwiękowych przerwanie typu IRQ. Wywołując przerwanie 80h (128 dziesiętnie) lub instrukcję `syscall` (w trybie 64-bitowym) uruchamiamy jedną z funkcji Linuksa. Którą? O tym zazwyczaj mówi rejestr akumulatora.
  W tym przypadku EAX = 4 (lub RAX = 1 w trybie 64-bitowym) i jest to funkcja zapisu do pliku - sys_write. Funkcja ta przyjmuje 3 argumenty:
  - W rejestrze EBX (lub RDI, w trybie 64-bitowym) podajemy numer (deskryptor) pliku, do którego chcemy pisać. U nas EBX (lub RDI) = 1 i jest to standardowe wyjście (zazwyczaj ekran).
  - W rejestrze ECX (lub RSI, w trybie 64-bitowym) podajemy adres danych, które chcemy zapisać do pliku.
  - W rejestrze EDX (lub RDX, w trybie 64-bitowym) podajemy, ile bajtów chcemy zapisać.

  Funkcje systemowe 32-bitowego Linuksa przyjmują co najwyżej 6 argumentów, *kolejno w rejestrach: EBX, ECX, EDX, ESI, EDI, EBP*. W EAX oczywiście jest numer funkcji, tak jak tutaj 4.

  Funkcje systemowe 64-bitowego Linuksa przyjmują także co najwyżej 6 argumentów, *kolejno w rejestrach: RDI, RSI, RDX, R10, R8, R9*. W RAX oczywiście jest numer funkcji, tak jak tutaj 1. Rejestry RCX i R11 są zamazywane.

- `mov eax, 1` / `mov rax, 60`

Do EAX lub RAX wpisujemy numer kolejnej funkcji systemowej - sys_exit, która spowoduje zamknięcie naszego programu.

- `int 80h / syscall`

  Przerwanie systemowe - uruchomienie funkcji wyjścia z programu. Numer błędu (taki DOS-owski errorlevel) zwykle umieszczamy w EBX/RDI, czego tutaj jednak nie zrobiliśmy.

- `(NASM) section .data / (FASM) segment readable writeable`

  Określa początek sekcji danych. Dane muszą być w osobnej części programu, bo inaczej nie można do nich zapisywać (a na jądrze 2.6 nawet odczytać).

- `tekst db "Czesc",0ah`

  Definicja napisu i znaku przejścia do nowej linii. O tym, jak deklarować zmienne powiem następnym razem.

- `dlugosc equ $ - tekst / dlugosc dd $ - tekst / dlugosc dq $ - tekst`

  Definiujemy stałą, która przyjmuje wartość: adres bieżący - adres początku napisu, czyli długość napisu. W pierwszym przypadku jest to stała kompilacji, w drugim i trzecim - zmienna, która będzie umieszczona w programie.

Programik nazywamy hello.asm i kompilujemy poleceniem (FASM):

```
fasm hello.asm hello
```

lub, dla NASMa:

```
nasm -f elf hello.asm
ld -o hello hello.o
```

lub, dla NASMa na systemie 64-bitowym:

```
nasm -f elf64 hello.asm
ld -o hello hello.o
```

Wyjaśnienie opcji:

- `-f elf` powoduje, że plik będzie skompilowany na 32-bitowy plik obiektowy typu ELF (Executable-Linkable Format, typowy dla większości Linuksów). Aby kompilować programy pod systemem 64-bitowym, należy użyć formatu `elf64`
- `-f elf64` powoduje, że plik będzie skompilowany na 64-bitowy plik obiektowy typu ELF
- `-o nazwa` spowoduje nazwanie programu wynikowego.

Uruchamiamy ./hello i cieszymy się swoim dziełem.
W dalszej części kursu będę przedstawiał programy *tylko 32-bitowe* i często tylko dla *jednego kompilatora*. Ta część będzie służyć Wam pomocą, jeśli chcielibyście pisać programy pod systemy 64-bitowe lub pod inny

kompilator.
Miłego eksperymentowania.

> Na świecie jest 10 rodzajów ludzi:
> ci, którzy rozumieją liczby binarne i ci, którzy nie.

Informacja dla użytkowników *BSD (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia

1. Poeksperymentujcie sobie, wstawiając różne znaki do napisu. Na przykład, znaki o kodach ASCII 10 (Line Feed), 13 (Carriage Return), 7 (Bell). Na przykład:

```
info    db      "Czesc.", 00, 01, 02, 07, 10, 13, 10, 13
```

Część 2 - Pamięć, czyli gdzie upychać coś, co się nie mieści w procesorze.

Poznaliśmy już rejestry procesora. Jak widać, jest ich ograniczona ilość i nie mają one zbyt dużego rozmiaru. Rejestry ogólnego przeznaczenia są co najwyżej 32-bitowe (4-bajtowe). Dlatego często programista musi niektóre zmienne umieszczać w pamięci. Przykładem tego był napis, który wyświetlaliśmy w poprzedniej części artykułu. Był on zadeklarowany dyrektywą DB, co oznacza declare byte. Ta dyrektywa niekoniecznie musi deklarować dokładnie 1 bajt. Tak jak widzieliśmy, można nią deklarować napisy lub kilka bajtów pod rząd. Teraz omówimy rodzinę dyrektyw służących właśnie do rezerwowania pamięci.

Ogólnie, zmienne można deklarować jako bajty (dyrektywą DB, coś jak char w języku C), słowa (word = 16 bitów = 2 bajty, coś jak short w C) dyrektywą DW, podwójne słowa DD (double word = dword = 32bity = 4 bajty, jak long w C), potrójne słowa pword = 6 bajtów - PW, poczwórne słowa DQ (quad word = qword = 8 bajtów, typ long long), tbyte = 10 bajtów - DT (typ long double w C).
Sekcja kodu jest tylko do odczytu, więc zmienne, które chcemy móc rzeczywiście zmienić, *musimy umieścić w sekcji danych*. Od tej pory umawiamy się więc, że każda zmienna znajduje się w obszarze section .data (dla NASMa) lub segment readable writeable (dla FASMa).

Przykłady:
(przeskocz przykłady)

```
section .data
; FASM: segment readable writeable

dwa             db 2
szesc_dwojek    db 2, 2, 2, 2, 2, 2
litera_g        db "g"
_ax             dw 4c00h        ; 2-bajtowa liczba całkowita
alfa            dd 12348765h    ; 4-bajtowa liczba całkowita

;liczba_a       dq 1125         ; 8-bajtowa liczba całkowita.
                                ; FASM przyjmie, NASM
                                ; starszy niż wersja 2.00 nie.

                                ; dla NASMa zamienimy to na
                                ; postać równoważną: 2 razy
                                ; po 4 bajty:
liczba_a        dd 1125, 0

liczba_e        dq 2.71         ; liczba zmiennoprzecinkowa
                                ; podwójnej precyzji (8 bajtów),

                        ; 10-bajtowa liczba całkowita:
;duza_liczba    dt 6af4aD8b4a43ac4d33h
                        ; FASM ani NASM tego nie przyjmie.
                        ; Zrobimy to tak:
duza_liczba     dd 43ac4d33h, 0f4aD8b4ah; czemu z zerem z przodu?
                                        ; Czytaj dalej
```

```
                db 6ah

pi              dt 3.141592     ; FASM i NASM

;nie_init       db ?    ; nie zainicjalizowany bajt.
                        ; Wartość nieznana. NASM ani FASM tak
                        ; tego nie przyjmie. Należy użyć:

nie_init        resb 1          ; NASM
;nie_init       rb 1            ; FASM

napis1          db "NaPis1."
xxx             db 1
                db 2
                db 3
                db 4
```

Zwróćcie uwagę na sposób rozbijania dużych liczb na poszczególne bajty: najpierw deklarowane są młodsze bajty, a potem starsze (np. dd 11223344h = db 44h, 33h, 22h, 11h). To działa, gdyż procesory Intela i AMD (i wszystkie inne klasy x86) są procesorami typu little-endian, co znaczy, że najmłodsze bajty danego ciągu bajtów są umieszczane przez procesor w najniższych adresach pamięci. Dlatego my też tak deklarujemy nasze zmienne.

Ale z kolei takie coś:

```
beta    db aah
```

nie podziała. Dlaczego? *KAŻDA liczba musi zaczynać się od cyfry.* Jak to obejść? Tak:

```
beta    db 0aah
```

czyli poprzedzić zerem.
Nie podziała również to:

```
0gamma  db      9
```

Dlaczego? Etykiety (dotyczy to tak danych, jak i kodu programu) *nie mogą zaczynać się od cyfr*.
A co, jeśli chcemy zadeklarować zmienną, powiedzmy, składającą się z 234 bajtów równych zero? Trzeba je wszystkie napisać?
Ależ skąd! Należy użyć operatora TIMES. Odpowiedź na pytanie brzmi:

```
section .data

zmienna         TIMES    234    db      0
nazwa                   ilość   typ    co zduplikować
```

lub, w składni FASMa:

```
segment readable writeable

; 234 razy zarezerwuj bajt wartości 0:
zmienna2:       times    234    db      0
```

Rezerwacja obszaru bez określania jego wartości wyglądałaby mniej więcej tak:

```
section .data
```

```
; FASM: segment readable writeable

zmienna         resb    234             ; NASM
zmienna2        rb      234             ; FASM
```

A co, jeśli chcemy mieć dwuwymiarową tablicę podwójnych słów o wymiarach 25 na 34?
Robimy dla NASMa na przykład tak:

```
section .data

Tablica         times   25*34   dd      0
```

a dla FASMa:

```
segment readable writeable

; 25*34 razy zarezerwuj dword wartości 0:
Tablica2:       times   25*34   dd      0
```

Do obsługi takich tablic przydadzą się bardziej skomplikowane sposoby adresowania zmiennych. O tym za moment.

Zmiennych trzeba też umieć używać.
Do uzyskania adresu danej zmiennej używa się nazwy tej zmiennej, tak jak widzieliśmy wcześniej. Zawartość zmiennej otrzymuje się poprzez umieszczenie jej nazwy w nawiasach kwadratowych. Oto przykład:

```
section .data
; FASM: segment readable writeable

rejestr_eax     dd      1
rejestr_bx      dw      0
rejestr_cl      db      0
...
        mov     [rejestr_bx], bx
        mov     cl, [rejestr_cl]
        mov     eax, [rejestr_eax]
        int     80h
```

Zauważcie zgodność rozmiarów zmiennych i rejestrów.
Możemy jednak mieć problem w skompilowaniu czegoś takiego:

```
mov     [jakas_zmienna], 2
```

Dlaczego? Kompilator wie, że gdzieś zadeklarowaliśmy jakas_zmienna, ale nie wie, czy było to

```
jakas_zmienna   db      0
```

czy

```
jakas_zmienna   dw      22
```

czy może

```
jakas_zmienna   dd      "g"
```

Chodzi o to, aby pokazać, jaki rozmiar ma obiekt docelowy. Nie będzie problemów, gdy napiszemy:

```
mov     word [jakas_zmienna], 2
```

I to obojętnie, czy zmienna była bajtem (wtedy następny bajt będzie równy 0), czy słowem (wtedy będzie ono miało wartość 2) czy może podwójnym słowem lub czymś większym (wtedy 2 pierwsze bajty zostaną zmienione, a pozostałe nie). Dzieje się tak dlatego, że zmienne zajmują kolejne bajty w pamięci, najmłodszy bajt w komórce o najmniejszym adresie. Na przykład:

```
xxx     dd      8
```

jest równoważne:

```
xxx     db      8,0,0,0
```

oraz:

```
xxx     db      8
        db      0
        db      0
        db      0
```

Te przykłady nie są jedynymi sposobami adresowania zmiennych (poprzez nazwę). Na procesorach 32-bitowych (od 386) odnoszenie się do pamięci może odbywać się wg schematu:

[zmienna + rej_baz + rej_ind * skala +- liczba]
gdzie:

- zmienna oznacza nazwę zmiennej i jest to liczba obliczana przez kompilator lub linker
- rej_baz (rejestr bazowy) = jeden z rejestrów EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
- rej_ind (rejestr indeksowy) = jeden z rejestrów EAX, EBX, ECX, EDX, ESI, EDI, EBP (bez ESP)
- mnożnik (scale) = 1, 2, 4 lub 8 (gdy nie jest podany, przyjmuje się 1)

Przykłady:

```
mov     al, [ nazwa_zmiennej+2 ]
mov     [ edi-23 ], cl
mov     dl, [ ebx + esi*2 + nazwa_zmiennej+18 ]
```

Na procesorach 64-bitowych odnoszenie się do pamięci może odbywać się wg schematu:

[zmienna + rej_baz + rej_ind * skala +- liczba]
gdzie:

- zmienna oznacza nazwę zmiennej i jest to liczba obliczana przez kompilator lub linker
- rej_baz (rejestr bazowy) = jeden z rejestrów RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8, ..., R15, a nawet RIP (ale wtedy nie można użyć żadnego rejestru indeksowego)

- rej_ind (rejestr indeksowy) = jeden z rejestrów RAX, RBX, RCX, RDX, RSI, RDI, RBP, R8, ..., R15 (bez RSP i RIP)
- mnożnik (scale) = 1, 2, 4 lub 8 (gdy nie jest podany, przyjmuje się 1)

Dwie zasady:

- między nawiasami kwadratowymi nie można mieszać rejestrów różnych rozmiarów
- w trybie 64-bitowym nie można do adresowania używać rejestrów cząstkowych: R*D, R*W, R*B.

Przykłady:

```
mov     al, [ nazwa_zmiennej+2 ]
mov     [ rdi-23 ], cl
mov     dl, [ rbx + rsi*2 + nazwa_zmiennej+18 ]
mov     rax, [rax+rbx*8-34]
mov     rax, [ebx]
mov     r8d, [ecx-11223344]
mov     cx, [r8]
```

A teraz inny przykład: spróbujemy wczytać 5 elementów o numerach 1, 3, 78, 25, i 200 (pamiętajmy, że liczymy od zera) z tablicy zmienna (tej o 234 bajtach, zadeklarowanej wcześniej) do kilku rejestrów 8-bitowych. Operacja nie jest trudna i wygląda po prostu tak:

```
mov     al, [ zmienna + 1 ]
mov     ah, [ zmienna + 3 ]
mov     cl, [ zmienna + 78 ]
mov     ch, [ zmienna + 25 ]
mov     dl, [ zmienna + 200 ]
```

Oczywiście, kompilator nie sprawdzi za Was, czy takie elementy tablicy rzeczywiście istnieją - o to musicie zadbać sami.
W powyższym przykładzie rzuca się w oczy, że ciągle używamy słowa zmienna, bo wiemy, gdzie jest nasza tablica. Jeśli tego nie wiemy (dynamiczne przydzielanie pamięci), lub z innych przyczyn nie chcemy ciągle pisać zmienna, możemy posłużyć się bardziej złożonymi sposobami adresowania. Po chwili zastanowienia bez problemu stwierdzicie, że powyższy kod można bez problemu zastąpić czymś takim i też będzie działać:

```
mov     ebx, zmienna
mov     al, [ ebx + 1 ]
mov     ah, [ ebx + 3 ]
mov     cl, [ ebx + 78 ]
mov     ch, [ ebx + 25 ]
mov     dl, [ ebx + 200 ]
```

Teraz trudniejszy przykład: spróbujmy dobrać się do kilku elementów 2-wymiarowej tablicy dwordów zadeklarowanej wcześniej (tej o rozmiarze 25 na 34). Mamy 25 wierszy po 34 elementy każdy. Aby do EAX wpisać pierwszy element pierwszego wiersza, piszemy oczywiście tylko:

```
mov     eax, [Tablica]
```

Ale jak odczytać 23 element 17 wiersza? Otóż, sprawa nie jest taka trudna, jakby się mogło wydawać. Ogólny schemat wygląda tak (zakładam, że ostatni wskaźnik zmienia się najszybciej, potem przedostatni itd. - pamiętamy, że rozmiar elementu wynosi 4):

```
Tablica[17][23] = [ Tablica + (17*długość wiersza + 23)*4 ]
```

No to piszemy:

```
      mov     ebx, Tablica
      mov     esi, 17
jakas_petla:
      imul    esi, 34         ; ESI=ESI*34=17 * długość wiersza
      add     esi, 23         ; ESI=ESI+23=17 * długość wiersza + 23
      mov     eax, [ ebx + esi*4 ]    ; mnożymy numer elementu przez
                                      ; rozmiar elementu
      ...
```

Można było to zrobić po prostu tak:

```
      mov     eax, [ Tablica + (17*34 + 23)*4 ]
```

ale poprzednie rozwiązanie (na rejestrach) jest wprost idealne do pętli, w której robimy coś z coraz to innym elementem tablicy.

Podobnie ((numer_wiersza*długość_wiersza1 + numer_wiersza*długość_wiersza2 + ...)*rozmiar_elementu) adresuje się tablice wielowymiarowe. Schemat jest następujący:

```
      Tablica[d1][d2][d3][d4]    - 4 wymiary o długościach wierszy
                                        d1, d2, d3 i d4

      Tablica[i][j][k][m] = [ Tablica + (i*d2*d3*d4+j*d3*d4+k*d4+m)*
                              *rozmiar_elementu ]
```

Teraz powiedzmy, że mamy taką tablicę:

```
      dword tab1[24][78][13][93]
```

Aby dobrać się do elementu tab1[5][38][9][55], piszemy:

```
      mov     eax, [ tab1 + (5*78*13*93 + 38*13*93 + 9*93 + 55)*4 ]
```

Pytanie: do jakich segmentów się to odnosi? Przecież mamy kilka rejestrów segmentowych, które mogą wskazywać na zupełnie co innego.
Odpowiedź:
Na rejestrach 32-bitowych mamy:

1. jeśli pierwszym w kolejności rejestrem jest EBP lub ESP, używany jest SS
2. w pozostałych przypadkach używany jest DS

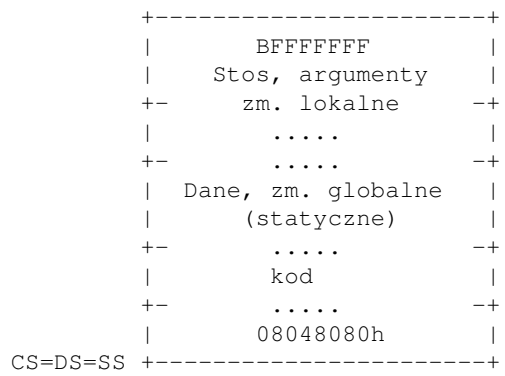
W systemach 64-bitowych segmenty odchodzą w zapomnienie.
Domyślne ustawianie można zawsze obejść używając przedrostków, np.

```
      mov     ax, [ss:si]
      mov     [gs:eax+ebx*2-8], cx
```

Organizacja pamięci w Linuksie.

W systemie Linux każdy program dostaje swoją własną przestrzeń, nie jest możliwe zapisywanie zmiennych lub kodu innych programów (z wyjątkami, np. debugery). Teoretycznie rozmiar owej przestrzeni wynosi tyle, ile można zaadresować w ogóle całym procesorem, czyli 2^32 = 4 GB na procesorach 32-bitowych. Obszar ten jest jednak od góry trochę ograniczony przez sam system, ale nie będziemy się tym zajmować.

Struktura programu po uruchomieniu jest dość prosta: cały kod, dane i stos (o tym za chwilę) znajdują się w jednym segmencie, rozciągającym się na całą wspomnianą przestrzeń. Na moim systemie wykonywanie zaczyna się pod adresem 08048080h w tej przestrzeni.
(przeskocz ilustrację pamięci programu w Linuksie)



Najniżej w pamięci znajduje się kod, za nim dane, a na końcu - stos.
Jak w takim razie realizowana jest ochrona kodu przed zapisem?
W samym procesorze istnieje mechanizm stronicowania, który umożliwia przyznanie odpowiednich praw do danych stron pamięci (zwykle strona ma 4kB). Tak więc, nasz duży segment jest podzielony na strony z kodem, danymi i stosem.

Stos.

Przyszła pora na omówienie, czym jest stos.
Otóż, stos jest po prostu kolejnym segmentem pamięci. Są na nim umieszczane dane tymczasowe, np. *adres powrotny z funkcji, jej parametry wywołania, jej zmienne lokalne*. Służy też do zachowywania zawartości rejestrów.
Obsługa stosu jest jednak zupełnie inna.
Po pierwsze, stos jest budowany od góry na dół! Rysunek będzie bardzo pomocny:
(przeskocz rysunek stosu)

```
Adres
                SS
        +-------------------+
100h    |                   |
        +-------------------+   <----- ESP = 100h
9eh     |                   |
        +-------------------+
9ch     |                   |
        +-------------------+
9ah     |                   |
        +-------------------+
98h     |                   |
        +-------------------+
96h     |                   |
...             ....
```

Na tym rysunku ESP=100h, czyli ESP wskazuje na komórkę o adresie 100h w segmencie SS.
Dane na stosie umieszcza się instrukcją PUSH a zdejmuje instrukcją POP. Push jest równoważne parze instrukcji:

```
sub     esp, ..     ; odejmowana liczba zależy od
                    ; rozmiaru obiektu w bajtach
mov     [ss:esp], ..
```

a pop:

```
mov     .., [ss:esp]
add     esp, ..
```

Tak więc, po wykonaniu instrukcji PUSH AX i PUSH DX powyższy stos będzie wyglądał tak:
(przeskocz ilustrację działania PUSH)

```
Stos po wykonaniu  PUSH AX i PUSH DX, czyli
        sub     esp, 2
        mov     [ss:esp], ax
        sub     esp, 2
        mov     [ss:esp], dx

                SS
        +-------------------+
100h    |                   |
        +-------------------+
9eh     |       AX          |
        +-------------------+
9ch     |       DX          |
        +-------------------+   <----- ESP = 9ch
...             ....
```

SP=9ch, pod [SP] znajduje się wartość DX, a pod [SP+2] - wartość AX. A po wykonaniu instrukcji POP EBX (tak, można zdjąć dane do innego rejestru, niż ten, z którego pochodziły):
(przeskocz ilustrację działania POP)

```
Stos po wykonaniu POP EBX, czyli
        mov     ebx, [ss:esp]
        add     esp, 4

                SS
```

```
        +-------------------+
100h    |                   |
        +-------------------+   <----- ESP = 100h
9eh     |       AX          |
        +-------------------+
9ch     |       DX          |
        +-------------------+
...             ....
```

Teraz ponownie SP=100h. Zauważcie, że dane są tylko kopiowane ze stosu, a nie z niego usuwane. Ale w żadnym przypadku nie można na nich już polegać. Dlaczego? Zobaczycie zaraz.
Najpierw bardzo ważna uwaga, która jest wnioskiem z powyższych rysunków.
Dane (które chcemy z powrotem odzyskać w niezmienionej postaci) położone na stosie instrukcją PUSH należy zdejmować kolejnymi instrukcjami POP *W ODWROTNEJ KOLEJNOŚCI* niż były kładzione.
Zrobienie czegoś takiego:

```
push    eax
push    edx
pop     eax
pop     edx
```

nie przywróci rejestrom ich dawnych wartości!

Przerwania i procedury a stos

Używaliśmy już instrukcji przerwania, czyli INT. Przy okazji omawiania stosu nadeszła pora, aby powiedzieć, co ta instrukcja w ogóle robi. Otóż, INT jest (w przybliżeniu) równoważne temu pseudo-kodowi:

```
pushfd                  ; włóż na stos rejestr stanu procesora
                        ; czyli flagi
push    cs              ; segment, w którym aktualnie pracujemy
push    eip_next        ; adres instrukcji po INT
jmp     procedura_obslugi_przerwania
```

Każda procedura obsługi przerwania (Interrupt Service Routine, ISR) kończy się instrukcją IRET (interrupt return), która odwraca powyższy kod, czyli z ISR procesor wraca do dalszej obsługi naszego programu.
Jednak oprócz instrukcji INT przerwania mogą być wywołana w inny sposób - przez sprzęt. Tutaj właśnie pojawiają się IRQ. Do urządzeń wywołujących przerwania IRQ należą między innymi karta dźwiękowa, modem, zegar, kontroler dysku twardego, itd...

Bardzo istotną rolę gra zegar, utrzymujący aktualny czas w systemie. Jak napisałem w jednym z artykułów, tyka on z częstotliwością ok. 18,2 Hz. Czyli ok. 18 razy na sekundę wykonywane są 3 PUSHe a po nich 3 POPy. Nie zapominajmy o push i pop wykonywanych w samej ISR tylko po to, aby zachować modyfikowane rejestry. Każdy PUSH zmieni to, co jest poniżej ESP.

Dlatego właśnie żadne dane poniżej ESP nie mogą być uznawane za wiarygodne.

Gdzie zaś znajdują się adresy procedur obsługi przerwań?
W pamięci, w Tabeli Deskryptorów Przerwań (Interrupt Descriptor Table, IDT), do której dostęp ma wyłącznie system operacyjny. Na pojedynczy deskryptor przerwania składa się oczywiście adres procedury

obsługi przerwania, jej deskryptor, prawa dostępu do niej i kilka innych informacji, które z punktu widzenia programisty nie są (na razie) istotne.

Mniej skomplikowana jest instrukcja CALL, która służy do wywoływania zwykłych procedur. W zależności od rodzaju procedury (near - zwykle w tym samym pliku/programie, far - np. w innym pliku/segmencie), instrukcja CALL wykonuje takie coś:

```
push    cs i kilka innych rzeczy ; tylko jeśli FAR
push    eip_next        ; adres instrukcji po CALL
```

Procedura może zawierać dowolne (nawet niesymetryczne ilości instrukcji PUSH i POP), ale pod koniec ESP musi być taki sam, jak był na początku, czyli wskazywać na prawidłowy adres powrotu, który ze stosu jest zdejmowany instrukcją RET (lub RETF). Dlatego nieprawidłowe jest takie coś:

```
zla_procedura:
        push    eax
        push    ebx
        add     eax, ebx
        ret
```

gdyż w chwili wykonania instrukcji RET na wierzchu stosu jest EBX, a nie adres powrotny! Błąd stosu jest przyczyną wielu trudnych do znalezienia usterek w programie.
Jak to poprawić bez zmiany sensu? Na przykład tak:

```
moja_procedura:
        push    eax
        push    ebx
        add     eax, ebx
        add     esp, 8
        ret
```

Teraz już wszystko powinno być dobrze. ESP wskazuje na dobry adres powrotny. Dopuszczalne jest też takie coś:

```
proc1:
        push    eax
        cmp     eax, 0          ; czy EAX jest zerem?
        je      koniec1         ; jeśli tak, to koniec1

        pop     ebx
        ret
koniec1:
        pop     ecx
        ret
```

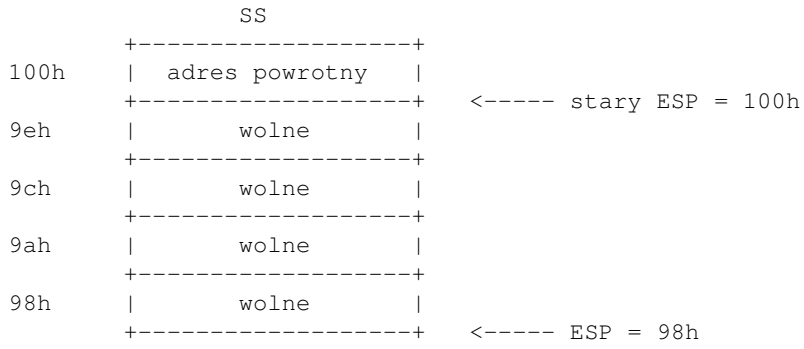
ESP ciągle jest dobrze ustawiony przy wyjściu z procedury mimo, iż jest 1 PUSH a 2 POPy.
Po prostu ZAWSZE należy robić tak, aby ESP wskazywał na poprawny adres powrotny, niezależnie od sposobu.

Alokacja zmiennych lokalnych procedury

Nie musi się to Wam od razu przydać, ale przy okazji stosu omówię, gdzie znajdują się zmienne lokalne

funkcji (np. takich w języku C) oraz jak rezerwować na nie miejsce.

Gdy program wykonuje instrukcję CALL, na stosie umieszczany jest adres powrotny (o czym już wspomniałem). Jako że nad nim mogą być jakieś dane ważne dla programu (na przykład zachowane rejestry, inne adresy powrotne), nie wolno tam nic zapisywać. Ale pod adresem powrotnym jest dużo miejsca i to tam właśnie programy umieszczają swoje zmienne lokalne.
Samo rezerwowanie miejsca jest dość proste: liczymy, ile łącznie bajtów nam potrzeba na własne zmienne i tyle właśnie odejmujemy od rejestru ESP, robiąc tym samym miejsce na stosie, które nie będzie zamazane przez instrukcje INT i CALL (gdyż one zamazują tylko to, co jest pod ESP).
Na przykład, jeśli nasze zmienne zajmują 8 bajtów, to odejmujemy te 8 od ESP i nasz nowy stos wygląda tak:



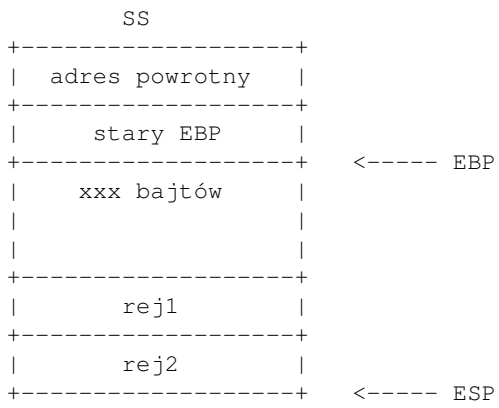
ESP wynosi 98h, nad nim jest 8 bajtów wolnego miejsca, po czym adres powrotny i inne stare dane.
Miejsce już mamy, korzystanie z niego jest proste - wystarczy odwoływać się do [ESP], [ESP+2], [ESP+4], [ESP+6]. Ale stanowi to pewien problem, bo po każdym wykonaniu instrukcji PUSH, te cyferki się zmieniają (bo przecież adresy się nie zmieniają, ale ESP się zmienia). Dlatego właśnie do adresowania zmiennych lokalnych często używa się innego rejestru niż ESP. Jako że domyślnym segmentem dla EBP jest segment stosu, wybór padł właśnie na ten rejestr (oczywiście, można używać dowolnego innego, tylko trzeba dostawiać SS: z przodu, co kosztuje za każdym razem 1 bajt).

Aby móc najłatwiej dostać się do swoich zmiennych lokalnych, większość funkcji na początku zrównuje EBP z ESP, potem wykonuje rezerwację miejsca na zmienne lokalne, a dopiero potem - zachowywanie rejestrów itp. (czyli swoje PUSH-e). Wygląda to tak:

```
push    ebp             ; zachowanie starego EBP
mov     ebp, esp        ; EBP = ESP

sub     esp, xxx        ; rezerwacja miejsca na zmienne lokalne
push    rej1            ; tu ESP się zmienia, ale EBP już nie
push    rej2
...

...
pop     rej2            ; tu ESP znów się zmienia, a EBP – nie
pop     rej1

mov     esp, ebp        ; zwalnianie zmiennych lokalnych
                        ;   można też (ADD ESP,xxx)
pop     ebp

ret
```

Przy instrukcji `MOV ESP, EBP` napisałem, że zwalnia ona zmienne lokalne. Zmienne te oczywiście dalej są na stosie, ale teraz są już poniżej ESP, a niedawno napisałem: *żadne dane poniżej ESP nie mogą być*

uznawane za wiarygodne.
Po pięciu pierwszych instrukcjach nasz stos wygląda tak:



Rejestr EBP wskazuje na starą wartość EBP, zaś ESP - na ostatni element włożony na stos.
I widać teraz, że zamiast odwoływać się do zmiennych lokalnych poprzez [ESP+liczba] przy ciągle zmieniającym się ESP, o wiele wygodniej odwoływać się do nich przez [EBP-liczba] (zauważcie: minus), bo EBP pozostaje niezmienione.

Często np. w disasemblowanych programach widać instrukcje typu `AND ESP, NOT 16` (lub `AND ESP, ~16` w składni NASM). Jedynym celem takich instrukcji jest wyrównanie ESP do pewnej pożądanej granicy, np. 16 bajtów (wtedy AND z wartością NOT 16, czyli FFFFFFF0h), żeby dostęp do zmiennych lokalnych trwał krócej. Gdy adres zmiennej np. czterobajtowej jest nieparzysty, to potrzeba dwóch dostępów do pamięci, żeby ją całą pobrać (bo można pobrać 32 bity z na raz w procesorze 32-bitowym i tylko z adresu podzielnego przez 4).

Ogół danych: adres powrotny, parametry funkcji, zmienne lokalne i zachowane rejestry nazywany jest czasem ramką stosu (ang. stack frame).
Rejestr EBP jest czasem nazywany wskaźnikiem ramki, gdyż umożliwia od dostęp do wszystkich istotnych danych poprzez stałe przesunięcia (offsety, czyli te liczby dodawane i odejmowane od EBP): zmienne lokalne są pod `[EBP-liczba]`, parametry funkcji przekazane z zewnątrz - pod `[EBP+liczba]`, zaś pod `[EBP]` jest stara wartość EBP. Jeśli wszystkie funkcje w programie zaczynają się tym samym prologiem: `PUSH EBP / MOV EBP, ESP`, to po wykonaniu instrukcji `MOV EBP, [EBP]` w EBP znajdzie się wskaźnik ramki ... procedury wywołującej. Jeśli znamy jej strukturę, można w ten sposób dostać się do jej zmiennych lokalnych.

Zainteresowanych szczegółami adresowania lub instrukcjami odsyłam do Intela: Intela i AMD Następnym razem o podstawowych instrukcjach języka asembler.

- Ilu programistów potrzeba, aby wymienić żarówkę?
- Ani jednego. To wygląda na problem sprzętowy.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)

Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia

1. Zadeklaruj tablicę 12 zmiennych mających po 10 bajtów:
    1. zainicjalizowaną na zera (pamiętaj o ograniczeniach kompilatora)
    2. niezainicjalizowaną

2. Zadeklaruj tablicę 12 słów (16-bitowych) o wartości BB (szesnastkowo), po czym do każdego z tych słów wpisz wartość FF szesnastkowo (bez żadnych pętli). Można (a nawet trzeba) użyć więcej niż 1 instrukcji. Pamiętaj o odległościach między poszczególnymi elementami tablicy. Naucz się różnych sposobów adresowania: liczba (nazwa zmiennej + numer), baza (rejestr bazowy + liczba), baza + indeks (rejestr bazowy + rejestr indeksowy).

3. Zadeklaruj dwuwymiarową tablicę bajtów o wartości 0 o wymiarach 13 wierszy na 5 kolumn, po czym do elementu numer 3 (przedostatni) w wierszu o numerze 12 (ostatni) wpisz wartość FF. Spróbuj użyć różnych sposobów adresowania.

Jak pisać programy w języku asembler pod Linuksem?

Część 3 - Podstawowe instrukcje, czyli poznajemy dialekt procesora.

Poznaliśmy już rejestry, omówiliśmy pamięć. Pora zacząć na nich operować.
Zanim jednak zaczniemy, proszę Was o to, abyście tej listy też *NIE uczyli się na pamięć*. Instrukcji jest dużo, a próba zrozumienia ich wszystkich na raz może spowodować niezły chaos. Co najwyżej przejrzyjcie tą listę kilka razy, aby wiedzieć mniej-więcej, co każda instrukcja robi.

Instrukcje procesora można podzielić na kilka grup:

- instrukcje przemieszczania danych
- instrukcje arytmetyki binarnej
- instrukcje arytmetyki dziesiętnej
- instrukcje logiczne
- operacje na bitach i bajtach
- instrukcje przekazujące kontrolę innej części programu (sterujące wykonywaniem programu)
- instrukcje operujące na łańcuchach znaków
- instrukcje kontroli flag
- instrukcje rejestrów segmentowych
- inne

Zacznijmy je po kolei omawiać (nie omówię wszystkich).

1. instrukcje przemieszczania danych.
   Tutaj zaliczymy już wielokrotnie używane `MOV` oraz kilka innych: `XCHG` , `PUSH` i `POP`.

2. arytmetyka binarna.
   `add do_czego,co` - dodaj
   `sub od_czego,co` - odejmij
   `inc coś / dec coś` - zwiększ/zmniejsz coś o 1
   `cmp co, z_czym` - porównaj. Wykonuje działanie odejmowania co minus z_czym, ale nie zachowuje wyniku, tylko ustawia flagi.
   Wynikiem może być ustawienie lub wyzerowanie jednej lub więcej flag - zaznaczenie wystąpienia jednego z warunków. Główne warunki to:
   - A - above - ponad (dla liczb traktowanych jako liczby bez znaku): co > z_czym
     (przeskocz przykład użycia warunku A)

     ```
     cmp al,bl
     ja al_wieksze_od_bl     ; ja -  jump if above
     ```
   - B - below - poniżej (bez znaku): co < z_czym
   - G - greater - więcej niż (ze znakiem): co > z_czym
   - L - lower - mniej niż (ze znakiem): co < z_czym
   - O - overflow - przepełnienie (ze znakiem, np. przebicie 32767 w górę) ostatniej operacji. Niekoniecznie używane przy cmp.
   - C - carry - przepełnienie (bez znaku, czyli przebicie np. 65535 w górę)

(przeskocz przykład użycia warunku C)

```
add al,bl
jc blad_przepelnienia   ; jc -  jump if carry
```

- E lub Z - equal (równy) lub zero. Te dwa warunki są równoważne.
(przeskocz przykłady użycia warunków równości)

```
cmp ax,cx
je ax_rowne_cx
...
sub bx,dx
jz bx_rowne_dx
```

- NE/NZ - przeciwieństwo poprzedniego: not equal/not zero.
- NA - not above, czyli nie ponad - mniejsze lub równe (ale dla liczb bez znaku)
- NB - not below, czyli nie poniżej - większe lub równe (dla liczb bez znaku)
- NG - not greater, czyli nie więcej - mniejsze lub równe (ale dla liczb ze znakiem)
- NL - not lower, czyli nie mniej - większe lub równe (dla liczb ze znakiem)
- NC - no carry
- AE/BE - above or equal (ponad lub równe), below or equal (poniżej lub równe)
- NO - no overflow

3. arytmetyka dziesiętna
   - NEG - zmienia znak.
   - MUL, IMUL - mnożenie, mnożenie ze znakiem (które uwzględnia liczby ujemne)
(przeskocz przykłady instrukcji mnożenia)

```
mul cl          ; AX = AL*CL
mul bx          ; DX:AX = AX*BX
mul esi         ; EDX:EAX = EAX*ESI
mul rdi         ; RDX:RAX = RAX*RDI

imul eax        ; EDX:EAX = EAX*EAX
imul ebx,ecx,2  ; EBX = ECX*2
imul ebx,ecx    ; EBX = EBX*ECX
imul si,5       ; SI = SI*5
```

Zapis rej1 : rej2 oznacza, że starsza część wyniku znajdzie się w pierwszym rejestrze podanej pary (DX, EDX, RDX), a młodsza - w drugim (AX, EAX, RAX), gdyż wynik mnożenia dwóch liczb o długości n bitów każda wymaga 2n bitów.

   - DIV, IDIV - dzielenie, dzielenie ze znakiem.
(przeskocz przykłady instrukcji dzielenia)

```
div cl  ; AL = (AX div CL), AH = (AX mod CL)
div bx  ; AX = (DX:AX div BX),
        ; DX = (DX:AX mod BX)
div edi ; EAX = (EDX:EAX div EDI),
        ; EDX = (EDX:EAX mod EDI)
div rsi ; RAX = (RDX:RAX div RSI),
        ; RDX = (RDX:RAX mod RSI)
```

Zapis rej1 : rej2 oznacza, że starsza część dzielnej jest oczekiwana w pierwszym rejestrze podanej pary (DX, EDX, RDX), a młodsza - w drugim (AX, EAX, RAX). Jeśli liczba mieści się w rejestrze dla młodszej części, rejestr starszy należy wyzerować. Słowo "div" w

powyższych zapisach oznacza iloraz, a mod - resztę z dzielenia (modulo).

4. Instrukcje bitowe (logiczne). AND, OR, XOR, NOT, TEST. Instrukcja TEST działa tak samo jak AND z tym, że nie zachowuje nigdzie wyniku, tylko ustawia flagi. Po krótce wytłumaczę te instrukcje:

   (przeskocz działanie instrukcji logicznych)

   ```
   0 AND 0 = 0     0 OR 0 = 0      0 XOR 0 = 0
   0 AND 1 = 0     0 OR 1 = 1      0 XOR 1 = 1
   1 AND 0 = 0     1 OR 0 = 1      1 XOR 0 = 1
   1 AND 1 = 1     1 OR 1 = 1      1 XOR 1 = 0
   NOT 0 = 1
   NOT 1 = 0
   ```

   Przykłady zastosowania:
   (przeskocz przykłady instrukcji logicznych)

   ```
   and ax,1        ; wyzeruje wszystkie bity z
                   ; wyjątkiem bitu numer 0.
   or ebx,1111b    ; ustawia (włącza) 4 dolne bity.
                   ; Reszta bez zmian.
   xor cx,cx       ; CX = 0
   not dh          ; DH ma 0 tam, gdzie miał 1
                   ; i na odwrót
   ```

5. Instrukcje przesunięcia bitów.
   1. SAL, SHL - shift left.
      bit7 = bit6, bit6 = bit5, ... , bit1 = bit0, bit0 = 0.

   2. SHR - shift logical right
      bit0 = bit1, bit1 = bit2, ... , bit6 = bit7, bit7 = 0

   3. SAR - shift arithmetic right
      bit0 = bit1, bit1 = bit2, ... , bit6 = bit7, *bit7* = *bit7* (bit znaku zachowany!)
      Najstarszy bit w rejestrze nazywa się czasem właśnie bitem znaku.

   4. ROL - rotate left
      bit7 = bit6, ... , bit1 = bit0, bit0 = stary bit7

   5. RCL - rotate through carry left
      carry flag CF = bit7, bit7 = bit6, ... , bit1 = bit0, bit0 = stara CF

   6. ROR - rotate right
      bit0 = bit1, ... , bit6 = bit7, bit7 = stary bit0

   7. RCR - rotate through carry right
      CF = bit0, bit0 = bit1, ... , bit6 = bit7, bit7 = stara CF

   Przy użyciu SHL można przeprowadzać szybkie mnożenie, a dzięki SHR - szybkie dzielenie. Np. SHL AX,1 jest równoważne przemnożeniu AX przez 2, SHL AX,5 - przez 2^5 = 32. SHR BX,4 dzieli bx przez 16.

6. Instrukcje sterujące wykonywaniem programu.

- Skoki warunkowe (patrz: warunki powyżej): JA=JNBE, JAE=JNB, JNA=JBE, JNAE=JB , JG=JNLE (jump if greater - dla liczb ze znakiem) = jump if not lower or equal, JNG=JLE, JGE=JNL, JNGE=JL, JO, JNO, JC, JNC, JS (jump if sign czyli bit7 wyniku jest równy 1), JNS, JP=JPE (jump if parity equal = liczba bitów równych jeden jest parzysta), JNP=JPO.
- Skoki bezwarunkowe: JMP, JMP SHORT, JMP FAR
- Uruchomienia procedur: CALL [NEAR/FAR]
- Powrót z procedury: RET/RETF.
- Przerwania: INT, INTO (wywołuje przerwanie INT4 w razie przepełnienia), BOUND (int 5)
- Instrukcje pętli: LOOP. Składnia: LOOP gdzieś. Jeśli CX jest różny od 0, to skacz gdzieś.

7. Operacje na łańcuchach znaków.
    1. LODS[B/W/D/Q] - Load Byte/Word/Dword/Qword
       MOV AL/AX/EAX/RAX , DS:[SI/ESI/RSI]
       ADD SI,1/2/4/8 ; ADD, gdy flaga kierunku DF = 0, SUB gdy DF = 1

    2. STOS[B/W/D/Q] - Store Byte/Word/Dword/Qword
       MOV ES:[DI/EDI/RDI], AL/AX/EAX/RAX
       ADD DI,1/2/4/8 ; ADD/SUB jak wyżej

    3. MOVS[B/W/D/Q] - Move Byte/Word/Dword/Qword
       MOV ES:[DI/EDI/RDI], DS:[SI/ESI/RSI] ; to nie jest instrukcja!
       ADD DI,1/2/4/8 ; ADD/SUB jak wyżej
       ADD SI,1/2/4/8

    4. CMPS[B/W/D/Q] - Compare Byte/Word/Dword/Qword
       CMP DS:[SI/ESI/RSI], ES:[DI/EDI/RDI] ; to nie jest instrukcja!
       ADD SI,1/2/4/8 ; ADD/SUB jak wyżej
       ADD DI,1/2/4/8

    5. SCAS[B/W/D/Q] - Scan Byte/Word/Dword/Qword
       skanuje łańcuch bajtów/słów/podwójnych słów/poczwórnych słów pod ES:[DI/EDI/RDI] w poszukiwaniu, czy jest tam wartość wskazana przez AL/AX/EAX/RAX.

   Do każdej z powyższych instrukcji można z przodu dodać przedrostek REP (repeat), co spowoduje, że będzie ona wykonywana, aż CX stanie się zerem, lub REPE/REPZ lub REPNE/REPNZ co spowoduje, że będzie ona wykonywana, dopóty CX nie jest zerem i jednocześnie ZF (flaga zera) =1 lub =0, odpowiednio.

8. Instrukcje wejścia/wyjścia do portów.
   Są one bardziej szczegółowo opisane w części poświęconej portom, ale podam tu skrót:
    - IN
      IN AL/AX/EAX, port/DX.
      Pobierz z portu 1/2/4 bajty i włóż do AL/AX/EAX (od najmłodszego). Jeśli numer portu jest mniejszy lub równy 255, można go podać bezpośrednio. Jeśli większy - trzeba użyć DX.

    - OUT
      OUT port/DX, AL/AX/EAX.
      Uwagi jak przy instrukcji IN.

9. Instrukcje flag

- STC/CLC - set carry / clear carry. Do flagi CF wstaw 1 lub 0, odpowiednio.
- STD/CLD. Ustaw DF = 1, DF = 0, odpowiednio.
- STI/CLI. Interrupt Flag IF := 1, IF := 0, odpowiednio. Gdy IF=0, przerwania sprzętowe są blokowane.
- Przenoszenie flag
  PUSHF / PUSHFD / PUSHFQ - umieść flagi na stosie (16, 32 i 64 bity flag, odpowiednio)
  POPF / POPFD / POPFQ - zdejmij flagi ze stosu (16/32/64 bity flag)
  SAHF / LAHF - zapisz AH w pierwszych 8 bitach flag / zapisz pierwsze 8 bitów flag w AH.

10. Instrukcja LEA - Load Effective Address.
    Wykonanie:

```
lea     rej, [pamięć]
```

    jest równoważne:
    (przeskocz pseudo-kod LEA)

```
mov     rej, pamięć             ; NASM/FASM
```

    Po co więc osobna instrukcja? Otóż, LEA przydaje sie w wielu sytuacjach do obliczania złożonych adresów. Kilka przykładów:

    1. Jak w 1 instrukcji sprawić, że EAX = EBP-12 ?
       Odpowiedź: `lea eax, [ebp-12]`
    2. Niech EBX wskazuje na tablicę o 20 elementach o rozmiarze 8 każdy. Jak do ECX zapisać adres 11-tego elementu, a do EDX elementu o numerze EDI?
       Odpowiedź: `lea ecx, [ebx + 11*8 ]` oraz `lea edx,[ebx+edi*8]`
    3. Jak w 1 instrukcji sprawić, że ESI = EAX*9?
       Odpowiedź: `lea esi, [eax + eax*8 ]`

Pominąłem mniej ważne instrukcje operujące na rejestrach segmentowych i klika innych instrukcji. Te, które tu podałem, wystarczają absolutnie na napisanie większości programów, które można zrobić.

Wszystkie informacje przedstawione w tej części pochodzą z tego samego źródła: Intel i AMD

> Byle głupiec potrafi napisać kod, który zrozumie komputer. Dobry programista pisze taki kod, który zrozumie człowiek.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia

1. Zapisz instrukcje: do rejestru AX dodaj 5, o rejestru SI odejmij 178.

2. Nie używając cyfry jeden napisz jedną instrukcję, która zmniejszy rejestr DX o jeden.

3. Przemnóż wartość rejestru EDI przez 2 na przynajmniej dwa różne sposoby po jednej instrukcji. Postaraj się nie używać instrukcji (I)MUL.

4. W jednej instrukcji podziel wartość rejestru BP przez 8.

5. Nie używając instrukcji MOV spraw, by DX miał wartość 0 (na przynajmniej 3 sposoby, każdy po jednej instrukcji).

6. Nie używając instrukcji przesuwania bitów SH* ani mnożenia *MUL przemnóż EBX przez 8. Możesz użyć więcej niż 1 instrukcji.

7. W dwóch instrukcjach spraw, by EDI równał się 7*ECX. Postaraj się nie używać instrukcji (I)MUL.

Część 4 - Pierwsze programy, czyli przełamywanie pierwszych lodów.

Znamy już rejestry, trochę instrukcji i zasad. No ale teoria jest niczym bez praktyki. Dlatego w tej części przedstawię kilka względnie prostych programów, które powinny rozbudzić wyobraźnię tworzenia.

Ten program spyta się użytkownika o imię i przywita się z nim:
(przeskocz program pytający o imię)

```
; Program witający się z użytkownikiem po imieniu
;
; Autor: Bogdan D.
; kontakt: bogdandr (at) op (dot) pl
;
; kompilacja:
; nasm -f elf czesc.asm
; ld -s -o czesc czesc.o
;
; kompilacja FASM:
; fasm czesc.asm czesc

; dla FASMa:
;format ELF executable
;entry _start

;segment readable executable            ; początek sekcji kodu

; dla NASMa:
section .text           ; początek sekcji kodu
global _start   ; _start będzie symbolem globalnym,
                ; od którego zacznie się wykonywanie programu

_start:
        mov     eax, 4          ; zapis do pliku
        mov     ebx, 1          ; na ekran
        mov     ecx, jak_masz   ; napis do wyświetlenia: pytanie
        mov     edx, jak_masz_dl        ; długość napisu
        int     80h             ; wyświetlamy

        mov     eax, 3          ; czytanie z pliku
        mov     ebx, 0          ; z klawiatury
        mov     ecx, imie       ; dokąd czytać?
        mov     edx, imie_dl    ; ile bajtów czytać?
        int     80h             ; wczytujemy


        mov     eax, 4          ; zapis do pliku
        mov     ebx, 1          ; na ekran
        mov     ecx, czesc      ; napis do wyświetlenia: "cześć"
        mov     edx, czesc_dl   ; długość napisu
        int     80h             ; wyświetlamy

        mov     eax, 4          ; zapis do pliku
```

```
                mov     ebx, 1          ; na ekran
                mov     ecx, imie       ; napis do wyświetlenia: imię
                mov     edx, imie_dl    ; długość napisu
                int     80h             ; wyświetlamy

                mov     eax, 1
                xor     ebx, ebx
                int     80h

        ; dla FASMa:
        ;segment readable writeable            ; początek sekcji danych

        section .data           ; początek sekcji danych

        jak_masz        db      "Jak masz na imie? "
        ; FASM: znak równości zamiast EQU
        jak_masz_dl     equ     $ - jak_masz

        ; rezerwuj 20 bajtów o wartości początkowej zero, na imię
        imie:           times 20 db 0
        ; FASM: znak równości zamiast EQU
        imie_dl         equ     $ - imie

        czesc           db      "Czesc "
        ; FASM: znak równości zamiast EQU
        czesc_dl        equ     $ - czesc
```

Następny program wypisuje na ekranie rejestr flag w postaci dwójkowej.
(przeskocz program wypisujący flagi)

```
; Program wypisujący flagi w postaci dwójkowej
;
; Autor: Bogdan D.
; kontakt: bogdandr (at) op (dot) pl
;
; kompilacja:
; nasm -f elf flagi.asm
; ld -s -o flagi flagi.o
;
; kompilacja FASM:
; fasm flagi.asm flagi


;format ELF executable          ; dla FASMa
; entry _start                  ; dla FASMa

; segment readable executable   ; dla FASMa

section .text           ; tu zaczyna się segment kodu,
                        ; nie jest to potrzebne

global _start           ; nazwa punktu rozpoczęcia programu.
                        ; FASM: usunąć tę linijkę

;CPU 386                ; będziemy tu używać rejestrów 32-bitowych.
                        ; Nie jest to potrzebne, gdyż
                        ; NASM domyślnie włącza wszystkie
                        ; możliwe instrukcje.
```

```
_start:                         ; etykieta początku programu

        pushfd                  ; 32 bity flag idą na stos

        pop     esi             ; flagi ze stosu do ESI

        mov     eax, "0"
        mov     ebx, nasze_flagi; EBX = adres bufora dla wartości flag
        xor     edi, edi        ; EDI = 0

        mov     cx, 32          ; tyle bitów i tyle razy trzeba
                                ; przejść przez pętlę

petla:                          ; etykieta oznaczająca początek pętli.

        and     al, "0" ; upewniamy się, że AL zawiera tylko
                                ; 30h="0", co zaraz się
                                ; może zmienić. Dokładniej,
                                ; czyścimy bity 0-3,
                                ; z których bit 0 może się zaraz zmienić

        shl     esi, 1          ; Przesuwamy bity w ESI o 1 w lewo.
                                ; 31 bit ESI idzie do CF

        adc     al, 0           ; ADC - add with carry. Do AL dodaj
                                ; 0 + wartość CF.
                                ; jeśli CF (czyli 31 bit ESI) = 1,
                                ; to AL := AL+1,
                                ; inaczej AL bez zmian

        mov     [ebx+edi], al   ; zapisz AL w buforze
        add     edi, 1

        loop    petla           ; przejdź na początek pętli,
                                ; jeśli nie skończyliśmy

        mov     eax, 4          ; funkcja zapisywania do pliku/na ekran
        mov     ebx, 1          ; 1 = ekran
        mov     ecx, nasze_flagi
        mov     edx, 32         ; długość tekstu
        int     80h             ; wypisz na ekran

        mov     byte [nasze_flagi],0ah
        mov     eax, 4          ; funkcja zapisywania do pliku/na ekran
        mov     ebx, 1          ; 1 = ekran
        mov     ecx, nasze_flagi
        mov     edx, 1          ; długość tekstu
        int     80h             ; wypisz na ekran przejście do nowej linii

        mov     eax, 1
        int     80h             ; wyjście z programu

; FASM: segment readable writeable
section .data                   ; dane już nie mogą być w sekcji kodu, gdyż
                                ; w Linuksie sekcja kodu programu jest
                                ; chroniona przed zapisem

nasze_flagi:    times   32      db "0"          ; "0" = 30h
```

Kompilujemy go następująco (wszystkie programy będziemy tak kompilować, chyba że powiem inaczej):

```
        nasm -f elf flagi.asm
```

```
        ld -s -o flagi flagi.o
```

lub:

```
        fasm flagi.asm flagi
```

Nie ma w tym programie wielkiej filozofii. Nie powinno być trudno go zrozumieć.

Teraz krótki programik, którego jedynym celem jest wyświetlenie na ekranie cyfr od 0 do 9, każda w osobnej linii:
(przeskocz program wypisujący cyfry)

```
; Program wypisuje na ekranie cyfry od 0 do 9
;
; kompilacja NASM:
; nasm -f elf cyfry.asm
; ld -s -o cyfry cyfry.o


section .text

global _start

; definiujemy stałe (NASM):

%define         lf      10              ; Line Feed
%define stdout  1       ; standardowe urządzenie wyjścia (zwykle ekran)
%define         sys_write 4             ; funkcja pisania do pliku

;
; kompilacja FASM:
; fasm cyfry.asm cyfry
;
; format ELF executable         ; dla FASMa
; entry _start
; segment readable executable

; definiujemy stałe (FASM):
; lf = 10
; stdout = 1
; sys_write = 4



_start:

        mov     eax, 0                  ; pierwsza wypisywana cyfra

wyswietlaj:
        call    _pisz_ld                ; uruchom procedurę wyświetlania
                                        ; liczby będącej w EAX
        call    _nwln                   ; uruchom procedurę, która
                                        ; przechodzi do nowej linii
        add     eax, 1                  ; zwiększamy cyfrę
        cmp     eax, 10                 ; sprawdzamy, czy ciągle EAX < 10
        jb      wyswietlaj              ; jeśli EAX < 10, to
                                        ; wyświetlamy cyfrę
```

```
        mov     eax, 1                  ; funkcja wyjścia z programu
        xor     ebx, ebx                ; kod wyjścia = 0
        int     80h                     ; wychodzimy


; następujące procedury (wyświetlanie liczby i przechodzenie
; do nowego wiersza) nie są aż tak istotne, aby omawiać je
; szczegółowo, gdyż w przyszłości będziemy używać tych samych
; procedur, ale z biblioteki, a te wstawiłem tutaj dla
; uproszczenia kompilacji programu.

; Ogólny schemat działania tej procedury wygląda tak:
; weźmy liczbę EAX=12345. Robimy tak:
; 1. dzielimy EAX przez 10. reszta = EDX = DL = 5.
; Zapisz do bufora. EAX = 1234 (iloraz)
; 2. dzielimy EAX przez 10. reszta = DL = 4.
; Zapisz do bufora. EAX=123 (iloraz)
; 3. dzielimy EAX przez 10. reszta = DL = 3.
; Zapisz do bufora. EAX=12
; 4. dziel EAX przez 10. DL = 2. zapisz. iloraz = EAX = 1
; 5. dziel EAX przez 10. DL = 1. zapisz. iloraz = EAX = 0.
; Przerywamy pętlę.
; Teraz w buforze są znaki: "54321". Wystarczy wypisać wspak
; i oryginalna liczba pojawia się na ekranie.

_pisz_ld:

; pisz32e
; we: EAX=liczba bez znaku do wypisania

        pushfd
        push    ecx
        push    edx
        push    eax
        push    esi

        xor     esi, esi
        mov     ecx, 10

._pisz_ld_petla:
        xor     edx, edx
        div     ecx

        or      dl, "0"
        mov     [_pisz_bufor+esi], dl   ; do bufora idą reszty z
                                        ; dzielenia przez 10,
        inc     esi                     ; czyli cyfry wspak

        test    eax, eax
        jnz     ._pisz_ld_petla

._pisz_ld_wypis:
        mov     al, [_pisz_bufor+esi-1] ; wypisujemy reszty wspak
        call    _pisz_z

        dec     esi
        jnz     ._pisz_ld_wypis

        pop     esi
        pop     eax
```

```
        pop     edx
        pop     ecx
        popfd

        ret


_pisz_z:

; pisz_z
; we: AL=znak do wypisania

        push    eax
        push    ebx
        push    ecx
        push    edx

        mov     [_pisz_bufor+39], al

        mov     eax, sys_write          ; funkcja zapisu do pliku
        mov     ebx, stdout             ; kierujemy na
                                        ; standardowe wyjście
        lea     ecx, [_pisz_bufor+39]
        mov     edx, 1
        int     80h

        pop     edx
        pop     ecx
        pop     ebx
        pop     eax

        ret


_nwln:

;pisze znak końca linii (Linux)

        push    eax

        mov     al, lf
        call    _pisz_z

        pop     eax
        ret


section .data
; FASM: segment readable writeable

_pisz_bufor:    times   40      db 0    ; miejsce na 40 cyferek
```

Następny twór nie jest wolnostojącym programem, ale pewną procedurą. Pobiera ona informacje z rejestru AL i wypisuje, co trzeba. Oto ona:
(przeskocz procedurę _pisz_ch)

```
; FASM: segment readable executable
section .text
```

```
_pisz_ch:

;we: AL=cyfra heksadecymalna do wypisania 0...15
; CF=1 jeśli błąd

        push eax                ; zachowaj modyfikowane rejestry: AX, Flagi
        pushfd

        cmp al,9                ; Sprawdzamy dane wejściowe :
                                ; AL jest w 0-9 czy w 10-15?
        ja _ch_hex              ; AL < 9. Skok do "_ch_hex"
        or al,"0"               ; 0 < AL < 9. Or ustawia 2 bity,
                                ; czyniąc z AL liczbę z
                                ; przedziału 30h - 39h, czyli od "0" do "9".
                                ; Można było napisać
                                ; "add al,30h", ale zdecydowałem się
                                ; na "or", bo jest
                                ; szybsze a efekt ten sam.

        jmp short _ch_pz        ; AL już poprawione. Skacz do miejsca,
                                ; gdzie wypisujemy znak.

_ch_hex:                        ; AL > 9. Może będzie to cyfra hex,
                                ; może nie.
        cmp al,15               ; AL > 15?
        ja _blad_ch             ; jeśli tak, to mamy błąd
        add al,"A"-10           ; Duży skok myślowy. Ale wystarczy to
                                ; rozbić na 2 kroki i
                                ; wszystko staje się jasne. Najpierw
                                ; odejmujemy 10 od AL.
                                ; Zamiast liczby od 10 do 15 mamy już
                                ; liczbę od 0 do 5. Teraz tą liczbę
                                ; dodajemy do "A", czyli kodu ASCII litery
                                ; A, otrzymując znak od "A" do "F"

_ch_pz:                         ; miejsce wypisywania znaków.
        mov     [znak], al

        mov     eax, 4          ; funkcja wypisywania
        mov     ebx, 1          ; ekran
        mov     ecx, znak
        mov     edx, 1
        int     80h


        popfd                   ; zdejmij ze stosu flagi
        clc                     ; CF := 0 dla zaznaczenia braku błędu
                                ; (patrz opis procedury)
        jmp short _ch_ok        ; skok do wyjścia

_blad_ch:                       ; sekcja obsługi błędu (AL > 15)
        popfd                   ; zdejmij ze stosu flagi
        stc                     ; CF := 1 na znak błędu

_ch_ok:                         ; miejsce wyjścia z procedury
        pop eax                 ; zdejmij modyfikowane rejestry

        ret                     ; return, powrót

; FASM: segment readable writeable
section .data
```

```
znak    db      0
```

To chyba nie było zbyt trudne, co?
Szczegóły dotyczące pisania procedur (i bibliotek) znajdują się w moim innym artykule.

Teraz pokażę pewien program, który wybrałem ze względu na dużą ilość różnych instrukcji i sztuczek.
Niestety, nie jest on krótki, ale wspólnie spróbujemy przez niego przejść.
(przeskocz program zliczający liczby pierwsze)

```
; Program liczy liczby pierwsze w przedziałach
;   2-10, 2-100, 2-1000,... 2-100.000
;
; Autor: Bogdan D.
; kontakt: bogdandr (at) op (dot) pl
;
; kompilacja:
;
; nasm -f elf ile_pier.asm
; ld -s -o ile_pier ile_pier.o
;
; kompilacja FASM:
; fasm ile_pier.asm ile_pier


; format ELF executable         ; tylko dla FASMa
; entry _start

; FASM: segment readable executable
section .text

global _start                   ; FASM: usunąć

_start:                         ; początek...

        xor ebx,ebx             ; EBX = liczba, którą sprawdzamy,
                                ; czy jest pierwsza. Zaczniemy od 3.
                                ; Poniżej jest 3 razy "inc"
                                ; (zwiększ o 1). Najpierw
                                ; EBX = 0, bo "xor rej,rej" zeruje dany
                                ; rejestr

        xor edi,edi             ; EDI = bieżący licznik liczb pierwszych

        xor ecx,ecx             ; ECX = stary licznik liczb
                                ; (z poprzedniego przedziału)
                                ; Chwilowo, oczywiście 0.

        inc ebx                 ; EBX = 1
        mov esi,10              ; ESI = bieżący koniec przedziału.
        inc edi                 ; EDI=1. uwzględniamy dwójkę, która
                                ; jest liczbą pierwsza
        inc ebx                 ; EBX=2, pierwsza liczba będzie = 3

petla:                          ; pętla przedziału

        cmp ebx,esi             ; czy koniec przedziału?
                                ; (ebx=liczba, esi=koniec przedziału)
        jae pisz                ; EBX >= ESI - idź do sekcji
                                ; wypisywania wyników
```

```
        mov ebp,2               ; EBP - liczby, przez które
                                ; będziemy dzielić.
                                ; pierwszy dzielnik = 2

        inc ebx                 ; zwiększamy liczbę. EBX=3. Będzie
                                ; to pierwsza sprawdzana

spr:                            ; pętla sprawdzania pojedynczej liczby

        mov eax,ebx             ; EAX = sprawdzana liczba
        xor edx,edx             ; EDX = 0
        div ebp                 ; EAX = EAX/EBP (EDX było=0),
                                ; EDX=reszta z dzielenia

        or edx,edx              ; instrukcja OR tak jak wiele innych,
                                ; ustawi flagę zera ZF na 1, gdy jej
                                ; wynik był zerem. W tym przypadku
                                ; pytamy: czy EDX jest zerem?

        jz petla                ; jeżeli dzieli się bez reszty
                                ; (reszta=EDX=0),
                                ; to nie jest liczbą pierwszą
                                ; i należy zwiększyć liczbę
                                ; sprawdzaną (inc ebx)

        inc ebp                 ; zwiększamy dzielnik

        cmp ebp,ebx             ; dzielniki aż do liczby
        jb spr                  ; liczba > dzielnik - sprawdzaj
                                ; dalej tą liczbę. Wiem, że
                                ; można było sprawdzać tylko do
                                ; SQRT(liczba) lub LICZBA/2, ale
                                ; wydłużyłoby to program i brakowało
                                ; mi już rejestrów...

juz:                            ; przerobiliśmy wszystkie dzielniki,
                                ; zawsze wychodziła reszta
                                ; więc liczba badana jest pierwsza

        inc edi                 ; zwiększamy licznik liczb znalezionych
        jmp petla               ; sprawdzaj kolejną liczbę aż
                                ; do końca przedziału

                                ; sekcja wypisywania informacji

pisz:

        push ebx                ; zachowujemy modyfikowane
                                ; a ważne rejestry
        push ecx

        mov     eax, 4
        mov     ebx, 1
        mov     ecx, przedzial
        mov     edx, dlugosc_przedzial
        int     80h             ; wypisujemy informację o przedziale

        mov     eax,esi         ; EAX=ESI=koniec przedziału
        call    _pisz_ld        ; wypisz ten koniec (EAX)

        mov     eax, 4
        mov     ebx, 1
```

```
        mov     ecx, dwuk
        mov     edx, 1
        int     80h             ; wypisujemy dwukropek

        pop ecx

        add ecx,edi             ; dodajemy poprzednią ilość
                                ; znalezionych liczb pierwszych
        mov eax,ecx             ; EAX = ilość liczb pierwszych
                                ; od 2 do końca bieżącego przedziału

        call _pisz_ld           ; wypisujemy tą ilość.

        pop ebx

        cmp esi,100000          ; 10^5
        jb dalej                ; ESI > 100.000? Tak - koniec,
                                ; bo dalej liczy zbyt długo
koniec:
        mov     eax, 4
        mov     ebx, 1
        mov     ecx, przedzial
        mov     edx, 1
        int     80h             ; wypisujemy znak nowej linii


        xor     ebx, ebx        ; kod wyjścia = 0
        mov     eax, 1
        int     80h             ; wyjście z programu

dalej:

        mov eax,esi             ; EAX=ESI
        shl eax,3               ; EAX = EAX*8
        shl esi,1               ; ESI=ESI*2
        add esi,eax             ; ESI=ESI*2+EAX*8=ESI*2+ESI*8=ESI*10.
                                ; Znacznie szybciej niż MUL

        xor edi,edi             ; bieżący licznik liczb

        jmp petla               ; robimy od początku...

_pisz_ld:

;we: EAX=liczba bez znaku do wypisania

        push ebx
        push ecx                ; zachowujemy modyfikowane rejestry
        push edx
        push eax
        push esi

        xor esi,esi             ; SI=0. Będzie wskaźnikiem w
                                ; powyższy bufor.

        mov ecx,10              ; będziemy dzielić przez 10,
                                ; aby uzyskiwać kolejne cyfry
                                ; Reszty z dzielenia pójdą do
                                ; bufora, potem będą wypisane
                                ; wspak, bo pierwsza reszta
                                ; jest przecież cyfrą jedności
_pisz_ld_petla:
```

```
        xor edx,edx             ; EDX=0

        div ecx                 ; EAX = EAX/ECX, EDX = reszta,
                                ; która mieści się w DL, bo to
                                ; jest tylko 1 cyfra dziesiętna.

        or  dl, "0"

        mov [_pisz_bufor+esi],dl        ; Cyfra do bufora.

        inc esi                 ; Zwiększ numer komórki w buforze,
                                ; do której będziemy teraz pisać

        or eax,eax              ; EAX = 0 ?

        jnz _pisz_ld_petla      ; Jeśli nie (JNZ), to skok do
                                ; początku pętli

_pisz_ld_wypis:

        mov     eax, 4
        mov     ebx, 1
        lea     ecx, [_pisz_bufor+esi-1]
        mov     edx, 1
        int     80h

        dec esi                 ; zmniejsz wskaźnik do bufora.

        jnz _pisz_ld_wypis      ; Jeśli ten wskaźnik (ESI) nie
                                ; jest zerem, wypisuj dalej

        pop esi                 ; odzyskaj zachowane rejestry
        pop eax
        pop edx
        pop ecx
        pop ebx

        ret                     ; powrót z procedury

; FASM: segment readable writeable
 section .data

 _pisz_bufor: times 20 db 0       ; miejsce na cyfry dla procedury

 przedzial       db      10,"Przedzial 2-"

; FASM:  dlugosc_przedzial       =     $ - przedzial
 dlugosc_przedzial       equ     $ - przedzial

 dwuk            db      ":"
```

Kilka uwag o tym programie:

- Czemu nie zrobiłem"`mov ebx,2` a potem `inc ebx`, które musiało być w pętli?
  Bo xor ebx,ebx jest krótsze i szybsze.

- Dobra. Więc czemu nie:

```
xor ebx,ebx
inc ebx
```

```
inc ebx
```

Te instrukcje operują na tym samym rejestrze i każda musi poczekać, aż poprzednia się zakończy. Współczesne procesory potrafią wykonywać niezależne czynności równolegle, dlatego wcisnąłem w środek jeszcze kilka niezależnych instrukcji.

- Ten program sprawdza za dużo dzielników. Nie można było sprawić, by sprawdzał tylko do np. połowy sprawdzanej liczby?
  Można było. Używając zmiennych w pamięci. Niechętnie to robię, bo w porównaniu z prędkością operacji procesora, pamięć jest wprost NIEWIARYGODNIE wolna. Zależało mi na szybkości.

- Czy nie prościej zamiast tych wszystkich SHL zapisać jedno MUL lub IMUL?
  Jasne, że prościej. Przy okazji dobre kilka[naście] razy wolniej.

- Dlaczego ciągle `xor rej,rej`?
  Szybsze niż `mov rej,0`, gdzie to zero musi być często zapisane 4 bajtami zerowymi. Tak więc i krótsze. Oprócz tego, dzięki instrukcji XOR lub SUB wykonanej na tym samym rejestrze, procesor wie, że ten rejestr już jest pusty. Może to przyśpieszyć niektóre operacje.

- Dlaczego na niektórych etykietach są jakieś znaki podkreślenia z przodu?
  Niektóre procedury są żywcem wyjęte z mojej biblioteki, pisząc którą musiałem zadbać, by przypadkowo nazwa jakieś mojej procedury nie była identyczna z nazwą jakiejś innej napisanej w programie korzystającym z biblioteki.
  Czy nie mogłem tego potem zmienić?
  Jasne, że mogłem. Ale nie było takiej potrzeby.

- Czemu `or rej,rej` a nie `cmp rej,0`?
  OR jest krótsze i szybsze. Można też używać `test rej,rej`, które nie zmienia zawartości rejestru.

- Czemu `or dl, "0"`?
  Bardziej czytelne niż `add/or dl,30h`. Chodzi o to, aby dodać kod ASCII zera. I można to zrobić bardziej lub mniej czytelnie.

- Co to w ogóle za instrukcja to `lea edx, [_pisz_bufor+si-1]` ?
  LEA - Load Effective Address: do rejestru EDX wpisz adres (elementu, którego) adres wynosi _pisz_bufor+SI-1. Tak więc od tej pory EDX = _pisz_bufor+SI-1, czyli wskazuje na ostatnią cyfrę w naszym buforze. Czemu odjąłem 1? Jak widać w kodzie, po wpisaniu cyfry do bufora, zwiększamy SI. Gdy nasza liczba już się skończy to SI pokazuje na następne wolne miejsce po ostatniej cyfrze, a chcemy, aby pokazywał na ostatnią. Stąd to minus jeden.

- Czemu FASM nie akceptuje EQU?
  FASM akceptuje EQU, tylko tutaj symbol był zdefiniowany po użyciu, co najwyraźniej przeszkadza FASMowi. Postawienie znaku równości zamiast EQU naprawiło sprawę.

- Czemu jest dwukropek po etykiecie zmiennej?
  Po to, aby FASM przyjął dyrektywę times. Bez dwukropka nie chciał skompilować.

Wiem, że ten program nie jest doskonały. Ale taki już po prostu napisałem...
Nie martwcie się, jeśli czegoś od razu nie zrozumiecie. Naprawdę, z czasem samo przyjdzie. Ja też przecież

nie umiałem wszystkiego od razu.

Inny program do liczb pierwszych znajdziecie tu: prime.txt.

Następnym razem coś o ułamkach i koprocesorze.

> Podstawowe prawo logiki:
> Jeżeli wiesz, że nic nie wiesz, to nic nie wiesz.
> Jeżeli wiesz, że nic nie wiesz, to coś wiesz.
> Więc nie wiesz, że nic nie wiesz.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia

(można korzystać z zamieszczonych tu procedur)

1. Napisz program, który na ekranie wyświetli liczby 90-100.
2. Napisz program sprawdzający, czy dana liczba (umieścisz ją w kodzie, nie musi być wczytywana znikąd) jest liczbą pierwszą.
3. Napisz program wypisujący dzielniki danej liczby (liczba też w kodzie).

Jak pisać programy w języku asembler pod Linuksem?

Część 5 - Koprocesor, czyli jak liczyć na ułamkach. Odwrotna Notacja Polska.

Jak zapewne większość wie, koprocesor (FPU = Floating Point Unit, NPX = Numerical Processor eXtension) służy do wykonywania działań matematycznych. Podstawowy procesor też oczywiście wykonuje działania matematyczne (dodawanie, mnożenie, ...) ale tylko na liczbach całkowitych. Z czasem jednak przyszła potrzeba wykonywania obliczeń na liczbach niecałkowitych, czyli ułamkach (liczbach zmiennoprzecinkowych). Dlatego firmy produkujące procesory zaczęły je wyposażać właśnie w układy wspomagające pracę na ułamkach. Do procesorów 8086, 80286, 80386 były dołączane jako osobne układy koprocesory: 8087, 80287, 80387 (80187 nie wprowadził żadnych istotnych nowości. Była to przeróbka 8087, a może nawet był to po prostu ten sam układ). Procesory 486SX miały jeszcze oddzielny koprocesor (80387) a od 486DX (w każdym razie u Intela) koprocesor był już na jednym chipie z procesorem. I tak jest do dziś.

Ale dość wstępu. Pora na szczegóły.

Typy danych

Zanim zaczniemy cokolwiek robić, trzeba wiedzieć, na czym ten cały koprocesor operuje.
Oprócz liczb całkowitych, FPU operuje na liczbach ułamkowych różnej precyzji:

- Pojedyncza precyzja. Liczby takie zajmują po 32 bity (4 bajty) i ich wartość maksymalna wynosi ok. 10^{39} (10^39). Znane są programistom języka C jako float.
- Podwójna precyzja. 64 bity (8 bajtów), max = ok. 10^{409} (10^409). W języku C są znane jako double
- Rozszerzona precyzja. 80 bitów (10 bajtów), max = ok. 10^{4930} (10^4930). W języku C są to long double

Jak widać, ilości bitów są oczywiście skończone. Więc nie każdą liczbę rzeczywistą da się dokładnie zapisać w postaci binarnej. Na przykład, jedna dziesiąta (0.1) zapisana dwójkowo jest ułamkiem nieskończonym okresowym! Poza tym, to, czego nas uczyli na matematyce, np. oczywista równość: a+(b-a)=b nie musi być prawdą w świecie ułamków w procesorze ze względu na brak precyzji!

Poza ułamkami, FPU umie operować na BCD (binary coded decimal). W takich liczbach 1 bajt dzieli się na 2 części po 4 bity, z których każda może mieć wartość 0-9. Cały bajt reprezentuje więc liczby od 0 do 99, w których cyfra jedności jest w młodszych 4 bitach a cyfra dziesiątek - w starszych 4 bitach.
Szczegółami zapisu liczb ułamkowych nie będziemy się tutaj zajmować. Polecam, tak jak poprzednio, strony Intela i strony AMD , gdzie znajduje się też kompletny opis wszystkich instrukcji procesora i koprocesora.

Rejestry koprocesora

Po zapoznaniu się z typami (a przede wszystkim z rozmiarami) liczb ułamkowych, powstaje pytanie: gdzie koprocesor przechowuje takie ilości danych?
FPU ma specjalnie do tego celu przeznaczonych 8 rejestrów, po 80 bitów każdy. W operacjach wewnętrznych (czyli bez pobierania lub zapisywania danych do pamięci) FPU zawsze używa rozszerzonej precyzji.

Rejestry danych nazwano st(0), st(1), ... , st(7) (NASM: st0 ... st7). Nie działają jednak one tak, jak zwykłe rejestry, lecz jak ... stos! To znaczy, że dowolnie dostępna jest tylko wartość ostatnio położona na stosie czyli wierzchołek stosu, w tym przypadku: st(0). Znaczy to, że do pamięci (FPU operuje tylko na własnych rejestrach lub pamięci - nie może używać rejestrów ogólnego przeznaczenia np. EAX itp.) może być zapisana tylko wartość z st(0), a każda wartość pobierana z pamięci idzie do st(0) a stare st(0) przesuwa się do st(1) itd. każdy rejestr przesuwa się o 1 dalej. Jeżeli brakuje na to miejsca, to FPU może wygenerować przerwanie (wyjątek) a rejestry danych będą prawdopodobnie zawierać śmieci.
Operowanie na rejestrach FPU będzie wymagało nieco więcej uwagi niż na zwykłych, ale i do tego można się przyzwyczaić.

Instrukcje koprocesora

Zacznijmy więc od omówienia kilku podstawowych instrukcji. Przez [mem] będę nazywał dane będące w pamięci (32-, 64- lub 80-bitowe, int oznacza liczbę całkowitą), st jest częstym skrótem od st(0). Jeżeli komenda kończy się na P to oznacza, że zdejmuje dane raz ze stosu, PP oznacza, że zdejmuje 2 razy, czyli st(0) i st(1).

1. Instrukcje przemieszczenia danych:
   - FLD/FILD [mem] - załaduj liczbę rzeczywistą/całkowitą z pamięci. Dla liczby rzeczywistej jest to 32, 64 lub 80 bitów. Dla całkowitej - 16, 32 lub 64 bity.
   - FST [mem32/64/80] - do pamięci idzie liczba ze st(0).
   - FSTP [mem32/64/80] - zapisz st(0) w pamięci i zdejmij je ze stosu. Znaczy to tyle, że st(1) o ile istnieje, staje się st(0) itd. każdy rejestr cofa się o 1.
   - FIST [mem16/32] - ewentualnie obciętą do całkowitej liczbę z st(0) zapisz do pamięci.
   - FISTP [mem16/32/64] - jak wyżej, tylko ze zdjęciem ze stosu.
   - FXCH st(i) - zamień st(0) z st(i).

2. Instrukcje ładowania stałych
   - FLDZ - załaduj zero. st(0) = 0.0
   - FLD1 - załaduj 1. st(0) = 1.0
   - FLDPI - załaduj pi.
   - FLDL2T - załaduj log2(10)
   - FLDL2E - załaduj log2(e)
   - FLDLG2 - załaduj log(2)=log10(2)
   - FLDLN2 - załaduj ln(2)

3. Działania matematyczne:

- dodawanie: FADD, składnia identyczna jak w odejmowaniu prostym

- odejmowanie:
  FSUB [mem32/64] st := st-[mem] ,
  FSUB st(0),st(i) st := st-st(i),
  FSUB st(i),st(0) st(i) := st(i)-st(0),
  FSUBP st(i), st(0) st(i) := st(i)-st(0) i zdejmij,
  FSUBP (bez argumentów) = FSUBP st(1),st(0) ,
  FISUB [mem16/32int] st := st-[mem]

- odejmowanie odwrotne:
  FSUBR [mem32/64] st := [mem]-st(0)
  FSUBR st(0),st(i) st := st(i)-st(0)
  FSUBR st(i),st(0) st(i) := st(0)-st(i)
  FSUBRP st(i),st(0) st(i) := st(0)-st(i) i zdejmij
  FSUBRP (bez argumentów) = FSUBRP st(1),st(0)
  FISUBR [mem16/32int] st := [mem]-st

- mnożenie: FMUL, składnia identyczna jak w odejmowaniu prostym.

- dzielenie: FDIV, składnia identyczna jak w odejmowaniu prostym.
- dzielenie odwrotne: FDIVR, składnia identyczna jak w odejmowaniu odwrotnym.

- wartość bezwzględna: FABS (bez argumentów) zastępuje st(0) jego wartością bezwzględną.
- zmiana znaku: FCHS: st(0) := -st(0).
- pierwiastek kwadratowy: FSQRT: st(0) := SQRT[st(0)]
- reszty z dzielenia: FPREM, FPREM1 st(0) := st(0) mod st(1).
- zaokrąglanie do liczby całkowitej: FRNDINT: st(0) := (int)st(0).

4. Komendy porównania:
   - FCOM/FCOMP/FCOMPP, FUCOM/FUCOMP/FUCOMPP, FICOM/FICOMP, FCOMI/FCOMIP, FUCOMI/FUCOMIP, FTST, FXAM.
   Tutaj trzeba trochę omówić sytuację. FPU oprócz rejestrów danych zawiera także rejestr kontrolny (16 bitów) i rejestr stanu (16 bitów).
   W rejestrze stanu są 4 bity nazwane C0, C1, C2 i C3. To one wskazują wynik ostatniego porównania, a układ ich jest taki sam, jak flag procesora, co pozwala na ich szybkie przeniesienie do flag procesora. Aby odczytać wynik porównania, należy zrobić takie coś:

```
fcom
fstsw   ax      ; tylko od 386. Inaczej:
                ; fstsw word ptr [zmienna] / mov ax,[zmienna]
sahf            ; AH -> flagi
```

   i używać normalnych komend JE, JB itp.
   FCOM st(n)/[mem] - porównaj st(0) z st(n) (lub zmienną w pamięci) bez zdejmowania st(0) ze stosu FPU
   FCOMP st(n)/[mem] - porównaj st(0) z st(n) (lub zmienną w pamięci) i zdejmij st(0)
   FCOMPP - porównaj st(0) z st(1) i zdejmij oba ze stosu
   FICOM [mem] - porównaj st(0) ze zmienną całkowitą 16- lub 32-bitową w pamięci
   FICOMP [mem] - porównaj st(0) ze zmienną całkowitą 16- lub 32-bitową w pamięci, zdejmij

st(0)
FCOMI st(0), st(n) - porównaj st(0) z st(n) i ustaw flagi *procesora*, nie tylko FPU
FCOMIP st(0), st(n) - porównaj st(0) z st(n) i ustaw flagi *procesora*, nie tylko FPU, zdejmij st(0)

Komendy kończące sie na I lub IP zapisują swój wynik bezpośrednio do flag procesora. Można tych flag od razu używać (JZ, JA, ...). Te komendy są dostępne tylko od 386.

FTST porównuje st(0) z zerem.
FXAM bada, co jest w st(0) - prawidłowa liczba, błąd (NaN = Not a Number) czy 0.

5. Instrukcje trygonometryczne:
   - FSIN, FCOS, FSINCOS, FPTAN, FPATAN
   st(0) := funkcja[st(0)].
   FPTAN = partial tangent = tangens,
   FPATAN = arcus tangens.

6. Logarytmiczne, wykładnicze:
   - FYL2X st(1) := st(1)*log2[st(0)] i zdejmij
   - FYL2XPI st(1) := st(1)*log2[st(0) + 1.0] i zdejmij
   - F2XM1 st(0) := 2^[st(0)] - 1

7. Instrukcje kontrolne:
   - FINIT/FNINIT - inicjalizacja FPU. Litera N po F oznacza, aby nie brać po uwagę potencjalnych niezałatwionych wyjątków.
   - FLDCW, FSTCW/FNSTCW - Load/Store control word - zapisuje 16 kontrolnych bitów do pamięci, gdzie można je zmieniać np. aby zmienić sposób zaokrąglania liczb.
   - FSTSW/FNSTSW - zapisz do pamięci (lub rejestru AX) słowo statusu, czyli stan FPU
   - FCLEX/FNCLEX - wyczyść wyjątki
   - FLDENV, FSTENV/FNSTENV - wczytaj/zapisz środowisko (rejestry stanu, kontrolny i kilka innych, bez rejestrów danych). Wymaga 14 albo 28 bajtów pamięci, w zależności od trybu pracy procesora (rzeczywisty - DOS lub chroniony - Windows/Linux).
   - FRSTOR, FSAVE/FNSAVE - jak wyżej, tylko że z rejestrami danych. Wymaga 94 lub 108 bajtów w pamięci, zależnie od trybu procesora.
   - FINCSTP, FDECSTP - zwiększ/zmniejsz wskaźnik stosu - przesuń st(0) na st(7), st(1) na st(0) itd. oraz w drugą stronę, odpowiednio.
   - FFREE - zwolnij podany rejestr danych
   - FNOP - no operation. Nic nie robi, ale zabiera czas.
   - WAIT/FWAIT - czekaj, aż FPU skończy pracę. Używane do synchronizacji z CPU.

Przykłady

Dość już teorii, pora na przykłady. Programiki te wymyśliłem pisząc ten kurs.

Przykład 1 (NASM):

(przeskocz program wyświetlający częstotliwość zegara)

```
; z wyświetlaniem:
;   nasm -O999 -f elf -o fpu1.o fpu1.asm
;   ld -s -o fpu1 fpu1.o bibl/lib/libasmio.a
; bez wyświetlania:
;   nasm -O999 -f elf -o fpu1.o fpu1.asm
;   ld -s -o fpu1 fpu1.o

section .text

; jeśli nie chcesz wyświetlania, usuń tę linijkę niżej:
%include "bibl/incl/linuxbsd/nasm/std_bibl.inc"

global _start

_start:
        finit                   ; zawsze o tym pamiętaj !!!!

        fild    dword [dzielna] ; ładujemy dzielną. st(0) = 1234DD
        fild    dword [dzielnik]; ładujemy dzielnik. st(0) = 10000h,
                                ; st(1) = 1234DD
        fdivp   st1             ; dzielimy. st(1) := st(1)/st(0) i pop.
                                ; st(0) ~= 18.2
                                ; FASM: fdivp st(1)

        fstp    tword [iloraz]  ; zapisujemy st(0) do pamięci i
                                ; zdejmujemy ze stosu

; jeśli nie chcesz wyświetlania, usuń te 3 linijki niżej:
        mov     edi, iloraz
        piszd80                 ; wyświetl wynik
        nwln                    ; przejdź do nowego wiersza

        mov     eax, 1
        xor     ebx, ebx
        int     80h

section .data

align 8                         ; NASM w tym miejscu dorobi kilka
                                ; NOP-ów (instrukcji nic nie
                                ; robiących, ale zabierających czas),
                                ; aby adres dzielił się
                                ; przez 8 (patrz dalej).

dzielna         dd 1234ddh      ; 4013 91a6 e800 0000 0000
dzielnik        dd 10000h

section .bss                    ; dane w sekcji nazwanej .BSS nie
                                ; będą fizycznie umieszczone
                                ; w programie, a dopiero w pamięci.
                                ; Zaoszczędzi to
                                ; miejsce = długość pliku.
                                ; Można tu umieszczać tylko
                                ; niezainicjalizowane dane.

iloraz          rest 1          ; rezerwacja 10 bajtów.
```

Teraz wersja dla FASMa:
(przeskocz ten program w wersji FASMa)

```
; z wyświetlaniem:
;   fasm fpu1.asm
;   ld -s -o fpu1 fpu1.o bibl/lib/libasmio.a
; bez wyświetlania:
;   fasm fpu1.asm
;   ld -s -o fpu1 fpu1.o

format ELF

public _start

section ".text" executable
; jeśli nie chcesz wyświetlania, usuń tę linijkę niżej:
include "bibl/incl/linuxbsd/fasm/std_bibl.inc"

_start:
        finit                   ; zawsze o tym pamiętaj !!!!

        fild    dword [dzielna] ; ładujemy dzielną. st(0) = 1234DD
        fild    dword [dzielnik]; ładujemy dzielnik. st(0) = 10000h,
                                ; st(1) = 1234DD
        fdivp   st1, st0        ; dzielimy. st(1) := st(1)/st(0) i
                                ; pop. st(0) ~= 18.2
                                ; FASM: fdivp st(1)

        fstp    tword [iloraz]  ; zapisujemy st(0) do pamięci i
                                ; zdejmujemy ze stosu

; jeśli nie chcesz wyświetlania, usuń te 3 linijki niżej:
        mov     edi, iloraz
        piszd80                 ; wyświetl wynik
        nwln                    ; przejdź do nowego wiersza

        mov     eax, 1
        xor     ebx, ebx
        int     80h

section ".data" writeable align 8

dzielna         dd 1234ddh      ; 4013 91a6 e800 0000 0000
dzielnik        dd 10000h

section ".bss" writeable
iloraz          dt 0.0
```

Ten przykład do zmiennej iloraz wstawia częstotliwość zegara komputerowego (ok. 18,2 Hz) i wyświetla ją przy użyciu jednej z procedur z mojej biblioteki (można to usunąć).
Należy zwrócić uwagę na zaznaczanie rozmiarów zmiennych (dword/tword). Dyrektywa align 8 ustawia kolejną zmienną/etykietę tak, że jej adres dzieli się przez 8 (qword = 8 bajtów). Dzięki temu, operacje na pamięci są szybsze (np. pobranie 8 bajtów: zamiast 3 razy pobierać po 4 bajty, bo akurat tak się zdarzyło, że miała jakiś nieparzysty adres, pobiera się 2x4 bajty). Rzecz jasna, skoro zmienna dzielna (i dzielnik) ma 4 bajty, to adresy zmiennych dzielnik i iloraz też będą podzielne przez 4. Ciąg cyfr po średniku to ułamkowa reprezentacja dzielnej. Skomplikowane, prawda? Dlatego nie chciałem tego omawiać.

Przykład 2: czy sinus liczby pi rzeczywiście jest równy 0 (w komputerze)?
(przeskocz program z sinusem)

```
; format ELF executable         ; tylko dla FASMa
```

```
; entry _start

; FASM: segment readable executable
section .text

global _start                   ; FASM: usunąć

_start:
        finit                   ; zawsze o tym pamiętaj !!!!

        fldpi                   ; wczytujemy PI
        fsin                    ; obliczamy sinus z PI
        ftst                    ; porównujemy st(0) z zerem.
        fstsw   ax              ; zapisujemy rejestr stanu
                                ; bezpośrednio w AX.


        sahf                    ; AH idzie do flag
        mov     ah,9            ; AH=9, flagi niezmienione
        je      jest_zero       ; st(0) = 0?
                                ; Jeśli tak, to wypisz, że jest

        mov     ecx, nie_zero
        mov     edx, dlugosc_nie_zero

        jmp     pisz

jest_zero:
        mov     ecx, zero
        mov     edx, dlugosc_zero

pisz:
        mov     eax, 4
        mov     ebx, 1
        int     80h             ; wypisz jedną z wiadomości.

        mov     eax, 1
        xor     ebx, ebx
        int     80h

; FASM: segment readable writeable
section .data

nie_zero        db      "Sin(PI) != 0",0ah
dlugosc_nie_zero        equ     $ - nie_zero
                ; FASM: "=" zamiast "equu"

zero            db      "Sin(PI) = 0",0ah
dlugosc_zero            equ     $ - zero
                ; FASM: "=" zamiast "equu"
```

Przykład 3: czy pierwiastek z 256 rzeczywiście jest równy 16, czy 200 jest kwadratem liczby całkowitej? (przeskocz ten program)

```
; format ELF executable         ; tylko dla FASMa
; entry _start

; FASM: segment readable executable
section .text

global _start                   ; FASM: usunąć
```

```
_start:
        finit                   ; zawsze o tym pamiętaj !!!!

        fild word  [dwa_pie_sze]; st(0) = 256
        fsqrt                   ; st(0) = sqrt(256)
        fild word [szesnascie]  ; st(0) = 16, st(1) = sqrt(256)
        fcompp                  ; porównaj st(0) i st(1) i
                                ; zdejmij oba. st: [pusty]

        fstsw   ax
        sahf

        mov     ah,9
        je      tak256
        mov     ecx, nie_256
        mov     edx, dlugosc_nie_256

        jmp     short pisz_256

tak256:
        mov     ecx, tak_256
        mov     edx, dlugosc_tak_256

pisz_256:

        mov     eax, 4
        mov     ebx, 1
        int     80h             ; wypisz stosowną wiadomość

                                ; do zapisu stanu stosu, czyli
                                ; rejestrów danych FPU można
                                ; używać takiego schematu zapisu,
                                ; który jest krótszy:
                                ; st:  (0),  (1),  (2),  ... , (7)

        fild word [dwiescie]    ; st: 200
        fsqrt                   ; st: sqrt(200)
        fld     st0             ; do st(0) wczytaj st(0).
                                ; st: sqrt(200), sqrt(200)
        frndint                 ; zaokrąglij do liczby całkowitej.
                                ; st:  (int)sqrt(200), sqrt(200)
        fcompp                  ; porównaj i zdejmij oba.
        fstsw ax
        sahf

        mov ah,9
        je tak200

        mov     ecx, nie_200
        mov     edx, dlugosc_nie_200

        jmp short pisz_200

tak200:
        mov     ecx, tak_200
        mov     edx, dlugosc_tak_200

pisz_200:

        mov     eax, 4
        mov     ebx, 1
```

```
        int     80h             ; wypisz stosowną wiadomość

        mov     eax, 1
        xor     ebx, ebx
        int     80h

; FASM: segment readable writeable
section .data

dwa_pie_sze     dw      256
dwiescie        dw      200
szesnascie      dw      16

nie_256         db      "SQRT(256) != 16", 0ah
dlugosc_nie_256         equ     $ - nie_256
                ; FASM: "=" zamiast "equ"

tak_256         db      "SQRT(256) = 16", 0ah
dlugosc_tak_256         equ     $ - tak_256
                ; FASM: "=" zamiast "equ"

nie_200 db "Liczba 200 nie jest kwadratem liczby calkowitej", 0ah
dlugosc_nie_200         equ     $ - nie_200
                ; FASM: "=" zamiast "equ"

tak_200 db "Liczba 200 jest kwadratem liczby calkowitej", 0ah
dlugosc_tak_200         equ     $ - tak_200
                ; FASM: "=" zamiast "equ"
```

Dwa ostatnie programiki zbiłem u siebie w jeden i przetestowałem. Wyszło, że sinus PI jest różny od zera, reszta była prawidłowa.

Oczywiście, w tych przykładach nie użyłem wszystkich instrukcji koprocesora (nawet spośród tych, które wymieniłem). Mam jednak nadzieję, że te proste programy rozjaśnią nieco sposób posługiwania się koprocesorem.

Odwrotna Notacja Polska (Reverse Polish Notation, RPN)

Ładnie brzmi, prawda? Ale co to takiego?
Otóż, bardzo dawno temu pewien polski matematyk, Jan Łukasiewicz, wymyślił taki sposób zapisywania działań, że nie trzeba w nim używać nawiasów. Była to notacja polska. Sposób ten został potem dopracowany przez Charlesa Hamblina na potrzeby informatyki - w ten sposób powstała Odwrotna Notacja Polska. W zapisie tym argumenty działania zapisuje przed symbolem tego działania. Dla jasności podam teraz kilka przykładów:
(przeskocz przykłady na ONP)

```
   Zapis tradycyjny                ONP
        a+b                     a b +
        a+b+c                   a b + c + ; ab+ stanowi pierwszy argument
                                          ; drugiego dodawania
        c+b+a                   c b + a +
        (a+b)*c                 a b + c *
        c*(a+b)                 c a b + *
        (a+b)*c+d               a b + c * d +
```

```
(a+b)*c+d*a             a b + c * d a * +
(a+b)*c+d*(a+c)         a b + c * d a c + * +
(a+b)*c+(a+c)*d         a b + c * a c + d * +
(2+5)/7+3/5             2 5 + 7 / 3 5 / +
```

Ale po co to komu i dlaczego mówię o tym akurat w tej części?
Powód jest prosty: jak sie dobrze przyjrzeć zapisowi działania w ONP, to można zobaczyć, że mówi on o kolejności działań, jakie należy wykonać na koprocesorze. Omówimy to na przykładzie:
(przeskocz ilustrację relacji między ONP a koprocesorem)

```
Zapis tradycyjny (jeden z powyższych przykładów):
(a+b)*c+(a+c)*d

Zapis w ONP:
a b + c * a c + d * +

Uproszczony kod programu:

fld     [a]
fld     [b]
faddp   st1, st0
fld     [c]
fmulp   st1, st0
fld     [a]
fld     [c]
faddp   st1, st0
fld     [d]
fmulp   st1, st0
faddp   st1, st0

Teraz st0 jest równe wartości całego wyrażenia.
```

Jak widać, ONP znacznie upraszcza przetłumaczenie wyrażenia na kod programu. Jednak, kod nie jest optymalny. Można byłoby na przykład zachować wartości zmiennych a i c na stosie i wtedy nie musielibyśmy ciągle pobierać ich z pamięci. Dlatego w krytycznych sekcjach kodu stosowanie zasad ONP nie jest zalecane. Ale w większości przypadków Odwrotna Notacja Polska sprawuje się dobrze i uwalnia programistów od obowiązku zgadywania kiedy i jakie działanie wykonać.
Pamiętajcie tylko, że *stos koprocesora może pomieścić tylko 8 zmiennych*!

Następnym razem o SIMD.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia

1. Napisz program, który sprawdzi (wyświetli stosowną informację), czy liczba PI dzielona przez samą siebie daje dokładne 1.0

2. Napisz program obliczający (nie wyświetlający) wartość 10*PI. Potem sprawdź, czy sinus tej liczby jest zerem.

3. Napisz program mówiący, która z tych liczb jest większa: PI czy log2(10).

4. Napisz program sprawdzający, czy 10*PI - PI - PI - PI - PI = 5*PI.

5. Zamień na ONP:
   a/c/d + b/c/d
   a/(c*d) + b/(c*d)
   (a+b)/c/d
   (a+b)/(c*d)

6. Zamień z ONP na zapis tradycyjny (daszek ^ oznacza potęgowanie):
   ab*cd*e/-
   a5/c7/ed-9/*+
   a3+b/de+6^-
   dc-7b*2^/

Jak pisać programy w języku asembler pod Linuksem?

Część 6 - SIMD, czyli jak działa MMX.

- A cóż to takiego to SIMD ?! - zapytacie.
Już odpowiadam.
SIMD = Single Instruction, Multiple Data = jedna instrukcja, wiele danych.
Jest to technologia umożliwiająca jednoczesne wykonywanie tej samej instrukcji na kilku wartościach. Na pewno znany jest wam co najmniej jeden przykład zastosowania technologii SIMD. Jest to MultiMedia Extensions, w skrócie MMX u Intela, a 3DNow! u AMD. Innym mniej znanym zastosowaniem jest SSE, które omówię później.
Zaczniemy od omówienia, jak właściwie działa to całe MMX.

MMX / 3DNow!

Technologia MMX operuje na 8 rejestrach danych, po 64 bity każdy, nazwanych mm0 ... mm7. Niestety, rejestry te nie są prawdziwymi (oddzielnymi) rejestrami - są częściami rejestrów koprocesora (które, jak pamiętamy, mają po 80 bitów każdy). Pamiętajcie więc, że *nie można naraz wykonywać operacji na FPU i MMX/3DNow!*.

Rejestry 64-bitowe służą do umieszczania w nich danych spakowanych. Na czym to polega? Zamiast mieć np. 32 bity w jednym rejestrze, można mieć dwa razy po 32. Tak więc rejestry mieszczą 2 podwójne słowa (dword, 32 bity) lub 4 słowa (word, 16 bitów) lub aż 8 spakowanych bajtów.

Zajmijmy się omówieniem instrukcji operujących na tych rejestrach.
Instrukcje MMX można podzielić na kilka grup (nie wszystkie instrukcje będą tu wymienione):

- instrukcje transferu danych:
  - MOVD mmi, rej32/mem32 (i=0,...,7)
  - MOVQ mmi, mmj/mem64 (i,j=0,...,7)

- instrukcje arytmetyczne:
  - dodawanie normalne: PADDB (bajty) / PADDW (słowa)/ PADDD (dword-y)

  - dodawanie z nasyceniem ze znakiem: PADDSB (bajty)/ PADDSW (słowa).
    Jeżeli wynik przekracza 127 lub 32767 (bajty/słowa), to jest do tej wartości zaokrąglany, a NIE jest tak, że nagle zmienia się na ujemny. Daje to lepszy efekt, np. w czasie słuchania muzyki czy oglądania filmu. Hipotetyczny przykład: 2 kolory szare z dadzą w sumie czarny a nie coś pośrodku skali kolorów.

  - dodawanie z nasyceniem bez znaku: PADDUSB / PADDUSW.
    Jeżeli wynik przekracza 255 lub 65535, to jest do tej wartości zaokrąglany.

- odejmowanie normalne: PSUBB (bajty)/ PSUBW (słowa)/ PSUBD (dword-y)

- odejmowanie z nasyceniem ze znakiem: PSUBSB (bajty)/ PSUBSW (słowa).
  Jeśli wynik jest mniejszy niż -128 lub -32768 to jest do tej wartości zaokrąglany.

- odejmowanie z nasyceniem bez znaku: PSUBUSB (bajty) / PSUBUSW (słowa)
  Jeśli wynik jest mniejszy niż 0, to staje się równy 0.

- mnożenie:
  - PMULHRWC, PMULHRIW, PMULHRWA- mnożenie spakowanych słów, zaokrąglanie, zapisanie tylko starszych 16 bitów wyniku (z 32).
  - PMULHUW - mnożenie spakowanych słów bez znaku, zachowanie starszych 16 bitów
  - PMULHW, PMULLW - mnożenie spakowanych słów bez znaku, zapisanie starszych/młodszych 16 bitów (odpowiednio).
  - PMULUDQ - mnożenie spakowanych dwordów bez znaku

- mnożenie i dodawanie: PMADDWD - do młodszego dworda rejestru docelowego idzie suma iloczynów 2 najmłodszych słów ze sobą i 2 starszych (bity 16-31) słów ze sobą. Do starszego dworda - suma iloczynów 2 słów 32-47 i 2 słów 48-63.

- instrukcje porównawcze:
  Zostawiają w odpowiednim bajcie/słowie/dwordzie same jedynki (FFh/FFFFh/FFFFFFFFh) gdy wynik porównania był prawdziwy, same zera - gdy fałszywy.
  - na równość PCMPEQB / PCMPEQW / PCMPEQD (EQ oznacza równość)
  - na większe niż: PCMPGTPB / PCMPGTPW / PCMPGTPD (GT oznacza greater than, czyli większy)

- instrukcje konwersji:
  - pakowanie: PACKSSWB / PACKSSDW, PACKUSWB - upychają słowa/dwordy do bajtów/słów i pozostawiają w rejestrze docelowym.
  - rozpakowania starszych części (unpack high): PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ - pobierają starsze części bajtów/słów/dwordów z jednego i drugiego rejestru, mieszają je i zostawiają w pierwszym.
  - rozpakowania młodszych części (unpack low): PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ - jak wyżej, tylko pobierane są młodsze części

- instrukcje logiczne:
  - PAND (bitowe AND)
  - PANDN (najpierw bitowe NOT pierwszego rejestru, potem jego bitowe AND z drugim rejestrem)
  - POR (bitowe OR)
  - PXOR (bitowe XOR)

- instrukcje przesunięcia (analogiczne do znanych SHL, SHR i SAR, odpowiednio):
  - w lewo: PSLLW (słowa) / PSLLD (dword-y), PSLLQ (qword)
  - w prawo, logiczne: PSRLW (słowa) / PSRLD (dword-y), PSRLQ (qword)
  - w prawo, arytmetyczne: PSRAW (słowa)/ PSRAD (dword-y)

- instrukcje stanu MMX:

- EMMS - Empty MMX State - ustawia rejestry FPU jako wolne, umożliwiając ich użycie. Ta instrukcja musi być wykonana za każdym razem, gdy kończymy pracę z MMX i chcemy zacząć pracę z FPU.

Rzadko która z tych instrukcji traktuje rejestr jako całość, częściej operuje na poszczególnych wartościach osobno, równolegle.

Spróbuję teraz podać kilka przykładów zastosowania MMX.

Przykład 1. Dodawanie dwóch tablic bajtów w pamięci. Bez MMX mogłoby to wyglądać mniej więcej tak:
(przeskocz dodawanie tablic)

```
; EDX - adres pierwszej tablicy bajtów
; ESI - adres drugiej tablicy bajtów
; EDI - adres docelowej tablicy bajtów
; ECX - liczba bajtów w tablicach. Przyjmiemy, że różna od zera...

        petla:
                mov al, [edx]   ; pobierz bajt z pierwszej
                add al, [esi]   ; dodaj bajt z drugiej
                mov [edi], al   ; zapisz bajt w docelowej
                inc edx         ; zwiększ o 1 indeksy tablic
                inc esi
                inc edi
                loop petla      ; działaj, dopóki ECX różne od 0.
```

A z MMX:
(przeskocz dodawanie tablic z MMX)

```
        mov ebx, ecx    ; EBX = liczba bajtów
        and ebx, 7      ; będziemy brać po 8 bajtów - obliczamy
                        ; więc resztę z dzielenia przez 8

        shr ecx, 3      ; dzielimy ECX przez 8
petla:
        movq mm0, [edx] ; pobierz 8 bajtów z pierwszej tablicy
        paddb mm0, [esi]; dodaj 8 spakowanych bajtów z drugiej
        movq [edi], mm0 ; zapisz 8 bajtów w tablicy docelowej
        add edx, 8      ; zwiększ indeksy do tablic o 8
        add esi, 8
        add edi, 8
        loop petla      ; działaj, dopóki ECX różne od 0.

        test ebx, ebx   ; czy EBX = 0?
        jz koniec       ; jeśli tak, to już skończyliśmy

        mov ecx, ebx    ; ECX = resztka, co najwyżej 7 bajtów.
                        ; te kopiujemy tradycyjnie
petla2:
        mov al, [edx]   ; pobierz bajt z pierwszej
        add al, [esi]   ; dodaj bajt z drugiej
        mov [edi], al   ; zapisz bajt w docelowej
        inc edx         ; zwiększ o 1 indeksy do tablic
        inc esi
        inc edi
```

```
        loop petla2     ; działaj, dopóki ECX różne od 0
koniec:

        emms            ; wyczyść rejestry MMX, by FPU mogło z nich korzystać
```

Podobnie będą przebiegać operacje PAND, POR, PXOR, PANDN.

Przy dużych ilościach danych, sposób drugi będzie wykonywał około 8 razy mniej instrukcji niż pierwszych, bo dodaje na raz 8 bajtów. I o to właśnie chodziło.

Przykład 2. Kopiowanie pamięci.
Bez MMX:
(przeskocz kopiowanie pamięci)

```
; DS:ESI - źródło
; ES:EDI - cel
; ECX - ilość bajtów
        mov ebx, ecx    ; EBX = ilość bajtów
        and ebx, 3      ; EBX = reszta z dzielenia liczby bajtów przez 4
        shr ecx, 2      ; ECX = liczba bajtów dzielona przez 4

        cld             ; kierunek: do przodu
        rep movsd       ; dword z DS:ESI idzie pod ES:EDI, EDI:=EDI+4,
                        ; ESI:=ESI+4, dopóki ECX jest różny od 0
        mov ecx, ebx    ; ECX = liczba pozostałych bajtów
        rep movsb       ; resztkę kopiujemy po bajcie
```

Z MMX:
(przeskocz kopiowanie pamięci z MMX)

```
                mov ebx, ecx    ; EBX = ilość bajtów
                and ebx, 7      ; EBX = reszta z dzielenia liczby bajtów
                                ; przez 8

                shr ecx, 3      ; ECX = liczba bajtów dzielona przez 8
        petla:
                movq mm0, [esi] ; MM0 = 8 bajtów z tablicy pierwszej
                movq [edi], mm0 ; kopiujemy zawartość MM0 pod [EDI]
                add esi, 8      ; zwiększamy indeksy tablic o 8
                add edi, 8
                loop petla      ; działaj, dopóki ECX różne od 0

                mov ecx, ebx    ; ECX = liczba pozostałych bajtów
                cld             ; kierunek: do przodu
                rep movsb       ; resztkę kopiujemy po bajcie

                emms            ; wyczyść rejestry MMX
```

lub, dla solidniejszych porcji danych:
(przeskocz kolejne kopiowanie pamięci)

```
                mov ebx, ecx    ; EBX = ilość bajtów
                and ebx, 63     ; EBX = reszta z dzielenia liczby bajtów
                                ; przez 64
                shr ecx, 6      ; ECX = liczba bajtów dzielona przez 64
        petla:
                ; kopiuj 64 bajty spod [ESI] do rejestrów MM0, ... MM7
```

```
        movq mm0, [esi]
        movq mm1, [esi+8]
        movq mm2, [esi+16]
        movq mm3, [esi+24]
        movq mm4, [esi+32]
        movq mm5, [esi+40]
        movq mm6, [esi+48]
        movq mm7, [esi+56]

        ; kopiuj 64 bajty z rejestrów MM0, ... MM7 do [EDI]
        movq [edi   ], mm0
        movq [edi+8 ], mm1
        movq [edi+16], mm2
        movq [edi+24], mm3
        movq [edi+32], mm4
        movq [edi+40], mm5
        movq [edi+48], mm6
        movq [edi+56], mm7

        add esi, 64     ; zwiększ indeksy do tablic o 64
        add edi, 64
        loop petla      ; działaj, dopóki ECX różne od 0

        mov ecx, ebx    ; ECX = liczba pozostałych bajtów
        cld             ; kierunek: do przodu
        rep movsb       ; resztkę kopiujemy po bajcie

        emms            ; wyczyść rejestry MMX
```

Przykład 3. Rozmnożenie jednego bajtu na cały rejestr MMX.
(przeskocz rozmnażanie bajtu)

```
; format ELF executable         ; tylko dla FASMa
; entry _start

; FASM: segment readable executable
section .text

global _start                   ; FASM: usunąć tą linijkę

_start:

        movq mm0, [wart1]       ; mm0 = 00 00 00 00 00 00 00 33
                                ;  (33h = kod ASCII cyfry 3)

        punpcklbw mm0, mm0      ; do najmłodszego słowa włóż najmłodszy bajt
                                ; mm0 i najmłodszy bajt mm0 (czyli ten sam)
                                ; mm0 = 00 00 00 00 00 00 33 33

        punpcklwd mm0, mm0      ; do najmłodszego dworda włóż dwa razy
                                ; najmłodsze słowo mm0
                                ; mm0 = 00 00 00 00 33 33 33 33

        punpckldq mm0, mm0      ; do najmłodszego (i jedynego) qworda włóż 2x
                                ; najmłodszy dword mm0 obok siebie
                                ; mm0 = 33 33 33 33 33 33 33 33

        movq [wart2], mm0
```

```
        emms                    ; wyczyść rejestry MMX

        mov     eax, 4
        mov     ebx, 1
        mov     ecx, wart2
        mov     edx, 9          ; wartość2 + znak nowej linii
        int     80h             ; wyświetl

        mov     eax, 1
        xor     ebx, ebx
        int     80h

; FASM: segment readable writeable
section .data

wart1:  db      "3"
        times 7 db 0                    ; trójka i 7 bajtów zerowych

wart2:  times   8       db      2       ; 8 bajtów o wartości 2 != 33h

nowa_linia      db      0ah
```

Kompilujemy, uruchamiamy i ... rzeczywiście na ekranie pojawia się upragnione osiem trójek!

Technologia MMX może być używana w wielu celach, ale jej najbardziej korzystną cechą jest właśnie równoległość wykonywanych czynności, dzięki czemu można oszczędzić czas procesora.

Technologia SSE

Streaming SIMD Extensions (SSE), Pentium III lub lepszy oraz najnowsze procesory AMD
Streaming SIMD Extensions 2 (SSE 2), Pentium 4 lub lepszy oraz AMD64
Streaming SIMD Extensions 3 (SSE 3), Xeon lub lepszy oraz AMD64

Krótko mówiąc, SSE jest dla MMX tym, czym FPU jest dla CPU. To znaczy, SSE przeprowadza równoległe operacje na liczbach ułamkowych.
SSE operuje już na całkowicie osobnych rejestrach nazwanych xmm0, ..., xmm7 po 128 bitów każdy. W trybie 64-bitowym dostępne jest dodatkowych 8 rejestrów: xmm8, ..., xmm15.
Prawie każda operacja związana z danymi w pamięci musi mieć te dane ustawione na 16-bajtowej granicy, czyli jej adres musi się dzielić przez 16. Inaczej generowane jest przerwanie (wyjątek).

SSE 2 różni się od SSE kilkoma nowymi instrukcjami konwersji ułamek-liczba całkowita oraz tym, że może operować na liczbach ułamkowych rozszerzonej precyzji (64 bity).

U AMD częściowo 3DNow! operuje na ułamkach, ale co najwyżej na dwóch gdyż są to rejestry odpowiadające MMX, a więc 64-bitowe. 3DNow! Pro jest odpowiednikiem SSE w procesorach AMD. Odpowiedniki SSE2 i SSE3 pojawił się w AMD64.

Instrukcje SSE (nie wszystkie będą wymienione):

- Przemieszczanie danych:
  - MOVAPS - move aligned packed single precision floating point values - przemieść ułożone (na granicy 16 bajtów) spakowane ułamki pojedynczej precyzji (4 sztuki po 32 bity)
  - MOVUPS - move unaligned (nieułożone) packed single precision floating point values
  - MOVSS - move scalar (1 sztuka, najmłodsze 32 bity rejestru) single precision floating point value

- Arytmetyczne:
  - ADDPS - add packed single precision floating point values = dodawanie czterech ułamków do czterech
  - ADDSS - add scalar single precision floating point values = dodawanie jednego ułamka do innego
  - MULPS - mnożenie spakowanych ułamków, równolegle, 4 pary
  - MULSS - mnożenie jednego ułamka przez inny
  - DIVPS - dzielenie spakowanych ułamków, równolegle, 4 pary
  - DIVSS - dzielenie jednego ułamka przez inny
  - obliczanie odwrotności ułamków, ich pierwiastków, odwrotności pierwiastków, znajdowanie wartości największej i najmniejszej

- Logiczne:
  - ANDPS - logiczne AND spakowanych wartości (ale oczywiście tym bardziej zadziała dla jednego ułamka w rejestrze)
  - ANDNPS - AND NOT (najpierw bitowe NOT pierwszego rejestru, potem jego bitowe AND z drugim rejestrem) dla spakowanych
  - ORPS - OR dla spakowanych
  - XORPS - XOR dla spakowanych

- Instrukcje porównania: CMPPS, CMPSS, (U)COMISS
- Instrukcje tasowania i rozpakowywania. Podobne działanie jak odpowiadające instrukcje MMX.
- Instrukcje konwersji ułamek->liczba całkowita i na odwrót.
- Instrukcje operujące na liczbach całkowitych 64-bitowych (lub 128-bitowych w SSE 2)

W większości przypadków instrukcje dodane w SSE 2 różnią się od powyższych ostatnią literą, którą jest D, co oznacza double precision, np. MOVAPD.

No i krótki przykładzik. Inne wersja procedury do kopiowania pamięci. Tym razem z SSE.
(przeskocz kopiowanie pamięci z SSE)

```
; Tylko jeśli ESI i EDI dzieli się przez 16! Inaczej używać MOVUPS.
                mov ebx, ecx    ; EBX = ilość bajtów
                and ebx, 127    ; EBX = reszta z dzielenia liczby bajtów
                                ; przez 128
                shr ecx, 7      ; ECX = liczba bajtów dzielona przez 128
        petla:
                ; kopiuj 128 bajtów spod [ESI] do rejestrów XMM0, ... XMM7
                movaps xmm0, [esi]
                movaps xmm1, [esi+16]
                movaps xmm2, [esi+32]
                movaps xmm3, [esi+48]
                movaps xmm4, [esi+64]
                movaps xmm5, [esi+80]
                movaps xmm6, [esi+96]
```

```
        movaps xmm7, [esi+112]

        ; kopiuj 128 bajtów z rejestrów XMM0, ... XMM7 do [EDI]
        movaps [edi    ], xmm0
        movaps [edi+16 ], xmm1
        movaps [edi+32 ], xmm2
        movaps [edi+48 ], xmm3
        movaps [edi+64 ], xmm4
        movaps [edi+80 ], xmm5
        movaps [edi+96 ], xmm6
        movaps [edi+112], xmm7

        add esi, 128    ; zwiększ indeksy do tablic o 128
        add edi, 128
        loop petla      ; działaj, dopóki ECX różne od 0

        mov ecx, ebx    ; ECX = liczba pozostałych bajtów
        cld             ; kierunek: do przodu
        rep movsb       ; resztkę kopiujemy po bajcie
```

Nie jest to ideał, przyznaję. Można było np. użyć instrukcji wspierających pobieranie danych z pamięci: PREFETCH.

A teraz coś innego: rozdzielanie danych. Przypuśćmy, że z jakiegoś urządzenia (lub pliku) czytamy bajty w postaci XYXYXYXYXY..., a my chcemy je rozdzielić na 2 tablice, zawierające tylko XXX... i YYY... (oczywiście bajty mogą mieć różne wartości, ale idea jest taka, że co drugi chcemy mieć w drugiej tablicy). Oto, jak można tego dokonać z użyciem SSE. *To jest tylko fragment programu.*
(przeskocz rozdzielanie bajtów)

```
mov     eax, 4                  ; funkcja zapisu do pliku
mov     ebx, 1                  ; na stdout (ekran)
mov     ecx, dane_pocz
mov     edx, dane_pocz_dl
int     80h

mov     eax, 4                  ; funkcja zapisu do pliku
mov     ebx, 1                  ; na stdout (ekran)
mov     ecx, dane
mov     edx, dane_dl
int     80h                     ; wypisz dane początkowe


movaps          xmm0, [dane]
movaps          xmm1, xmm0
        ; XMM1=XMM0 = X1Y1 X2Y2 X3Y3 X4Y4 X5Y5 X6Y6 X7Y7 X8Y8

packuswb        xmm0, xmm0
        ; XMM0 = Y1Y2 Y3Y4 Y5Y6 Y7Y8 Y1Y2 Y3Y4 Y5Y6 Y7Y8

psrlw           xmm1, 8
        ; XMM1 = 0 X1 0 X2 0 X3 0 X4 0 X5 0 X6 0 X7 0 X8
packuswb        xmm1, xmm1
        ; XMM1 = X1X2 X3X4 X5X6 X7X8 X1X2 X3X4 X5X6 X7X8


movq    [dane2], xmm0   ; dane2 ani dane1 już nie mają adresu
                        ; podzielnego przez 16,
                        ; więc nie można użyć MOVAPS
                        ; a my i tak chcemy tylko 8 bajtów
```

```
        movq    [dane1], xmm1

        mov     eax, 4                  ; funkcja zapisu do pliku
        mov     ebx, 1                  ; na stdout (ekran)
        mov     ecx, dane_kon
        mov     edx, dane_kon_dl
        int     80h

        mov     eax, 4                  ; funkcja zapisu do pliku
        mov     ebx, 1                  ; na stdout (ekran)
        mov     ecx, dane1
        mov     edx, dane1_dl
        int     80h                     ; wypisz pierwsze dane końcowe

        mov     eax, 4                  ; funkcja zapisu do pliku
        mov     ebx, 1                  ; na stdout (ekran)
        mov     ecx, dane2
        mov     edx, dane2_dl
        int     80h                     ; wypisz drugie dane końcowe

        mov     eax, 1
        xor     ebx, ebx
        int     80h



section .data
        ; FASM: segment readable writeable align 32

align   16                      ; dla SSE

dane            db      "ABCDEFGHIJKLMNOP", 10
dane_dl         equ     $ - dane

dane1           db      0, 0, 0, 0, 0, 0, 0, 0, 10, 9
dane1_dl        equ     $ - dane1

dane2           db      0, 0, 0, 0, 0, 0, 0, 0, 10
dane2_dl        equ     $ - dane2

dane_pocz db "Program demonstrujacy SSE. Dane na poczatku: ", 10, 9
dane_pocz_dl    equ     $ - dane_pocz

dane_kon        db      "Dane na koncu: ", 10, 9
dane_kon_dl     equ     $ - dane_kon
```

Po szczegółowy opis wszystkich instrukcji odsyłam, jak zwykle do Intela i AMD

Instrukcje typu SIMD wspomagają szybkie przetwarzanie multimediów: dźwięku, obrazu. Omówienie każdej instrukcji w detalu jest niemożliwe i niepotrzebne, gdyż szczegółowe opisy są zamieszczone w książkach Intela lub AMD.

Miłej zabawy.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia

1. Z dwóch zmiennych typu qword wczytaj do dwóch dowolnych rejestrów MMX (które najlepiej od razu skopiuj do innych), po czym wykonaj wszystkie możliwe dodawania i odejmowania. Wynik każdego zapisz w oddzielnej zmiennej typu qword.

2. Wykonaj operacje logiczne OR, AND i XOR na 64 bitach na raz (wczytaj je do rejestru MMX, wynik zapisz do pamięci).

3. Wczytajcie do rejestru MMX wartość szesnastkową 30 31 30 31 30 31 30 31, po czym wykonajcie różne operacje rozpakowania i pakowania, zapiszcie i wyświetlcie wynik jak każdy normalny ciąg znaków.

4. Wczytajcie do rejestrów XMM po 4 liczby ułamkowe dword, wykonajcie dodawania i odejmowania, po czym sprawdźcie wynik koprocesorem.

Jak pisać programy w języku asembler pod Linuksem?

Część 7 - Porty, czyli łączność między procesorem a innymi urządzeniami.

Nie zastanawialiście się kiedyś, jak procesor komunikuje się z tymi wszystkimi urządzeniami, które znajdują się w komputerze?
Teraz zajmiemy się właśnie sposobem, w jaki procesor uzyskuje dostęp do urządzeń zewnętrznych (zewnętrznych dla procesora, niekoniecznie tych znajdujących się poza obudową komputera).

Mimo że procesor może porozumiewać z urządzeniami przez wydzielone obszary RAM-u, to głównym sposobem komunikacji (gdy nie chcemy lub nie możemy używać sterowników) ciągle pozostają porty. Jeśli chcecie, możecie wykonać komendę cat /proc/ioports, która powie, które urządzenie zajmuje które porty.

Porty są to specjalne adresy, pod które procesor może wysyłać dane. Stanowią oddzielną strefę adresową (16-bitową, jak dalej zobaczymy, więc najwyższy teoretyczny numer portu wynosi 65535), choć czasami do niektórych portów można dostać się przez pamięć RAM. Są to porty mapowane do pamięci (memory-mapped), którymi nie będziemy się zajmować.

Lista przerwań Ralfa Brown'a (RBIL) zawiera plik ports.lst (który czasami trzeba osobno utworzyć - szczegóły w dokumentacji). W pliku tym znajdują się szczegóły dotyczące całkiem sporej liczby portów odpowiadającym różnym urządzeniom. I tak, mamy np.

- Kontrolery DMA
- Programowalny kontroler przerwań (Programmable Interrupt Controller, PIC)
- Programowalny czasomierz (Programmable Interval Timer, PIT)
- Kontroler klawiatury
- CMOS
- SoundBlaster i inne karty dźwiękowe
- Karty graficzne i inne karty rozszerzeń (np. modem)
- Porty COM, LPT
- Kontrolery dysków twardych
- i wiele, wiele innych...

No dobrze, wiemy co ma który port i tak dalej, ale jak z tego skorzystać?

Procesor posiada 2 instrukcje przeznaczone specjalnie do tego celu. Są to IN i OUT.
Ich podstawowa składnia wygląda tak:

```
in al/ax/eax, numer_portu
out numer_portu, al/ax/eax
```

Uwagi:

1. Jeśli numer_portu > 255, to w jego miejsce musimy użyć rejestru DX

2. Do operacji na portach nie można używać innych rejestrów niż AL, AX lub EAX.
3. Wczytane ilości bajtów zależą od rejestru, a ich pochodzenie - od rodzaju portu:
   - jeśli port `num` jest 8-bitowy, to
     `IN AL, num` wczyta 1 bajt z portu o numerze `num`
     `IN AX, num` wczyta 1 bajt z portu `num` (do AL) i 1 bajt z portu `num+1` (do AH)
     `IN EAX, num` wczyta po 1 bajcie z portów `num`, `num+1`, `num+2` i `num+3` i umieści w odpowiednich częściach rejestru EAX (od najmłodszej)
   - jeśli port `num` jest 16-bitowy, to
     `IN AX, num` wczyta 2 bajty z portu o numerze `num`
     `IN EAX, num` wczyta 2 bajty z portu o numerze `num` i 2 bajty z portu o numerze `num+1`
   - jeśli port `num` jest 32-bitowy, to
     `IN EAX, num` wczyta 4 bajty z portu o numerze `num`
4. Podobne uwagi mają zastosowane dla instrukcji `OUT`

Teraz byłaby dobra pora na jakiś przykład (mając na uwadze dobro swojego komputera, *NIE URUCHAMIAJ PONIŻSZYCH KOMEND*):

```
in      al, 0   ; pobierz bajt z portu 0
out     60h, eax; wyślij 4 bajty na port 60h

mov     dx, 300 ; 300 > 255, więc musimy użyć DX
in      al, dx  ; wczytaj 1 bajt z portu 300
out     dx, ax  ; wyślij 2 bajty na port 300
```

Nie rozpisywałem się tutaj za bardzo, bo ciekawsze i bardziej użyteczne przykłady znajdują się w moich mini-kursach (programowanie diód na klawiaturze, programowanie głośniczka).

Jak już wspomniałem wcześniej, porty umożliwiają dostęp do wielu urządzeń. Jeśli więc chcesz poeksperymentować, nie wybieraj portów zajętych np. przez kontrolery dysków twardych, gdyż zabawa portami może prowadzić do utraty danych lub uszkodzenia sprzętu.
Dlatego właśnie w nowszych systemach operacyjnych (tych pracujących w trybie chronionym, jak np. Linux) dostęp do portów jest zabroniony dla zwykłych aplikacji (o prawa dostępu do portów trzeba prosić system operacyjny - zaraz zobaczymy, jak to zrobić).
Jak więc działają np. stare DOS-owe gry? Odpowiedź jest prosta: nie działają w trybie chronionym. Windows uruchamia je w trybie udającym tryb rzeczywisty (taki, w jakim pracuje DOS), co umożliwia im pełną kontrolę nad sprzętem.
Wszystkie programy, które dotąd pisaliśmy też uruchamiają się w tym samym trybie, więc mają swobodę w dostępie np. do głośniczka czy karty dźwiękowej. Co innego programy pisane w nowszych kompilatorach np. języka C - tutaj może już być problem. Ale na szczęście my nie musimy się tym martwić...

Jeszcze jeden ciekawy przykład - używanie CMOSu. CMOS ma 2 podstawowe porty: 70h, zwany portem adresu i 71h, zwany portem danych. Operacje są proste i składają się z 2 kroków:

1. Na port 70h wyślij numer komórki (1 bajt), którą chcesz odczytać lub zmienić. Polecam plik cmos.lst z RBIL, zawierający szczegółowy opis komórek CMOS-u
2. Na port 71h wyślij dane, jeśli chcesz zmienić komórkę lub z portu 71h odczytaj dane, jeśli chcesz odczytać komórkę

Oto przykład. Odczytamy tutaj godzinę w komputerze, a konkretnie - sekundy:

```
        mov     eax, 101        ; funkcja systemowa "sys_ioperm":
        mov     ebx, 70h        ; poczynając od portu 70h
        mov     ecx, 20         ; tyle bajtów będziemy mogli wysłać/odebrać
        mov     edx, 71h        ; końcowy numer portu
        int     80h             ; niestety, musimy być root'em

        cmp     eax, 0          ; sprawdzamy, czy błąd. Nie wiem,
                                ; co ta funkcja ma
                                ; zwracać, ale ten sposób zdaje
                                ; się działać

        jl      koniec          ; jeśli wystąpił błąd, to zapis do
                                ; portów, do których nie mamy uprawnień,
                                ; zakończy się  "Segmentation fault"
                                ; ( "Naruszenie ochrony pamięci"  )
        mov     al, 0
        out     70h, al

                                ; ustaw przerwę na milion nanosekund, czyli
                                ; jedną milisekundę
        mov     dword [ts1+timespec.tv_sec], 0
        mov     dword [ts1+timespec.tv_nsec], 1000000
; w FASMie:
;       mov     dword [ts1.tv_sec], 0
;       mov     dword [ts1.tv_nsec], 1000000

        mov     eax, 162        ; sys_nanosleep
        mov     ebx, ts1        ; adres struktury timespec
        mov     ecx, 0          ; adres wynikowej struktury timespec
        int     80h             ; wykonaj przerwę w programie

        in      al, 71h

    koniec:
        ; ...

; w FASMie:
;       segment readable writeable
section .data

; w FASMie:
;struc timespec
;{
;       .tv_sec:                rd 1
;       .tv_nsec:               rd 1
;}
;
;ts1: timespec

struc timespec
                .tv_sec:        resd 1
                .tv_nsec:       resd 1
endstruc

ts1 istruc timespec
```

Wszystko jasne, oprócz bloku z wywołaniem sys_nanosleep. Po co to komu, pytacie?
Przy współczesnych częstotliwościach procesorów, CMOS (jak z resztą i inne układy) może po prostu nie zdążyć z odpowiedzią na naszą prośbę, gdyż od chwili wysłania numeru komórki do chwili odczytania danych mija za mało czasu. Dlatego robimy sobie przerwę na kilkanaście taktów zegara procesora.
Kiedyś między operacjami na CMOSie zwykło się pisać `jmp short $+2`, co też oczywiście nie robiło nic,

poza zajmowaniem czasu (to jest po prostu skok o 2 bajty do przodu od miejsca, gdzie zaczyna się ta 2-bajtowa instrukcja, czyli skok do następnej instrukcji), ale ta operacja już nie trwa wystarczająco długo, aby ją dalej stosować.

Komunikacja z urządzeniami nie zawsze jednak musi wymagać uprawnień administratora i korzystania z funkcji sys_ioperm. Sporo rzeczy (np. z klawiaturą) można zrobić, korzystając z funkcji sys_ioctl.

W dzisiejszych czasach porty już nie są tak często używane, jak były kiedyś. Jest to spowodowane przede wszystkim wspomnianym trybem chronionym oraz tym, że wszystkie urządzenia mają już własne sterowniki (mające większe uprawnienia do manipulowania sprzętem), które zajmują się wszystkimi operacjami I/O. Programista musi jedynie uruchomić odpowiednią funkcję i niczym się nie przejmować.
Dawniej, portów używało się do sterowania grafiką czy wysyłania dźwięków przez głośniczek lub karty dźwiękowe. Teraz tym wszystkim zajmuje się za nas system operacyjny. Dzięki temu możemy się uchronić przed zniszczeniem sprzętu.

Mimo iż rola portów już nie jest taka duża, zdecydowałem się je omówić, gdyż po prostu czasami mogą się przydać. I nie będziecie zdziwieni, gdy ktoś pokaże wam kod z jakimiś dziwnymi instrukcjami IN i OUT...

Szczegóły dotyczące instrukcji dostępu do portów także znajdziecie, jak zwykle, u AMD i Intela
Miłej zabawy.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia

1. Zapoznaj się z opisem CMOSu i napisz program, który wyświetli bieżący czas w postaci gg:mm:ss (z dwukropkami). Pamiętaj o umieszczeniu opóźnień w swoim programie i o uprawnieniach.

Część 8 - Zaawansowane programy, czyli zobaczmy, co ten język naprawdę potrafi.

No cóż, nie jesteśmy już amatorami i przyszła pora, aby przyjrzeć się temu, w czym asembler wprost błyszczy: algorytmy intensywne obliczeniowo. Specjalnie na potrzeby tego kursu napisałem następujący programik. Zaprezentuję w nim kilka sztuczek i pokażę, do jakich rozmiarów (tutaj: 2 instrukcje) można ścisnąć główną pętlę programu.
Oto ten programik:
(przeskocz program obliczający sumę liczb)

```
; Program liczący sumę liczb od 1 do liczby wpisanej z klawiatury
;
; Autor: Bogdan D.
;
; kompilacja:
;
; nasm -O999 -f elf ciag_ar.asm
; ld -s -o ciag_ar ciag_ar.o bibl/lib/libasmio.a
;
; fasm ciag_ar.asm ciag_ar.o
; ld -s -o ciag_ar ciag_ar.o bibl/lib/libasmio.a

; FASM:
; format ELF
; include "bibl/incl/linuxbsd/fasm/std_bibl.inc"
; section ".text" executable
; public _start

; NASM
%include "bibl/incl/linuxbsd/nasm/std_bibl.inc"
section .text
global _start



_start:
        pisz
        db      "Program liczy sume liczb od 1 do podanej liczby.",lf
        db      "Podaj liczbe calkowita: ",0

        we32e           ; pobieramy z klawiatury liczbę do rejestru EAX
        jnc     liczba_ok       ; flaga CF=1 oznacza błąd


blad:
        pisz
        db      lf, "Zla liczba!",lf,0

        wyjscie 1               ; mov ebx, 1 / mov eax, 1 / int 80h

liczba_ok:
        test    eax, eax        ; jeśli EAX=0, to też błąd
        jz      blad
```

```
        mov     ebx, eax        ; zachowaj liczbę. EBX=n
        xor     edx, edx        ; EDX = nasza suma
        mov     ecx, 1

petla:
        add     edx, eax        ; dodaj liczbę do sumy
        sub     eax, ecx        ; odejmij 1 od liczby
        jnz     petla           ; liczba różna od zera?
                                ; to jeszcze raz dodajemy

        pisz
        db      lf, "Wynik z sumowania 1+2+3+...+n= ",0

        mov     eax, edx        ; EAX = wynik
        pisz32e                 ; wypisz EAX

        mov     eax, ebx        ; przywrócenie liczby
        add     eax, 1          ; EAX = n+1
        mul     ebx             ; EDX:EAX = EAX*EBX = n*(n+1)

        shr     edx, 1
        rcr     eax, 1          ; EDX:EAX = EDX:EAX/2

        pisz
        db      lf, "Wynik ze wzoru: n(n+1)/2= ",0

        pisz64          ; wypisuje na ekranie 64-bitową liczbę całkowitą
                        ; z EDX:EAX


        nwln
        wyjscie 0
```

Jak widać, nie jest on ogromny, a jednak spełnia swoje zadanie. Teraz przeanalizujemy ten krótki programik:

- Komentarz nagłówkowy.
  Mówi, co program robi oraz kto jest jego autorem. Może zawierać informacje o wersji programu, o niestandardowym sposobie kompilacji/uruchomienia i wiele innych szczegółów.

- pisz, we32e, pisz32e oraz pisz64.
  To są makra uruchamiające procedury z mojej biblioteki. Używam ich, bo są sprawdzone i nie muszę ciągle umieszczać kodu tych procedur w programie.

- Makro wyjscie zawiera w sobie kod wyjścia z programu, napisany obok.

- `test rej, rej / jz ... / jnz ...`
  Instrukcja `TEST` jest szybsza niż `cmp rej, 0` i nie zmienia zawartości rejestru, w przeciwieństwie do `OR`. Jest to najszybszy sposób na sprawdzenie, wartość rejestru wynosi 0.

- Pętla główna.
  Jak widać, najpierw do sumy dodajemy n, potem n-1, potem n-2, i na końcu 1. Umożliwiło to znaczne skrócenie kodu pętli, a więc zwiększenie jej szybkości. Napisanie `sub eax, ecx` zamiast `sub eax, 1` skraca rozmiar instrukcji i powoduje jej przyspieszenie, gdyż dzięki temu w samej pętli procesor operuje już tylko na rejestrach.

- `shr edx, 1 / rcr eax, 1`
  Wynik musimy podzielić przez 2, zgodnie ze wzorem. Niestety, nie ma instrukcji `shr` dla 64 bitów. Więc trzeba ten brak jakoś obejść. Najpierw, `shr edx, 1` dzieli EDX przez 2, a bit 0 ląduje we fladze CF. Teraz, `rcr eax, 1` (rotate THROUGH CARRY) wartość CF (czyli stary bit 0 EDX) umieści w bicie 31 EAX. I o to chodziło!

Poniższy programik też napisałem dla tego kursu. Ma on pokazać złożone sposoby adresowania oraz instrukcje warunkowego przesunięcia (CMOV..):
(przeskocz program z macierzą)

```
; Program wczytuje od użytkownika macierz 3x3, po czym znajduje
; element największy i najmniejszy
;
; Autor: Bogdan D.
;
; kompilacja:
; nasm -O999 -f elf macierze.asm
; ld -s -o macierze macierze.o -Lbibl/lib -lasmio
;
; fasm macierze.asm macierze.o
; ld -s -o macierze macierze.o -Lbibl/lib -lasmio

; FASM:
; format ELF
; include "bibl/incl/linuxbsd/fasm/std_bibl.inc"
; section ".text" executable
; public _start
; rozmiar = 3

; NASM
%include "bibl/incl/linuxbsd/nasm/std_bibl.inc"
%define rozmiar 3
section .text
global _start


_start:

        pisz
        db      "Prosze podac wszystkie 9 elementow macierzy,"
        db      lf, "a ja znajde najwiekszy i najmniejszy.",lf,0

        xor     edx, edx                        ; ECX = 0
        mov     ebx, macierz

petla_wczyt:
        pisz
        db      "Prosze podac element numer ",0
        mov     eax, edx
        add     eax, 1
        pisz32e                         ; wypisz numer elementu

        push    ebx
        push    edx

        mov     eax, 4
```

```
        mov     ebx, 1
        mov     ecx, dwukspc
        mov     edx, 2
        int     80h                     ; wypisz dwukropek i spację

        pop     edx
        pop     ebx

        we32e                           ; wczytaj element
        jc      blad
        mov     [ebx+4*edx], eax        ; umieść w macierzy

        add     edx, 1                  ; zwiększ licznik elementów
                                        ; i równocześnie pozycję w macierzy

        cmp     edx, rozmiar*rozmiar
        jb      petla_wczyt

        jmp     wczyt_gotowe

blad:
        pisz
        db      lf, "Zla liczba!",lf,0

        wyjscie 1

wczyt_gotowe:
                                        ; EBP = max, EDI = min

        mov     ebp, [ebx]
        mov     edi, [ebx]              ; pierwszy element
        mov     edx, 1
        mov     eax, 1
        mov     esi, rozmiar*rozmiar

znajdz_max_min:
        mov     ecx, [ ebx + 4*edx ]
        cmp     ebp, ecx                ; EBP < macierz[edx] ?
        cmovb   ebp, ecx                ; jeśli tak, to EBP = macierz[edx]

        cmp     edi, ecx                ; EDI > macierz[edx] ?
        cmova   edi, ecx                ; jeśli tak, to EDI = macierz[edx]

        add     edx, eax
        cmp     edx, esi
        jb      znajdz_max_min

        pisz
        db      lf, "Najwiekszy element: ",0
        mov     eax, ebp
        pisz32e

        pisz
        db      lf, "Najmniejszy element: ",0
        mov     eax, edi
        pisz32e

        nwln
        wyjscie 0

; FASM: section ".data" writeable
section .data
```

```
macierz         times   rozmiar*rozmiar         dd 0
dwukspc         db ": "
```

Przypatrzmy się teraz miejscom, gdzie można zwątpić w swoje umiejętności:

- `mov [ebx+4*edx], eax`
  EBX = adres macierzy. EDX = 0, 1, 2, ..., rozmiar*rozmiar=9. Elementy macierzy mają rozmiar po 4 bajty każdy, stąd EDX mnożymy przez 4. Innymi słowy, pierwszy EAX idzie do [ebx+4*0]=[ebx], drugi do [ebx+4] (na 2 miejsce macierzy), trzeci do [ebx+8] itd.

- Fragment kodu:

```
mov     ecx, [ ebx + 4*edx ]
cmp     ebp, ecx        ; EBP < macierz[edx] ?
cmovb   ebp, ecx        ; jeśli tak, to EBP = macierz[edx]

cmp     edi, ecx        ; EDI > macierz[edx] ?
cmova   edi, ecx        ; jeśli tak, to EDI = macierz[edx]

add     edx, eax
cmp     edx, esi
jb      znajdz_max_min
```

  Najpierw, do ECX idzie aktualny element. Potem porównujemy EBX z tym elementem i, gdy EBP < ECX, kopiujemy ECX do EBP. Do tego właśnie służy instrukcja `CMOVB` (Conditional MOVe if Below). Instrukcje z rodziny `(F)CMOV` umożliwiają pozbywanie się skoków warunkowych, które obniżają wydajność kodu.
  Podobnie, porównujemy EDI=min z ECX.
  Potem, zwiększamy EDX o 1 i sprawdzamy, czy nie przeszliśmy przez każdy element macierzy.

Powyższy program trudno nazwać intensywnym obliczeniowo, bo ograniczyłem rozmiar macierzy do 3x3. Ale to był tylko przykład. Prawdziwe programy mogą operować na macierzach/tablicach zawierających miliony elementów. Podobny program napisany w HLLu jak C czy Pascal po prostu zaliczyłby się na śmierć.

Teraz pokażę program, który ewoluował od nieoptymalnej formy (zawierającej np. więcej skoków warunkowych w głównej pętli oraz inne nieoptymalne instrukcje) do czegoś takiego:
(przeskocz program znajdujący liczby magiczne)

```
; L_mag.asm
;
; Program wyszukuje liczby, które są sumą swoich dzielników
;
; Autor: Bogdan D.
; kontakt: bogdandr (at) op (dot) pl
;
; nasm -O999 -f elf l_mag.asm
; ld -s -o l_mag l_mag.o
;
; fasm l_mag.asm l_mag

; FASM:
; format ELF executable
; entry _start
; segment readable executable
```

```
; lf = 10

; NASM
%define lf 10           ; znak przejścia do nowej linii (Line Feed)
section .text
global _start


_start:
        mov     ebx,1           ; liczba początkowa

        mov     ebp,1

align 16
start2:
        mov     esi,ebx         ; ESI = liczba

        mov     ecx,ebp         ; EBP = 1
        shr     esi,1           ; zachowanie połowy liczby

        xor     edi,edi         ; suma dzielników=0

align 16
petla:
        xor     edx,edx         ; dla dzielenia
        nop
        cmp     ecx,esi         ; czy ECX (dzielnik)>liczba/2?
        mov     eax,ebx         ; przywrócenie liczby do dzielenia
        nop
        ja      dalej2          ; Jeśli ECX > ESI, to koniec
                                ; dzielenia tej liczby

        nop
        div     ecx             ; EAX = EDX:EAX / ECX, EDX=reszta

        nop
        nop
        add     ecx,ebp         ; zwiększamy dzielnik o 1
        nop

        test    edx,edx         ; czy ECX jest dzielnikiem?
                                ; (czy EDX=reszta=0?)
        nop
        nop
        jnz     petla           ; nie? - dzielimy przez następną liczbę

                                ; tak? -
        lea     edi,[edi+ecx-1] ; dodajemy dzielnik do sumy,
                                ; nie sprawdzamy na przepełnienie.
                                ; ECX-1 bo dodaliśmy EBP=1 do ECX po DIV.

        jmp     short petla     ; dzielimy przez kolejną liczbę
        ud2

align 16
dalej2:
        cmp     ebx,edi         ; czy to ta liczba?
                                ; (czy liczba=suma dzielników?)
        jne     nie             ; nie

        push    ebx
```

```
        mov     eax, 4
        mov     ebx, 1
        mov     ecx, jest
        mov     edx, jest_dlugosc
        int     80h             ; tak - napis "znaleziono"

        pop     ebx

        mov     eax,ebx
        call    pl              ; wypisujemy liczbę

align 16
nie:

        cmp     ebx,0ffffffffh  ; czy już koniec zakresu?
        nop
        je      koniec          ; tak

        add     ebx,ebp         ; nie, zwiększamy liczbę badana o 1
        nop
        jmp     start2          ; i idziemy od początku
        ud2

align 16
koniec:

        push    ebx

        mov     eax, 4
        mov     ebx, 1
        mov     ecx, meta
        mov     edx, meta_dlugosc
        int     80h

        pop     eax
        call    pl              ; wypisujemy ostatnią sprawdzoną liczbę

        mov     eax, 4
        mov     ebx, 1
        mov     ecx, nwln
        mov     edx, 1
        int     80h             ; wypisz znak nowej linii

        mov     eax, 1
        xor     ebx, ebx
        int     80h

        ud2

align 16
pc:                             ; wypisuje cyfrę w AL
        push    eax
        or      al, "0"
        mov     [cyfra], al

        push    ebx
        push    ecx
        push    edx

        mov     eax, 4
        mov     ebx, 1
        mov     ecx, cyfra
```

```
        mov     edx, 1
        int     80h

        pop     edx
        pop     ecx
        pop     ebx
        pop     eax

        ret
        ud2

align 16
pl:                     ; pisze liczbę dzisięciocyfrową w EAX
        mov     ecx,1000000000
        xor     edx,edx
        div     ecx
        call    pc

        mov     eax,edx
        mov     ecx,100000000
        xor     edx,edx
        div     ecx
        call    pc

        mov     eax,edx
        mov     ecx,10000000
        xor     edx,edx
        div     ecx
        call    pc

        mov     eax,edx
        mov     ecx,1000000
        xor     edx,edx
        div     ecx
        call    pc

        mov     eax,edx
        mov     ecx,100000
        xor     edx,edx
        div     ecx
        call    pc

        mov     eax,edx
        mov     ecx,10000
        xor     edx,edx
        div     ecx
        call    pc

        mov     eax,edx
        xor     edx,edx
        mov     ecx,1000
        div     ecx
        call    pc

        mov     eax,edx
        mov     cl,100
        div     cl
        mov     ch,ah
        call    pc

        mov     al,ch
        xor     ah,ah
```

```
        mov     cl,10
        div     cl
        mov     ch,ah
        call    pc

        mov     al,ch
        call    pc
        ret
        ud2


; FASM:  segment readable writeable
section .data align=4

jest    db      lf,"Znaleziono: "
jest_dlugosc    equ     $-jest  ; FASM: "=" zamiast "equ"

meta    db      lf,"Koniec. ostatnia liczba: "
meta_dlugosc    equ     $-meta  ; FASM: "=" zamiast "equ"

cyfra   db      0
nwln    db      lf
```

A oto analiza:

- Pętla główna:
  Dziel EBX przez kolejne przypuszczalne dzielniki. Jeśli trafisz na prawdziwy dzielnik (reszta=EDX=0), to dodaj go do sumy=EDI.
  Unikałem ustawiania obok siebie takich instrukcji, które zależą od siebie, jak np. `CMP / JA` czy `DIV / ADD`

- Nie za dużo tych `NOP`'ów?
  Nie. Zamiast czekać na wynik poprzednich instrukcji, procesor zajmuje się... robieniem niczego. Ale jednak sie zajmuje. Współczesne procesory potrafią wykonywać wiele niezależnych instrukcji praktycznie równolegle. Więc w czasie, jak procesor czeka na wykonanie poprzednich instrukcji, może równolegle wykonywać NOPy. Zwiększa to przepustowość, utrzymuje układy dekodujące w ciągłej pracy, kolejka instrukcji oczekujących na wykonanie nie jest pusta.

- Co robi instrukcja `lea edi,[edi+ecx-1]` ?
  LEA - Load Effective Address. Do rejestru EDI załaduj ADRES (elementu, którego) ADRES wynosi EDI+ECX-1. Czyli, w paskalowej składni: EDI := EDI+ECX-1. Do EDI dodajemy znaleziony dzielnik. Musimy odjąć 1, bo wcześniej (po dzieleniu) zwiększyliśmy ECX o 1.

- Co robi instrukcja `UD2` i czemu jest umieszczona po instrukcjach JMP ?
  Ta instrukcja (UnDefined opcode 2) wywołuje wyjątek wykonania nieprawidłowej instrukcji przez procesor. Umieściłem ją w takich miejscach, żeby nigdy nie była wykonana.
  Po co ona w ogóle jest w tym programie w takich miejscach?
  Ma ona interesującą właściwość: powstrzymuje jednostki dekodujące instrukcje od dalszej pracy. Po co dekodować instrukcje, które i tak nie będą wykonane (bo były po skoku bezwarunkowym)? Strata czasu.

- Po co ciągle `align 16` ?
  Te dyrektywy są tylko przed etykietami, które są celem skoku. Ustawianie kodu od adresu, który dzieli się przez 16 może ułatwić procesorowi umieszczenie go w całej jednej linii pamięci podręcznej (cache). Mniej instrukcji musi być pobieranych z pamięci (bo te, które są najczęściej wykonywane już

są w cache), więc szybkość dekodowania wzrasta. Układania kodu i danych zwiększa ogólną wydajność programu

O tych wszystkich sztuczkach, które tu zastosowałem, można przeczytać w podręcznikach dotyczących optymalizacji programów, wydanych zarówno przez Intel, jak i AMD (u AMD są też wymienione sztuczki, których można użyć do optymalizacji programów napisanych w języku C).
Podaję adresy (te same co zwykle): AMD, Intel

Życzę ciekawej lektury i miłej zabawy.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia

1. Napisz program obliczający Największy Wspólny Dzielnik i Najmniejszą Wspólną Wielokrotność dwóch liczb wiedząc, że:
   NWD(a,b) = NWD(b, reszta z dzielenia a przez b) i NWD(n,0)=n (algorytm Euklidesa)
   NWW(a,b) = a*b / NWD(a,b)

2. Napisz program rozkładający daną liczbę na czynniki pierwsze (liczba może być umieszczona w kodzie programu).

Część 9 - Narzędzia programisty, czyli co może nam pomagać w programowaniu.

Debugery

(przeskocz debugery)

Wszystkim się może zdarzyć, że nieustanne, wielogodzinne gapienie się w kod programu nic nie daje i program ciągle nie chce nam działać. Wtedy z pomocą przychodzą debugery. W tej części zaprezentuję kilka wartych uwagi programów tego typu.

Debugery programów Linuksowych:

1. GDB, czyli Gnu Debugger + nakładki, np. DDD
   Podstawowy debuger pracujący w trybie tekstowym (nakładka DDD - w graficznym). Składnia podstawowa to AT&T (odwrotna do zwykłej składni Intela), podobnie jak w Gnu as i GCC.
   Aby używać GDB, nasz program musimy skompilować *BEZ opcji* `-s` u linkera (aby zostały zachowane symbole).

   Krótki kurs obsługi:
   - Uruchomienie jest proste - wystarczy `gdb naszprog`.
   - Aby gdb stanął na wybranej funkcji, należy wpisać `break nazwa_funkcji`. To powinno ustawić pułapkę (breakpoint) na pierwszej instrukcji tej funkcji. Nie zawsze jednak breakpoint działa na procedurze początkowej _start - wtedy tuż po `_start:` (w naszym programie) należy wstawić instrukcję `nop` i tuż po niej postawić etykietę, na której już bez problemów ustawimy działający breakpoint.
   - Aby zdisasemblować konkretną funkcję, piszemy `disassemble nazwa_funkcji`. Zostanie wyświetlony kod podanej funkcji do najbliższej etykiety. Jeśli wolimy składnię Intela, piszemy `set disassembly-flavor intel`. Jeśli nie, to `set disassembly-flavor att`.
   - Aby wyświetlić rejestry, piszemy `info r`, aby wyświetlić konkretny rejestr, piszemy na przykład `print /x $eax`.
   - Aby zmienić wartość rejestru, piszemy na przykład `set $ebx=33`.
   - Aby wyświetlić rejestry koprocesora, piszemy `info float`.
   - Aby wyświetlić zawartość pamięci, piszemy na przykład `x 0x08048081` lub `print /x zmienna`. Aby wyświetlić więcej niż jedno 32-bitowe słowo, dajemy liczbę słów po ukośniku po komendzie `x`, na przykład `x/8 0x08048081`. Zamiast adresu można podać etykietę.
   - Aby zmienić wartość zmiennej w pamięci, używamy na przykład `set variable var1 = 0x1` lub `set variable *0x8049094 = 0x2`, jeśli znamy tylko adres.
   - Aby pobrać adres zmiennej, używamy na przykład `print /x &var1`.
   - Listę zdefiniowanych funkcji dostajemy po `info functions`.

- Bieżący stos wywołań można otrzymać komendą `info stack`.
- Aby przejść o 1 instrukcję dalej, piszemy `stepi`.
- Pomoc możemy wyświetlić, wpisując `help`.

2. Private ICE - PICE
   Ze zrzutów ekranowych na jego stronie domowej (pice.sf.net) wygląda całkiem obiecująco. Poza tym, jest to system-level debugger, czyli może on wnikać w zakamarki systemu. Szczegółów obsługi również niestety nie znam, gdyż kompilacja wymaga zabawy z kodem i posiadania źródeł jądra.

3. Valgrind
   Może nie do końca jest to debuger, ale narzędzie do analizy pamięci. Pozwala wykryć między innymi wycieki pamięci, miejsca spowalniające program oraz poprawić wydajność pamięci podręcznej.

Wiem, że nie wszyscy od razu z entuzjazmem rzucą się do ściągania i testowania przedstawionych wyżej programów i do debugowania własnych.
Niektórzy mogą uważać, że odpluskwiacz nie jest im potrzebny. Może i tak być, ale nie zawsze i nie u wszystkich. Czasem (zwykle po długim sterczeniu przed ekranem) przychodzi chęć do użycia czegoś, co tak bardzo może ułatwić nam wszystkim życie.
Pomyślcie, że gdyby nie było debugerów, znajdowanie błędów w programie musielibyśmy pozostawić naszej nie zawsze wyćwiczonej wyobraźni. Dlatego zachęcam Was do korzystania z programów tego typu (tylko tych posiadanych legalnie, oczywiście).

Warto jeszcze wspomnieć o dwóch programach: strace i ltrace. Pozwalają one na śledzenie, których funkcji systemowych i kiedy dany program używa. Jeśli coś Wam nie działa, można spojrzeć, na których wywołaniach funkcji są jakieś problemy. Uruchomienie jest proste: `strace ./waszprogram`

Edytory i disasemblery/hex-edytory

(przeskocz ten dział)

Do pisania programów w asemblerze wystarczy najzwyklejszy edytor tekstu (Emacs, VI, Joe, PICO, LPE, ...), ale jeśli nie podoba się Wam żaden z edytorów, to możecie wejść na stronę The Free Country.com - edytory, gdzie przedstawionych jest wiele edytorów dla programistów. Może znajdziecie coś dla siebie.
Zawsze można też przeszukać SourceForge.net

Kolejną przydatną rzeczą może okazać się disasembler lub hex-edytor. Jest to program, który podobnie jak debugger czyta plik i ewentualnie tłumaczy zawarte w nim bajty na instrukcje asemblera, jednak bez możliwości uruchomienia czytanego programu.
Disasemblery mogą być przydatne w wielu sytuacjach, np. gdy chcemy modyfikować pojedyncze bajty po kompilacji programu, zobaczyć adresy zmiennych, itp.
Oto 2 przykłady programów tego typu:

- Biew: biew.sf.net
- HTE: hte.sf.net

I ponownie, jeśli nie spodoba się Wam żaden z wymienionych, to możecie wejść na stronę The Free Country.com - disasemblery lub na SourceForge.net aby poszukać wśród pokazanych tam programów czegoś dla siebie.

Programy typu MAKE

Programy typu MAKE służą do automatyzacji budowania dużych i małych projektów. Taki program działa dość prosto: uruchamiamy go, a on szuka pliku o nazwie Makefile w bieżącym katalogu i wykonuje komendy w nim zawarte. Teraz zajmiemy się omówieniem podstaw składni pliku Makefile.

W pliku takim są zadania do wykonania. Nazwa zadania zaczyna się w pierwszej kolumnie, kończy dwukropkiem. Po dwukropku są podane nazwy zadań (lub plików) , od wykonania których zależy wykonanie tego zadania. W kolejnych wierszach są komendy służące do wykonania danego zadania.
UWAGA: komendy NIE MOGĄ zaczynać się od pierwszej kolumny! Należy je pisać je po jednym tabulatorze (ale nie wolno zamiast tabulatora stawiać ośmiu spacji).
Aby wykonać dane zadanie, wydajemy komendę `make nazwa_zadania`. Jeśli nie podamy nazwy zadania (co jest często spotykane), wykonywane jest zadanie o nazwie all (wszystko).

A teraz krótki przykład:
(przeskocz przykład)

```
all:    kompilacja linkowanie
        echo "Wszystko zakonczone pomyslnie"

kompilacja:
        nasm -O999 -f elf -o plik1.o plik1.asm
        nasm -O999 -f elf -o plik2.o plik2.asm
        nasm -O999 -f elf -o plik3.o plik3.asm

        fasm plik4.asm plik4.o
        fasm plik5.asm plik5.o
        fasm plik6.asm plik6.o

linkowanie:     plik1.o plik2.o plik3.o plik4.o plik5.o plik6.o
        ld -s -o wynik plik1.o plik2.o plik3.o plik4.o \
                 plik5.o plik6.o

help:
        echo "Wpisz make bez argumentow"
```

Ale MAKE jest mądrzejszy, niż może się to wydawać!
Mianowicie: jeśli stwierdzi, że wynik został stworzony PÓŹNIEJ niż pliki .o podane w linii zależności, to nie wykona bloku linkowanie, bo nie ma to sensu skoro program wynikowy i tak jest aktualny. MAKE robi tylko to, co trzeba. Oczywiście, niezależnie od wieku plików .o , dział kompilacja i tak zostanie wykonany (bo nie ma zależności, więc MAKE nie będzie sprawdzał wieku plików).

Znak odwrotnego ukośnika \ powoduje zrozumienie, że następna linia jest kontynuacją bieżącej, znak krzyżyka # powoduje traktowanie reszty linijki jako komentarza.

Jeśli w czasie wykonywanie któregokolwiek z poleceń w bloku wystąpi błąd (ściśle mówiąc, to gdy błąd zwróci wykonywane polecenie, jak u nas FASM czy NASM), to MAKE *natychmiast przerywa działanie* z informacją o błędzie i nie wykona żadnych dalszych poleceń (pamiętajcie więc o umieszczeniu w zmiennej środowiskowej PATH ścieżki do kompilatorów).

W powyższym pliku widać jeszcze jedno: zmiana nazwy któregoś z plików lub jakieś opcji sprawi, że trzeba ją będzie zmieniać wielokrotnie, w wielu miejscach pliku. Bardzo niewygodne w utrzymaniu, prawda?
Na szczęście z pomocą przychodzą nam ... zmienne, które możemy deklarować w Makefile i które zrozumie program MAKE.
Składnia deklaracji zmiennej jest wyjątkowo prosta i wygląda tak:

```
NAZWA_ZMIENNEJ = wartość
```

A użycie:

```
$(NAZWA_ZMIENNEJ)
```

Polecam nazwy zmiennych pisać wielkimi literami w celu odróżnienia ich od innych elementów. Pole wartości zmiennej może zawierać dowolny ciąg znaków.
Jeśli chcemy, aby treść polecenia NIE pojawiała się na ekranie, do nazwy tego polecenia dopisujemy z przodu znak małpki @, np.

```
@echo "Wszystko zakonczone pomyslnie"
```

Uzbrojeni w te informacje, przepisujemy nasz wcześniejszy Makefile:
(przeskocz drugi przykład)

```
# Mój pierwszy Makefile

FASM            = fasm  # ale można tu w przyszłości wpisać pełną ścieżkę

NASM            = nasm
NASM_OPCJE      = -O999 -f elf

LD              = ld
LD_OPCJE        = -s

PLIKI_O         = plik1.o plik2.o plik3.o plik4.o plik5.o plik6.o
PROGRAM         = wynik

all:    kompilacja linkowanie
        @echo "Wszystko zakonczone pomyslnie"

kompilacja:
        $(NASM) $(NASM_OPCJE) -o plik1.o plik1.asm
        $(NASM) $(NASM_OPCJE) -o plik2.o plik2.asm
        $(NASM) $(NASM_OPCJE) -o plik3.o plik3.asm

        $(FASM) plik4.asm plik4.o
        $(FASM) plik5.asm plik5.o
        $(FASM) plik6.asm plik6.o

linkowanie:     $(PLIKI_O)
        $(LD) $(LD_OPCJE) -o $(PROGRAM) $(PLIKI_O)

help:
        @echo "Wpisz make bez argumentow"
```

Oczywiście, w końcowym Makefile należy napisać takie regułki, które pozwolą na ewentualną kompilację pojedynczych plików, np.

```
plik1.o:        plik1.asm plik1.inc
```

```
	$(NASM) $(NASM_OPCJE) -o plik1.o plik1.asm
```

Choć na razie być może niepotrzebna, umiejętność pisania Makefile'ów może się przydać już przy projektach zawierających tylko kilka modułów (bo nikt nigdy nie pamięta, które pliki są aktualne, a które nie).
O tym, ile Makefile może zaoszczędzić czasu przekonałem się sam, pisząc swoją bibliotekę - kiedyś kompilowałem każdy moduł z osobna, teraz wydaję jedno jedyne polecenie `make` i wszystko się samo robi.
Makefile z biblioteki jest spakowany razem z nią i możecie go sobie zobaczyć.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Jak pisać programy w języku asembler pod Linuksem?

Część 10 - Nie jesteśmy sami, czyli jak łączyć asemblera z innymi językami.

Jak wiemy, w asemblerze można napisać wszystko. Jednak nie zawsze wszystko trzeba pisać w tym języku. W tej części pokażę, jak asemblera łączyć z innymi językami. Są na to 2 sposoby:

- Wstawki asemblerowe wpisywane bezpośrednio w kod programu
- Osobne moduły asemblerowe dołączane potem do modułów napisanych w innych językach

Postaram się z grubsza omówić te dwa sposoby na przykładzie języków Pascal, C i Fortran 77. Uprzedzam jednak, że moja znajomość języka Pascal i narzędzi związanych z tym językiem jest słaba.

Pascal

(przeskocz Pascal-a)

Wstawki asemblerowe realizuje się używając słowa asm. Oto przykład:

```
{ Linux używa składni AT&T do asemblera - jak zauważycie,
 argumenty instrukcji są odwrócone. }

program pas1;

begin
 asm movl $4,%eax
 end;
end.
```

Można też stosować nieco inny sposób - deklarowanie zmiennej reprezentującej rejestry procesora. Poniższy wycinek kodu prezentuje to właśnie podejście (wywołuje przerwanie 13h z AH=48h, DL=80h, DS:DX wskazującym na obiekt a):

```
uses crt,dos;

Var
   regs: Registers;

BEGIN
   clrscr();
   With regs DO
     Begin
       Ah:=$48;
       DL:=$80;
       DS:=seg(a);
       DX:=ofs(a);
     End;
```

```
Intr($13,regs);
```

Teraz zajmiemy się bardziej skomplikowaną sprawą - łączenie modułów napisanych w Pascal-u i asemblerze. Pascal dekoruje nazwy zmiennych i procedur, dorabiając znak podkreślenia z przodu. Jakby tego było mało, do nazwy procedury dopisywana jest informacja o jej parametrach. Tak więc z kodu

```
var
 c:integer;
 d:char;

procedure aaa(a:integer;b:char);
```

otrzymujemy symbole: _C, _D oraz _AAA$INTEGER$CHAR.

Oprócz tego, zwykle w Pascal-u argumenty na stos szły od lewej do prawej, ale z tego co widzę teraz, to Free Pascal Compiler działa odwrotnie - argumenty idą na stos wspak. W naszym przykładzie najpierw na stos pójdzie zmienna typu char, a potem typu integer (obie rozszerzone do rozmiaru DWORDa).
Jedno jest pewne: jeżeli twoja procedura jest uruchamiana z programu napisanego w Pascal-u, to ty sprzątasz po sobie stos - należy przy wyjściu z procedury wykonać `RET liczba`, gdzie liczba=rozmiar wszystkich parametrów włożonych na stos (wszystkie parametry są rozmiaru co najmniej DWORD).
Jeśli to ty uruchamiasz procedury napisane w Pascal-u, to nie musisz się martwić o zdejmowanie parametrów ze stosu.

Samo dołączanie modułów odbywa się na linii poleceń, najlepiej w tym celu użyć linkera (po uprzednim skompilowaniu innych modułów na pliki obiektowe).

C i C++

(przeskocz C i C++)

Wstawki asemblerowe zaczynają się słowami __asm (a kończą nawiasem zamykającym). W Linuksie wyglądają one nieco dziwnie i to nie tylko ze względu na odwrotną składnię AT&T:

```
__asm ("movl $4,%eax\n"
        "movb $0xff,%bl\n");
```

Jak widać, po każdej instrukcji trzeba dać znak przejścia do nowej linii (w jednej linii może być tylko 1 instrukcja asemblera). Można dorzucić też znak tabulacji \t.
Wygląd bloków __asm jest złożony. Po szczegóły odsyłam do stron przeznaczonych temu zagadnieniu. W szczególności, możecie poczytać podręcznik GCC (sekcje: 5.34 i 5.35), strony DJGPP oraz (w języku polskim) stronę pana Danileckiego.
U siebie też mam krótkie porównanie tych składni.

W C i C++ można, podobnie jak w Pascal-u, deklarować zmienne reprezentujące rejestry procesora. Plik nagłówkowy BIOS.H (niestety tylko w Windows) oferuje nam kilka możliwości. Oto przykład:

```
#include <bios.h>
...
```

```
REGS rejestry;
...
        rejestry.x.ax = 0x13;
        rejestry.h.bl = 0xFF;
        int86 (0x10, rejestry, rejestry);
```

Łączenie modułów jest prostsze niż w Pascal-u. Język zwykle C dekoruje nazwy, dodając znak podkreślenia z przodu, ale nie w Linuksie, gdzie po prostu nic nie jest dorabiane.
W Linuksie deklaracja funkcji zewnętrznej wygląda po prostu tak:

```
extern void naszafunkcja (int parametr, char* parametr2);
```

UWAGA - w języku C++ sprawy są trudniejsze nawet niż w Pascal-u. Dlatego, jeśli chcemy, aby nazwa naszej funkcji była niezmieniona (poza tym, że ewentualnie dodamy podkreślenie z przodu) i jednocześnie działała w C++, zawsze przy deklaracji funkcji w pliku nagłówkowym, należy dodać `extern "C"`, np.

```
#ifdef __cplusplus
extern "C" {
#endif

extern void naszafunkcja (int parametr, char* a);

#ifdef  __cplusplus
}
#endif
```

W systemach 32-bitowych parametry przekazywane są *OD PRAWEJ DO LEWEJ*, czyli pierwszy parametr (u nas powyżej: int) będzie włożony na stos jako ostatni, czyli będzie najpłycej, a ostatni (u nas: char*) będzie najgłębiej.
W systemach 64-bitowych sprawa wygląda trudniej: parametry, w zależności od klasy, są przekazywane (od LEWEJ do PRAWEJ):

- na stosie, jeśli ich rozmiar przekracza 8 bajtów lub zawiera pola niewyrównane co do adresu
- kolejno w rejestrach RDI, RSI, RDX, RCX, R8, R9, jeśli jest klasy całkowitej (mieści się w rejestrze ogólnego przeznaczenia)
- kolejno w rejestrach XMM0 ... XMM7 lub ich górnych częściach, jeśli jest klasy SSE lub SSEUP, odpowiednio
- w obszarze pamięci, jeśli jest klasy zmiennoprzecinkowej/zespolonej

W C/C++ to funkcja *uruchamiająca* zdejmuje włożone parametry ze stosu, a *NIE* funkcja uruchamiana.

Na systemach 32-bitowych parametry całkowitoliczbowe do 32 bitów zwracane są w rejestrze EAX (lub jego częściach: AL, AX, w zależności od rozmiaru), 64-bitowe w EDX:EAX, zmiennoprzecinkowe w ST0. Wskaźniki w 32-bitowych kompilatorach są 32-bitowe i są zwracane w EAX (w 16-bitowych zapewne w AX).
Struktury są wkładane na stos od ostatnich pól, a jeśli funkcja zwraca strukturę przez wartość, np.
`struct xxx f ( struct xxx a )`
to tak naprawdę jest traktowana jak taka funkcja:
`void f ( struct xxx *tu_bedzie_wynik, struct xxx a )`
czyli jako ostatni na stos wkładany jest adres struktury, do której ta funkcja ma włożyć strukturę wynikową.

Na systemach 64-bitowych sprawa ponownie wygląda inaczej. Tu także klasyfikuje się typ zwracanych danych, które są wtedy przekazywane:

- w pamięci, której adres przekazano w RDI (tak, jakby był to pierwszy parametr) - tak na przykład można zwracać struktury. Po powrocie, RAX będzie zawierał przekazany adres
- w kolejnym wolnym rejestrze z grupy RAX, RDX, jeśli klasa jest całkowita
- w kolejnym wolnym rejestrze z grupy XMM0, XMM1, jeśli klasa to SSE
- w górnej części ostatniego używanego rejestru SSE, jeśli klasa to SSEUP
- w ST0, jeśli klasa jest zmiennoprzecinkowa
- razem z poprzednią wartością w ST0, jeśli klasa to X87UP
- część rzeczywista w ST0, a część urojona w ST1, jeśli klasa jest zespolona

Polecam do przeczytania x64 ABI (np. dokument x64-abi.pdf, do znalezienia w Internecie).

Dołączanie modułów (te napisane w asemblerze muszą być uprzednio skompilowane) odbywa się na linii poleceń, z tym że tym razem możemy użyć samego kompilatora (GCC), aby wykonał za nas łączenie (nie musimy uruchamiać linkera LD).

Teraz krótki 32-bitowy przykładzik (użyję NASMa i GCC):

```
; NASM - casm1l.asm

; use32 nie jest potrzebne w Linuksie, ale też nie zaszkodzi
section .text use32

global  suma

suma:

; po wykonaniu push ebp i mov ebp, esp:
; w [ebp]    znajduje się stary EBP
; w [ebp+4]  znajduje się adres powrotny z procedury
; w [ebp+8]  znajduje się pierwszy parametr,
; w [ebp+12] znajduje się drugi parametr
; itd.

%idefine        a       [ebp+8]
%idefine        b       [ebp+12]

        push    ebp
        mov     ebp, esp

        mov     eax, a
        add     eax, b

; LEAVE = mov esp, ebp / pop ebp
        leave
        ret
```

I jeszcze plik casml.c:

```
#include <stdio.h>

extern int suma (int a, int b);

int c=1, d=2;

int main()
{
        printf("%d\n", suma(c,d));
        return 0;
```

```
}
```

Kompilacja wygląda tak:

```
nasm -f elf casm1l.asm
gcc -o casm casm1.c casm1l.o
```

Po uruchomieniu programu na ekranie pojawia się oczekiwana cyfra 3.

Może się zdarzyć też, że chcemy tylko korzystać z funkcji języka C, ale główną część programu chcemy napisać w asemblerze. Nic trudnego: używane funkcje deklarujemy jako zewnętrzne, ale *uwaga - swoją funkcję główną musimy nazwać main*. Jest tak dlatego, że teraz punkt startu programu nie jest w naszym kodzie, lecz w samej bibliotece języka C. Program zaczyna się między innymi ustawieniem tablic argumentów listy poleceń i zmiennych środowiska. Dopiero po tych operacjach biblioteka C uruchamia funkcję main instrukcją CALL.
Inną ważną sprawą jest to, że naszą funkcję główną powinniśmy zakończyć instrukcją RET (zamiast normalnych instrukcji wyjścia z programu), która pozwoli przekazać kontrolę z powrotem do biblioteki C, umożliwiając posprzątanie (np. wyrzucenie buforów z wyświetlonymi informacjami w końcu na ekran).
Krótki (także 32-bitowy) przykładzik:

```
section .text

global main

extern printf

main:

        ; printf("Liczba jeden to: %d\n", 1);
        push    dword 1         ; drugi argument
        push    dword napis     ; pierwszy argument
        call    printf          ; uruchomienie funkcji
        add     esp, 2*4        ; posprzątanie stosu

        ; return 0;
        xor     eax, eax
        ret                     ; wyjście z programu

section .data

napis: db "Liczba jeden to: %d", 10, 0
```

Kompilacja powinna odbyć się tak:

```
nasm -o casm2.o -f elf casm2.asm
gcc -o casm2 casm2.o
```

Jedna uwaga: funkcje biblioteki C mogą zamazać nam zawartość wszystkich rejestrów (poza EBX, EBP, ESI, EDI w systemach 32-bitowych, i RBX, RBP, R12, R13, R14, R15 na systemach 64-bitowych), więc *nie wolno nam polegać na zawartości rejestrów* po uruchomieniu jakiejkolwiek funkcji C.

Fortran 77

W tym języku nie wiem nic o wstawkach asemblerowych, więc przejdziemy od razu do łączenia modułów.
Fortran dekoruje nazwy, stawiając znak podkreślenia *PO* nazwie funkcji lub zmiennej (wyjątkiem jest funkcja główna - blok PROGRAM - która nazywa się `MAIN__`, z dwoma podkreśleniami).
Nie musimy pisać externów, ale jest kilka reguł przekazywania parametrów:

- parametry przekazywane są od prawej do lewej, czyli tak jak w C.
- jeśli to jest tylko możliwe, wszystkie parametry przekazywane są przez referencję, czyli przez wskaźnik. Gdy to jest niemożliwe, przekazywane są przez wartość.
- jeśli na liście parametrów pojawia się łańcuch znakowy, to na stosie przed innymi parametrami umieszczana jest jego długość.
- wyniki są zwracane w tych samych miejscach, co w języku C.

Na przykład, następujący kod:

```
        REAL FUNCTION aaa (a, b, c, i)

                CHARACTER a*(*)
                CHARACTER b*(*)
                REAL c
                INTEGER i

                aaa = c
        END

[...]
                CHARACTER x*8
                CHARACTER y*5
                REAL z,t
                INTEGER u

                t=aaa (x, y, z, u)
[...]
```

będzie przetłumaczony na asemblera tak (samo uruchomienie funkcji):

```
        push    5
        push    8
        push    u_      ; adres, czyli offset zmiennej "u"
        push    z_
        push    y_
        push    x_

        call    aaa_
```

(to niekoniecznie musi wyglądać tak ładnie, gdyż zmienne x, y, u i z są lokalne w funkcji MAIN__, czyli są na stosie, więc ich adresy mogą wyglądać jak [ebp-28h] lub podobnie).

Funkcja uruchamiająca sprząta stos po uruchomieniu (podobnie jak w C).
Dołączać moduły można bezpośrednio z linii poleceń (w każdym razie kompilatorem F77/G77).

Podam teraz przykład łączenia Fortrana 77 i asemblera (użyję NASMa i F77):

```
; NASM - asm1fl.asm

section .text use32

global  suma_

suma_:

; po wykonaniu push ebp i mov ebp, esp:
; w [ebp]    znajduje się stary EBP
; w [ebp+4]  znajduje się adres powrotny z procedury
; w [ebp+8]  znajduje się pierwszy parametr,
; w [ebp+12] znajduje się drugi parametr
; itd.

%idefine        a       [ebp+8]
%idefine        b       [ebp+12]

        push    ebp
        mov     ebp, esp

; przypominam, że nasze parametry są w rzeczywistości
; wskaźnikami do prawdziwych parametrów

        mov     edx, a          ; EDX = wskaźnik do 1-szego parametru
        mov     eax, [edx]      ; EAX = 1-szy parametr
        mov     edx, b
        add     eax, [edx]

; LEAVE = mov esp, ebp / pop ebp
        leave
        ret
```

I teraz plik asmfl.f:

```
PROGRAM funkcja_zewnetrzna

INTEGER a,b,suma

a=1
b=2

WRITE (*,*) suma(a,b)

END
```

Po skompilowaniu:

```
nasm -f elf asm1fl.asm
g77 -o asmfl asmfl.f asm1fl.o
```

i uruchomieniu, na ekranie ponownie pojawia się cyfra 3.

Informacji podanych w tym dokumencie *NIE* należy traktować jako uniwersalnych, jedynie słusznych reguł działających w każdej sytuacji. Aby uzyskać kompletne informacje, należy zapoznać się z dokumentacją posiadanego kompilatora.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia

1. Napisz plik asemblera, zawierający funkcję obliczania reszty z dzielenia dwóch liczb całkowitych. Następnie, połącz ten plik z programem napisanym w dowolnym innym języku (najlepiej w C/C++, gdyż jest najpopularniejszy) w taki sposób, by Twoją funkcję można było uruchamiać z tamtego programu. Jeśli planujesz łączyć asemblera z C, upewnij się że Twoja funkcja działa również z programami napisanymi w C++.

Jak pisać programy w języku asembler pod Linuksem?

Część 11 - Pamięć jest nietrwała, czyli jak posługiwać się plikami.

Jak wiemy, wszystkich danych nie zmieścimy w pamięci. A nawet jeśli zmieścimy, to pozostaną tam tylko do najbliższego wyłączenia prądu. Dlatego trzeba je zapisywać do pliku, a potem umieć je z tego pliku odczytać. W tej części zajmiemy się właśnie operacjami na plikach.

Do operowania na plikach posłużymy się kilkoma funkcjami przerwania 80h:

- EAX = 5 (sys_open) - otwarcie/utworzenie pliku.
  EBX = adres nazwy pliku (zakończonej bajtem zerowym).
  ECX = flagi (atrybuty) - 0=Tylko do odczytu, 1=Tylko do zapisu, 2=Odczyt i zapis, 0100h=Utwórz.
  EDX = tryb otwarcia (rozkład bitów jest taki sam, jak przy uprawnieniach do pliku, w kolejności: zapis, odczyt, uruchomienie dla właściciela, grupy i innych).
  W EAX funkcja zwraca deskryptor pliku.

- EAX = 8 (sys_creat) - utworzenie pliku.
  EBX = adres nazwy pliku (zakończonej bajtem zerowym).
  ECX = tryb utworzenia (bity takie same jak w EDX dla EAX=5).
  W EAX funkcja zwraca deskryptor pliku.

- EAX = 3 (sys_read) - odczyt z pliku.
  EBX = deskryptor (specjalny numerek) pliku.
  ECX = adres bufora, do którego będziemy czytać.
  EDX = ilość bajtów do odczytania.
  W EAX funkcja zwraca ilość odczytanych bajtów.

- EAX = 4 (sys_write) - zapis do pliku.
  EBX = deskryptor pliku.
  ECX = adres bufora, z którego będą pobierane dane do zapisu.
  EDX = ilość bajtów do zapisania. Jak zapewne sobie przypominacie, tej właśnie funkcji używaliśmy do wyświetlania napisów na ekranie, z EBX = 1 (1 = standardowe urządzenie wyjścia).
  W EAX funkcja zwraca ilość zapisanych bajtów.

- EAX = 6 (sys_close) - zamyka plik.
  EBX = deskryptor pliku.

- EAX = 19 (sys_lseek) - przechodzenie na określoną pozycję w pliku.
  EBX = deskryptor pliku.
  ECX = długość skoku (może być ujemna).
  EDX mówi, skąd wyruszamy: 0 - początek pliku, 1 - bieżąca pozycja w pliku, 2 - koniec pliku.
  Zwraca w EAX bieżącą pozycję w pliku.

- EAX = 10 (sys_unlink) - usuwa plik.
  EBX = adres nazwy pliku (zakończonej bajtem zerowym).

Błędy (podobnie jak w innych funkcjach Linuksowych) są zwykle sygnalizowane przez EAX < 0.
Po szczegóły odsyłam do mojego spisu funkcji systemowych, linuxasembly.org, www.lxhp.in-berlin.de/lhpsyscal.html oraz do stron manuala dotyczących poszczególnych funkcji, na przykład man 2 open.

Przykładowe użycie tych funkcji:
(przeskocz przykłady)
Utworzenie pliku i zapisanie czegoś do niego:

```
mov     eax, 8          ; numer funkcji - tworzenie pliku
mov     ebx, nazwa      ; adres nazwy pliku
mov     edx, 111111111b ; tryb otwierania - ósemkowo 777
int     80h

cmp     eax, 0
jl      blad            ; czy wystąpił błąd?

mov     ebx, eax        ; EBX = deskryptor pliku

mov     eax, 4          ; numer funkcji - zapis
                        ; EBX = deskryptor pliku
mov     ecx, bufor      ; adres bufora
mov     edx, 1024       ; ilość bajtów
int     80h

cmp     eax, 0
jl      blad            ; czy wystąpił błąd?

mov     eax, 6          ; numer funkcji - zamknij
                        ; EBX = deskryptor pliku
int     80h

cmp     eax, 0
jl      blad            ; czy wystąpił błąd?
```

Otwarcie istniejącego pliku, odczytanie i zapisanie czegoś do niego:

```
mov     eax, 5          ; numer funkcji - otwieranie pliku
mov     ebx, nazwa      ; adres nazwy pliku
mov     ecx, 2          ; zapis i odczyt
mov     edx, 111111111b ; tryb otwierania - ósemkowo 777
int     80h

cmp     eax, 0
jl      blad            ; czy wystąpił błąd?

mov     ebx, eax        ; EBX = deskryptor pliku

mov     eax, 3          ; numer funkcji - odczyt
                        ; EBX = deskryptor pliku
mov     ecx, bufor      ; adres bufora
mov     edx, 1024       ; ilość bajtów
int     80h

cmp     eax, 0
```

```
        jl      blad            ; czy wystąpił błąd?

        ; .... operacje na bajtach z pliku, na przykład
        xor     byte [bufor], 0ffh

        mov     eax, 4          ; numer funkcji - zapis
                                ; EBX = deskryptor pliku
        mov     ecx, bufor      ; adres bufora
        mov     edx, 1024       ; ilość bajtów
        int     80h

        ; Zauważcie, że zapisane bajty wylądowały po odczytanych, gdyż nie
        ; zmieniliśmy pozycji w pliku, a ostatnia operacja (odczyt) zostawiła
        ; ją tuż po odczytanych bajtach

        cmp     eax, 0
        jl      blad            ; czy wystąpił błąd?

        mov     eax, 6          ; numer funkcji - zamknij
                                ; EBX = deskryptor pliku
        int     80h

        cmp     eax, 0
        jl      blad            ; czy wystąpił błąd?
```

A teraz prawdziwy przykład. Będzie to nieco uszczuplona (pominąłem wczytywanie nazwy pliku) wersja mojego programu na_male.asm. Program ten zamienia wszystkie wielkie litery w podanym pliku na ich małe odpowiedniki. Reszta znaków pozostaje bez zmian. Jedna rzecz jest warta uwagi - nigdzie nie zmieniam rejestru EBX, więc ciągle w nim jest deskryptor pliku i nie muszę tego uchwytu zapisywać do pamięci. A teraz kod:
(przeskocz na_male.asm)

```
; Program zamienia wszystkie litery w podanym pliku z wielkich na male.
;
; Autor: Bogdan D.
; kontakt: bogdandr (at) op (dot) pl
;
; nasm -O999 -f elf na_male.asm
; ld -s -o na_male na_male.o

section .text

global _start

_start:
        mov     eax, 4
        mov     ebx, 1
        mov     ecx, info
        mov     edx, info_dl
        int     80h             ; wypisanie informacji o programie

        mov     eax, 5
        mov     ebx, plik
        mov     ecx, 2
        mov     edx, 111000000b ; 700 - zabroń innym dostępu
        int     80h

        cmp     eax, 0

        jnl     otw_ok
```

```
        call    plik_blad       ; uruchamiamy tę procedurę,
                                ; gdy wystąpił błąd

        jmp     zamk_ok         ; jeśli nie udało się nam nawet
                                ; otworzyć pliku, to od razu
                                ; wychodzimy z programu.


otw_ok:
        mov     ebx, eax        ; zapisujemy deskryptor pliku
        mov     ebp, 400h       ; EBP = rozmiar bufora.

czytaj:
        mov     eax, 3          ; funkcja czytania
                                ; EBX = deskryptor
        mov     ecx, bufor      ; adres bufora, dokąd czytamy
        mov     edx, ebp
        int     80h

czyt_ok:
        xor     edi, edi        ; EDI będzie wskaźnikiem do bufora.
                                ; Na początku go zerujemy.

        cmp     eax, edx        ; czy ilość bajtów odczytana (EAX) =
                                ; = ilość żądana (EDX) ?
        jne     przy_eof        ; jeśli nie, to plik się skończył

zamiana:
        mov     dl, [bufor+edi] ; wczytujemy znak z bufora do DL

        cmp     dl, "A"
        jb      znak_ok
        cmp     dl, "Z"
        ja      znak_ok

        or      dl, 20h         ; jeśli okazał się wielką literą,
                                ; zamieniamy go na małą
        mov     [bufor+edi],dl  ; i zapisujemy w miejsce,
                                ; gdzie poprzednio był

znak_ok:
        inc     edi             ; przechodzimy do innych znaków
        loop    zamiana         ; aż przejdziemy przez cały bufor
                                ; (CX = BP = 400h)

        mov     ecx, eax        ; ECX = ilość przeczytanych bajtów

        mov     eax, 19         ; funkcja przejścia do innej
                                ; pozycji w pliku
                                ; EBX = deskryptor
        neg     ecx             ; ECX = - ilość przeczytanych bajtów
        mov     edx, 1          ; wyruszamy z bieżącej pozycji
        int     80h

        cmp     eax, 0
        jnl     idz_ok
        call    plik_blad

idz_ok:                         ; po udanym przeskoku

        mov     eax, 4          ; funkcja zapisu do pliku
                                ; EBX = deskryptor
```

```
        mov     ecx, bufor
        mov     edx, ebp        ; EDX = EBP = 400h = długość bufora.
        int     80h

        cmp     eax, 0
        jg      czytaj          ; i idziemy czytać nową partię danych
                                ; (jeśli nie ma błędu)

        call    plik_blad

        jmp     zamk

przy_eof:                       ; gdy jesteśmy już przy końcu pliku.

;       xor     edi, edi        ; EDI już = 0 (zrobione wcześniej)

        mov     ebp, eax        ; EBP = ilość przeczytanych znaków
        mov     ecx, eax        ; ECX = ilość przeczytanych znaków

zamiana2:
        mov     dl, [bufor+edi] ; pobieramy znak z bufora do DL

        cmp     dl, "A"
        jb      znak_ok2
        cmp     dl, "Z"
        ja      znak_ok2

        or      dl,20h          ; jeśli okazał się wielką literą,
                                ; zamieniamy go na małą
        mov     [bufor+edi],dl  ; i zapisujemy w miejsce,
                                ; gdzie poprzednio był

znak_ok2:
        inc     edi             ; przechodzimy do innych znaków
        loop    zamiana2        ; aż przejdziemy przez cały bufor
                                ; (CX = BP = ilość bajtów)

        mov     ecx, eax        ; EDX = ilość przeczytanych bajtów

        mov     eax, 19         ; funkcja przejścia do innej
                                ; pozycji w pliku
                                ; EBX = deskryptor
        neg     ecx             ; ECX = - ilość przeczytanych bajtów
        mov     edx, 1          ; wyruszamy z bieżącej pozycji
        int     80h

        cmp     eax, 0
        jnl     idz_ok2
        call    plik_blad

idz_ok2:                        ; po udanym przeskoku

        mov     eax, 4          ; funkcja zapisu do pliku
                                ; EBX = deskryptor
        mov     ecx, bufor
        mov     edx, ebp        ; EDX=EBP=ilość przeczytanych bajtów
        int     80h

        cmp     eax, 0
        jnl     zamk            ; i zamykamy plik (jeśli nie ma błędu)

        call    plik_blad
```

```
zamk:
        mov     eax, 6          ; zamykamy plik
                                ; EBX = deskryptor
        int     80h


zamk_ok:
        mov     eax, 1
        xor     ebx, ebx
        int     80h


plik_blad:                      ; procedura wyświetla informację
                                ; o tym, że wystąpił błąd i
                                ; wypisuje numer tego błędu.
        push    eax
        push    ebx
        push    ecx
        push    edx
        push    ebx

        mov     eax, 4
        mov     ebx, 1
        mov     ecx, blad_plik
        mov     edx, blad_plik_dl
        int     80h             ; wypisanie informacji o tym,
                                ; że wystąpił błąd

        pop     ebx
        call    pl              ; wypisanie numeru błędu

        mov     eax, 4
        mov     ebx, 1
        mov     ecx, nwln
        mov     edx, 1
        int     80h             ; przejście do nowej linii

        pop     edx
        pop     ecx
        pop     ebx
        pop     eax

        ret

pl:

piszrej:

;we: ebx - rejestr do wypisania (hex)
;wy: rejestr, niszczone: eax

        mov     eax, ebx
        shr     eax, 28
        call    pc2
        mov     eax, ebx
        shr     eax, 24
        and     al, 0fh
        call    pc2
        mov     eax, ebx
        shr     eax, 20
        and     al, 0fh
```

```
        call    pc2
        mov     eax, ebx
        shr     eax, 16
        and     al, 0fh
        call    pc2
        mov     ax, bx
        shr     ax, 12
        and     al, 0fh
        call    pc2
        mov     ax, bx
        shr     ax, 8
        and     al, 0fh
        call    pc2
        mov     al, bl
        shr     al, 4
        and     al, 0fh
        call    pc2
        mov     al, bl
        and     al, 0fh
        call    pc2

        ret

pc2:
;we: AL - cyfra hex
;wy: pisze cyfrę, niszczone: nic

        push    eax
        push    ebx
        push    ecx
        push    edx

        cmp     al, 9
        ja      hex
        or      al, "0"
        jmp     short pz
hex:
        add     al, "A"-10

pz:
        mov     [cyfra], al
        mov     eax, 4
        mov     ebx, 1
        mov     ecx, cyfra
        mov     edx, 1
        int     80h

        pop     edx
        pop     ecx
        pop     ebx
        pop     eax

        ret

section .data

align 16
bufor           times 400h db 0         ; bufor wielkości 1 kilobajta
;plik           times 80 db 0
plik            db "aaa.txt",0          ; nazwa pliku

info    db "Program zamienia wielkie litery w pliku na male.",10
```

```
info_dl         equ     $-info

input1          db "Podaj nazwe pliku do przetworzenia: "
input1_dl       equ     $-input1

zla_nazwa       db 10, "Zla nazwa pliku."
zla_nazwa_dl    equ     $-zla_nazwa

blad_plik       db 10,"Blad operacji na pliku. Kod: "
blad_plik_dl    equ     $-blad_plik

cyfra           db 0
```

Ten program chyba nie był za trudny, prawda? Cała treść skupia się na odczytaniu paczki bajtów, ewentualnej ich podmianie i zapisaniu ich w to samo miejsce, gdzie były wcześniej.

Pliki są podstawowym sposobem przechowywania danych. Myślę więc, że się ze mną zgodzicie, iż opanowanie ich obsługi jest ważne i nie jest to aż tak trudne, jakby się mogło wydawać.

[Poprzednia część kursu] (Alt+3)
[Kolejna część kursu] (Alt+4)
[Spis treści off-line] (Alt+1)
[Spis treści on-line] (Alt+2)
[Ułatwienia dla niepełnosprawnych] (Alt+0)

Ćwiczenia

1. Napisz program, który wykona po kolei następujące czynności:
    1. Utworzy nowy plik
    2. Zapisze do niego 256 bajtów o wartościach od 00 do FF (nie musicie zapisywać po 1 bajcie)
    3. Zamknie ten plik
    4. Otworzy ponownie ten sam plik
    5. Zapisze odczytane bajty w nowej tablicy 256 słów w taki sposób:

        ```
        00 00 00 01 00 02 00 03 00 04 .... 00 FD 00 FE 00 FF
        ```

        czyli każdy oddzielony bajtem zerowym (należy przeczytać wszystkie bajty, po czym ręcznie je przenieść gdzie indziej i wzbogacić)
    6. Zamknie otwarty plik
    7. Usunie ten plik

Jak pisać programy w języku asembler pod Linuksem?

Część 12 - Czego od nas pragną, czyli linia poleceń programu. Zmienne środowiska.

Teraz zajmiemy się dość istotną sprawą z punktu widzenia programisty i użytkownika oprogramowania: linią poleceń. Nie wszyscy lubią podawać dane programowi w czasie jego pracy i odpowiadać na pytania o dane. Często (o ile jest to możliwe) można tego oszczędzić i zamiast bezustannie zadawać użytkownikowi pytania, przeczytać, co wpisano nam w linię poleceń. Umożliwia to pisanie programów, które raz uruchomione z prawidłową linią poleceń nie pytają już się o nic a tylko wykonują swoją pracę bez przeszkadzania użytkownikom.

Przejdźmy więc do szczegółów. Jeśli ktoś z Was zna język C, to na pewno wie, jak zadeklarować funkcję główną programu tak, aby mogła odczytać parametry i zmienne środowiska. Deklaracja taka wygląda zazwyczaj tak:

```
int main (int argc, char *argv[], char *env[])
```

gdzie:

- `argc` - liczba całkowita mówiąca o tym, z jaką ilością parametrów uruchomiono nasz program.
- `char *argv[]` - tablica wskaźników do poszczególnych parametrów. Tutaj, `argv[0]` - nazwa uruchomionego programu, `argv[1]` - pierwszy parametr programu itd.
- `char *env[]` - tablica wskaźników do zmiennych środowiskowych.

Ale gdzie są te zmienne?
Na stosie, oczywiście!
Po wykonaniu typowego prologu do funkcji (czyli `push ebp / mov ebp, esp`), zmienna argc znajduje się w [ebp+4], wskaźniki do parametrów linii poleceń zaczynają się od [ebp+8] i idą w górę stosu, po nich jest wskaźnik zerowy i dalej w górę są wskaźniki do zmiennych środowiska, też zakończone wskaźnikiem zerowym.

Wszystko ładnie wygląda w teorii, ale jak tego używać?
Aby odpowiedzieć na to pytanie, napisałem ten oto krótki programik. Jedynym celem jego życia jest wyświetlenie, z iloma argumentami go wywołano (co najmniej jeden - nazwa programu), wyświetlenie tych argumentów i zmiennych środowiska.

A teraz kod:
(przeskocz program wyświetlający linię poleceń)

```
; Program wyświetla własną linię poleceń i zmienne środowiskowe.
;
; Autor: Bogdan D.
; kontakt: bogdandr (at) op (dot) pl
;
; nasm -O999 -f elf liniap.asm
; ld -s -o liniap liniap.o bibl/lib/libasmio.a
```

```
;
; fasm liniap.asm liniap.o
; ld -s -o liniap liniap.o bibl/lib/libasmio.a

        ; przyda się nam moja biblioteczka
%include "bibl/incl/linuxbsd/nasm/std_bibl.inc"
section .text
global _start

; FASM:
; format ELF
; include "bibl/incl/linuxbsd/fasm/std_bibl.inc"
; section ".text" executable
; public _start


_start:
        push    ebp             ; typowy prolog, o którym wspomniałem
        mov     ebp, esp

%idefine        argc    ebp+4           ; ilość parametrów
%idefine        argv    ebp+8           ; parametry

; FASM:
;       argc    equ     ebp+4           ; ilość parametrów
;       argv    equ     ebp+8           ; parametry


        mov     eax, [argc]             ; EAX = ilość parametrów
        pisz32e                         ; wypisz EAX
        nwln                            ; przejdź do nowej linii

        xor     edi, edi                ; zerujemy licznik
                                        ; wyświetlonych parametrów

wypisz_argv:
        cmp     edi, eax                ; czy ilość wyświetlonych =
                                        ; = ilość parametrów?
        je      koniec_wypisz_argv      ; jeśli tak, to koniec
                                        ; wyświetlania parametrów

        mov     esi, [argv+edi*4]       ; pobierz parametr numer EDI.
                                        ; każdy wskaźnik jest 4-bajtowy,
                                        ; dlatego mnożymy EDI przez 4.

        pisz_esi                        ; wypisz napis pod adresem ESI
                                        ; czyli nasz parametr
        nwln                            ; przejdź do nowej linii

        add     edi, 1                  ; wybieramy kolejny parametr
        jmp     short   wypisz_argv     ; i idziemy pisać od nowa

koniec_wypisz_argv:
                        ; parametry sie skończyły. Teraz będzie jeden
                        ; wskaźnik zerowy i zmienne środowiska

        inc     edi                     ; przeskocz wskaźnik zerowy

wypisz_env:
        mov     esi, [argv+edi*4]       ; pobierz zmienną środowiskową

        test    esi, esi                ; sprawdź, czy nie wskaźnik zerowy
```

```
        jz      koniec_wypisz_env       ; jeśli zero, to skończyliśmy

        pisz_esi                        ; wypisz zmienną środowiskową
        nwln                            ; przejdź do nowej linii

        add     edi, 1                  ; przechodzimy na kolejna zmienną
        jmp     short   wypisz_env      ; i wypisujemy dalej


koniec_wypisz_env:
        wyjscie                         ; koniec...
```

Jak widać, nie było to aż takie trudne jak się mogło zdawać na początku. Właśnie poznaliście kolejną rzecz, która jest łatwa w użyciu, a możliwości której są duże. Teraz będziecie mogli śmiało zacząć pisać programy, których jedynym kanałem komunikacyjnym z użytkownikiem będzie linia poleceń, co znacznie uprości ich obsługę.

Tylko pamiętajcie o dodaniu kodu wyświetlającego sposób użycia programu, gdy nie podano mu żadnych parametrów.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia

1. Napisz program, który utworzy plik podany jako parametr. Jeśli podano drugi parametr (oddzielony od pierwszego spacją), zapisz jego wartość do tego pliku. Jeśli nie podano żadnych parametrów, niech program wypisze stosowną wiadomość.

2. Napisz program, który oblicza NWD (patrz część 8) dwóch liczb podanych na linii poleceń. Jeśli nie podano wystarczającej liczby parametrów, niech program wyświetli stosowną wiadomość.

Część 13 - Operacje na bitach, czyli to, w czym asembler błyszczy najbardziej.

W tej części poznamy ważną grupę instrukcji - operacje na bitach. Te właśnie instrukcje odróżniają asemblera od innych języków, gdzie rzadko pojawia się możliwość działania na tych najmniejszych jednostkach informacji (odpowiednie operatory istnieją w językach C i Pascal, ale inne języki, jak np. Fortran 77, są tego pozbawione).
Mimo iż o wszystkich instrukcjach opisanych w tej części już wspomniałem przy okazji omawiania podstawowych rozkazów procesora, to instrukcje bitowe są ważne i zasługują na oddzielny rozdział, poświęcony w całości tylko dla nich.

Zdawać by się mogło, że z takim jednym, maleńkim bitem niewiele da się zrobić: można go wyczyścić (wyzerować), ustawić (wstawić do niego 1) lub odwrócić jego bieżącą wartość. Ale te operacje mają duże zastosowania i dlatego ich poznanie jest niezbędne. Jeśli sobie przypomnicie, to używaliśmy już wielokrotnie takich instrukcji jak AND czy XOR. Teraz przyszedł czas, aby poznać je bliżej.

Instrukcja NOT

(przeskocz NOT)
Instrukcja NOT (logiczna negacja - to *NIE* jest to samo, co zmiana znaku liczby!) jest najprostszą z czterech podstawowych operacji logicznych i dlatego to od niej rozpocznę wstęp do instrukcji bitowych.
NOT jest instrukcją jednoargumentową, a jej działanie wygląda tak:

```
NOT 0 = 1
NOT 1 = 0
```

Używamy tej instrukcji wtedy, gdy chcemy naraz odwrócić wszystkie bity w zmiennej lub rejestrze. Na przykład, jeśli AX zawiera 0101 0011 0000 1111 (530Fh), to po wykonaniu NOT AX w rejestrze tym znajdzie się wartość 1010 1100 1111 0000 (ACF0h). Dodanie obu wartości powinno dać FFFFh.
NOT może mieć zastosowanie tam, gdzie wartość logiczna fałsz ma przyporządkowaną wartość zero, a prawda - wartość FFFFh, gdyż NOT w tym przypadku dokładnie przekłada prawdę na fałsz.

Instrukcja AND

(przeskocz AND)
Instrukcji AND (logicznej koniunkcji) najprościej używać do wyzerowania bitów. Tabelka działania AND wygląda tak:

```
0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1
```

No ale jakie to może mieć zastosowanie?
Powiedzmy teraz, że chcemy sprawdzić, czy bit numer 4 (numerację będę podawał od zera) rejestru AX jest równy 1, czy 0. Tutaj nie wystarczy proste porównanie CMP, gdyż reszta rejestru może zawierać nie wiadomo co. Z pomocą przychodzi nam właśnie instrukcja AND. Poniżej pseudo-przykład:

```
and     ax, 0000 0000 0001 0000b        ; (and ax, 16)
```

Teraz, jeśli bit numer 4 (odpowiadający wartości 2^4=16) był równy 1, to cały AX przyjmie wartość 16, jeśli zaś był równy zero, to cały AX będzie zerem. Na nasze szczęście, instrukcja AND ustawia odpowiednio flagi procesora, więc rozwiązaniem naszego problemiku będzie kod:

```
and     ax, 16
jz      bit_4_byl_zerem
;jnz    bit_4_nie_byl_zerem
```

A jakieś zastosowanie praktyczne?
Już podaję: zamiana małych liter na wielkie. W kodzie ASCII litery małe od wielkich różnią się tylko tym, że mają ustawiony bit numer 5. Tak więc po wykonaniu:

```
mov     al, "a"
and     al, 5fh         ; 5fh = 0101 1111 - czyścimy bit 5
                        ; (i 7 przy okazji)
```

w rejestrze AL będzie kod wielkiej litery A.
Inne zastosowanie znajdziecie w moim kursie programowania głośniczka:

```
in      al, 61h
and     al, not 3               ; zerujemy bity 0 i 1
                                ; NASM:  and al,~3
out     61h, al
```

W tym kodzie instrukcja AND posłużyła nam do wyczyszczenia bitów 0 i 1 (NOT 3 = NOT 0000 0011 = 1111 1100).

Jak zauważyliście, instrukcja AND niszczy zawartość rejestru, oprócz interesujących nas bitów. Jeśli zależy Wam na zachowaniu rejestru, użyjcie instrukcji TEST. Działa ona identycznie jak AND, ale nie zapisuje wyniku działania. Po co nam więc taka instrukcja? Otóż, wynik nie jest zapisywany, ale TEST ustawia dla nas flagi identycznie jak AND. Pierwszy kod przepisany z instrukcją TEST będzie wyglądał tak:

```
test    ax, 16
jz      bit_4_byl_zerem
;jnz    bit_4_nie_byl_zerem
```

Teraz nasz program będzie ciągle działać prawidłowo, ale tym razem zawartość rejestru AX została zachowana.
Jest jeszcze jedno ciekawe zastosowanie instrukcji TEST:

```
test    ax, ax
```

I co to ma niby robić? Wykonuje `AND AX, AX` , nigdzie nie zapisuje wyniku i tylko ustawia flagi.
No właśnie! Ustawia flagi, w tym flagę zera ZF. To, co widzicie powyżej to *najwydajniejszy* sposób na to, aby sprawdzić czy wartość rejestru nie jest zerem.

Instrukcja OR

(przeskocz OR)
Instrukcja OR (logiczna alternatywa) w prosty sposób służy do ustawiania bitów (wpisywania do nich 1).
Tabelka działania wygląda następująco:

```
0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1
```

Jeśli na przykład chcemy, aby 2 najmłodsze bity rejestru BX były się równe 1, a nie chcemy naruszać innych bitów (czyli MOV jest wykluczone), możemy to zrobić tak:

```
or      bx, 0000 0000 0000 0011         ; (or bx, 3)
```

Zastosowanie tego jest proste. Podam 2 przykłady. Pierwszy z nich jest wyjęty z mojej procedury wytwarzającej dźwięk w głośniczku (i kursu poświęconego temu zagadnieniu):

```
in      al, 61h
or      al, 3                           ; ustawiamy bity 0 i 1
out     61h, al
```

Przykład drugi jest odwróceniem operacji AND na znakach ASCII:

```
mov     al, "A"
or      al, 20h                 ; 20h = 0010 0000 - ustawiamy bit 5
```

teraz w AL powinien być kod małej literki a.
Instrukcja OR nie ma swojego odpowiednika, jakim jest TEST dla AND. Ale za to ma inne ciekawe zastosowanie - można nią sprawdzić, czy 2 rejestry naraz nie są zerami (to jest *najlepszy* sposób - bez żadnych CMP, JNZ/JZ itp.):

```
or      ax, bx
```

Podobnie, jak w instrukcji AND, flaga zera będzie ustawiona, gdy wynik operacji jest zerem - a to może się zdarzyć tylko wtedy, gdy AX i BX są *jednocześnie* zerami.
Zauważcie, że nie można do tego celu użyć instrukcji AND. Dlaczego? Podam przykład: niech AX=1 i BX = 8. AX i BX nie są oczywiście równe zero, ale:

```
        0000 0000 0000 0001     (=AX)
AND     0000 0000 0000 1000     (=BX)
        =
        0000 0000 0000 0000
```

Dlatego zawsze należy przemyśleć efekt działania instrukcji.

Instrukcja XOR

(przeskocz XOR)
Instrukcji XOR (eXclusive OR, logiczna alternatywa wykluczająca) używa się do zmiany stanu określonego bitu z 0 na 1 i odwrotnie.
Działanie XOR jest określone tak:

```
0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0
```

Zauważmy także, że dla dowolnych a i b mamy:
(a XOR b) XOR b = a
a XOR 0 = a
a XOR -1 = NOT a (-1 = FF w bajcie, FFFF w słowie i FFFFFFFF w dwordzie)
a XOR a = 0
Z tej ostatniej równości wynika natychmiast, że wyXORorwanie rejestru z samym sobą zawsze go wyzeruje. W ten sposób otrzymujemy jeden z dwóch *najwydajniejszych* sposobów na wyzerowanie rejestru:

```
xor     rej, rej
```

Drugi sposób to SUB rej,rej.

Teraz przykład: chcemy, aby wartość rejestru AX stała się równa 1 gdy rejestr był wyzerowany, a zerem, gdy była w tym rejestrze jedynka. Oto, jak możemy to zrobić:

```
        cmp     ax, 1
        je      wyzeruj
        mov     ax, 1
        jmp     koniec
wyzeruj:
        mov     ax, 0
koniec:
```

Ale wersja optymalna wygląda tak:

```
xor     ax, 1
```

gdyż mamy:

```
wartość AX:     0000 0000 0000 0001             0000 0000 0000 0000
        XOR     0000 0000 0000 0001             0000 0000 0000 0001
        =
nowy AX:        0000 0000 0000 0000             0000 0000 0000 0001
```

Jak widać, jest to o wiele prostsze i wydajniejsze rozwiązanie. Dlatego właśnie dobrze jest, gdy pozna się instrukcje logiczne.

Instrukcje przesuwania bitów

(przeskocz instrukcje przesuwania)
Instrukcje przesuwania bitów (shift) przemieszczają bity, nie zmieniając ich wzajemnego położenia (przesuwają grupowo). To wyjaśnienie może się wydawać bardzo pokrętne, ale spokojnie - zaraz wszystko się wyjaśni.
Na początek powiem, że jest kilka takich instrukcji (które też były podane w rozdziale o podstawowych instrukcjach procesora):

- SHL - shift left (shift logical left) = przesunięcie (logicznie) w lewo
- SAL - shift arithmetic left = przesunięcie (arytmetycznie) w lewo
- SHR - shift logical right = przesunięcie (logiczne) w prawo
- SAR - shift arithmetic right = przesunięcie (arytmetyczne)
- SHLD/SHRD = przesunięcia logiczne w lewo/prawo o podwójnej precyzji

Działanie każdej z tych instrukcji pokażę na przykładzie.
Niech na początku AX = 1010 0101 1010 0101 (A5A5h).

SHL i równoważna SAL działa tak (zakładając, że przesuwamy o jeden): najstarszy bit jest we fladze CF, każdy inny bit wchodzi na miejsce bitu starszego o 1, a do bitu zerowego wkładane jest zero.
Po wykonaniu SHL AX,3 wartość AX będzie więc wynosić 0010 1101 0010 1000 (2D28h), gdyż wszystkie bity przesunęliśmy o 3 miejsca w lewo, oraz CF=1 (bo jako ostatnia z rejestru wyleciała jedynka).

Instrukcja SHR działa w drugą stronę niż SHL: bit zerowy jest umieszczany we fladze CF, każdy inny bit wchodzi na miejsce bitu młodszego o 1, a do najstarszego bitu wkładane jest zero.
Dlatego teraz po wykonaniu SHR AX,1 w rejestrze AX będzie 0001 0110 1001 0100 (1694h), bo poprzednie bity AX przesunęliśmy o 1 miejsce w prawo, oraz CF=0.
SAR różni się od SHR nie tylko nazwą, ale też działaniem. Słowo arytmetyczne w nazwie NIE jest tu bez

znaczenia. Gdy SAR działa na liczbach ze znakiem, to zachowuje ich znak (bit7), tzn wykonuje to samo, co SHR, ale zamiast wkładać zero do najstarszego bitu, wstawia tam jego bieżącą wartość.
Z poprzedniego przykładu mamy, że AL = 94h = 1001 0100. Gdy teraz wykonamy SAR AL,2 to jako wynik otrzymamy 1110 0101 (E5h), bo wszystkie bity poszły o 2 miejsca w prawo o bit 7 został zachowany, i CF=0.

SHLD i SHRD wykonują to samo, co SHL i SHR ale na 2 rejestrach naraz (no niezupełnie). Na przykład wykonanie `SHLD EAX,EBX,3` spowoduje że 3 najstarsze bity EAX zostaną wyrzucone (i CF=ostatni z wyrzuconych) oraz 3 najstarsze bity EBX przejdą na nowo powstałe miejsca w 3 najmłodszych bitach EAX. Ale uwaga: EBX pozostaje *niezmieniony* ! I to jest właśnie przyczyna użycia słów no niezupełnie.

Ale nie sposób powiedzieć o SHL i SHR bez podania najbardziej popularnego zastosowania: szybkie mnożenie i dzielenie.
Jak można mnożyć i dzielić tylko przesuwając bity, pytacie?
Otóż, sprawa jest bardzo prosta. Wpiszcie do AX jedynkę i wykonajcie kilka razy SHL AX,1 za każdym razem sprawdzając zawartość AX. Jak zauważycie, w AX będą kolejno 1,2,4,8,16,... Czyli za każdym razem zawartość AX się podwaja.
Ogólnie, `SHL rej,n` mnoży zawartość rejestru przez 2^n. Na przykład `SHL AX,4` przemnoży AX przez 2^4 = 16.
Ale co zrobić, gdy chcemy mnożyć przez coś innego niż 2^n?
Odpowiedź jest równie prosta, np. AX * 10 = (AX*8) + (AX*2) - z tym się chyba zgodzicie. A od tego już tylko 1 krok do

```
mov     bx, ax
shl     ax, 3           ; AX = AX*8
shl     bx, 1           ; BX = BX*2 = AX*2
add     ax, bx          ; AX = AX*10
```

Ale niekoniecznie musimy dodawać wyniki. Zauważcie, że AX * 15 = (AX*8) + (AX*4) + (AX*2) + AX. Trzeba byłoby wykonać 3 SHL i 3 ADD. Ale my skorzystamy z innego rozwiązania: AX * 15 = (AX*16) - AX. Już tylko 1 SHL i 1 SUB. Stąd mamy:

```
mov     bx, ax
shl     ax, 4           ; AX = AX*16
sub     ax, bx
```

Dokładnie w ten sam sposób działa dzielenie (tylko oczywiście przy dzieleniu używamy SHR/SAR i niestety szybko możemy dzielić tylko przez potęgi dwójki). Pilnujcie tylko, aby używać tej właściwej instrukcji! Jak wiemy, 65534 = 0FFFEh = -2 . Teraz, oczywiście FFFE SHR 1 = 7FFFh = 32767 (=65534/2) a FFFE SAR 1 = FFFF = -1 (= -2/2). Widać różnicę, prawda? Pamiętajcie, że SAR patrzy na znak i go zachowuje.

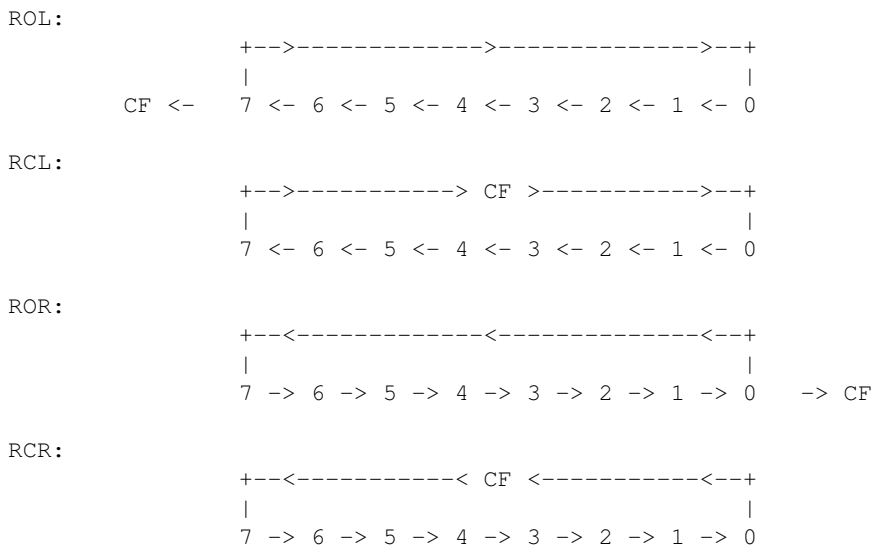
Używanie SHL dla mnożenia i (zwłaszcza) SHR dla dzielenia może znacznie przyśpieszyć nasze programy, gdyż instrukcje MUL i DIV są dość wolne.

Instrukcje rotacji bitów

(przeskocz instrukcje rotacji)
Teraz przedstawię kolejną grupę instrukcji bitowych - instrukcje rotacji bitów. W tej grupie są tylko 4 instrukcje:

- ROL - rotate left = obrót w lewo.
  Ta instrukcja robi tyle, co SHL, lecz zamiast do bitu zerowego wkładać zero, wkłada tam bieżącą wartość najstarszego bitu (przy okazji zachowując go także we fladze CF).
  bit7 = bit6, ... , bit1 = bit0, bit0 = stary bit7
- RCL - rotate through carry left = obrót w lewo z użyciem flagi CF. Ta instrukcja jest podobna do ROL z jedną różnicą: wartość wstawiana do najmłodszego bitu jest brana z flagi CF, a nie od razu z najstarszego bitu. Po wzięciu bieżącej wartości CF, najstarszy bit jest do niej zapisywany.
  carry flag CF = bit7, bit7 = bit6, ... , bit1 = bit0, bit0 = stara CF
- ROR - rotate right = obrót w prawo. Ta instrukcja robi tyle, co SHR, lecz zamiast do najstarszego bitu wkładać zero, wkłada tam bieżącą wartość najmłodszego bitu (przy okazji zachowując go także we fladze CF).
  bit0 = bit1, ... , bit6 = bit7, bit7 = stary bit0
- RCR - rotate through carry right = obrót w prawo z użyciem flagi CF. Ta instrukcja jest podobna do ROR z jedną różnicą: wartość wstawiana do najstarszego bitu jest brana z flagi CF, a nie od razu z najmłodszego bitu. Po wzięciu bieżącej wartości CF, najmłodszy bit jest do niej zapisywany.
  CF = bit0, bit0 = bit1, ... , bit6 = bit7, bit7 = stara CF

Schematyczne działanie tych instrukcji na bajtach widać na tych rysunkach:
(przeskocz rysunki)



W przypadku ROL i ROR, to *ostatni* wyjęty z jednej strony a włożony z drugiej strony bit zostaje też zapisany do flagi CF.
RCR i RCL działają tak, że bit, który ma zostać wstawiony, jest pobierany z CF, a wypchnięty bit ląduje w CF, a nie od razu na nowym miejscu.

No to kilka przykładów:

```
0011 1100  ROL  2  =  1111 0000  (tak samo jak SHL)
0011 1100  ROL  3  =  1110 0001

1111 0000  ROR  1  =  0111 1000  (tak samo jak SHR)
1010 0011  ROR  5  =  0001 1101
```

Zastosowanie tych instrukcji znalazłem jedno: generowanie chaosu w rejestrach...
Po co to mi? Na przykład generatory liczb pseudo-losowych z mojej biblioteki korzystają z tych właśnie instrukcji (a także z kilku poprzednich, np. XOR).

Instrukcje testowania i szukania bitów

(przeskocz instrukcje BT*)
Ostatnia już grupa rozkazów procesora to instrukcje testowania i szukania bitów. W tej grupie znajdują się:

- BT - Bit Test
- BTC - Bit Test and Complement
- BTR - Bit Test and Reset
- BTS - Bit Test and Set
- BSF - Bit Scan Forward
- BSR - Bit Scan Reverse

Teraz po kolei omówię działanie każdej z nich.
Instrukcje BT* przyjmują 2 argumenty: miejsce, gdzie mają znaleźć dany bit i numer tego bitu, a zwracają wartość tego bitu we fladze CF. Ponadto, BTS ustawia znaleziony bit na 1, BTR czyści znaleziony bit a BTC odwraca znaleziony bit.
Kilka przykładów:

```
bt      eax, 21         ; umieść 21. bit EAX w CF
jc      bit_jest_1
...
bts     cl, 2           ; umieść 2. bit CL w CF i ustaw go
jnc     bit_2_byl_zerem
...
btc     dh, 5           ; umieść 5. bit DH w CF i odwróć go
jc      bit_5_byl_jeden
```

Instrukcje Bit Scan przyjmują 2 argumenty: pierwszy z nich to rejestr, w którym będzie umieszczona pozycja (numer od zera począwszy) pierwszego bitu, którego wartość jest równa 1 znalezionego w drugim argumencie instrukcji. Dodatkowo, BSF szuka tego pierwszego bitu zaczynając od bitu numer 0, a BSR od najstarszego (numer 7, 15 lub 31 w zależnosci od rozmiaru drugiego argumentu).

Teraz szybki przykładzik:

```
mov     ax, 1010000b
bsf     bx, ax
bsr     cx, ax
```

Po wykonaniu powyższych instrukcji w BX powinno być 4, a w CX - 6 (bity liczymy od zera).

Jak pewnie zauważyliście, w kilku miejscach w tym tekście wyraźnie podkreśliłem słowa najwydajniejszy i im podobne. Chciałem w ten sposób uzmysłowić Wam, że operacje logiczne / binarne są bardzo ważną grupą instrukcji. Używanie ich, najlepiej wraz z instrukcją LEA służącą do szybkich rachunków, może kilkakrotnie (lub nawet kilkunastokrotnie) przyśpieszyć najważniejsze części Waszych programów (np. intensywne obliczeniowo pętle o milionach powtórzeń - patrz np. program L_mag.asm z 8. części tego kursu).

Dlatego zachęcam Was do dobrego opanowania instrukcji binarnych - po prostu umożliwia to pisanie programów o takiej wydajności, o której inni mogą tylko pomarzyć...

Po szczegółowy opis wszystkich instrukcji odsyłam, jak zwykle do : Intela i AMD

Ciekawe operacje na bitach (w języku C)

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia

1. W jednej komendzie policz:
    1. iloraz z dzielenia EDI przez 4
    2. resztę z dzielenia EDI przez 4
    3. największą liczbę mniejszą lub równą EDI dzielącą sie przez 4

   Wskazówka: 4 = 2^2 oraz możliwe reszty z dzielenia przez 4 to 0, 1, 2 i 3 i zajmują one co najwyżej 2 bity.

2. W jednej komendzie:
    1. ustaw bity 0, 11, 4 i 7 rejestru CX, nie ruszając pozostałych
    2. wyczyść bity 9, 2, 7 i 25 rejestru ESI, nie ruszając pozostałych
    3. przełącz (zmień wartość na odwrotną) bity 16, 4, 21, 1 i 10 rejestru EAX, nie ruszając pozostałych
    4. spraw, by wartość rejestru AL=18h zmieniła się na 80h, bez instrukcji MOV
    5. spraw, by wartość rejestru AL=18h zmieniła się na 81h, bez instrukcji MOV
    6. przełącz bit 23 rejestru EDX nie ruszając pozostałych, a jego starą wartość umieść we fladze CF

Część 14 - Wielokrotna precyzja, czyli co robić, gdy dane nie mieszczą się w rejestrach.

Czasami w naszych programach zachodzi potrzeba, aby posługiwać się np. liczbami przekraczającymi 4 czy nawet 8 bajtów, a my mamy tylko rejestry 32-bitowe (lub czasem 16-bitowe).
Co wtedy zrobić?
Odpowiedzi na to właśnie pytanie postaram się udzielić w tej części kursu.

Do naszych celów posłuży coś, co się nazywa arytmetyką wielokrotnej precyzji (ang. Multiprecision Arithmetic). Generalną zasadą będzie zajmowanie się obliczeniami po kawałku (bo z resztą inaczej się nie da) i zapamiętywanie, czy z poprzedniego kawałka wynieśliśmy coś w pamięci (do tego celu w prosty sposób wykorzystamy flagę CF, która wskazuje właśnie, czy nie nastąpiło przepełnienie).
Najpierw kilka ustaleń:

1. Będę tutaj używał rejestrów 32-bitowych, ale w razie potrzeby dokładnie te same algorytmy działają także dla rejestrów innych rozmiarów.
2. Zmienne arg1 i arg2 mają po 16 bajtów (128 bitów) każda. Na potrzeby nauki wystarczy w sam raz.
3. Zmienna wynik ma tyle samo bajtów, co arg1 i arg2, z wyjątkiem mnożenia, gdzie oczywiście musi być dwa razy większa.
4. Zmienna wynik na początku zawiera zero.
5. Kod nie zawsze będzie optymalny, ale chodzi mi o to, aby był jak najbardziej jasny i przejrzysty.

A więc do dzieła.

Dodawanie

(przeskocz dodawanie)
Dodawanie, podobnie jak uczyli nas w szkole, zaczynamy od najmłodszych cyfr (cyfr jedności) - tyle że zamiast pojedynczych cyferek będziemy dodawać całe 32-bitowe kawałki naraz. Flaga CF powie nam, czy z poprzedniego dodawania wynosimy coś w pamięci (nawet z dodawania dużych liczb wyniesiemy co najwyżej 1 bit w pamięci). To coś trzeba oczywiście dodać potem do wyższej części wyniku.
No to dodajemy:
(przeskocz program do dodawania)

```
mov     eax, [arg1]
add     eax, [arg2]     ; dodajemy 2 pierwsze części liczb
mov     [wynik], eax    ; i ich sumę zapisujemy w pierwszej
                        ; części wyniku. Flaga CF mówi, czy
                        ; wynosimy coś w pamięci

mov     eax, [arg1+4]
adc     eax, [arg2+4]   ; dodajemy drugie części + to,
                        ; co wyszło z poprzedniego dodawania
                        ; [arg1] i [arg2] (a to jest w fladze
                        ; CF, stąd instrukcja ADC zamiast ADD)
```

```
mov     [wynik+4], eax  ; całość:[arg1+4]+[arg2+4]+"w pamięci"
                        ; z pierwszego dodawania zapisujemy tu
                        ; Flaga CF zawiera (lub nie) bit
                        ; "w pamięci", ale tym razem z ADC

                        ; podobnie reszta działania:
mov     eax, [arg1+8]
adc     eax, [arg2+8]
mov     [wynik+8], eax

mov     eax, [arg1+12]
adc     eax, [arg2+12]
mov     [wynik+12], eax

jc      blad_przepelnienie
```

Odejmowanie

(przeskocz odejmowanie)
W szkole uczyli nas, że zaczynamy od najmłodszych cyfr i ewentualnie pożyczamy od starszych. Tutaj będziemy robić dokładnie tak samo! Wymaga to jednak poznania nowej instrukcji - SBB (Subtract with Borrow). Działa ona tak samo, jak zwykła instrukcja odejmowania SUB, ale dodatkowo odejmuje wartość flagi CF, czyli 1 lub 0, w zależności od tego, czy w poprzednim kroku musieliśmy pożyczać czy też nie. Ewentualną pożyczkę trzeba oczywiście odjąć od wyższej części wyniku.
Piszmy więc (od arg1 będziemy odejmować arg2):
(przeskocz program do odejmowania)

```
mov     eax, [arg1]
sub     eax, [arg2]             ; odejmujemy 2 pierwsze części
mov     [wynik], eax            ; i zapisujemy wynik
                                ; flaga CF mówi, czy była pożyczka

mov     eax, [arg1+4]
sbb     eax, [arg2+4]           ; odejmujemy razem z pożyczką (CF),
                                ; jeśli w poprzednim odejmowaniu
                                ; musieliśmy coś pożyczać

mov     [wynik+4], eax          ; wynik: [arg1+4]-[arg2+4]-pożyczka
                                ; z pierwszego odejmowania
                                ; CF teraz zawiera pożyczkę z SBB

                                ; podobnie reszta działania:
mov     eax, [arg1+8]
sbb     eax, [arg2+8]
mov     [wynik+8], eax

mov     eax, [arg1+12]
sbb     eax, [arg2+12]
mov     [wynik+12], eax

jc      arg1_mniejszy_od_arg2
```

Zmiana znaku liczby

(przeskocz NEG)
Teraz zajmiemy się negacją (zmianą znaku liczby). Ta operacja jest o tyle dziwna, że wykonujemy ją od góry (od najstarszych bajtów) i po negacji niższych trzeba zadbać o pożyczkę we wszystkich wyższych częściach. Popatrzcie (będziemy negować arg1):
(przeskocz program do negacji)

```
neg     dword [arg1+12]         ; negujemy najstarszą część

neg     dword [arg1+8]          ; negujemy drugą od góry
sbb     dword [arg1+12], 0      ; jeśli była pożyczka od starszej
                                ; (a prawie zawsze będzie), to tę
                                ; pożyczkę odejmujemy od starszej

neg     dword [arg1+4]          ; negujemy kolejną część i odejmujemy
                                ; pożyczki od starszych części
sbb     dword [arg1+8], 0
sbb     dword [arg1+12], 0

neg     dword [arg1]            ; negujemy kolejną część i odejmujemy
                                ; pożyczki od starszych części
sbb     dword [arg1+4], 0
sbb     dword [arg1+8], 0
sbb     dword [arg1+12], 0
```

Dla większych liczb nie wygląda to za ciekawie. Dlatego najprostszym sposobem będzie po prostu odjęcie danej liczby od zera, do czego zastosujemy poprzedni algorytm odejmowania.

Mnożenie

(przeskocz mnożenie)
Mnożenie jest nieco bardziej skomplikowane, ale ciągle robione tak jak w szkole, czyli od prawej. Ustalmy dla wygody, że arg1 zawiera ABCD, a arg2 - PQRS (każda z liter oznacza 32 bajty). Ogólny schemat wygląda teraz tak:
przeskocz schemat mnożenia

```
  A  B  C  D
* P  Q  R  S
=
            D*S
         C*S
      B*S
   A*S
         D*R
      C*R
   B*R
A*R
      D*Q
   C*Q
```

```
                        B*Q
                      A*Q
                           D*P
                        C*P
                      B*P
                  + A*P
              =
                     F  G  H  I  J  K  L
```

```
        [wynik]    = L = D*S
        [wynik+4]  = K = C*S + D*R
        [wynik+8]  = J = B*S + C*R + D*Q
        [wynik+12] = I = A*S + B*R + C*Q + D*P
        [wynik+16] = H = A*R + B*Q + C*P
        [wynik+20] = G = A*Q + B*P
        [wynik+24] = F = A*P
        (rzecz jasna, każdy iloczyn zajmuje 2 razy po 4 bajty, np. L zajmuje
         [wynik] i częściowo [wynik+4], ale tutaj podałem tylko miejsca,
         gdzie pójdą najmłodsze części każdego w iloczynów)
```

Obliczenia wyglądałyby tak (pamiętamy, że wynik operacji MUL jest w EDX:EAX):
(przeskocz program mnożenia)

```
        ; przez rozpoczęciem procedury zmienna "wynik" musi być wyzerowana!
;[wynik] = L = D*S

        mov  eax, dword [arg1]          ; EAX = D
        mul  dword [arg2]               ; EDX:EAX = D*S
        mov  dword [wynik], eax
        mov  dword [wynik+4], edx


;[wynik+4]  = K = C*S + D*R

        mov  eax, dword [arg1+4]        ; EAX = C
        mul  dword [arg2]               ; EDX:EAX = C*S
        add  dword [wynik+4], eax
        adc  dword [wynik+8], edx

        adc  dword [wynik+12], 0

        mov  eax, dword [arg1]          ; EAX = D
        mul  dword [arg2+4]             ; EDX:EAX = D*R
        add  dword [wynik+4], eax
        adc  dword [wynik+8], edx

        adc  dword [wynik+12], 0

;[wynik+8]  = J = B*S + C*R + D*Q

        mov  eax, dword [arg1+8]        ; EAX = B
        mul  dword [arg2]               ; EDX:EAX = B*S
        add  dword [wynik+8], eax
        adc  dword [wynik+12], edx

        adc  dword [wynik+16], 0

        mov  eax, dword [arg1+4]        ; EAX = C
        mul  dword [arg2+4]             ; EDX:EAX = C*R
        add  dword [wynik+8], eax
```

```
        adc  dword [wynik+12], edx

        adc  dword [wynik+16], 0

        mov  eax, dword [arg1]          ; EAX = D
        mul  dword [arg2+8]             ; EDX:EAX = D*Q
        add  dword [wynik+8], eax
        adc  dword [wynik+12], edx

        adc  dword [wynik+16], 0

;[wynik+12] = I = A*S + B*R + C*Q + D*P

        mov  eax, dword [arg1+12]       ; EAX = A
        mul  dword [arg2]               ; EDX:EAX = A*S
        add  dword [wynik+12], eax
        adc  dword [wynik+16], edx

        adc  dword [wynik+20], 0

        mov  eax, dword [arg1+8]        ; EAX = B
        mul  dword [arg2+4]             ; EDX:EAX = B*R
        add  dword [wynik+12], eax
        adc  dword [wynik+16], edx

        adc  dword [wynik+20], 0

        mov  eax, dword [arg1+4]        ; EAX = C
        mul  dword [arg2+8]             ; EDX:EAX = C*Q
        add  dword [wynik+12], eax
        adc  dword [wynik+16], edx

        adc  dword [wynik+20], 0

        mov  eax, dword [arg1]          ; EAX = D
        mul  dword [arg2+12]            ; EDX:EAX = D*P
        add  dword [wynik+12], eax
        adc  dword [wynik+16], edx

        adc  dword [wynik+20], 0

;[wynik+16] = H = A*R + B*Q + C*P

        mov  eax, dword [arg1+12]       ; EAX = A
        mul  dword [arg2+4]             ; EDX:EAX = A*R
        add  dword [wynik+16], eax
        adc  dword [wynik+20], edx

        adc  dword [wynik+24], 0

        mov  eax, dword [arg1+8]        ; EAX = B
        mul  dword [arg2+8]             ; EDX:EAX = B*Q
        add  dword [wynik+16], eax
        adc  dword [wynik+20], edx

        adc  dword [wynik+24], 0

        mov  eax, dword [arg1+4]        ; EAX = C
        mul  dword [arg2+12]            ; EDX:EAX = C*P
        add  dword [wynik+16], eax
        adc  dword [wynik+20], edx
```

```
        adc  dword [wynik+24], 0

;[wynik+20] = G = A*Q + B*P

        mov  eax, dword [arg1+12]       ; EAX = A
        mul  dword [arg2+8]             ; EDX:EAX = A*Q
        add  dword [wynik+20], eax
        adc  dword [wynik+24], edx

        adc  dword [wynik+28], 0

        mov  eax, dword [arg1+8]        ; EAX = B
        mul  dword [arg2+12]            ; EDX:EAX = B*P
        add  dword [wynik+20], eax
        adc  dword [wynik+24], edx

        adc  dword [wynik+28], 0

;[wynik+24] = F = A*P

        mov  eax, dword [arg1+12]       ; EAX = A
        mul  dword [arg2+12]            ; EDX:EAX = A*P
        add  dword [wynik+24], eax
        adc  dword [wynik+28], edx

        adc  dword [wynik+32], 0
```

Dzielenie

(przeskocz dzielenie)

Dzielenie dwóch liczb dowolnej długości może być kłopotliwe i dlatego zajmiemy się przypadkiem dzielenia dużych liczb przez liczbę, która mieści się w 32 bitach. Dzielić będziemy od najstarszych bajtów do najmłodszych. Jedna sprawa zasługuje na uwagę: między dzieleniami będziemy *zachowywać resztę w EDX (nie będziemy go zerować)*, gdyż w taki sposób otrzymamy prawidłowe wyniki. Oto algorytm (dzielimy arg1 przez 32-bitowe arg2):

(przeskocz program dzielenia)

```
        mov     ebx, [arg2]     ; zachowujemy dzielnik w wygodnym miejscu

        xor     edx, edx        ; przed pierwszym dzieleniem zerujemy EDX
        mov     eax, [arg1+12]  ; najstarsze 32 bity
        div     ebx
        mov     [wynik+12], eax ; najstarsza część wyniku już jest policzona

                                ; EDX bez zmian! Zawiera teraz resztkę
                                ; z [wynik+12], która jest mniejsza od
                                ; EBX. Ta resztka będzie teraz starszą
                                ; częścią liczby, którą dzielimy.
        mov     eax, [arg1+8]
        div     ebx
        mov     [wynik+8], eax

                                ; EDX bez zmian!
        mov     eax, [arg1+4]
        div     ebx
```

```
mov     [wynik+4], eax

                        ; EDX bez zmian!
mov     eax, [arg1]
div     ebx
mov     [wynik], eax
                        ; EDX = reszta z dzielenia
```

Jeśli wasz dzielnik może mieć więcej niż 32 bity, to trzeba użyć algorytmu podobnego do tego, którego uczyliśmy się w szkole. Ale po takie szczegóły odsyłam do AoA (patrz ostatni akapit w tym tekście).

Operacje logiczne i bitowe

(przeskocz operacje bitowe)
Przerobiliśmy już operacje arytmetyczne, przyszła więc kolej na operacje logiczne.
Na szczęście operacje bitowe AND, OR, XOR i NOT nie zależą od wyników poprzednich działań, więc po prostu wykonujemy je na odpowiadających sobie częściach zmiennych i niczym innym się nie przejmujemy.
Oto przykład (obliczenie arg1 AND arg2):
przeskocz AND

```
mov     eax, [arg1]
and     eax, [arg2]
mov     [wynik], eax

mov     eax, [arg1+4]
and     eax, [arg2+4]
mov     [wynik+4], eax

mov     eax, [arg1+8]
and     eax, [arg2+8]
mov     [wynik+8], eax

mov     eax, [arg1+12]
and     eax, [arg2+12]
mov     [wynik+12], eax
```

Pozostałe trzy (OR, XOR i NOT) będą przebiegać dokładnie w ten sam sposób.

Sprawa z przesunięciami (SHL/SHR) i rotacjami jest nieco bardziej skomplikowana, gdyż bity wychodzące z jednej części zmiennej muszą jakoś znaleźć się w wyższej części. Ale spokojnie, nie jest to aż takie trudne, gdy przypomnimy sobie, że ostatni wyrzucony bit ląduje we fladze CF.
A co zrobić, gdy chcemy przesuwać o więcej niż jeden bit (wszystkie wyrzucone bity nie znajdą się przecież naraz w CF)?
Niestety, trzeba to robić po jednym bicie na raz. Ale ani SHL ani SHR nie pobiera niczego z flagi CF. Dlatego użyjemy operacji rotacji bitów przez flagę CF.
Pora na przykład (SHL arg1, 2):
(przeskocz SHL)

```
shl     dword [arg1], 1         ; wypychamy najstarszy bit do CF
rcl     dword [arg1+4], 1       ; wypchnięty bit wyląduje tutaj w
                                ; bicie numer 0, a najstarszy zostaje
                                ; wypchnięty do CF
```

```
rcl     dword [arg1+8], 1       ; najstarszy bit z [arg1+4] staje się
                                ; tutaj najmłodszym, a najstarszy z
                                ; tej części ląduje w CF
rcl     dword [arg1+12], 1      ; najstarszy bit z [arg1+8] staje się
                                ; tutaj najmłodszym, a najstarszy z
                                ; tej części ląduje w CF

                                ; mamy już SHL o 1 pozycję. Teraz
                                ; drugi raz (dokładnie tak samo):
shl     dword [arg1], 1
rcl     dword [arg1+4], 1
rcl     dword [arg1+8], 1
rcl     dword [arg1+12], 1
```

Podobnie będzie przebiegać operacja SHR (rzecz jasna, SHR wykonujemy *OD GÓRY*):
(przeskocz SHR)

```
; SHR arg1, 1

shr     dword [arg1+12], 1      ; wypychamy najmłodszy bit do CF
rcr     dword [arg1+8], 1       ;wypchnięty bit wyląduje tutaj w bicie
                                ; najstarszym, a najmłodszy zostaje
                                ; wypchnięty do CF
rcr     dword [arg1+4], 1       ; najmłodszy bit z [arg1+8] staje się
                                ; tutaj najstarszym, a najmłodszy z
                                ; tej części ląduje w CF
rcr     dword [arg1], 1         ; najmłodszy bit z [arg1+4] staje się
                                ; tutaj najstarszym, a najmłodszy z
                                ; tej części ląduje w CF
```

Gorzej jest z obrotami (ROL, ROR, RCL, RCR), gdyż ostatni wypchnięty bit musi się jakoś znaleźć na początku. Oto, jak możemy to zrobić (pokażę ROL arg1, 1):
(przeskocz ROL)

```
; najpierw normalny SHL:

shl     dword [arg1], 1
rcl     dword [arg+4], 1
rcl     dword [arg+8], 1
rcl     dword [arg1+12], 1

; teraz ostatni bit jest w CF. Przeniesiemy go do
; najmłodszego bitu EBX.

mov     ebx, 0                  ; tu nie używać XOR! (zmienia flagi)
rcl     ebx, 1                  ; teraz EBX = CF
          ADC ebx, 0)           ; (można też użyć

; i pozostaje nam już tylko dopisać najmłodszy bit w wyniku:

or      [arg1], ebx             ; lub ADD - bez różnicy
```

ROL o więcej niż 1 będzie przebiegać dokładnie tak samo (ten sam kod trzeba powtórzyć wielokrotnie). Sprawa z RCL różni się niewiele od tego, co pokazałem wyżej. Ściśle mówiąc, SHL zamieni się na RCL i nie musimy zajmować się bitem, który wychodzi do CF (bo zgodnie z tym, jak działa RCL ten bit musi tam pozostać). Cała operacja będzie więc wyglądać po prostu tak:
przeskocz RCL

```
rcl     dword [arg1], 1
rcl     dword [arg+4], 1
rcl     dword [arg+8], 1
rcl     dword [arg1+12], 1
```

Operacje ROR i RCR przebiegają podobnie:
(przeskocz ROR)

```
; ROR arg1, 1

; najpierw normalny SHR (pamiętajcie, że od góry):

shr     dword [arg1+12], 1
rcr     dword [arg1+8], 1
rcr     dword [arg1+4], 1
rcr     dword [arg1], 1         ; najmłodszy bit został wypchnięty

; teraz ostatni bit jest w CF. Przeniesiemy go do
; najstarszego bitu EBX.

mov     ebx, 0                  ; tu nie używać XOR! (zmienia flagi)
rcr     ebx, 1                  ; teraz EBX = 00000000 lub 80000000h

; i pozostaje nam już tylko dopisać najstarszy bit w wyniku:

or      [arg1+12], ebx
```

I już tylko prosty RCR:
(przeskocz RCR)

```
rcr     dword [arg1+12], 1
rcr     dword [arg1+8], 1
rcr     dword [arg1+4], 1
rcr     dword [arg1], 1
```

Porównywanie liczb

(przeskocz porównywanie)

Porównywanie należy oczywiście zacząć od najstarszej części i schodzić do coraz to niższych części. Pierwsza różniąca się para porównywanych elementów powie nam, jaka jest relacja między całymi liczbami. Porównywać można dowolną ilość bitów na raz, w tym przykładzie użyję podwójnych słów (32 bity) i będę sprawdzał na równość:
(przeskocz program do porównywania)

```
mov     eax, [arg1+12]
cmp     eax, [arg2+12]  ; porównujemy najstarsze części
jne     nie_rowne
mov     eax, [arg1+8]
cmp     eax, [arg2+8]
jne     nie_rowne
```

```
mov     eax, [arg1+4]
cmp     eax, [arg2+4]
jne     nie_rowne
mov     eax, [arg1]
cmp     eax, [arg2]     ; porównujemy najmłodsze części
jne     nie_rowne
jmp     rowne
```

To by było na tyle z rozszerzonej arytmetyki. Mam nadzieję, że algorytmy te wytłumaczyłem wystarczająco dobrze, abyście mogli je zrozumieć. Jeśli nawet coś nie jest od razu jasne, to należy przejrzeć rozdział o instrukcjach procesora i wrócić tutaj - to powinno rozjaśnić wiele ewentualnych wątpliwości.

Niektóre algorytmy zawarte tutaj wziąłem ze wspaniałej książki Art of asembler (Art of asembly Language Programming, AoA) autorstwa *Randalla Hyde'a*. Książkę tę zawsze i wszędzie polecam jako świetny materiał do nauki nie tylko samego asemblera, ale także architektury komputerów i logiki. Książka ta dostępna jest *ZA DARMO*.

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia

1. Napisz program, który zrobi, co następuje:
    1. Przemnoży EAX przez EBX (wartości dowolne, ale nie za małe) i zachowa wynik (najlepiej w rejestrach).
    2. Przemnoży ECX przez EBP.
    3. Jeśli dowolny wynik wyszedł zero (sprawdzić każdy co najwyżej 1 instrukcją), to niech przesunie te drugi w prawo o 4 miejsca. Jeśli nie, to niech doda je do siebie.

Jak pisać programy w języku assembler pod Linuksem?
Część 14 - Operacje o wielokrotnej precyzji, czyli co zrobić, gdy liczby nie mieszczą się w rejestrach.

Czasami w naszych programach zachodzi potrzeba, aby posługiwać się np. liczbami przekraczającymi 4 czy nawet 8 bajtów, a my mamy tylko rejestry 32-bitowe (lub czasem 16-bitowe).
Co wtedy zrobić?
Odpowiedzi na to właśnie pytanie postaram się udzielić w tej części kursu.

Do naszych celów posłuży coś, co sę nazywa "arytmetyką wielokrotnej precyzji" (ang. Multiprecision Arithmetic). Generalną zasadą będzie zajmowanie się obliczeniami "po kawałku" (bo z resztą inaczej się nie da) i zapamiętywanie, czy z poprzedniego kawałka wynieśliśmy coś "w pamięci" (do tego celu w prosty sposób wykorzystamy flagę CF, która wskazuje właśnie, czy nie nastąpiło przepełnienie).
Najpierw kilka ustaleń:

1. Będę tutaj używał rejestrów 32-bitowch, ale w razie potrzeby dokładnie te same algorytmy działają dla rejestrów 16-bitowych.
2. Zmienne "arg1" i "arg2" mają po 16 bajtów, tj. po 128 bitów każda. Na potrzeby nauki wystarczy w sam raz.
3. Zmienna "wynik" ma tyle samo bajtów, co "arg1" i "arg2", z wyjątkiem mnożenia, gdzie oczywiście musi być dwa razy większa.
4. Zmienna "wynik" na początku zawiera zero.
5. Kod nie będzie optymalny, ale chodzi mi o to, aby był jak najbardziej jasny i przejrzysty.

A więc do dzieła.

Zacznijmy od dodawania. Dodawanie, podobnie jak uczyli nas w szkole, zaczynamy od najmłodszych cyfr (cyfr jedności) - tyle że zamiast pojedynczych cyferek będziemy dodawać całe 32-bitowe kawałki naraz. Flag CF powie nam, czy z poprzedniego dodawania wynosimy coś w pamięci. To coś trzeba oczywiście dodać potem do wyższej części wyniku.
No to dodajemy:

```
mov     eax, [arg1]             ; najpierw dodajemy najmłodsze części
                                ; (pierwsze 4 bajty każdej ze zmiennych)
add     eax, [arg2]
mov     [wynik], eax            ; i wynik zapisujemy w najmłodszych 4 bajtach wyniku

ADC     dword [wynik+4],0       ; bardzo ważne! Do bezpośrednio starszej od naszej
                                ; części wyniku dodajemy coś, co było "w pamięci".
                                ; Jest to oczywiście 1 lub 0, w zależności od tego,
                                ; czy w pierwszym dodawaniu
                                ; nastąpiło przepełnienie, czy też nie.


mov     eax, [arg1+4]           ; dodajemy drugie 4 bajty
add     eax, [arg2+4]

adc     dword [wynik+8], 0      ; z tego dodawania przed chwilą też mogło zostać coś
                                ; "w pamięci"

ADD     [wynik+4], eax          ; zauważcie ADD, a nie MOV. Robiąc MOV skasowalibyśmy
                                ; to, co zapisaliśmy w poprzednim ruchu do [wynik+4]

adc     dword [wynik+8], 0      ; z tego dodawania przed chwilą też mogło zostać coś
                                ; "w pamięci"
```

```
        mov     eax, [arg1+8]           ; dodajemy trzecie 4 bajty
        add     eax, [arg2+8]
        adc     dword [wynik+12], 0

        add     [wynik+8], eax
        adc     dword [wynik+12], 0



        mov     eax, [arg1+12]          ; dodajemy czwarte 4 bajty
        add     eax, [arg2+12]
;       adc     dword [wynik+16], 0     ; patrz komentarz niżej

        add     [wynik+12], eax

        jc      blad_przepelnienie      ; nie musimy tego robić, gdy zmienna "wynik" jest
                                        ; większa rozmiarowo niż arg1 i arg2. Wtedy robimy:
;       adc     dword [wynik+16], 0
```

To chyba było dość proste i naturalne.
Trochę bardziej optymalna wersja wygląda tak:

```
        mov     eax, [arg1]
        add     eax, [arg2]
        mov     [wynik], eax

        mov     eax, [arg1+4]
        adc     eax, [arg2+4]           ; w 1 kroku dodajemy drugi argument + to,
                                        ; co wyszło z poprzedniego dodawania
        mov     [wynik+4], eax

        mov     eax, [arg1+8]
        adc     eax, [arg2+8]
        mov     [wynik+8], eax

        mov     eax, [arg1+12]
        adc     eax, [arg2+12]
        mov     [wynik+12], eax

        jc      blad_przepelnienie
```


Teraz odejmowanie. W szkole uczyli nas, że zaczynamy od najmłodszych cyfr i ewentualnie "pożyczamy" od starszych. Tutaj będziemy robić dokładnie tak samo! Wymaga to jednak poznania nowej instrukcji - SBB (Substract with Borrow). Działa ona tak samo, jak zwykła instrukcja odejmowania SUB, ale dodatkowo odemuje wartość flagi CF, czyli 1 lub 0, w zależności od tego, czy w poprzednim kroku musieliśmy "pożyczać" czy też nie. Ewentualną "pożyczkę" trzeba oczywiście odjąć od wyższej części wyniku.
Piszmy więc (od "arg1" będziemy odejmować "arg2"):

```
        mov     eax, [arg1]             ; odejmujemy najpierw najmłodsze 4 bajty
        sub     eax, [arg2]

        mov     [wynik], eax            ; zapisujemy wynik

        sbb     dword [wynik+4], 0      ; jeśli musieliśmy pożyczać, to pożyczkę odejmujemy od
                                        ; starszej części wyniku.
```

```
        mov     eax, [arg1+4]           ; drugie 4 bajty
        sub     eax, [arg2+4]

        sbb     dword [wynik+8], 0      ; odejmujemy pożyczkę od starszej części

        ADD     [wynik+4], eax          ; ponownie zauważcie ADD, a nie MOV. Tutaj musi się
                                        ; znaleźć wynik odejmowania ewentualnie pomniejszony
                                        ; o pożyczkę. To właśnie zrobiliśmy, tylko w
                                        ; odwrotnej kolejności (najpierw odjęliśmy pożyczkę,
                                        ; potem dodaliśmy wynik).

        mov     eax, [arg1+8]           ; trzecie 4 bajty
        sub     eax, [arg2+8]

        sbb     dword [wynik+12], 0     ; odejmujemy pożyczkę od starszej części

        add     [wynik+8], eax


        mov     eax, [arg1+12]          ; czwarte 4 bajty
        sub     eax, [arg2+12]

;       sbb     dword [wynik+16], 0     ; tylko gdy wynik jest większy rozmiarowo od
                                        ; argumentów. W innym przypadku:

        jc      arg1_mniejszy_od_arg2   ; gdy odjęliśmy już wszystkie części, a ciągle musimy
                                        ; pożyczać, to oznacza, że w ogóle "arg1" był mniejszy
                                        ; od "arg2". Tylko pamiętajcie, że gdy ten skok
                                        ; zostanie wykonany, to i tak musicie wykonać to:

        add     [wynik+12], eax
```

Powyższy kod także nie powinien być trudny do zrozumienia (po prostu ciągle wykonujemy te same cztery operacje, przesuwając się tylko za każdym razem do góry).
Bardziej optymalna wersja:

```
        mov     eax, [arg1]
        sub     eax, [arg2]
        mov     [wynik], eax

        mov     eax, [arg1+4]
        sbb     eax, [arg2+4]           ; odejmujemy razem z pożyczką
        mov     [wynik+4], eax

        mov     eax, [arg1+8]
        sbb     eax, [arg2+8]
        mov     [wynik+8], eax

        mov     eax, [arg1+12]
        sbb     eax, [arg2+12]
        mov     [wynik+12], eax

        jc      arg1_mniejszy_od_arg2
```


Teraz zajmiemy się negacją (zmianą znaku liczby). Ta operacja jest o tyle "dziwna", że wykonujemy ją "od góry" (od najstarszych bajtów) i po negacji niższych trzeba zadbać o "pożyczkę" we wszystkich wyższych częściach.
Popatrzcie (będziemy negować "arg1"):

```
        neg     dword [arg1+12]

        neg     dword [arg1+8]
        sbb     dword [arg1+12], 0

        neg     dword [arg1+4]
        sbb     dword [arg1+8], 0
        sbb     dword [arg1+12], 0

        neg     dword [arg1]
        sbb     dword [arg1+4], 0
        sbb     dword [arg1+8], 0
        sbb     dword [arg1+12], 0
```

Dla większych liczb nie wygląda to za ciekawie. Dlatego najprostszym sposobem będzie po prostu odjęcie danej liczby od zera, do czego zastosujemy zwykły algorytm odejmowania.

Mnożenie jest nieco bardziej skomplikowane, ale ciągle robione tak jak w szkole, czyli od prawej. Ustalmy dla wygody, że arg1 zawiera ABCD, a arg2 - PQRS (każda z liter oznacza 32 bajty). Ogólny schemat wygląda teraz tak:

```
                          A  B  C  D
                        * P  Q  R  S
                        --------------
                                 D*S
                              C*S
                           B*S
                        A*S
                              D*R
                           C*R
                        B*R
                     A*R
                           D*Q
                        C*Q
                     B*Q
                  A*Q
                        D*P
                     C*P
                  B*P
              + A*P
             -----------------------------
                   F  G  H  I  J  K  L
```

```
[wynik]    = L = D*S
[wynik+4]  = K = C*S + D*R
[wynik+8]  = J = B*S + C*R + D*Q
[wynik+12] = I = A*S + B*R + C*Q + D*P
[wynik+16] = H = A*R + B*Q + C*P
[wynik+20] = G = A*Q + B*P
[wynik+24] = F = A*P
(rzecz jasna, każdy iloczyn zajmuje 2 razy po 4 bajty, np. L zajmuje [wynik] i
```

```
częściowo [wynik+4], ale tutaj podałem tylko miejsca, gdzie pójdą najmłodsze części
każdego w iloczynów)

Policzenie L i K wyglądałoby tak:

        mov     eax, [arg1]             ; EAX = D
        mul     dword [arg2]            ; EDX:EAX = D*S
        mov     [wynik], eax
        mov     [wynik+4], edx

        mov     eax, [arg1+4]           ; EAX = C
        mul     dword [arg2]            ; EDX:EAX = C*S
        add     [wynik+4], eax
        adc     [wynik+8], edx

        mov     eax, [arg1]             ; EAX = D
        mul     dword [arg2+4]          ; EDX:EAX = D*R
        add     [wynik+4], eax
        adc     [wynik+8], edx

;       adc     [wynik+12], 0           ; gdy będziemy dalej liczyć
```

Jak widać, nie jest to sprawa prosta, dlatego nie umieszczam tutaj pełnego rozwiązania.

Dzielenie dwóch liczb dowolnej długości może być kłopotliwe i dlatego zajmiemy się przypadkiem dzielenia dużych liczb przez liczbę, która mieści się w 32 bitach. Dzielić będziemy od najstarszych bajtów do najmłodszych. Jedna sprawa zasługuje na uwagę: między dzieleniami będziemy zachowywać resztę w EDX (nie będziemy go zerować), gdyż w taki sposób otrzymamy prawidłowe wyniki. Oto algorytm (dzielimy "arg1" przez 32-bitowe "arg2"):

```
        mov     ebx, [arg2]             ; zachowujemy dzielnik w wygodnym miejscu

        xor     edx, edx                ; przed pierwszym dzieleniem zerujemy EDX
        mov     eax, [arg1+12]          ; najstarsze 32 bity
        div     ebx
        mov     [wynik+12], eax         ; najstarsza część wyniku już jest policzona

                                        ; EDX bez zmian!
        mov     eax, [arg1+8]
        div     ebx
        mov     [wynik+8], eax

                                        ; EDX bez zmian!
        mov     eax, [arg1+4]
        div     ebx
        mov     [wynik+4], eax

                                        ; EDX bez zmian!
        mov     eax, [arg1]
        div     ebx
        mov     [wynik], eax
                                        ; EDX = reszta z dzielenia
```

Jeśli wasz dzielnik może mieć więcej niż 32 bity, to trzeba użyć algorytmu podobnego do tego, którego uczyliśmy się w szkole. Ale po takie szczegóły odsyłam do AoA (patrz ostatni akapit w tym tekście).

Przerobiliśmy już operacje arytmetyczne, przyszła więc kolej na operacje logiczne.
Na szczęście operacje bitowe AND, OR, XOR i NOT nie zależą od wyników poprzednich działań, więc po prostu wykonujemy je na odpowiadających sobie częściach zmiennych i niczym innym się nie przejmujemy.
Oto przykład (obliczenie "arg1" AND "arg2"):

```
        mov     eax, [arg1]
        and     eax, [arg2]
        mov     [wynik], eax

        mov     eax, [arg1+4]
        and     eax, [arg2+4]
        mov     [wynik+4], eax

        mov     eax, [arg1+8]
        and     eax, [arg2+8]
        mov     [wynik+8], eax

        mov     eax, [arg1+12]
        and     eax, [arg2+12]
        mov     [wynik+12], eax
```

Pozostałe trzy (OR, XOR i NOT) będą przebiegać dokładnie w ten sam sposób.

Sprawa z przesunięciami (SHL/SHR) i rotacjami jest nieco bardziej skomplikowana, gdyż bity wychodzące z jednej części zmiennej muszą jakoś znaleźć się w wyższej części. Ale spokojnie, nie jest to aż takie trudne, gdy przypomnimy sobie, że ostatni wyrzucony bit ląduje we fladze CF.
A co zrobić, gdy chcemy przesuwać o więcej niż jeden bit (wszystkie wyrzucone bity nie znajdą się przecież naraz w CF)?
Niestety, trzeba to robić po jednym bicie na raz. Ale ani SHL ani SHR nie pobiera niczego z flagi CF. Dlatego użyjemy operacji rotacji bitów przez flagę CF.
Pora na przykład (SHL "arg1", 2):

```
        shl     dword [arg1], 1         ; wypychamy najstarszy bit
        rcl     dword [arg+4], 1        ; wypchnięty bit wyląduje tutaj w bicie nr 0, a
                                        ; najstarszy zostaje wypchnięty
        rcl     dword [arg+8], 1
        rcl     dword [arg1+12], 1

                                        ; mamy już SHL o 1 pozycję. Teraz drugi raz
                                        ; (dokładnie tak samo):
        shl     dword [arg1], 1
        rcl     dword [arg+4], 1
        rcl     dword [arg+8], 1
        rcl     dword [arg1+12], 1
```

Podobnie będzie przebiegać operacja SHR (rzecz jasna, SHR wykonujemy OD GÓRY):

```
; SHR "arg1", 1

        shr     dword [arg1+12], 1
        rcr     dword [arg1+8], 1
        rcr     dword [arg1+4], 1
        rcr     dword [arg1], 1
```

Gorzej jest z obrotami (ROL, ROR, RCL, RCR), gdyż ostatni wypchnięty bit musi się jakoś znaleźć na

początku. Oto, jak możemy to zrobić (pokażę ROL "arg1", 1):

```
; najpierw normalny SHL:

        shl     dword [arg1], 1
        rcl     dword [arg+4], 1
        rcl     dword [arg+8], 1
        rcl     dword [arg1+12], 1

; teraz ostatni bit jest w CF. Przeniesiemy go do najmłodszego bitu EBX.

        mov     ebx, 0                  ; nie używać XOR! (zmienia flagi)
        rcl     ebx, 1                  ; teraz EBX = CF (można też użyć "ADC ebx, 0")

; i pozostaje nam już tylko dopisać najmłodszy bit w wyniku:

        or      [arg1], ebx             ; lub ADD - bez różnicy
```

ROL o więcej niż 1 będzie przebiegać dokładnie tak samo (ten sam kod trzeba powtórzyć wielokrotnie). Sprawa z RCL różni się niewiele od tego, co pokazałem wyżej. Ściśle mówiąc, SHL zamieni się na RCL i nie musimy zajmować się bitem, który wychodzi do CF (bo zgodnie z tym, jak działa RCL ten bit musi tam pozostać). Cała "operacja" będzie więc wyglądać po prostu tak:

```
        rcl     dword [arg1], 1
        rcl     dword [arg+4], 1
        rcl     dword [arg+8], 1
        rcl     dword [arg1+12], 1
```

Operacje ROR i RCR przebiegają podobnie:

```
; ROR "arg1", 1

; najpierw normalny SHR (pamiętajcie, że od góry):

        shr     dword [arg1+12], 1
        rcr     dword [arg1+8], 1
        rcr     dword [arg1+4], 1
        rcr     dword [arg1], 1         ; najmłodszy bit został wypchnięty

; teraz ostatni bit jest w CF. Przeniesiemy go do najstarszego bitu EBX.

        mov     ebx, 0                  ; nie używać XOR! (zmienia flagi)
        rcr     ebx, 1                  ; teraz EBX = 00 00 00 00  lub  80 00 00 00h

; i pozostaje nam już tylko dopisać najstarszy bit w wyniku:

        or      [arg1+12], ebx
```

I już tylko prosty RCR:

```
        rcr     dword [arg1+12], 1
        rcr     dword [arg1+8], 1
        rcr     dword [arg1+4], 1
        rcr     dword [arg1], 1
```


To by było na tyle z rozszerzonej ayrtmetyki. Mam nadzieję, że algorytmy te wytłumaczyłem wystarczająco dobrze, abyście mogli je zrozumieć. Jeśli nawet coś nie jest od razu jasne, to należy przejrzeć rozdział o instrukcjach procesora i wrócić tutaj - to powinno rozjaśnić wiele ewentualnych wątpliwości.

Niektóre algorytmy zawarte tutaj wziąłem ze wspaniałej książki Art of Assembler (Art of Assembly Language Programming, AoA) autorstwa Randalla Hyde'a. Książkę tę zawsze i wszędzie polecam jako świetny materiał do nauki nie tylko samego assemblera, ale także architektury komputerów i logiki. Książka ta dostępna jest ZA DARMO ze strony http://webster.cs.ucr.edu

---

Ćwiczenia:

1. Napisz program, który zrobi, co następuje:
    a. Przemnoży EAX przez EBX (wartości dowolne, ale nie za małe) i zachowa wynik (najlepiej w rejestrach).
    b. Przemnoży ECX przez EBP.
    c. Jeśli dowolny wynik wyszedł zero (sprawdzić każdy co najwyżej 1 instrukcją), to niech przesunie te drugi w prawo o 4 miejsca. Jeśli nie, to niech doda je do siebie.

Jak pisać programy w języku asembler pod Linuksem?

Część 15 - Pętle i warunki - czyli o tym, jak używać bloków kontrolnych.

Wszystkie języki wysokiego poziomu mają pewne bloki kontrolne i pętle, co może w oczach niektórych osób stanowić przewagę nad asemblerem. Dlatego teraz pokażę, jak przepisać te wszystkie struktury z wykorzystaniem asemblera, często uzyskując kod lepszy niż ten wytworzony przez kompilatory języków wyższego poziomu.
Zanim zaczniemy, dodam, że nie każdy język wysokiego poziomu posiada opcje kompilacji warunkowej (coś w stylu #ifdef w języku C), ale za to KAŻDY dobry kompilator języka asembler ma taką opcję wbudowaną! Po szczegóły odsyłam do instrukcji posiadanego kompilatora.

Bloki decyzyjne (warunkowe) if/else if/else.

(przeskocz bloki warunkowe)

Przetłumaczenie czegoś takiego na asembler nie jest trudne i opiera się na instrukcjach CMP oraz odpowiednich skokach warunkowych. Pokażę to na przykładzie (będę używał składni języka C, gdyż posiada wszystkie struktury, które chciałbym omówić):
(przeskocz schemat bloku if/else)

```
if (a == b)             /* czy a jest równe b? */
{
        /* część 1 */
}
else if (a == c)
{
        /* część 2 */
}
else
{
        /* część 3 */
}
```

Powyższy kod można po prostu zastąpić czymś takim (zakładam zmienne 32-bitowe):
(przeskocz asemblerowy schemat bloku if/else)

```
        mov     eax, [a]
        cmp     eax, [b]
        jne     elseif1

        ; część 1

        jmp     po_if1
elseif1:
        cmp     eax, [c]        ; pamiętajmy, że [a] już jest w EAX
        jne     else1

        ; część 2
```

```
        jmp     po_if1
else1:

        ; część 3

po_if1:
```

Na szczególną uwagę zasługuje przypadek porównywania zmiennej do zera, gdzie zamiast instrukcji `CMP EAX, 0` użyjemy instrukcji `TEST EAX, EAX`.
Jeśli zaś trafi się Wam dość prosty kod w stylu:
(przeskocz przykład if/else)

```
if (a == b)             /* czy a jest równe b? */
{
        d = a;          /* wstaw wartość a do d */
}
else if (a == c)
{
        d = b;
}
else
{
        d = 0;
}
```

lub wyrażenie warunkowe, czyli coś postaci:

```
        d = (a == b)? a : 0;
```

To możecie (a nawet powinniście) użyć instrukcji warunkowego kopiowania danych CMOV*. Instrukcje te powodują znacznie wydajniejszą pracę procesora (który już nie musi co dwie instrukcje skakać i czytać nowych instrukcji). Pierwszy fragment kodu po przetłumaczeniu mógłby wyglądać tak:
(przeskocz tłumaczenie przykładu if/else)

```
xor     edx, edx        ; domyślna wartość, którą wstawimy
                        ; do zmiennej D wynosi zero

mov     eax, [a]
cmp     eax, [b]
cmove   edx, eax        ; gdy a == b, to do EDX wstaw
                        ; wartość A, czyli EAX

cmp     eax, [c]
cmove   edx, [b]        ; gdy a == c, to do EDX wstaw wartość B

mov     [d], edx        ; do D wstaw wynik operacji
                        ; (A, B lub domyślne 0)
```

A drugi:
(przeskocz tłumaczenie wyrażenia warunkowego)

```
xor     edx, edx        ; domyślna wartość to 0

mov     eax, [a]
cmp     eax, [b]        ; porównaj A z B

cmove   edx, eax        ; gdy równe, to EDX=[a]
```

```
mov     [d], edx        ; do D wstaw wynik operacji
```

Tylko nowoczesne kompilatory języka C potrafią wyczyniać takie sztuczki.
Podobne instrukcje istnieją także dla liczb i rejestrów zmiennoprzecinkowych: FCMOV*.

Pętle.

(przeskocz pętle)

Z pętlami jest trochę gorzej, gdyż jest ich więcej rodzajów.
Zacznijmy od pętli o znanej z góry ilości przejść (powtórzeń, iteracji), czy pętli typu

```
for (wyrażenia początkowe; warunek wykonywania; wyrażenie końcowe)
```

Na przykład:
(przeskocz przykład pętli for)

```
for (i=1; i<10; i=i+1)
{
        j=j+i;
}
```

zostałoby przetłumaczone na:
(przeskocz tłumaczenie tego przykładu)

```
        mov     ecx, 1          ; ECX to zmienna I. i=1
petla_for:
        cmp     ecx, 10
        jae     koniec_petli    ; wychodzimy, gdy i >= 10

        add     eax, ecx        ; EAX to zmienna J. j=j+i

        add     ecx, 1          ; i=i+1
        jmp     short petla_for
koniec_petli:
```

Jeśli warunkiem zakończenia pętli jest to, że pewna zmienna osiągnie zero, można stosować instrukcję LOOP.
Przykład:
(przeskocz drugą pętlę for)

```
for (i=10; i>0; i--)
{
        j=j+i;
}
```

może zostać przetłumaczony na 2 sposoby:
(przeskocz sposoby tłumaczenia)

```
; sposób 1:
        mov     ecx, 10         ; ECX to zmienna I. i=1
petla_for:
        cmp     ecx, 0          ; lub: test ecx, ecx
        jbe     koniec_petli    ; wychodzimy, gdy i <= 0
```

```
        add     eax, ecx        ; EAX to zmienna J. j=j+i

        sub     ecx, 1          ; i=i-1
        jmp     short petla_for
koniec_petli:



; sposób 2:
        mov     ecx, 10         ; ECX to zmienna I. i=1
petla_for:
        add     eax, ecx        ; EAX to zmienna J. j=j+i
        loop    petla_for       ; zmniejsz ECX o 1 i jeśli różny od
                                ;    zera, skocz do: petla_for
```

Pamiętajmy jednak, że instrukcja LOOP działa tylko na rejestrze (E)CX, więc jeśli chcemy mieć kilka zagnieżdżonych pętli, to przed każdą z nich (rozpoczynającą się zmianą rejestru ECX) musimy zachować ten rejestr (np. na stosie), a po zakończeniu pętli musimy przywrócić jego poprzednią wartość.

Sprawa z pętlami o nieznanej ilości powtórzeń nie jest o wiele trudniejsza. Pętla typu for jest całkowicie równoważna pętli while. Właśnie z tego skorzystamy, a kod niewiele będzie się różnić budową od poprzedniego przykładu.
Powiedzmy, że mamy taką pętlę:
(przeskocz ten przykład)

```
for (i=100; i+1<=n; i=i+2)
{
        j=j+i+4;
}
```

Możemy ją zastąpić równoważną konstrukcją:
(przeskocz zamianę for na while)

```
i=100;
while (i+1 <= n)
{
        j=j+i+4;
        i=1+2;
}
```

Otrzymujemy kod:
(przeskocz tłumaczenie while)

```
        mov     ecx, 100        ; ECX to zmienna I. i=100
nasza_petla:
        mov     ebx, ecx
        add     ebx, 1          ; EBX = i+1
        cmp     ebx, [n]        ; sprawdzamy, czy i+1 <= n
        ja      koniec_while    ; wychodzimy, gdy i+1 > n

        add     eax, ecx        ; EAX to zmienna J. j=j+i
        add     eax, 4          ; j=j+i+4

        add     ecx, 2          ; i=i+2
        jmp     short nasza_petla
koniec_while:
```

Ostatni rodzaj pętli to pętle typu do-while (repeat...until). Taka pętla różni się tym od poprzedniczek, że warunek jest sprawdzany po wykonaniu kodu pętli (czyli taka pętla zawsze będzie wykonana co najmniej raz). Daje to pewne możliwości optymalizacji kodu.
Popatrzmy na taki przykład:
(przeskocz przykład do-while)

```
do
{
        i=i+1;
        j=j-1;
} while ((i < n) && (j > 1));
```

Warunek wyjścia to: i >= n LUB j <= 1.
A teraz popatrzcie, co można z tym zrobić:
(przeskocz tłumaczenie do-while)

```
petla_do:
        add     ecx, 1          ; ECX to zmienna I. i=i+1
        add     edx, 1          ; EDX to zmienna J. j=j+1

        cmp     ecx, [n]
        jae     koniec          ; i >= n jest jednym z warunków
                                ; wyjścia. Drugiego nie musimy
                                ; sprawdzać, bo wynik i tak
                                ; będzie prawdą
        cmp     edx, 1
        jbe     koniec          ; j <= 1 to drugi warunek wyjścia

        jmp     petla_do
koniec:
```

Można przepisać to w lepszy sposób:
(przeskocz lepszy sposób)

```
petla_do:
        add     ecx, 1          ; ECX to zmienna I. i=i+1
        add     edx, 1          ; EDX to zmienna J. j=j+1

        cmp     ecx, [n]
        jae     koniec          ; i >= n jest jednym z warunków
                                ; wyjścia. Drugiego nie musimy
                                ; sprawdzać, bo wynik i tak
                                ; będzie prawdą

                                ; jeśli nadal tutaj jesteśmy,
                                ; to z pewnością i < n.
        cmp     edx, 1
        ja      petla_do        ; j <= 1 to drugi warunek
                                ; wyjścia. Jeśli j > 1,
                                ; to kontynuuj wykonywanie pętli.
                                ; Jeśli j < 1, to po prostu
                                ; opuszczamy pętlę:
koniec:
```

Jeśli warunek kontynuacji lub wyjścia z pętli jest wyrażeniem złożonym, to:

- jeśli składa się z alternatyw (działań typu OR, ||), to na pierwszym miejscu sprawdzajcie te warunki, które mają największą szansę być prawdziwe. Oszczędzicie w ten sposób czasu na bezsensowne

sprawdzanie reszty warunków (wynik i tak będzie prawdą).
- jeśli składa się z koniunkcji (działań typu AND, &&), to na pierwszym miejscu sprawdzajcie te warunki, które mają największą szansę być fałszywe. Wynik całości i tak będzie fałszem.

Przykłady:

```
1)    a == 0 || (b > 1 && c < 2)
2)    (b < d || a == 1) && c > 0
```

W przypadku 1 najpierw sprawdzamy, czy a jest równe zero. Jeśli jest, to cały warunek jest prawdziwy. Jeśli nie jest, sprawdzamy najpierw ten z dwóch pozostałych, który ma największą szansę bycia fałszywym (jeśli któryś jest fałszywy, to wynik jest fałszem).
W przypadku 2 najpierw sprawdzamy, czy c jest większe od zera. Jeśli nie jest, to cały warunek jest fałszywy. Jeśli jest, to potem sprawdzamy ten z pozostałych warunków, który ma większą szansę bycia prawdziwym (jeśli któryś jest prawdziwy, to wynik jest prawdą).

Decyzje wielowariantowe (wyrażenia typu switch/case)

(przeskocz decyzje wielowariantowe)

Fragment kodu:
(przeskocz schemat switch/case)

```
switch (a)
{
        case 1: .....
        case 2: .....
        ....
        default: .....
}
```

w prosty sposób rozkłada się na serię wyrażeń if i else if (oraz else, o ile podano sekcję default). Te zaś już umiemy przedstawiać w asemblerze. Jest jednak jedna ciekawa sprawa: jeśli wartości poszczególnych przypadków case *są zbliżone* (coś w stylu 1, 2, 3 a nie 1, 20, 45), to możemy posłużyć się tablicą skoków (ang. jump table). W tej tablicy przechowywane są adresy fragmentów kodu, który ma zostać wykonany, gdy zajdzie odpowiedni warunek. Brzmi to trochę pokrętnie, dlatego szybko pokażę przykład.
(przeskocz przykład switch/case)

```
switch (a)
{
        case 1:
                j=j+1;
                break;
        case 2:
                j=j+4;
                break;
        case 4:
                j=j+23;
                break;
        default:
                j=j-1;
}
```

A teraz tłumaczenie:
(przeskocz tłumaczenie przykładu switch/case)

```
                mov     eax, [a]
                cmp     eax, 1                  ; jeśli a < 1 lub a > 5,
                                                ; to na pewno default
                jb      sekcja_default

                cmp     eax, 5
                ja      sekcja_default

                jmp     [przypadki + eax*2 - 2]

        przyp1:
                add     dword ptr [j], 1        ; NASM/FASM: bez słowa PTR
                jmp     koniec

        przyp2:
                add     dword ptr [j], 4        ; NASM/FASM: bez słowa PTR
                jmp     koniec

        przyp4:
                add     dword ptr [j], 23       ; NASM/FASM: bez słowa PTR
                jmp     koniec

        sekcja_default:
                sub     dword ptr [j], 1

        koniec:

        ....
        przypadki:      dw przyp1, przyp2, sekcja_default, przyp4
```

Kod najpierw sprawdza, czy a ma szansę być w którymś z przypadków (jeśli nie jest, to oczywiście wykonujemy sekcję default). Potem, jeśli a=1, to skacze pod etykietę w `zmienne [przypadki + 1*2 - 2 ] = [przypadki] = przyp1`.
Podobnie, jeśli a=2, skoczymy do przyp2. Jeśli a=3, skoczymy do sekcji default, a jeśli a=4, skoczymy do sekcji przyp4.

Od razu widać wielką zaletę takiego rozwiązania: w *jednej jedynej instrukcji* wiemy, gdzie musimy skoczyć. Jak liczba przypadków będzie wzrastać, zauważymy też wadę tego rozwiązania: rozmiar tablicy szybko rośnie (wynosi on różnicę między wartością najwyższą możliwą a najniższą możliwą pomnożoną przez 2 bajty). Dlatego to rozwiązanie jest nieprzydatne dla możliwych wartości: {1, 20, 45} (42 wartości z 45 byłyby nieużywane, czyli wskazujące na sekcję default - zdecydowane marnotrawienie pamięci). W takim przypadku lepiej użyć tradycyjnej metody if/else if/else.

Mam nadzieję, że wiedza zawarta w tej części kursu umożliwi Wam pisanie lepszych i bardziej złożonych programów niż to było do tej pory. Teraz będziecie wiedzieć, co tak właściwie robię kompilatory, tłumacząc niektóre wyrażenia kontrolne. Ta wiedza pomoże Wam pisać lepsze programy w językach wyższego poziomu (gdyż już teraz wiecie, jak zapisywać wyrażenia logiczne tak, by dostać najbardziej wydajny kod).

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia

1. Zaimplementować zdanie:
   Jeśli EAX jest równy EBX lub ECX nie jest równy EBP, to do EDX wstaw EAX, inaczej do ECX wstaw 0.

2. Zaimplementować zdanie (użyć instrukcji warunkowego przesuwania):
   Jeśli EAX jest równy EBX lub ECX nie jest równy EBP, to do EDX wstaw EAX, inaczej do EDX wstaw 0.

3. Napisać program, który liczy sumę liczb od 10 do dowolnej liczby wpisanej w kodzie/czytanej z linii poleceń.

4. Zaimplementować zdanie:
   Dopóki ECX jest większe od 1, zmniejsz ECX o 2.

5. Zaimplementować zdanie:
   Zwiększaj EAX o 3, dopóki EAX jest mniejsze od 100.

Część 16 - Operacje na łańcuchach znaków. Wyrażenia regularne.

Jak wiemy, łańcuch znaków to nic innego jak jednowymiarowa tablica bajtów. Dlatego podane tutaj informacje tak samo działają dla tablic.

W zestawie instrukcji procesora przeznaczonych jest klika rozkazów przeznaczonych specjalnie do operacji na łańcuchach znaków: MOVS, CMPS, SCAS, LODS, STOS. To nimi właśnie teraz się zajmiemy.

Rozkazy te operują na łańcuchach spod DS:[ESI/RSI] lub ES:[EDI/RDI] lub obydwu. Rejestry segmentowe nie będą tutaj grać dużej roli bo pokazują zawsze na ten sam segment, więc będziemy je pomijać. Oraz, zajmiemy się omówieniem instrukcji tylko na ESI oraz EDI, pomijając rejestry 64-bitowe, dla których wszystko wygląda analogicznie.

Instrukcje występują w 4 formach: *B, *W, *D (dla 32-bitowych) i *Q (dla 64-bitowych). Operują one odpowiednio na bajtach, słowach, podwójnych słowach i danych 64-bitowych. Po każdym wykonaniu pojedynczej operacji zwiększają rejestry ESI i EDI o 1, 2, 4 lub 8, przechodząc tym samym na następne elementy.

UWAGA: Zwiększaniem rejestrów *SI i *DI steruje flaga kierunku DF: jeśli równa 0, oba rejestry są zwiększane, jeśli 1 - są zmniejszane o odpowiednią liczbę (co pozwala np. na przeszukiwanie łańcuchów wspak). Flagę DF można wyczyścić instrukcją CLD, a ustawić instrukcją STD.

MOVS

(przeskocz MOVS)

Zasadą działania tej instrukcji jest przeniesienie odpowiedniej ilości bajtów spod [ESI] i umieszczenie ich pod [EDI]. Ale przeniesienie co najwyżej 4 bajtów to przecież żaden wysiłek:

```
mov     eax, [esi]
mov     [edi], eax
```

Dlatego wymyślono prefiks REP (powtarzaj). Jest on ważny tylko dla instrukcji operujących na łańcuchach znaków oraz instrukcji INS i OUTS. Powoduje on powtórzenie działania instrukcji ECX razy. Teraz już widać możliwości tej instrukcji. Chcemy przenieść 128 bajtów? Proszę bardzo:

```
mov     esi, zrodlo
mov     edi, cel
```

```
cld                             ; idź do przodu
mov     ecx, 128
rep     movsb
```

Oczywiście, dwie ostatnie linijki można było zastąpić czymś takim i też by podziałało:

```
mov     ecx, 32
rep     movsd
```

Sposób drugi oczywiście jest lepszy, bo ma mniej operacji (choć najwiecej czasu i tak zajmuje samo rozpędzenie się instrukcji REP).

Instrukcji REP MOVS* można używać do przenoszenia małej ilości danych. Gdy ilości danych rosną, lepiej sprawują się MMX i SSE (patrz: część 6.)

CMPS

(przeskocz CMPS)

Ta instrukcja porównuje odpowiednią ilość bajtów spod [ESI] i [EDI]. Ale nas oczywiście nie interesuje porównywanie pojedynczych ilości. Myślimy więc o prefiksie REP, ale po chwili zastanowienia dochodzimy do wniosku, że w ten sposób otrzymamy tylko wynik ostatniego porównania, wszystkie wcześniejsze zostaną zaniedbane. Dlatego wymyślono prefiksy REPE/REPZ (powtarzaj, dopóki równe/flaga ZF ustawiona) oraz REPNE/REPNZ (powtarzaj, dopóki nie równe/flaga ZF = 0).
Na przykład, aby sprawdzić równość dwóch łańcuchów, zrobimy tak:

```
mov     esi, lancuch1
mov     edi, lancuch2

mov     ecx, 256        ; tyle bajtów maksymalnie chcemy porównać
cld
repe    cmpsb           ; dopóki są równe, porównuj dalej
jnz     lancuchy_nie_rowne
```

REPE przestanie działać na pierwszych różniących się bajtach. W ECX otrzymamy pewną liczbę. Różnica liczby 256 i tej liczby mówi o ilości identycznych znaków i jednocześnie o tym, na której pozycji znajdują się różniące się znaki.
Oczywiście, jeśli po ukończeniu REPE rejestr ECX=0, to znaczy że sprawdzono wszystkie znaki (i wszystkie dotychczas były równe). Wtedy flagi mówią o ostatnim porównaniu.
REPE CMPS ustawia flagi jak normalna instrukcja CMP.

SCAS

(przeskocz SCAS)

Ta instrukcja przeszukuje łańcuch znaków pod [EDI] w poszukiwaniu bajtu z AL, słowa z AX lub podwójnego słowa z EAX. Służy to do szybkiego znalezienia pierwszego wystąpienia danego elementu w łańcuchu.
Przykład: znaleźć pozycję pierwszego wystąpienia litery Z w łańcuchu lancuch1:

```
        mov     al, "Z"         ; poszukiwany element
        mov     edi, lancuch1
        mov     ecx, 256
        cld
        repne   scasb           ; dopóki sprawdzany znak różny
                                ; od "Z", szukaj dalej

        je      znaleziono

        mov     edi, -1         ; gdy nie znaleziono, zwracamy -1
        jmp     koniec

    znaleziono:
        sub     edi, lancuch1   ; EDI = pozycja znalezionego znaku w łańcuchu
```

REPNE przestanie działać w dwóch przypadkach: ECX=0 (wtedy nie znaleziono szukanego elementu) oraz wtedy, gdy ZF=1 (gdy po drodze natrafiła na szukany element, wynik porównania ustawił flagę ZF).

LODS

(przeskocz LODS)

Instrukcje LODS* pobierają do AL/AX/EAX odpowiednią ilość bajtów spod [ESI]. Jak widać, prefiksy REP* nie mają tutaj sensu, bo w rejestrze docelowym i tak zawsze wyląduje ostatni element.
Ale za to tej instrukcji można używać do pobierania poszczególnych znaków do dalszego sprawdzania, np.

```
        cld
    petla:
        lodsb                           ; pobierz kolejny znak

        cmp     al, 13
        jne     nie_enter

        cmp     al, "0"
        je      al_to_zero

        ....

        loop    petla
```

STOS

(przeskocz STOS)

Instrukcja ta umieszcza AL/AX/EAX pod [EDI]. Poza oczywistym zastosowaniem, jakim jest np. zapisywanie kolejnych podawanych przez użytkownika znaków gdzieś do tablicy, STOS można też użyć do szybkiej inicjalizacji tablicy w czasie działania programu lub do wyzerowania pamięci:

```
mov     edi, tablica

mov     eax, 11223344h
mov     ecx, 1000
cld
rep     stosd
...

tablica: times 1000 dd 0
```

Wyrażenia regularne

Wyrażenia regularne (regular expressions, regex) to po prostu ciągi znaczków, przy użyciu których możemy opisywać dowolne łańcuchy znaków (adresy e-mail, WWW, nazwy plików z pełnymi ścieżkami, ...).
Na wyrażenie regularne składają się różne symbole. Postaram się je teraz po krótce omówić.

- aaa (dowolny ciąg znaków) - reprezentuje wszystkie łańcuchy, które go zawierają, np. laaaaaaaaaato.
- ^ - oznacza początek linii (wiersza). Na przykład wyrażenie ^asembler reprezentuje wszystkie linie, które zaczynają się od ciągu znaków asembler. Innymi słowy, każda linia zaczynająca się od asembler pasuje do tego wyrażenia.
- $ - oznacza koniec linii. Na przykład wyrażenie asm$ reprezentuje wszystkie linie, które kończą się na asm.
- . (kropka) - dowolny znak (z wyjątkiem znaku nowej linii). Na przykład wyrażenie ^a.m$ reprezentuje linie, które zawierają w sobie tylko a*m, gdzie gwiazdka to dowolny znak (w tym cyfry). Do tego wzorca pasują asm, aim, a0m i wiele innych.
- | (Shift+BackSlash)- oznacza alternatywę. Na przykład wyrażenie a|b|z reprezentuje dowolną z tych trzech liter i żadną inną.
- (,) - nawiasy służą do grupowania wyrazów. Na przykład ^(aa)|(bb)|(asm) reprezentuje linie, które zaczynają się od aa, bb lub asm.
- [,] - wyznaczają klasę znaków. Na przykład wszystkie wyrazy zaczynające się od k, a lub j pasują do wzorca [ajk].*. Można tutaj podawać przedziały znaków - wtedy 2 skrajne znaki oddzielamy myślnikiem, np. [a-z]. Umieszczenie w środku znaku daszka ^ oznacza przeciwieństwo, np. [^0-9]

reprezentuje znaki, które nie są cyfrą (a tym samym wszystkie ciągi nie zawierające cyfr).
- ? - oznacza co najwyżej 1 wystąpienie poprzedzającego znaku lub grupy. Na przykład, ko?t reprezentuje wyrazy kot i kt, ale już nie koot.
- * - oznacza dowolną ilość wystąpień poprzedzającego znaku/grupy. Wyrażenie ko*t reprezentuje więc wyrazy kt, kot, koot, kooot, itd.
- + - oznacza co najmniej jedno wystąpienie poprzedzającego znaku/grupy. Na przykład al(fa)+ reprezentuje alfa, alfafa, alfafafa itd, ale nie al.
- {n} - oznacza dokładnie n wystąpień poprzedzającego znaku/grupy. Wyrażenie [0-9]{7} reprezentuje więc dowolny ciąg składający się dokładnie z 7 cyfr.
- {n,} - oznacza co najmniej n wystąpień poprzedzającego znaku/grupy. Wyrażenie [a-z]{2,} reprezentuje więc dowolny ciąg znaków składający się co najmniej z 2 małych liter.
- {n,m} - oznacza co najmniej n i co najwyżej m wystąpień poprzedzającego znaku/grupy. Więc wyrażenie [A-M]{3,7} reprezentuje dowolny ciąg znaków składający się z co najmniej 3 i co najwyżej 7 wielkich liter z przedziału od A do M.
- Jeśli w łańcuchu może wystąpić któryś ze znaków specjalnych, należy go w wyrażeniu poprzedzić odwrotnym ukośnikiem \.

Dalsze przykłady:

- ([a-zA-Z0-9]+\.?)+[a-zA-Z]+@([a-zA-Z0-9]+\.)+[a-zA-Z]{2,4} - adres e-mail (zapisany tak, by login ani domena nie kończyły się kropką)
- ([a-zA-Z]{3,6}://)?([a-zA-Z0-9/\-]+\.?)+[a-zA-Z0-9]+(#[a-zA-Z0-9\-]+)? - adres (z protokołem lub bez) zasobu na serwerze (zapisany tak, by nie kończył się kropką, może zawierać myślniki a w ostatnim członie także znak #)

Poprzednia część kursu (Alt+3)
Kolejna część kursu (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia

1. Napisać program zawierający 2 tablice DWORDów o wymiarach 17 na 31, po czym w trakcie działania programu wypełnić każde pole pierwszej wartością FFEEDDCCh. Potem, 8 pierwszych elementów skopiować do drugiej tablicy, a resztę drugiej wypełnić wartością BA098765h. Wtedy porównać zawartość obu tablic i wyliczyć pierwszą pozycję, na której się różnią (powinna oczywiście wynosić 9)

2. Napisać wyrażenie regularne, które opisze:
   - wszystkie wyrażenia deklaracji zmiennych: DB, DW, DP, DQ, DT
   - znacznik HTML bez atrybutów, czyli coś wyglądające tak: < PRE > lub tak: < /LI > (bez spacji).
   - liczbę szesnastkową dowolnej niezerowej długości z ewentualnym przedrostkiem 0x albo (do wyboru) przyrostkiem H lub h.

Jak pisać programy w języku asembler pod Linuksem?

Część 17 - Pobieranie i wyświetlanie, czyli jak komunikować się ze światem.

O ile wyświetlanie i pobieranie od użytkownika tekstów jest łatwe do wykonania - wystarczy uruchomić tylko jedną funkcję systemową (eax=3 lub 4 przerwania 80h) - to pobieranie i wyświetlanie na przykład liczb wcale nie jest takie proste i każdemu może przysporzyć problemów. W tej części podam parę algorytmów, dzięki którym każdy powinien sobie z tym poradzić.

Wyświetlanie tekstu

(przeskocz wyświetlanie tekstu)
Co prawda wszyscy już to umieją, ale dla porządku też o tym wspomnę.
Wszyscy znają funkcję EAX=4 przerwania Linuksa - w EBX podajemy deskryptor, na który wyświetlamy (1 oznacza standardowe wyjście, najczęściej ekran), w ECX - adres bufora z napisem do wyświetlenia, a w EDX - ilość bajtów do wyświetlenia. Po wywołaniu `int 80h` w EAX dostajemy ilość zapisanych bajtów (jeśli EAX jest ujemny, to wystąpił błąd).
Zawsze można też wyświetlać tekst ręcznie.

Pobieranie tekstu

(przeskocz pobieranie tekstu)
Do pobierania tekstów od użytkownika służy funkcja EAX=3 przerwania Linuksa - w EBX podajemy deskryptor, z którego czytamy (0 oznacza standardowe wejście, najczęściej klawiaturę), w ECX - adres bufora na dane, a w EDX - ilość bajtów do przeczytania. Po wywołaniu `int 80h` w buforze dostajemy dane, a w EAX - ilość przeczytanych bajtów (jeśli EAX jest ujemny, to wystąpił błąd).

Wyświetlanie liczb

(przeskocz wyświetlanie liczb)
Są generalnie dwa podejścia do tego problemu:

1. dzielenie przez coraz mniejsze potęgi liczby 10 (zaczynając od najwyższej odpowiedniej) i wyświetlanie ilorazów
2. dzielenie przez 10 i wyświetlanie reszt wspak

Podejście pierwsze jest zilustrowane takim kodem dla liczb 16-bitowych (0-65535):

```
        mov     ax, [liczba]
        xor     dx, dx
        mov     cx, 10000
        div     cx
        or      al, '0'
        ; wyświetl AL jako znak
        mov     ax, dx
        xor     dx, dx
        mov     cx, 1000
        div     cx
        or      al, '0'
        ; wyświetl AL jako znak
        mov     ax, dx
        mov     cl, 100
        div     cl
        or      al, '0'
        ; wyświetl AL jako znak
        mov     al, ah
        xor     ah, ah
        mov     cl, 10
        div     cl
        or      ax, '00'
        ; wyświetl AL jako znak
        ; potem wyświetl AH jako znak
```

Jak widać, im więcej cyfr może mieć liczba, tym więcej będzie takich bloków. Trzeba zacząć od najwyższej możliwej potęgi liczby 10, bo inaczej może dojść do przepełnienia. W każdym kroku dzielnik musi mieć o jedno zero mniej, gdyż inaczej nie uda się wyświetlić prawidłowego wyniku (może być dwucyfrowy i wyświetli się tylko jakiś znaczek). Ponadto, jeśli liczba wynosi na przykład 9, to wyświetli się 00009, czyli wiodące zera nie będą skasowane. Można to oczywiście ominąć.

Podejście drugie jest o tyle wygodniejsze, że można je zapisać za pomocą pętli. Jest to zilustrowane procedurą _pisz_ld z części czwartej oraz kodem z mojej biblioteki:

```
        mov     ax, [liczba]
        xor     si, si                  ; indeks do bufora
        mov     cx, 10                  ; dzielnik
_pisz_l_petla:                          ; wpisujemy do bufora reszty z
                                        ; dzielenia liczby przez 10,
        xor     dx, dx                  ; czyli cyfry wspak
        div     cx                      ; dziel przez 10
        or      dl, '0'                 ; dodaj kod ASCII cyfry zero
        mov     [_pisz_bufor+si], dl    ; zapisz cyfrę do bufora
        inc     si                      ; zwiększ indeks
        test    ax, ax                  ; dopóki liczba jest różna od 0
        jnz     _pisz_l_petla

_pisz_l_wypis:
        mov     al, [_pisz_bufor+si-1]  ; pobierz znak z bufora
        call    far _pisz_z             ; wyświetla znak
        dec     si                      ; przejdź na poprzedni znak
        jnz     _pisz_l_wypis
```

Zmienna _pisz_bufor to bufor odpowiedniej liczby bajtów.

Pobieranie liczb

(przeskocz pobieranie liczb)
Do tego zagadnienia algorytm jest następujący:

1. wczytaj łańcuch znaków od razu w całości lub wczytuj znak po znaku w kroku 3
2. wstępnie ustaw wynik na 0
3. weź kolejny znak z wczytanego łańcucha znaków (jeśli już nie ma, to koniec)
4. zamień go na jego wartość binarną. Jeśli znak wczytałeś do AL, to wystarczy:
   `sub al, '0'`
5. przemnóż bieżący wynik przez 10
6. dodaj do niego wartość AL otrzymaną z kroku 4
7. skacz do 3

Przykładową ilustrację można znaleźć także w mojej bibliotece:

```
        xor     bx, bx          ; miejsce na liczbę
l_petla:
        call    far _we_z       ; pobierz znak z klawiatury

        cmp     al, lf          ; czy Enter?
        je      l_juz           ; jeśli tak, to wychodzimy
        cmp     al, cr
        je      l_juz
                                ; przepuszczamy Spacje:
        cmp     al, spc
        je      l_petla

        cmp     al, '0'         ; jeśli nie cyfra, to błąd
        jb      l_blad
        cmp     al, '9'
        ja      l_blad

        and     al, 0fh         ; izolujemy wartość (sub al, '0')
        mov     cl, al
        mov     ax, bx

        shl     bx, 1           ; zrobimy miejsce na nową cyfrę
        jc      l_blad

        shl     ax, 1
        jc      l_blad
        shl     ax, 1
        jc      l_blad
        shl     ax, 1
        jc      l_blad

        add     bx, ax          ; BX=BX*10 - bieżącą liczbę mnożymy przez 10
        jc      l_blad

        add     bl, cl          ; dodajemy cyfrę
        adc     bh, 0
        jc      l_blad          ; jeśli przekroczony limit, to błąd

        jmp     short l_petla
l_juz:
        ; wynik w AX
```

Sprawdzanie rodzaju znaku

(przeskocz sprawdzanie rodzaju znaku)
Powiedzmy, że użytkownik naszego programu wpisał nam jakieś znaki (tekst, liczby). Jak teraz sprawdzić, co dokładnie otrzymaliśmy? Sprawa nie jest trudna, lecz wymaga czasem zastanowienia i tablicy ASCII pod ręką.

1. Cyfry.
   Cyfry w kodzie ASCII zajmują miejsca od 30h (zero) do 39h (dziewiątka). Wystarczy więc sprawdzić, czy wczytany znak mieści się w tym zakresie:

```
cmp     al, '0'
jb      nie_cyfra
cmp     al, '9'
ja      nie_cyfra
; tu wiemy, że AL reprezentuje cyfrę.
; Pobranie wartości tej cyfry:
and     al, 0fh ; skasuj wysokie 4 bity, zostaw 0-9
```

2. Litery.
   Litery, podobnie jak cyfry, są uporządkowane w kolejności w dwóch osobnych grupach (najpierw wielkie, potem małe). Aby sprawdzić, czy znak w AL jest literą, wystarczy kod

```
        cmp     al, 'A'
        jb      nie_litera      ; na pewno nie litera
        cmp     al, 'Z'
        ja      sprawdz_male    ; na pewno nie wielka,
                                ; sprawdź małe
        ; tu wiemy, że AL reprezentuje wielką literę.
        ; ...
sprawdz_male:
        cmp     al, 'a'
        jb      nie_litera      ; na pewno nie litera
        cmp     al, 'z'
        ja      nie_litera
        ; tu wiemy, że AL reprezentuje małą literę.
```

3. Cyfry szesnastkowe.
   Tu sprawa jest łatwa: należy najpierw sprawdzić, czy dany znak jest cyfrą. Jeśli nie, to sprawdzamy, czy jest wielką literą z zakresu od A do F. Jeśli nie, to sprawdzamy, czy jest małą literą z zakresu od a do f. Wystarczy połączyć powyższe fragmenty kodu. Wyciągnięcie wartości wymaga jednak więcej kroków:

```
; jeśli AL to cyfra '0'-'9'
and     al, 0fh
; jeśli AL to litera 'A'-'F'
sub     al, 'A' - 10
; jeśli AL to litera 'a'-'f'
sub     al, 'a' - 10
```

   Jeśli AL jest literą, to najpierw odejmujemy od niego kod odpowiedniej (małej lub wielkiej) litery A. Dostajemy wtedy wartość od 0 do 5. Aby dostać realną wartość danej litery w kodzie szesnastkowym, wystarczy teraz dodać 10. A skoro AL-'A'+10 to to samo, co AL-('A'-10), to już wiecie, skąd się

wzięły powyższe instrukcje.

4. Przerabianie wielkich liter na małe i odwrotnie.
   Oczywistym sposobem jest odjęcie od litery kodu odpowiedniej litery A (małej lub wielkiej), po czym dodanie kodu tej drugiej, czyli:

```
; z małej na wielką
sub     al, 'a'
add     al, 'A'
; z wielkiej na małą
sub     al, 'A'
add     al, 'a'
```

   lub nieco szybciej:

```
; z małej na wielką
sub     al, 'a' - 'A'
; z wielkiej na małą
sub     al, 'A' - 'a'
```

   Ale jest lepszy sposób: patrząc w tabelę kodów ASCII widać, że litery małe od wielkich różnią się tylko jednym bitem - bitem numer 5. Teraz widać, że wystarczy

```
; z małej na wielką
and     al, 5fh
; z wielkiej na małą
or      al, 20h
```

Wyświetlanie liczb niecałkowitych

(przeskocz wyświetlanie liczb całkowitych)
To zagadnienie można rozbić na dwa etapy:

1. wyświetlenie części całkowitej liczby
2. wyświetlenie części ułamkowej liczby

Do wyświetlenia części całkowitej może nam posłużyć procedura wyświetlania liczb całkowitych, wystarczy z danej liczby wyciągnąć część całkowitą następującym kodem:

```
frndint                         ; jeśli liczba była w ST0
fistp   qword [cz_calkowita]
```

Pojawia się jednak problem, gdy część całkowita nie zmieści się nawet w 64 bitach. Wtedy trzeba skorzystać z tego samego sposobu, który był podany dla liczb całkowitych: ciągłe dzielenie przez 10 i wyświetlenie reszt z dzielenia wspak.
W tym celu ładujemy na stos FPU część całkowitą z naszej liczby oraz liczbę 10:

```
frndint                         ; jeśli liczba była w ST0
fild    word [dziesiec]         ; zmienna zawierająca wartość 10
fxch    st1                     ; stos: ST0=część całkowita, ST1=10
```

Stos koprocesora zawiera teraz część całkowitą naszej liczby w ST0 i wartość 10 w ST1. Po wykonaniu

```
fprem                           ; stos: ST0=mod (część całkowita,10), ST1=10
```

w ST0 dostajemy resztę z dzielenia naszej liczby przez 10 (czyli cyfrę jedności, do wyświetlenia jako ostatnią). Resztę tę zachowujemy do bufora na cyfry. Teraz dzielimy liczbę przez 10:

```
                                ; ST0=część całkowita, ST1=10
fdiv    st0, st1                ; ST0=część całkowita/10, ST1=10
frndint                         ; ST0=część całkowita z poprzedniej
                                ; podzielonej przez 10, ST1=10
```

i powtarzamy całą procedurę do chwili, w której część całkowita stanie się zerem, co sprawdzamy takim na przykład kodem:

```
ftst                            ; zbadaj liczbę w ST0 i ustaw flagi FPU
fstsw   [status]                ; zachowaj flagi FPU do zmiennej
mov     ax, [status]
sahf                            ; zapisz AH do flag procesora
jnz     powtarzamy_dzielenie
```

Po wyświetleniu części całkowitej należy wyświetlić separator (czyli przecinek), po czym zabrać się do wyświetlania części ułamkowej. To jest o tyle prostsze, że uzyskane cyfry można od razu wyświetlić, bez korzystania z żadnego bufora.
Algorytm jest podobny jak dla liczb całkowitych, z tą różnicą, że teraz liczba jest na każdym kroku mnożona przez 10:

```
                                ; ST0=część ułamkowa, ST1=10
fmul    st0, st1                ; ST0=część ułamkowa * 10, ST1=10
fist    word [liczba]           ; cyfra (część ułamkowa*10) do zmiennej
```

Po wyświetleniu wartości znajdującej się we wskazanej zmiennej, należy odjąć ją od bieżącej liczby, dzięki czemu na stosie znów będzie liczba mniejsza od jeden i będzie można powtórzyć procedurę:

```
fild    word [liczba]           ; ST0=część całkowita,
                                ; ST1=część całkowita + część ułamkowa,
                                ; ST2=10
fsubp   st1, st0                ; ST0=nowa część ułamkowa, ST1=10
```

Po każdej iteracji sprawdzamy, czy liczba jeszcze nie jest zerem (podobnie jak powyżej).

Pobieranie liczb niecałkowitych

Procedurę wczytywania liczb niecałkowitych można podzielić na dwa etapy:

1. wczytanie części całkowitej
2. wczytanie części ułamkowej

Wczytywanie części całkowitej odbywa się podobnie, jak dla liczb całkowitych: bieżącą liczbę pomnóż przez 10, po czym dodaj aktualnie wprowadzoną cyfrę. Kluczowa część kodu wyglądać może więc podobnie do tego fragmentu:

```
; kod wczytujący cyfrę ładuje ją do zmiennej WORD [cyfra]
                                ; ST0=10, ST1=aktualna liczba
fmul    st1, st0                ; ST0=10, ST1=liczba*10
fild    word [cyfra]            ; ładujemy ostatnią cyfrę,
                                ; ST0=cyfra, ST1=10, ST2=10 * liczba
faddp   st2, st0                ; ST0=10, ST1=liczba*10 + cyfra
```

Procedurę tę powtarza się do chwili napotkania separatora części ułamkowej (czyli przecinka, ale można akceptować też kropkę). Od chwili napotkania separatora następuje przejście do wczytywania części ułamkowej.
Aby wczytać część ułamkową, najlepiej powrócić do algorytmu z dzieleniem. Wszystkie wprowadzane cyfry najpierw ładujemy do bufora, potem odczytujemy wspak, dodajemy do naszej liczby i dzielimy ją przez 10. Zasadnicza część pętli mogłaby wyglądać podobnie do tego:

```
fild    word [cyfra]    ; ST0=cyfra, ST0=bieżąca część ułamkowa, ST2=10
faddp   st1, st0        ; ST0=cyfra+bieżąca część ułamkowa, ST1=10
fdiv    st0, st1        ; ST0=nowa liczba/10 = nowy ułamek, ST1=10
```

Po wczytaniu całej części ułamkowej pozostaje tylko dodać ją do uprzednio wczytanej części całkowitej i wynik gotowy.

Pamiętajcie o dobrym wykorzystaniu stosu koprocesora: nigdy nie przekraczajcie ośmiu elementów i nie zostawiajcie więcej, niż otrzymaliście jako parametry.

Poprzednia część kursu (Alt+3)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Ćwiczenia

1. Korzystając z przedstawionych tu algorytmów, napisz algorytmy wczytujące i wyświetlające liczby dziesiętne 8-bitowe.
2. Korzystając z przedstawionych tu algorytmów, napisz algorytmy wczytujące i wyświetlające liczby szesnastkowe 16-bitowe (wystarczy zmienić liczby, przez które mnożysz i dzielisz oraz to, jakie znaki są dozwolone i wyświetlane - dochodzą litery od A do F).

14.02.2010

Dynamiczna alokacja pamięci pod Linuksem

Już w średnio zaawansowanych programach pojawia się potrzeba dynamicznego rezerwowania pamięci, w czasie działania programu, nie wiedząc z góry, ile pamięci będzie potrzeba. Na przykład, użytkownik podaje nam rozmiar tablicy a my musimy taką tablicę utworzyć i na niej operować (nie znając wcześniej nawet maksymalnego jej rozmiaru). Rozwiązaniem takich problemów jest właśnie dynamiczna alokacja pamięci. Pod Linuksem pamięć alokuje się funkcją sys_brk (ustalającą najwyższy dostępny adres w sekcji danych). Przyjmuje ona jeden argument:

- EBX = 0, jeśli chcemy otrzymać aktualny najwyższy dostępny dla nas adres w sekcji danych. Tą wartość powiększymy potem o żądany rozmiar pamięci.
- EBX różny od 0, jeśli chcemy ustawić nowy najwyższy adres w sekcji danych. Adres musi być rozsądny co do wartości i taki, by rezerwowana pamięć nie wchodziła na biblioteki załadowane dynamicznie podczas samego uruchamiania programu.

Jeśli coś się nie udało, sys_brk zwróci -1 (i ustawi odpowiednio zmienną errno) lub też zwróci ujemny kod błędu.
Oczywiście, argument funkcji może być większy (alokacja) lub mniejszy (zwalnianie pamięci) od wartości zwróconej przez sys_brk przy EBX=0.

Jak widać, teoria nie jest skomplikowana. Przejdźmy więc może do przykładu. Ten krótki programik ma za zadanie zaalokować 16kB pamięci (specjalnie tak dużo, żeby przekroczyć 4kB - rozmiar jednej strony pamięci - i udowodnić, że pamięć rzeczywiście została przydzielona) i wyzerować ją (normalnie zapisywanie po nieprzydzielonej pamięci skończy się zamknięciem programu przez system).
(przeskocz program)

```
; Dynamiczna alokacja pamięci pod Linuksem
;
; Autor: Bogdan D., bogdandr (at) op.pl
;
; kompilacja:
;
; nasm -f elf -o alok_linux.o alok_linux.asm
; ld  -o alok_linux alok_linux.o


section .text
global  _start

_start:

        mov     eax, 45         ; sys_brk
        xor     ebx, ebx
        int     80h

        add     eax, 16384      ; tyle chcemy zarezerwować
        mov     ebx, eax
        mov     eax, 45         ; sys_brk
        int     80h

        cmp     eax, 0
        jl      .problem        ; jeśli błąd, to wychodzimy i nic się
                                ; nie wyświetli
```

```
        mov     edi, eax        ; EDI = najwyższy dostępny adres

        sub     edi, 4          ; EDI -> ostatni DWORD dostępny dla nas
        mov     ecx, 4096       ; tyle DWORDów zaalokowaliśmy
        xor     eax, eax        ; będziemy zapisywać zera
        std                     ; idziemy wspak
        rep     stosd           ; zapisujemy cały zarezerwowany obszar
        cld                     ; przywracamy flagę DF do normalnego stanu

        mov     eax, 4
        mov     ebx, 1
        mov     ecx, info
        mov     edx, info_dl
        int     80h             ; wyświetlenie napisu

.problem:

        mov     eax, 1
        xor     ebx, ebx
        int     80h



section .data

info            db      "Udana alokacja pamieci.", 10
info_dl         equ     $ - info
```

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Pisanie własnych bibliotek w języku asembler pod Linuksem

Pewnie zdarzyło się już wam usłyszeć o kimś innym:
Ależ on(a) jest świetnym(ą) programistą(ką)! Nawet pisze własne biblioteki!
Pokażę teraz, że nie jest to trudne, nie ważne jak przerażającym się to może wydawać. Osoby, które przeczytają ten artykuł i zdobędą troszkę wprawy będą mogły mówić:
Phi, a co to za filozofia pisać własne biblioteki!

Zacznijmy więc od pytania: co powinno się znaleźć w takiej bibliotece?
Mogą to być:

- Funkcje wejścia i wyjścia, podobnie jak np. w języku C
- Funkcje, które już przepisywaliśmy ze 20 razy w różnych programach
- Sprawdzone funkcje, napisane przez kogoś innego, których nie umielibyśmy sami napisać, lub które po prostu mogą się przydać

Co to zaś jest to owa biblioteka?
Jest to plik na który składa się skompilowany kod, a więc np. pliki .o. Sama biblioteka najczęściej ma rozszerzenie .a (gdy zawiera statyczny kod) lub .so.* (dla bibliotek współdzielonych). Biblioteka eksportuje na zewnątrz nazwy procedur w niej zawartych, aby linker wiedział, jaki adres podać programowi, który chce skorzystać z takiej procedury.
Będę w tym artykule używał składni i linii poleceń NASMa (Netwide Assembler) z linkerem LD i archiwizatorem AR.

Napiszmy więc jakiś prosty kod źródłowy. Oto on:
(przeskocz przykładowy moduł biblioteki)

```
;  Biblioteka Standardowa
;  Emisja dźwięku przez głośniczek
;  Autor: Bogdan Drozdowski, 09.2002
;  kontakt: bogdandr MAŁPKA op.pl
;  Wersja Linux: 05.02.2004
;  Ostatnia modyfikacja: 29.08.2004

%include "../incl/linuxbsd/nasm/n_system.inc"

global  _graj_dzwiek

KIOCSOUND       equ     0x4B2F

section .data

konsola         db      "/dev/console", 0

struc timespec
        .tv_sec:        resd 1
        .tv_nsec:       resd 1
```

```
endstruc

t1 istruc timespec
t2 istruc timespec

segment biblioteka_dzwiek


_graj_dzwiek:

;  graj

;  wejście:     BX = żądana częstotliwość dźwięku w Hz, co najmniej 19
;               CX:DX = czas trwania dźwięku w mikrosekundach
;
;  wyjście:     CF = 0 - wykonane bez błędów
;               CF = 1 - błąd:   BX za mały

        pushfd
        push    eax
        push    ebx
        push    ecx
        push    edx
        push    esi

        cmp     bx, 19          ;najniższa możliwa częstotliwość to ok. 18Hz
        jb      ._graj_blad

        push    ecx
        push    edx
        push    ebx

        mov     eax, sys_open   ; otwieramy konsolę do zapisu
        mov     ebx, konsola
        mov     ecx, O_WRONLY
        mov     edx, 600q
        int     80h

        cmp     eax, 0
        jg      .otw_ok

        mov     eax, 1          ; jak nie otworzyliśmy konsoli,
                                ; piszemy na standardowe wyjście
.otw_ok:
        mov     ebx, eax        ; EBX = uchwyt do pliku
        mov     esi, eax        ; ESI = uchwyt do pliku

        mov     eax, sys_ioctl  ; sys_ioctl = 54
        mov     ecx, KIOCSOUND
        xor     edx, edx        ; wyłączenie ewentualnych dźwięków
        int     80h

        pop     ebx             ; BX = częstotliwość
        mov     eax, 1234ddh
        xor     edx, edx
        div     ebx             ; EAX=1234DD/EBX - ta liczba idzie do ioctl

        mov     edx, eax
        mov     ebx, esi        ; EBX = uchwyt do konsoli lub stdout
        mov     eax, sys_ioctl
        int     80h
```

```
        pop     edx
        pop     ecx

                                ; pauza o długości CX:DX mikrosekund:
        mov     eax, ecx
        shl     eax, 16
        mov     ebx, 1000000
        mov     ax, dx          ; EAX = CX:DX
        xor     edx, edx
        div     ebx
        mov     [t1+timespec.tv_sec], eax ;  EAX = ilość sekund

        mov     ebx, 1000
        mov     eax, edx
        mul     ebx
        mov     [t1+timespec.tv_nsec], eax ;  EAX = ilość nanosekund

        mov     eax, sys_nanosleep
        mov     ebx, t1
        mov     ecx, t2
        int     80h             ; robimy przerwę...


        mov     eax, sys_ioctl
        mov     ebx, esi        ; EBX = uchwyt do konsoli/stdout
        mov     ecx, KIOCSOUND
        xor     edx, edx        ; wyłączamy dźwięk
        int     80h

        cmp     ebx, 2          ; nie zamykamy stdout
        jbe     ._graj_koniec

        mov     eax, sys_close  ; sys_close = 6
        int     80h


._graj_koniec:
        pop     esi
        pop     edx
        pop     ecx
        pop     ebx
        pop     eax
        popfd
        clc                     ;  zwróć brak błędu

        ret


._graj_blad:
        pop     esi
        pop     edx
        pop     ecx
        pop     ebx
        pop     eax

        popfd
        stc                     ;  zwróć błąd

        ret
```

Jest to moja procedura wytwarzająca dźwięk w głośniczku (patrz mój inny artykuł). Trochę tego jest, co? Ale jest tu dużo spraw, które można omówić. Zacznijmy więc po kolei:

1. `global...`
   Funkcje, które mają być widoczne na zewnątrz tego pliku, a więc możliwe do użycia przez inne programy, muszą być zadeklarowane jako public (w NASMie: global). Tutaj jest to na wszelki wypadek. Niektóre kompilatory domyślnie traktują wszystkie symbole jako publiczne, inne nie. Jeśli w programie będziemy chcieli korzystać z takiej funkcji, należy ją zadeklarować jako extrn (FASM) lub extern (NASM).

2. Deklaracja segmentu
   Żaden przyzwoity kompilator nie pozwoli na pisanie kodu poza jakimkolwiek segmentem (no chyba, że domyślnie zakłada segment kodu, jak NASM). Normalnie, w zwykłych programach, rolę tę pełni dyrektywa section .text.

3. Gwiazdki lub inne elementy oddzielające (tu usunięte)
   Mogą się wydawać śmieszne lub niepotrzebne, ale gdy liczba procedur w pliku zaczyna sięgać 10-20, to NAPRAWDĘ zwiększają czytelność kodu, oddzielając procedury, dane itd.

4. Deklaracja procedury (wcześniej zadeklarowanej jako global)
   Znak podkreślenia z przodu jest tylko po to, by w razie czego nie był identyczny z jakąś etykietą w programie korzystającym z biblioteki.

5. To, czego procedura oczekuje i to, co zwraca.
   Jedną procedurę łatwo zapamiętać. Ale co zrobić, gdy jest ich już 100? Analizować kod każdej, aby sprawdzić, co robi, bo akurat szukamy takiej jednej....? No przecież nie.

6. Dobrą techniką programowania jest deklaracja stałych w stylu EQU (lub #define w C). Zamiast nic nie znaczącej liczby można użyć wiele znaczącego zwrotu, co przyda się dalej w kodzie. I nie kosztuje to ani jednego bajtu. Oczywiście, ukrywa to część kodu (tutaj: numery portów), ale w razie potrzeby zmienia się tą wielkość tylko w 1 miejscu, a nie w 20.

7. Zachowywanie zmienianych rejestrów (push)
   Poza wartościami zwracanymi nic nie może być zmienione! Nieprzyjemnym uczuciem byłoby spędzenie kilku godzin przy odpluskwianiu (debugowaniu) programu tylko dlatego, że ktoś zapomniał zachować jakiegoś rejestru, prawda?

8. Sprawdzanie warunków wejścia, czy są prawidłowe. Zawsze należy wszystko przewidzieć.

9. Kod procedury. Z punktu widzenia tego artykułu jego treść jest dla nas nieistotna.

10. Punkt(y) wyjścia
    Procedura może mieć dowolnie wiele punktów wyjścia. Tutaj zastosowano dwa, dla dwóch różnych sytuacji:
    1. parametr był dobry, procedura zakończyła się bez błędów

    2. parametr był zły, zwróć informację o błędzie

Mamy więc już plik źródłowy. Co z nim zrobić? Skompilować, oczywiście!

```
nasm -f elf naszplik.asm
```

(-f - określ format pliku wyjściowego: Executable-Linkable Format, typowy dla Linuksa)

Mamy już plik naszplik.o. W pewnym sensie on już jest biblioteką! I można go używać w innych programach, np. w pliku program2.asm mamy (FASM):

```
...
extrn _graj_dzwiek              ; NASM: extern _graj_dzwiek

...
...
mov bx,440
mov cx,0fh
mov dx,4240h
call _graj_dzwiek
...
```

I możemy teraz zrobić:

```
nasm -f elf program2.asm
ld -s -o program2 program2.o naszplik.o
```

a linker zajmie się wszystkim za nas - utworzy plik program2, zawierający także naszplik.o. Jaka z tego korzyść? Plik program2.asm może będzie zmieniany w przyszłości wiele razy, ale naszplik.asm/.o będzie ciągle taki sam. A w razie chęci zmiany procedury _graj_dzwiek wystarczy ją zmienić w jednym pliku i tylko jego ponownie skompilować, bez potrzeby wprowadzania tej samej zmiany w kilkunastu innych programach. Te programy wystarczy tylko ponownie skompilować z nową biblioteką, bez jakichkolwiek zmian kodu.

No dobra, ale co z plikami .a?
Otóż są one odpowiednio połączonymi plikami .o. I wszystko działa tak samo.

No ale jak to zrobić?
Służą do tego specjalne programy, w DOSie nazywane librarian (bibliotekarz). My tutaj użyjemy archiwizatora AR. Pliki .o, które chcemy połączyć w bibliotekę podajemy na linii poleceń:

```
ar -r libasm.a plik1.o plik2.o
```

I otrzymujemy plik libasm.a, który można dołączać linkerem do programów:

```
ld -s -o naszprog naszprog.o -L/ścieżka_do_pliku.a -lasm
```

lub:

```
ld -s -o naszprog naszprog.o /ścieżka_do_pliku.a/libasm.a
```

Biblioteki współdzielone .so

Prawie wszystkie programy w Linuksie używają podstawowej biblioteki systemu - biblioteki języka C. Wyobrażacie sobie, ile miejsca w pamięci zajęłyby wszystkie używane kopie tej biblioteki? Na pewno niemało. A poradzono sobie z tym, tworząc specjalny rodzaj plików - bibliotekę współdzieloną, ładowaną i łączoną z programem dynamicznie (w chwili uruchomienia). Pliki te (o rozszerzeniu .so) są odpowiednikami plików DLL znanych z systemów Windows. Teraz pokażę, jak pisać i kompilować takie pliki. Wszystko to znajdziecie też w dokumentacji kompilatora NASM.

Reguły są takie:

1. Dalej trzymajcie się wszystkich powyższych uwag do kodu (komentarze itp.).

2. NIE możemy już się odwoływać normalnie do swoich własnych zmiennych!
   Dlaczego? Przyczyna jest prosta: biblioteki współdzielone są pisane jako kod niezależny od pozycji (Position-Independent Code, PIC) i po prostu nie wiedzą, pod jakim adresem zostaną załadowane przez system. Adres może za każdym razem być inny. Do swoich zmiennych musimy się więc odwoływać trochę inaczej, niż to było do tej pory. Do biblioteki współdzielonej linker dołącza strukturę Globalnej Tablicy Offsetów (Global Offset Table, GOT). Biblioteka deklaruje ją jako zewnętrzną i korzysta z niej do ustalenia adresu swojego kodu. Wystarczy wykonać `call zaraz / zaraz: pop ebx` i już adres etykiety zaraz znajduje się w EBX. Dodajemy do niego adres GOT od początku sekcji (_GLOBAL_OFFSET_TABLE_ wrt ..gotpc) i adres początku sekcji, otrzymując realny adres tablicy GOT + adres etykiety zaraz. Potem już tylko wystarczy odjąć adres etykiety zaraz i już EBX zawiera adres GOT. Do zmiennych możemy się teraz odnosić poprzez `[ebx+nazwa_zmiennej]`.

3. Kompilacja i łączenie.
   O ile kompilacja NASMem jest taka, jak zawsze, to łączenie programu jest zdecydowanie inne. Popatrzcie na opcje LD:
   - `-shared`
     Mówi o tym, że LD ma zbudować bibliotekę współdzieloną, zamiast zwyczajnego pliku wykonywalnego. LD zadba o wszystko, co trzeba (GOT itd).
   - `-soname biblso.so.1`
     Nazwa biblioteki. Ale uwaga - NIE jest to nazwa pliku, tylko wewnętrzna nazwa samej biblioteki. Jak będziecie dodawać kolejne wersje, to nie zmieniajcie nazwy wewnętrznej, tylko nazwę pliku .so, a zróbcie dowiązanie symboliczne do tego pliku, z nazwą taką jak wewnętrzna nazwa biblioteki, np. waszabibl.so.1 jako link do waszabibl.so.1.1.5.

4. Deklaracje zmiennych i funkcji globalnych.
   Każda funkcja, którą chcemy zrobić globalną (widoczną dla programów korzystających z biblioteki), musi być zadeklarowana nie tylko jako extern, ale musimy podać też, że jest to funkcja. Pełna dyrektywa wygląda teraz:

   ```
   global nazwaFunkcji:function
   ```

   Przy eksportowaniu danych dodajemy słowo data i rozmiar danych, np. dla tablic:

```
global tablica1:data tablica1_dlugosc

tablica1:               resb    100
tablica1_dlugosc        equ     $ - tablica1
```

5. Uruchamianie funkcji zewnętrznych (np. z biblioteki C)
   Sprawa jest już dużo prostsza niż w przypadku danych. Funkcję zewnętrzną deklarujemy oczywiście słowem extern, a zamiast call nazwaFunkcji piszemy

   ```
   call nazwaFunkcji wrt ..plt
   ```

   PLT oznacza Procedure Linkage Table, czyli tablicę linkowania procedur (funkcji). Zawiera ona skoki do odpowiednich miejsc, gdzie znajduje się dana funkcja.

A oto gotowy przykład. Biblioteka eksportuje jedną funkcję, która po prostu wyświetla napis.
(przeskocz przykładową bibliotekę współdzieloną)

```
; Przykład linuksowej biblioteki współdzielonej .so
;
; Autor: Bogdan D., bogdandr (at) op.pl
;
; kompilacja:
;   nasm -f elf -o biblso.o biblso.asm
;   ld -shared -soname biblso.so.1 -o biblso.so.1 biblso.o


section .text
extern  _GLOBAL_OFFSET_TABLE_   ; zewnętrzny, uzupełniony przez linker
global  info:function           ; eksportowana funkcja
;extern printf                  ; funkcja zewnętrzna

; makro do pobierania adresu GOT; wynik w EBX.
%imacro wez_GOT 0

        call    %%zaraz
        %%zaraz:
        pop     ebx
        add     ebx, _GLOBAL_OFFSET_TABLE_ + $$ - %%zaraz wrt ..gotpc
%endmacro

info:
                                ; zachowanie zmienianych rejestrów
        push    eax
        push    ebx
        push    ecx
        push    edx

        wez_GOT                 ; pobieramy adres GOT
        push    ebx             ; zachowujemy EBX

        mov     eax, 4          ; funkcja pisania do pliku
                        ; do ECX załaduj ADRES napisu (stad LEA a nie MOV)
        lea     ecx, [ebx + napis wrt ..gotoff]
        mov     ebx, 1          ; plik = 1 = standardowe wyjście (ekran)
        mov     edx, napis_dl   ; długość napisu
        int     80h             ; wyświetl
```

```
; a tak uruchamiamy funkcje zewnętrzne:
        pop     ebx             ; przywracamy EBX
        lea     ecx, [ebx + napis wrt ..gotoff] ; ECX = adres napisu
        push    ecx             ; adres na stos (jak dla funkcji z C)
;       call    printf wrt ..plt        ; uruchomienie funkcji
        add     esp, 4          ; usunięcie argumentów ze stosu

                                ; przywracanie rejestrów
        pop     edx
        pop     ecx
        pop     ebx
        pop     eax

        xor     eax, eax        ; funkcja zwraca 0 jako brak błędu
        ret

section .data

napis           db      "Jestem biblioteka wspoldzielona!", 10, 0
napis_dl        equ     $ - napis
```

Program sprawdzający, czy biblioteka działa jest wyjątkowo prosty: jedno uruchomienie funkcji z biblioteki i wyjście. Na uwagę zasługuje jednak ta długa linijka z uruchomieniem LD. Przyjrzyjmy się bliżej:

- -dynamic-linker /lib/ld-linux.so.2
  Mówi o nazwie programu, którego trzeba użyć do dynamicznego łączenia. Bez tej opcji nasz program nie podziała i dostaniemy błąd Accessing a corrupted shared library

- -nostdlib
  Nie dołącza żadnych standardowych bibliotek.

- -o biblsotest biblsotest.o
  Nazwy pliku wyjściowego i wejściowego.

- biblso.so.1
  Biblioteka, z którą należy połączyć ten program

(przeskocz test biblioteki współdzielonej)

```
; Program testujący linuksową bibliotekę współdzieloną .so
;
; Autor: Bogdan D., bogdandr (at) op.pl
;
; kompilacja:
; nasm -f elf -o biblsotest.o biblsotest.asm
; ld -dynamic-linker /lib/ld-linux.so.2 -nostdlib \
;       -o biblsotest biblsotest.o biblso.so.1


section .text
global  _start

extern  info

_start:
```

```
call    info

mov     eax, 1
xor     ebx, ebx
int     80h
```

Jeśli dostajecie błąd /usr/lib/libc.so.1: bad ELF interpreter: No such file or directory, to utwórzcie w katalogu /usr/lib (jako root) plik libc.so.1 jako dowiązanie symboliczne do libc.so i upewnijcie się, że plik /usr/lib/libc.so ma prawa wykonywania dla wszystkich.

Jeśli system nie widzi biblioteki współdzielonej (a nie chcecie jej pakować do globalnych katalogów jak /lib czy /usr/lib), należy ustawić dodatkową ścieżkę ich poszukiwania.
Ustawcie sobie zmienną środowiskową LD_LIBRARY_PATH tak, by zawierała ścieżki do Waszych bibliotek. Ja u siebie mam ustawioną `LD_LIBRARY_PATH=$HOME:.` , co oznacza, że poza domyślnymi katalogami, ma być przeszukany także mój katalog domowy oraz katalog bieżący (ta kropka po dwukropku), jakikolwiek by nie był.

Ładowanie bibliotek w czasie pracy programu

Gdy nasz program jest na sztywno (statycznie lub nie) łączony z jakąś biblioteką współdzieloną, to w trakcie jego uruchamiania system szuka pliku tej biblioteki, aby móc uruchomić nasz program. Jeśli system nie znajdzie biblioteki, to nawet nie uruchomi naszego programu. Czasem jednak chcemy mieć możliwość zareagowania na taki problem. Oczywiście, bez kluczowych bibliotek nie ma szans uruchomić programu, ale całą resztę można dość łatwo ładować w czasie działania programu. Daje to pewne korzyści:

1. oszczędza pamięć - ładujemy tylko te biblioteki, których nam naprawdę potrzeba, a tuż po zakończeniu pracy z biblioteką, można zwolnić pamięć przez nią zajmowaną.
2. daje możliwość reagowania na brak biblioteki - na przykład można wyświetlić komunikat, że niektóre funkcje programu będą niedostępne. Ale program może nadal działać i wykonać swoje zadanie.

Ładowanie bibliotek w czasie pracy programu polega na wykorzystaniu funkcji z biblioteki libdl. Konkretnie, użyjemy trzech funkcji:

1. dlopen - otwiera i ładuje bibliotekę
   Przyjmuje ona dwa argumenty. Od lewej (ostatni wkładany na stos) są to: nazwa pliku biblioteki współdzielonej (razem ze ścieżką, jeśli jest w niestandardowej) oraz jedna z liczb: RTLD_LAZY (wartość 1), RTLD_NOW (wartość 2), RTLD_GLOBAL (wartość 100h). Określają one sposób dostępu do funkcji w bibliotece, odpowiednio są to:
   - RTLD_LAZY - znajduj adres funkcji w chwili jej wywołania.
   - RTLD_NOW - znajduj adres funkcji od razu, w czasie ładowania biblioteki
   - RTLD_GLOBAL - symbole biblioteki (nazwy funkcji) będą od razu widoczne dla programu tak, jakby biblioteka była włączona na stałe do programu.

   Funkcja dlopen zwraca (w EAX) adres załadowanej biblioteki, którego będziemy potem używać.

2. dlsym - wyławia z biblioteki adres interesującej nas funkcji
   Ta funkcja też przyjmuje dwa argumenty. Od lewej (ostatni wkładany na stos) są to: adres biblioteki, który otrzymaliśmy od funkcji dlopen oraz nazwa funkcji, która nas interesuje *jako łańcuch znaków*.
   Funkcja dlsym zwraca nam (w EAX) adres żądanej funkcji.

3. dlclose - zamyka załadowaną bibliotekę
   Jedynym argumentem tej funkcji jest adres biblioteki, który otrzymaliśmy od funkcji dlopen.

Jest też funkcja systemowa sys_uselib, ale jej dokumentacja jest skromna. W użyciu pewnie byłaby trudniejsza niż libdl.

Pora na przykładowy program. Jego zadaniem będzie załadować bibliotekę biblso.so.1, którą utworzyliśmy w poprzednim podrozdziale, oraz uruchomienie jej jedynej funkcji - info. Oto kod w składnie NASM:
(przeskocz program ładujący bibliotekę)

```
; Program korzystający z biblioteki współdzielonej tak, że
;       nie musi być z nią łączony
;
; Autor: Bogdan D., bogdandr (na) op . pl
;
; kompilacja:
;   nasm -f elf -o shartest.o shartest.asm
;   gcc -s -o shartest shartest.o -ldl

section .text

; będziemy korzystać z biblioteki języka C, więc nasza funkcja
;  główna musi się nazywać  main
global main

%define RTLD_LAZY       0x00001 ; znajduj adres funkcji w chwili wywołania
%define RTLD_NOW        0x00002 ; znajduj adres funkcji od razu, w czasie
                                ; ładowania biblioteki
%define RTLD_GLOBAL     0x00100 ; czy symbole będą od razu widoczne

extern dlopen
extern dlsym
extern dlclose

main:

        push    dword RTLD_LAZY ; ładowanie na żądanie
        push    dword bibl      ; adres nazwy pliku
        call    dlopen          ; otwieramy bibliotekę
        add     esp, 2*4        ; zwalniamy argumenty ze stosu

        test    eax, eax        ; sprawdzamy, czy nie błąd (EAX=0)
        jz      .koniec

        mov     [uchwyt], eax   ; zachowujemy adres biblioteki

        push    dword funkcja   ; adres nazwy żądanej funkcji
        push    dword [uchwyt]  ; adres biblioteki
        call    dlsym           ; szukamy adresu
        add     esp, 2*4

        mov     [adr_fun], eax  ; EAX = znaleziony adres
        call    eax             ; uruchomienie bezpośrednie
        call    [adr_fun]       ; uruchomienie pośrednie

        push    dword [uchwyt]  ; adres biblioteki
        call    dlclose         ; zwalniamy ją z pamięci
        add     esp, 1*4
```

```
.koniec:
        ret                     ; zakończenie funkcji main

section .data

bibl            db      "biblso.so.1", 0        ; nazwa biblioteki
funkcja         db      "info", 0               ; nazwa szukanej funkcji
uchwyt          dd      0
adr_fun         dd      0
```

Muszę wspomnieć o dwóch dość ważnych rzeczach.
Pierwszą jest sposób kompilacji. Skoro łączymy nasz program z biblioteką C, to nasza funkcja główna musi się teraz nazywać main, a *NIE* _start (gdyż funkcja _start już jest w bibliotece języka C). Kompilacja wygląda teraz tak, jak napisałem w programie:

```
nasm -f elf -o shartest.o shartest.asm
gcc -s -o shartest shartest.o -ldl
```

W tym przypadku kompilator GCC uruchamia za nas linker LD, który dołączy niezbędne biblioteki.

Drugą rzeczą jest domyślna ścieżka poszukiwania bibliotek współdzielonych. Jeśli nie chcecie zaśmiecać systemu (lub nie macie uprawnień), pakując swoje biblioteki do /lib czy /usr/lib, ustawcie sobie zmienną środowiskową LD_LIBRARY_PATH tak, by zawierała ścieżki do Waszych bibliotek. Ja u siebie mam ustawioną `LD_LIBRARY_PATH=$HOME:.` , co oznacza, że poza domyślnymi katalogami, ma być przeszukany także mój katalog domowy oraz katalog bieżący (ta kropka po dwukropku), jakikolwiek by nie był.

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

14.02.2010

Wyświetlanie obrazków BMP pod Linuksem

Jeśli przejrzeliście mój poprzedni kurs związany z grafiką, to umiecie już coś samodzielnie narysować.
Ale przecież w Internecie (i nie tylko) jest tyle ciekawych rysunków, nie mówiąc już o tych, które moglibyście stworzyć dla jakiegoś specjalnego celu, np. swojej własnej gry. Dlatego teraz pokażę, jak takie rysunki wyświetlać. Ze względu na prostotę formatu, wybrałem pliki typu BMP (bitmapy). Plik, który wyświetlimy, powinien mieć rozmiar 320x200 pikseli w 256 kolorach (można oczywiście wziąć dowolną inną rozdzielczość, ale trzeba wtedy dobrać tryb graficzny).
Wszystkie operacje na plikach zostały już przez mnie szczegółowo opisane w jednej z części mojego kursu, więc tutaj nie będziemy poświęcać im zbyt wiele uwagi.
Ale przejdźmy wreszcie do interesujących nas szczegółów.

Powinniście zaopatrzyć się w cokolwiek, co opisuje format BMP. Informacje, z których będę tutaj korzystał, znalazłem w Internecie (niestety, nie pamiętam już gdzie, ale możecie poszukać na Wotsit.org).
A oto nagłówek pliku BMP (składnia języka Pascal niestety, info: Piotr Sokolowski, 6 maja 1998r):
(przeskocz opis nagłówka)

```
Type
 TBitMapHeader =
  Record
        bfType :             Word; (dwa bajty)
        bfSize :             LongInt; (cztery bajty)
        bfReserved :         LongInt;
        bfOffBits :          LongInt;
        biSize :             LongInt;
        biWidth :            LongInt;
        biHeight :           LongInt;
        biPlanes :           Word;
        biBitCount :         Word;
        biCompression :      LongInt;
        biSizeImage :        LongInt;
        biXPelsPerMeter :    LongInt;
        biYPelsPerMeter :    LongInt;
        biClrUsed :          LongInt;
        biClrImportant :     LongInt;
  End;
```

Gdzie:

- bftype - jest to dwubajtowa sygnatura BM
- bfsize - czterobajtowy rozmiar pliku
- bfreserved - pole zarezerwowane (0)
- bfoffbits - przesunięcie (adres) początku danych graficznych
- bisize - podaje rozmiar nagłówka
- biwidth - wysokość bitmapy w pikselach
- biheight - szerokość bitmapy w pikselach
- biplanes - liczba planów (prawie zawsze ma wartość 1)
- bibitcound - ilość bitów na piksel. Przyjmuje wartość 1,4,8 lub 24.
- bicompression - sposób kompresji
- bisizeimag - rozmiar obrazka w bajtach. W przypadku bitmapy nieskompresowanej równe 0.
- biXpelspermeter, biYpelspermeter - ilość pikseli na metr
- biclrused - ilość kolorów istniejącej palety, a używanych właśnie przez bitmapę

- biclrimporant - określa, który kolor bitmapy jest najważniejszy, gdy równy 0 to wszystkie są tak samo istotne.

Ale spokojnie - nie musicie znać tych wszystkich pól, bo my nie będziemy wszystkich używać. Ściśle mówiąc, nie będziemy używać ani jednego z tych pól!
No to po co to wszystko?
Po to, aby znać długość nagłówka pliku (54 bajty), który ominiemy przy analizie pliku.

Po nagłówku idzie paleta 256 kolorów * 4 bajty/kolor = kolejny 1kB. Jeśli macie jakieś wątpliwości co do tego jednego kilobajta, to słusznie. Oczywiście, do opisu koloru wystarczą 3 bajty (odpowiadające kolorom czerwonemu, zielonemu i niebieskiemu - RGB), co daje razem 768 bajtów. Co czwarty bajt nie zawiera żadnej istotnej informacji i będziemy go pomijać (zmienna z w programie).

Zaraz po palecie jest obraz, piksel po pikselu. Niestety, nie jest to tak logiczne ustawienie, jak byśmy sobie tego życzyli. Otóż, pierwsze 320 bajtów to ostatni wiersz obrazka, drugie 320 - przedostatni, itd. Dlatego trzeba będzie troszkę pokombinować.
W tym artykule też wykorzystam możliwości biblioteki SVGAlib, ze względu na prostotę jej opanowania. Aby móc z niej korzystać, musicie zainstalować pakiety svgalib oraz svgalib-devel lub po prostu samemu skompilować bibliotekę, jeśli pakiety nie są dostępne.

Do działania programów pod X-ami potrzebne mogą być uprawnienia do pliku /dev/console a pod konsolą tekstową - do pliku /dev/mem.

Zwróćcie uwagę na *sposób kompilacji* poniższego programu. Korzystamy z bibliotek dostępnych dla programistów języka C, więc do łączenia programu w całość najlepiej użyć GCC - zajmie się on dołączeniem wszystkich niezbędnych bibliotek. A skoro używamy gcc, to funkcja główna zamiast _start, *musi się nazywać main* - tak samo jak funkcja główna w programach napisanych w C. I tak samo, zamiast funkcji wychodzenia z programu, możemy użyć komendy RET, aby zamknąć program.

Ale dobierzmy się wreszcie do kodu (składnia FASM):
(przeskocz program)

```
; Program wyświetlający obrazek BMP pod Linuksem z wykorzystaniem SVGAlib
;
; Autor: Bogdan D., bogdandr (at) op.pl
;
; kompilacja:
;   fasm bmp.fasm
;   gcc -o bmp bmp.o -lvga

format ELF
section ".text" executable

public  main

G320x200x256    = 5
TEXT            = 0

extrn   vga_setmode
extrn   vga_drawscansegment
extrn   vga_setpalvec


main:
        mov     eax, 5          ; otwieranie pliku
```

```
        mov     ebx, plik       ; adres nazwy
        mov     ecx, 0          ; odczyt
        mov     edx, 400o       ; - R-- --- ---
        int     80h

        cmp     eax, 0          ; czy błąd?
        jng     koniec

        mov     ebx, eax        ; EBX = deskryptor

        mov     eax, 19         ; zmiana pozycji w pliku
        mov     ecx, 54         ; idziemy tyle bajtów...
        mov     edx, 0          ; ...od początku pliku
        int     80h

        cmp     eax, 0
        jge     seek_ok

problem:                        ; tu trafiamy po błędzie obsługi pliku
        push    eax
        mov     eax, 6
        int     80h             ; zamykamy plik
        pop     eax

        jmp     koniec

seek_ok:
        xor     esi, esi        ; indeks do tablicy "paleta"

czytaj_pal:
        mov     eax, 3          ; czytaj z pliku
        mov     ecx, b          ; pod ten adres
        mov     edx, 4          ; 4 bajty (zmienne b, g, r i z)
        int     80h

        cmp     eax, 0          ; czy błąd?
        jl      problem

                                ; ustawiamy paletę:
        mov     al, [r]
        shr     al, 2           ; dzielimy przez 4, ograniczając liczby do
                                ; przedziału 0-63
        and     eax, 3fh        ; zerujemy pozostałe bity
        mov     [paleta+esi], eax     ; paleta[esi] = [r] / 4

        mov     al, [g]
        shr     al, 2
        and     eax, 3fh
        mov     [paleta+esi+1*4], eax ; paleta[esi] = [g] / 4

        mov     al, [b]
        shr     al, 2
        and     eax, 3fh
        mov     [paleta+esi+2*4], eax ; paleta[esi] = [b] / 4

        add     esi, 3*4        ; przejdź o 3 miejsca dalej -
                                ; na kolejne wartości RGB
                                ; każde miejsce zajmuje 4 bajty

        cmp     esi, 256*3*4    ; sprawdź, czy nie zapisaliśmy
                                ; już wszystkich kolorów
        jb      czytaj_pal
```

```
wyslij_palete:
        push    dword G320x200x256
        call    vga_setmode     ; ustawiamy tryb graficzny:
                                ; 320x200 w 256 kolorach
        add     esp, 1*4        ; zdejmujemy argument ze stosu

        push    dword paleta
        push    dword 256
        push    dword 0
        call    vga_setpalvec   ; ustawiamy paletę barw
        add     esp, 3*4        ; zdejmujemy argumenty ze stosu

        mov     edi, 200        ; tyle linii wyświetlimy
obrazek:
        push    edi             ; zachowaj EDI
        mov     eax, 3          ; czytaj z pliku
        mov     ecx, kolor      ; do tej zmiennej
        mov     edx, 320        ; 320 bajtów (jedną linię obrazu)
        int     80h

        cmp     eax, 0
        jge     .dalej

                                ; w razie błędu wyłączamy tryb graficzny
        push    dword TEXT
        call    vga_setmode     ; ustawiamy tryb tekstowy 80x25
        add     esp, 1*4

        jmp     problem
.dalej:

        push    dword 320       ; tyle bajtów na raz wyświetlić
        dec     edi             ; teraz EDI = numer linii na ekranie (0-199)
        push    edi             ; numer linii, na której wyświetlić dane
        push    dword 0         ; numer kolumny
        push    dword kolor     ; dane do wyświetlenia
        call    vga_drawscansegment
        add     esp, 4*4

        pop     edi             ; przywróć EDI

        dec     edi             ; zmniejsz numer wyświetlanej linii
        jnz     obrazek         ; jeśli EDI różne od zera, rób kolejne linie

        mov     eax, 6
        int     80h             ; zamknij plik

        mov     eax, 3
        xor     ebx, ebx
        mov     ecx, z
        mov     edx, 1
        int     80h             ; czekamy na klawisz

        push    dword TEXT
        call    vga_setmode     ; ustawiamy tryb tekstowy 80x25
        add     esp, 1*4

        xor     eax, eax        ; kod błędu=0 (brak błędu)
koniec:
        mov     ebx, eax
        mov     eax, 1
```

```
        int     80h             ; wyjście z programu


section ".data" writeable

plik            db      "test.bmp", 0
paleta:         times 768*4 db 0
b               db 0
g               db 0
r               db 0
z               db 0
kolor:          times 320 db 0
```

Mam nadzieję, że kod jest dość jasny. Nawet jeśli znacie asemblera tylko w takim stopniu, w jakim to jest możliwe po przeczytaniu mojego kursu, zrozumienie tego programu nie powinno sprawić Wam kłopotów.

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

14.02.2010

Pisanie boot-sektorów

Gdy już choć średnio znacie assemblera, to po pewnym czasie pojawiają się pytania (mogą one być spowodowane tym, co usłyszeliście na grupach dyskusyjnych lub Waszą własną ciekawością):

1. Co się dzieje, gdy ma zostać uruchomiony jest system operacyjny?
2. Skąd BIOS ma wiedzieć, którą część systemu uruchomić?
3. Jak BIOS odróżnia systemy operacyjne, aby móc je uruchomić?

Odpowiedź na pytanie 2 brzmi: nie wie. Odpowiedź na pytanie 3 brzmi: wcale. Wszystkie Wasze wątpliwości rozwieje odpowiedź na pytanie 1.

Gdy zakończył się POST (Power-On Self Test), wykrywanie dysków i innych urządzeń, BIOS przystępuje do czytania pierwszych sektorów tych urządzeń, na których ma być szukany system operacyjny (u mnie jest ustawiona kolejność: CD-ROM, stacja dyskietek, dysk twardy).
Gdy znajdzie sektor odpowiednio zaznaczony: bajt nr 510 = 55h i bajt 511 = AAh (pamiętajmy, że 1 sektor ma 512 bajtów, a liczymy od zera), to wczytuje go pod adres bezwzględny 07C00h i uruchamia kod w nim zawarty (po prostu wykonuje JMP). Nie należy jednak polegać na tym, że CS = 0 i IP=7C00h (choć najczęściej tak jest).
To właśnie boot-sektor jest odpowiedzialny za ładowanie odpowiednich części właściwego systemu operacyjnego. Na komputerach z wieloma systemami operacyjnymi sprawa też nie jest tak bardzo skomplikowana. Pierwszy sektor dysku twardego, zwany Master Boot Record (MBR), uruchamia program ładujący (tzw. Boot Manager, jak LILO czy GRUB), który z kolei uruchamia boot-sektor wybranego systemu operacyjnego.

My oczywiście nie będziemy operować na dyskach twardych, gdyż byłoby to niebezpieczne. Z dyskietkami zaś można ekperymentować do woli...
A instrukcja jest prosta: umieszczamy nasz programik w pierwszym sektorze dyskietki, zaznaczamy go odpowiednimi ostatnimi bajtami i tyle. No właśnie... niby proste, ale jak o tym pomyśleć to ani to pierwsze, ani to drugie nie jest sprawą banalną.

Do zapisania naszego bootsektorka na dyskietkę możemy oczywiście użyć "gotowców" - programów typu rawwrite itp. Ma to pewne zalety - program był już używany przez dużą liczbę osób, jest sprawdzony i działa. Ale coś by było nie tak, gdybym w kursie programowania w assemblerze kazał Wam używać cudzych programów. Do napisania sowjego własnego programu zapisującego dany plik w pierwszy sektor dyskietki w zupełności wystarczy Wam wiedza uzyskana po przeczytaniu części mojego kursu poświęconej operacjom na plikach wraz z tym (wyciąg oczywiście ze Spisu Przerwań Ralfa Brown'a):

```
INT 13 - DISK - WRITE DISK SECTOR(S)
        AH = 03h
        AL = number of sectors to write (must be nonzero)
        CH = low eight bits of cylinder number
        CL = sector number 1-63 (bits 0-5)
             high two bits of cylinder (bits 6-7, hard disk only)
        DH = head number
        DL = drive number (bit 7 set for hard disk)
        ES:BX -> data buffer
Return: CF set on error
        CF clear if successful
```

Jak widać, sprawa już staje się prosta. Oczywiście, AL=1 (bo zapisujemy 1 sektor), DX=0 (bo stacja ma 2 głowice, a pierwsza ma numer 0, zaś numer dysku 0 wskazuje stację A:), CX=1 (bo numery sektorów zaczynają się od 1, a zapisujemy w pierwszym cylindrze i ma on numer 0).

Tak więc, schemat jest taki:

- Otwórz plik
- Przeczytaj z niego 512 bajtów do pamięci
- Zapisz je na dyskietce
- Zamknij plik

Sprawa jest tak prosta, że tym razem nie podam "gotowca".

Gdy już mamy program zapisujący bootsektor na dyskietkę, trzeba się postarać o to, aby nasz programik (który ma stać się tym bootsektorem) miał dokładnie 512 bajtów i aby 2 ostatnie jego bajty to 55h, AAh. Oczywiście, nie będziemy ręcznie dokładać tylu bajtów, ile trzeba, aby dopełnić nasz program do tych 512. Zrobi to za nas kompilator. Wystarczy po całym kodzie i wszystkich danych, na szarym końcu, umieścić takie coś (NASM/FASM):

```
                times 510 - ($ - start) db 0
                dw 0aa55h
```

Dla TASMa powinno to wyglądać mniej więcej tak:

```
                db 510 - ($ - offset start) dup (0)
                dw 0aa55h
end start
```

To wyrażenie mówi tyle: od bieżącej pozycji w kodzie odejmij pozycję początku kodu (tym samym obliczając długość całego kodu), otrzymaną liczbę odejmij od 510 - i dołóż tyle właśnie bajtów zerowych. Gdy już mamy program długości 510 bajtów, to dokładamy jeszcze znacznik i wszystko jest dobrze.

Jest jednak jeszcze jedna sprawa, o której nie wspomniałem - ustawienie DS i wartości "org" dla naszego kodu. Otóż, jeśli stwierdzimy, że nasz kod powinien zaczynać się od offsetu 0 w naszym segmencie, to ustawmy sobie "org 0" i DS=07C0h (tak, ilość zer się zgadza), ale możemy też mieć "org 7C00h" i DS=0. Żadne z tych nie wpływa w żaden sposób na długość otrzymanego programu, a należy o to zadbać, gdyż nie mamy gwarancji, że DS będzie pokazywał na nasze dane po uruchomieniu bootsektora.

Teraz, uzbrojeni w niezbędną wiedzę, zasiadamy do pisania kodu naszego bootsektora. Nie musi to być coś wielkiego - tutaj pokażę coś, co w lewym górnym rogu ekranu pokaże cyfrę "1" (o bezpośredniej manipulacji ekranem możecie przeczytać w moim innym artykule) i po naciśnięciu dowolnego klawisza zresetuje komputer (na jeden ze sposobów podanych w jeszcze innym artykule...).
Oto nasz kod (NASM):

```
; nasm -o boot.bin -f bin boot.asm

org 7c00h                               ; lub "org 0"

start:
        mov     ax, 0b800h
        mov     es, ax                  ; ES = segment pamięci ekranu

        mov     byte [es:0], '1'        ; piszemy '1'

        xor     ah, ah
        int     16h                     ; czekamy na klawisz

        mov     bx, 40h
```

```
        mov     ds, bx
        mov     word [ds:72h], 1234h    ; 40h:72h = 1234h - wybieramy gorący reset
        jmp     0ffffh:0000h            ; reset

times 510 - ($ - start) db 0            ; dopełnienie do 510 bajtów
dw 0aa55h                               ; znacznik
```

Nie było to długie ani trudne, prawda? Rzecz jasna, nie można w bootsektorach używać żadnych przerwań systemowych, np. DOSowego int 21h, bo żaden system po prostu nie jest uruchomiony i załadowany. Tak napisany programik kompilujemy do formatu binarnego. W TASMie wyglądałoby to jakoś tak (po dodaniu w programie dyrektyw ".model tiny", ".code", ".8086" i "end start"):

```
tasm bootsec1.asm
tlink bootsec1.obj,bootsec1.bin /t
```

Po kompilacji umieszczamy go na dyskietce przy użyciu programu napisanego już przez nas wcześniej. Resetujemy komputer (i upewniamy się, że BIOS spróbuje uruchomić system z dyskietki), wkładamy dyskietkę i.... cieszymy się swoim dziełem (co prawda ta jedynka będzie mało widoczna, ale rzeczywiście znajduje się na ekranie).

Zauważcie też, że ani DOS ani Windows nie rozpoznaje już naszej dyskietki, mimo iż przedtem była sformatowana. Dzieje się tak dlatego, że w bootsektorze umieszczane są informacje o dysku. "Prawidłowy" DOSowy/Windowsowy bootsektor powinien się zaczynać tak:

```
org 7c00h                       ; lub "org 0", oczywiście

start:
        jmp short kod
        nop

;=================================

        db '        '           ; nazwa OS i wersja OEM (8B)
        dw 512                  ; bajtów/sektor (2B)
        db 1                    ; sektory/jednostkę alokacji (1B)
        dw 1                    ; zarezerwowane sektory (2B)
        db 2                    ; liczba tablic alokacji (1B)
        dw 224                  ; liczba pozycji w katalogu głównym (2B), zwykle 224
        dw 2880                 ; liczba sektorów (2B)
        db 0f0h                 ; Media Descriptor Byte (1B)
        dw 9                    ; sektory/FAT (2B)
        dw 18                   ; sektory/ścieżkę (2B)
        dw 2                    ; liczba głowic (2B)
        dd 0                    ; liczba ukrytych sektorów (4B)
        dd 0                    ; liczba sektorów (część 2), jeśli wcześniej 0 (4B)
        db 0                    ; numer dysku (1B)
        db 0                    ; zarezerw. (1B)
        db 0                    ; rozszerzona sygnatura bloku ładującego (1B)
        dd 0bbbbbddddh          ; numer seryjny dysku (4B)
        db '           '        ; etykieta (11B)
        db 'FAT 12  '           ; typ FAT (8B), zwykle "FAT 12  "

;=================================

kod:
        ; tutaj dopiero kod bootsektora
```

Ta porcja oczywiście uszczupla ilość kodu, którą można umieścić w bootsektorze. Nie jest to jednak duży problem, gdyż i tak jedyną rolą większości bootsektorów jest uruchomienie innych programów (tzw. second stage boot-loaders), które dopiero zajmują się ładowaniem właściwego systemu. Jeszcze ciekawostka: co wypisuje BIOS, gdy dysk jest niewłaściwy (tj. niesystemowy)?
Otóż - nic! BIOS bardzo chętnie przeszedłby do kolejnego urządzenia.
Dlaczego więc tego nie robi i skąd ten napis o niewłaściwym dysku systemowym??
Odpowiedź jest prosta - sformatowana dyskietka posiada bootsektor!
Dla BIOSu jest wszystko OK, uruchamia więc ten bootsektor. Dopiero ten wypisuje informację o niewłaściwym dysku, czeka na naciśnięcie klawisza, po czym uruchamia int 19h. O tym, co robi przerwanie 19h możecie przeczytać w artykule o resetowaniu.

Miłego bootowania systemu!
P.S. Jeśli nie chcecie przy najdrobniejszej zmianie kodu resetować komputera, możecie poszukać w Internecie programów, które symulują procesor (w tym fazę ładowania systemu). Jednym z takich programów jest Bochs, który znajdziecie tu: http://bochs.sourceforge.net/.

14.02.2010

Pisanie boot-sektorów pod Linuksem

Gdy już choć średnio znacie asemblera, to po pewnym czasie pojawiają się pytania (mogą one być spowodowane tym, co usłyszeliście lub Waszą własną ciekawością):

1. Co się dzieje, gdy ma zostać uruchomiony jest system operacyjny?
2. Skąd BIOS ma wiedzieć, którą część systemu uruchomić?
3. Jak BIOS odróżnia systemy operacyjne, aby móc je uruchomić?

Odpowiedź na pytanie 2 brzmi: nie wie. Odpowiedź na pytanie 3 brzmi: wcale. Wszystkie Wasze wątpliwości rozwieje odpowiedź na pytanie 1.

Gdy zakończył się POST (Power-On Self Test), wykrywanie dysków i innych urządzeń, BIOS przystępuje do czytania pierwszych sektorów tych urządzeń, na których ma być szukany system operacyjny (u mnie jest ustawiona kolejność: CD-ROM, stacja dyskietek, dysk twardy).
Gdy znajdzie sektor odpowiednio zaznaczony: bajt nr 510 = 55h i bajt 511 = AAh (pamiętajmy, że 1 sektor ma 512 bajtów, a liczymy od zera), to wczytuje go pod adres bezwzględny 07C00h i uruchamia kod w nim zawarty (po prostu wykonuje skok pod ten adres). Nie należy jednak polegać na tym, że segment kodu CS = 0, a adres instrukcji IP=7C00h (choć najczęściej tak jest).
To właśnie boot-sektor jest odpowiedzialny za ładowanie odpowiednich części właściwego systemu operacyjnego. Na komputerach z wieloma systemami operacyjnymi sprawa też nie jest tak bardzo skomplikowana. Pierwszy sektor dysku twardego, zwany Master Boot Record (MBR), zawiera program ładujący (Boot Manager, jak LILO czy GRUB), który z kolei uruchamia boot-sektor wybranego systemu operacyjnego.

My oczywiście nie będziemy operować na dyskach twardych, gdyż byłoby to niebezpieczne. Z dyskietkami zaś można eksperymentować do woli...
A instrukcja jest prosta: umieszczamy nasz programik w pierwszym sektorze dyskietki, zaznaczamy go odpowiednimi ostatnimi bajtami i tyle. No właśnie... niby proste, ale jak o tym pomyśleć to ani to pierwsze, ani to drugie nie jest sprawą banalną.

Do zapisania naszego bootsektorka na dyskietkę możemy oczywiście użyć gotowców - programów typu `dd` itp. Ma to pewne zalety - program był już używany przez dużą liczbę osób, jest sprawdzony i działa.
Przykładowy sposób użycia (po skompilowaniu bootsektora):

```
dd count=1 if=boot.bin of=/dev/fd0
```

Ale coś by było nie tak, gdybym w kursie programowania w asemblerze kazał Wam używać cudzych programów. Do napisania swojego własnego programu zapisującego dany plik w pierwszym sektorze dyskietki w zupełności wystarczy Wam wiedza uzyskana po przeczytaniu części mojego kursu poświęconej operacjom na plikach.
Schemat działania jest taki:

- Otwórz plik zawierający skompilowany bootsektor
- Przeczytaj z niego 512 bajtów (do zadeklarowanej tablicy w pamięci)
- Zamknij ten plik

- Otwórz plik /dev/fd0 do zapisu
- Zapisz do niego wcześniej odczytane dane
- Zamknij ten plik

Sprawa jest tak prosta, że tym razem nie podam gotowca.

Gdy już mamy program zapisujący bootsektor na dyskietkę, trzeba się postarać o to, aby nasz programik (który ma stać się tym bootsektorem) miał dokładnie 512 bajtów i aby 2 ostatnie jego bajty to 55h, AAh. Oczywiście, nie będziemy ręcznie dokładać tylu bajtów, ile trzeba, aby dopełnić nasz program do tych 512. Zrobi to za nas kompilator. Wystarczy po całym kodzie i wszystkich danych, na szarym końcu, umieścić takie coś:

```
times 510 - ($ - start) db 0
dw 0aa55h
```

To wyrażenie mówi tyle: od bieżącej pozycji w kodzie odejmij pozycję początku kodu (tym samym obliczając długość całego kodu), otrzymaną liczbę odejmij od 510 - i dołóż tyle właśnie bajtów zerowych. Gdy już mamy program długości 510 bajtów, to dokładamy jeszcze znacznik i wszystko jest dobrze.

Jest jednak jeszcze jedna sprawa, o której nie wspomniałem - ustawienie DS i wartości ORG dla naszego kodu. Otóż, jeśli stwierdzimy, że nasz kod powinien zaczynać się od offsetu 0 w naszym segmencie, to ustawmy sobie org 0 i DS=07C0h (tak, ilość zer się zgadza), ale możemy też mieć org 7C00h i DS=0. Żadne z tych nie wpływa w żaden sposób na długość otrzymanego programu, a należy o to zadbać, gdyż nie mamy gwarancji, że DS będzie pokazywał na nasze dane po uruchomieniu bootsektora.

Teraz, uzbrojeni w niezbędną wiedzę, zasiadamy do pisania kodu naszego bootsektora. Nie musi to być coś wielkiego - tutaj pokażę coś, co w lewym górnym rogu ekranu pokaże cyfrę jeden (o bezpośredniej manipulacji ekranem możecie przeczytać w moim innym artykule) i po naciśnięciu dowolnego klawisza zresetuje komputer (na jeden ze sposobów podanych w jeszcze innym artykule...).
Oto nasz kod (NASM):
(przeskocz przykładowy bootsektor)

```
; nasm -o boot.bin -f bin boot.asm

org 7c00h                       ; lub     "org 0"

start:
        mov     ax, 0b800h
        mov     es, ax          ; ES = segment pamięci ekranu

        mov     byte [es:0], "1" ; piszemy "1"

        xor     ah, ah
        int     16h             ; czekamy na klawisz

        mov     bx, 40h
        mov     ds, bx
        mov     word [ds:72h], 1234h    ; 40h:72h = 1234h -
                                        ; wybieramy gorący reset

        jmp     0ffffh:0000h            ; reset
```

```
times 510 - ($ - start) db 0            ; dopełnienie do 510 bajtów
dw 0aa55h                               ; znacznik
```

Nie było to długie ani trudne, prawda? Rzecz jasna, nie można w bootsektorach używać żadnych funkcji systemowych, np. linuksowego int 80h, bo żaden system po prostu nie jest uruchomiony i załadowany. Tak napisany programik kompilujemy do formatu binarnego. Po kompilacji umieszczamy go na dyskietce przy użyciu programu napisanego już przez nas wcześniej. Resetujemy komputer (i upewniamy się, że BIOS spróbuje uruchomić system z dyskietki), wkładamy dyskietkę i.... cieszymy się swoim dziełem (co prawda ta jedynka może być mało widoczna, ale rzeczywiście znajduje się na ekranie).

Zauważcie też, że system nie rozpoznaje już naszej dyskietki, mimo iż przedtem była sformatowana. Dzieje się tak dlatego, że w bootsektorze umieszczane są informacje o dysku.
Bootsektor typu FAT12 (vfat) powinien się zaczynać mniej-więcej tak:
(przeskocz systemowy obszar bootsektora)

```
org 7c00h                       ; lub org 0, oczywiście

start:
        jmp short kod
        nop

        db "         "   ; nazwa OS i wersja OEM (8B)
        dw 512           ; bajtów/sektor (2B)
        db 1             ; sektory/jednostkę alokacji (1B)
        dw 1             ; zarezerwowane sektory (2B)
        db 2             ; liczba tablic alokacji (1B)
        dw 224           ; liczba pozycji w katalogu głównym (2B)
                         ; 224 to typowa wartość
        dw 2880          ; liczba sektorów (2B)
        db 0f0h          ; Media Descriptor Byte (1B)
        dw 9             ; sektory/FAT (2B)
        dw 18            ; sektory/ścieżkę (2B)
        dw 2             ; liczba głowic (2B)
        dd 0             ; liczba ukrytych sektorów (4B)
        dd 0             ; liczba sektorów (część 2),
                         ; jeśli wcześniej było 0 (4B)
        db 0             ; numer dysku (1B)
        db 0             ; zarezerwowane (1B)
        db 0             ; rozszerzona sygnatura bloku ładującego
        dd 0bbbbbddddh   ; numer seryjny dysku (4B)
        db "           "; etykieta (11B)
        db "FAT 12  "    ; typ FAT (8B), zwykle  "FAT 12  "

kod:
        ; tutaj dopiero kod bootsektora
```

Ta porcja danych oczywiście uszczupla ilość kodu, którą można umieścić w bootsektorze. Nie jest to jednak duży problem, gdyż i tak jedyną rolą większości bootsektorów jest uruchomienie innych programów (second stage bootloaders), które dopiero zajmują się ładowaniem właściwego systemu.

Jeszcze ciekawostka: co wypisuje BIOS, gdy dysk jest niewłaściwy (niesystemowy)?
Otóż - nic! BIOS bardzo chętnie przeszedłby do kolejnego urządzenia.
Dlaczego więc tego nie robi i skąd ten napis o niewłaściwym dysku systemowym??
Odpowiedź jest prosta - sformatowana dyskietka posiada bootsektor!
Dla BIOSu jest wszystko OK, uruchamia więc ten bootsektor. Dopiero ten wypisuje informację o niewłaściwym dysku, czeka na naciśnięcie klawisza, po czym uruchamia int 19h. O tym, co robi przerwanie 19h możecie przeczytać w artykule o resetowaniu.

Miłego bootowania systemu!
P.S. Jeśli nie chcecie przy najdrobniejszej zmianie kodu resetować komputera, możecie poszukać w Internecie programów, które symulują procesor (w tym fazę ładowania systemu). Jednym z takich programów jest Bochs.

Co dalej?

Mimo iż bootsektor jest ograniczony do 512 bajtów, to może w dość łatwy sposób posłużyć do wczytania do pamięci o wiele większych programów. Wystarczy użyć funkcji czytania sektorów. W spisie Przerwań Ralfa Brown'a czytamy:
(przeskocz opis int 13h, ah=2)

```
INT 13 - DISK - READ SECTOR(S) INTO MEMORY
        AH = 02h
        AL = number of sectors to read (must be nonzero)
        CH = low eight bits of cylinder number
        CL = sector number 1-63 (bits 0-5)
             high two bits of cylinder (bits 6-7, hard disk only)
        DH = head number
        DL = drive number (bit 7 set for hard disk)
        ES:BX -> data buffer
Return: CF set on error
        CF clear if successful
```

Wystarczy więc wybrać nieużywany segment pamięci, np. ES=8000h i począwszy od offsetu BX=0, czytać sektory zawierające nasz kod, zwiększając BX o 512 za każdym razem. Kod do załadowania nie musi być oczywiście w postaci pliku na dyskietce, to by tylko utrudniło pracę (gdyż trzeba wtedy czytać tablicę plików FAT). Najłatwiej załadować kod tym samym sposobem, co bootsektor, ale oczywiście do innych sektorów. Najłatwiej to zrobić, sklejając skompilowany plik bootsektora ze skompilowanym kodem i potem nagrać na dyskietkę:

```
cat boot.bin system.bin > wszystko.bin
dd if=wszystko.bin of=/dev/fd0
```

Po załadowaniu całego potrzebnego kodu do pamięci przez bootsektor, wystarczy wykonać skok:

```
jmp     8000h:0000h
```

Wtedy kontrolę przejmuje kod wczytany z dyskietki.

Ale jest jeden kruczek - trzeba wiedzieć, jakie numery cylindra, głowicy i sektora podać do funkcji czytające sektory, żeby rzeczywiście odczytała te właściwe.
Struktura standardowej dyskietki jest następująca: 512 bajtów na sektor, 18 sektorów na ścieżkę, 2 ścieżki na cylinder (bo są dwie strony dyskietki, co daje 36 sektorów na cylinder), 80 cylindrów na głowicę. Razem 2880 sektorów po 512 bajtów, czyli 1.474.560 bajtów.
Mając numer sektora (bo wiemy, pod jakimi sektorami zapisaliśmy swój kod na dyskietce), odejmujemy od niego 1 (tak by zawsze wszystkie numery sektorów zaczynały się od zera), po czym dzielimy go przez 36.

Uzyskany iloraz to numer cylindra (rejestr CH), reszta zaś oznacza numer sektora w tymże cylindrze (rejestr CL). Jeśli ta reszta jest większa bądź równa 18, należy wybrać głowicę numer 1 (rejestr DH), zaś od numeru sektora (rejestr CL) odjąć 18. W przeciwnym przypadku należy wybrać głowicę numer 0 i nie robić nic z numerem sektora.
W ten sposób otrzymujemy wszystkie niezbędne dane i możemy bez przeszkód w pętli czytać kolejne sektory zawierające nasz kod.
Całą tę procedurę ilustruje ten przykładowy kod:
(przeskocz procedurę czytania sektorów)

```
secrd:
;wejście: ax=sektor, es:bx -> dane

        dec ax          ; z numerów 1-36 na 0-35
        mov cl,36       ; liczba sektorów na cylinder = 36
        xor dx,dx       ; zakładamy na początek: głowica 0, dysk 0 (a:)
        div cl          ; AX (numer sektora) dzielimy przez 36
        mov ch,al       ; AL=cylinder, AH=przesunięcie względem
                        ;       początku cylindra, czyli sektor
        cmp ah,18       ; czy numer sektora mniejszy od 18?
        jb .sec_ok      ; jeśli tak, to nie robimy nic
        sub ah,18       ; jeśli nie, to odejmujemy 18
        inc dh          ; i zmieniamy głowicę
.sec_ok:
        mov cl, ah      ; CL = numer sektora
        mov ax,0201h    ; odczytaj 1 sektor
        inc cl          ; zwiększ z powrotem z zakresu 0-17 do 1-18

        push dx         ; niektóre biosy niszczą DX, nie ustawiają
                        ;       flagi CF, lub zerują flagę IF
        stc
        int 13h         ; wykonaj czytanie
        sti
        pop dx
```

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Rozpoznawanie typu procesora

(przeskocz wykrywanie procesora)

Jak zapewne wiecie, wiele programów (systemy operacyjne, gry, ...) potrafi jakoś dowiedzieć się, na jakim procesorze zostały uruchomione. Rozpoznanie typu procesora umożliwia np. uruchomienie dodatkowych optymalizacji w programie lub odmowę dalszego działania, jeśli program musi korzystać z instrukcji niedostępnych na danym procesorze.
Wykrywanie rodzaju CPU i FPU nie jest trudne i pokażę teraz, jak po kolei sprawdzać typ procesora (nie można przecież zacząć sprawdzania od najwyższych).
Informacje, które tutaj podam, są oczywiście słuszne dla wszystkich procesorów rodziny x86 (AMD, Cyrix, ...), a nie tylko Intela.
Generalnie sposób wykrywania pod Linuksem jest jeden (nie wliczając żadnych dodatkowych funkcji jądra czy też czytania z /proc/cpuinfo): poprzez rejestr E-FLAG.

1. 80386
   (przeskocz 80386)
   Na tym procesorze nie można zmienić bitu numer 18 we flagach (wiemy, że rejestr flag ma 32 bity). Bit ten odpowiada za Alignment Check i spowoduje przerwanie m.in wtedy, gdy SP nie będzie podzielne przez 4. Dlatego, zanim będziemy testować ten bit, musimy zachować SP i wyzerować jego najmłodsze 2 bity.
   (przeskocz kod dla 80386)

```
mov     dx, sp
and     sp, ~3          ; aby uniknąć AC fault.
                        ; FASM: and sp, not 3
pushfd                  ; flagi na stos
pop     eax             ; EAX = E-flagi
mov     ecx, eax        ; zachowanie EAX
xor     eax, 40000h     ; zmiana bitu 18
push    eax             ; EAX na stos
popfd                   ; E-flagi = EAX
pushfd                  ; flagi na stos
pop     eax             ; EAX = flagi
xor     eax, ecx  ; czy takie same? jeśli tak, to 386
mov     sp, dx          ; przywrócenie SP
jz      jest_386
```

2. 80486
   (przeskocz 80486)
   Na tym procesorze nie można zmienić bitu 21 we flagach. Jeśli ten bit można zmienić, to procesor obsługuje instrukcję CPUID, której będziemy używać do dalszego rozpoznania. Kod:
   (przeskocz kod dla 80486)

```
pushfd                  ; flagi na stos
pop     eax             ; EAX = E-flagi
mov     ecx, eax        ; zachowanie EAX
xor     eax, 200000h    ; zmiana bitu 21
push    eax             ; EAX na stos
popfd                   ; E-flagi = EAX
pushfd                  ; flagi na stos
pop     eax             ; EAX = flagi
xor     eax, ecx  ; czy takie same? jeśli tak, to 486
jz      jest_486
jmp     jest_586
```

Zanim omówię sposób korzystania z instrukcji CPUID, zajmijmy się sposobem rozpoznania typu koprocesora.

Koprocesor

(przeskocz wykrywanie koprocesora)

Tutaj możliwości są tylko 4: brak koprocesora, 8087, 80287, 80387. No to do roboty.

1. czy w ogóle jest jakiś koprocesor?
   (przeskocz test na istnienie FPU)
   To sprawdzamy bardzo łatwo. Jeśli nie ma koprocesora, to w chwili wykonania instrukcji FPU może wystąpić przerwanie INT6 (nieprawidłowa instrukcja), ale nie o tym sposobie chciałem powiedzieć. Koprocesor można wykryć, jeśli słowo stanu zostanie zapisane prawidłowo. Oto kod:
   (przeskocz test na istnienie FPU)

```
fninit                  ; inicjalizacja zeruje rejestry

; wpisujemy jakąś niezerowa wartość:
mov     word [_fpu_status], 5a5ah

; zapisz słowo statusowe do pamięci:
fnstsw  [_fpu_status]
mov     ax, [_fpu_status]
or      al, al  ; jeśli zapisało dobrze (zera oznaczają
                ; puste rejestry), to jest FPU

jz      jest_FPU
```

2. 8087
   (przeskocz 8087)
   Sztuczka polega na wykorzystaniu instrukcji FDISI (wyłączenie przerwań), która rzeczywiście coś robi tylko na 8087. Po wyłączeniu przerwań w słowie kontrolnym zostaje włączony bit numer 7.
   (przeskocz kod dla 8087)

```
; zachowaj słowo kontrolne do pamięci:
fnstcw  [_fpu_status]

; wyłączamy wszystkie
; przerwania (poprzez słowo kontrolne):
and     word [_fpu_status], 0ff7fh

; załaduj słowo kontrolne z pamięci:
fldcw   [_fpu_status]

fdisi           ; wyłączamy wszystkie przerwania
                ; (jako instrukcja)

; zachowaj słowo kontrolne do pamięci:
fstcw   [_fpu_status]
test    byte [_fpu_status], 80h ; bit 7 ustawiony?

jz      nie_8087        ; jeśli nie, to nie jest to 8087
```

3. 80287

(przeskocz 80287)
Koprocesor ten nie odróżnia minus nieskończoności od plus nieskończoności. Kod na sprawdzenie tego wygląda tak:
(przeskocz kod dla 80287)

```
finit

fld1            ; st(0)=1
fldz            ; st(0)=0,st(1)=1
fdivp   st1     ; tworzymy nieskończoność,
                ; dzieląc przez 0
fld     st0     ; st(1):=st(0)=nieskończoność
fchs            ; st(0)= minus nieskończoność

                ; porównanie st0 z st1 i
                ; zdjęcie obu ze stosu
fcompp          ; 8087/287: -niesk. = +niesk.,
                ; 387: -niesk. != +niesk.

fstsw   [_fpu_status]   ; zapisz status do pamięci
mov     ax, [_fpu_status]       ; AX = status

sahf            ; zapisz AH we flagach. tak sie składa,
                ; że tutaj również flaga ZF wskazuje na
                ; równość argumentów.
jz      jest_287
jmp     jest_387
```

Dalsze informacje o procesorze - instrukcja CPUID

Od procesorów 586 (choć niektóre 486 też podobno ją obsługiwały), Intel i inni wprowadzili instrukcję CPUID. Pozwala ona odczytać wiele różnych informacji o procesorze (konkretny typ, rozmiary pamięci podręcznych, dodatkowe rozszerzenia, ...).
Korzystanie z tej instrukcji jest bardzo proste: do EAX wpisujemy numer (0-3) i wywołujemy instrukcję, np.

```
mov     eax, 1
cpuid
```

Teraz omówię, co można dostać przy różnych wartościach EAX.

1. EAX=0
   (przeskocz EAX=0)
   EAX = maksymalny numer funkcji dla CPUID.
   EBX:EDX:ECX = marka procesora (12 znaków ASCII).
   Intel - GenuineIntel
   AMD - AuthenticAMD
   NexGen - NexGenDriven
   Cyrix, VIA - CyrixInstead
   RISE - RiseRiseRise,
   Centaur Technology/IDT - CentaurHauls (programowalne, może być inne)

United Microelectronics Corporation - UMC UMC UMC
Transmeta Corporation - GenuineTMx86
SiS - SiS SiS SiS
National Semiconductor - Geode by NSC.

2. EAX=1
(przeskocz EAX=1)
EAX = informacje o wersji:
   - bity 0-3: stepping ID
   - bity 4-7: model
   - bity 8-11: rodzina. Wartości mogą być od 4 (80486) do 7 (Itanium) oraz 15 (co znaczy sprawdź rozszerzone informacje o rodzinie)
   - bity 12-13: typ procesora (0=Original OEM Processor, 1=Intel Overdrive, 2=Dual)
   - bity 16-19 (jeśli jest taka możliwość): rozszerzona informacja o modelu.
   - bity 20-27 (jeśli jest taka możliwość): rozszerzona informacja o rodzinie.

EDX = cechy procesora (tutaj akurat z procesorów Intela; najpierw numery bitów):
   - 0: procesor zawiera FPU
   - 1: Virtual 8086 Mode Enchancements
   - 2: Debugging Extensions
   - 3: Page Size Extension
   - 4: Time Stamp Counter
   - 5: Model Specific Registers
   - 6: Physical Address Extensions
   - 7: Machine Check Exception
   - 8: instrukcja CMPXCHG8B
   - 9: procesor zawiera Zaawansowany Programowalny Kontroler Przerwań (APIC)
   - 11: instrukcje SYSENTER i SYSEXIT
   - 12: Memory Type Range Registers
   - 13: Page Table Entries Global Bit
   - 14: Machine Check Architecture
   - 15: instrukcje CMOV*
   - 16: Page Attribute Table
   - 17: 32-bit Page Size Extensions
   - 18: numer seryjny procesora
   - 19: instrukcja CLFLUSH
   - 21: Debug Store
   - 22: monitorowanie temperatury i możliwość modyfikacji wydajności procesora
   - 23: technologia MMX
   - 24: instrukcje FXSAVE i FXRSTOR
   - 25: technologia SSE
   - 26: technologia SSE2
   - 27: Self-Snoop
   - 28: technologia Hyper-Threading
   - 29: monitorowanie temperatury, układy kontroli temperatury
   - 31: Pending Break Enable

3. EAX=2
EBX, ECX, EDX = informacje o pamięci podręcznej cache i TLB

Nawet te informacje, które tu przedstawiłem są już bardzo szczegółowe i z pewnością nie będą takie same na wszystkich procesorach. To jest tylko wstęp. Dalsze informacje można znaleźć na stronach producentów procesorów, np. AMD, Intel, ale także tutaj: Sandpile, Lista przerwań Ralfa Brown'a (plik opcodes.lst)

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Pobieranie i ustawianie daty oraz godziny pod Linuksem

Bieżąca data i godzina w systemie Linux jest przechowywana w postaci tzw. znacznika czasu. Jest to liczba oznaczająca ilość sekund, które upłynęły od pierwszego dnia stycznia roku 1970, od północy czasu UTC (GMT). Znacznik czasu pobierany jest funkcją sys_time (numer 13), a nowy czas można ustawić za pomocą funkcji sys_stime (numer 25).
Jeśli nie chcemy korzystać z żadnych bibliotek, to ta forma jest dość niewygodna w użyciu. Dlatego przedstawię tu sposoby przerabiania znacznika czasu na formę tradycyjną i odwrotnie.

Artykuł ten opracowałem na podstawie kodu źródłowego linuksowej biblioteki języka C (glibc), konkretnie - na podstawie pliku glibc/time/offtime.c.

Zmiana znacznika czasu na formę tradycyjną

(przeskocz do konwersji w drugą stronę)

1. dzielimy znacznik przez liczbę sekund przypadającą na dzień (60*60*24), zachowujemy iloraz jako liczbę dni oraz resztę z tego dzielenia
2. do reszty dodajemy przesunięcie naszego czasu od GMT, w sekundach (60*60 w czasie zimowym, 2*60*60 w czasie letnim)
3. jeśli reszta jest mniejsza od zera, to dodajemy do niej liczbę sekund przypadających na dzień, aż stanie się większa od zera, za każdym razem zmniejszając liczbę dni z pierwszego kroku
4. jeśli reszta jest większa od liczby sekund dnia, to odejmujemy do niej liczbę sekund przypadających na dzień, aż stanie się mniejsza od tej liczby, za każdym razem zwiększając liczbę dni z pierwszego kroku
5. dzielimy resztę przez liczbę sekund przypadającą na godzinę. Iloraz zachowujemy jako obliczoną godzinę, resztę zapisujemy do zmiennej, którą dalej nazywamy resztą
6. resztę z poprzedniego kroku dzielimy przez liczbę sekund w minucie. Iloraz zachowujemy jako liczbę minut bieżącego czasu, resztę - jako liczbę sekund
7. do liczby dni dodajemy 4 (jako że 1-szy stycznia 1970 był czwartkiem), a wynik dzielimy przez 7. Resztę (jeśli jest ujemna, dodajemy 7) z tego dzielenia zachowujemy jako dzień tygodnia (0 oznacza niedzielę)
8. do zmiennej Y wstawiamy 1970
9. w pętli wykonuj działania:
    1. sprawdź, czy liczba dni jest mniejsza od zera lub większa od liczby dni w roku Y. Jeśli nie zachodzi ani to, ani to, wyjdź z pętli.
       W tym kroku należy sprawdzić, czy Y jest przestępny. Każdy rok, który dzieli się przez 4, lecz nie dzieli się przez 100 jest przestępny. Dodatkowo, każdy rok, który dzieli się przez 400, jest przestępny.
    2. do nowej zmiennej YG wstaw sumę Y oraz ilorazu z dzielenia liczby dni przez 365. Jeśli reszta z dzielenia liczby dni przez 365 była ujemna, od YG odejmij jeden.
    3. od liczby dni odejmij różnicę między YG a Y pomnożoną przez 365
    4. od liczby dni odejmij wynik procedury DODATEK (omówiona później), wykonanej na liczbie YG-1

5. do liczby dni dodaj wynik procedury DODATEK (omówiona później), wykonanej na liczbie Y-1
6. do Y wstaw YG

10. do numeru dnia w roku wstaw bieżącą wartość liczby dni
11. sprawdź, w którym miesiącu znajduje się dzień o tym numerze i zapisz ten miesiąc. Od liczby dni odejmij sumaryczną liczbę dni we wszystkich poprzednich miesiącach.
12. do dnia miesiąca wstaw bieżącą liczbę dni powiększoną o 1

Procedura DODATEK składa się z kroków:

1. podziel podany rok przez 4 i zachowaj wynik. Jeśli reszta wyszła mniejsza od zera, od wyniku odejmij 1
2. podziel podany rok przez 100 i zachowaj wynik. Jeśli reszta wyszła mniejsza od zera, od wyniku odejmij 1
3. podziel podany rok przez 400 i zachowaj wynik. Jeśli reszta wyszła mniejsza od zera, od wyniku odejmij 1
4. od pierwszego wyniku odejmij drugi, po czym dodaj trzeci, a całość zwróć jako wynik procedury

Cały ten skomplikowany algorytm jest ukazany w tym oto programie (składnia FASM):
(przeskocz program)

```
; Program wyliczający bieżącą datę i godziną na podstawie bieżącego
;       znacznika czasu. Program NIC NIE WYŚWIETLA.
;
; Autor: Bogdan D., bogdandr (at) op.pl
;
; kompilacja:
;   fasm dataczas.fasm

format ELF executable
segment executable
entry main

SEK_NA_GODZ     = (60 * 60)             ; liczba sekund w godzinie
SEK_NA_DZIEN    = (SEK_NA_GODZ * 24)    ; liczba sekund w dobie
LETNI           = 1                     ; 0, gdy zimowy, 1 gdy letni
PRZES_GMT       = 1*SEK_NA_GODZ + LETNI*SEK_NA_GODZ  ; przesunięcie od GMT

main:
        mov     eax, 13
        xor     ebx, ebx
        int     80h     ; pobierz aktualny czas w sekundach
        mov     [czas], eax

        mov     ebx, SEK_NA_DZIEN
        xor     edx, edx
        idiv    ebx     ; liczba sekund / liczba sekund w dniu = liczba dni

        add     edx, PRZES_GMT  ; dodaj strefę czasową

        ; jeśli reszta sekund < 0, dodajemy do niej liczbę sekund dnia,
        ; ale równocześnie zmniejszamy liczbę dni (EAX)
spr_reszte:
        cmp     edx, 0
        jge     reszta_ok
```

```
        add     edx, SEK_NA_DZIEN
        sub     eax, 1

        jmp     spr_reszte

reszta_ok:

        ; jeśli reszta sekund > liczba sekund w dniu, odejmujemy od niej
        ; liczbę sekund dnia, ale równocześnie zwiększamy liczbę dni (EAX)
spr_reszte2:
        cmp     edx, SEK_NA_DZIEN
        jl      reszta_ok2

        sub     edx, SEK_NA_DZIEN
        add     eax, 1

        jmp     spr_reszte2

reszta_ok2:

        mov     [l_dni], eax
        mov     [reszta], edx

        mov     eax, edx        ; EAX = reszta
        mov     ebx, SEK_NA_GODZ
        xor     edx, edx
        idiv    ebx     ; EAX = numer godziny, reszta - minuty+sekundy

        mov     [godz], al      ; zachowujemy godzinę
        mov     [reszta], edx   ; i nową resztę

        mov     eax, edx
        mov     ecx, 60
        xor     edx, edx
        idiv    ecx             ; nową resztę dzielimy przez 60

        mov     [min], al       ; iloraz to liczba minut
        mov     [sek], dl       ; a reszta - liczba sekund

        ; znajdujemy dzień tygodnia
        mov     eax, [l_dni]
        add     eax, 4  ; 1970-1-1 to czwartek
        mov     ebx, 7
        xor     edx, edx
        idiv    ebx     ; EAX = dzień tygodnia

        cmp     dl, 0
        jge     dzient_ok
        add     dl, 7   ; dodajemy 7, jeśli był mniejszy od zera
dzient_ok:
        mov     [dzient], dl


        ; początek pętli z punktu 9
spr_dni:
        mov     eax, [y]
        call    czy_przest      ; ECX = 0, gdy Y jest przestępny.

        cmp     dword [l_dni], 0
        jl      zmien_dni       ; sprawdzamy, czy liczba dni < 0
```

```
        mov     esi, 365
        test    ecx, ecx
        jnz     .przest_ok
        add     esi, 1          ; dodajemy 1 dzień w roku przestępnym
.przest_ok:

        cmp     [l_dni], esi
        jl      koniec_spr_dni  ; sprawdzamy, czy liczba dni >= 365/366

zmien_dni:

        mov     esi, 365
        mov     eax, [l_dni]
        xor     edx, edx
        idiv    esi             ; EAX = liczba dni/365
        mov     ecx, eax        ; zachowujemy do ECX
        cmp     edx, 0
        jge     .edx_ok1
        sub     ecx, 1          ; jeśli reszta < 0, to odejmujemy 1
.edx_ok1:
        add     ecx, [y]        ; ECX = liczba dni/365 + Y +1 lub +0
        mov     [yg], ecx       ; zachowaj do YG

        sub     ecx, [y]
        imul    ecx, ecx, 365   ; ECX = (YG-Y)*365

        push    ecx
        mov     eax, [yg]
        sub     eax, 1
        call    dodatek         ; wylicz DODATEK na YG-1 i zapisz w [przest]
        pop     ecx
        add     ecx, [przest]   ; ECX = (YG-Y)*365+DODATEK(YG-1)

        push    ecx
        mov     eax, [y]
        sub     eax, 1
        call    dodatek         ; wylicz DODATEK na Y-1 i zapisz w [przest]
        pop     ecx
        sub     ecx, [przest]   ; ECX=(YG-Y)*365+DODATEK(YG-1)-DODATEK(Y-1)

        sub     [l_dni], ecx    ; odejmij całość na raz od liczby dni

        mov     eax, [yg]
        mov     [y], eax        ; do Y wstaw YG

        jmp     spr_dni         ; i na początek pętli

koniec_spr_dni:
        mov     eax, [y]
        ;sub    eax, 1900
        mov     [rok], ax       ; zapisz wyliczony rok
        call    czy_przest      ; ECX = 0, gdy przestępny

        mov     eax, [l_dni]
        mov     [dzienr], ax    ; zapisz numer dnia w roku

        ; sprawdzimy, do którego miesiąca należy wyliczony numer dnia
        xor     esi, esi        ; zakładamy rok nieprzestępny
        mov     ebx, 2          ; zaczynamy od pierwszego miesiąca
        test    ecx, ecx
        jnz     .nie_przest
        add     esi, 13*2       ;jeśli przestępny,bierzemy drugą grupę liczb
```

```
.nie_przest:
        ; szukamy miesiąca. EAX = numer dnia w roku
        cmp     ax, [dni1+esi+ebx]  ; porównujemy numer dnia z sumą dni aż
                                    ; do NASTĘPNEGO miesiąca
        jbe     mies_juz            ; jeśli już mniejszy, przerywamy
        add     ebx, 2              ; sprawdzamy kolejny miesiąc
        jmp     .nie_przest

mies_juz:
                ; aby dostać numer dnia w miesiącu, odejmujemy od numeru dnia
                ;sumę liczb dni we wszystkich POPRZEDNICH miesiącach, stąd -2
        sub     ax, [dni1+esi+ebx-2]
        inc     al      ; i dodajemy jeden, żeby nie liczyć od zera
        mov     [dzien], al     ; zapisujemy dzień miesiąca

        shr     ebx, 1  ; numer znalezionego miesiąca dzielimy przez 2, bo
                        ; są 2 bajty na miesiąc
        mov     [mies], bl      ; i zachowujemy

        mov     eax, 1
        xor     ebx, ebx
        int     80h     ; koniec programu


dodatek:
; oblicza DODATEK dla roku podanego w EAX
        push    eax
        push    ebx
        push    ecx
        push    edx
        push    esi
        push    edi

        mov     esi, 4
        mov     edi, 100
        mov     ebx, 400
        and     eax, 0ffffh

        push    eax
        xor     edx, edx
        idiv    esi             ; dziel EAX przez 4
        mov     ecx, eax        ; zachowaj wynik
        cmp     edx, 0          ; sprawdź resztę
        jge     .edx_ok1
        sub     ecx, 1          ; jeśli reszta < 0, od wyniku odejmij 1
.edx_ok1:

        pop     eax
        push    eax
        xor     edx, edx
        idiv    edi             ; dziel EAX przez 100
        sub     ecx, eax        ; odejmij od bieżącego wyniku
        cmp     edx, 0          ; sprawdź resztę
        jge     .edx_ok2
        add     ecx, 1          ; jeśli reszta < 0, od wyniku odejmij 1
.edx_ok2:

        pop     eax
        xor     edx, edx
        idiv    ebx             ; dziel EAX przez 400
        add     ecx, eax        ; dodaj do bieżącego wyniku
        cmp     edx, 0          ; sprawdź resztę
```

```
        jge     .edx_ok3
        sub     ecx, 1          ; jeśli reszta < 0, od wyniku odejmij 1
.edx_ok3:

        mov     [przest], ecx   ; zachowaj wynik

        pop     edi
        pop     esi
        pop     edx
        pop     ecx
        pop     ebx
        pop     eax
        ret

; zwraca 0 w ECX, gdy rok podany w EAX jest przestępny, 1 - gdy nie jest
czy_przest:
        push    eax
        push    ebx
        push    edx

        xor     ecx, ecx

        push    eax
        xor     edx, edx
        mov     ebx, 4
        idiv    ebx             ; dziel EAX przez 4
        pop     eax
        test    edx, edx
        jnz     .nie_jest       ; reszta różna od zera oznacza, że się nie
                                ; dzieli, czyli nie może być przestępny

        ; będąc tu wiemy, że rok dzieli się przez 4
        push    eax
        xor     edx, edx
        mov     ebx, 100
        idiv    ebx             ; dziel EAX przez 100
        pop     eax
        test    edx, edx
        jnz     .jest           ; reszta różna od zera oznacza, że się nie
                                ; dzieli przez 100, a dzielił się przez 4,
                                ; czyli jest przestępny


        ; będąc tu wiemy, że rok dzieli się przez 4 i przez 100
        push    eax
        xor     edx, edx
        mov     ebx, 400
        idiv    ebx             ; dziel EAX przez 400
        pop     eax
        test    edx, edx
        jz      .jest           ; reszta równa zero oznacza, że się dzieli
                                ; przez 400, czyli jest przestępny

.nie_jest:
        mov     ecx, 1

.jest:
        pop     edx
        pop     ebx
        pop     eax
        ret
```

```
segment readable writeable

l_dni   dd      0       ; wyliczona liczba dni
reszta  dd      0       ; reszta z dzieleń
y       dd      1970    ; początkowa wartość Y
yg      dd      0       ; zmienna YG
przest  dd      0       ; dodatek
czas    dd      0       ; znacznik czasu

rok     dw      0       ; bieżący rok
mies    db      0       ; bieżący miesiąc
dzien   db      0       ; bieżący dzień miesiąca
dzient  db      0       ; bieżący dzień tygodnia
dzienr  dw      0       ; bieżący dzień roku

godz    db      0       ; bieżąca godzina
min     db      0       ; bieżąca minuta
sek     db      0       ; bieżąca sekunda

; liczby dni poprzedzających każdy miesiąc w roku zwykłym i przestępnym
dni1    dw      0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365
        dw      0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366
```

Zmiana formy tradycyjnej na znacznik czasu

Ten algorytm jest o wiele prostszy. Mianowicie:
Znacznik czasu = SEKUNDY + MINUTY*60 + GODZINY*60*60 + DZIEŃ_ROKU*60*60*24 + LATA_OD_1970*60*60*24*365 + LATA_PRZESTĘPNE_OD_1970*60*60*24
Wystarczy jedynie obliczyć, którym dniem w roku jest bieżący dzień (znając dzień miesiąca, korzystamy z tablicy w powyższym programie i do określonej liczby dodajemy bieżący numer dnia) oraz ile było lat przestępnych od roku 1970 do bieżącego (według znanych reguł, wystarczy w pętli dla każdego roku uruchomić procedurę czy_przest z poprzedniego programu).
Zauważcie, że tyle, ile było lat przestępnych, tyle dodajemy dni, nie całych lat.

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Zabawa diodami na klawiaturze pod Linuksem

Aby uczynić swój program bardziej atrakcyjnym wzrokowo i pochwalić się swoimi umiejętnościami, można sprawić, aby diody na klawiaturze wskazujące stan Num Lock, Caps Lock, Scroll Lock zaczęły migotać w jakimś rytmie.
Teraz pokażę, jak to zrobić.

Do manipulowania urządzeniem znakowym (klawiaturą) posłużymy się funkcją systemową sys_ioctl (numer 54). Najpierw jednak trzeba się dowiedzieć, jakich komend możemy używać. W tym celu udajemy się do pliku /usr/include/linux/kd.h lub do manuala, jeśli mamy w nim stronę ioctl_list, a w nim mamy (po angielsku):
(przeskocz komendy)

```
#define KDGETLED 0x4B31  /*zwróć bieżący stan diód */
#define KDSETLED 0x4B32  /*ustaw stan diód (światełek, nie flag)*/
#define LED_SCR  0x01    /*dioda scroll lock */
#define LED_NUM  0x02    /*dioda num lock */
#define LED_CAP  0x04    /*dioda caps lock */
```

Jedną z tych dwóch pierwszych wartości wpiszemy do ECX.
Tymczasem w EBX musimy mieć deskryptor otwartego pliku /dev/console (do zapisu) lub wartość 1, która oznacza standardowe urządzenie wyjścia - STDOUT.
W EDX podajemy ostatni parametr: wartość 0-7 jeśli ustawiamy stan diód lub wskaźnik (adres) na zmienną typu DWORD, która otrzyma bieżąca wartość stanu diód (także 0-7).

Widać więc, co musi zrobić nasz program: zachować bieżący stan diód, dowolnie je pozmieniać (i zadbać by efekt był widoczny - robić pauzy), po czym przywrócić poprzedni stan.
Oto, jak taki program mógłby wyglądać (używanie załączników z mojej biblioteki nie jest konieczne - w kodzie mówię, jak i co zamienić).
(przeskocz przykładowy program)

```
; Program manipuluje diodami klawiatury
;
; Autor: Bogdan D.
; Kontakt: bogdandr (at) op (dot) pl
;
; nasm -f elf klaw.asm
; ld -s -o klaw klaw.o


section .text

                ; nie musicie z tego korzystać:
%include "bibl/incl/linuxbsd/nasm/n_system.inc"
%include "bibl/incl/linuxbsd/nasm/n_const.inc"

%define KDGETLED        0x4b31
%define KDSETLED        0x4b32

global _start

_start:
```

```
; 1. otwieramy /dev/console, w trybie tylko do zapisu lub
; korzystamy ze STDOUT

        mov     eax, sys_open   ; sys_open = 5 (otwieramy plik)
        mov     ebx, konsola    ; adres nazwy pliku
        mov     ecx, O_WRONLY   ; O_WRONLY = 01
        mov     edx, 777q       ; RWX dla wszystkich
        int     80h

        cmp     eax, 0
        jge     .ok             ; jak nie ma błędu, to jedziemy dalej

                                ; w przypadku błędu korzystamy ze STDOUT
        mov     eax, stdout     ; stdout = 1

; 2. pobieramy aktualny stan diód

.ok:
        mov     ebx, eax        ; EBX = deskryptor pliku

        mov     eax, sys_ioctl  ; sys_ioctl = 54 - manipulacja urządzeniem
        mov     ecx, KDGETLED   ; pobierz stan diód
        mov     edx, stare_diody        ; adres DWORDa, który otrzyma
                                        ; aktualny stan diód
        int     80h

        mov     eax, sys_ioctl  ; sys_ioctl = 54
        mov     ecx, KDSETLED   ; ustawiamy stan diód
        mov     edx, 7          ; wszystkie włączone
        int     80h

        mov     cx, 7
        mov     dx, 0a120h      ; opóźnienie pół sekundy
        call    pauza

; przywracamy poprzedni stan diód

        mov     eax, sys_ioctl
        mov     ecx, KDSETLED           ; ustawiamy stan diód
        mov     edx, [stare_diody]      ; EDX = poprzedni stan diód
        int     80h

        cmp     ebx, stdout     ; czy otworzyliśmy konsolę, czy STDOUT?
        jle     .koniec         ; nie zamykamy STDOUT

        mov     eax, sys_close  ; zamykamy otwarty plik konsoli
        int     80h

.koniec:
        wyjscie                 ; czyli
                                ;       mov     eax, 1
                                ;       xor     ebx, ebx
                                ;       int     80h


pauza:                  ; procedura pauzująca przez CX:DX milisekund
        push    ebx
        push    ecx
        push    edx

        mov     ax, cx
        shl     eax, 16
```

```
        mov     ebx, 1000000
        mov     ax, dx          ; EAX = CX:DX
        xor     edx, edx
        div     ebx             ; CX:DX dzielimy przez milion
        mov     [t1 + timespec.tv_sec], eax     ; EAX = ilość sekund

        mov     ebx, 1000
        mov     eax, edx        ; EAX = pozostała ilość mikrosekund
        mul     ebx
        mov     [t1 + timespec.tv_nsec], eax    ; EAX = ilość nanosekund

        mov     eax, sys_nanosleep      ; funkcja numer 162
        mov     ebx, t1
        mov     ecx, t2
        int     80h

        pop     edx
        pop     ecx
        pop     ebx

        ret

section .data

stare_diody     dd      0
konsola         db      "/dev/console",0

; Struktura timespec jest zdefiniowana w pliku n_system.inc
;struc timespec
;               .tv_sec:                resd 1
;               .tv_nsec:               resd 1
;endstruc

t1 istruc timespec
t2 istruc timespec
```

Dalsze eksperymenty pozostawiam czytelnikom. Pamiętajcie, że istnieje aż 8 różnych kombinacji stanów diód i można przecież robić różne odstępy czasowe między zmianą stanu.

Miłej zabawy.

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Rysowanie w trybie graficznym pod Linuksem

W Linuksie oczywiście nie można rysować bezpośrednio, czyli programować kartę graficzną lub zapisywać do pamięci ekranu (dostępnej np. w DOSie pod segmentem A000h). Zamiast tego, większość roboty wykonają za nas biblioteki oraz moduły jądra, odpowiedzialne za urządzenia (jeśli programujemy np. framebuffer). W tym artykule wykorzystam możliwości biblioteki SVGAlib, ze względu na prostotę jej opanowania, oraz z biblioteki Allegro ze względu na jej wieloplatformowość, łatwość używania i możliwość zapisania obrazów do pliku.

SVGAlib

(przeskocz do SVGAlib z mapowaniem pamięci)

Aby móc korzystać z SVGAlib, musicie zainstalować pakiety svgalib oraz svgalib-devel lub po prostu samemu skompilować bibliotekę, jeśli pakiety nie są dostępne.

Będziemy się zajmować dwoma trybami (ale nic nie stoi na przeszkodzie, aby skorzystać z dowolnego innego). Będą to: tryb 320x200 w 256 kolorach i oczywiście tryb tekstowy (ten, do którego wrócimy po zakończeniu programu). Do ustawienia bieżącego trybu służy funkcja vga_setmode. Przyjmuje ona jeden argument - numer trybu (G320x200x256 = 5, TEXT = 0).

Do zmiany bieżącego koloru służy funkcja vga_setcolor. Jedynym jej argumentem jest numer koloru (np. 1-niebieski, 2-zielony, 3-jasnoniebieski, 4-czerwony, 5-fioletowy, 6-brązowy, 7-biały).

Do narysowania pojedynczego piksela służy funkcja vga_drawpixel. Przyjmuje ona dwa argumenty. Od lewej (ostatni wkładany na stos) są to: współrzędna X oraz współrzędna Y punktu do zapalenia. Punkt o współrzędnych (0,0) to lewy górny róg ekranu.

Współrzędna X rośnie w prawo, a Y - w dół ekranu.

Do narysowania linii służy funkcja vga_drawline. Przyjmuje ona 4 argumenty. Od lewej (ostatni wkładany na stos) są to: współrzędna X początku linii, współrzędna Y początku linii, współrzędna X końca linii, współrzędna Y końca linii.

Aby nasz rysunek był widoczny choć przez chwilę, skorzystamy z funkcji systemowej sys_nanosleep, podając jej adres struktury timespec mówiącej, jak długą przerwę chcemy. Więcej szczegółów w innych artykułach oraz w opisie przerwania int 80h.

Do działania programów pod X-ami potrzebne mogą być uprawnienia do pliku /dev/console a pod konsolą tekstową - do pliku /dev/mem.

Jak widać, teoria nie jest trudna, więc przejdźmy od razu do przykładowych programów.

Pierwszy z nich ma zaprezentować rysowanie pojedynczego piksela oraz dowolnych linii. Zwróćcie uwagę na *sposób kompilacji*. Korzystamy z bibliotek dostępnych dla programistów języka C, więc do łączenia programu w całość najlepiej użyć GCC - zajmie się on dołączeniem wszystkich niezbędnych bibliotek. A

skoro używamy gcc, to funkcja główna zamiast _start, *musi się nazywać main* - tak samo jak funkcja główna w programach napisanych w C. I tak samo, zamiast funkcji wychodzenia z programu, możemy użyć komendy RET, aby zamknąć program.

(przeskocz program rysujący linie)

```
; Program do rysowania linii dowolnej z wykorzystaniem SVGAlib
;
; Autor: Bogdan D., bogdandr (at) op.pl
;
; kompilacja:
; nasm -O999 -f elf -o graf1.o graf1.asm
; gcc -o graf1 graf1.o -lvga


section .text
global  main

; reszta trybów dostępna w /usr/include/vga.h (wymagany svgalib-devel)
%define TEXT         0
%define G320x200x256 5

; deklaracje funkcji zewnętrznych:

extern  vga_setmode
extern  vga_setcolor
extern  vga_drawline
extern  vga_drawpixel

main:
        push    dword G320x200x256
        call    vga_setmode             ; ustawiamy tryb graficzny:
                                        ; 320x200 w 256 kolorach
        add     esp, 4                  ; zdejmujemy argument ze stosu

        push    dword 5                 ; ustawiamy kolor (5=fioletowy)
        call    vga_setcolor
        add     esp, 4

        push    dword 100               ; współrzędna y punktu
        push    dword 160               ; współrzędna x punktu
        call    vga_drawpixel           ; rysujemy piksel
        add     esp, 8

        push    dword 6                 ; ustawiamy kolor (6=brązowy)
        call    vga_setcolor
        add     esp, 4

        push    dword 160
        push    dword 320
        push    dword 0
        push    dword 0
        call    vga_drawline            ; linia od lewego górnego
                                        ; narożnika do środka prawego boku
        add     esp, 16

        push    dword 7                 ; ustawiamy kolor (7=biały)
        call    vga_setcolor
        add     esp, 4

        push    dword 10
```

```
        push    dword 20
        push    dword 110
        push    dword 50
        call    vga_drawline
        add     esp, 16

        mov     dword [t1+timespec.tv_nsec], 0
        mov     dword [t1+timespec.tv_sec], 5           ; czekaj 5 sekund

        mov     eax, 162                ; sys_nanosleep
        mov     ebx, t1                 ; adres struktury mówiącej,
                                        ; ile chcemy czekać
        mov     ecx, 0
        int     80h                     ; robimy przerwę...

        push    dword TEXT
        call    vga_setmode             ; ustawiamy tryb tekstowy 80x25
        add     esp, 4

        xor     eax, eax                ; zerowy kod zakończenia (bez błędu)
        ret                             ; powrót z funkcji main i
                                        ; zakończenie programu

section .data


struc timespec
        .tv_sec:        resd 1
        .tv_nsec:       resd 1
endstruc

t1              istruc timespec
```

Drugi program rysuje okrąg. Środek tego okręgu jest w środku ekranu, kolejne punkty (łącznie będzie ich 360) obliczam następująco: współrzędna x = współrzędna x środka + r*cos(t), y = y_środka + r*sin(t), po przerobieniu kąta t na radiany. Do liczenia tych sinusów i kosinusów wykorzystuję FPU.
(przeskocz program rysujący okrąg)

```
; Program do rysowania okręgu z wykorzystaniem SVGAlib
;
; Autor: Bogdan D., bogdandr (at) op.pl
;
; kompilacja:
; nasm -O999 -f elf -o kolo_linux.o kolo_linux.asm
; gcc -o kolo_linux kolo_linux.o -lvga


section .text
global  main

; reszta trybów dostępna w /usr/include/vga.h (wymagany svgalib-devel)
%define TEXT         0
%define G320x200x256 5

extern  vga_setmode
extern  vga_setcolor
extern  vga_drawpixel

main:
        push    dword G320x200x256
        call    vga_setmode             ; ustawiamy tryb graficzny:
```

```
                                ; 320x200 w 256 kolorach
        add     esp, 4          ; zdejmujemy argument ze stosu

        push    dword 2         ; ustawiamy kolor
        call    vga_setcolor
        add     esp, 4

        mov     ebx, 360

        finit                   ; poniżej będę zapisywał stan
                                ; rejestrów FPU, od st0 do st7
        fldpi                   ; pi
        fild    word [sto80]    ; 180, pi

        fdivp   st1, st0        ; pi/180

        fld1                    ; 1, pi/180
        fild    word [r]        ; r, 1, pi/180
        fldz                    ; kąt=0, r, 1, pi/180

rysuj:
        fld     st0             ; kąt, kąt, r, 1, pi/180

        fmul    st4             ; kąt w radianach

        fsin                    ; sin(kąt), kąt, r, 1, pi/180
        fmul    st2             ; sin(kąt)*r, kąt, r, 1, pi/180

        fistp   dword [wys]     ; kąt, r, 1, pi/180

        fld     st0             ; kąt, kąt, r, 1, pi/180
        fmul    st4             ; kąt w radianach
        fcos                    ; cos(kąt), kąt, r, 1, pi/180
        fmul    st2             ; r*cos(kąt), kąt, r, 1, pi/180

        fistp   dword [szer]    ; kąt, r, 1, pi/180

        mov     eax, [wys]
        mov     edx, [szer]
        add     eax, 100        ; dodajemy współrzędną y środka
        add     edx, 160        ; dodajemy współrzędną x środka
        push    eax             ; umieszczamy współrzędne na stosie
        push    edx
        call    vga_drawpixel   ; rysujemy piksel
        add     esp, 8

        fadd    st0, st2        ; kąt = kąt + 1

        dec     ebx
        jnz     rysuj

        mov     dword [t1+timespec.tv_nsec], 0
        mov     dword [t1+timespec.tv_sec], 5          ; 5 sekund

        mov     eax, 162        ; sys_nanosleep
        mov     ebx, t1         ; adres struktury mówiącej,
                                ; ile chcemy czekać
        mov     ecx, 0
        int     80h             ; robimy przerwę...

        push    dword TEXT
        call    vga_setmode     ; ustawiamy tryb tekstowy 80x25
```

```
        add     esp, 4

        xor     eax, eax                ; zerowy kod zakończenia (bez błędu)
        ret                             ; powrót z funkcji main

section .data


struc timespec
        .tv_sec:        resd 1
        .tv_nsec:       resd 1
endstruc

t1              istruc timespec

r               dw      50              ; promień okręgu
szer            dd      0
wys             dd      0
sto80           dw      180
```

SVGAlib z mapowaniem pamięci

(przeskocz do Allegro)

Aby móc korzystać z SVGAlib, musicie zainstalować pakiety svgalib oraz svgalib-devel lub po prostu samemu skompilować bibliotekę, jeśli pakiety nie są dostępne.

UWAGA: zmieni się sposób kompilacji programu w stosunku do tradycyjnych, asemblerowych programów. Korzystamy z bibliotek dostępnych dla programistów języka C, więc do łączenia programu w całość najlepiej użyć GCC - zajmie się on dołączeniem wszystkich niezbędnych bibliotek. A skoro używamy gcc, to funkcja główna zamiast _start, *musi się nazywać main* - tak samo jak funkcja główna w programach napisanych w C. I tak samo, zamiast funkcji wychodzenia z programu, możemy użyć komendy RET, aby zamknąć program. Sama kompilacja przebiega następująco:

```
nasm -O999 -f elf -o graf2.o graf2.asm
gcc -o graf2 graf2.o -lvga
```

Będziemy się zajmować dwoma trybami (ale nic nie stoi na przeszkodzie, aby skorzystać z dowolnego innego). Będą to: tryb 320x200 w 256 kolorach i oczywiście tryb tekstowy (ten, do którego wrócimy po zakończeniu programu). Do ustawienia bieżącego trybu służy funkcja vga_setmode. Przyjmuje ona jeden argument - numer trybu (G320x200x256 = 5, TEXT = 0).

Przed rozpoczęciem pracy ustawiamy tryb graficzny 320x200, wykonując

```
        extern  vga_setmode
                ...
                push    dword 5                 ; G320x200x256
                call    vga_setmode             ; ustawiamy tryb graficzny:
                                                ; 320x200 w 256 kolorach
                add     esp, 4                  ; zdejmujemy argument ze stosu
```

Ale teraz zajmiemy się rysowaniem bez funkcji SVGAlib, poprzez zapis do odpowiednich komórek pamięci. Pamięć trybów graficznych znajduje się w segmencie A000, co odpowiada liniowemu adresowi A0000, licząc od adresu 0. Oczywiście system, ze względów bezpieczeństwa, nie pozwoli nam bezpośrednio pisać pod ten

adres, więc musimy sobie poradzić w inny sposób. Sposób ten polega na otwarciu specjalnego pliku urządzenia, który symbolizuje całą pamięć w komputerze - /dev/mem. Na większości systemów otwarcie tego pliku wymaga uprawnień administratora.
Po otwarciu pliku mamy dwie możliwości. Pierwsza to poruszać się po nim funkcjami do zmiany pozycji w pliku, oraz odczytywać i zapisywać funkcjami odczytu i zapisu danych z i do pliku. Może to być powolne, ale sposób jest. Druga możliwość to zmapować plik do pamięci, po czym korzystać z niego jak ze zwykłej tablicy. Tę możliwość opiszę teraz szczegółowo.
Otwieranie pliku odbywa się za pomocą tradycyjnego wywołania:

```
mov     eax, 5          ; sys_open
mov     ebx, pamiec     ; adres nazwy pliku "/dev/mem", 0
mov     ecx, 2          ; O_RDWR, zapis i odczyt
mov     edx, 666o       ; pełne prawa
int     80h
...
pamiec          db      "/dev/mem", 0
```

Drugim krokiem jest zmapowanie naszego otwartego pliku do pamięci. Odbywa się to za pomocą funkcji systemowej sys_mmap2. Przyjmuje ona 6 argumentów:

1. EBX = adres, pod jaki chcielibyśmy zmapować plik. Najlepiej podać zero, wtedy system sam wybierze dogodny adres
2. ECX = długość mapowanego obszaru pliku, w bajtach. Podamy to 100000h, by na pewno objąć obszar zaczynający się A0000 i długości 64000 bajtów (tyle, ile trzeba na jeden ekran w trybie 320x200)
3. EDX = tryb dostępu do zmapowanej pamięci. Jeśli chcemy odczyt i zapis, podamy tutaj PROT_READ=1 + PROT_WRITE=2
4. ESI = tryb współdzielenia zmapowanej pamięci. Podamy tu MAP_SHARED=1 (współdzielona, nie prywatna)
5. EDI = deskryptor otwartego pliku, który chcemy zmapować
6. EBP = adres początkowy w pliku, od którego mapować. Adres ten jest podawany w jednostkach strony systemowej, której wielkość może być różna na różnych systemach. Najłatwiej podać tu zero, a do adresów dodawać potem A0000

Po pomyślnym wykonaniu, system zwróci nam w EAX adres zmapowanego obszaru pamięci, którego możemy używać (w przypadku błędu otrzymujemy wartość od -4096 do -1 włącznie). Przykładowe wywołanie wygląda więc tak:

```
mov     eax, 192                ; sys_mmap2
xor     ebx, ebx                ; jądro wybierze adres
mov     ecx, 100000h            ; długość mapowanego obszaru
mov     edx, 3                  ; PROT_READ | PROT_WRITE, możliwość
                                ; zapisu i odczytu
mov     esi, 1                  ; MAP_SHARED - tryb współdzielenia
mov     edi, [deskryptor]       ; deskryptor pliku pamięci, otrzymany
                                ; z sys_open w poprzednim kroku
mov     ebp, 0                  ; adres początkowy w pliku
int     80h
```

Teraz wystarczy już korzystać z otrzymanego wskaźnika, na przykład:

```
mov     byte [eax+0a0000h], 7
```

Kolejne adresy w pamięci oznaczają kolejne piksele określonego wiersza. Po przekroczeniu 320 bajtów, kolejny bajt oznacza pierwszy piksel kolejnego wiersza i tak dalej.
Bajty zapisywane w pamięci (czyli kolory pikseli) mają takie same wartości, jak w tradycyjnym podejściu: 1-niebieski, 2-zielony, 3-jasnoniebieski, 4-czerwony, 5-fioletowy, 6-brązowy, 7-biały.

Zmiany, które zapiszemy w pamięci, mogą jednak nie od razu pojawić się w pliku (czyli na ekranie w tym przypadku). Aby wymusić fizyczny zapis danych, korzysta się z funkcji sys_msync. Przyjmuje ona 3 argumenty:

1. EBX = adres początku danych do synchronizacji
2. ECX = ilość bajtów do zsynchronizowania
3. EDX = 0 lub zORowane flagi: MS_ASYNC=1 (wykonaj asynchronicznie), MS_INVALIDATE=2 (unieważnij obszar po zapisaniu), MS_SYNC (wykonaj synchronicznie)

Przykładowe wywołanie wygląda więc tak:

```
mov     eax, 144                ; sys_msync
mov     ebx, 0a0000h            ; adres startowy
mov     ecx, 4000               ; ile zsynchronizować
mov     edx, 0                  ; flagi
int     80h
```

Po zakończeniu pracy należy przywrócić tryb tekstowy:

```
push    dword 0                 ; TEXT
call    vga_setmode             ; ustawiamy tryb tekstowy 80x25
add     esp, 4
```

oraz odmapować plik:

```
mov     eax, 91                 ; sys_munmap
mov     ebx, [wskaznik]         ; wskaźnik otrzymany z sys_mmap2
mov     ecx, 100000h            ; ilość bajtów
int     80h
```

i zamknąć:

```
mov     eax, 6                  ; sys_close
mov     ebx, [deskryptor]       ; deskryptor pliku "/dev/mem"
int     80h
```

Jeśli Wasza grafika ma czesto się zmieniać (na przykład jest to animacja), to pisanie bezpośrednio do zmapowanej pamięci (lub pamięci wideo, jeśli macie dostęp) może być zbyt powolne, by efekty graficzne były zadowalające. Ale można to obejść na dwa sposoby: uruchamiać sys_msync dopiero po zapełnieniu całego ekranu lub cały ekran najpierw zbudować sobie w osobnym buforze, po czym jednym ruchem wrzucić cały ten bufor do zmapowanej pamięci czy pamięci wideo.

Jak widać, mapowanie plików do pamięci jest wygodne, gdyż nie trzeba ciągle skakać po pliku funkcją sys_lseek i wykonywać kosztownych czasowo wywołań innych funkcji systemowych. Warto więc się z tym zaznajomić. Należy jednak pamiętać, że nie wszystkie pliki czy urządzenia dają się zmapować do pamięci - nie należy wtedy zamykać swojego programu z błędem, lecz korzystać z tradycyjnego interfejsu funkcji plikowych.

Allegro

Biblioteka Allegro powinna na większości systemów być dostępna jako gotowy pakiet, ale w razie czego można ją pobrać na przykład ze strony alleg.sf.net.

Pierwszymi funkcjami, jakie w ogóle należy uruchomić przez rozpoczęciem czegokolwiek są install_allegro (w języku C - allegro_init) i install_keyboard. Pierwsza służy do inicjalizacji biblioteki. Jako parametry oczekuje: liczbę SYSTEM_AUTODETECT=0, adres zmiennej do przechowywania błędów (ale uwaga, próba deklaracji i użycia errno w naszym programie może skończyć się błędem linkera, więc lepiej podać adres jakiegoś naszego własnego DWORDa) oraz wskaźnika na elementy uruchamiane przy wyjściu (u nas wpiszemy NULL). Druga funkcja nie przyjmuje żadnych argumentów, a służy do instalacji funkcji odpowiedzialnych za działanie klawiatury. Allegro samo zajmuje się klawiaturą, więc standardowe funkcje czytania z klawiatury mogą nie działać. Do czytania klawiszy służy funkcja readkey. Nie przyjmuje ona żadnych argumentów, a zwraca wartość przeczytanego klawisza.

Biblioteka pozwala na ustawienie wielu rozdzielczości, my zajmiemy się rozdzielczością 640x480 w 8-bitowej głębi kolorów. Do ustawienia głębi kolorów służy funkcja set_color_depth przyjmująca jeden argument - wartość owej głębi, czyli w naszym przypadku 8.

Po inicjalizacji biblioteki, instalacji klawiatury i ustawieniu głębi kolorów można przystąpić do ustawienia trybu graficznego. Robi się to za pomocą funkcji set_gfx_mode. Przyjmuje ona 5 argumentów: sterownik (u nas będziemy korzystać z autowykrywania, wpisując tu liczbę GFX_AUTODETECT=0), szerokość żądanego trybu w pikselach, wysokość trybu, szerokość okna widoku i wysokość okna widoku. U nas oknem widoku będzie cały ekran, więc ostatnie dwa parametry przyjmą wartość zero, a całe wywołanie (w języku C) będzie miało postać:

```
set_gfx_mode ( GFX_AUTODETECT, 640, 480, 0, 0 );
```

Jeśli ustawienie rozdzielczości się nie powiedzie, wywołanie funkcji zwróci wartość niezerową.

Po skończeniu pracy z Allegro należy wywołać funkcję allegro_exit w celu zamknięcia i odinstalowania biblioteki z programu. Funkcja ta nie przyjmuje żadnych argumentów.

Aby ustawić domyślną paletę kolorów, wywołujemy funkcję set_palette. Jako jej jedyny parametr podajemy zewnętrzną (pochodzącą z Allegro) zmienną default_palette.

Do czyszczenia ekranu (a właściwie wypełnienia go określonym kolorem) służy funkcja clear_to_color. Jej pierwszy parametr mówi, co ma zostać wyczyszczone - u nas chcemy wyczyścić cały ekran, więc będzie to zmienna z Allegro o nazwie screen. Drugi parametr tej funkcji to kolor, jakim chcemy wypełnić ekran. Zero oznacza czarny.

Do wyświetlania tekstu na ekranie w trybie tekstowym służy funkcja allegro_message. Jej jedyny argument to tekst do wyświetlenia. Aby wyświetlić tekst w trybie graficznym, najpierw należy podjąć decyzję, czy tekst ma być na tle, czy tło ma go przykryć. Jeśli tło ma być pod tekstem, należy jednorazowo wywołać text_mode, jako parametr podając liczbę -1 (minus jeden). Potem można już wyświetlać tekst funkcją textout. Przyjmuje ona 5 argumentów: gdzie wyświetlić (u nas znów screen), jaką czcionką (skorzystamy z domyślnej czcionki w zmiennej Allegro o nazwie font), co wyświetlić (adres naszego napisu), współrzędna X, współrzędna Y oraz żądany kolor.

Współrzędna X rośnie w prawo, a Y - w dół ekranu.

Ale przejdźmy wreszcie do wyświetlania podstawowych elementów.
Linię wyświetla się funkcją line, przyjmującą 6 argumentów: gdzie wyświetlić (tak, znowu screen), współrzędna X początku, współrzędna Y początku, współrzędna X końca, współrzędna Y końca, kolor.
Kolor, jak w każdej innej funkcji, możemy podawać ręcznie jako liczbę, ale możemy też uruchomić funkcję makecol, podając jej wartości od 0 do 255 kolejno dla kolorów: czerwonego, zielonego, niebieskiego, a wynik tej funkcji podajemy tam, gdzie podalibyśmy kolor.
Okręgi wyświetla się funkcją circle, przyjmującą 5 argumentów: gdzie wyświetlić (i znowu screen), współrzędna X środka, współrzędna Y środka, promień i kolor.

Po omówieniu tego, co ma być w programie jeszcze dwa słowa o tworzeniu programu. O ile kompilacja pliku w asemblerze jest taka jak zawsze, to linkowanie najlepiej przeprowadzić za pomocą GCC. Normalnie naszą funkcję główna nazwalibyśmy main, ale Allegro posiada własną funkcję main, a oczekuje, że nasza funkcja główna będzie sie nazywać _mangled_main (z podkreśleniem z przodu). Ponadto, Allegro oczekuje, że zadeklarujemy zmienną globalną _mangled_main_address i wpiszemy do niej adres _mangled_main. W języku C robi to za nas makro END_OF_MAIN. Program linkuje się następującą komendą:
`gcc -o program program.o `allegro-config --libs``
Zwróćcie uwagę na odwrotne apostrofy. Sprawią one, że wynik zawartej w nich komendy (a więc niezbędne biblioteki) zostanie przekazany do GCC, dzięki czemu znajdzie on wszystko, co potrzeba.

A oto przykładowy program. Wyświetla on tekst, linię i okrąg, po czym czeka na naciśnięcie jakiegokolwiek klawisza. Po naciśnięciu klawisza biblioteka Allegro jest zamykana i program się kończy.
(przeskocz przykładowy program Allegro)

```
; Program demonstracyjny biblioteki Allegro
;
; Autor: Bogdan D., bogdandr (at) op.pl
;
; kompilacja:
; nasm -O999 -f elf -o graf2.o graf2.asm
; gcc -o graf2 graf2.o `allegro-config --libs`

section .text
; wymagane przez Allegro:
global _mangled_main
global _mangled_main_address

; deklaracje elementów zewnętrznych:
extern  install_allegro
extern  install_keyboard
extern  set_color_depth
extern  set_gfx_mode
extern  allegro_exit
extern  text_mode
extern  set_palette
extern  default_palette
extern  clear_to_color
extern  screen
extern  textout
extern  font
extern  line
extern  makecol
extern  circle
extern  readkey
```

```
%define GFX_AUTODETECT 0                ; autowykrywanie sterownika

_mangled_main:
        ; inicjalizacja biblioteki:
        push    dword 0
        push    err                     ; nasza zmienna do błędów
        push    dword 0
        call    install_allegro
        add     esp, 3*4                ; zdjęcie parametrów ze stosu

        ; instalacja klawiatury
        call    install_keyboard

        ; ustawienie głębi kolorów:
        push    dword 8
        call    set_color_depth
        add     esp, 1*4                ; zdjęcie parametrów ze stosu

        ; ustawienie rozdzielczości:
        push    dword 0                 ; wysokość okna
        push    dword 0                 ; szerokość okna
        push    dword 480               ; wysokość całego trybu
        push    dword 640               ; szerokość całego trybu
        push    dword GFX_AUTODETECT
        call    set_gfx_mode
        add     esp, 5*4

        ; sprawdź, czy się udało
        cmp     eax, 0
        jne     koniec

        ; ustaw tło pod tekstem
        push    dword -1
        call    text_mode
        add     esp, 1*4

        ; ustaw domyślną paletę
        push    dword default_palette
        call    set_palette
        add     esp, 1*4

        ; wyczyść ekran
        push    dword 0                 ; czyść na czarno
        push    dword [screen]          ; co czyścić
        call    clear_to_color
        add     esp, 2*4

        ; wyświetl napis
        push    dword 15                ; kolor
        push    dword 10                ; współrzędna Y
        push    dword 10                ; współrzędna X
        push    dword napis             ; napis do wyświetlenia
        push    dword [font]            ; czcionka
        push    dword [screen]          ; gdzie wyświetlić
        call    textout
        add     esp, 6*4

        ; stwórz kolor biały do narysowania linii
        push    dword 255               ; składowa niebieska
        push    dword 255               ; składowa zielona
        push    dword 255               ; składowa czerwona
```

```
        call    makecol
        add     esp, 3*4

        ; narysuj linię
        push    eax                     ; kolor
        push    dword 240               ; współrzędna Y końca
        push    dword 320               ; współrzędna X końca
        push    dword 400               ; współrzędna Y początku
        push    dword 540               ; współrzędna X początku
        push    dword [screen]
        call    line
        add     esp, 6*4

        ; stwórz kolor zielony do narysowania koła
        push    dword 0
        push    dword 255
        push    dword 0
        call    makecol
        add     esp, 3*4

        ; narysuj koło
        push    eax                     ; kolor
        push    dword 20                ; promień
        push    dword 240               ; współrzędna Y środka
        push    dword 320               ; współrzędna X środka
        push    dword [screen]
        call    circle
        add     esp, 5*4

        ; czekaj na klawisz
        call    readkey

koniec:
        ; zamknij Allegro
        call    allegro_exit
        ; powróć z naszej funkcji głównej
        ret

section .data
napis                   db      "Allegro", 0    ; napis do wyświetlenia
_mangled_main_address   dd      _mangled_main   ; wymagane
err                     dd      0               ; nasza zmienna błędów
```

Jak widać, biblioteka Allegro jest tylko trochę trudniejsza od SVGAlib, ale jej możliwości są znacznie większe. Tutaj pokazałem tylko ułamek grafiki dwuwymiarowej. Allegro potrafi też wyświetlać grafikę trójwymiarową, wyliczać transformacje, zapisywać wyświetlane obrazy do pliku oraz odtwarzać muzykę (w końcu to jest biblioteka do gier, nie tylko graficzna). Jak widzicie, jest jeszcze wiele możliwości przed Wami do odkrycia. Miłej zabawy!

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Programowanie myszy pod Linuksem

Wbudowanie obsługi myszy do swoich programów wcale nie musi być trudne. Nie musimy bawić się w pisanie sterowników do każdej możliwej myszki ani bezpośrednio rozmawiać ze sprzętem. Cała ta robota została już za nas zrobiona. Komunikacją z myszą zajmie się serwer myszy GPM, a my zajmiemy się tylko uruchamianiem odpowiednich funkcji, jakie ten serwer oferuje.
Podane tu przeze mnie informacje pochodzą z moich własnych interpretacji pliku /usr/include/gpm.h oraz ze świetnego artykułu linuxjournal.com/article/4600 i wymagają zainstalowanego pakietu gpm-devel.
Teraz po kolei przedstawię i omówię czynności, jakie należy wykonać. Rzecz jasna, serwer myszy musi być uruchomiony.

1. Otwarcie połączenia.
   Aby otworzyć połączenie z serwerem, należy najpierw odpowiednio wypełnić strukturę Gpm_Connect postaci:
   (przeskocz strukturę Gpm_Connect)

```
struc Gpm_Connect
        .eventMask      resw    1
        .defaultMask    resw    1
        .minMod         resw    1
        .maxMod         resw    1
        .pid            resd    1
        .vc             resd    1
endstruc
```

   Nas interesują tylko 4 pierwsze pola.
   Do eventMask wstawimy NOT 0 (czyli -1), a do defaultMask zero, co oznacza, że interesują nas wszystkie typy zdarzeń.
   Do minMod wstawimy 0, a do maxMod - NOT 0, co oznacza, że interesują nas wszystkie klawisze modyfikujące (Ctrl, Alt, ...).

   Po wypełnieniu struktury uruchamiamy funkcję Gpm_Open z dwoma parametrami: adres naszej struktury oraz wartość 0. Jeśli wystąpi błąd, Gpm_Open zwróci wartość -1. Typową przyczyną jest brak uprawnień do pliku gniazda serwera. Jako root należy wtedy wpisać chmod o+rwx /dev/gpmctl. Jeśli chcemy, aby kursor myszy był widoczny, należy do zmiennej globalnej serwera gpm_visiblepointer wstawić wartość 1.

2. Ustalenie własnej funkcji obsługi zdarzeń (komunikatów odbieranych przez serwer od myszy).
   Nic prostszego. Do zmiennej globalnej gpm_handler wpisujemy adres naszej procedury. Dzięki temu serwer będzie wiedział, gdzie jest funkcja, którą należy uruchomić, gdy nastąpi zdarzenie.

3. Oczekiwanie na zdarzenia.
   Aby odbierać zdarzenia, należy skorzystać z funkcji Gpm_Getc. Przyjmuje ona jeden argument: adres struktury FILE opisującej plik, z którego odbierane mają być zdarzenia. Wpisujemy tam standardowe wejście (jako zmienną z biblioteki języka C). Jeśli nastąpi jakieś zdarzenie, nasza procedura obsługi zdarzeń zostanie automatycznie uruchomiona z właściwymi parametrami.

4. Zamykanie połączenia.
   Robimy to zwykle wtedy, gdy Gpm_Getc otrzymała znak końca pliku (DWORD -1). Samo

zamknięcie polega na uruchomieniu funkcji Gpm_Close (bez argumentów).

Procedura obsługi zdarzeń

Jest to kluczowa funkcja obsługi myszy. Dostaje ona na stosie dwa argumenty:

1. pierwszy od lewej (ostatni wkładany na stos) - wskaźnik do struktury Gpm_Event, opisującej dane zdarzenie.
2. drugi od lewej - wskaźnik do dodatkowych danych

Jeśli w swojej procedurze chcecie tylko coś wyświetlić, to te argumenty nie są wam potrzebne. Ale struktura Gpm_Event niesie ze sobą wiele przydatnych informacji, które teraz objaśnię.
Sama struktura wygląda tak:
(przeskocz strukturę Gpm_Event)

```
struc Gpm_Event
        .buttons        resb    1
        .modifiers      resb    1
        .vc             resw    1
        .dx             resw    1
        .dy             resw    1
        .x              resw    1
        .y              resw    1
        .type           resd    1
        .clicks         resd    1
        .margin         resd    1
        .wdx            resw    1
        .wdy            resw    1
endstruc
```

Pole buttons mówi o tym, które przyciski zostały naciśnięte. Wystarczy użyć instrukcji TEST z jedną z wartości GPM_B_* (podanych w programie).
Pole modifiers mówi o tym, które z klawiszy modyfikujących były aktywne w chwili zdarzenia. Można użyć instrukcji TEST z jedną z wartości (1 << KG_*).
Pola X oraz Y to oczywiście współrzędne zdarzenia (w czasie ruchu zmieniają się).
I najważniejsze chyba pole: type, opisujące rodzaj zdarzenia. Użyjcie instrukcji TEST z jedną z wartości GPM_*, aby określić typ zdarzenia. Dla kliknięcia najczęściej będzie to wartość 20 (czyli 16+4) - kliknięcie jednym przyciskiem.

Przykładowy program

Aby poskładać całą tą wiedzę i pozwolić wam uchronić się od części błędów, przedstawiam poniżej przykładowy program. Jego zadaniem jest wypisanie stosownego napisu w punkcie kliknięcia (najlepiej testować na konsoli tekstowej). Wykorzystane w programie sekwencje kontrolne terminala są omówione w

artykule Bezpośredni dostęp do ekranu.
Program zakończy działanie po odebraniu znaku końca pliku (na konsoli należy nacisnąć Ctrl+D).
(przeskocz program reagujący na mysz)

```
; Programowanie myszy w asemblerze z wykorzystaniem GPM
;
; autor: Bogdan D., bogdandr (at) op.pl
;
; kompilacja:
; nasm -O999 -f elf -o mysz.o mysz.asm
; gcc -o mysz mysz.o -lgpm

section .text
global  main

; struktura służąca połączeniu się z serwerem (/usr/linux/gpm.h)
struc Gpm_Connect
        .eventMask      resw    1
        .defaultMask    resw    1
        .minMod         resw    1
        .maxMod         resw    1
        .pid            resd    1
        .vc             resd    1
endstruc

; struktura opisująca zdarzenie myszy: ruch, kliknięcie itp.
struc Gpm_Event
        .buttons        resb    1
        .modifiers      resb    1
        .vc             resw    1
        .dx             resw    1
        .dy             resw    1
        .x              resw    1
        .y              resw    1
        .type           resd    1
        .clicks         resd    1
        .margin         resd    1
        .wdx            resw    1
        .wdy            resw    1
endstruc

; przyciski (pole "buttons" w Gpm_Event)

; FASM: GPM_B_* = x
%define GPM_B_DOWN      32              ; naciśnięcie przycisku
%define GPM_B_UP        16              ; zwolnienie przycisku
%define GPM_B_FOURTH    8
%define GPM_B_LEFT      4
%define GPM_B_MIDDLE    2
%define GPM_B_RIGHT     1
%define GPM_B_NONE      0

; typy zdarzeń (pole "type" w Gpm_Event)

; FASM: GPM_* = x
%define   GPM_MOVE      1
%define   GPM_DRAG      2
%define   GPM_DOWN      4
%define   GPM_UP        8
%define   GPM_SINGLE    16
%define   GPM_DOUBLE    32
```

```
%define   GPM_TRIPLE    64
%define   GPM_MFLAG     128
%define   GPM_HARD      256
%define   GPM_ENTER     512
%define   GPM_LEAVE     1024

; numery bitów klawiszy (pole "modifiers" w Gpm_Event)
; z /usr/include/linux/keyboard.h

; FASM: KG_* = x
%define KG_SHIFT        0
%define KG_CTRL         2
%define KG_ALT          3
%define KG_ALTGR        1
%define KG_SHIFTL       4
%define KG_SHIFTR       5
%define KG_CTRLL        6
%define KG_CTRLR        7
%define KG_CAPSSHIFT    8


; FASM: extrn zamiast extern
extern  Gpm_Open        ; funkcja otwierająca połączenie z serwerem
extern  Gpm_Close       ; funkcja zamykająca połączenie z serwerem
extern  Gpm_Getc        ; funkcja pobierająca znak i uruchamiająca
                        ; obsługę zdarzeń
extern  gpm_handler     ; tu wpiszemy adres naszej funkcji
                        ; obsługi zdarzeń myszy
extern  stdin           ; standardowe wejście, skąd będziemy czytać znaki
extern  gpm_visiblepointer      ; zmienna mówiąca o tym,
                                ; czy kursor jest widoczny

main:
        mov     eax, 4
        mov     ebx, 1
        mov     ecx, czysc
        mov     edx, czysc_dl
        int     80h                     ; czyścimy ekran

        mov     dword [gpm_visiblepointer], 1   ; kursor ma być widoczny

                        ; akceptujemy wszystkie zdarzenia myszy
        mov     word [conn + Gpm_Connect.eventMask], ~ 0
                        ; FASM: not 0
        mov     word [conn + Gpm_Connect.defaultMask], 0

                        ; akceptujemy wszystkie klawisze modyfikujące
        mov     word [conn + Gpm_Connect.minMod], 0
        mov     word [conn + Gpm_Connect.maxMod], ~ 0
                        ; FASM: not 0

        push    dword 0
        push    dword conn
        call    Gpm_Open                ; otwieramy połączenie z serwerem
        add     esp, 8                  ; usuwamy argumenty ze stosu

        cmp     eax, -1                 ; EAX = -1 oznacza błąd
        jne     .jest_ok

        mov     ebx, eax
        mov     eax, 1
        int     80h                     ; zwracamy kod -1
```

```
.jest_ok:
                        ; wpisujemy adres naszej funkcji:
        mov     dword [gpm_handler], obsluga

.czytaj:
                        ; wczytujemy znak z klawiatury lub zdarzenie myszy
        push    dword [stdin]
        call    Gpm_Getc                ; pobieramy znak/zdarzenie.
                ; Funkcja obsługi zostanie uruchomiona automatycznie.
        add     esp, 4

        cmp     eax, -1                 ; -1 oznacza EOF, koniec pliku
        jne     .czytaj

        call    Gpm_Close               ; zamykamy połączenie z serwerem

        mov     eax, 4
        mov     ebx, 1
        mov     ecx, czysc
        mov     edx, czysc_dl
        int     80h                     ; czyścimy ekran

        mov     bx, 1
        mov     cx, 1
        call    zmiana_poz              ; zmiana pozycji kursora tekstowego

        mov     eax, 1
        xor     ebx, ebx
        int     80h                     ; wyjście z programu



; procedura obsługi zdarzenia myszy. otrzymuje 2 argumenty:
; wskaźnik na zdarzenie i wskaźnik na dane
; prototyp w C wygląda tak:
; int obsluga( Gpm_Event *ev, void *dane );

obsluga:
        push    ebp
        mov     ebp, esp
                        ; [ebp] = stary EBP
                        ; [ebp+4] = adres powrotny
                        ; [ebp+8] = pierwszy parametr (wsk. na zdarzenie)
                        ; [ebp+12] = drugi parametr (wsk. na dane)

%define         ev      ebp+8
%define         dane    ebp+12

                                ; interesują nas tylko kliknięcia:
        mov     eax, [ev]       ; [ev], a nie ev, jest wskaźnikiem
                                ; na strukturę
        test    dword [eax + Gpm_Event.type], GPM_DOWN
        jz      .koniec

        push    ebx
        push    ecx
        push    edx

                                        ; wyświetlimy typ zdarzenia
        mov     ecx, [ev]
```

```
        mov     eax, [ecx + Gpm_Event.type]
                                ; cała zawartość i tak mieści się w AX
        xor     edx, edx
        mov     bx, 1000
        div     bx              ; maksymalny typ to 1024, więc zaczynamy
                                ;  dzielenie od 1000
        add     "0"             ; zmiana wyniku na ASCII
        mov     [numer], al

        mov     eax, edx        ; EAX = reszta z poprzedniego dzielenia
        mov     bl, 100
        div     bl
        add     al, "0"
        mov     [numer+1], al

        shr     ax, 8           ; AX = AH = reszta z poprzedniego dzielenia
        mov     bl, 10
        div     bl
        add     ax, "00"
        mov     [numer+2], ax   ; na miejsce trzecie wstawiamy iloraz,
                                ; a na czwarte - od razu resztę


        mov     dh, 10
        mov     ax, [ecx + Gpm_Event.x] ; x = kolumna
        div     dh
        add     ax, "00"                ; zamiana ilorazu i reszty na ASCII
        mov     [miejsce+4], ax         ; od razu wstawiamy iloraz i resztę

        mov     ax, [ecx + Gpm_Event.y] ; y = wiersz
        div     dh
        add     ax, "00"
        mov     [miejsce+1], ax

        mov     bx, [ecx + Gpm_Event.x]
        mov     cx, [ecx + Gpm_Event.y]
        call    zmiana_poz      ; zmiana pozycji kursora tekstowego

        mov     eax, 4
        mov     ebx, 1
        mov     ecx, zdarz
        mov     edx, zdarz_dl
        int     80h             ; na ustalonej przed chwila pozycji
                                ; piszemy info o zdarzeniu

        pop     edx
        pop     ecx
        pop     ebx

.koniec:
        xor     eax, eax        ;zerowa wartość zwracana oznacza brak błędu
        leave                   ; przywracamy ESP i EBP
        ret



; procedura zmiany pozycji kursora tekstowego (nie myszy)
; wykorzystuje sekwencje kontrolne terminala
zmiana_poz:

;  BX = nowa kolumna
;  CX = nowy wiersz
```

```
        push    eax
        push    ebx
        push    ecx
        push    edx

        mov     dh, 10
        mov     ax, bx                  ; AX = kolumna znaku
        and     ax, 0FFh
        div     dh                      ; dzielimy przez 10
        add     ax, "00"                ; zmiana ilorazu i reszty na ASCII
        mov     [kolumna], ax           ; zapisanie ilorazu i reszty

        mov     ax, cx                  ; AX = wiersz znaku
        and     ax, 0FFh
        div     dh
        add     ax, "00"
        mov     [wiersz], ax

        mov     eax, 4
        mov     ebx, 1
        mov     ecx, pozycja
        mov     edx, pozycja_dl
        int     80h             ; wyświetlenie sekwencji kontrolnej

        pop     edx
        pop     ecx
        pop     ebx
        pop     eax

        ret


section .data

zdarz           db      "Zdarzenie: "
numer           db      "xxxx w "       ; typ zdarzenia
miejsce         db      "(ww,kk)"       ; pozycja na ekranie
zdarz_dl        equ     $ - zdarz


ESC             equ     1Bh             ; kod ASCII klawisza Escape

                                        ; sekwencja zmiany pozycji kursora
pozycja         db      ESC, "["
wiersz          db      "00;"
kolumna         db      "00H"
pozycja_dl      equ     $ - pozycja

czysc           db      ESC, "[2J"      ; sekwencja czyszczenia ekranu
czysc_dl        equ     $ - czysc

section .bss

        ; conn jest strukturą typu Gpm_Connect:
conn            istruc  Gpm_Connect
```

Porty szeregowe i równoległe

Niektórym programom nie wystarcza działanie na samym procesorze czy sprzęcie znajdującym się w komputerze. Czasem trzeba połączyć się z jakimś urządzeniem zewnętrznym, takim jak modem zewnętrzny czy drukarka. Celem tego artykułu jest właśnie pokazanie, jak to zrobić.

Porty równoległe

(przeskocz porty równoległe)

Tutaj sprawa jest dość prosta. Porty równoległe nie wymagają żadnych ustawień, ewentualnie tylko tryb pracy, ustawiany w BIOSie. Praca z portem równoległym pod Linuksem sprowadza się do czytania i zapisu do specjalnych plików - `/dev/parportN` (N - liczba), które reprezentują porty równoległe. O tym, jak obsługiwać pliki, napisałem w kursie.

Porty szeregowe

Porty szeregowe są trudniejsze w obsłudze. Czasem wystarczy, podobnie jak dla portów równoległych, po prostu czytać i zapisywać do specjalnych plików, ale to nie zawsze może wystarczyć. Jest tak, gdyż porty szeregowe mają swoje ustawienia:

- szybkość transmisji (w bodach = bitach na sekundę): od 75 do nawet 4 milionów
- ilość bitów danych - od 5 do 8
- kontrola parzystości - brak, parzysta, nieparzysta, mark (znacznik) i space
- kontrola przepływu - brak, programowa (XON/XOFF) i sprzętowa (RTS/CTS)
- bity stopu - 1, półtora lub 2

Pierwszym krokiem jest otwarcie specjalnego pliku urządzenia, zazwyczaj `/dev/ttySx` (x - liczba). O tym, jak otwierać pliki, zapisywać i odczytywać z nich dane, napisałem w kursie.
Jeśli trzeba ustawić parametry portu, wykonuje się to funkcją systemową sys_ioctl (numer 54). Przyjmuje ona w tym przypadku 3 argumenty:

- EBX = deskryptor portu, otrzymany z otwarcia pliku urządzenia
- ECX = komenda (TCGETS=0x00005401 dla pobrania parametrów portu, TCSETS=0x00005402 dla ustawienia)
- EDX = adres struktury termios, która otrzyma dane lub zawiera parametry do ustawienia. Struktura ta wygląda tak:

```
struc __kernel_termios
    .c_iflag:   resd 1;         flagi trybu wejścia
    .c_oflag:   resd 1;         flagi trybu wyjścia
    .c_cflag:   resd 1;         flagi trybu kontroli
    .c_lflag:   resd 1;         flagi trybu lokalnego
    .c_line:    resb 1;         obsługa linii
    .c_cc:      resb 32;        znaki kontrolne
```

```
        endstruc
```

Najpierw należy pobrać bieżące parametry portu, potem zmienić te, które chcemy i wysłać je do portu. Poniżej przedstawiam przykładowy kod w składni NASMa. Najpierw otwiera on plik portu, potem odczytuje bieżące argumenty, i ustawia nowe:

- we flagach kontrolnych - szybkość 115200 bps (B115200), 8 bitów danych (CS8), połączenie lokalne (CLOCAL), możliwość odczytywania (CREAD)
- we flagach wejściowych - sprawdzanie parzystości (INPCK)
- we flagach wyjściowych nic nie jest potrzebne
- we flagach obsługi linii - tryb kanoniczny (ICANON): dane są przesyłane linijkami (dopiero znak Entera powoduje wysłanie danych)
- w znakach kontrolnych - znak VKILL ma być ignorowany, a minimalna liczba znaków (VMIN) wynosi 1

Wszystkie wykorzystane stałe można znaleźć w pliku `/usr/include/bits/termios.h`. Polecam zapoznanie się, gdyż można tam znaleźć wiele ciekawych opcji, na przykład automatyczne tłumaczenie znaków CR na LF i na odwrót.

```
        mov     eax, 5          ; otwieranie pliku
        mov     ebx, port       ; nazwa
        mov     ecx, 402o       ; odczyt i zapis, nie terminal kontrolujący
        mov     edx, 777o       ; rwx dla wszystkich
        int     80h

        cmp     eax, 0
        jl      koniec
        mov     ebx, eax        ; EBX = deskryptor

        ; pobieranie i ustawianie parametrów portu
%define TCGETS 0x00005401
%define TCSETS 0x00005402

        mov     eax, 54         ; sys_ioctl
        ;ebx = deskryptor
        mov     ecx, TCGETS     ; pobierz parametry
        mov     edx, termios
        int     80h

        cmp     eax, 0
        jl      koniec

%define B115200  0010002o
%define CS8   0000060o
%define CLOCAL  0004000o
%define CREAD   0000200o
        mov dword [termios+__kernel_termios.c_cflag],B115200|CS8|CLOCAL|CREAD
%define INPCK   0000020o
        mov     dword [termios+__kernel_termios.c_iflag], INPCK
        mov     dword [termios+__kernel_termios.c_oflag], 0
%define ICANON  0000002o
        mov     dword [termios+__kernel_termios.c_lflag], ICANON
%define VKILL 3
%define VMIN 6
        mov     byte [termios+__kernel_termios.c_cc+VKILL], 0
        mov     byte [termios+__kernel_termios.c_cc+VMIN], 1

        mov     eax, 54         ; sys_ioctl
```

```
        ;ebx = deskryptor
        mov     ecx, TCSETS     ; ustaw parametry
        mov     edx, termios
        int     80h

        cmp     eax, 0
        jnl     ioctl_set_ok
...
section .data
port            db      "/dev/ttyS0", 0
termios         istruc  __kernel_termios
```

Po wykonaniu tego kodu można normalnie czytać z urządzenia i zapisywać do niego jak do zwykłego pliku. Na uwagę zasługuje wartość 402o wpisana do ECX przed otwarciem pliku. Mówi ona, że chcemy dostęp do odczytu i zapisu, ale dodatkowo włączona jest opcja O_NOCTTY (400o). Sprawi ona, że otwarte przez nas urządzenie znakowe *NIE* stanie się terminalem kontrolującym programu. W innym przypadku czytanie ze standardowego wejścia kończyłoby się czytaniem z wybranego portu szeregowego, a próba wyświetlenia napisu wysyłałaby bajty na ten port.

Polecam lekturę Serial Programming HOWTO.

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Zarządzanie zasilaniem komputera.

Jeśli zastanawialiście się kiedyś, jak wyłączać dyski twarde lub resetować komputer używając tylko oprogramowania (nie naciskając żadnych przycisków), to w tym artykule powinniście znaleźć odpowiedź na wszystkie wasze pytania.

Dyski twarde.

(przeskocz dyski twarde)

Oczywiście, jak na porządny system przystało, nie możemy operować bezpośrednio na dysku twardym. Ale umożliwi na to sam system, poprzez funkcję systemową sys_ioctl. Poniżej przedstawiam działający (choć, aby otworzyć plik urządzenia dysku twardego, musiałem mieć uprawnienia root'a) kod zatrzymujący dysk twardy. Kod ten napisałem dzięki analizie kodu źródłowego programu hdparm (i tam, oraz do stron podręcznika man: ioctl, ioctl_list, odsyłam po szczegóły). Dla tych, którzy nie korzystają z mojej biblioteki, podaję liczbowe odpowiedniki stałych.
Oto program (składnia NASM):
(przeskocz program zatrzymujący dysk twardy)

```
%include "bibl/incl/linuxbsd/nasm/n_system.inc"         ; stałe systemowe

section .text

global _start

_start:
        mov     eax, sys_open                   ; =5. otwieramy plik....
        mov     ebx, dysk                       ; ....twardego dysku
        mov     ecx, O_RDONLY|O_NONBLOCK        ; 04000q ósemkowo
        int     80h

        cmp     eax, 0
        jle     koniec                          ; jeśli wystąpił błąd,
                                                ; wychodzimy od razu

        mov     ebx, eax                ; zachowujemy deskryptor pliku

        mov     eax, sys_ioctl                  ; =54
        ; EBX = deskryptor pliku
        mov     ecx, 0x031f                     ; komenda specjalna dysku
        mov     edx, args1                      ; pierwsze argumenty
        int     80h

        mov     eax, sys_ioctl
        ; EBX = deskryptor pliku
        mov     ecx, 0x031f                     ; komenda specjalna dysku
        mov     edx, args2                      ; drugie argumenty
        int     80h
```

```
        mov     eax, sys_close                  ; =6
        ; EBX = deskryptor pliku
        int     80h                             ; zamykamy otwarty plik

koniec:
        mov     eax, sys_exit                   ; =1
        xor     ebx, ebx
        int     80h                             ; wychodzimy z programu

section .data
args1   db      0e6h, 0, 0, 0                   ; skopiowane z hdparm.c
args2   db      99h, 0, 0, 0                    ; też skopiowane
dysk    db      "/dev/hda",0                    ; pierwszy dysk,hdb = drugi
```

UWAGA: należy odczekać chwilę, aż program się zakończy. Przez czas działania programu komputer może przestać reagować.
Po zatrzymaniu twardego dysku można go uruchomić wykonując dowolną operację na systemie plików (na przykład wyświetlić zawartość bieżącego katalogu).

Resetowanie komputera lub wyłączanie go.

(przeskocz resetowanie komputera)

Do zresetowania komputera posłuży nam funkcja systemowa sys_reboot, której podamy odpowiednie parametry. Oto program natychmiast resetujący komputer:
(przeskocz program resetujacy komputer)

```
%include "bibl/incl/linuxbsd/nasm/n_system.inc"

section .text

global _start

_start:
        mov     eax, sys_reboot                 ; =88
        mov     ebx, 0fee1deadh                 ; wymagana stała
        mov     ecx, 672274793                  ; wymagana stała
                                ; EDX = tryb restartu. U nas: zwykły reset
        mov     edx, LINUX_REBOOT_CMD_RESTART   ; =0x01234567
        int     80h

        ; to, co jest poniżej nigdy nie zostanie wykonane.
        mov     eax, sys_exit                   ; =1
        xor     ebx, ebx
        int     80h
```

A oto program natychmiast wyłączający komputer:
(przeskocz program wyłączający komputer)

```
%include "bibl/incl/linuxbsd/nasm/n_system.inc"
```

```
section .text

global _start

_start:
        mov     eax, sys_reboot                 ; =88
        mov     ebx, 0fee1deadh                 ; wymagana stała
        mov     ecx, 672274793                  ; wymagana stała
                        ; EDX = tryb restartu. U nas: wyłączenie prądu
        mov     edx, LINUX_REBOOT_CMD_POWER_OFF ; =0x4321FEDC
        int     80h

        ; to, co jest poniżej nigdy nie zostanie wykonane.
        mov     eax, sys_exit                   ; =1
        xor     ebx, ebx
        int     80h
```

UWAGA: ze względu na to, że podane programy wyłączają/resetują komputer natychmiast po uruchomieniu, *NIE* zalecam ich stosowania, gdyż może to być *niezdrowe dla systemu plików*.

Wyłączanie monitora.

No może nie zupełnie będzie to wyłączenie monitora, ale istotnie, obrazu nie będzie. Do wyłączenia obrazu wykorzystamy bibliotekę svgalib (do kompilacji potrzebny będzie też pakiet svgalib-devel).
Całość jest bardzo prosta: inicjujemy bibliotekę funkcją vga_init, wyłączamy obraz funkcją vga_screenoff, czekamy na klawisz (funkcja systemowa sys_read czytająca ze standardowego wejścia) i ponownie włączamy obraz funkcją vga_screenon. Funkcje vga nie przyjmują żadnych argumentów.
Cały program wygląda tak:
(przeskocz program wyłączajacy monitor)

```
; Program wyłączający monitor z wykorzystaniem SVGAlib
;
; Autor: Bogdan D., bogdandr (at) op.pl
; kompilacja:
;   nasm -O999 -f elf -o mon_off.o mon_off.asm
;   gcc -o mon_off mon_off.o -lvga

section .text
global  main

extern  vga_screenoff
extern  vga_screenon
extern  vga_init

%idefine        stdin           0
%idefine        sys_read        3

main:
        call    vga_init
        call    vga_screenoff

        mov     eax, sys_read   ; funkcja czytania z pliku
        mov     ebx, stdin      ; standardowe wejście
        mov     ecx, znak       ; adres, dokąd czytać
        mov     edx, 1          ; wczytaj 1 bajt
```

```
        int     80h

        call    vga_screenon

        ret

section .data

znak            db      0
```

Zwróćcie uwagę na sposób kompilacji. Skoro korzystamy z bibliotek języka C, to do kompilacji użyjemy GCC. Wtedy zaś funkcja główna musi się nazywać main - tak samo, jak w programach w języku C (i tak samo, jak w C, można ją zakończyć komendą RET). Potrzebne funkcje po prostu deklarujemy jako zewnętrzne (extern).

Program ten uruchomiłem też w środowisku graficznym. Nic złego się nie stało, ale po zakończeniu programu należy przejść na konsolę tekstową i wrócić na graficzną, aby odzyskać obraz.

Do działania programu pod X-ami potrzebne mogą być uprawnienia do pliku /dev/console a pod konsolą tekstową - do pliku /dev/mem.

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Pisanie programów sieciowych pod Linuksem

Linuks jest systemem typowo sieciowym. Nawet niektóre usługi systemowe działają jako serwery sieciowe, umożliwiając dostęp maszynom z zewnątrz. A ja bez niepotrzebnego zagłębiania się w porty, protokoły i inne szczegóły dotyczące sieci, pokażę teraz, jak napisać prosty, własny serwerek i klienta do niego.

Komunikacja w sieci odbywa się z wykorzystaniem wielu różnych elementów. Podstawowym pojęciem jest gniazdo (ang. socket). Jest to logiczne (czyli nie istniejące fizyczne) urządzenie będące podstawową bramką, przez którą przepływają informacje. Gniazdko tworzy się funkcją socket (z biblioteki języka C, tak jak wszystkie późniejsze). Przyjmuje ona 3 argumenty (patrz: man 2 socket):

1. domena - określa typ połączenia. My wykorzystamy wartość PF_INET=2, oznaczającą protokoły internetowe (IPv4)
2. typ gniazda - określa, czy gniazdo jest datagramowe, strumieniowe, surowe (raw) itp. My wykorzystamy gniazdo strumieniowe SOCK_STREAM=1 i protokół TCP (Transmission Control Protocol), który gwarantuje wiarygodne, dwustronne połączenie.
3. protokół, jeśli nie jest on jednoznacznie wyznaczony. U nas TCP jest jednoznacznie wyznaczony przez typ gniazda (strumieniowe), więc ten argument przyjmuje wartość 0.

Jeśli utworzenie gniazda nie udało się, funkcja socket zwróci wartość -1. Jeśli się udało, zwróci liczbę całkowitą - deskryptor otwartego gniazda (podobnie, jak w plikach). Po zakończeniu pracy gniazdo można zamknąć funkcją close.

Od chwili utworzenia gniazda dalszy kod w serwera i klienta różnią się, więc omówię je po kolei.

Serwer

(przeskocz opis serwera)

Jak wiemy, zadaniem serwera jest nasłuchiwanie połączeń od klientów. Aby to osiągnąć, należy wykonać następujące kroki.

1. Przypisanie gniazda do adresu.
   W chwili utworzenia, gniazdo nie jest jeszcze przypisane do adresu, a przecież trzeba jakoś określić, na jakim adresie i porcie nasłuchuje nasz serwer. Służy do tego funkcja bind. Przyjmuje ona następujące argumenty (patrz: man 2 bind):
   1. gniazdo, które utworzyliśmy funkcją socket
   2. adres struktury sockaddr, którą zaraz się zajmiemy
   3. długość tejże struktury

   Choć definicja funkcji bind mówi o strukturze sockaddr, to funkcji tej podaje się odpowiednio rzutowany wskaźnik do struktury sockaddr_in. Ta struktura wygląda tak:
   (przeskocz strukturę sockaddr_in)

```
struc sockaddr_in
      .sin_family resw 1      ; rodzina adresów
```

```
        .sin_port   resw 1      ; numer portu
        .sin_addr   resd 1      ; adres
                    resb 8      ; dopełnienie do 16 bajtów
endstruc
```

Do pola sin_family wpisujemy AF_INET=2, oznaczające rodzinę adresów internetowych.
Do pola sin_port wpisujemy numer portu, na którym będzie nasłuchiwał nasz serwer. Ale uwaga - nie bezpośrednio! Najpierw numer portu musi zostać przetłumaczony na sieciowy porządek bajtów funkcją htons (patrz: man htons). Dopiero wynik funkcji, której podajemy numer portu, wpisujemy w to pole. Programy bez uprawnień administratora mogą korzystać tylko z portów o numerach powyżej 1023.
Do pola sin_addr wpisujemy wartość INADDR_ANY=0, co oznacza, że chcemy nasłuchiwać na dowolnym adresie.

W przypadku błędu, bind zwraca -1.

2. Włączenie nasłuchiwania na danym gnieździe.
   Aby włączyć nasłuchiwanie na danym gnieździe, należy użyć funkcji listen. Przyjmuje ona dwa argumenty (patrz: man 2 listen):
   1. gniazdo, utworzone funkcją socket z adresem przypisanym funkcją bind
   2. maksymalną liczbę klientów oczekujących w kolejce na obsługę

   W przypadku błędu, listen zwraca -1.
   Jeśli funkcja listen się powiedzie, to można z serwerem przejść w tryb demona (o tym w innym kursie).

Po włączeniu nasłuchiwania na gnieździe możemy zacząć przyjmować połączenia od klientów. Przyjęcie połączenia odbywa się funkcją accept. Przyjmuje ona trzy argumenty (patrz: man 2 accept):

1. nasłuchujące gniazdo
2. zero lub adres struktury sockaddr (lub tej samej sockaddr_in, którą podaliśmy dla bind). Struktura ta otrzyma dane o kliencie (np. jego adres)
3. adres zmiennej zawierającej długość struktury z parametru numer 2

Gdy klient już się połączył, accept zwraca deskryptor nowego gniazda, które będzie służyć do komunikacji z klientem.

Klient

(przeskocz opis klienta)

W porównaniu z serwerem, w kliencie jest mniej pracy. Po utworzeniu gniazda do połączenia się z serwerem wystarczy jedna funkcja - connect. Przyjmuje ona trzy argumenty (patrz: man 2 connect):

1. gniazdo utworzone funkcją socket
2. adres struktury sockaddr
3. długość tejże struktury

Tutaj także zamiast struktury sockaddr przekazujemy adres struktury sockaddr_in. Jednak trzeba ją trochę inaczej wypełnić.

Pola sin_family i sin_port wypełniamy tak samo, jak dla bind. W końcu chcemy się połączyć do tego samego portu, na którym nasłuchuje serwer.
Pole sin_addr wypełniamy adresem IP serwera. Oczywiście nie wprost jako łańcuch znaków, ale odpowiednio przerobionym. Do przerobienia łańcucha znaków 127.0.0.1 (oznaczającego zawsze bieżący komputer dla niego samego) na właściwą postać posłuży nam funkcja inet_aton. Przyjmuje ona 2 argumenty (patrz: man inet_aton):

1. adres łańcucha znaków z adresem w zapisie dziesiętnym kropkowym (ttt.xxx.yyy.zzz)
2. adres struktury in_addr, która otrzyma wynik

Struktura in_addr jest jedyną składową pola sin_addr w naszej strukturze sockaddr_in i to adres tego właśnie pola podajemy funkcji inet_aton.

Po poprawnym wykonaniu połączenia funkcją connect, można przystąpić do wymiany danych.

Wymiana danych

(przeskocz wymianę danych)

Po dokonaniu połączenia obie strony - klient i serwer - mają gotowe gniazda, którymi mogą się komunikować. Do wymiany danych służą dwie podstawowe funkcje: send i recv. Obie przyjmują dokładnie te same cztery parametry (patrz: man 2 send, man 2 recv):

1. gniazdo, które jest połączone z klientem/serwerem
2. adres bufora odbiorczego/nadawczego
3. długość tego bufora
4. specjalne flagi, jeśli jest taka potrzeba. U nas będzie to zero.

Przykład

Po przebrnięciu przez tą trudną teorię możemy wreszcie przystąpić do pisania programów. Wiem, że sucha teoria nie umożliwi natychmiastowego napisania programów serwera i klienta (jest wiele pułapek, na które trzeba zwrócić uwagę), dlatego prezentuję tutaj przykładowe programy serwera i klienta (składnia NASMa).
Serwer:
(przeskocz program serwera)

```
; Program serwera
;
; autor: Bogdan D., bogdandr (at) op.pl
;
; kompilacja:
; nasm -O999 -f elf -o serwer.o serwer.asm
; gcc -o serwer serwer.o

section .text
global  main            ; będziemy korzystać z biblioteki C, więc
```

```
                        ; funkcja główna musi się nazywać "main"

; definicje kilku przydatnych stałych
%define PF_INET         2
%define AF_INET         PF_INET
%define SOCK_STREAM     1
%define INADDR_ANY      0

%define NPORTU          4242
%define MAXKLIENT       5       ; maksymalna liczba klientów

; zewnętrzne funkcje z biblioteki C, z których będziemy korzystać
extern  daemon
extern  socket
extern  listen
extern  accept
extern  bind
extern  htons
extern  recv
extern  send
extern  close

main:
        push    dword 0
        push    dword SOCK_STREAM
        push    dword AF_INET
        call    socket                  ; tworzymy gniazdo:
                                        ;socket(AF_INET,SOCK_STREAM,0);
        add     esp, 12                 ; usuwamy argumenty ze stosu

        cmp     eax, 0                  ; EAX < 0 oznacza błąd
        jl      .sock_blad

        mov     [gniazdo], eax          ; zachowujemy deskryptor gniazda

        push    word NPORTU
        call    htons                   ; przerabiamy numer portu na
                                        ; właściwy format
                                        ; htons(NPORTU);
        add     esp, 2

                        ; wpisujemy przerobiony numer portu:
        mov     [adres+sockaddr_in.sin_port], ax
                        ; rodzina adresów internetowych:
        mov     word [adres+sockaddr_in.sin_family], AF_INET
                        ; akceptujemy każdy adres
        mov     dword [adres+sockaddr_in.sin_addr], INADDR_ANY

        push    dword sockaddr_in_size
        push    dword adres
        push    dword [gniazdo]
        call    bind                    ; przypisujemy gniazdo do adresu:
                                ; bind(gniazdo,&adres,sizeof(adres));
        add     esp, 12

        cmp     eax, 0
        jl      .bind_blad

        push    dword MAXKLIENT
        push    dword [gniazdo]
        call    listen                  ; włączamy nasłuchiwanie:
                                        ; listen(gniazdo,MAXKLIENT);
```

```
        add     esp, 8

        cmp     eax, 0
        jl      .list_blad

        push    dword 1
        push    dword 1
        call    daemon                  ; przechodzimy w tryb demona
        add     esp, 8                  ; usuniecie argumentów ze stosu

        mov     dword [rozmiar], sockaddr_in_size

.czekaj:
        push    dword rozmiar           ; [rozmiar] zawiera rozmiar
                                        ; struktury sockaddr_in
        push    dword adres
        push    dword [gniazdo]
        call    accept                  ; czekamy na połączenie
                                ; accept(gniazdo,&adres,&rozmiar)
        add     esp, 12
        cmp     eax, 0
        jl      .czekaj

        mov     [gniazdo_kli], eax      ; gdy accept się udało,
                                        ; zwraca nowe gniazdo klienta

.rozmowa:
        push    dword 0
        push    dword buf_d
        push    dword bufor
        push    dword [gniazdo_kli]
        call    recv                    ; odbieramy dane;
                        ; recv(gniazdo_kli,&bufor,sizeof(bufor),0);
        add     esp, 16

        cmp     eax, 0                  ; jeśli błąd, to czekamy ponownie
        jl      .rozmowa

        cmp     byte [bufor], "Q"       ; ustalamy, że Q kończy transmisję
        je      .koniec

        mov     ecx, buf_d
        mov     edi, bufor
        xor     eax, eax
        cld
        rep     stosb                   ; czyścimy bufor

        push    dword 0
        push    dword 2
        push    dword ok
        push    dword [gniazdo_kli]
        call    send                    ; wysyłamy dane
                                        ; (na cokolwiek odpowiadamy "OK")
                                        ; send(gniazdo_kli,&ok,2,0);
        add     esp, 16

        jmp     .rozmowa                ; i czekamy od nowa

.koniec:
        push    dword 0
        push    dword buf_d
        push    dword bufor
```

```
        push    dword [gniazdo_kli]
        call    send                    ; wysyłamy Q, które jest w buforze
        add     esp, 16

        push    dword [gniazdo_kli]
        call    close                   ; zamykamy gniazdo klienta
        add     esp, 4

; jeśli chcemy, aby serwer nasłuchiwał kolejnych połączeń, piszemy tu:
;;;     jmp     .czekaj
; serwera nie da się wtedy inaczej zamknąć niż przez zabicie procesu

        push    dword [gniazdo]
        call    close                   ; zamykamy gniazdo główne serwera
        add     esp, 4

        mov     eax, 1
        xor     ebx, ebx
        int     80h                     ; wychodzimy z programu


; obsługa błędów:

.sock_blad:
        mov     eax, 4
        mov     ebx, 1
        mov     ecx, blad_socket
        mov     edx, blad_socket_d
        int     80h                     ; wyświetlenie napisu

        mov     eax, 1
        mov     ebx, 1
        int     80h                     ; wyjście z programu z
                                        ; odpowiednim kodem błędu

.bind_blad:
        mov     eax, 4
        mov     ebx, 1
        mov     ecx, blad_bind
        mov     edx, blad_bind_d
        int     80h

        push    dword [gniazdo]
        call    close                   ; zamykamy gniazdo

        mov     eax, 1
        mov     ebx, 2
        int     80h

.list_blad:
        mov     eax, 4
        mov     ebx, 1
        mov     ecx, blad_listen
        mov     edx, blad_listen_d
        int     80h

        push    dword [gniazdo]
        call    close                   ; zamykamy gniazdo

        mov     eax, 1
        mov     ebx, 3
        int     80h
```

```
section .data

                                        ; deskryptory gniazd:
gniazdo         dd      0
gniazdo_kli     dd      0

bufor           times   20      db      0       ; bufor odbiorczo-nadawczy
buf_d           equ     $ - bufor               ; długość bufora

                                                ; komunikaty błędów:
blad_socket     db      "Problem z socket!", 10
blad_socket_d   equ     $ - blad_socket

blad_bind       db      "Problem z bind!", 10
blad_bind_d     equ     $ - blad_bind

blad_listen     db      "Problem z listen!", 10
blad_listen_d   equ     $ - blad_listen

ok              db      "OK"            ; to, co wysyłamy

struc sockaddr_in

        .sin_family     resw    1               ; rodzina adresów
        .sin_port       resw    1               ; numer portu
        .sin_addr       resd    1               ; adres

                        resb    8               ; dopełnienie do 16 bajtów
endstruc

adres           istruc  sockaddr_in             ; adres jako zmienna, która
                                                ; jest strukturą
rozmiar         dd      sockaddr_in_size        ; rozmiar struktury
```

Klient:
(przeskocz program klienta)

```
; Program klienta
;
; autor: Bogdan D., bogdandr (at) op.pl
;
; kompilacja:
; nasm -O999 -f elf -o klient.o klient.asm
; gcc -o klient klient.o

section .text
global  main            ; będziemy korzystać z biblioteki C, więc
                        ; funkcja główna musi się nazywać "main"

; definicje kilku przydatnych stałych
%define PF_INET         2
%define AF_INET         PF_INET
%define SOCK_STREAM     1
%define INADDR_ANY      0

%define NPORTU          4242

; zewnętrzne funkcje z biblioteki C, z których będziemy korzystać
```

```
extern  socket
extern  connect
extern  htons
extern  recv
extern  send
extern  close
extern  inet_aton

main:
        push    dword 0
        push    dword SOCK_STREAM
        push    dword AF_INET
        call    socket                  ; tworzymy gniazdo:
                                        ; socket(AF_INET,SOCK_STREAM,0);
        add     esp, 12                 ; usuwamy argumenty ze stosu

        cmp     eax, 0                  ; EAX < 0 oznacza błąd
        jle     .sock_blad

        mov     [gniazdo], eax          ; zachowujemy deskryptor gniazda

                                        ; rodzina adresów internetowych:
        mov     word [adres+sockaddr_in.sin_family], AF_INET

        push    dword (adres + sockaddr_in.sin_addr)
        push    dword localhost
        call    inet_aton               ; przerabiamy adres 127.0.0.1 na
                                        ; właściwy format
        add     esp, 8
        test    eax, eax                ; EAX = 0 oznacza, że adres
                                        ; był nieprawidłowy
        jz      .inet_blad

        push    word NPORTU
        call    htons                   ; przerabiamy numer portu
                                        ; na właściwy format
        add     esp, 2
                            ; wpisujemy przerobiony numer portu:
        mov     word [adres+sockaddr_in.sin_port], ax

        push    dword sockaddr_in_size
        push    dword adres
        push    dword [gniazdo]
        call    connect                 ; łączymy się z serwerem:
                            ; connect(gniazdo,&adres,sizeof(adres));
        add     esp, 12

        cmp     eax, 0
        jne     .conn_blad

.rozmowa:
        mov     eax, 3
        mov     ebx, 0
        mov     ecx, bufor
        mov     edx, buf_d
        int     80h                     ; wczytujemy dane ze
                                        ; standardowego wejścia

        push    dword 0
        push    dword buf_d
        push    dword bufor
        push    dword [gniazdo]
```

```
        call    send                    ; wysyłamy to, co wczytaliśmy:
                                ; send(gniazdo,&bufor,sizeof(bufor),0);
        add     esp, 16

        cmp     eax, 0
        jl      .send_blad

        mov     ecx, buf_d
        mov     edi, bufor
        xor     eax, eax
        cld
        rep     stosb                   ; czyścimy bufor

.odbieraj:
        push    dword 0
        push    dword buf_d
        push    dword bufor
        push    dword [gniazdo]
        call    recv                    ; odbieramy dane od serwera:
                                ; recv(gniazdo,&bufor,sizeof(bufor),0);
        add     esp, 16

        cmp     eax, 0
        jl      .odbieraj

        mov     eax, 4
        mov     ebx, 1
        mov     ecx, odebrano
        mov     edx, odebrano_dl
        int     80h                     ; wypisujemy, co odebraliśmy

        cmp     byte [bufor], "Q"               ; "Q" kończy transmisję
        jne     .rozmowa

        push    dword [gniazdo]
        call    close                   ; zamykamy gniazdo
        add     esp, 4

        mov     eax, 1
        xor     ebx, ebx
        int     80h                     ; wychodzimy z programu


; sekcja obsługi błędów


.sock_blad:
        mov     eax, 4
        mov     ebx, 1
        mov     ecx, blad_socket
        mov     edx, blad_socket_d
        int     80h                     ; wyświetlenie napisu

        mov     eax, 1
        mov     ebx, 1
        int     80h                     ; wyjście z programu z
                                        ; odpowiednim kodem błędu

.conn_blad:
        mov     eax, 4
        mov     ebx, 1
        mov     ecx, blad_connect
```

```
        mov     edx, blad_connect_d
        int     80h

        push    dword [gniazdo]
        call    close                   ; zamykamy gniazdo

        mov     eax, 1
        mov     ebx, 2
        int     80h

.inet_blad:
        mov     eax, 4
        mov     ebx, 1
        mov     ecx, blad_inet
        mov     edx, blad_inet_d
        int     80h

        push    dword [gniazdo]
        call    close                   ; zamykamy gniazdo

        mov     eax, 1
        mov     ebx, 3
        int     80h

.send_blad:
        mov     eax, 4
        mov     ebx, 1
        mov     ecx, blad_send
        mov     edx, blad_send_d
        int     80h

        push    dword [gniazdo]
        call    close                   ; zamykamy gniazdo

        mov     eax, 1
        mov     ebx, 4
        int     80h

.recv_blad:
        mov     eax, 4
        mov     ebx, 1
        mov     ecx, blad_recv
        mov     edx, blad_recv_d
        int     80h

        push    dword [gniazdo]
        call    close                   ; zamykamy gniazdo

        mov     eax, 1
        mov     ebx, 5
        int     80h

section .data

gniazdo         dd      0                       ; deskryptor gniazda

odebrano        db      "Serwer: "
bufor           times   20      db      0       ; bufor nadawczo-odbiorczy
buf_d           equ     $ - bufor               ; długość bufora
                db      10                      ; przejście do nowej linii
odebrano_dl     equ     $ - odebrano
```

```
                                                ; komunikaty błędów
blad_socket     db      "Problem z socket!", 10
blad_socket_d   equ     $ - blad_socket

blad_connect    db      "Problem z connect!", 10
blad_connect_d  equ     $ - blad_connect

blad_inet       db      "Problem z inet_aton!", 10
blad_inet_d     equ     $ - blad_inet

blad_send       db      "Problem z send!", 10
blad_send_d     equ     $ - blad_send

blad_recv       db      "Problem z recv!", 10
blad_recv_d     equ     $ - blad_recv


localhost       db      "127.0.0.1", 0          ; adres, z którym
                                                ; będziemy się łączyć

struc sockaddr_in

        .sin_family     resw    1               ; rodzina adresów
        .sin_port       resw    1               ; numer portu
        .sin_addr       resd    1               ; adres

                        resb    8               ; dopełnienie do 16 bajtów
endstruc

adres           istruc  sockaddr_in             ; adres jako zmienna,
                                                ; która jest strukturą
```

Jako że programy te korzystają z biblioteki języka C, ich kompilacja musi wyglądać trochę inaczej niż zwykle:

```
nasm -f elf -o plik.o plik.asm
gcc -o plik plik.o
```

Po kompilacji najpierw oczywiście uruchamiamy serwer poleceniem `./serwer` (program serwera sam przejdzie w tło). Możecie sprawdzić, co się stanie, jeśli dwa razy spróbujecie uruchomić serwer lub uruchomicie klienta bez uruchomionego serwera. Oczywiście, serwer może też być klientem innego serwera (np. po odebraniu danych przerabiać je i przekazywać dalej).

Funkcje sieciowe przerwania int 80h

(przeskocz int 80h)

Korzystanie z sieci jest oczywiście możliwe także bez pośrednictwa biblioteki języka C. W końcu każda tak istotna funkcja przecież musi być zaprogramowana jako część jądra.
Interfejs sieciowy jądra to jedna funkcja - sys_socketcall (numer 102). Przyjmuje ona dwa argumenty. Pierwszy (w EBX) to funkcja, którą chcemy uruchomić. Każda wspomniana wcześniej funkcja z biblioteki C ma swój numer. Są to: dla socket - 1, dla bind - 2, connect - 3, listen - 4, accept - 5, send - 9, recv - 10.

Funkcja close jest tą samą, której używa się do zamykania plików (a więc EBX=[gniazdo], EAX=6, int 80h). Drugim argumentem (w ECX) jest adres reszty argumentów, które podalibyśmy funkcji z biblioteki C. Można je bez przeszkód w tej samej kolejności, co wcześniej, umieścić na stosie, po czym wykonać instrukcję `mov ecx, esp`. Z resztą, tak to właśnie robi biblioteka C (plik sysdeps/unix/sysv/linux/i386/socket.S w źródłach glibc, tam jednak jest "ecx+4", gdyż należy przeskoczyć jeszcze adres powrotny z funkcji). Można te dane umieścić oczywiście w swojej sekcji danych i podać ich adres, ale dane te muszą być jedna po drugiej dokładnie w takiej kolejności, w jakiej znajdowałyby na stosie (czyli *od lewej do prawej* na wzrastających adresach). Po prostu po kolei, według deklaracji C, od lewej do prawej.

Do omówienia zostają jeszcze funkcje pomocnicze - htons i inet_aton.
Funkcja htons jest dość prosta w budowie (plik sysdeps/i386/htons.S w źródłach glibc), jej treść mieści się w takim oto makrze (zakładając, że argument jest w EAX):

```
%macro htons 0
        and     eax, 0FFFFh
        ror     ax, 8
%endm
```

Czyli po prostu zeruje górną połowę EAX i zamienia zawartość rejestrów AH i AL między sobą.

Funkcja inet_aton (plik resolv/inet_addr.c w źródłach glibc) jest trochę trudniejsza. Wolę znacznie wszystko skrócić i powiedzieć, że adres należy załadować do rejestru EAX binarnie, czyli na przykład z 127.0.0.1 dostajemy EAX=7F000001h, a z 192.168.0.2 - EAX=C0A80002h. Potem trzeba odwrócić kolejność bajtów. Najlepiej od początku skorzystać z następującego makra:

```
%macro adr2bin 4

        mov     al, %4
        shl     eax, 8
        mov     al, %3
        shl     eax, 8
        mov     al, %2
        shl     eax, 8
        mov     al, %1
%endm

; użycie:
        adr2bin 127, 0, 0, 1       ; dla adresu 127.0.0.1
        adr2bin 192, 168, 45, 243  ; dla adresu 192.168.45.243
```

którego wynik (EAX) zapisujemy do pierwszych czterech bajtów pola sin_addr struktury sockaddr_in (co normalnie funkcja inet_aton robiła automatycznie).
To całe odwracanie bierze się z tego, że porządek bajtów w protokole TCP jest typu big-endian, a procesory zgodne z Intelem są typu little-endian.
O tym, jak pisać demony korzystając wyłącznie z przerwania int 80h, napisałem w mini-kursie poświęconym temu zagadnieniu.

Funkcje sieciowe w systemie 64-bitowym

(przeskocz system 64-bitowy)

Obsługa sieci różni się nieco na systemach 64-bitowych w porównaniu z systemami 32-bitowymi. Nie tylko zmienia się numer funkcji, ale teraz poszczególne operacje sieciowe mają swoje własne funkcje systemowe. Są to: socket - 41, connect - 42, accept - 43, sendto - 44, recvfrom - 45, bind - 49, listen - 50. Reszta parametrów jest przekazywana nie na stosie, a w kolejnych rejestrach, zgodnie z interfejsem systemu 64-bitowego (kolejno w rejestrach: RDI, RSI, RDX, R10, R8, R9). Samo wywołanie systemu następuje instrukcją syscall, a nie poprzez przerwanie 80h. Przykładowe wywołania funkcji wyglądają więc następująco:

```
        mov     rax, 41                 ; socket
        mov     rdi, AF_INET
        mov     rsi, SOCK_STREAM
        mov     rdx, IPPROTO_TCP
        syscall

        mov     rax, 42                 ; connect
        mov     rdi, [socket]
        mov     rsi, sock_struc
        mov     rdx, sockaddr_in_size
        syscall

        mov     rax, 44                 ; sendto
        mov     rdi, [socket]
        mov     rsi, buf
        mov     rdx, buf_ile
        mov     r10, 0
        syscall

        mov     rax, 49                 ; bind
        mov     rdi, [socket]
        mov     rsi, sock_struc
        mov     rdx, sockaddr_in_size
        syscall

        mov     rax, 50                 ; listen
        mov     rdi, [socket]
        mov     rsi, MAXKLIENT
        syscall

        mov     rax, 43                 ; accept
        mov     rdi, [socket]
        mov     rsi, sock_struc
        mov     rdx, sockaddr_in_size
        syscall

        mov     rax, 45                 ; recvfrom
        mov     rdi, [socket_client]
        mov     rsi, buf
        mov     rdx, buf_ile
        mov     r10, 0
        syscall

        ...
struc sockaddr_in
        .sin_family:    resw 1
        .sin_port:      resw 1
        .sin_addr:      resd 1
                        resb 8
endstruc

sock_struc istruc sockaddr_in
```

Funkcje htons i inet_aton są takie same, jak dla systemów 32-bitowych (bo przecież kolejność bajtów przesyłanych w sieci się nie zmienia).

Warto jeszcze wspomnieć o dwóch sprawach. Pierwsza to programy strace i ltrace. Pozwalają one na śledzenie, których funkcji systemowych i kiedy dany program używa. Jeśli coś Wam nie działa, wyłączcie tryb demona w serwerze, po czym uruchomcie `strace ./serwer` i patrzcie, na których wywołaniach funkcji są jakieś problemy. Podobnie możecie oczywiście zrobić z klientem, np. na drugim terminalu. Po szczegóły odsyłam do stron manuala.

Drugą sprawa jest dla tych z Was, którzy poważnie myślą o pisaniu aplikacji sieciowych. Jest to zbiór norm RFC (Request For Comment). Opisują one wszystkie publicznie używane protokoły, np. HTTP, SMTP czy POP3: rfc-editor.org.

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Programowanie głośniczka w asemblerze pod Linuksem

Czy nie myślicie czasem, jakby to było, gdyby można było wzbogacić swój program oprócz efektu wizualnego, także o efekt dźwiękowy?
Programowanie kart dźwiękowych (zwłaszcza tych nowoczesnych) może sprawiać niemałe kłopoty. Stary, poczciwy PC-Speaker jest jednak urządzeniem względnie prostym w programowaniu.
I to właśnie tutaj udowodnię. Najpierw troszkę teorii, potem - do dzieła!

Linuks jest systemem działającym w pełni w trybie chronionym. Dlatego bez uprawnień administratora nie możemy bezpośrednio pisać do interesujących nas portów (42h, 43h, 61h).
Na szczęście istnieje funkcja systemowa sys_ioctl (numer 54) i to ona nam pomoże w ożywieniu głośniczka systemowego. Funkcja ta jako parametry przyjmuje:

- EBX = deskryptor otwartego pliku (dla nas będzie to /dev/console lub standardowe wyjście - STDOUT) do zapisu
- ECX = stała KIOCSOUND = 0x4B2F (patrz: /usr/include/linux/kd.h)
- EDX =
    - 0 gdy chcemy wyłączyć dźwięk
    - 1234DDh / częstotliwość, gdy chcemy mieć dźwięk o żądanej częstotliwości

Ale to nie wszystko. Chcemy, by nasz dźwięk chwilę potrwał. W tym celu skorzystamy z funkcji sys_nanosleep (numer 162). Jej składnia jest prosta:

- EBX = adres struktury timespec wyglądającej tak (w składni FASM):

```
struc   timespec
 {
        .tv_sec         rd 1
        .tv_nsec        rd 1
 }
```

  i zawierającej wpisane ilości sekund i nanosekund, które należy odczekać.
- ECX = adres struktury timespec, do której funkcja zapisze wynik swojego działania.

Jak widać schemat działania naszego programu jest dość prosty:

1. Otworzyć /dev/console do zapisu. W przypadku niepowodzenia użyć STDOUT
2. Ewentualnie wywołać sys_ioctl z EDX=0 w celu wyłączenia aktualnie trwającego dźwięku
3. Tyle razy ile trzeba, wywołać sys_ioctl z odpowiednimi wartościami w EDX i korzystać z funkcji sys_nanosleep
4. Wywołać sys_ioctl z EDX=0 w celu wyłączenia dźwięku
5. Jeśli otworzyliśmy /dev/console, zamknąć ten deskryptor

Przykładowy program wygląda tak (używanie załączników z mojej biblioteki nie jest konieczne - w kodzie mówię, jak i co zamienić):

(przeskocz program)

```
; Program wytwarzający dźwięki z głośniczka przez sys_ioctl
; Autor: Bogdan D.
; Kontakt: bogdandr (at) op (dot) pl
;
; kompilacja:
;   fasm spkr.asm spkr

format ELF executable
entry _start

segment readable executable

include "bibl/incl/linuxbsd/fasm/fasm_system.inc"

KIOCSOUND       = 0x4B2F

_start:
        mov     eax, sys_open   ; sys_open = 5
        mov     ebx, konsola
        mov     ecx, O_WRONLY   ; O_WRONLY = 1
        mov     edx, 777o
        int     80h

        cmp     eax, 0          ; czy wystąpił błąd (EAX < 0) ?
        jg      .otw_ok

        mov     eax, 1          ; jak nie otworzyliśmy konsoli, piszemy
                                ; na STDOUT (1)
.otw_ok:
        mov     ebx, eax        ; EBX = uchwyt do pliku

        mov     eax, sys_ioctl  ; sys_ioctl = 54
        mov     ecx, KIOCSOUND
        xor     edx, edx        ; wyłączenie ewentualnych dźwięków
        int     80h

        mov     eax, sys_ioctl
        mov     edx, 2711       ; 2711 = 1234DDh/440. 440 Hz to dźwięk A
        int     80h

        mov     cx, 0fh
        mov     dx, 4240h       ; 0F4240h to 1 milion dziesiętnie
        call    pauza

        mov     eax, sys_ioctl
        mov     ecx, KIOCSOUND
        xor     edx, edx        ; wyłączamy dźwięk
        int     80h

        cmp     ebx, 2          ; sprawdzamy, czy używamy /dev/console
                                ; czy STDOUT
        jbe     .koniec

        mov     eax, sys_close  ; sys_close = 6
        int     80h             ; zamykamy otwarty plik konsoli

.koniec:
        mov     eax, 1
        xor     ebx, ebx
        int     80h
```

```
pauza:                          ;procedura pauzująca przez CX:DX milisekund
        push    ebx
        push    ecx
        push    edx

        mov     ax, cx
        shl     eax, 16
        mov     ebx, 1000000
        mov     ax, dx                  ; EAX = CX:DX
        xor     edx, edx
        div     ebx                     ; CX:DX dzielimy przez milion
        mov     [t1.tv_sec], eax        ; EAX = ilość sekund

        mov     ebx, 1000
        mov     eax, edx                ;EAX = pozostała ilość mikrosekund
        mul     ebx
        mov     [t1.tv_nsec], eax       ; EAX = ilość nanosekund

        mov     eax, sys_nanosleep      ; funkcja numer 162
        mov     ebx, t1
        mov     ecx, t2
        int     80h

        pop     edx
        pop     ecx
        pop     ebx

        ret

segment readable writeable

konsola         db      "/dev/console", 0

struc   timespec
 {
        .tv_sec         rd 1
        .tv_nsec        rd 1
 }

t1 timespec
t2 timespec
```

Mam nadzieję, że podałem wystarczająco informacji, abyście samodzielnie zaczęli programować głośniczek. Jeśli mi się nie udało, to zawsze możecie skorzystać z gotowej procedury z mojej biblioteki.
Jeśli program nie powoduje wydawania żadnych dźwięków, może trzeba wkompilować obsługę głośniczka do jądra (lub załadować odpowiedni moduł). Czasem mogą być potrzebne uprawnienia administratora.

To już koniec. Miłej zabawy!

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Opis funkcji systemowych syscall: 0-50

Jeśli jakaś funkcja zakończy się błędem, w RAX zwracana jest wartość ujemna z przedziału od -4096 do -1 włącznie.
Z drugiej strony, opisy funkcji na stronach manuala mówią, że zwracane jest -1, a wartość błędu jest zapisywana do zmiennej errno z biblioteki GLIBC. Dzieje się tak tylko w przypadku, gdy korzystamy z interfejsu języka C (czyli deklarujemy i uruchamiamy zewnętrzne funkcje odpowiadające wywołaniom systemowym i linkujemy nasz program z biblioteką języka C), a nie bezpośrednio z wywołań systemowych (czyli syscall).

Najaktualniejsze informacje o funkcjach systemowych można znaleźć zazwyczaj w sekcji 2 (lub 3) manuala, np. man 2 open
Najnowsze wersje stron manuala można znaleźć tu: www.kernel.org/pub/linux/docs/man-pages.

Napis ASCIIZ oznacza łańcuch znaków ASCII zakończony znakiem/bajtem Zerowym.
Jeśli potrzeba, przy każdej funkcji jest odnośnik do opisu argumentów i innych dodatkowych informacji: typów danych, wartości błędów, możliwych wartości parametrów itp.

Podstawowe funkcje syscall: 0-50

Numer/ RAX	Opis	Argumenty	Zwraca
0	Czytanie z pliku (sys_read)	RDI = deskryptor pliku RSI = adres bufora docelowego RDX = liczba bajtów do przeczytania	RAX=liczba przeczytanych bajtów RAX = błąd EAGAIN, EBADF, EFAULT, EINTR, EINVAL, EIO, EISDIR
1	Zapis do pliku (sys_write)	RDI = deskryptor pliku RSI = adres bufora źródłowego RDX = liczba bajtów do zapisania	RAX=liczba zapisanych bajtów RAX = błąd EAGAIN, EBADF, EFAULT, EINTR, EINVAL, EIO, ENOSPC, EPIPE
2	Otwarcie pliku (sys_open)	RDI = adres nazwy pliku ASCIIZ RSI = bity dostępu RDX = prawa dostępu / tryb	RAX=deskryptor pliku RAX = błąd EACCES, EEXIST, EFAULT, EISDIR, ELOOP, EMFILE, ENAMETOOLONG, ENFILE, ENOENT, ENODEV, ENODIR, ENOMEM, ENOSPC, ENXIO, EROFS, ETXTBSY
3	Zamknięcie pliku (sys_close)	RDI = deskryptor pliku	RAX = 0 RAX = błąd EBADF, EINTR, EIO

4	Pobierz status pliku (sys_stat)	RDI = adres nazwy pliku ASCIIZ. Jeśli plik jest linkiem, to zwracany jest status obiektu docelowego. RSI = adres struktury stat	RAX = 0 RAX = błąd
5	Pobierz status pliku (sys_fstat)	RDI = deskryptor otwartego pliku RSI = adres struktury stat	RAX = 0 RAX = błąd
6	Pobierz status pliku (sys_lstat)	RDI = adres nazwy pliku ASCIIZ. Jeśli plik jest linkiem, to zwracany jest status linku, a nie obiektu docelowego. RSI = adres struktury stat	RAX = 0 RAX = błąd
7	Czekaj na zdarzenia na deskryptorze (sys_poll)	RDI = adres tablicy struktur pollfd RSI = liczba struktur pollfd w tablicy RDX = max. czas na oczekiwanie w milisekundach (-1 = nieskończoność)	RAX = liczba odpowiednich deskryptorów RAX = 0, gdy czas upłynął RAX = błąd EFAULT, EINTR, EINVAL
8	Zmiana bieżącej pozycji w pliku (sys_lseek)	RDI = deskryptor pliku RSI = liczba bajtów, o którą chcemy się przesunąć RDX = odkąd zaczynamy ruch	RAX = nowa pozycja względem początku pliku RAX = błąd EBADF, EINVAL, EISPIPE
9	Mapuj plik/urządzenie do pamięci (sys_mmap)	-- zgodne z man 2 mmap-- RDI = proponowany adres początkowy RSI = długość mapowanego obszaru RDX = ochrona R10 = flagi mapowania R8 = deskryptor mapowanego pliku, jeśli mapowanie nie jest anonimowe R9 = offset początku mapowanych danych w pliku	RAX = rzeczywisty adres mapowania RAX = błąd
10	Kontrola dostępu do obszaru pamięci (sys_mprotect)	RDI = adres obszaru pamięci (wyrównany do granicy strony) RSI = długość tego obszaru	RAX=0 RAX = błąd EACCES, ENOMEM, EINVAL, EFAULT

		w bajtach (względem strony pamięci) RDX = bity włączające ochrone	
11	Odmapuj plik/urządzenie z pamięci (sys_munmap)	RDI = adres początkowy obszaru RSI = ilość bajtów	RAX = 0 RAX = błąd
12	Alokacja i dealokacja pamięci (sys_brk)	RDI = 0, aby poznać aktualny najwyższy adres sekcji .bss RDI = (wirtualny) adres nowego wierzchołka .bss, powyżej spodu sekcji danych i poniżej bibliotek	RAX = nowy najwyższy adres RAX = błąd ENOMEM
13	Pobierz i zmień procedurę obsługi sygnału (sys_rt_sigaction)	RDI = numer sygnału RSI = adres struktury sigaction opisującą bieżącą procedurę RDX = adres struktury sigaction opisującą starą procedurę R10 = rozmiar struktury sigset_t	RAX = 0 RAX=błąd EINVAL, EFAULT
14	Pobierz i zmień blokowane sygnały (sys_rt_sigprocmask)	RDI = działanie RSI = adres zestawu sygnałów (tablicy 32 DWORDów) RDX = adres zestawu sygnałów, który otrzyma starą maskę sygnałów R10 = rozmiar struktury sigset_t	RAX = 0 RAX=błąd EINVAL, EFAULT
15	Powrót z procedury obsługi sygnału (sys_rt_sigreturn)	-- funkcja wewnętrzna, nie używać-- RDI = parametr zależny od architektury	nigdy nie powraca
16	Manipulacja urządzeniem znakowym (sys_ioctl)	RDI = deskryptor pliku RSI = kod komendy (man 2 ioctl_list) RDX = adres zapisywalnego obszaru danych lub innej struktury, zależy od komendy	RAX = 0 RAX = błąd EBADF, EFAULT, EINVAL, ENOTTY
17	Czytaj z danej pozycji w pliku (sys_pread/sys_pread64)	RDI = deskryptor otwartego pliku RSI = adres bufora, który otrzyma dane RDX = ilość bajtów do	RAX = ilość przeczytanych bajtów (wskaźnik pozycji w pliku pozostaje bez zmian) RAX = błąd (jak w sys_read)

		odczytania R10 = pozycja, z której zacząć czytanie	
18	Zapisuj na danej pozycji w pliku (sys_pwrite/sys_pwrite64)	RDI = deskryptor otwartego pliku RSI = adres bufora, z którego pobierać dane do zapisania RDX = ilość bajtów do zapisania R10 = pozycja, od której zacząć zapisywanie	RAX = ilość zapisanych bajtów (wskaźnik pozycji w pliku pozostaje bez zmian) RAX = błąd (jak w sys_read)
19	Czytaj wektor (sys_readv)	RDI = deskryptor otwartego obiektu, z którego będą czytane dane RSI = adres tablicy struktur iovec RDX = liczba struktur iovec, do których będą czytane dane	RAX = 0 RAX = błąd EWOULDBLOCK, EBADF, EINTR, EINVAL, ENOLCK
20	Zapisz wektor (sys_writev)	RDI = deskryptor otwartego obiektu, do którego będą zapisane dane RSI = adres tablicy struktur iovec RDX = liczba struktur iovec, z których będą czytane dane do zapisania	RAX = 0 RAX = błąd EWOULDBLOCK, EBADF, EINTR, EINVAL, ENOLCK
21	Sprawdź uprawnienia dostępu do pliku (sys_access)	RDI = adres nazwy pliku ASCIIZ RSI = prawa dostępu / tryb (wartości R_OK, W_OK, X_OK)	RAX = 0 RAX = błąd - każdy związany z systemem plików i plikami
22	Utwórz potok (sys_pipe)	RDI = adres tablicy dwóch DWORDów	RAX = 0 i pod [RDI]: deskryptor odczytu z potoku fd(0) pod [RDI], deskryptor zapisu do potoku fd(1) pod [RDI+4] RAX = błąd EFAULT, EMFILE, ENFILE
23	Oczekiwanie zmiany stanu deskryptoru(ów) (sys_select)	RDI = najwyższy numer spośród deskryptorów + 1 RSI = adres tablicy deskryptorów sprawdzanych, czy można z nich czytać RDX = adres tablicy deskryptorów	RAX = całkowita liczba deskryptorów, ktora pozostała w tablicach RAX = 0, gdy skończył się czas RAX = wystąpił błąd

		sprawdzanych, czy można do nich pisać R10 = adres tablicy deskryptorów sprawdzanych, czy nie wystąpił u nich wyjątek R8 = adres struktury timeval zawierającą maksymalny czas oczekiwania	
24	Oddanie procesora innym procesom (sys_sched_yield)	nic	RAX = 0 RAX = błąd.
25	Przemapuj adres wirtualny (sys_mremap)	RDI = stary adres RSI = rozmiar obszaru do przemapowania RDX = żądany rozmiar R10 = zero lub flagi przemapowania R8 = nowy adres, jeśli dano flagę MREMAP_FIXED	RAX = wskaźnik do nowego obszaru RAX = sygnał lub błąd EFAULT, EAGAIN, ENOMEM, EINVAL
26	Synchronizuj mapowany plik z pamięcią (sys_msync)	RDI = adres do zrzucenia na dysk (zostaną zrzucone zmodyfikowane strony pamięci zawierającej ten adres i co najwyżej RSI-1 zmienionych następnych) RSI = ilość bajtów/rozmiar obszaru do zrzucenia na dysk RDX = 0 lub zORowane flagi	RAX = 0 RAX = błąd EBUSY, EIO, ENOMEM, EINVAL, ENOLCK
27	Pobierz informację, czy strony pamięci są w rdzeniu procesu (sys_mincore)	RDI = adres początkowy sprawdzanych bajtów RSI = liczba sprawdzanych bajtów RDX = adres tablicy bajtów zdolnej pomieścić tyle bajtów, ile stron pamięci jest sprawdzanych. Najmłodszy bit w każdym bajcie będzie mówił o tym, czy dana strona pamięci jest obecna (=1), czy zrzucona na dysk (=0)	RAX = 0 RAX = błąd EAGAIN, EINVAL, EFAULT, ENOMEM
28	Porada dla jądra o uzyciu pamięci (sys_madvise, sys_madvise1)	RDI = adres początkowy bajtów, których dotoczy porada	RAX = 0 RAX = błąd EAGAIN, EINVAL, EFAULT, ENOMEM

		RSI = liczba tych bajtów RDX = porada	
29	Alokuj współdzielony segment pamięci (sys_shmget)	RDI = klucz opisujący segment pamięci RSI = długość segmentu RDX = flagi shmget	RAX = identyfiator segmentu RAX = błąd EACCES, EEXIST, EINVAL, ENFILE, ENOENT, ENOMEM, ENOSPC, EPERM
30	Podłącz współdzielony segment pamięci (sys_shmat)	RDI = identyfiator segmentu pamięci RSI = adres podłączenia segmentu do pamięci procesu lub 0 RDX = flagi shmat	RAX = adres podłączonego segmentu RAX = błąd EACCES, EINVAL, ENOMEM
31	Kontrola współdzielonej pamięci (sys_shmctl)	RDI = identyfiator segmentu pamięci RSI = rozkaz RDX = adres struktury shmid_ds	RAX = wartość odpowiednia dla rozkazu RAX = błąd EACCES, EINVAL, EFAULT, EIDRM, ENOMEM, EOVERFLOW, EPERM
32	Zduplikuj deskryptor pliku (sys_dup)	RDI = stary deskryptor	RAX = nowy deskryptor RAX = błąd EBADF, EMFILE
33	Zamień deskryptor zduplikowanym deskryptorem pliku (sys_dup2)	RDI = deskryptor do zduplikowania RSI = deskryptor, do którego powinien być przyznany duplikat	RAX = zduplikowany deskryptor RAX = błąd EBADF, EMFILE, EBUSY, EINTR
34	Pauza - śpij aż do otrzymania sygnału (sys_pause)	nic	wraca tylko po sygnale, o ile procedura jego obsługi ma powrót. RAX = EINTR po sygnale
35	Pauza w wykonywaniu programu (sys_nanosleep)	RDI = adres struktury timespec RSI = NULL lub adres modyfikowalnej struktury timespec, która otrzyma resztkę czasu, która została	RAX = 0 RAX = sygnał lub błąd EINTR, EINVAL
36	Pobierz wartość czasomierza (sys_getitimer)	RDI = numer czasomierza RSI = adres struktury itimerval, która otrzyma wartość czasomierza	RAX = 0 RAX = błąd
37	Alarm - wysłanie sygnału SIGALARM (sys_alarm)	RDI = sekundy	RAX = 0 lub liczba sekund do wykonania poprzednich alarmów

38	Ustaw wartość czasomierza (sys_setitimer)	RDI = numer czasomierza RSI = adres struktury itimerval zawierającej nową wartość czasomierza RSI = adres struktury itimerval, która otrzyma starą wartość czasomierza	RAX = 0 RAX = błąd
39	Pobierz identyfikator bieżącego procesu (sys_getpid)	nic	RAX = PID bieżącego procesu
40	Kopiuj dane między deskryptorami plików (sys_sendfile, sys_sendfile64)	RDI = deskryptor pliku wyjściowego, otwartego do zapisu RSI = deskryptor pliku wejściowego RDX = adres 64-bitowej zmiemnnej - numeru bajtu w pliku źródłowym, od którego zacząć kopiować R10 = liczba bajtów do skopiowania	RAX = liczba zapisanych bajtów RAX = błąd EBADF, EAGAIN, EINVAL, ENOMEM, EIO, EFAULT
41	Tworzenie gniazda (sys_socket)	RDI = domena tworzonego gniazda RSI = typ tworzonego gniazda RDX = protokół dla tworzonego gniazda (zwykle 0, patrz: man 5 protocols, man 3 getprotoent)	RAX = deskryptor nowego gniazda RAX = błąd EINVAL, EACCES, EAFNOSUPPORT, EMFILE, ENFILE, ENOBUFS/ENOMEM, EPROTONOSUPPORT
42	Połączenie gniazda (sys_connect)	RDI = deskryptor gniazda RSI = adres struktury sockaddr (zależnej od protokołu), opisującej adres, z którym się łaczymy RDX = wielkość struktury, na którą wskazuje RSI	RAX = 0 RAX = błąd EACCES, EPERM, EADDRINUSE, EAFNOSUPPORT, EAGAIN, EALREADY, EBADF, ECONNREFUSED, EFAULT, EINPROGRESS, EINTR, EISCONN, ENETUNREACH, ENOTSOCK, ETIMEDOUT
43	Przyjmowanie połączenia na gnieździe (sys_accept)	RDI = deskryptor gniazda RSI = adres struktury sockaddr (zależnej od protokołu), która otrzyma adres, z którego połączono się do tego gniazda RDX = adres zmiennej (32-bitowej), która otrzyma	RAX = deskryptor gniazda do połączonego klienta RAX = błąd EAGAIN/EWOULDBLOCK, EBADF, ECONNABORTED, EFAULT, EINTR, EINVAL, EMFILE, ENFILE, ENOBUFS, ENOMEM, ENOTSOCK, EOPNOTSUPP, EPROTO, EPERM

		wielkość struktury, na którą wskazuje RSI	
44	Wysyłanie danych gniazdem (sys_sendto)	RDI = deskryptor gniazda RSI = adres bufora z danymi do wysłania RDX = liczba bajtów z bufora do wysłania R10 = flagi sendto R8 = adres struktury sockaddr (zależnej od protokołu), opisującej adres, do którego wysłać dane. Ignorowane dla gniazd opartych o połączenie R9 = wielkość struktury, na którą wskazuje R8. Ignorowane dla gniazd opartych o połączenie	RAX = liczba wysłanych znaków RAX = błąd EACCES, EAGAIN, EWOULDBLOCK, EBADF, ECONNRESET, EDESTADDRREQ, EFAULT, EINTR, EINVAL, EISCONN, EMSGSIZE, ENOBUFS, ENOMEM, ENOTCONN, ENOTSOCK, EOPNOTSUPP, EPIPE
45	Odbieranie danych gniazdem (sys_recvfrom)	RDI = deskryptor gniazda RSI = adres bufora na odebrane danw RDX = liczba bajtów w buforze R10 = flagi recvfrom R8 = adres struktury sockaddr (zależnej od protokołu), która otrzyma adres, z którego otrzymano dane. Może być 0. R9 = adres zmiennej (32-bitowej), która otrzyma wielkość struktury, na którą wskazuje R8. Może być 0.	RAX = liczba odebranych znaków RAX = błąd EAGAIN, EWOULDBLOCK, EBADF, ECONNREFUSED, EFAULT, EINTR, EINVAL, ENOMEM, ENOTCONN, ENOTSOCK
46	Wysyłanie wiadomości gniazdem (sys_sendmsg)	RDI = deskryptor gniazda RSI = adres struktury msghdr opisującej wiadomość RDX = flagi sendmsg (te same, co dla sendto)	RAX = liczba wysłanych znaków RAX = błąd EACCES, EAGAIN, EWOULDBLOCK, EBADF, ECONNRESET, EDESTADDRREQ, EFAULT, EINTR, EINVAL, EISCONN, EMSGSIZE, ENOBUFS, ENOMEM, ENOTCONN, ENOTSOCK, EOPNOTSUPP, EPIPE
47	Odbieranie wiadomości gniazdem (sys_recvmsg)	RDI = deskryptor gniazda RSI = adres struktury msghdr opisującej wiadomość RDX = flagi recvmsg (te same, co dla recvfrom)	RAX = liczba odebranych znaków RAX = błąd EAGAIN, EWOULDBLOCK, EBADF, ECONNREFUSED, EFAULT, EINTR, EINVAL, ENOMEM, ENOTCONN, ENOTSOCK

48	Zamknięcie części połączenia (sys_shutdown)	RDI = deskryptor gniazda RSI = sposób zamknięcia (SHUT_RD=0 blokuje odczyty, SHUT_WR=1 blokuje zapisy, SHUT_RDWR=2 blokuje odczyty i zapisy)	RAX = 0 RAX = błąd EBADF, ENOTCONN, ENOTSOCK
49	Przypisanie gniazda do adresu (sys_bind)	RDI = deskryptor gniazda RSI = adres struktury sockaddr (zależnej od protokołu), opisującej adres, do którego chcemy przypisać gniazdo RDX = wielkość struktury, na którą wskazuje RSI	RAX = 0 RAX = błąd EACCES, EADDRINUSE, EBADF, EINVAL, ENOTSOCK, EADDRNOTAVAIL, EFAULT, ELOOP, ENAMETOOLONG, ENOENT, ENOMEM, ENOTDIR, EROFS
50	Oczekiwanie na połączenia na gnieździe (sys_listen)	RDI = deskryptor gniazda RSI = maksymalna liczba połączeń oczekujących	RAX = 0 RAX = błąd EADDRINUSE, EBADF, ENOTSOCK, EOPNOTSUPP

Kolejna część (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Opis funkcji systemowych syscall: 51-100

Jeśli jakaś funkcja zakończy się błędem, w RAX zwracana jest wartość ujemna z przedziału od -4096 do -1 włącznie.
Z drugiej strony, opisy funkcji na stronach manuala mówią, że zwracane jest -1, a wartość błędu jest zapisywana do zmiennej errno z biblioteki GLIBC. Dzieje się tak tylko w przypadku, gdy korzystamy z interfejsu języka C (czyli deklarujemy i uruchamiamy zewnętrzne funkcje odpowiadające wywołaniom systemowym i linkujemy nasz program z biblioteką języka C), a nie bezpośrednio z wywołań systemowych (czyli syscall).

Najaktualniejsze informacje o funkcjach systemowych można znaleźć zazwyczaj w sekcji 2 (lub 3) manuala, np. man 2 open
Najnowsze wersje stron manuala można znaleźć tu: www.kernel.org/pub/linux/docs/man-pages.

Napis ASCIIZ oznacza łańcuch znaków ASCII zakończony znakiem/bajtem Zerowym.
Jeśli potrzeba, przy każdej funkcji jest odnośnik do opisu argumentów i innych dodatkowych informacji: typów danych, wartości błędów, możliwych wartości parametrów itp.

Podstawowe funkcje syscall: 51-100

Numer/ RAX	Opis	Argumenty	Zwraca
51	Pobierz nazwę gniazda (sys_getsockname)	RDI = deskryptor gniazda RSI = adres struktury sockaddr (zależnej od protokołu), opisującej adres, w którym system zwróci adres, do którego gniazdo jest przypisane RDX = adres zmiennej (32-bitowej), która otrzyma wielkość struktury, na którą wskazuje RSI	RAX = 0 RAX = błąd EBADF, EFAULT, EINVAL, ENOBUFS, ENOTSOCK
52	Pobierz nazwę klienta podłączonego do gniazda (sys_getpeername)	RDI = deskryptor gniazda RSI = adres struktury sockaddr (zależnej od protokołu), opisującej adres, w którym system zwróci adres klienta podłączonego do tego gniazda RDX = adres zmiennej (32-bitowej), która otrzyma wielkość struktury, na którą wskazuje RSI	RAX = 0 RAX = błąd EBADF, EFAULT, EINVAL, ENOBUFS, ENOTCONN, ENOTSOCK
53	Stwórz parę połączonych gniazd (sys_socketpair)	RDI = domena tworzonego gniazda RSI = typ tworzonego gniazda RDX = protokół dla tworzonego gniazda (zwykle 0, patrz: man 5 protocols, man 3 getprotoent) R10 = adres tablicy dwóch DWORDów, w których zostaną umieszczone deskryptory nowych	RAX = 0 RAX = błąd EAFNOSUPPORT, EFAULT, EMFILE, ENFILE, EOPNOTSUPP, EPROTONOSUPPORT

		gniazd	
54	Ustaw opcje gniazda (sys_setsockopt)	RDI = deskryptor gniazda RSI = poziom, do którego odnosi się opcja: SOL_SOCKET=1 dla gniazda lub numer protokołu (patrz: man 5 protocols, man 3 getprotoent) RDX = nazwa opcji (odpowiednia dla protokołu) R10 = adres zmiennej lub zmiennych z wartościami opcji (odpowiednich dla protokołu) R8 = liczba wartości opcji w zmiennej, której adres jest w R10	RAX = 0 RAX = błąd EBADF, EFAULT, EINVAL, ENOPROTOOPT, ENOTSOCK
55	Pobierz opcje gniazda (sys_getsockopt)	RDI = deskryptor gniazda RSI = poziom, do którego odnosi się opcja: SOL_SOCKET=1 dla gniazda lub numer protokołu (patrz: man 5 protocols, man 3 getprotoent) RDX = nazwa opcji (odpowiednia dla protokołu) R10 = adres zmiennej lub zmiennych, które otrzymają wartości opcji R8 = liczba miejsc na wartości opcji w zmiennej, której adres jest w R10	RAX = 0 RAX = błąd EBADF, EFAULT, EINVAL, ENOPROTOOPT, ENOTSOCK
56	Utwórz klon procesu (sys_clone)	RDI = flagi klonowania RSI = wkaźnik na oddzielny stos klona RDX = wskaźnik na strukturę pt_regs lub 0	RAX = numer PID klona lub RAX = błąd EAGAIN, ENOMEM, EINVAL, EPERM
57	Uruchomienie nowego procesu (sys_fork)	RDI = adres struktury pt_regs	RAX = id procesu potomnego (PID) RAX=błąd EAGAIN, ENOMEM
58	Utwórz proces potomny i zablokuj rodzica (sys_vfork)	nic	RAX = PID procesu potomnego RAX = błąd EAGAIN, ENOMEM
59	Uruchomienie innego programu (sys_execve)	RDI=adres nazwy (ze ścieżką) programu ASCIIZ RSI = adres zakończonej dwordem 0 listy adresów argumentów uruchamianego programu ASCIIZ RDX = adres zakończonej dwordem 0 listy adresów zmiennych środowiska dla uruchamianego programu ASCIIZ	nie wraca do programu wywołującego RAX = błąd E2BIG, EACCES, EINVAL, EOIO, EISDIR, ELIBBAD, ELOOP, ENFILE, ENOEXEC, ENOENT, ENOMEM, ENOTDIR, EFAULT, ENAMETOOLONG, EPERM, ETXTBUSY

60	Wyjście z programu (sys_exit)	RDI = kod wyjścia (errorlevel)	nie wraca do programu wywołującego
61	Czekaj na zakończenie procesu (sys_wait4)	RDI = PID procesu potomnego lub specyfikacja RSI = NULL lub adres zmiennej DWORD, która otrzyma status RDX = opcje R10 = adres struktury rusage	RAX = PID zakończonego procesu RAX = błąd RAX = 0 dla WNOHANG
62	Wyślij sygnał do procesu (sys_kill)	RDI = numer PID procesu (patrz też specyfikacja) RSI = numer sygnału	RAX = 0 RAX = błąd EINVAL, EPERM, ESRCH
63	Pobierz informację o jądrze (sys_uname)	RDI = adres struktury utsname	RAX = 0 RAX = błąd EINVAL, EPERM, EFAULT
64	Pobierz identyfikator zbioru semaforów (sys_semget)	RDI = klucz identyfikujący RSI = liczba semaforów do utworzenia, gdy klucz ma wartość IPC_PRIVATE=20 lub gdy nie ma jeszcze semaforów powiązanych z tym kluczem, a flagi zawierają IPC_CREAT RDX = flagi: 0 lub zORowane wartości IPC_CREAT=00001000 (ósemkowo) i IPC_EXCL=02000 (ósemkowo), a dodatkowo przy tworzeniu semaforów, 9 najmłodszych bitów to uprawnienia (takie same, jak w funkcji open)	RAX = identyfikator zestawu semaforów RAX = błąd EACCES, EEXIST, EINVAL, ENOENT, ENOMEM, ENOSPC
65	Operacje na semaforze (sys_semop)	RDI = identyfikator zestawu semaforów RSI = adres tablicy struktur sembuf RDX = liczba elementów w tablicy spod RSI	RAX = 0 RAX = błąd E2BIG, EACCES, EAGAIN, EFAULT, EFBIG, EIDRM, EINTR, EINVAL, ENOMEM, ERANGE
66	Kontrola semaforu (sys_semctl)	RDI = identyfikator zestawu semaforów RSI = numer semafora RDX = komenda dla semaforów R10 = unia semun (przekazywany jest jeden z elementów)	RAX = wynik komendy RAX = błąd EACCES, EFAULT, EIDRM, EINVAL, EPERM, ERANGE
67			

	Odłącz współdzielony segment pamięci (sys_shmdt)	RDI = adres współdzielonego obszaru pamięci (zwrócony przez shmat)	RAX = 0 RAX = błąd EINVAL
68	Pobierz identyfikator kolejki wiadomości (sys_msgget)	RDI = klucz identyfikujący. Tworzona jest nowa kolejka gdy klucz ma wartość IPC_PRIVATE=20 lub gdy nie ma jeszcze kolejki powiązanej z tym kluczem a flagi zawierają IPC_CREAT RSI = flagi: 0 lub zORowane wartości IPC_CREAT=00001000 (ósemkowo) i IPC_EXCL=02000 (ósemkowo), a dodatkowo przy tworzeniu kolejki, 9 najmłodszych bitów to uprawnienia (takie same, jak w funkcji open)	RAX = identyfikator kolejki wiadomości RAX = błąd EACCES, EEXIST, ENOENT, ENOMEM, ENOSPC
69	Wyślij wiadomość do kolejki wiadomości (sys_msgsnd)	RDI = identyfikator kolejki wiadomości RSI = adres struktury z wiadomością do wysłania RDX = liczba bajtów do wysłania R10 = flagi: 0 lub IPC_NOWAIT=04000 (ósemkowo)	RAX = 0 RAX = błąd EACCES, EAGAIN, EFAULT, EIDRM, EINTR, EINVAL, ENOMEM
70	Odbierz wiadomość z kolejki wiadomości (sys_msgrcv)	RDI = identyfikator kolejki wiadomości RSI = adres struktury wiadomości, która otrzyma odebraną wiadomość RDX = maksymalna liczba bajtów do odebrania R10 = typ wiadomości (0 oznacza pierwszą, R10 > 0 oznacza pierwszą tego typu, R10 < 0 oznacza pierwszą o typie niewiększym od bezwzględnej wartości R10) R8 = flagi msgrcv	RAX = liczba bajtów zapisanych do bufora RAX = błąd E2BIG, EACCES, EAGAIN, EFAULT, EIDRM, EINTR, EINVAL, ENOMSG
71	Kontrola kolejki wiadomości (sys_msgctl)	RDI = identyfikator kolejki wiadomości RSI = komenda RDX = adres struktury msqid_ds	RAX = wynik komendy RAX = błąd EACCES, EFAULT, EIDRM, EINVAL, EPERM
72	Kontrola nad deskryptorem pliku (sys_fcntl)	RDI = deskryptor pliku RSI = kod komendy RDX zależy od komendy	RAX zależy od komendy RAX = błąd EACCES, EAGAIN, EBADF, EDEADLK, EFAULT, EINTR, EINVAL, EMFILE, ENOLOCK, EPERM
73	Zmień blokowanie plików (sys_flock)	RDI = deskryptor otwartego pliku RSI = operacja do wykonania	RAX = 0 RAX = błąd EWOULDBLOCK,

			EBADF, EINTR, EINVAL, ENOLCK
74	Zapisz pamięć podręczną na dysk (sys_fsync)	RDI = deskryptor pliku, który ma być zsynchronizowany na dysk	RAX = 0 RAX = błąd
75	Zapisz bufory danych pliku na dysk (sys_fdatasync)	RDI = deskryptor pliku, którego DANE będą zsynchronizowane (ale np. czas dostępu nie będzie zmieniony)	RAX = 0 RAX = błąd EBADF, EIO, EROFS
76	Skróć plik (sys_truncate)	RDI = adres nazwy pliku ASCIIZ RSI = ilość bajtów, do której ma zostać skrócony plik	RAX = 0 RAX = błąd
77	Skróć plik (sys_ftruncate)	RDI = deskryptor pliku otwartego do zapisu RSI = ilość bajtów, do której ma zostać skrócony plik	RAX = 0 RAX = błąd
78	Pobierz wpisy o katalogach (sys_getdents)	RDI = deskryptor otwartego katalogu RSI = adres obszaru pamięci na struktury dirent RDX = rozmiar obszaru pamięci pod [RSI]	RAX = 0 RAX = błąd EBADF, EFAULT, EINVAL, ENOENT, ENOTDIR
79	Pobierz bieżący katalog roboczy (sys_getcwd)	RDI = adres bufora, który otrzyma ścieżkę RSI = długość tego bufora	RAX = RDI RAX=NULL, gdy błąd ERANGE, EACCES, EFAULT, EINVAL, ENOENT
80	Zmiana katalogu (sys_chdir)	RDI = adres nazwy nowego katalogu ASCIIZ	RAX = 0 RAX = błąd EACCES, EBADF, EFAULT, EIO, ELOOP, ENAMETOOLONG, ENOENT, ENOMEM, ENOTDIR
81	Zmień katalog roboczy (sys_fchdir)	RDI = deskryptor otwartego katalogu	RAX = 0 RAX = błąd EBADF, EACCES i inne
82	Przenieś plik/Zmień nazwę pliku (sys_rename)	RDI=adres starej nazwy (i ewentualnie ścieżki) ASCIIZ RSI=adres nowej nazwy (i ewentualnie ścieżki) ASCIIZ	RAX = 0 RAX = błąd EBUSY, EEXIST, EISDIR, ENOTEMPTY, EXDEV (i inne błędy systemu plików)

83	Utwórz katalog (sys_mkdir)	RDI = adres ścieżki/nazwy ASCIIZ RSI = prawa dostępu / tryb	RAX = 0 RAX = błąd - każdy związany z systemem plików lub prawami dostępu
84	Usuń katalog (sys_rmdir)	RDI = adres ścieżki/nazwy ASCIIZ	RAX = 0 RAX = błąd EACCES, EBUSY, EFAULT, ELOOP, ENAMETOOLONG, ENOENT, ENOMEM, ENOTDIR, ENOTEMPTY, EPERM, EROFS
85	Utworzenie pliku (sys_creat, nie create!)	RDI = adres nazwy pliku ASCIIZ RSI = prawa dostępu / tryb	RAX = deskryptor pliku RAX = błąd EACCES, EEXIST, EFAULT, EISDIR, ELOOP, EMFILE, ENAMETOOLONG, ENFILE, ENOENT, ENODEV, ENODIR, ENOMEM, ENOSPC, ENXIO, EROFS, ETXTBSY
86	Utworzenie twardego dowiązania do pliku (sys_link)	RDI = adres nazwy istniejącego pliku ASCIIZ RSI = adres nazwy nowego pliku ASCIIZ	RAX = 0 RAX=błąd EACCES, EIO, EPERM, EEXIST, EFAULT, ELOOP, EMLINK, ENAMETOOLONG, ENOENT, ENOMEM, ENOSPC, ENOTDIR, EROFS, EXDEV
87	Usunięcie pliku (sys_unlink)	RDI = adres nazwy pliku ASCIIZ	RAX = 0 RAX=błąd EACCES, EFAULT, EIO, EISDIR, ELOOP, ENOENT, ENAMETOOLONG, ENOMEM, ENOTDIR, EPERM, EROFS
88	Stwórz dowiązanie symboliczne do pliku (sys_symlink)	RDI = adres nazwy pliku ASCIIZ RSI = adres nazwę linku ASCIIZ	RAX = 0 RAX = błędy związane z uprawnieniami lub systemem plików
89			

	Przeczytaj zawartość linku symbolicznego (sys_readlink)	RDI = adres nazwy dowiązania symbolicznego ASCIIZ RSI = adres bufora, który otrzyma pzreczytaną informację RDX = długość bufora	RAX = liczba przeczytanych znaków RAX = błąd
90	Zmiana uprawnień (sys_chmod)	RDI = adres nazwy pliku/katalogu ASCIIZ RSI = nowe prawa dostępu	RAX = 0 RAX = błąd EACCES, EBADF, EFAULT, EIO, ELOOP, ENAMETOOLONG, ENOENT, ENOMEM, ENOTDIR, EPERM, EROFS
91	Zmiana uprawnień (sys_fchmod)	RDI = deskryptor otwartego pliku RSI = nowe prawa dostępu	RAX = 0 RAX = błąd
92	Zmiana właściciela pliku (sys_chown)	RDI=adres ścieżki do pliku RSI = UID nowego właściciela RDX = GID nowej grupy	RAX = 0 RAX = błąd np. EPERM, EROFS, EFAULT, ENOENT, ENAMETOOLONG, ENOMEM, ENOTDIR, EACCES, ELOOP
93	Zmiana właściciela (sys_fchown)	RDI = deskryptor otwartego pliku RSI = nowy numer użytkownika RDX = nowy numer grupy	RAX = 0 RAX = błąd
94	Zmiana właściciela (sys_lchown)	RDI = adres nazwy pliku/katalogu ASCIIZ RSI = nowy numer użytkownika RDX = nowy numer grupy	RAX = 0 RAX = błąd EPERM, EROFS, EFAULT, ENAMETOOLONG, ENOENT, ENOMEM, ENOTDIR, EACCES, ELOOP i inne
95	Ustaw maskę uprawnień przy tworzeniu plików (sys_umask)	RDI = maska, patrz prawa dostępu / tryb Gdy tworzymy plik o uprawnieniach X, naprawdę ma on uprawnienia X AND (NOT umask)	RAX = poprzednia umask
96	Pobierz czas (sys_gettimeofday)	RDI = adres struktury timeval RSI = adres struktury timezone	RAX = 0 i wynik zapisany w strukturach RAX = błąd EFAULT, EINVAL, EPERM
97	Pobierz limity zasobów (sys_getrlimit)	RDI = numer zasobu RSI = adres struktury rlimit	RAX = 0 RAX = błąd EFAULT, EINVAL,

			EPERM
98	Pobierz zużycie zasobów (sys_getrusage)	RDI = numer użytkownika (who) RSI = adres struktury rusage	RAX = 0 RAX = błąd EFAULT, EINVAL, EPERM
99	Pobierz statystyki systemowe (sys_sysinfo)	RDI = adres struktury sysinfo	RAX = 0 RAX = błąd
100	Pobierz czasy procesów (sys_times)	RDI = adres struktury tms	RAX = liczba taktów zegara EAX = błąd

Poprzednia część (Alt+3)
Kolejna część (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Opis funkcji systemowych syscall: 101-150

Jeśli jakaś funkcja zakończy się błędem, w RAX zwracana jest wartość ujemna z przedziału od -4096 do -1 włącznie.
Z drugiej strony, opisy funkcji na stronach manuala mówią, że zwracane jest -1, a wartość błędu jest zapisywana do zmiennej errno z biblioteki GLIBC. Dzieje się tak tylko w przypadku, gdy korzystamy z interfejsu języka C (czyli deklarujemy i uruchamiamy zewnętrzne funkcje odpowiadające wywołaniom systemowym i linkujemy nasz program z biblioteką języka C), a nie bezpośrednio z wywołań systemowych (czyli syscall).

Najaktualniejsze informacje o funkcjach systemowych można znaleźć zazwyczaj w sekcji 2 (lub 3) manuala, np. man 2 open
Najnowsze wersje stron manuala można znaleźć tu: www.kernel.org/pub/linux/docs/man-pages.

Napis ASCIIZ oznacza łańcuch znaków ASCII zakończony znakiem/bajtem Zerowym.
Jeśli potrzeba, przy każdej funkcji jest odnośnik do opisu argumentów i innych dodatkowych informacji: typów danych, wartości błędów, możliwych wartości parametrów itp.

Podstawowe funkcje syscall: 101-150

Numer/ RAX	Opis	Argumenty	Zwraca
101	Śledzenie procesu (sys_ptrace)	RDI = żądane działanie RSI = identyfikator PID żądanego procesu RDX = adres w procesie docelowym R10 = adres w procesie śledzącym	RAX zależne od działania RAX = błąd EIO, EFAULT, EPERM, ESRCH
102	Pobierz identyfikator użytkownika (sys_getuid)	nic	RAX = numer UID
103	Opcje logowania (sys_syslog)	RDI = komenda syslog RSI = adres bufora znakowego RDX = ilość bajtów (patrz opis RDI)	RAX = ilość bajtów (patrz opis RDI) lub 0 RAX = błąd EINVAL, EPERM, ERESTARTSYS, ENOSYS
104	Pobierz ID grupy bieżącego procesu (sys_getgid)	nic	RAX = ID grupy
105	Ustaw identyfikator użytkownika (sys_setuid)	RDI = nowy UID	RAX = 0 RAX = błąd EPERM

106	Ustaw ID grupy bieżącego procesu (sys_setgid)	RDI = nowy ID grupy	RAX = 0 RAX = błąd EPERM
107	Pobierz efektywne ID użytkownika bieżącego procesu (sys_geteuid)	nic	RAX = UID
108	Pobierz efektywne ID grupy bieżącego procesu (sys_getegid)	nic	RAX = GID
109	Ustaw ID grupy procesu (sys_setpgid)	RDI = ID procesu (PID) RSI = ID grupy	RAX = 0 RAX = błąd EINVAL, EPERM, ESRCH
110	Pobierz PID procesu rodzica (sys_getppid)	nic	RAX = PID rodzica
111	Pobierz ID grupy procesu rodzica (sys_getpgrp)	nic	RAX = GID rodzica RAX=błąd EINVAL, EPERM, ESRCH
112	Stwórz sesję, ustaw ID grupy (sys_setsid)	nic	RAX = ID procesu uruchamiającego RAX=błąd EPERM
113	Ustaw realny i efektywny ID użytkownika (sys_setreuid)	RDI = realny ID użytkownika (UID) RSI = efektywny UID	RAX = 0 RAX = błąd EPERM
114	Ustaw realny i efektywny ID grupy (sys_setregid)	RDI = realny ID grupy (GID) RSI = efektywny GID	RAX = 0 RAX = błąd EPERM
115	Pobierz liczbę dodatkowych grup (sys_getgroups)	RDI = rozmiar tablicy z RSI RSI = adres tablicy, gdzie zostaną zapisane GID-y (DWORDY) grup dodatkowych	RAX = liczba dodatkowych grup procesu RAX = błąd EFAULT, EINVAL, EPERM
116	Ustaw liczbę dodatkowych grup (sys_setgroups)	RDI = rozmiar tablicy z RSI RSI = adres tablicy, gdzie zawierającą GID-y (DWORDY)	RAX = 0 RAX = błąd EFAULT, EINVAL, EPERM

117	Ustaw różne ID użytkownika (sys_setresuid)	RDI = realny UID lub -1 (wtedy jest bez zmian) RSI = efektywny UID lub -1 (bez zmian) RDX = zachowany (saved) UID lub -1 (bez zmian)	RAX = 0 RAX = błąd EPERM
118	Pobierz różne ID użytkownika (sys_getresuid)	RDI = adres DWORDa, który otrzyma realny UID RSI = adres DWORDa, który otrzyma efektywny UID RDX = adres DWORDa, który otrzyma zachowany UID	RAX = 0 RAX = błąd EFAULT
119	Ustaw realny, efektywny i zachowany ID grupy (sys_setresgid)	RDI = realny GID RSI = efektywny GID RDX = zachowany (saved) GID	RAX = 0 RAX = błąd EPERM
120	Pobierz realny, efektywny i zachowany ID grupy (sys_getresgid)	RDI = adres DWORDa, który otrzyma realny GID RSI = adres DWORDa, który otrzyma efektywny GID RDX = adres DWORDa, który otrzyma zachowany (saved) GID	RAX = 0 RAX = błąd EFAULT
121	Pobierz ID grupy procesów dla danego procesu (sys_getpgid)	RDI = PID danego procesu	RAX = ID grupy procesów RAX = błąd ESRCH
122	Ustal UID przy sprawdzaniu systemów plików (sys_setfsuid)	RDI = nowy ID użytkownika	RAX = stary UID (zawsze)
123	Ustal GID przy sprawdzaniu systemów plików (sys_setfsgid)	RDI = nowy ID grupy	RAX = stary GID (zawsze)
124	Pobierz ID sesji dla procesu (sys_getsid)	RDI = PID procesu, którego ID sesji chcemy znać	RAX = ID sesji RAX = błąd EPERM, ESRCH
125	Pobierz możliwości procesu (sys_capget)	RDI = adres struktury cap_user_header_t RSI = adres struktury cap_user_data_t	RAX = RDI RAX=NULL, gdy błąd EPERM, EINVAL
126	Ustaw możliwości procesu (sys_capset)	RDI = adres struktury cap_user_header_t RSI = adres struktury	RAX = RDI RAX=NULL, gdy błąd EPERM, EINVAL

		cap_user_data_t	
127	Pobierz sygnały oczekujące (sys_rt_sigpending)	RDI = adres zestawu sygnałów, który otrzyma oczekujące sygnały RSI = rozmiar struktury sigset_t	RAX = 0 RAX=błąd EFAULT
128	Synchronicznie czekaj na zakolejkowane sygnały (sys_rt_sigtimedwait)	RDI = adres zestawu sygnałów, na które czekać RSI = adres struktury siginfo, która otrzyma informację o sygnale RDX = adres struktury timespec określającej czas oczekiwania R10 = rozmiar struktury sigset_t	RAX = numer sygnału RAX=błąd EINVAL, EINTR, EAGAIN, EFAULT
129	Zakolejkuj sygnał dla procesu (sys_rt_sigqueueinfo)	RDI=PID procesu, który ma otrzymać sygnał RSI=numer sygnału RDX=adres struktury siginfo_t do wysłania procesowi razem z sygnałem	RAX = 0 RAX=błąd EFAULT, EPERM
130	Czekaj na sygnał (sys_rt_sigsuspend)	RDI = adres zestawu sygnałów, na które czekać RSI = rozmiar struktury sigset_t	RAX = -1 RAX=błąd EINTR, EFAULT, EINVAL
131	Ustaw alternatywny stos dla procedur obsługi sygnałów (sys_sigaltstack)	RDI = adres struktury stack_t, opisującej nowy stos RSI = adres struktury stack_t, opisującej stary stos; lub NULL (ewentualnie RDX = adres nowego wierzchołka stosu)	RAX = 0 RAX = błąd EPERM, EINVAL, ENOMEM
132	Zmień czas dostępu do pliku (sys_utime)	RDI = adres nazwy pliku (ASCIIZ) RSI = adres struktury utimbuf, NULL gdy chcemy bieżący czas	RAX = 0 RAX = błąd EACCES, ENOENT, EPERM, EROFS
133	Utworzenie spliku specjalnego (sys_mknod)	RDI = adres ścieżki ASCIIZ RSI = typ urządzenia OR prawa dostępu RDX,R10 - wynik działania makra makedev	RAX = 0 RAX = błąd EACCES, EEXIST, EFAULT, EINVAL, ELOOP, ENAMETOOLONG, ENOENT, ENOMEM, ENOSPC, ENOTDIR, EPERM, EROFS

134	Wybierz współdzieloną bibliotekę (sys_uselib)	RDI = adres nazwy biblioteki ASCIIZ	RAX = 0 RAX = błąd EACCES, ENOEXEC
135	Ustal domenę wykonowania procesu (sys_personality)	RDI = numer nowej domeny	RAX = numer starej domeny RAX = błąd
136	Info o zamontowanym systemie plików (sys_ustat)	--zamiast tego, używaj statfs-- RDI = numer główny:poboczny urządzenia / RDI -> 64 bity numeru urządzenia RSI = adres struktury ustat	RAX = 0 RAX = błąd EFAULT, EINVAL, ENOSYS
137	Pobierz statystyki systemu plików (sys_statfs)	RDI = adres nazwy dowolnego pliku w zamontowanym systemie plików RSI = adres struktury statfs	RAX = 0 RAX = błąd
138	Pobierz statystyki systemu plików (sys_fstatfs)	RDI = deskryptor dowolnego otwartego pliku w zamontowanym systemie plików RSI = adres struktury statfs	RAX = 0 RAX = błąd
139	Info o systemie plików (sys_sysfs)	RDI = opcja RSI, RDX - zależne od RDI	RAX zależne od RDI RAX = błąd EINVAL, EFAULT
140	Podaj priorytet szeregowania zadań (sys_getpriority)	RDI = czyj priorytet zmieniamy RSI = identyfiaktor procesu, grupy procesów lub użytkownika, którego priorytet zmieniamy (0=bieżący)	RAX = aktualny priorytet dla wybranego obiektu (od 1 do 40)
141	Ustaw priorytet szeregowania zadań (sys_setpriority)	RDI = czyj priorytet zmieniamy RSI = identyfiaktor procesu, grupy procesów lub użytkownika, którego priorytet zmieniamy (0=bieżący) RDX = nowy priorytet -20...19	RAX = 0 RAX = błąd

142	Ustaw parametry szeregowania zadań (sys_sched_setparam)	RDI = PID procesu RSI = adres struktury sched_param, zawierającej dane	RAX = 0 RAX = błąd EINVAL, ESRCH, EPERM
143	Pobierz parametry szeregowania zadań (sys_sched_getparam)	RDI = PID procesu RSI = adres struktury sched_param, która otrzyma wynik	RAX = 0 RAX = błąd EINVAL, ESRCH, EPERM
144	Ustaw parametry/algorytm szeregowania zadań (sys_sched_setsheduler)	RDI = PID procesu RSI = polityka RSI = adres struktury sched_param, zawierającej dane	RAX = 0 RAX = błąd EINVAL, ESRCH, EPERM
145	Pobierz parametry/algorytm szeregowania zadań (sys_sched_getsheduler)	RDI = PID procesu	RAX = polityka RAX = błąd EINVAL, ESRCH, EPERM
146	Pobierz maksymalny priorytet statyczny (sys_sched_get_priority_max)	RDI = polityka	RAX = maksymalny priorytet dla tej polityki RAX = błąd EINVAL
147	Pobierz minimalny priorytet statyczny (sys_sched_get_priority_min)	RDI = polityka	RAX = minimalny priorytet dla tej polityki RAX = błąd EINVAL
148	Pobierz długość czasu w szeregowaniu cyklicznym (sys_sched_rr_get_interval)	RDI = PID procesu (0 = ten proces) RSI = adres strukury timeval, która otrzyma wynik	RAX = 0 RAX = błąd ESRCH, ENOSYS
149	Zablokowanie stron w pamięci (sys_mlock)	RDI = adres obszaru pamięci (wyrównany do wielokrotności rozmiaru strony pamięci) RSI = długość obszaru pamięci	RAX = 0 RAX = błąd EINVAL, EAGAIN, ENOMEM
150	Odblokowanie stron pamięci (sys_munlock)	RDI = adres obszaru pamięci (wyrównany do wielokrotności rozmiaru strony pamięci) RSI = długość obszaru pamięci	RAX = 0 RAX = błąd EINVAL, ENOMEM

Poprzednia część (Alt+3)
Kolejna część (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Opis funkcji systemowych syscall: 151-200

Jeśli jakaś funkcja zakończy się błędem, w RAX zwracana jest wartość ujemna z przedziału od -4096 do -1 włącznie.
Z drugiej strony, opisy funkcji na stronach manuala mówią, że zwracane jest -1, a wartość błędu jest zapisywana do zmiennej errno z biblioteki GLIBC. Dzieje się tak tylko w przypadku, gdy korzystamy z interfejsu języka C (czyli deklarujemy i uruchamiamy zewnętrzne funkcje odpowiadające wywołaniom systemowym i linkujemy nasz program z biblioteką języka C), a nie bezpośrednio z wywołań systemowych (czyli syscall).

Najaktualniejsze informacje o funkcjach systemowych można znaleźć zazwyczaj w sekcji 2 (lub 3) manuala, np. man 2 open
Najnowsze wersje stron manuala można znaleźć tu: www.kernel.org/pub/linux/docs/man-pages.

Napis ASCIIZ oznacza łańcuch znaków ASCII zakończony znakiem/bajtem Zerowym.
Jeśli potrzeba, przy każdej funkcji jest odnośnik do opisu argumentów i innych dodatkowych informacji: typów danych, wartości błędów, możliwych wartości parametrów itp.

Podstawowe funkcje syscall: 151-200

Numer/ RAX	Opis	Argumenty	Zwraca
151	Zablokowanie całej pamięci procesu (sys_mlockall)	RDI = flagi blokowania pamięci	RAX = 0 RAX = błąd EINVAL, ENOMEM, EAGAIN, EPERM
152	Odblokowanie całej pamięci procesu (sys_munlockall)	nic	RAX = 0 RAX = błąd.
153	Wirtualnie odłącz bieżący terminal (sys_vhangup)	nic	RAX = 0 RAX = błąd EPERM
154	Zmień tablicę LDT (sys_modify_ldt)	RDI = numer funkcji RSI = adres miejsca na przechowanie danych RDX = liczba bajtów obszaru pod [RSI]	RAX = liczba przeczytanych bajtów lub 0 (gdy zapisywano) RAX = błąd EINVAL, ENOSYS, EFAULT
155	Zmień główny system plików/katalog (sys_pivot_root)	RDI = adres łańcucha znaków - nowy główny katalog bieżącego procesu	RAX = 0 RAX = błąd EBUSY,

		RSI = adres łańcucha znaków - otrzyma stary główny katalog bieżącego procesu	EINVAL, EPERM, ENOTDIR + błędy sys_stat
156	Zmień parametry jądra (sys_sysctl)	RDI = adres struktury sysctl_args	RAX = 0 RAX = błąd EPERM, ENOTDIR, EFAULT
157	Dzialania na procesie (sys_prctl)	RDI = opcja RSI, RDX, R10, R8 = argumenty	RAX = 0 lub 1 RAX = błąd EINVAL
158	Ustaw stan wątku zależny od architektury (sys_arch_prctl)	RDI = kod podfunkcji RSI = parametr podfunkcji (wartość do wstawienia lub adres zmiennej w przypadku pobierania)	RAX = 0 RAX = błąd EFAULT, EINVAL, EPERM
159	Dopasowanie zegara w jądrze (sys_adjtimex)	RDI = adres struktury timex	RAX = stan zegara (patrz timex) RAX = błąd EINVAL, EPERM, EFAULT
160	Ustaw limity zasobów (sys_setrlimit)	RDI = numer zasobu RSI = adres struktury rlimit	RAX = 0 RAX = błąd EFAULT, EINVAL, EPERM
161	Zmień katalog główny (sys_chroot)	RDI = adres nazwy/ścieżki nowego katalogu głównego	RAX = 0 RAX = błąd - każdy zależny od systemu plików
162	Zapisz pamięć podręczną na dysku (sys_sync)	nic	RAX zawsze = 0 i nie ma żadnych błędów
163	Włącz/wyłącz zapisywanie kończonych procesów (sys_acct)	RDI = adres nazwy pliku, gdzie ma być zapisywana informacja o kończonych procesach lub NULL, gdy chcemy wyłączyć takie zapisywanie.	RAX = 0 RAX = błąd ENOSYS, ENOMEM, EPERM, EACCES, EIO, EUSERS
164	Ustaw czas (sys_settimeofday)	RDI = adres struktury timeval RSI = adres struktury timezone	RAX = 0 RAX = błąd EFAULT,

			EINVAL, EPERM
165	Montowanie systemu plików (sys_mount)	RDI = adres nazwy urządzenia/pliku specjalnego RSI = adres ścieżki do punktu montowania RDX = adres nazwy systemu plików R10 = flagi montowania R8 = adres dodatkowych danych, niezależne od urządzenia	RAX = 0 RAX = błąd - każdy, który może się zdarzyć w systemie plików lub jądrze
166	Odmontowanie systemu plików 2 (sys_umount2)	RDI = adres nazwy zamontowanego pliku specjalnego/katalogu ASCIIZ RSI = flaga = 1, by siłą odmonotwać, inaczej 0	RAX = 0 RAX = błąd - każdy związany z systemem plików
167	Uruchomienie pliku wymiany (sys_swapon)	RDI = adres ścieżki do pliku/urządzenia swap RSI = flagi wymiany	RAX = 0 RAX = błąd
168	Wyłączenie pliku wymiany (sys_swapoff)	RDI = adres ścieżki i nazwy pliku/urządzenia swap	RAX = 0 RAX = błąd
169	Reboot systemu (sys_reboot)	RDI = pierwsza liczba magiczna = 0FEE1DEADh RSI = druga liczba magiczna = 672274793 lub 85072278 lub 369367448 RDX = flaga R10 = adres dodatkowego argumentu (tylko przy RESTART2)	RAX = 0 RAX = błąd
170	Ustaw nazwę hosta dla systemu (sys_sethostname)	RDI = adres nazwy hosta RSI = długość nazwy	RAX = 0 RAX = błąd EFAULT, EINVAL, EPERM
171	Ustal nazwę domeny (sys_setdomainname)	RDI = adres łańcucha znaków, zawierającego domenę RSI = długość tego łańcucha znaków	RAX = 0 RAX = błąd EINVAL, EPERM, EFAULT
172	Ustaw prawa dostępu do wszystkich portów (sys_iopl)	RDI = poziom IOPL od 0 (normalny proces) do 3	RAX = 0 RAX = błąd

173	Zmień prawa dostępu do portów (sys_ioperm)	RDI = początkowy numer portu RSI = ilość bajtów, które będzie można wysłać/odebrać RDX = końcowy numer portu	RAX = 0 RAX = błąd
174	Utwórz wpis ładowalnego modułu jądra (sys_create_module)	RDI = adres nazwy modułu RSI = długość nazwy	RAX = adres modułu w jądrze RAX = błąd EINVAL, EPERM, EFAULT, EEXIST, ENOMEM
175	Inicjalizacja modułu jądra (sys_init_module)	RDI = adres nazwy modułu RSI = adres struktury module	RAX = 0 RAX = błąd EINVAL, EPERM, EFAULT, ENOENT, EBUSY
176	Usuń wpis nieużywanego modułu jądra (sys_delete_module)	RDI = adres nazwy modułu (0 oznacza usunięcie wpisów wszystkich nieużywanych modułów, które można usunąć automatycznie)	RAX = 0 RAX = błąd EINVAL, EPERM, EFAULT, ENOENT, EBUSY
177	Pobierz symbole eksportowane przez jądro i moduły (sys_get_kernel_syms)	RDI = adres struktury kernel_sym (0 oznacza, że chcemy tylko pobrać liczbę symboli)	RAX = liczba symboli RAX = błąd EINVAL, EPERM, EFAULT, ENOENT, EBUSY
178	Zapytaj o moduł (sys_query_module)	RDI = adres nazwy modułu lub NULL (jądro) RSI = numer podfunkcji RDX = adres bufora R10 = rozmiar bufora R8 = adres DWORDa	RAX = 0 RAX = błąd EFAULT, ENOSPC, EINVAL, ENOENT
179	Zarządzanie limitami dyskowymi (sys_quotactl)	RDI = komenda limitu RSI = adres nazwy pliku urządzenia blokowego, który ma być zarządzany RDX = identyfikator UID lub GID R10 = adres dodatkowej struktury danych (zależy od komendy w RDI)	RAX = 0 RAX = błąd EINVAL, EPERM, EFAULT, ENOENT, EBUSY, ENOTBLK, ESRCH, EUSERS, EACCES
180	Interfejs demona NFS (sys_nfsservctl)	RDI = komenda RSI = adres struktury nfsctl_arg RDX = adres unii union nfsctl_res	RAX = 0 RAX = błąd
181		zarezerwowane dla LiS/STREAMS	zawsze RAX = ENOSYS

	Funkcja systemowa sys_getpmsg		
182	Funkcja systemowa sys_putpmsg	zarezerwowane dla LiS/STREAMS	zawsze RAX = ENOSYS
183	Funkcja systemowa sys_afs_syscall	niezaimplementowane	zawsze RAX = ENOSYS
184	Zarezerwowane (sys_tuxcall)	niezaimplementowane	zawsze RAX = ENOSYS
185	Funkcja systemowa sys_security	niezaimplementowane	zawsze RAX = ENOSYS
186	Pobierz identyfikator wątku (sys_gettid)	nic	RAX = id wątku
187	Czytaj kilka stron pliku z wyprzedzeniem do pamięci podręcznej (sys_readahead)	RDI = deskryptor pliku RSI = miejsce w pliku, od którego zacząć RDX = ilość bajtów do przeczytania	RAX = -EBADF, gdy błąd
188	Ustaw wartość atrybutu rozszerzonego (sys_setxattr)	RDI = adres ścieżki pliku RSI = adres nazwy atrybutu RDX = wartość atrybutu R10 = długość atrybutu R8 = flaga (1=utwórz, 2=zamień)	RAX = 0 RAX = błąd
189	Ustaw wartość atrybutu rozszerzonego (sys_lsetxattr)	RDI = adres ścieżki pliku, funkcja nie podąża za dowiązaniami symbolicznymi RSI = adres nazwy atrybutu RDX = wartość atrybutu R10 = długość atrybutu R8 = flaga (1=utwórz, 2=zamień)	RAX = 0 RAX = błąd
190	Ustaw wartość atrybutu rozszerzonego (sys_fsetxattr)	RDI = deskryptor pliku RSI = adres nazwy atrybutu RDX = wartość atrybutu R10 = długość atrybutu R8 = flaga (1=utwórz, 2=zamień)	RAX = 0 RAX = błąd
191	Pobierz wartość atrybutu rozszerzonego (sys_getxattr)	RDI = adres ścieżki pliku RSI = adres nazwy atrybutu RDX = wartość atrybutu R10 = długość atrybutu	RAX = 0 RAX = błąd
192	Pobierz wartość atrybutu rozszerzonego (sys_lgetxattr)	RDI = adres ścieżki pliku, funkcja nie podąża za dowiązaniami symbolicznymi RSI = adres nazwy atrybutu RDX = wartość atrybutu	RAX = 0 RAX = błąd

		R10 = długość atrybutu	
193	Pobierz wartość atrybutu rozszerzonego (sys_fgetxattr)	RDI = deskryptor pliku RSI = adres nazwy atrybutu RDX = wartość atrybutu R10 = długość atrybutu	RAX = 0 RAX = błąd
194	Pobierz listę nazw atrybutów rozszerzonych pliku (sys_listxattr)	RDI = adres ścieżki pliku RSI = adres tablicy na nazwy RDX = długość tablicy	RAX = 0 RAX = błąd
195	Pobierz listę nazw atrybutów rozszerzonych pliku (sys_llistxattr)	RDI = adres ścieżki pliku, funkcja nie podąża za dowiązaniami symbolicznymi RSI = adres tablicy na nazwy RDX = długość tablicy	RAX = 0 RAX = błąd
196	Pobierz listę nazw atrybutów rozszerzonych pliku (sys_flistxattr)	RDI = deskryptor pliku RSI = adres tablicy na nazwy RDX = długość tablicy	RAX = 0 RAX = błąd
197	Usuń atrybut rozszerzony pliku (sys_removexattr)	RDI = adres ścieżki pliku RSI = adres nazwy atrybutu do usunięcia	RAX = 0 RAX = błąd
198	Usuń atrybut rozszerzony pliku (sys_lremovexattr)	RDI = adres ścieżki pliku, funkcja nie podąża za dowiązaniami symbolicznymi RSI = adres nazwy atrybutu do usunięcia	RAX = 0 RAX = błąd
199	Usuń atrybut rozszerzony pliku (sys_fremovexattr)	RDI = deskryptor pliku RSI = adres nazwy atrybutu do usunięcia	RAX = 0 RAX = błąd
200	Zabij pojedyncze zadanie (sys_tkill)	RDI = PID zadania (niekoniecznie całego procesu) RSI = numer sygnału do wysłania	RAX = 0 RAX = błąd EINVAL, ESRCH, EPERM

Poprzednia część (Alt+3)
Kolejna część (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Opis funkcji systemowych syscall: 201-250

Jeśli jakaś funkcja zakończy się błędem, w RAX zwracana jest wartość ujemna z przedziału od -4096 do -1 włącznie.
Z drugiej strony, opisy funkcji na stronach manuala mówią, że zwracane jest -1, a wartość błędu jest zapisywana do zmiennej errno z biblioteki GLIBC. Dzieje się tak tylko w przypadku, gdy korzystamy z interfejsu języka C (czyli deklarujemy i uruchamiamy zewnętrzne funkcje odpowiadające wywołaniom systemowym i linkujemy nasz program z biblioteką języka C), a nie bezpośrednio z wywołań systemowych (czyli syscall).

Najaktualniejsze informacje o funkcjach systemowych można znaleźć zazwyczaj w sekcji 2 (lub 3) manuala, np. man 2 open
Najnowsze wersje stron manuala można znaleźć tu: www.kernel.org/pub/linux/docs/man-pages.

Napis ASCIIZ oznacza łańcuch znaków ASCII zakończony znakiem/bajtem Zerowym.
Jeśli potrzeba, przy każdej funkcji jest odnośnik do opisu argumentów i innych dodatkowych informacji: typów danych, wartości błędów, możliwych wartości parametrów itp.

Podstawowe funkcje syscall: 201-250

Numer/ RAX	Opis	Argumenty	Zwraca
201	Pobierz czas (sys_time)	RDI = NULL lub adres bufora, który otrzyma kopię wyniku	RAX = ilość sekund od 1 Stycznia 1970 minus 1 RAX = błąd EFAULT
202	Szybka funkcja blokowania (sys_futex)	RDI = sprawdzany adres RSI = operacja RDX = wartość R10 = adres struktury timespec (czas oczekiwania) lub 0	RAX zależy od operacji RAX = błąd EINVAL, EFAULT
203	Ustaw maskę procesorów dla procesu (sys_sched_setaffinity)	RDI = PID procesu, którego maskę ustawiamy (0=bieżący) RSI = długość maski pod [RDX] RDX = adres maski bitowej. Najmłodszy bit maski oznacza, czy proces może być wykonany na pierwszym procesorze logicznym i tak dalej	RAX = 0 RAX = błąd EINVAL, EFAULT, ESRCH, EPERM
204	Pobierz maskę procesorów dla procesu (sys_sched_getaffinity)	RDI = PID procesu, którego maskę pobieramy (0=bieżący) RSI = długość maski pod [RDX] RDX = adres maski bitowej. Najmłodszy bit maski oznacza, czy proces może być wykonany na	RAX = 0 RAX = błąd EINVAL, EFAULT, ESRCH, EPERM

		pierwszym procesorze logicznym i tak dalej	
205	Ustaw wpis w obszarze lokalnym wątku TLS (sys_set_thread_area)	RDI = adres struktury user_desc	RAX = 0 RAX = -EINVAL, -EFAULT, -ESRCH
206	Utwórz asynchroniczny kontekst we/wy (sys_io_setup)	RDI = liczba zradzeń, które kontekst może otrzymać RSI = adres DWORDa (o wartosci zero), który otrzyma uchwyt do nowego kontekstu	RAX = 0 RAX = błąd -EINVAL, -EFAULT, -ENOSYS, -ENOMEM, -EAGAIN
207	Zniszcz asynchroniczny kontekst we/wy (sys_io_destroy)	RDI = uchwyt do usuwanego kontekstu	RAX = 0 RAX = błąd -EINVAL, -EFAULT, -ENOSYS
208	Pobierz zdarzenia we/wy (sys_io_getevents)	RDI = uchwyt do kontekstu RSI = minimalna liczba zdarzeń do pobrania RDX = maksymalna liczba zdarzeń do pobrania R10 = adres tablicy struktur io_event R8 = adres struktury timespec (czas oczekiwania) lub 0	RAX = liczba odczytanych zdarzeń RAX = błąd -EINVAL, -EFAULT, -ENOSYS
209	Wyślij zdarzenia we/wy do przetworzenia (sys_io_submit)	RDI = uchwyt do kontekstu RSI = liczba adresów struktur pod [RDX] RDX = adres tablicy adresów struktur iocb opisujących zdarzenia do przetworzenia	RAX = liczba wysłanych bloków we/wy RAX = błąd -EINVAL, -EFAULT, -ENOSYS, -EBADF, -EAGAIN
210	Przerwij operację we/wy (sys_io_cancel)	RDI = uchwyt do kontekstu RSI = adres struktury iocb, opisującej operację do przerwania RDX = adres struktury io_event, która otrzyma przerwane działanie	RAX = 0 RAX = błąd -EINVAL, -EFAULT, -ENOSYS, -EBADF, -EAGAIN
211	Pobierz wpis w obszarze lokalnym wątku TLS (sys_get_thread_area)	RDI = adres struktury user_desc	RAX = 0 RAX = błąd EINVAL, EFAULT
212	Pobierz ścieżkę wejścia do katalogu (sys_lookup_dcookie)	RDI? = wartość opisująca wpis o katalogu RSI? = adres bufora, który otrzyma	RAX = długość ścieżki RAX = błąd ENAMETOOLONG, EPERM,

		ścieżkę RDX? = długość tego bufora	EINVAL, ENOMEM, ERANGE, EFAULT
213	Utwórz deskryptor pliku epoll (sys_epoll_create)	RDI = ilość deskryptorów do zarezerwowania	RAX = nowy deskryptor pliku RAX = błąd ENOMEM, EINVAL, EMFILE, ENFILE
214	sys_epoll_ctl_old	niezaimplementowane	zawsze RAX = ENOSYS
215	sys_epoll_wait_old	niezaimplementowane	zawsze RAX = ENOSYS
216	Przemapuj strony pamięci / stwórz nieliniowe mapowanie pliku (sys_remap_file_pages)	RDI = początkowy adres stron pamięci RSI = rozmiar przemapowywanego obszaru pamięci RDX = 0 (już nieużywane, musi być 0) R10 = offset w pliku mierzony w jednostkach strony systemowej R8 = flagi (znaczenie taie jak w sys_mmap, ale tu tylko MAP_NONBLOCK jest uznawane)	RAX = 0 RAX = błąd EINVAL
217	Pobierz wpisy o katalogach, wersja 64-bitowa (sys_getdents64)	RDI = deskryptor otwartego katalogu RSI = adres obszaru pamięci na struktury dirent RDX = rozmiar obszaru pamięci pod [RSI]	RAX = 0 RAX = błąd EBADF, EFAULT, EINVAL, ENOENT, ENOTDIR
218	Utwórz wskaźnik do ID wątku (sys_set_tid_address)	RDI = wskaźnik (adres), na którego wartość ma być ustawiona zmienna clear_child_tid jądra	RAX = PID bieżącego procesu
219	sys_restart_syscall	brak danych	brak danych
220	Operacja na semaforze z czasem (sys_semtimedop)	RDI = identyfikator zestawu semaforów RSI = adres tablicy struktur sembuf RDX = liczba elementów w tablicy spod RSI R10 = adres struktury timespec, zawierającej maksymalny czas oczekiwania	RAX = 0 RAX = błąd E2BIG, EACCES, EAGAIN, EFAULT, EFBIG, EIDRM, EINTR, EINVAL, ENOMEM, ERANGE
221	Zadeklaruj wzorce dostępu	RDI = deskryptor pliku	RAX = 0

	(sys_fadvise64_64)	RSI = początek obszaru w pliku (offset) RDX = długość obszaru pliku R10 = wzorzec dostepu	RAX = błąd EBADF, ESPIPE, EINVAL
222	Utwórz POSIX-owy licznik czasu (sys_timer_create)	RDI = ID zegara, który będzie podstawą mierzenia czasu RSI = 0 lub adres struktury sigevent RDX = adres zmiennej trzymającej adres DWORDa, który otrzyma ID nowego zegara	RAX = 0 RAX = błąd EAGAIN, EINVAL, ENOTSUPP
223	Nastaw POSIX-owy licznik czasu (sys_timer_settime)	RDI = ID zegara RSI = flagi (patrz: manual) RDX = adres struktury itimerspec R10 = adres struktury itimerspec	RAX = 0 RAX = błąd EINVAL
224	Pobierz pozostały czas na POSIX-owym liczniku czasu (sys_timer_gettime)	RDI = ID zegara RSI = adres struktury itimerspec, która otrzyma wynik	RAX = 0 RAX = błąd EINVAL
225	Pobierz liczbę przekroczeń POSIX-owego licznika czasu (sys_timer_getoverrun)	RDI = ID zegara	RAX = liczba przekroczeń RAX = błąd EINVAL
226	Usuń POSIX-owy licznik czasu (sys_timer_delete)	RDI = ID zegara	RAX = 0 RAX = błąd EINVAL
227	Ustaw czas na danym zegarze (sys_clock_settime)	RDI = ID zegara RSI = adres struktury timespec zawierającej czas do ustawienia	RAX = 0 RAX = błąd EINVAL, EFAULT, EINVAL
228	Pobierz czas na danym zegarze (sys_clock_gettime)	RDI = ID zegara RSI = adres struktury timespec, która otrzyma czas	RAX = 0 RAX = błąd EINVAL, EFAULT, EINVAL
229	Pobierz precyzję danego zegara (sys_clock_getres)	RDI = ID zegara RSI = adres struktury timespec	RAX = 0 RAX = błąd EINVAL, EFAULT, EINVAL
230	Przerwa w oparciu o dany zegar (sys_clock_nanosleep)	RDI = ID zegara RSI = flagi (TIMER_ABSTIME=1) kontorlujące rodzaj czasu oczekiwania (względny lub nie) RDX = adres struktury timespec, która zawiera czas czekania	RAX = 0 RAX = błąd EINTR, EFAULT, ENOTSUPP

		R10 = adres struktury timespec (lub NULL), która otrzyma pozostały czas	
231	Zakończ wszystkie wątki procesu (sys_exit_group)	RDI = status (kod wyjścia)	nigdy nie powraca
232	Czekaj na deskryptorze pliku epoll (sys_epoll_wait)	RDI = deskryptor epoll RSI = adres tablicy struktur epoll_event RDX = maksymalna liczba zdarzeń, na które będziemy czekać R10 = czas czekania w milisekundach (-1 = nieskończoność)	RAX = liczba deskrptorów gotowych RAX = błąd EFAULT, EINTR, EBADF, EINVAL
233	Kontroluj deskryptor pliku epoll (sys_epoll_ctl)	RDI = deskryptor epoll RSI = kod operacji RDX = deskryptor pliku R10 = adres struktury epoll_event	RAX = 0 RAX = błąd ENOMEM, EBADF, EPERM, EINVAL
234	Wyślij sygnał do pojedynczego procesu (sys_tgkill)	RDI = identyfikator grupy wątków (niekoniecznie całego procesu) RSI = identyfikator wątku, który ma otrzymać sygnał RDX = numer sygnału do wysłania	RAX = 0 RAX = błąd EINVAL, ESRCH, EPERM
235	Zmień czas dostępu do pliku (sys_utimes)	RDI = adres nazwy pliku (ASCIIZ) RSI = adres tablicy 2 struktur timeval, NULL gdy chcemy bieżący czas. Pierwsza struktura opisuje czas dostępu, druga - czas zmiany	RAX = 0 RAX = błąd EACCES, ENOENT, EPERM, EROFS
236	sys_vserver	niezaimplementowane	zawsze RAX = ENOSYS
237	Ustaw politykę dla zakresu pamięci (sys_mbind)	RDI = adres początku obszaru RSI = długość obszaru RDX = polityka R10 = adres DWORDa zawierającego maskę bitową węzłów R8 = długość maski w bitach R9 = flagi polityki	RAX = 0 RAX = błąd EFAULT, EINVAL, ENOMEM, EIO
238	Ustaw politykę pamięci NUMA (sys_set_mempolicy)	RDI = polityka RSI = adres DWORDa zawierającego maskę bitową węzłów RDX = długość maski w bitach	RAX = 0 RAX = błąd
239	Pobierzy politykę pamięci NUMA (sys_get_mempolicy)	RDI = adres DWORDa, który otrzyma politykę	RAX = 0 RAX = błąd

		RSI = NULL lub adres DWORDa, który otrzyma maskę węzłów RDX = maksymalna długość maski w bitach + 1 R10 = sprawdzany adres, jeśli potrzebny R8 = NULL lub MPOL_F_ADDR=2 (wtedy będzie zwrócona polityka dotycząca podanego adersu)	
240	Otwórz kolejkę wiadomości (sys_mq_open)	RDI = adres nazwy, która musi zaczynać się ukośnikiem RSI = flagi dostępu (RDX = tryb) (R10 = adres struktury mq_attr lub NULL)	RAX = deskryptor kolejki RAX = błąd EACCES, EINVAL, EEXIST, EMFILE, ENAMETOOLONG, ENFILE, ENOENT, ENOMEM, ENOSPC
241	Usuń kolejkę wiadomości (sys_mq_unlink)	RDI = adres nazwy, która musi zaczynać się ukośnikiem	RAX = 0 RAX = błąd EACCES, ENAMETOOLONG, ENOENT
242	Wyślij wiadomość do kolejki (sys_mq_timedsend)	RDI = deskryptor kolejki RSI = adres napisu - treści wiadomości RDX = długość wiadomości R10 = priorytet (<=32768, wiadomości o większym numerze są przed tymi o mniejszym) R8 = adres struktury timespec, opisującej czas BEZWZGLĘDNY przedawnienia	RAX = 0 RAX = błąd EAGAIN, EBADF, EINTR, EINVAL, EMSGSIZE, ETIMEOUT
243	Odbierz wiadomość z kolejki (sys_mq_timederceive)	RDI = deskryptor kolejki RSI = adres bufora na treśc wiadomości RDX = długość bufora R10 = NULL lub adres DWORDa, który otrzyma priorytet wiadomości R8 = adres struktury timespec, opisującej czas BEZWZGLĘDNY przedawnienia	RAX = 0 RAX = błąd EAGAIN, EBADF, EINTR, EINVAL, EMSGSIZE, ETIMEOUT, EBADMSG
244	Powiadamianie o wiadomościach (sys_mq_notify)	RDI = deskryptor kolejki RSI = NULL (brak powiadomień) lub adres struktury sigevent	RAX = 0 RAX = błąd EBADF, EBUSY, EINVAL, ENOMEM
245	sys_mq_getsetattr		brak danych

		NIE UŻYWAĆ Interfejs do mq_getattr, mq_setattr	
246	sys_kexec_load	brak danych	brak danych
247	Czekaj na zmianę stanu innego procesu (sys_waitid)	RDI = typ identyfikatora (0=czekaj na dowolnego potomka, 1=czekaj na proces o danym PID, 2=czekaj na członka grupy o danym GID) RSI = identyfikator: PID lub GID (nieważny dla RDI=0) RDX = adres struktury siginfo R10 = opcje opisujące, na jakie zmiany czekamy	RAX = 0, wypełniona struktura siginfo RAX = błąd ECHILD, EINTR, EINVAL
248	sys_add_key	brak danych	brak danych
249	sys_request_key	brak danych	brak danych
250	sys_keyctl	brak danych	brak danych

Poprzednia część (Alt+3)
Kolejna część (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Opis funkcji systemowych syscall: 251-298

Jeśli jakaś funkcja zakończy się błędem, w RAX zwracana jest wartość ujemna z przedziału od -4096 do -1 włącznie.
Z drugiej strony, opisy funkcji na stronach manuala mówią, że zwracane jest -1, a wartość błędu jest zapisywana do zmiennej errno z biblioteki GLIBC. Dzieje się tak tylko w przypadku, gdy korzystamy z interfejsu języka C (czyli deklarujemy i uruchamiamy zewnętrzne funkcje odpowiadające wywołaniom systemowym i linkujemy nasz program z biblioteką języka C), a nie bezpośrednio z wywołań systemowych (czyli syscall).

Najaktualniejsze informacje o funkcjach systemowych można znaleźć zazwyczaj w sekcji 2 (lub 3) manuala, np. man 2 open
Najnowsze wersje stron manuala można znaleźć tu: www.kernel.org/pub/linux/docs/man-pages.

Napis ASCIIZ oznacza łańcuch znaków ASCII zakończony znakiem/bajtem Zerowym.
Jeśli potrzeba, przy każdej funkcji jest odnośnik do opisu argumentów i innych dodatkowych informacji: typów danych, wartości błędów, możliwych wartości parametrów itp.

Podstawowe funkcje syscall: 251-298

Numer/ RAX	Opis	Argumenty	Zwraca
251	Ustaw priorytet kolejkowania We/wy procesu (sys_ioprio_set)	RDI = typ RSI = informacja zależna od typu RDX = nowy priorytet	RAX = 0 RAX = błąd ESRCH, EPERM, EINVAL
252	Pobierz priorytet kolejkowania We/wy procesu (sys_ioprio_get)	RDI = typ RSI = informacja zależna od typu	RAX = priorytet RAX = błąd ESRCH, EPERM, EINVAL
253	Inicjalizacja kolejki zdarzeń inotify (sys_inotify_init)	nic	RAX = deskryptor kolejki RAX = błąd EMFILE, ENFILE, ENOMEM
254	Dodaj obiekt obserwowany kolejki zdarzeń inotify (sys_inotify_add_watch)	RDI = deskryptor kolejki inotify RSI = adres nazwy pliku ASCIIZ RDX = flagi inotify	RAX = deskryptor obserwacji danego pliku RAX = błąd EACCES, EBADF, EFAULT, EINVAL, ENOMEM, ENOSPC

255	Usuń obserwację obiektu z kolejki zdarzeń inotify (sys_inotify_rm_watch)	RDI = deskryptor kolejki inotify RSI = deskryptor obserwacji	RAX = 0 RAX = błąd EBADF, EINVAL
256	sys_migrate_pages	brak danych	brak danych
257	Otwórz plik względnie do katalogu (sys_openat)	RDI = deskryptor otwartego katalogu RSI = adres nazwy pliku ASCIIZ. Jeśli ścieżka jest względna, jest brana jako względna względem podanego katalogu zamiast bieżącego katalogu procesu RDX = bity dostępu R10 = prawa dostępu / tryb	RAX = 0 RAX = błąd EBADF, ENOTDIR
258	Utwórz katalog względnie do katalogu (sys_mkdirat)	RDI = deskryptor otwartego katalogu RSI = adres ścieżki/nazwy ASCIIZ. Jeśli ścieżka jest względna, jest brana jako względna względem podanego katalogu zamiast bieżącego katalogu procesu RDX = prawa dostępu / tryb	RAX = 0 RAX = błąd - każdy związany z systemem plików lub prawami dostępu
259	Utworzenie pliku specjalnego względnie do katalogu (sys_mknodat)	RDI = deskryptor otwartego katalogu RSI = adres ścieżki/nazwy ASCIIZ. Jeśli ścieżka jest względna, jest brana jako względna względem podanego katalogu zamiast bieżącego katalogu procesu RDX = typ urządzenia OR prawa dostępu R10, R8 - wynik działania (zmodyfkiowanego) makra makedev	RAX = 0 RAX = błąd EACCES, EEXIST, EFAULT, EINVAL, ELOOP, ENAMETOOLONG, ENOENT, ENOMEM, ENOSPC, ENOTDIR, EPERM, EROFS
260	Zmiana właściciela obiektu położonego względnie do katalogu (sys_fchownat)	RDI = deskryptor otwartego katalogu RSI = adres ścieżki/nazwy ASCIIZ. Jeśli ścieżka jest względna, jest brana jako względna względem podanego katalogu zamiast bieżącego katalogu procesu RDX = identyfikator UID nowego właściciela obiektu R10 = identyfikator GID grupy, która stanie się właścicielem obiektu R8 = 0 lub wartość AT_SYMLINK_NOFOLLOW=100h, wtedy nie będzie podążał za dowiązaniami symbolicznymi	RAX = 0 RAX = błąd EBADF, ENOTDIR, EINVAL
261	Zmiana czasów dostępu i zmian pliku położonego względnie do katalogu (sys_futimesat)	RDI = deskryptor otwartego katalogu RSI = adres ścieżki/nazwy ASCIIZ. Jeśli ścieżka jest względna, jest brana jako względna względem podanego katalogu	RAX = 0 RAX = błąd EBADF, ENOTDIR

		zamiast bieżącego katalogu procesu RDX = adres tablicy 2 struktur timeval, NULL gdy chcemy bieżący czas. Pierwsza struktura opisuje czas dostępu, druga - czas zmiany	
262	Pobierz status obiektu położonego względnie do katalogu (sys_fstatat64)	RDI = deskryptor otwartego katalogu RSI = adres ścieżki/nazwy ASCIIZ. Jeśli ścieżka jest względna, jest brana jako względna względem podanego katalogu zamiast bieżącego katalogu procesu RDX = adres struktury stat na dane R10 = 0 lub AT_SYMLINK_NOFOLLOW (=0x100), gdy nie chcemy dereferencjonować dowiązań	RAX = 0 RAX = błąd EBADF, ENOTDIR, EINVAL, błędy sys_stat
263	Usuń obiekt względnie do katalogu (sys_ulinkat)	RDI = deskryptor otwartego katalogu RSI = adres ścieżki/nazwy ASCIIZ. Jeśli ścieżka jest względna, jest brana jako względna względem podanego katalogu zamiast bieżącego katalogu procesu RDX = 0 lub AT_REMOVEDIR (=0x200), gdy chcemy usuwać katalogi	RAX = 0 RAX = błąd EBADF, ENOTDIR, EINVAL
264	Przenieś plik/Zmień nazwę pliku względnie do katalogu (sys_renameat)	RDI = deskryptor otwartego katalogu źródłowego RSI = adres starej nazwy (i ewentualnie ścieżki) ASCIIZ RDX = deskryptor otwartego katalogu docelowego R10 = adres nowej nazwy (i ewentualnie ścieżki) ASCIIZ	RAX = 0 RAX = błąd EBUSY, EEXIST, EISDIR, ENOTEMPTY, EXDEV (i inne błędy systemu plików), EBADF, ENOTDIR
265	Utworzenie twardego dowiązania do pliku względnie do katalogu (sys_linkat)	RDI = deskryptor otwartego katalogu źródłowego RSI = adres nazwy istniejącego pliku ASCIIZ RDX = deskryptor otwartego katalogu docelowego R10 = adres nazwy nowego pliku ASCIIZ R8 = 0 (flagi)	RAX = 0 RAX=błąd EACCES, EIO, EPERM, EEXIST, EFAULT, ELOOP, EMLINK, ENAMETOOLONG, ENOENT, ENOMEM, ENOSPC, ENOTDIR, EROFS, EXDEV
266	Stwórz dowiązanie symboliczne do pliku względnie do katalogu (sys_symlinkat)	RDI = adres nazwy pliku ASCIIZ RSI = deskryptor otwartego katalogu docelowego RDX = adres nazwy linku ASCIIZ	RAX = 0 RAX = błędy związane z uprawnieniami lub systemem plików
267	Przeczytaj zawartość linku symbolicznego względnie	RDI = deskryptor otwartego katalogu źródłowego	RAX = 0 RAX = błąd EBADF,

	do katalogu (sys_readlinkat)	RSI = adres nazwy dowiązania symbolicznego ASCIIZ RDX = adres bufora, który otrzyma przeczytaną informację R10 = długość bufora	ENOTDIR
268	Zmiana uprawnień obiektu względnie do katalogu (sys_fchmodat)	RDI = deskryptor otwartego katalogu źródłowego RSI = adres nazwy pliku ASCIIZ RDX = nowe prawa dostępu R10 = flagi, musi być zero	RAX = 0 RAX = błąd EBADF, ENOTDIR
269	Sprawdź uprawnienia dostępu do pliku (sys_faccessat)	RDI = deskryptor otwartego katalogu źródłowego RSI = adres nazwy pliku ASCIIZ RDX = prawa dostępu / tryb (wartości R_OK, W_OK, X_OK) R10 = flagi: 0 lub AT_SYMLINK_NOFOLLOW=100h (nie podążaj za dowiązaniami symbolicznymi) lub AT_EACCESS=0x200 (sparwdzanie według efektywnych UID i GID)	RAX = 0 RAX = błąd EBADF, ENOTDIR, EINVAL
270	sys_pselect6	brak danych	brak danych
271	Czekaj na zdarzenia na deskryptorze (sys_ppoll)	RDI = adres tablicy struktur pollfd RSI = liczba struktur pollfd w tablicy RDX = adres struktury timespec - czas oczekiwania lub 0 (nieskończoność) R10 = adres struktury sigset_t	RAX = liczba odpowiednich deskryptorów RAX = 0, gdy czas upłynął RAX = błąd EFAULT, EINTR, EINVAL, EBADF, ENOMEM
272	Cofnij współdzielanie (sys_unshare)	RDI = CLONE_FILES, CLONE_FS lub CLONE_NEWNS spośród flag	RAX = 0 RAX=błąd EPERM, ENOMEM, EINVAL
273	sys_set_robust_list	brak danych	brak danych
274	sys_get_robust_list	brak danych	brak danych
275	Spleć dane z/do potoku (sys_splice)	RDI = wejściowy deksryptor pliku RSI = offset w pliku wejściowym, skąd czytać dane (0 dla potoków) RDX = wyjściowy deksryptor pliku	RAX = liczba przeniesionych bajtów RAX=błąd EBADF, ESPIPE, ENOMEM,

		R10 = offset w pliku wyjściowym, dokąd zapisać dane (0 dla potoków) R8 = maksymalna liczba bajtów do przeniesienia R9 = flagi	EINVAL
276	Duplikowanie zawartości potoku (sys_tee)	RDI = wejściowy deksryptor pliku RSI = wyjściowy deksryptor pliku RDX = maksymalna liczba bajtów do przeniesienia R10 = flagi (te same, co dla sys_splice)	RAX = liczba zduplikowanych bajtów RAX=błąd ENOMEM, EINVAL
277	Synchronizuj segment pliku z dyskiem (sys_sync_file_range)	RDI = deskryptor pliku RSI = początek zakresu danych do synchronizacji / RSI -> 64 bity adresu początku danych? RDX = liczba bajtów do zsynchronizowania / RDX -> 64-bitowa liczba bajtów do zsynchronizowania? R10 = flagi synchronizacji	RAX = 0 RAX=błąd EBADF, EIO, EINVAL, ENOMEM, ENOSPC, ESPIPE
278	Spleć strony pamięci do potoku (sys_vmsplice)	RDI = wejściowy deskryptor potoku RSI = adres tablicy struktur iovec RDX = liczba elementów w tablicy pod [RSI] R10 = flagi (te same, co dla sys_splice)	RAX = liczba bajtów przeniesionych do potoku RAX=błąd EBADF, ENOMEM, EINVAL
279	Przesuń strony pamięci procesu (sys_move_pages)	RDI = PID procesu RSI = liczba stron do przeniesienia RDX = adres tablicy adresów stron R10 = adres tablicy liczb całkowitych określających żądane miejsce dla danej strony R8 = adres tablicy na liczby całkowite, które otrzymają status wykonanej operacji R9 = flagi określające typ stron do przeniesienia (MPOL_MF_MOVE=2 oznacza tylko strony procesu, MPOL_MF_MOVE_ALL=4 oznacza wszystkie strony)	RAX = 0 RAX=błąd E2BIG, EACCES, EFAULT, EINVAL, ENODEV, ENOENT, EPERM, ESRCH
280	Zmień znaczniki czasu pliku (sys_utimensat)	RDI = deskryptor otwartego katalogu RSI = adres nazwy pliku RDX = adres dwuelementowej tablicy struktur timespec, opisujących czas dostępu i zmiany R10 = flagi: 0 lub AT_SYMLINK_NOFOLLOW=0x100 (nie podążaj za dowiązaniami symbolicznymi)	RAX = 0 RAX=błąd EACCES, EBADF, EFAULT, EINVAL, ELOOP, ENAMETOOLONG, ENOENT, ENOTDIR, EPERM, EROFS, ESRCH
281	Czekaj na deskryptorze pliku epoll	RDI = deskryptor epoll RSI = adres tablicy struktur epoll_event	RAX = liczba deskrptorów gotowych

	(sys_epoll_pwait)	RDX = maksymalna liczba zdarzeń, na które będziemy czekać R10 = czas czekania w milisekundach (-1 = nieskończoność) R8 = adres zestawu sygnałów (tablicy 32 DWORDów), które też przerwą czekanie	RAX = błąd EFAULT, EINTR, EBADF, EINVAL
282	Stwórz deskryptor pliku do otrzymywania sygnałów (sys_signalfd)	RDI = deskryptor: -1, gdy tworzymy nowy lub nieujemny, gdy zmieniamy istniejący RSI = adres maski sygnałów (sigset), które chcemy otrzymywać	RAX = deskryptor pliku RAX=błąd EBADF, EINVAL, EMFILE, ENFILE, ENODEV, ENOMEM
283	Stwórz nowy czasomierz (sys_timerfd_create)	RDI = identyfikator zegara (CLOCK_REALTIME=0 lub CLOCK_MONOTONIC=1) RSI = flagi: 0 lub zORowane wartości TFD_NONBLOCK (=04000 ósemkowo, operacje mają być nieblokujące), TFD_CLOEXEC (=02000000 ósemkowo, zamknij w czasie wywołania exec)	RAX = deskryptor pliku RAX=błąd EINVAL, EMFILE, ENFILE, ENODEV, ENOMEM
284	Stwórz deskryptor pliku do otrzymywania zdarzeń (sys_eventfd)	RDI = wstępna wartość licznika zdarzeń	RAX = deskryptor pliku RAX=błąd EINVAL, EMFILE, ENFILE, ENODEV, ENOMEM
285	Manipulacja przestrzenią pliku (sys_fallocate)	RDI = deskryptor pliku RSI = tryb = FALLOC_FL_KEEP_SIZE (alokuje i zeruje przestrzeń na dysku w podanym zakresie bajtów) = 1 RDX = offset początku zakresu bajtów w pliku R10 = ilość bajtów w zakresie	RAX = liczba deskrptorów gotowych RAX = błąd EFAULT, EINTR, EBADF, EINVAL
286	Uruchom lub zatrzymaj czasomierz (sys_timerfd_settime)	RDI = deskryptor pliku czasomierza RSI = flagi (0 uruchamia czasomierz względy, TFD_TIMER_ABSTIME=1 - bezwzględny) RDX = adres struktury itimerspec, zawierającej początkowy czas R10 = adres struktury itimerspec, która otrzyma aktualny czas	RAX = 0 RAX=błąd EINVAL, EFAULT, EBADF
287	Pobierz czas na czasomierzu (sys_timerfd_gettime)	RDI = deskryptor pliku czasomierza RSI = adres struktury itimerspec, która otrzyma aktualny czas	RAX = 0 RAX=błąd EINVAL, EFAULT, EBADF
288	sys_paccept	brak danych	brak danych

289	Stwórz deskryptor pliku do otrzymywania sygnałów (sys_signalfd4)	RDI = deskryptor: -1, gdy tworzymy nowy lub nieujemny, gdy zmieniamy istniejący RSI = adres maski sygnałów (sigset), które chcemy otrzymywać RDX = flagi: 0 lub zORowane wartości SFD_NONBLOCK (=04000 ósemkowo, operacje mają być nieblokujące), SFD_CLOEXEC (=02000000 ósemkowo, zamknij w czasie wywołania exec)	RAX = deskryptor pliku RAX=błąd EBADF, EINVAL, EMFILE, ENFILE, ENODEV, ENOMEM
290	Stwórz deskryptor pliku do otrzymywania zdarzeń (sys_eventfd2)	RDI = wstępna wartość licznika zdarzeń RSI = flagi: 0 lub zORowane wartości EFD_NONBLOCK (=04000 ósemkowo, operacje mają być nieblokujące), EFD_CLOEXEC (=02000000 ósemkowo, zamknij w czasie wywołania exec)	RAX = deskryptor pliku RAX=błąd EINVAL, EMFILE, ENFILE, ENODEV, ENOMEM
291	Utwórz deskryptor pliku epoll (sys_epoll_create1)	RDI = flagi: 0 lub EPOLL_CLOEXEC (=02000000 ósemkowo, zamknij w czasie wywołania exec)	RAX = nowy deskryptor pliku RAX = błąd ENOMEM, EINVAL, EMFILE, ENFILE
292	Zamień deskryptor zduplikowanym deskryptorem pliku (sys_dup3)	RDI = deskryptor do zduplikowania RSI = deskryptor, do którego powinien być przyznany duplikat RDX = flagi: 0 lub O_CLOEXEC (=02000000 ósemkowo, zamknij w czasie wywołania exec)	RAX = zduplikowany deskryptor RAX = błąd EBADF, EMFILE, EBUSY, EINTR, EINVAL
293	Utwórz potok (sys_pipe2)	RDI = adres tablicy dwóch DWORDów RSI = flagi: 0 lub zORowane wartości O_NONBLOCK (=04000 ósemkowo, operacje mają być nieblokujące), O_CLOEXEC (=02000000 ósemkowo, zamknij w czasie wywołania exec)	RAX = 0 i pod [RDI]: deskryptor odczytu z potoku fd(0) pod [RDI], deskryptor zapisu do potoku fd(1) pod [RDI+4] RAX = błąd EFAULT, EMFILE, ENFILE, EINVAL
294	Inicjalizacja kolejki zdarzeń inotify (sys_inotify_init1)	RDI = flagi: 0 lub zORowane wartości IN_NONBLOCK (=04000 ósemkowo, operacje mają być nieblokujące), IN_CLOEXEC (=02000000 ósemkowo, zamknij w czasie wywołania exec)	RAX = deskryptor kolejki RAX = błąd EMFILE, ENFILE, ENOMEM, EINVAL
295	sys_preadv	brak danych	brak danych

296	sys_pwritev	brak danych	brak danych
297	sys_rt_tgsigqueueinfo	brak danych	brak danych
298	sys_perf_counter_open	brak danych	brak danych

Poprzednia część (Alt+3)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Jeśli jakaś funkcja zakończy się błędem, w RAX zwracana jest wartość ujemna z przedziału od -4096 do -1 włącznie.
Z drugiej strony, opisy funkcji na stronach manuala mówią, że zwracane jest -1, a wartość błędu jest zapisywana do zmiennej errno z biblioteki GLIBC. Dzieje się tak tylko w przypadku, gdy korzystamy z interfejsu języka C (czyli deklarujemy i uruchamiamy zewnętrzne funkcje odpowiadające wywołaniom systemowym i linkujemy nasz program z biblioteką języka C), a nie bezpośrednio z wywołań systemowych (czyli syscall).

Najaktualniejsze informacje o funkcjach systemowych można znaleźć zazwyczaj w sekcji 2 (lub 3) manuala, np. man 2 open
Najnowsze wersje stron manuala można znaleźć tu: www.kernel.org/pub/linux/docs/man-pages.

Napis ASCIIZ oznacza łańcuch znaków ASCII zakończony znakiem/bajtem Zerowym.
Jeśli potrzeba, przy każdej funkcji jest odnośnik do opisu argumentów i innych dodatkowych informacji: typów danych, wartości błędów, możliwych wartości parametrów itp.

Podstawowe funkcje syscall: 301-324

Numer/ RAX	Opis	Argumenty	Zwraca
-	sys_getcpu	brak danych	brak danych
-	Czekaj na zmianę stanu innego procesu (sys_waitpid)	RDI = id procesu / specyfikacja RSI = NULL lub adres zmiennej DWORD, która otrzyma status RDX = opcje	RAX=PID zakończonego procesu [RSI] = (jeśli podano adres bufora) stan wyjścia procesu RAX = błąd ECHILD, EINVAL, ERESTARTSYS
-	Funkcja systemowa sys_break (porzucone)	Istnieje tylko dla zachowania zgodności	RAX = błąd ENOSYS
-	Funkcja systemowa sys_oldstat (porzucone)		
-	Odmontowanie systemu plików (sys_umount)	RDI = adres nazwy pliku specjalnego lub katalogu (zamontowanego)	nic RAX = błąd - każdy, który może się zdarzyć w systemie plików lub jądrze
-	Ustaw czas systemowy (sys_stime)	RDI = nowy czas jako liczba sekund, które upłynęły od 1 Stycznia 1970	nic RAX = błąd EPERM
-			

	Funkcja systemowa sys_oldfstat (porzucone)		
-	Funkcja systemowa sys_stty (porzucone)	--nieużywane od 2.0--	zawsze RAX = -1
-	Funkcja systemowa sys_gtty (porzucone)	--nieużywane od 2.0--	zawsze RAX = -1
-	Zmień priorytet procesu (sys_nice)	RDI = liczba, o którą zwiększyć numer priorytetu (czyli zmniejszyć sam priorytet)	nic RAX = błąd EPERM
-	Pobierz bieżącą datę i czas - sys_ftime (przestarzałe)	--zamiast tego, używaj time, gettimeofday-- RDI = adres struktury timeb	zawsze RAX = 0
-	Funkcja systemowa sys_prof (porzucone)	niezaimplementowane w jądrach 2.4	zawsze RAX = -1, błąd ENOSYS
-	Ustaw procedurę obsługi sygnału (sys_signal)	RDI = numer sygnału RSI = adres procedury przyjmującej `int` i zwracającą `void` (nic) lub wartość SIG_IGN=1 (ignoruj sygnał) lub SIG_DFL=0 (resetuj sygnał na domyślne zachowanie)	RAX = adres poprzedniej procedury obsługi RAX = błąd SIG_ERR
-	Funkcja systemowa sys_lock (porzucone)	--nieużywane od 2.0--	zawsze RAX = -1
-	Funkcja systemowa sys_mpx (porzucone)	--nieużywane od 2.0--	zawsze RAX = -1
-	Pobierz/ustaw limity zasobów (sys_ulimit)	--nieużywane (zamiast tego używaj getrlimit, setrlimit, sysconf)-- man 3 ulimit RDI = komenda, patrz: sys_ulimit RSI = nowylimit	RAX = aktualny limit RAX = -1, gdy błąd
-	Funkcja systemowa sys_oldolduname (porzucone)		
-	Pobierz/ustal procedurę obsługi sygnału (sys_sigaction)	RDI = numer sygnału RSI = adres struktury sigaction opisującą bieżącą procedurę RDX = adres struktury sigaction opisującą starą procedurę	nic RAX=błąd EINVAL, EINTR, EFAULT
-	Pobierz maskę synałów procesu (sys_sgetmask)	--przestarzałe (zamiast tego używaj sys_sigprocmask)--	RAX = maska sygnałów bieżącego procesu

-	Ustaw maskę synałów procesu (sys_ssetmask)	--przestarzałe (zamiast tego używaj sys_sigprocmask)-- RDI = nowa maska sygnałów procesu	RAX = poprzednia maska sygnałów
-	Zastąpienie dla sigpause - sys_sigsuspend	RDI = adres nowej maski sygnałowej procesu - struktury sigset_t	RAX = 0 RAX = -1, gdy błąd
-	Pobierz trwające blokujące sygnały (sys_sigpending)	RDI = adres maski sygnałów - struktury sigset_t	RAX = 0 RAX = -1, gdy błąd
-	Czytaj katalog (sys_readdir)	RDI = deskryptor otwartego katalogu RSI = adres struktury dirent RDX = liczba struktur do odczytania (ignorowane, czytana jest 1 struktura)	RAX = 1 RAX = 0 na końcu katalogu RAX = -1, gdy błąd
-	Profilowanie czasu wykonywania (sys_profil)	--man 3 profil-- RDI = adres tablicy WORDów RSI = długość tej tablicy, na którą pokazuje RDI RDX = offset początkowy R10 = mnożnik	zawsze RAX = 0
-	Funkcja systemowa sys_olduname (porzucone)		
-	Spowoduj bezczynność procesu 0 (sys_idle)	nic	dla procesu nr 0 nigdy nie wraca. Dla pozostałych zwraca RAX = -1 (EPERM)
-	Przejdź w tryb wirtualny 8086 (sys_vm86old)	--to było przed jądrem 2.0.38-- RDI = adres struktury vm86_struct	RAX = 0 RAX = -1, gdy błąd
-	Komunikacja międzyprocesowa SysV (sys_ipc)	RDI = numer wywoływanej funkcji RSI, RDX, R10 = parametry 1-3 wywoływanej funkcji R8 = adres dalszych parametrów, jeśli trzeba R9 = parametr piąty	zależy od wywoływanej funkcji
-	Powrót z procedury obsługi sygnału (sys_sigreturn)	RDI = argument zależny od architektury, używany przez jądro	nigdy nie powraca
-			

	Zmiana listy blokowanych sygnałów (sys_sigprocmask)	RDI = co zrobić RSI = adres struktury sigset_t RDX = adres struktury sigset_t (do przechowania starej maski) lub 0	RAX = 0 RAX = -1, gdy błąd EINVAL, EPERM, EFAULT
-	Demon wypróżniania buforów (sys_bdflush)	RDI = komenda demona RSI = dodatkowy parametr, zależny od komendy	RAX=0, gdy sukces i RDI>0 RAX = -1, gdy błąd EPERM, EFAULT, EBUSY, EINVAL
-	Zmiana bieżącej pozycji w dużym pliku (sys_llseek)	RDI = deskryptor otwartego pliku ECX:RSI = liczba bajtów, o którą chcemy się przesunąć RDX = adres QWORDa, który otrzyma nową pozycję w pliku (big endian?) R10 = odkąd zaczynamy ruch	RAX = 0 RAX = błąd EBADF, EINVAL
-	Oczekiwanie zmiany stanu deskryptoru(ów) (sys_newselect)	RDI = najwyższy numer spośród deskryptorów + 1 (co najwyżej FILE_MAX) RSI = adres tablicy deskryptorów (lub 0) sprawdzanych, czy można z nich czytać RDX = adres tablicy deskryptorów (lub 0) sprawdzanych, czy można do nich pisać R10 = adres tablicy deskryptorów (lub 0) sprawdzanych, czy nie wystąpił u nich wyjątek R8 = adres struktury timeval zawierającą maksymalny czas oczekiwania	RAX = całkowita liczba deskryptorów, ktora pozostała w tablicach RAX = 0, gdy skończył się czas RAX = -1, gdy wystąpił błąd EBADF, EINVAL, ENOMEM, EINTR
-	Uruchom tryb wirtualny 8086 (sys_vm86)	RDI = kod funkcji RSI = adres struktury vm86plus_struct	(zalezy od numeru funkcji) RAX = -1, gdy błąd EFAULT
-	Pobierz limity zasobów (sys_ugetrlimit)	patrz: sys_getrlimit (?)	nic RAX = błąd EFAULT, EINVAL, EPERM
-	Mapuj urządzenie lub plik do pamięci (sys_mmap2)	RDI = proponowany adres początkowy RSI = ilość bajtów pliku do zmapowania RDX = ochrona R10 = flagi mapowania R8 = deskryptor mapowanego pliku,	RAX = adres zmapowanego obszaru RAX = -1, gdy błąd (takie same jak w sys_mmap + EFAULT)

		jeśli mapowanie nie jest anonimowe R9 = offset początku mapowanych danych w pliku, liczony w jednostkach wielkości strony systemowej zamiast w bajtach	
-	Skróć plik, wersja 64-bitowa (sys_truncate64)	RDI = adres nazwy pliku ASCIIZ RSI = ilość bajtów, do której ma zostać skrócony plik (niższy DWORD) RDX = ilość bajtów, do której ma zostać skrócony plik (wyższy DWORD)	RAX = 0 RAX = -1, gdy błąd
-	Skróć plik, wersja 64-bitowa (sys_ftruncate64)	RDI = deskryptor pliku otwartego do zapisu RSI = ilość bajtów, do której ma zostać skrócony plik (niższy DWORD) RDX = ilość bajtów, do której ma zostać skrócony plik (wyższy DWORD)	RAX = 0 RAX = -1, gdy błąd
-	Pobierz status pliku, wersja 64-bitowa (sys_stat64)	RDI = adres nazwy pliku ASCIIZ. Jeśli plik jest linkiem, to zwracany jest status obiektu docelowego. RSI = adres struktury stat64	RAX = 0 RAX = -1, gdy błąd
-	Pobierz status pliku, wersja 64-bitowa (sys_lstat64)	RDI = adres nazwy pliku ASCIIZ. Jeśli plik jest linkiem, to zwracany jest status linku, a nie obiektu docelowego. RSI = adres struktury stat64	RAX = 0 RAX = -1, gdy błąd
-	Pobierz status pliku, wersja 64-bitowa (sys_fstat64)	RDI = deskryptor otwartego pliku RSI = adres struktury stat64	RAX = 0 RAX = -1, gdy błąd
-	Zmiana właściciela (sys_lchown32)	RDI = adres nazwy pliku/katalogu ASCIIZ RSI = nowy numer użytkownika RDX = nowy numer grupy	nic RAX = błąd EPERM, EROFS, EFAULT, ENAMETOOLONG, ENOENT, ENOMEM, ENOTDIR, EACCES, ELOOP i inne
-	Pobierz identyfikator użytkownika (sys_getuid32)	nic	RAX = numer UID
-	Pobierz ID grupy bieżącego procesu (sys_getgid32)	nic	RAX = ID grupy
-		nic	RAX = efektywny UID

	Pobierz efektywne ID użytkownika bieżącego procesu (sys_geteuid32)		
-	Pobierz efektywne ID grupy bieżącego procesu (sys_getegid32)	nic	RAX = efektywny GID
-	Ustaw realny i efektywny ID użytkownika (sys_setreuid32)	RDI = realny ID użytkownika (UID) RSI = efektywny UID	nic RAX = błąd EPERM
-	Ustaw realny i efektywny ID grupy (sys_setregid32)	RDI = realny ID grupy (GID) RSI = efektywny GID	nic RAX = błąd EPERM
-	Pobierz liczbę dodatkowych grup (sys_getgroups32)	RDI = rozmiar tablicy z RSI RSI = adres tablicy, gdzie zostaną zapisane GID-y (DWORDY) grup dodatkowych	RAX = liczba dodatkowych grup procesu RAX = -1 oznacza błąd (EFAULT, EINVAL, EPERM)
-	Ustaw liczbę dodatkowych grup (sys_setgroups32)	RDI = rozmiar tablicy z RSI RSI = adres tablicy, gdzie zawierającą GID-y (DWORDY)	RAX = 0 RAX = -1 oznacza błąd (EFAULT, EINVAL, EPERM)
-	Zmiana właściciela (sys_fchown32)	RDI = deskryptor otwartego pliku RSI = nowy numer użytkownika RDX = nowy numer grupy	RAX = 0 RAX = -1, gdy błąd
-	Ustaw różne ID użytkownika (sys_setresuid32)	RDI = realny UID lub -1 (wtedy jest bez zmian) RSI = efektywny UID lub -1 (bez zmian) RDX = zachowany (saved) UID lub -1 (bez zmian)	RAX = 0 RAX = -1, gdy błąd EPERM
-	Pobierz różne ID użytkownika (sys_getresuid32)	RDI = adres DWORDa, który otrzyma realny UID RSI = adres DWORDa, który otrzyma efektywny UID RDX = adres DWORDa, który otrzyma zachowany UID	RAX = 0 RAX = -1, gdy błąd EFAULT
-	Ustaw realny, efektywny i zachowany ID grupy (sys_setresgid32)	RDI = realny GID RSI = efektywny GID RDX = zachowany (saved) GID	RAX = 0 RAX = -1, gdy błąd EPERM
-	Pobierz realny, efektywny i zachowany ID grupy	RDI = adres DWORDa, który otrzyma realny GID	RAX = 0 RAX = -1, gdy błąd EFAULT

	(sys_getresgid32)	RSI = adres DWORDa, który otrzyma efektywny GID RDX = adres DWORDa, który otrzyma zachowany (saved) GID	
-	Zmiana właściciela pliku (sys_chown32)	RDI=adres ścieżki do pliku RSI = UID nowego właściciela RDX = GID nowej grupy	RAX = 0 RAX = -1, gdy błąd np. EPERM, EROFS, EFAULT, ENOENT, ENAMETOOLONG, ENOMEM, ENOTDIR, EACCES, ELOOP
-	Ustaw identyfikator użytkownika (sys_setuid32)	RDI = nowy UID	nic RAX = błąd EPERM
-	Ustaw ID grupy bieżącego procesu (sys_setgid32)	RDI = nowy ID grupy	nic RAX = błąd EPERM
-	Ustal UID przy sprawdzaniu systemów plików (sys_setfsuid32)	RDI = nowy ID użytkownika	RAX = stary UID (zawsze)
-	Ustal GID przy sprawdzaniu systemów plików (sys_setfsgid32)	RDI = nowy ID grupy	RAX = stary GID (zawsze)
-	Kontrola nad deskryptorem pliku, wersja 64-bitowa (sys_fcntl64)	RDI = deskryptor pliku RSI = kod komendy RDX zależy od komendy	RAX zależy od komendy RAX = błąd EACCES, EAGAIN, EBADF, EDEADLK, EFAULT, EINTR, EINVAL, EMFILE, ENOLOCK, EPERM
-	brak danych	-	-
-	Funkcja systemowa sys_alloc_hugepages	zaimplementowane tylko w jądrach 2.5.36 - 2.5.54, więc nie będę omawiał	zawsze RAX = -1, błąd ENOSYS
-	Funkcja systemowa sys_free_hugepages	zaimplementowane tylko w jądrach 2.5.36 - 2.5.54, więc nie będę omawiał	zawsze RAX = -1, błąd ENOSYS
-	Pobierz statystyki systemu plików, wersja 64-bitowa (sys_statfs64)	RDI = adres nazwy dowolnego pliku w zamontowanym systemie plików RSI adres struktury statfs64	RAX = 0 RAX = -1, gdy błąd

-	Pobierz statystyki systemu plików, wersja 64-bitowa (sys_fstatfs64)	RDI = deskryptor dowolnego otwartego pliku w zamontowanym systemie plików RSI = adres struktury statfs64	RAX = 0 RAX = -1, gdy błąd
-	sys_setaltroot	nieużywane	brak danych
-	Funkcja systemowa sys_oldlstat (porzucone)		

Poprzednia część (Alt+3)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Pisanie modułów jądra Linuksa

Do jądra systemu Linuks na stałe wkompilowane są tylko najważniejsze sterowniki podstawowych urządzeń (np. dyski twarde), gdyż umieszczanie tam wszystkich to strata pamięci a przede wszystkim czasu na uruchomienie i wyłączenie się sterowników do urządzeń nieistniejących w danym komputerze. Dlatego sterowniki do urządzeń opcjonalnych umieszczono w modułach jądra, ładowanych przez system na żądanie. Moduł jądra to najzwyklejszy skompilowany plik w standardowym formacie ELF. Musi eksportować na zewnątrz dwie funkcje: init_module, służącą do inicjalizacji modułu (i uruchamianą w czasie jego ładowania) oraz cleanup_module, służącą do wykonania czynności koniecznych do prawidłowego zakończenia pracy (uruchamianą w czasie usuwania modułu z jądra).
Funkcja init_module musi być tak napisana, że w przypadku sukcesu zwraca zero, a w przypadku porażki - najlepiej jedną ze znanych ujemnych wartości błędu, która dobrze będzie opisywać problem.

Sporo informacji dotyczących jądra 2.4 przenosi się na jądro 2.6, więc w sekcji poświęconej jądru 2.6 powiem tylko, co się zmieniło w stosunku do 2.4.

Najprostszy moduł jądra 2.4

(przeskocz najprostszy moduł)

Zgodnie z tym, co powiedziałem wyżej, najprostszy moduł wygląda tak:
(przeskocz kod najprostszego modułu)

```
format ELF

section ".text" executable      ; początek sekcji kodu

; eksportowanie dwóch wymaganych funkcji
public  init_module
public  cleanup_module

; deklaracja zewnętrznej funkcji, służącej do wyświetlania
extrn   printk

init_module:
        push    dword napis1    ; napis do wyświetlenia
        call    printk
        pop     eax             ; zdejmujemy argumenty ze stosu

        xor     eax, eax        ; zero oznacza brak błędu
        ret

cleanup_module:
        push    dword napis2
        call    printk
        pop     eax

        ret

section ".data" writeable
```

```
napis1          db      "<1> Jestem w init_module."   , 10, 0
napis2          db      "<1> Jestem w cleanup_module.", 10, 0

section ".modinfo"
__module_kernel_version db      "kernel_version=2.4.26", 0
__module_license        db      "license=GPL", 0
__module_author         db      "author=Bogdan D.", 0
__module_description    db "description=Pierwszy modul jadra.", 0
```

Zauważcie kilka spraw:

1. Wyświetlanie napisów odbywa się wewnętrzną funkcją jądra - printk. Działa ona podobnie do funkcji printf z języka C, która na etapie ładowania jądra jest oczywiście niedostępna.
   W skrócie: adres napisu podajemy na stosie, poprzedzając dodatkowymi danymi w odwrotnej kolejności, jeśli funkcja w ogóle ma wyświetlić jakieś zmienne w napisie, np. %d (liczba całkowita). Będzie to dokładniej pokazane na przykładowym module.
   Napis powinien się zaczynać wyrażeniem <N>, gdzie N to pewna liczba. Ma to pozwolić jądru rozróżnić powagę wiadomości. Nam wystarczy za N wstawiać 1.

   Jeśli wyświetlanych napisów nie widać na ekranie, to na pewno pojawią się po komendzie `dmesg` (zwykle na końcu) oraz w pliku /var/log/messages.

2. Składnia jest dla kompilatora FASM.
   Moduły kompilowane NASMem z niewiadomych przyczyn nie chciały mi wchodzić do jądra.

3. Każda funkcja jądra uruchamiana jest w konwencji C, czyli my sprzątamy argumenty ze stosu.

4. Nowa sekcja - modinfo.
   Zawiera informacje, dla której wersji jądra moduł jest przeznaczony, kto jest jego autorem, na jakiej jest licencji, argumenty. Nazwy zmiennych muszą pozostać bez zmian, treść po znakach równości powinniście pozmieniać według potrzeb.

Moduł ten, po kompilacji (`fasm modul_hello.asm`) instaluje się jako *root* komendą

```
insmod ./modul_hello.o
```

a usuwa z jądra - komendą

```
rmmod modul_hello
```

(zauważcie brak rozszerzenia .o).
Listę modułów obecnych w jądrze można otrzymać komendą `lsmod`.

Pokażę teraz, jak zarejestrować urządzenie znakowe, zająć dla niego zasoby IRQ oraz zakres portów i pamięci.

Rejestracja urządzenia znakowego

(przeskocz rejestrację urządzenia znakowego)

Do rejestracji urządzenia znakowego (czyli takiego, z którego można odczytywać po bajcie, w

przeciwieństwie do np. dysku twardego) służy eksportowana przez jądro funkcja `register_chrdev`. Przyjmuje ona 3 argumenty. Od lewej (ostatni wkładany na stos) są to:

1. Numer główny urządzenia, który sobie wybraliśmy. Można podać zero, wtedy jądro przydzieli nam jakiś wolny. Numer główny to pierwszy z dwóch numerów (drugi to poboczny), widoczny w szczegółowym widoku plików z katalogu /dev, np.

   ```
   crw-rw-rw-  1 root root 1, 5 sie 16 15:28 /dev/zero
   ```

   urządzenie /dev/zero ma numer główny 1 i poboczny 5, litera C na początku oznacza właśnie urządzenie znakowe. Inne oznaczenia to D (katalog), S (gniazdo), B (urządzenie blokowe), P (FIFO), L (dowiązanie symboliczne).
2. Adres nazwy urządzenia w postaci ciągu znaków zakończonego bajtem zerowym.
3. Adres struktury file_operations, do której wpiszemy adresy odpowiednich funkcji do operacji na pliku. Najważniejsze są: otwieranie, zamykanie, zapis i czytanie z urządzenia. Sama struktura wygląda tak dla jądra 2.4:
   (przeskocz strukturę file_operations)

   ```
   struct file_operations {
           struct module *owner;
           loff_t (*llseek) (struct file *, loff_t, int);
           ssize_t (*read) (struct file*, char*, size_t, loff_t *);
           ssize_t (*write) (struct file *, const char *, size_t,
                   loff_t *);
           int (*readdir) (struct file *, void *, filldir_t);
           unsigned int (*poll) (struct file *,
                   struct poll_table_struct *);
           int (*ioctl) (struct inode*, struct file*, unsigned int,
                   unsigned long);
           int (*mmap) (struct file *, struct vm_area_struct *);
           int (*open) (struct inode *, struct file *);
           int (*flush) (struct file *);
           int (*release) (struct inode *, struct file *);
           int (*fsync) (struct file*,struct dentry*, int datasync);
           int (*fasync) (int, struct file *, int);
           int (*lock) (struct file *, int, struct file_lock *);
           ssize_t (*readv) (struct file *, const struct iovec *,
                   unsigned long, loff_t *);
           ssize_t (*writev) (struct file *, const struct iovec *,
                   unsigned long, loff_t *);
   };
   ```

   Każde pole tej struktury to DWORD. Do podstawowej funkcjonalności wystarczy wypełnić pola: trzecie, czwarte, dziewiąte i jedenaste (zamykanie pliku). Jeśli jakiejś funkcji nie planujemy pisać, należy na odpowiadające jej miejsce w tej strukturze wpisać zero.

Jeśli podaliśmy tej funkcji nasz własny numer główny urządzenia, to jeśli rejestracja się udała, `register_chrdev` zwróci zero w EAX. Jeśli poprosiliśmy o przydzielenie nam numeru głównego, to jeśli rejestracja się powiedzie, `register_chrdev` zwróci liczbę większą od zera, która to liczba będzie przeznaczonym dla naszego urządzenia numerem głównym.
UWAGA: Funkcja `register_chrdev` nie tworzy pliku urządzenia w katalogu /dev. O to musimy zadbać sami, po załadowaniu modułu.

Wyrejestrowanie urządzenia znakowego następuje poprzez wywołanie funkcji `unregister_chrdev`. Pierwszy argument od lewej (ostatni na stos) to przydzielony urządzeniu numer główny, a drugi - adres nazwy

urządzenia.

Rejestracja portów wejścia-wyjścia oraz obszaru pamięci

(przeskocz rejestrację portów i pamięci)

Zarezerwowanie tych zasobów jest dość łatwe. Należy tylko uruchomić funkcję `__request_region`. Przyjmuje ona 4 argumenty. Od lewej (ostatni wkładany na stos) są to:

1. Typ zasobu. Jeśli chcemy zarezerwować porty, podajemy tu adres zmiennej `ioport_resource`, jeśli pamięć - `iomem_resource`. Obie zmienne są eksportowane przez jądro, więc można je zadeklarować jako zewnętrzne dla modułu.
2. Początkowy numer portu lub początkowy adres pamięci.
3. Długość zakresu portów lub pamięci
4. Adres nazwy urządzenia.

W przypadku niepowodzenia, funkcja zwraca zero (w EAX).

Oba te rodzaje zasobów zwalnia się funkcją `__release_region`. Jako swoje argumenty przyjmuje ona 3 pierwsze z powyższych (typ oraz początek i długość zakresu).

Rejestracja zasobu IRQ

(przeskocz rejestrację IRQ)

Zasoby żądania przerwania (IRQ) rejestruje się funkcją `request_irq`. Przyjmuje ona aż 5 argumentów typu DWORD. Od lewej (ostatni wkładany na stos) są to:

1. Numer przerwania IRQ, które chcemy zająć.
2. Adres naszej funkcji, która będzie obsługiwać przerwania. Prototyp takiej funkcji wygląda tak:

   ```
   void handler (int irq, void *dev_id, struct pt_regs *regs);
   ```

   Jak widać, będzie można ze stosu otrzymać informacje, które przerwanie zostało wywołane oraz przez jakie urządzenie. Ostatni argument podobno jest już rzadko używany.
3. Wartość SA_INTERRUPT = 0x20000000
4. Adres nazwy urządzenia.
5. Adres struktury file_operations, uzupełnionej adresami funkcji

Jeśli zajęcie przerwania się nie powiedzie, funkcja zwróci wartość ujemną.

Zwolnienie przerwania odbywa się poprzez wywołanie funkcji `free_irq`. Jej pierwszy argument od lewej (ostatni na stos) to nasz numer IRQ, a drugi - adres naszej struktury file_operations.

Przykład modułu jądra 2.4

(przeskocz do jądra 2.6)

Pokazany niżej program zarejestruje programowe urządzenie znakowe (czyli takie, dla którego nie ma odpowiednika w sprzęcie, jak np. /dev/null) z IRQ 4, zakresem portów 600h-6FFh, zakresem pamięci 80000000h - 8000FFFFh oraz z podstawowymi operacjami: otwieranie, zamykanie, czytanie, zapis, zmiana pozycji. Dla uproszczenia kodu nie sprawdzam, czy dane zakresy są wolne. Jeśli okażą się zajęte, jądro zwróci błąd i moduł się nie załaduje.
(przeskocz kod modułu)

```
; Przykładowy moduł jądra 2.4
;
; Autor: Bogdan D., bogdandr (na) op . pl
;
; kompilacja:
;   fasm modul_dev_fasm.asm

format ELF
section ".text" executable

; eksportowanie wymaganych funkcji
public  init_module
public  cleanup_module

; importowanie używanych funkcji i symboli
extrn   printk
extrn   register_chrdev
extrn   unregister_chrdev
extrn   request_irq
extrn   free_irq

extrn   __check_region
extrn   __request_region
extrn   __release_region
extrn   ioport_resource
extrn   iomem_resource

; zakresy zasobów, o które poprosimy jądro
PORTY_START     = 0x600
PORTY_ILE       = 0x100

RAM_START       = 0x80000000
RAM_ILE         = 0x00010000

; stałe potrzebne do rezerwacji przerwania IRQ.
SA_INTERRUPT    = 0x20000000
NUMER_IRQ       = 4

; funkcja inicjalizacji modułu
init_module:
        pushfd

        ; rejestrowanie urządzenia znakowego:
        push    dword file_oper
        push    dword nazwa
        push    dword 0                 ; numer przydziel dynamicznie
        call    register_chrdev
```

```
        add     esp, 3*4                ; usuwamy argumenty ze stosu

        cmp     eax, 0                  ; sprawdzamy, czy błąd
        jg      .dev_ok

        ; tu wiemy, że jest błąd. wyświetlmy to.
        push    eax                     ; argument do informacji o błędzie
        push    dword dev_blad          ; adres informacji o błędzie
        call    printk                  ; wyświetl informację o błędzie
        add     esp, 1*4                ; specjalnie usuwam tylko 1*4

        pop     eax                     ; wychodzimy z oryginalnym błędem
        jmp     .koniec

.dev_ok:

        mov     [major], eax

        ; rezerwacja portów wejścia-wyjścia
        push    dword nazwa
        push    dword PORTY_ILE
        push    dword PORTY_START
        push    dword ioport_resource
        call    __request_region
        add     esp, 4*4

        test    eax, eax                ; sprawdzamy, czy błąd
        jnz     .iop_ok

        push    eax                     ; argument do informacji o błędzie
        push    dword porty_blad                ; adres informacji o błędzie
        call    printk                  ; wyświetl informację o błędzie
        add     esp, 1*4                ; potem pop eax

        ; wyrejestrowanie urządzenia
        push    dword nazwa
        push    dword [major]
        call    unregister_chrdev
        add     esp, 2*4

        pop     eax                     ; wychodzimy z oryginalnym błędem
        jmp     .koniec

.iop_ok:

        ; rezerwacja pamięci
        push    dword nazwa
        push    dword RAM_ILE
        push    dword RAM_START
        push    dword iomem_resource
        call    __request_region
        add     esp, 4*4

        test    eax, eax                ; sprawdzamy, czy błąd
        jnz     .iomem_ok

        push    eax
        push    dword ram_blad
        call    printk                  ; wyświetl informację o błędzie
        add     esp, 1*4                ; potem pop eax

        ; wyrejestrowanie urządzenia
```

```
        push    dword nazwa
        push    dword [major]
        call    unregister_chrdev
        add     esp, 2*4

        ; zwolnienie zajętych przez nas portów
        push    dword PORTY_ILE
        push    dword PORTY_START
        push    dword ioport_resource
        call    __release_region
        add     esp, 3*4

        pop     eax                     ; wychodzimy z oryginalnym błędem
        jmp     .koniec

.iomem_ok:
        ; przydzielanie przerwania IRQ:
        push    dword file_oper
        push    dword nazwa
        push    dword SA_INTERRUPT
        push    dword obsluga_irq
        push    dword NUMER_IRQ
        call    request_irq
        add     esp, 5*4

        cmp     eax, 0
        jge     .irq_ok

        push    eax
        push    dword irq_blad
        call    printk                  ; wyświetl informację o błędzie
        add     esp, 1*4                ; potem pop eax

        ; wyrejestrowanie urządzenia
        push    dword nazwa
        push    dword [major]
        call    unregister_chrdev
        add     esp, 2*4

        ; zwolnienie zajętych przez nas portów
        push    dword PORTY_ILE
        push    dword PORTY_START
        push    dword ioport_resource
        call    __release_region
        add     esp, 3*4

        ; zwolnienie zajętej przez nas pamięci
        push    dword RAM_ILE
        push    dword RAM_START
        push    dword iomem_resource
        call    __release_region
        add     esp, 3*4

        pop     eax                     ; wychodzimy z oryginalnym błędem
        jmp     .koniec

.irq_ok:

        ; wyświetlenie informacji o poprawnym uruchomieniu modułu
        push    dword NUMER_IRQ
        push    dword [major]
        push    dword uruch
```

```
        call    printk
        add     esp, 3*4

        xor     eax, eax                ; zero - brak błędu

.koniec:

        popfd
        ret

; funkcja uruchamiana przy usuwaniu modułu
cleanup_module:
        pushfd
        push    eax

        ; zwolnienie numeru IRQ:
        push    dword file_oper
        push    dword NUMER_IRQ
        call    free_irq
        add     esp, 2*4

        ; wyrejestrowanie urządzenia:
        push    dword nazwa
        push    dword [major]
        call    unregister_chrdev
        add     esp, 2*4

        ; zwolnienie zajętych przez nas portów
        push    dword PORTY_ILE
        push    dword PORTY_START
        push    dword ioport_resource
        call    __release_region
        add     esp, 3*4

        ; zwolnienie zajętej przez nas pamięci
        push    dword RAM_ILE
        push    dword RAM_START
        push    dword iomem_resource
        call    __release_region
        add     esp, 3*4

        ; wyświetlenie informacji o usunięciu modułu
        push    dword usun
        call    printk
        add     esp, 1*4

        pop     eax
        popfd
        ret

; nasza funkcja obsługi przerwania. Ta tutaj nie robi nic, ale
;       pokazuje rozmieszczenie argumentów na stosie
obsluga_irq:
        push    ebp
        mov     ebp, esp

; [ebp] = stary EBP
; [ebp+4] = adres powrotny
; [ebp+8] = arg1
; ...

                irq     equ     ebp+8
```

```
                dev_id  equ     ebp+12
                regs    equ     ebp+16

        leave
        ret


; Zdefiniowane operacje na urządzeniu

; Czytanie z urządzenia - zwracamy żądanej długości ciąg bajtów 1Eh.
; To urządzenie staje się nieskończonym źródłem, podobnie jak /dev/zero
czytanie:
        push    ebp
        mov     ebp, esp

        ; rozmieszczenie argumentów na stosie:
        s_file  equ     ebp+8   ; wskaźnik na strukturę file
        bufor   equ     ebp+12  ; adres bufora na dane
        l_jedn  equ     ebp+16  ; żądana liczba bajtów
        loff    equ     ebp+20  ; żądany offset czytania

        pushfd
        push    edi
        push    ecx

        mov     ecx, [l_jedn]
        mov     al, 0x1e
        cld
        mov     edi, [bufor]
        rep     stosb           ; zapychamy bufor bajtami 1Eh

        pop     ecx
        pop     edi
        popfd

        mov     eax, [l_jedn]   ; zwracamy tyle, ile chciano

        leave
        ret

zapis:
        push    ebp
        mov     ebp, esp

        ; nic fizycznie nie zapisujemy, zwracamy tylko liczbę bajtów,
        ;       którą mieliśmy zapisać
        mov     eax, [l_jedn]

        leave
        ret

przejscie:
zamykanie:
otwieranie:
        xor     eax, eax
        ret



section ".data" writeable

major   dd      0       ; numer główny urządzenia przydzielany przez jądro
```

```
; adresy funkcji operacji na tym urządzeniu
file_oper:      dd 0, przejscie, czytanie, zapis, 0, 0, 0, 0, otwieranie, 0
                dd zamykanie, 0, 0, 0, 0, 0

dev_blad        db      "<1>Blad rejestracji urzadzenia: %d.", 10, 0
irq_blad        db      "<1>Blad przydzielania IRQ: %d.", 10, 0
porty_blad      db      "<1>Blad przydzielania portow:  EAX=%d", 10, 0
ram_blad        db      "<1>Blad przydzielania pamieci: EAX=%d", 10, 0


uruch           db      "<1>Modul zaladowany. Maj=%d, IRQ=%d", 10, 0
usun            db      "<1>Modul usuniety.", 10, 0

nazwa           db      "test00", 0
sciezka         db      "/dev/test00", 0

section ".modinfo"
__module_kernel_version db      "kernel_version=2.4.26", 0
__module_license        db      "license=GPL", 0
__module_author         db      "author=Bogdan D.", 0
__module_description    db      "description=Pierwszy modul jadra", 0
__module_device         db      "device=test00", 0
```

Powyższy moduł po kompilacji najprościej zainstalować w jądrze stosując taki oto skrypt:
(przeskocz skrypt instalacji)

```
#!/bin/bash

PLIK="modul_dev_fasm.o"         # Tu wstawiacie swoją nazwę
NAZWA="test00"

# umieszczenie modułu w jądrze.
/sbin/insmod $PLIK $* || { echo "Problem insmod!" ; exit -1; }

# wyszukanie naszej nazwy modułu wśród wszystkich
/sbin/lsmod | grep `echo $PLIK | sed 's/[^a-z]/ /g' | awk '{print $1}' `
# wyświetlenie informacji o zajmowanych zasobach
grep $NAZWA /proc/devices
grep $NAZWA /proc/ioports
grep $NAZWA /proc/iomem
grep $NAZWA /proc/interrupts

# znalezienie i wyświetlenie numeru głównego urządzenia
NR=`grep $NAZWA /proc/devices | awk '{print $1}'`
echo "Major = $NR"

# ewentualne usunięcie starego pliku urządzenia
rm -f /dev/$NAZWA

# fizyczne utworzenie pliku urządzenia w katalogu /dev
# wykonanie funkcji sys_mknod z modułu NIE działa
mknod /dev/$NAZWA c $NR 0
ls -l /dev/$NAZWA

# krótki test: czytanie 512 bajtów i sprawdzenie ich zawartości
dd count=1 if=/dev/$NAZWA of=/x && hexdump /x && rm -f /x
```

Wystarczy ten skrypt zachować np. pod nazwą `instal.sh`, nadać prawo wykonywania komendą `chmod u+x instal.sh` i uruchamiać poprzez `./instal.sh`, oczywiście jako *root*. Jeśli załadowanie modułu

się uda, skrypt wyświetli przydzielone modułowi zasoby - porty, IRQ, pamięć - poprzez zajrzenie do odpowiednich plików katalogu /proc. Skrypt utworzy też plik urządzenia w katalogu /dev z odpowiednim numerem głównym oraz wykona prosty test.

Odinstalować moduł można łatwo takim oto skryptem:

```
#!/bin/bash

PLIK="modul_dev_fasm"   # Tu wstawiacie swoją nazwę, bez rozszerzenia .o
NAZWA="test00"

/sbin/rmmod $PLIK && rm -f /dev/$NAZWA
```

Najprostszy moduł jądra 2.6

(przeskocz najprostszy moduł jądra 2.6)

Najprostszy moduł jądra 2.6 wygląda tak:
(przeskocz kod najprostszego modułu jądra 2.6)

```
format ELF
section ".init.text" executable align 1
section ".text" executable align 4

public init_module
public cleanup_module

extrn printk

init_module:
        push    dword str1
        call    printk
        pop     eax
        xor     eax, eax
        ret

cleanup_module:
        push    dword str2
        call    printk
        pop     eax
        ret

section ".modinfo" align 32
__kernel_version        db      "kernel_version=2.6.16", 0
__mod_vermagic db "vermagic=2.6.16 686 REGPARM 4KSTACKS gcc-4.0", 0
__module_license        db      "license=GPL", 0
__module_author         db      "author=Bogdan D.", 0
__module_description    db      "description=Pierwszy modul jadra 2.6", 0

section "__versions" align 32
        dd      0xfa02c634
  n1:   db      "struct_module"
        times   64-4-($-n1) db 0

        dd      0x1b7d4074
  n2:   db      "printk"
        times   64-4-($-n2) db 0
```

```
section ".data" writeable align 4

str1            db      "<1> Jestem w init_module(). ", 10, 0
str2            db      "<1> Jestem w cleanup_module(). ", 10, 0

section ".gnu.linkonce.this_module" writeable align 128

align 128
__this_module:          ; łączna długość: 512 bajtów
                        dd 0, 0, 0

                .nazwa: db "modul", 0
                        times 64-4-($-.nazwa) db 0

                        times 100 db 0
                        dd init_module
                        times 220 db 0
                        dd cleanup_module
                        times 112 db 0
```

Od razu widać sporo różnic, prawda? Omówmy je po jednej sekcji na raz:

1. `.init.text`
   W zasadzie powinny być co najmniej dwie: `.init.text`, zawierająca procedurę inicjalizacji oraz `.exit.text`, zawierająca procedurę wyjścia.

   Dodatkowo, może być oczywiście sekcja danych `.data` i kodu `.text`.

   Jeśli podczas próby zainstalowania modułu dostajecie komunikat Accessing a corrupted shared library (Dostęp do uszkodzonej biblioteki współdzielonej), to pogrzebcie w sekcjach - doróbcie `.text`, usuńcie `.init.text`, zamieńcie kolejność itp.

2. `.gnu.linkonce.this_module`
   Ta jest najważniejsza. Bez niej próba instalacji modułu w jadrze zakończy się komunikatem No module found in object (w pliku obiektowym nie znaleziono modułu). Zawartość tej sekcji to struktura typu module o nazwie __this_module. Najlepiej zrobicie, przepisując tą powyżej do swoich modułów, zmieniając tylko nazwę modułu oraz funkcje wejścia i wyjścia.

   Możecie też skorzystać z następującego makra:

```
        macro   gen_this_module         name*, entry, exit
        {
                section '.gnu.linkonce.this_module' writeable align 128

                align 128
                __this_module:
                                dd 0, 0, 0
                .mod_nazwa:     db name, 0
                                times 64-4-($-.mod_nazwa) db 0
                                times 100 db 0
                                if entry eq
                                        dd init_module
                                else
                                        dd entry
                                end if
                                times 220 db 0
                                if exit eq
```

```
        dd cleanup_module
else
        dd exit
end if
times 112 db 0

}
```

Korzysta sie niego dość łatwo: wystarczy podać nazwę modułu, która ma być wyświetlana po komendzie `lsmod` oraz nazwy procedur wejścia i wyjścia z modułu, np.

```
gen_this_module "nasz_modul", init_module, cleanup_module
```

To wywołanie makra należy umieścić tam, gdzie normalnie ta sekcja by się znalazła, czyli np. po ostatniej deklaracji czegokolwiek w sekcji danych. W każdym razie *NIE* tak, żeby było to w środku jakiejkolwiek sekcji.

3. `modinfo`
   Sekcja ta wzbogaciła się w stosunku do tej z jądra 2.4 o tylko jeden, ale za to bardzo ważny wpis - vermagic. U większości z Was ten napis będzie się różnił od mojego tylko wersją jądra. W oryginale wygląda on tak:
   (przeskocz definicję vermagic)

```
#define VERMAGIC_STRING                                 \
  UTS_RELEASE " "                                       \
  MODULE_VERMAGIC_SMP MODULE_VERMAGIC_PREEMPT           \
  MODULE_ARCH_VERMAGIC                                  \
  "gcc-" __stringify(__GNUC__) "." __stringify(__GNUC_MINOR__)
#define MODULE_ARCH_VERMAGIC MODULE_PROC_FAMILY \
        MODULE_REGPARM MODULE_STACKSIZE
```

   a można go znaleźć w podkatalogach asm* katalogu INCLUDE w źródłach jądra oraz w pliku VERMAGIC.H.

4. `__versions`
   Ta sekcja zawiera informacje o wersjach procedur, z których nasz moduł korzysta. Struktura jest dość prosta: najpierw jako DWORD wpisujemy numerek odpowiadający danej funkcji jądra, a znaleziony w pliku MODULE.SYMVERS w katalogu głównym źródeł jadra. Zaraz za numerkiem wpisujemy nazwę naszej funkcji, dopełnioną zerami do 64 bajtów.

   Ta sekcja nie jest wymagana do prawidłowej pracy modułu, ale powinna się w każdym znaleźć, żeby nie pojawiały się komunikaty o zanieczyszczeniu jądra (kernel tainted).

   Całą tę sekcję możecie wygenerować, korzystając z mojego skryptu symvers-fasm.txt. Wystarczy uruchomić `perl symvers-fasm.pl wasz_modul.asm`.

Rezerwacja zasobów w jądrze 2.6

(przeskocz rezerwację zasobów w jądrze 2.6)

Rezerwacja zasobów w jądrze 2.6 z zewnątrz (czyli z perspektywy języka C) nie różni się od tej z jądra 2.4.

Ale tak naprawdę zaszły dwie istotne zmiany:

1. Struktura file_operations
   W jądrze 2.6 wygląda tak:
   (przeskocz strukturę file_operations w jądrze 2.6)

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file*,char __user*,size_t,
                loff_t*);
        ssize_t (*aio_read) (struct kiocb *, char __user *,
                size_t, loff_t);
        ssize_t (*write) (struct file *, const char __user *,
                size_t, loff_t *);
        ssize_t (*aio_write) (struct kiocb *, const char __user*,
                size_t, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *,
                struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *,
                unsigned int, unsigned long);
        long (*unlocked_ioctl) (struct file *, unsigned int,
                unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int,
                unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *,
                int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*readv) (struct file *, const struct iovec *,
                unsigned long, loff_t *);
        ssize_t (*writev) (struct file *, const struct iovec *,
                unsigned long, loff_t *);
        ssize_t (*sendfile) (struct file *, loff_t *, size_t,
                read_actor_t, void *);
        ssize_t (*sendpage) (struct file *, struct page *, int,
                size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *,
                unsigned long, unsigned long, unsigned long,
                unsigned long);
        int (*check_flags)(int);
        int (*dir_notify)(struct file *filp, unsigned long arg);
        int (*flock) (struct file *, int, struct file_lock *);
};
```

2. Sposób przekazywania parametrów
   Moje jądro dystrybucyjne zostało skompilowane tak, żeby trzy pierwsze parametry do każdej procedury z wyjątkiem printk przekazywało w rejestrach: EAX, EDX, ECX, a resztę na stosie. Aby sprawdzić, czy u Was też tak jest, wykonajcie komendy:

```
grep -R regpar /lib/modules/`uname -r`/build/|grep Makefile
grep -R REGPAR /lib/modules/`uname -r`/build/|grep config
```

   Jeśli ich wyniki zawierają takie coś:

```
CONFIG_REGPARM=y
#define CONFIG_REGPARM 1
```

to prawdopodobnie też tak macie. Możecie wtedy bez przeszkód używać makra URUCHOM, które umieszczę w module poniżej. Jeśli nie, możecie je zmodyfikować. Potrzeba modyfikacji może wynikać z zawieszania się całego systemu podczas próby zainstalowania modułu.

Przykład modułu jądra 2.6

(przeskocz przykład modułu jądra 2.6)

Podobnie, jak w jądrze 2.4, pokazany niżej program zarejestruje programowe urządzenie znakowe (czyli takie, dla którego nie ma odpowiednika w sprzęcie, jak np. /dev/null) z IRQ 4, zakresem portów 600h-6FFh, zakresem pamięci 80000000h - 8000FFFFh oraz z podstawowymi operacjami: otwieranie, zamykanie, czytanie, zapis, zmiana pozycji. Dla uproszczenia kodu nie sprawdzam, czy dane zakresy są wolne. Jeśli okażą się zajęte, jądro zwróci błąd i moduł się nie załaduje.

```
format ELF
section ".text" executable align 4

public  init_module
public  cleanup_module

extrn   printk
extrn   register_chrdev
extrn   unregister_chrdev
extrn   request_irq
extrn   free_irq

extrn   __request_region
extrn   __release_region
extrn   ioport_resource
extrn   iomem_resource

PORTY_START     = 0x600
PORTY_ILE       = 0x100

RAM_START       = 0x80000000
RAM_ILE         = 0x00010000

SA_INTERRUPT    = 0x20000000
NUMER_IRQ       = 4

macro   uruchom         funkcja, par1, par2, par3, par4, par5
{
        if ~ par5 eq
                push    dword par5
        end if
        if ~ par4 eq
                push    dword par4
        end if
        if ~ par3 eq
                mov     ecx, par3
        end if
        if ~ par2 eq
                mov     edx, par2
```

```
        end if
        if ~ par1 eq
                mov     eax, par1
        end if
        call    funkcja
        if ~ par5 eq
                add     esp, 4
        end if
        if ~ par4 eq
                add     esp, 4
        end if
}

init_module:
        pushfd

        ; rejestrowanie urządzenia znakowego:
        uruchom register_chrdev, 0, nazwa, file_oper

        cmp     eax, 0
        jg      .dev_ok

        ; wyświetlenie informacji o błędzie
        push    eax
        push    dword dev_blad
        call    printk
        add     esp, 1*4                ; specjalnie tylko 1*4

        pop     eax                     ; wychodzimy z oryginalnym błędem
        jmp     .koniec

.dev_ok:

        mov     [major], eax

        ; rejestrowanie zakresu portów
 uruchom __request_region, ioport_resource, PORTY_START, PORTY_ILE, nazwa

        test    eax, eax
        jnz     .iop_ok

        push    eax
        push    dword porty_blad
        call    printk
        add     esp, 1*4                ; potem pop eax

        ; wyrejestrowanie urządzenia
        uruchom unregister_chrdev, [major], nazwa

        pop     eax                     ; wychodzimy z oryginalnym błędem
        jmp     .koniec

.iop_ok:

        ; rejestrowanie zakresu pamięci
        uruchom __request_region, iomem_resource, RAM_START, RAM_ILE, nazwa

        test    eax, eax
        jnz     .iomem_ok

        push    eax
        push    dword ram_blad
```

```
        call    printk
        add     esp, 1*4                ; potem pop eax

        ; wyrejestrowanie urządzenia
        uruchom unregister_chrdev, [major], nazwa

        ; wyrejestrowanie zakresu portów
        uruchom __release_region, ioport_resource, PORTY_START, PORTY_ILE

        pop     eax                     ; wychodzimy z oryginalnym błędem
        jmp     .koniec

.iomem_ok:

        ; przydzielanie przerwania IRQ:
 uruchom request_irq, NUMER_IRQ, obsluga_irq, SA_INTERRUPT, nazwa, file_oper

        cmp     eax, 0
        jge     .irq_ok

        push    eax
        push    dword irq_blad
        call    printk
        add     esp, 1*4                ; potem pop eax

        ; wyrejestrowanie urządzenia
        uruchom unregister_chrdev, [major], nazwa

        ; wyrejestrowanie zakresu portów
        uruchom __release_region, ioport_resource, PORTY_START, PORTY_ILE

        ; wyrejestrowanie zakresu pamięci
        uruchom __release_region, iomem_resource, RAM_START, RAM_ILE

        pop     eax                     ; wychodzimy z oryginalnym błędem
        jmp     .koniec

.irq_ok:

        ; wyświetlenie informacji o załadowaniu modułu
        push    dword NUMER_IRQ
        push    dword [major]
        push    dword uruch
        call    printk
        add     esp, 3*4

        xor     eax, eax

.koniec:

        popfd
        ret

; funkcja uruchamiana przy usuwaniu modułu
cleanup_module:
        pushfd
        push    eax

        ; zwolnienie numeru IRQ:
        uruchom free_irq, NUMER_IRQ, file_oper

        ; wyrejestrowanie urządzenia:
```

```
        uruchom unregister_chrdev, [major], nazwa

        ; wyrejestrowanie zakresu portów
        uruchom __release_region, ioport_resource, PORTY_START, PORTY_ILE

        ; wyrejestrowanie zakresu pamięci
        uruchom __release_region, iomem_resource, RAM_START, RAM_ILE

        push    dword usun
        call    printk
        add     esp, 1*4

        pop     eax
        popfd
        ret

; deklaracja wygląda tak:
; void handler (int irq, void *dev_id, struct pt_regs *regs);
; ostatni argument zwykle nieużywany

section ".text" executable align 4

obsluga_irq:
        push    ebp
        mov     ebp, esp

        ; tu Wasz kod

        leave
        ret

; Zdefiniowane operacje:

czytanie:
;       ssize_t (*read) (struct file *, char *, size_t, loff_t *);
        push    ebp
        mov     ebp, esp

        loff    equ     ebp+8

        pushfd
        push    edi
        push    ecx

        mov     al, 0x1e
        cld
        mov     edi, edx
        rep     stosb

        pop     ecx
        pop     edi
        popfd

        ; mówimy, że przeczytano tyle bajtów, ile żądano
        mov     eax, ecx

        leave
        ret

zapis:
;       ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
        push    ebp
```

```
        mov     ebp, esp

        ; nic fizycznie nie zapisujemy, zwracamy tylko liczbę bajtów,
        ;       którą mieliśmy zapisać (trzeci parametr)
        mov     eax, ecx

        leave
        ret

przejscie:
zamykanie:
otwieranie:
        xor     eax, eax
        ret



section ".data" writeable align 4

major   dd      0       ; numer główny urządzenia przydzielany przez jądro

; adresy funkcji operacji na tym urządzeniu
file_oper:      dd 0, przejscie, czytanie, 0, zapis, 0, 0, 0, 0, 0, 0, 0
                dd otwieranie, 0, zamykanie, 0, 0, 0, 0, 0, 0, 0, 0, 0
                dd 0, 0, 0
                dd 0, 0, 0

dev_blad        db      "<1>Blad rejestracji urzadzenia: %d.", 10, 0
irq_blad        db      "<1>Blad przydzielania IRQ: %d.", 10, 0
porty_blad      db      "<1>Blad przydzielania portow:  EAX=%d", 10, 0
ram_blad        db      "<1>Blad przydzielania pamieci: EAX=%d", 10, 0


uruch           db      "<1>Modul zaladowany. Maj=%d, IRQ=%d", 10, 0
usun            db      "<1>Modul usuniety.", 10, 0

nazwa           db      "test00", 0, 0
sciezka         db      "/dev/test00", 0

section ".modinfo" align 32
__kernel_version        db      "kernel_version=2.6.16", 0
__mod_vermagic db "vermagic=2.6.16 686 REGPARM 4KSTACKS gcc-4.0",0
__module_license        db      "license=GPL", 0
__module_author         db      "author=Bogdan D.", 0
__module_description    db      "description=Pierwszy modul jadra 2.6", 0
__module_device         db      "device=test00", 0
__module_depends        db      "depends=", 0

; nieistotne, wzięte ze skompilowanego modułu C:
__mod_srcversion        db      "srcversion=F5CE0CFFE0191EDB2F816D4", 0

section "__versions" align 32

____versions:
        dd      0xfa02c634              ; Z MODULE.SYMVERS
  n1:   db      "struct_module", 0
        times   64-4-($-n1) db 0

        dd      0x1b7d4074
  n2:   db      "printk", 0
        times   64-4-($-n2) db 0
```

```
        dd      0xb5145e00
  n3:   db      "register_chrdev", 0
        times   64-4-($-n3) db 0

        dd      0xc192d491
  n4:   db      "unregister_chrdev", 0
        times   64-4-($-n4) db 0

        dd      0x26e96637
  n5:   db      "request_irq", 0
        times   64-4-($-n5) db 0

        dd      0xf20dabd8
  n6:   db      "free_irq", 0
        times   64-4-($-n6) db 0

        dd      0x1a1a4f09
  n7:   db      "__request_region", 0
        times   64-4-($-n7) db 0

        dd      0xd49501d4
  n8:   db      "__release_region", 0
        times   64-4-($-n8) db 0

        dd      0x865ebccd
  n9:   db      "ioport_resource", 0
        times   64-4-($-n9) db 0

        dd      0x9efed5af
  n10:  db      "iomem_resource", 0
        times   64-4-($-n10) db 0


section ".gnu.linkonce.this_module" writeable align 128

align 128
__this_module:          ; łączna długość: 512 bajtów
                        dd 0, 0, 0
        .mod_nazwa:     db "modul_dev_fasm", 0
                        times 64-4-($-.mod_nazwa) db 0
                        times 100 db 0
                        dd init_module
                        times 220 db 0
                        dd cleanup_module
                        times 112 db 0
```

Do instalacji i usuwania modułu z jądra można użyć tych samych skryptów, które były dla jądra 2.4, zmieniając ewentualnie nazwę pliku modułu.

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Bezpośredni dostęp do ekranu pod Linuksem

Jeśli chodzi o wyświetlanie informacji na ekranie, nie jesteśmy ograniczeni tylko do pisania w miejscu, gdzie akurat znajduje się kursor. Na pewno widzieliście jakiś program, który mimo iż miał tekstowy interfejs, to jednak pisał po ekranie, gdzie mu się podobało. Tym właśnie się teraz zajmiemy.

Pisanie z wykorzystaniem sekwencji kontrolnych terminala

(przeskocz sekwencje kontrolne)
Każdy program terminala ma inne sekwencje kontrolne i jeśli chcecie pisać programy, które będą działać na każdym terminalu, zainteresujcie się biblioteką ncurses. Tutaj opiszę tylko kilka sekwencji standardowego terminala xterm.

Pierwsza sprawa: co to właściwie jest znak kontrolny (sekwencja kontrolna)?
Jest to specjalny ciąg znaków określających zachowanie się terminala. Kilka już na pewno znacie: BEL (dźwięk), CR/LF (przechodzenie do nowej linii), TAB (tabulator). Teraz dojdą jeszcze dwa: zmiana koloru tekstu i tła oraz przechodzenie do wyznaczonej pozycji na ekranie.
Korzystałem z pliku xterm_controls.txt. Możecie skorzystać także z tego pliku lub z informacji na stronie podręcznika - man 4 console_codes.

Kolorowanie tekstu

(przeskocz kolorowanie)

Sekwencja kontrolna odpowiedzialna za kolor tekstu i tła wygląda tak:
`ESC[(atr);(lit);(tło)m`
gdzie:

- ESC to kod klawisza Escape, czyli 1Bh. Jeśli ktoś jest ciekawy, to z klawiatury otrzymuje się ten znak wciskając Ctrl+V, po czym ESC.

- (atr) = atrybut znaku. Jest to jedna z poniższych wartości (jako tekst, nie binarnie):
  - 0 - przywróć wszystkie atrybuty (powrót do trybu normalnego)
  - 1 - jasny (zwykle włącza też wytłuszczoną czcionkę)
  - 2 - przyciemnienie znaku
  - 3 - podkreślenie
  - 5 - mruganie znaku
  - 7 - odwrócenie kolorów
  - 8 - ukryty

- (lit) = kolor litery:
  - 30 - czarny
  - 31 - czerwony
  - 32 - zielony
  - 33 - żółty
  - 34 - niebieski
  - 35 - magenta (różowy)
  - 36 - cyan (błękitny)
  - 37 - biały

- (tło) = kolor tła:
  - 40 - czarny
  - 41 - czerwony
  - 42 - zielony
  - 43 - żółty
  - 44 - niebieski
  - 45 - magenta (różowy)
  - 46 - cyan (błękitny)
  - 47 - biały

- m - dosłownie: litera małe m

Na przykład, aby napisać coś na czerwono i przywrócić oryginalne kolory konsoli, należy normalnie (czyli przy użyciu int 80h z EAX=4, EBX=1, ECX=adres, EDX=długość) wyświetlić taki oto ciąg znaków:
`1bh, "[0;31;40m Napis", 1bh, "[0;37;40m".`
Ten ostatni ciąg przywraca domyślne kolory terminala (szarobiały na czarnym tle). Jeśli używacie terminala używającego innego zestawu kolorów niż szarobiały na czarnym tle, możecie wstawić własne wartości, tak samo jak dla zwykłych napisów - terminal zapamięta ustawienia. Możecie też spróbować takiej sekwencji:
`1bh, "[0;31;40m Napis", 1bh, "[0;39;49m".`
Wartości 39 i 49 przywracają domyślne kolory, odpowiednio dla znaków i tła.
Można też spróbować przywrócenia domyślnych wartości wszystkich atrybutów (nie tylko kolorów) bez ustawiania nowych wartości:
`1bh, "[0;31;40m Napis", 1bh, "[0m".`

Zmiana bieżącej pozycji kursora

(przeskocz teorię)

Sekwencja kontrolna odpowiedzialna za ustalanie pozycji kursora wygląda tak:
`ESC [ w ; k H`
gdzie:

- ESC to kod klawisza Escape, czyli 1Bh.
- w = numer wiersza (jeśli nie podano, przyjmuje się 1)
- k = numer kolumny (jeśli nie podano, przyjmuje się 1)
- H - dosłownie: litera wielkie H

Na przykład, jeśli chcemy coś napisać w dziesiątym wierszu dziesiątej kolumny, należy normalnie (czyli przy użyciu int 80h z EAX=4, EBX=1, ECX=adres, EDX=długość) wyświetlić ciąg znaków:
`1bh, "[10;10HNapis"`

A oto obiecany program do rysowania ramek:
(przeskocz program)

```
; Rysowanie okienek z ramka
;
; Autor: Bogdan D.
;
; nasm -O999 -o ramki.o -f elf ramki.asm
; ld -s -o ramki ramki.o

section .text
global _start


_start:
        mov     eax, 4
        mov     ebx, 1
        mov     ecx, czysc
        mov     edx, czysc_dl
        int     80h                     ; wyświetlamy sekwencję,
                                        ; która wyczyści ekran

        mov     ax, (36<<8)+44          ; kolor znaków, kolor tła:
                                        ; żółty na niebieskim
        mov     bx, 1                   ; kolumna Lewa-Górna (L-G)
        mov     cx, 1                   ; wiersz L-G
        mov     si, 9                   ; kolumna Prawa-Dolna (P-D)
        mov     bp, 9                   ; wiersz P-D
        call    rysuj_okienko

        mov     ax, (37<<8)+40          ; biały na czarnym
        mov     bx, 10
        mov     cx, 10
        mov     si, 20
        mov     bp, 16
        call    rysuj_okienko

        mov     eax, 4
        mov     ebx, 1
        mov     ecx, nwln
        mov     edx, 1
        int     80h                     ; wyświetlamy znak przejścia
                                        ; do nowej linii

        mov     eax, 1
        xor     ebx, ebx
```

```
        int     80h                     ; wyjście z programu


rysuj_okienko:

; wejście:
;
;  AH = atrybut znaku (kolor)
;  AL = kolor tła
;  BX = kolumna lewego górnego rogu
;  CX = wiersz lewego górnego rogu
;  SI = kolumna prawego dolnego rogu
;  BP = wiersz prawego dolnego rogu
;
; wyjście:
;  nic

; podwójne ramki ASCII
;r_p    equ     0bah                    ; prawa boczna
;r_pg   equ     0bbh                    ; prawa górna (narożnik)
;r_pd   equ     0bch                    ; prawa dolna

;r_g    equ     0cdh                    ; górna
;r_d    equ     r_g                     ; dolna

;r_l    equ     r_p                     ; lewa boczna
;r_lg   equ     0c9h                    ; lewa górna
;r_ld   equ     0c8h                    ; lewa dolna


r_p     equ     "|"                     ; prawa boczna
r_pg    equ     "\"                     ; prawa górna (narożnik)
r_pd    equ     "/"                     ; prawa dolna

r_g     equ     "="                     ; górna
r_d     equ     r_g                     ; dolna

r_l     equ     r_p                     ; lewa boczna
r_lg    equ     "/"                     ; lewa górna
r_ld    equ     "\"                     ; lewa dolna

spacja  equ     20h


        push    bx
        push    cx

        mov     dl, r_lg
        call    znak                    ; rysujemy lewy górny narożnik

        push    bx
        mov     dl, r_g                 ; będziemy rysować górną krawędź

                                        ;dopóki BX<SI, rysuj górną krawędź
.rysuj_gora:
        inc     bx
        cmp     bx, si
        je      .dalej
        call    znak
        jmp     short .rysuj_gora

.dalej:
```

```
        mov     dl, r_pg
        call    znak                    ; rysujemy prawy górny narożnik
        pop     bx
        push    bx

                                        ; rysujemy środek
                                        ;dopóki CX<BP, rysuj wnętrze ramki
.rysuj_srodek:
        inc     cx
        cmp     cx, bp
        je      .ostatni

        mov     dl, r_l
        call    znak                    ; zaczynamy od lewego brzegu ramki

        push    bx
        mov     dl, spacja              ; w środku będą spacje
.rysuj_srodek2:
        inc     bx
        cmp     bx, si          ; dopóki BX<SI, rysuj wnętrze (spacje)
        je      .dalej2
        call    znak
        jmp     short .rysuj_srodek2

.dalej2:
        mov     dl, r_p
        call    znak                    ; rysujemy prawy brzeg
        pop     bx

        jmp     short .rysuj_srodek

.ostatni:
        mov     dl, r_ld
        call    znak                    ; rysujemy lewy dolny narożnik
        pop     bx

        mov     dl, r_d         ; będziemy rysować dolną krawędź ramki
.rysuj_dol:
        inc     bx
        cmp     bx, si                  ;dopóki BX<SI, rysuj dolną krawędź
        je      .dalej3
        call    znak
        jmp     short .rysuj_dol

.dalej3:
        mov     dl, r_pd
        call    znak                    ; rysujemy prawy dolny narożnik

        pop     cx
        pop     bx

        ret


znak:

;  AH = atrybut znaku (kolor)
;  AL = kolor tła
;  BX = kolumna znaku
;  CX = wiersz znaku
;  DL = znak
```

```
        push    eax
        push    ebx
        push    ecx
        push    edx

        push    ax
        mov     dh, 10
        shr     ax, 8                   ; AX = kolor znaku
        div     dh                      ; AL = AL/10, AH = AL mod 10
        add     ax, "00"                ; do ilorazu i reszty dodajemy
                                        ; kod ASCII cyfry zero
        mov     [fg], ax                ; do [fg] zapisujemy numer
                                        ; koloru znaku

        pop     ax
        and     ax, 0FFh                ; AX = kolor tła
        div     dh                      ; dzielimy przez 10
        add     ax, "00"
        mov     [bg], ax

        mov     ax, bx                  ; AX = kolumna znaku
        and     ax, 0FFh
        div     dh                      ; dzielimy przez 10
        add     ax, "00"
        mov     [kolumna], ax

        mov     ax, cx                  ; AX = wiersz znaku
        and     ax, 0FFh
        div     dh                      ; dzielimy przez 10
        add     ax, "00"
        mov     [wiersz], ax

        mov     [znaczek], dl           ; zapisujemy, jaki znak
                                        ; mamy wyświetlić


        mov     eax, 4
        mov     ebx, 1
        mov     ecx, pozycja
        mov     edx, napis_dl
        int     80h                     ; wyświetlamy napis wraz z
                                        ; przejściem na odpowiednią pozycję

        pop     edx
        pop     ecx
        pop     ebx
        pop     eax

        ret


section .data

ESC             equ     1Bh

pozycja         db      ESC, "["        ; sekwencja zmiany pozycji kursora
wiersz          db      "00;"
kolumna         db      "00H"
napis           db      ESC, "["        ; sekwencja zmiany koloru
atr             db      "0;"
fg              db      "00;"
bg              db      "00m"
```

```
znaczek         db      "x"             ; znak, który będziemy wyświetlać
napis_dl        equ     $ - pozycja

czysc           db      ESC, "[2J"      ; sekwencja czyszcząca cały ekran
czysc_dl        equ     $ - czysc

nwln            db      10
```

Pisanie z wykorzystaniem urządzeń znakowych /dev/vcsaN

Innym sposobem na poruszanie się po ekranie jest zapis do specjalnych urządzeń znakowych - plików /dev/vcsaN (możliwe, że potrzebne będą uprawnienia root'a).

Na stronach podręcznika man vcsa (a konkretnie to w przykładowym programie) widać, że format tych plików jest dość prosty - na początku są 4 bajty, odpowiadające: liczbie wierszy, liczbie kolumn (bo przecież mogą być różne rozdzielczości) oraz pozycji x i y kursora. Potem idą kolejno znaki widoczne na ekranie (od lewego górnego rogu wzdłuż wierszy) i ich atrybuty. Atrybuty te są takie same, jak w kursie dla DOSa i podobnie jak tam, starsze 4 bity oznaczają kolor tła, a młodsze - kolor znaku.

Teraz widzicie, że to nic trudnego - wystarczy otworzyć plik, odczytać wymiary ekranu i zapisywać odpowiednie bajty na odpowiednich pozycjach (używając funkcji poruszania się po pliku lub, po zmapowaniu pliku do pamięci, po prostu pisać po pamięci).
Oto przykładowy program:
(przeskocz program z vcsa)

```
; Program bezpośrednio zapisujący do pliku konsoli
;
; Autor: Bogdan D., bogdandr MAŁPKA op KROPKA pl
;
; kompilacja:
;
; nasm -O999 -f elf -o konsola.o konsola.asm
; ld -s -o konsola konsola.o


%idefine        sys_exit                1
%idefine        sys_read                3
%idefine        sys_write               4
%idefine        sys_open                5
%idefine        sys_close               6
%idefine        sys_lseek               19
%define         SEEK_SET                0
%define         O_RDWR                  02o

; pozycja, pod którą coć wyświetlimy
%define         nasz_wiersz             10
%define         nasza_kolumna           10

section .text

global _start
```

```
_start:
        mov     eax, sys_open           ; otwieranie pliku
        mov     ebx, plik               ; nazwa pliku
        mov     ecx, O_RDWR             ; odczyt i zapis
        mov     edx, 600q               ; odczyt i zapis dla użytkownika
        int     80h                     ; otwieramy plik

        cmp     eax, 0
        jl      .koniec

        mov     ebx, eax                ; uchwyt do pliku

        mov     eax, sys_read           ; czytanie z pliku (najpierw
                                        ; atrybuty konsoli)
        mov     ecx, konsola            ; dokąd czytać
        mov     edx, 4                  ; ile czytać
        int     80h


        mov     eax, sys_lseek          ; przejście na właściwa pozycję

        movzx   ecx, byte [l_kolumn]
        imul    ecx, nasz_wiersz
        add     ecx, nasza_kolumna      ;ECX=wiersz*długość wiersza+kolumna

        shl     ecx, 1                  ; ECX *= 2, bo na ekranie są: bajt
                                        ; znaku i bajt atrybutu
        add     ecx, 4                  ; +4, bo będziemy szli
                                        ; od początku pliku

        mov     edx, SEEK_SET           ; od początku pliku
        int     80h

        mov     eax, sys_write          ; pisanie do pliku
        mov     ecx, znak               ; co zapisać
        mov     edx, 2                  ; ile zapisać
        int     80h

        mov     eax, sys_close          ; zamknięcie pliku
        int     80h

        xor     eax, eax                ; EAX = 0 = bez błędu

.koniec:
        mov     ebx, eax
        mov     eax, sys_exit
        int     80h                     ;wyjście z kodem zero lub z błędem,
                                        ; który był przy otwarciu pliku


section .data

plik    db      "/dev/vcsa1", 0 ; plik 1-szej konsoli tekstowej

                                        ; atrybuty czytanej konsoli:
konsola:
l_wierszy       db      0
l_kolumn        db      0
kursor_x        db      0
kursor_y        db      0
```

```
                                ; znak z atrybutem, który wyświetlimy:
znak            db      "*"
atrybut         db      43h             ; błękit na czerwonym
```

Pisanie z wykorzystaniem mapowania pamięci

Jeszcze jednym sposobem na pisanie po ekranie jest zapisywanie bezpośrednio do pamięci trybu tekstowego. Pamięć ta znajduje się w segmencie B800, co odpowiada liniowemu adresowi B8000, licząc od adresu 0. Oczywiście system, ze względów bezpieczeństwa, nie pozwoli nam bezpośrednio pisać pod ten adres, więc musimy sobie poradzić w inny sposób. Sposób ten polega na otwarciu specjalnego pliku urządzenia, który symbolizuje całą pamięć w komputerze - /dev/mem. Na większości systemów otwarcie tego pliku wymaga uprawnień administratora.
Po otwarciu pliku mamy dwie możliwości. Pierwsza to poruszać się po nim funkcjami do zmiany pozycji w pliku, oraz odczytywać i zapisywać funkcjami odczytu i zapisu danych z i do pliku. Może to być powolne, ale sposób jest. Druga możliwość to zmapować plik do pamięci, po czym korzystać z niego jak ze zwykłej tablicy. Tę możliwość opiszę teraz szczegółowo.
Otwieranie pliku odbywa się za pomocą tradycyjnego wywołania:

```
mov     eax, 5          ; sys_open
mov     ebx, pamiec     ; adres nazwy pliku "/dev/mem", 0
mov     ecx, 2          ; O_RDWR, zapis i odczyt
mov     edx, 666o       ; pełne prawa
int     80h
...
pamiec          db      "/dev/mem", 0
```

Drugim krokiem jest zmapowanie naszego otwartego pliku do pamięci. Odbywa się to za pomocą funkcji systemowej sys_mmap2. Przyjmuje ona 6 argumentów:

1. EBX = adres, pod jaki chcielibyśmy zmapować plik. Najlepiej podać zero, wtedy system sam wybierze dogodny adres
2. ECX = długość mapowanego obszaru pliku, w bajtach. Podamy to 100000h, by na pewno objąć obszar zaczynający się B8000 i długości 4000 bajtów (tyle, ile trzeba na jeden ekran w trybie tekstowym, na znaki i ich atrybuty)
3. EDX = tryb dostępu do zmapowanej pamięci. Jeśli chcemy odczyt i zapis, podamy tutaj PROT_READ=1 + PROT_WRITE=2
4. ESI = tryb współdzielenia zmapowanej pamięci. Podamy tu MAP_SHARED=1 (współdzielona, nie prywatna)
5. EDI = deskryptor otwartego pliku, który chcemy zmapować
6. EBP = adres początkowy w pliku, od którego mapować. Adres ten jest podawany w jednostkach strony systemowej, której wielkość może być różna na różnych systemach. Najłatwiej podać tu zero, a do adresów dodawać potem B8000

Po pomyślnym wykonaniu, system zwróci nam w EAX adres zmapowanego obszaru pamięci, którego możemy używać (w przypadku błędu otrzymujemy wartość od -4096 do -1 włącznie). Przykładowe wywołanie wygląda więc tak:

```
mov     eax, 192                ; sys_mmap2
xor     ebx, ebx                ; jądro wybierze adres
mov     ecx, 100000h            ; długość mapowanego obszaru
mov     edx, 3                  ; PROT_READ | PROT_WRITE, możliwość
                                ; zapisu i odczytu
mov     esi, 1                  ; MAP_SHARED - tryb współdzielenia
mov     edi, [deskryptor]       ; deskryptor pliku pamięci, otrzymany
                                ; z sys_open w poprzednim kroku
mov     ebp, 0                  ; adres początkowy w pliku
int     80h
```

Teraz wystarczy już korzystać z otrzymanego wskaźnika, na przykład:

```
mov     byte [eax+0b8000h], 'A'
```

Ekran w trybie tekstowym składa się z 80*25=2000 znaków, a każdy z nich ma po sobie bajt argumentu, mówiący o kolorze znaku i tła:
b8000 - znak 1, w lewym górnym rogu
b8001 - atrybut znaku 1
b8002 - znak 2, znajdujący się o 1 pozycję w prawo od znaku 1
b8003 - atrybut znaku 2
....

Czym zaś jest atrybut?
Jest to bajt mówiący o kolorze danego znaku i kolorze tła dla tego znaku. Bity w tym bajcie oznaczają:
3-0 - kolor znaku (16 możliwości)
6-4 - kolor tła (8 możliwości)
7 - miganie znaku (jeśli nie działa, to oznacza, że mamy 16 kolorów tła zamiast 8)

Jeszcze tylko wystarczy omówić kolory odpowiadające poszczególnym bitom i możemy coś pisać.
Oto te kolory:
Czarny - 0, niebieski - 1, zielony - 2, błękitny - 3, czerwony - 4, różowy - 5, brązowy - 6, jasnoszary (ten standardowy) - 7, ciemnoszary - 8, jasnoniebieski - 9, jasnozielony - 10, jasnobłękitny - 11, jasnoczerwony - 12, jasnoróżowy - 13, żółty - 14, biały - 15.

Zmiany, które zapiszemy w pamięci, mogą jednak nie od razu pojawić się w pliku (czyli na ekranie w tym przypadku). Aby wymusić fizyczny zapis danych, korzysta się z funkcji sys_msync. Przyjmuje ona 3 argumenty:

1. EBX = adres początku danych do synchronizacji
2. ECX = ilość bajtów do zsynchronizowania
3. EDX = 0 lub zORowane flagi: MS_ASYNC=1 (wykonaj asynchronicznie), MS_INVALIDATE=2 (unieważnij obszar po zapisaniu), MS_SYNC (wykonaj synchronicznie)

Przykładowe wywołanie wygląda więc tak:

```
mov     eax, 144                ; sys_msync
mov     ebx, 0b8000h            ; adres startowy
mov     ecx, 4000               ; ile zsynchronizować
mov     edx, 0                  ; flagi
int     80h
```

Po zakończeniu pracy z plikiem, możemy go odmapować:

```
mov     eax, 91                 ; sys_munmap
mov     ebx, [wskaznik]         ; wskaźnik otrzymany z sys_mmap2
mov     ecx, 100000h            ; ilość bajtów
int     80h
```

i zamknąć:

```
mov     eax, 6                  ; sys_close
mov     ebx, [deskryptor]       ; deskryptor pliku "/dev/mem"
int     80h
```

Jak widać, mapowanie plików do pamięci jest wygodne, gdyż nie trzeba ciągle skakać po pliku funkcją sys_lseek i wykonywać kosztownych czasowo wywołań innych funkcji systemowych. Warto więc się z tym zaznajomić. Należy jednak pamiętać, że nie wszystkie pliki czy urządzenia dają się zmapować do pamięci - nie należy wtedy zamykać swojego programu z błędem, lecz korzystać z tradycyjnego interfejsu funkcji plikowych.

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

14.02.2010

Pisanie programów rezydentnych pod Linuksem

W Linuksie są co najmniej dwa sposoby na uruchomienie programu w tle:

1. dodanie znaczka ampersand (&) do uruchomienia programu, np. program param1 param2 param3 &.
2. skorzystanie z programu screen, np. screen -m -d program param1 param2.

Ale nie jesteśmy programistami po to, by liczyć, że nasz program zostanie właśnie tak uruchomiony. Teraz pokażę, jak samemu zadbać o działanie swojego programu w tle. Najpierw posłuży nam do tego funkcja daemon (patrz: man 3 daemon) z biblioteki języka C. Dlatego sposób pisania programu będzie troszkę inny niż zwykle:

1. zamiast LD do łączenia programu, skorzystamy z GCC (który widząc rozszerzenie .o naszego pliku, połączy go odpowiednio z biblioteką języka C bez kompilowania)
2. skoro korzystamy z GCC i biblioteki C (która ma już własny symbol _start), to nasz kod będzie się zaczynał etykietą main (taką samą, jak programy w języku C)
3. funkcja daemon musi być zadeklarowana jako zewnętrzna (extern)

Aby było widać, że nasz demon (odpowiednik TSR w Linuksie) rzeczywiście działa, umieścimy w nim pętlę co jakiś czas wyświetlającą jakiś napis. W celu odmierzania przerwy skorzystamy z funkcji systemowej sys_nanosleep (numer 162).
Jak widać ze strony podręcznika, funkcja daemon przyjmuje 2 argumenty w postaci liczb całkowitych (DWORD):

- pierwszy argument (ostatni wkładany na stos) mówi o tym, czy nie zmieniać katalogu pracy na katalog główny /. My nie chcemy zmieniać, więc wstawimy wartość 1.
- drugi argument (pierwszy wkładany na stos) mówi o tym, czy nie zamykać strumieni wejścia i wyjścia. My nie chcemy zamykać, więc wstawimy wartość 1.

Po omówieniu tego wszystkiego, przejdźmy wreszcie do przykładowego programu. Jego zadaniem jest przejście w tryb demona i wyświetlanie co 5 sekund wskazanego tekstu w nieskończoność. Działanie będzie więc widoczne na terminalu, z którego uruchamiacie program. Sam program można będzie oczywiście zobaczyć na liście działających procesów (komenda `ps -A`). Jedynym sposobem na zakończenie programu jest zabicie go poleceniem kill.

A oto kod dla NASMa:
(przeskocz program)

```
; Program przechodzący w tryb demona systemowego
;
; autor: Bogdan D., bogdandr (at) op.pl
;
; kompilacja:
; nasm -f elf -o demon.o demon.asm
; gcc -o demon demon.o


extern daemon                   ; deklaracja funkcji zewnętrznej

section .text                   ; początek sekcji kodu
```

```
global main                     ;symbol "main" musi być globalny dla GCC

main:
        push    dword 1         ; drugi argument
        push    dword 1         ; pierwszy argument
        call    daemon          ; uruchomienie funkcji daemon
        add     esp, 8          ; usunięcie argumentów ze stosu

                        ; przerwa między kolejnymi napisami
                        ; będzie trwać 5 sekund i 0 nanosekund:
        mov     dword [t1+timespec.tv_nsec], 0
        mov     dword [t1+timespec.tv_sec], 5

.petla:
        mov     eax, 4          ; funkcja zapisywania do pliku
        mov     ebx, 1          ; standardowe wyjście
        mov     ecx, napis      ; co wypisać
        mov     edx, napis_dl   ; długość napisu
        int     80h

        mov     eax, 162        ; funkcja sys_nanosleep
        mov     ebx, t1         ; tyle czekać
        mov     ecx, 0  ; ewentualny adres drugiej struktury timespec
        int     80h             ;  robimy przerwę...

        jmp     .petla          ; i od nowa....

                                ; poniższy kod nie będzie wykonany
        mov     eax, 1
        xor     ebx, ebx
        int     80h             ; wyjście z programu

section .data

napis           db      "Tu Twoj demon mowi.", 10
napis_dl        equ     $ - napis

struc timespec                  ; definicja struktury timespec
                                ; (tylko jako typ danych)
        .tv_sec:        resd 1
        .tv_nsec:       resd 1
endstruc

t1 istruc timespec              ; tworzymy zmienną t1 jako całą
                                ; strukturę timespec
```

Kompilacja i łączenie odbywa się tak:

```
        nasm -f elf -o demon.o demon.asm
        gcc -o demon demon.o
```

O jednej rzeczy należy wspomnieć: sam fakt, że program jest demonem *NIE* musi oznaczać, że działa na prawach administratora (i całe zabezpieczenie systemu jest do niczego).

Programy rezydentne z wykorzystaniem int 80h

Funkcja daemon jest co prawda z biblioteki C, ale można ją przerobić na kod korzystający wyłącznie z

przerwania int 80h. Sam kod funkcji jest w pliku misc/daemon.c w źródłach biblioteki glibc. Nie jest on za długi i dość łatwo można go przerobić na takie oto makro:
(przeskocz makro)

```
%macro daemon 2
        ; pierwszy parametr: nochdir - czy nie zmieniać katalogu na główny?
        ; drugi parametr: noclose - czy nie zamykać stdin i stdout?

        mov     eax, 2
        int     80h             ; sys_fork

        cmp     eax, 0
        jl      %%koniec        ; EAX < 0 oznacza błąd

        test    eax, eax
        jz      %%dalej         ; EAX = 0 w procesie potomnym
                                ; EAX > 0 w procesie rodzica

        mov     eax, 1
        xor     ebx, ebx
        int     80h             ; sys_exit - rodzic kończy pracę.

 %%glowny:      db      "/", 0
 %%devnull:     db      "/dev/null", 0

 %%dalej:
        mov     eax, 66         ; sys_setsid
        int     80h             ; tworzymy nową sesję i ustawiamy GID

        cmp     eax, 0
        jl      %%koniec        ; EAX < 0 oznacza błąd

        %if %1 = 0

                mov     eax, 12         ; sys_chdir
                mov     ebx, %%glowny
                int     80h             ; zmieniamy katalog na główny
        %endif

        %if %2 = 0

                ; otwieramy /dev/null:
                mov     eax, 5
                mov     ebx, %%devnull
                mov     ecx, 2
                mov     edx, 0
                int     80h

                cmp     eax, 0
                jl      %%koniec        ; EAX < 0 oznacza błąd

                mov     ebx, eax        ; EBX = deskryptor /dev/null

                ;duplikujemy deskryptory standardowego wejścia, wyjścia i
                ;wyjścia błędów do deskryptora otwartego /dev/null, po czym
                ; ten /dev/null zamkamy

                mov     eax, 63
                mov     ecx, 0          ; wejście
                int     80h
```

```
                mov     eax, 63
                mov     ecx, 1          ; wyjście
                int     80h

                mov     eax, 63
                mov     ecx, 2          ; wyjście błędów
                int     80h

                mov     eax, 6
                int     80h             ; zamykamy /dev/null

        %endif
 %%koniec:

%endmacro

; użycie:
        daemon 1, 1     ; bez żadnych PUSH ani ADD ESP
```

Teraz można już powrócić do starego schematu programu, gdzie symbolem startowym był _start, a łączenie odbywało się za pomocą LD, a nie GCC z biblioteką C.

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

14.02.2010

Uruchamianie innych programów pod Linuksem

Czasem zdarza się, że z poziomu naszego własnego programu musimy uruchomić jakiś inny program lub polecenie systemowe. Służy do tego funkcja systemowa sys_execve (numer 11). Jej argumenty to kolejno:

- w EBX - adres nazwy programu do uruchomienia (ze ścieżką). Nazwa powinna być zakończona bajtem zerowym. Można uruchomić skrypt.
- w ECX - adres listy adresów argumentów dla uruchamianego programu. Lista powinna kończyć się DWORDem zerowym.
- w EDX - adres listy adresów zmiennych środowiskowych dla uruchamianego programu. Lista powinna kończyć się DWORDem zerowym.

Spróbujmy więc napisać jakiś prosty przykład - wyświetlenie napisu za pomocą programu echo.
(przeskocz program)

```
; Uruchamianie innych programów w asemblerze pod Linuksem
;
; Autor: Bogdan D., bogdandr (at) op.pl
;
; kompilacja:
;
; nasm -f elf -o exec_linux.o exec_linux.asm
; ld  -o exec_linux exec_linux.o


section .text
global  _start

_start:

        mov     eax, 11                 ; numer funkcji sys_execve
        mov     ebx, komenda            ; plik do uruchomienia
        mov     ecx, argumenty          ; adres tablicy argumentów
        mov     edx, srodowisko         ; adres tablicy środowiska
        int     80h

        mov     eax, 4
        mov     ebx, 1
        mov     ecx, info
        mov     edx, info_dl
        int     80h                     ; wyświetlenie napisu

        mov     eax, 1
        xor     ebx, ebx
        int     80h                     ; wyjście z programu


section .data

komenda         db      "/bin/echo", 0  ; program do uruchomienia
info            db      "Wykonalem program.", 10 ; napis do wyświetlenia
info_dl         equ     $ - info

argumenty       dd      komenda         ; argv[0] to nazwa programu
                dd      arg1            ; argv[1]
```

```
                dd      0               ; koniec argumentów

arg1            db      "Czesc!", 0     ; argument pierwszy

srodowisko      dd      home            ; jedna zmienna środowiskowa
                dd      0               ; koniec zmiennych środowiskowych

home            db      "HOME=/home/bogdan", 0 ; przykładowa zmienna
                                               ; środowiskowa $HOME
```

Jedna rzecz od razu powinna rzucić się w oczy: napis Wykonałem program *nie jest wyświetlany*. Dzieje się tak dlatego, że jeśli funkcja sys_execve wykonała się bez błędów, to ... nie powróci do naszego programu (tak, jak jest to napisane na stronie podręcznika: man execve). Duża wada, ale można to łatwo przeskoczyć, stosując wątki lub funkcje typu sys_fork lub sys_clone, w celu uruchomienia osobnego wątku lub procesu, który potem wykona sys_execve.

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Pisanie programów wielowątkowych pod Linuksem

Asembler, jak wszystkie inne strukturalne języki programowania pozwala pisać programy, w których ścieżka wykonywanych instrukcji jest tylko jedna. Mogą być rozwidlenia i pętle, ale zawsze wykonuje się tylko jedna rzecz na raz.
Wątki pozwalają na uruchomienie wielu niezależnych ścieżek, które będą wykonywane równolegle. Daje to duże możliwości programom, które wykonują kilka czynności na raz (np. czytanie z jednego pliku i zapisywanie przetworzonych danych do drugiego). Zysk jest też w programach sieciowych, a zwłaszcza serwerach. Po dodaniu obsługi wątków możliwe jest połączenie więcej niż jednego klienta w danej chwili.
Ale przejdźmy wreszcie do szczegółów.

Najpierw omówię trzy funkcje z biblioteki języka C (ściśle mówiąc, z biblioteki pthreads), które pozwolą nam zarządzać wątkami.

1. pthread_create - tworzenie nowego wątku.
   Funkcja ta przyjmuje 4 argumenty. Od lewej (ostatni wkładany na stos) są to:
   - adres zmiennej typu DWORD, która otrzyma identyfikator nowego wątku.
   - atrybuty nowego wątku, jeśli chcemy coś specjalnego. Zero oznacza domyślne argumenty.
   - adres funkcji wątku. Funkcja ta otrzyma na stosie adres dodatkowych danych, które można przekazać do wątku.
   - adres dodatkowych danych, które chcemy przekazać do wątku.

2. pthread_exit - zakończenie bieżącego wątku
   Funkcja ta kończy bieżący wątek. Wartość podana jako jedyny jej argument (adres danych) może być wykorzystana przez wątki podłączone (pthread_join) do tego wątku. Po zakończeniu wszystkich wątków, program kończy działanie z kodem 0.

3. pthread_yield - oddanie czasu procesora innym wątkom lub procesom
   Oczywiście, system operacyjny sam też przydziela czas procesora poszczególnym wątkom, ale wywołując tą funkcję możemy powiedzieć, by skrócił czas przeznaczony dla tego wątku i dał go innym. Przydaje się, gdy bieżący wątek chwilowo skończył pracę (np. zabrakło danych itp.). Funkcja nie przyjmuje żadnych argumentów.

Poniżej przedstawiam króciutki program, który pokaże, jak to wszystko działa. Program ma jeden raz wyświetlić napis pierwszy w funkcji głównej i 5 razy napis drugi w funkcji wątku.
(przeskocz program)

```
; Przykładowy program wielowątkowy w asemblerze
;
; Autor: Bogdan D., bogdandr (at) op.pl
;
; kompilacja:
; nasm -O999 -f elf -o watki.o watki.asm
; gcc -o watki watki.o -lpthread

section .text
global  main
```

```
; deklaracje funkcji zewnętrznych
extern  pthread_create
extern  pthread_exit

main:

        mov     eax, 4
        mov     ebx, 1
        mov     ecx, napis1
        mov     edx, napis1_dl
        int     80h                     ; wyświetlamy napis pierwszy

        push    dword 0                 ; dodatkowe dane
        push    dword watek             ; adres funkcji do uruchomienia
        push    dword 0                 ; atrybuty
        push    dword id_watku          ; gdzie zapisać ID
        call    pthread_create          ; utworzenie nowego wątku

; Nie należy wychodzić z programu funkcją sys_exit (EAX=1), gdyż
; zakończyłoby to wszystkie wątki programu. Zamiast tego, zamykamy tylko
; wątek główny.
        push    dword 0
        call    pthread_exit            ; zakończenie bieżącego wątku

watek:

        mov     dword [t1+timespec.tv_nsec], 0
        mov     dword [t1+timespec.tv_sec], 5           ; 5 sekund

        mov     esi, 5                  ; napis drugi wyświetlimy 5 razy
.petla:
        mov     eax, 162                ; sys_nanosleep
        mov     ebx, t1                 ; adres struktury mówiącej,
                                        ; ile chcemy czekać
        mov     ecx, 0
        int     80h                     ; robimy przerwę...

        mov     eax, 4
        mov     ebx, 1
        mov     ecx, napis2
        mov     edx, napis2_dl
        int     80h                     ; wyświetl napis drugi

        dec     esi
        jnz     .petla                  ; wykonuj pętlę, jeśli ESI != 0

        push    dword 0
        call    pthread_exit            ; zakończenie bieżącego wątku

section .data

napis1          db      "Funkcja glowna.", 10
napis1_dl       equ     $ - napis1

napis2          db      "Watek.", 10
napis2_dl       equ     $ - napis2

struc timespec
        .tv_sec:        resd 1
        .tv_nsec:       resd 1
endstruc
```

```
t1              istruc timespec

id_watku        dd      0       ; zmienna, która otrzyma ID nowego wątku
```

Ale wątki w programie to nie tylko same zyski. Największym problemem w programach wielowątkowych jest *synchronizacja wątków*.
Po co synchronizować? Po to, żeby program nie sprawiał problemów, gdy dwa lub więcej wątków odczytuje i zapisuje tą samą zmienną globalną (np. bufor danych).
Co zrobić, by np. wątek czytający przetwarzał dane dopiero wtedy, gdy inny wątek dostarczy te dane? Możliwości jest kilka:

- flaga - zmienna globalna.
  Na przykład ustalmy, że jeśli flaga jest równa zero, to bufor może być dowolnie używany (do zapisu i odczytu). Jeśli flaga jest równa np. jeden, to nie wolno wykonywać operacji na buforze (bo inny wątek już to robi) - należy poczekać, aż flaga będzie równa zero.
  Zaletą tego rozwiązania jest prostota jego utworzenia. Popatrzcie:

```
flaga   db      0
...
watek:
...
sprawdz_flage:
        cmp     byte [flaga], 1
        je      sprawdz_flage
        mov     byte [flaga], 1
...     ; tutaj nasze operacje
        mov     byte [flaga], 0
```

- mutex - poczytajcie pl.wikipedia.org/wiki/Mutex
- semafor ustawiający wątki w kolejkę do danego zasobu. Poczytajcie pl.wikipedia.org/wiki/Semafor_(informatyka)

Jak widać, pisanie programów wielowątkowych nie jest takie trudne, warto więc się tego nauczyć. Tym bardziej, że zyski są większe (napisanie po jednej funkcji na każde oddzielne zadanie), niż wysiłek (synchronizacja).

Wielowątkowość z przerwaniem 80h

Oczywiście, aby pisać programy wielowątkowe, nie musicie korzystać z żadnej biblioteki. Odpowiednie mechanizmy posiada sam interfejs systemu - przerwanie int 80h.
Skorzystam tutaj z funkcji sys_fork (numer 2). Jej jedynym argumentem jest adres struktury zawierającej wartości rejestrów dla nowego procesu, ale ten argument jest opcjonalny i może być zerem. Funkcja fork zwraca wartość mniejszą od zera, gdy wystąpił błąd, zwraca zero w procesie potomnym, zaś wartość większą od zera (PID nowego procesu) - w procesie rodzica. Proces potomny zaczyna działanie tuż po wywołaniu funkcji fork, czyli rodzic po wykonaniu funkcji fork i potomek zaczynają wykonywać dokładnie te same instrukcje. Procesy te można skierować na różne ścieżki, sprawdzając wartość zwróconą przez fork w EAX. Oto krótki przykład w składni FASMa:

```
format ELF executable
entry _start
segment executable
```

```
_start:
        mov     eax, 2          ; funkcja fork
        xor     ebx, ebx
        int     80h             ; wywołanie

        cmp     eax, 0
        jl      .koniec         ; EAX < 0 oznacza błąd

        ; poniższe instrukcje wykona zarówno rodzic, jak i potomek:

        cmp     eax, 0
        jg      .rodzic         ; EAX > 0 oznacza, że jesteśmy w
                                ; procesie rodzica

        ; tutaj ani EAX < 0, ani EAX > 0, więc EAX=0, czyli
        ; jesteśmy w procesie potomka
        ; kod poniżej (wyświetlenie i czekanie) wykona tylko potomek

        mov     dword [t1.tv_nsec], 0
        mov     dword [t1.tv_sec], 5    ; tyle sekund przerwy będziemy robić
                                        ; między wyświetlaniem napisów

.petla:
        mov     eax, 4          ; funkcja zapisywania do pliku
        mov     ebx, 1          ; standardowe wyjście
        mov     ecx, napis2     ; co wypisać
        mov     edx, napis2_dl  ; długość napisu
        int     80h

        mov     eax, 162        ; funkcja sys_nanosleep
        mov     ebx, t1         ; tyle czekać
        mov     ecx, 0          ;ewentualny adres drugiej struktury timespec
        int     80h             ;  robimy przerwę...

        jmp     .petla          ; i od nowa....

        ; kod poniżej (wyświetlenie i wyjście) wykona tylko rodzic
.rodzic:

        mov     eax, 4          ; funkcja zapisywania do pliku
        mov     ebx, 1          ; standardowe wyjście
        mov     ecx, napis1     ; co wypisać
        mov     edx, napis1_dl  ; długość napisu
        int     80h

.koniec:
        mov     eax, 1          ; funkcja wyjścia z programu
        xor     ebx, ebx
        int     80h

segment readable writeable

napis1          db      "Rodzic", 10
napis1_dl       =       $ - napis1
napis2          db      "Potomek", 10
napis2_dl       =       $ - napis1

struc timespec                  ; definicja struktury timespec
                                ; (tylko jako typ danych)
{
        .tv_sec:        rd 1
        .tv_nsec:       rd 1
```

```
}

t1 timespec             ; tworzymy zmienną t1 jako całą strukturę
```

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Opis funkcji przerwania int 80h: 1-50

Jeśli jakaś funkcja zakończy się błędem, w EAX/RAX zwracana jest wartość ujemna z przedziału od -4096 do -1 włącznie.
Z drugiej strony, opisy funkcji na stronach manuala mówią, że zwracane jest -1, a wartość błędu jest zapisywana do zmiennej errno z biblioteki GLIBC. Dzieje się tak tylko w przypadku, gdy korzystamy z interfejsu języka C (czyli deklarujemy i uruchamiamy zewnętrzne funkcje odpowiadające wywołaniom systemowym i linkujemy nasz program z biblioteką języka C), a nie bezpośrednio z wywołań systemowych (czyli przerwania int 80h).

Najaktualniejsze informacje o funkcjach systemowych można znaleźć zazwyczaj w sekcji 2 (lub 3) manuala, np. man 2 open
Najnowsze wersje stron manuala można znaleźć tu: www.kernel.org/pub/linux/docs/man-pages.

Napis ASCIIZ oznacza łańcuch znaków ASCII zakończony znakiem/bajtem Zerowym.
Jeśli potrzeba, przy każdej funkcji jest odnośnik do opisu argumentów i innych dodatkowych informacji: typów danych, wartości błędów, możliwych wartości parametrów itp.

Podstawowe funkcje przewania 80h: 1-50

Numer/ EAX	x86-64 RAX	Opis	Argumenty	Zwraca
1	60	Wyjście z programu (sys_exit)	EBX/RDI = kod wyjścia (errorlevel)	nie wraca do programu wywołującego
2	57	Uruchomienie nowego procesu (sys_fork)	EBX/RDI = adres struktury pt_regs	EAX=id procesu potomnego (PID) EAX=błąd EAGAIN, ENOMEM
3	0	Czytanie z pliku (sys_read)	EBX/RDI = deskryptor pliku ECX/RSI = adres bufora docelowego EDX/RDX = liczba bajtów do przeczytania	EAX=liczba przeczytanych bajtów EAX = błąd EAGAIN, EBADF, EFAULT, EINTR, EINVAL, EIO, EISDIR
4	1	Zapis do pliku (sys_write)	EBX/RDI = deskryptor pliku ECX/RSI = adres bufora źródłowego EDX/RDX = liczba bajtów do zapisania	EAX=liczba zapisanych bajtów EAX = błąd EAGAIN, EBADF, EFAULT, EINTR, EINVAL, EIO,

				ENOSPC, EPIPE
5	2	Otwarcie pliku (sys_open)	EBX/RDI = adres nazwy pliku ASCIIZ ECX/RSI = bity dostępu EDX/RDX = prawa dostępu / tryb	EAX=deskryptor pliku EAX = błąd EACCES, EEXIST, EFAULT, EISDIR, ELOOP, EMFILE, ENAMETOOLONG, ENFILE, ENOENT, ENODEV, ENODIR, ENOMEM, ENOSPC, ENXIO, EROFS, ETXTBSY
6	3	Zamknięcie pliku (sys_close)	EBX/RDI = deskryptor pliku	EAX = 0 EAX = błąd EBADF, EINTR, EIO
7	-	Czekaj na zmianę stanu innego procesu (sys_waitpid)	EBX/RDI = id procesu / specyfikacja ECX/RSI = NULL lub adres zmiennej DWORD, która otrzyma status EDX/RDX = opcje	EAX=PID zakończonego procesu [ECX/RSI] = (jeśli podano adres bufora) stan wyjścia procesu EAX = błąd ECHILD, EINVAL, ERESTARTSYS
8	85	Utworzenie pliku (sys_creat, nie create!)	EBX/RDI = adres nazwy pliku ASCIIZ ECX/RSI = prawa dostępu / tryb	EAX=deskryptor pliku EAX = błąd EACCES, EEXIST, EFAULT, EISDIR, ELOOP, EMFILE, ENAMETOOLONG, ENFILE, ENOENT, ENODEV, ENODIR, ENOMEM, ENOSPC, ENXIO, EROFS, ETXTBSY
9	86	Utworzenie twardego dowiązania do pliku (sys_link)	EBX/RDI = adres nazwy istniejącego pliku ASCIIZ ECX/RSI = adres nazwy nowego pliku ASCIIZ	EAX = 0 EAX=błąd EACCES, EIO, EPERM, EEXIST, EFAULT, ELOOP, EMLINK,

				ENAMETOOLONG, ENOENT, ENOMEM, ENOSPC, ENOTDIR, EROFS, EXDEV
10	87	Usunięcie pliku (sys_unlink)	EBX/RDI = adres nazwy pliku ASCIIZ	EAX = 0 EAX=błąd EACCES, EFAULT, EIO, EISDIR, ELOOP, ENOENT, ENAMETOOLONG, ENOMEM, ENOTDIR, EPERM, EROFS
11	59	Uruchomienie innego programu (sys_execve)	EBX/RDI=adres nazwy (ze ścieżką) programu ASCIIZ ECX/RSI = adres zakończonej dwordem 0 listy adresów argumentów uruchamianego programu ASCIIZ EDX/RDX = adres zakończonej dwordem 0 listy adresów zmiennych środowiska dla uruchamianego programu ASCIIZ	nie wraca do programu wywołującego EAX = błąd E2BIG, EACCES, EINVAL, EOIO, EISDIR, ELIBBAD, ELOOP, ENFILE, ENOEXEC, ENOENT, ENOMEM, ENOTDIR, EFAULT, ENAMETOOLONG, EPERM, ETXTBUSY
12	80	Zmiana katalogu (sys_chdir)	EBX/RDI = adres nazwy nowego katalogu ASCIIZ	EAX = 0 EAX = błąd EACCES, EBADF, EFAULT, EIO, ELOOP, ENAMETOOLONG, ENOENT, ENOMEM, ENOTDIR
13	201	Pobierz czas (sys_time)	EBX/RDI = NULL lub adres bufora, który otrzyma kopię wyniku	EAX = ilość sekund od 1 Stycznia 1970 minus 1 EAX = błąd EFAULT
14	133	Utworzenie spliku specjalnego (sys_mknod)	EBX/RDI = adres ścieżki ASCIIZ ECX/RSI = typ urządzenia OR prawa dostępu EDX/RDX,ESI/R10 - wynik działania makra makedev	EAX = 0 EAX = błąd EACCES, EEXIST, EFAULT, EINVAL, ELOOP, ENAMETOOLONG, ENOENT, ENOMEM, ENOSPC, ENOTDIR,

				EPERM, EROFS
15	90	Zmiana uprawnień (sys_chmod)	EBX/RDI = adres nazwy pliku/katalogu ASCIIZ ECX/RSI = nowe prawa dostępu	EAX = 0 EAX = błąd EACCES, EBADF, EFAULT, EIO, ELOOP, ENAMETOOLONG, ENOENT, ENOMEM, ENOTDIR, EPERM, EROFS
16	94	Zmiana właściciela (sys_lchown)	EBX/RDI = adres nazwy pliku/katalogu ASCIIZ ECX/RSI = nowy numer użytkownika EDX/RDX = nowy numer grupy	EAX = 0 EAX = błąd EPERM, EROFS, EFAULT, ENAMETOOLONG, ENOENT, ENOMEM, ENOTDIR, EACCES, ELOOP i inne
17	-	Funkcja systemowa sys_break (porzucone)	Istnieje tylko dla zachowania zgodności	EAX = błąd ENOSYS
18	-	Funkcja systemowa sys_oldstat (porzucone)		
19	8	Zmiana bieżącej pozycji w pliku (sys_lseek)	EBX/RDI = deskryptor pliku ECX/RSI = liczba bajtów, o którą chcemy się przesunąć EDX/RDX = odkąd zaczynamy ruch	EAX = nowa pozycja względem początku pliku EAX = błąd EBADF, EINVAL, EISPIPE
20	39	Pobierz identyfikator bieżącego procesu (sys_getpid)	nic	EAX = PID bieżącego procesu
21	165	Montowanie systemu plików (sys_mount)	EBX/RDI = adres nazwy urządzenia/pliku specjalnego ECX/RSI = adres ścieżki do punktu montowania EDX/RDX = adres nazwy systemu plików ESI/R10 = flagi montowania EDI/R8 = adres dodatkowych danych, niezależne od urządzenia	EAX = 0 EAX = błąd - każdy, który może się zdarzyć w systemie plików lub jądrze

22	-	Odmontowanie systemu plików (sys_umount)	EBX/RDI = adres nazwy pliku specjalnego lub katalogu (zamontowanego)	EAX = 0 EAX = błąd - każdy, który może się zdarzyć w systemie plików lub jądrze
23	105	Ustaw identyfikator użytkownika (sys_setuid)	EBX/RDI = nowy UID	EAX = 0 EAX = błąd EPERM
24	102	Pobierz identyfikator użytkownika (sys_getuid)	nic	EAX = numer UID
25	-	Ustaw czas systemowy (sys_stime)	EBX/RDI = nowy czas jako liczba sekund, które upłynęły od 1 Stycznia 1970	EAX = 0 EAX = błąd EPERM
26	101	Śledzenie procesu (sys_ptrace)	EBX/RDI = żądane działanie ECX/RSI = identyfikator PID żądanego procesu EDX/RDX = adres w procesie docelowym ESI/R10 = adres w procesie śledzącym	EAX zależne od działania EAX = błąd EIO, EFAULT, EPERM, ESRCH
27	37	Alarm - wysłanie sygnału SIGALARM (sys_alarm)	EBX/RDI = sekundy	EAX = 0 lub liczba sekund do wykonania poprzednich alarmów
28	-	Funkcja systemowa sys_oldfstat (porzucone)		
29	34	Pauza - śpij aż do otrzymania sygnału (sys_pause)	nic	wraca tylko po sygnale, o ile procedura jego obsługi ma powrót. EAX = EINTR po sygnale
30	132	Zmień czas dostępu do pliku (sys_utime)	EBX/RDI = adres nazwy pliku (ASCIIZ) ECX/RSI = adres struktury utimbuf, NULL gdy chcemy bieżący czas	EAX = 0 EAX = błąd EACCES, ENOENT, EPERM, EROFS
31	-		--nieużywane od 2.0--	zawsze EAX = -1

		Funkcja systemowa sys_stty (porzucone)		
32	-	Funkcja systemowa sys_gtty (porzucone)	--nieużywane od 2.0--	zawsze EAX = -1
33	21	Sprawdź uprawnienia dostępu do pliku (sys_access)	EBX/RDI = adres nazwy pliku ASCIIZ ECX/RSI = prawa dostępu / tryb (wartości R_OK, W_OK, X_OK)	EAX = 0 EAX = błąd - każdy związany z systemem plików i plikami
34	-	Zmień priorytet procesu (sys_nice)	EBX/RDI = liczba, o którą zwiększyć numer priorytetu (czyli zmniejszyć sam priorytet)	EAX = 0 EAX = błąd EPERM
35	-	Pobierz bieżącą datę i czas - sys_ftime (przestarzałe)	--zamiast tego, używaj time, gettimeofday-- EBX/RDI = adres struktury timeb	zawsze EAX = 0
36	162	Zapisz pamięć podręczną na dysku (sys_sync)	nic	EAX zawsze = 0 i nie ma żadnych błędów
37	62	Wyślij sygnał do procesu (sys_kill)	EBX/RDI = numer PID procesu (patrz też specyfikacja) ECX/RSI = numer sygnału	EAX = 0 EAX = błąd EINVAL, EPERM, ESRCH
38	82	Przenieś plik/Zmień nazwę pliku (sys_rename)	EBX/RDI=adres starej nazwy (i ewentualnie ścieżki) ASCIIZ ECX/RSI=adres nowej nazwy (i ewentualnie ścieżki) ASCIIZ	EAX = 0 EAX = błąd EBUSY, EEXIST, EISDIR, ENOTEMPTY, EXDEV (i inne błędy systemu plików)
39	83	Utwórz katalog (sys_mkdir)	EBX/RDI = adres ścieżki/nazwy ASCIIZ ECX/RSI = prawa dostępu / tryb	EAX = 0 EAX = błąd - każdy związany z systemem plików lub prawami dostępu
40	84	Usuń katalog (sys_rmdir)	EBX/RDI = adres ścieżki/nazwy ASCIIZ	EAX = 0 EAX = błąd EACCES, EBUSY, EFAULT, ELOOP,

				ENAMETOOLONG, ENOENT, ENOMEM, ENOTDIR, ENOTEMPTY, EPERM, EROFS
41	32	Zduplikuj deskryptor pliku (sys_dup)	EBX/RDI = stary deskryptor	EAX = nowy deskryptor EAX = błąd EBADF, EMFILE (,EINVAL)
42	22	Utwórz potok (sys_pipe)	EBX/RDI = adres tablicy dwóch DWORDów	EAX = 0 i pod [EBX/RDI]: deskryptor odczytu z potoku fd(0) pod [EBX/RDI], deskryptor zapisu do potoku fd(1) pod [EBX/RDI+4] EAX = błąd EFAULT, EMFILE, ENFILE
43	100	Pobierz czasy procesów (sys_times)	EBX/RDI = adres struktury tms	EAX = liczba taktów zegara EAX = błąd
44	-	Funkcja systemowa sys_prof (porzucone)	niezaimplementowane w jądrach 2.4	zawsze EAX = ENOSYS
45	12	Alokacja i dealokacja pamięci (sys_brk)	EBX/RDI = 0, aby poznać aktualny najwyższy adres sekcji .bss EBX/RDI = (wirtualny) adres nowego wierzchołka .bss, powyżej spodu sekcji danych i poniżej bibliotek	EAX = nowy najwyższy adres EAX = błąd ENOMEM
46	106	Ustaw ID grupy bieżącego procesu (sys_setgid)	EBX/RDI = nowy ID grupy	EAX = 0 EAX = błąd EPERM
47	104	Pobierz ID grupy bieżącego procesu (sys_getgid)	nic	EAX = ID grupy
48	-	Ustaw procedurę obsługi sygnału (sys_signal)	EBX/RDI = numer sygnału ECX/RSI = adres procedury przyjmującej `int` i zwracającą	EAX = adres poprzedniej procedury obsługi

			`void` (nic) lub wartość SIG_IGN=1 (ignoruj sygnał) lub SIG_DFL=0 (resetuj sygnał na domyślne zachowanie)	EAX = błąd SIG_ERR
49	107	Pobierz efektywne ID użytkownika bieżącego procesu (sys_geteuid)	nic	EAX = UID
50	108	Pobierz efektywne ID grupy bieżącego procesu (sys_getegid)	nic	EAX = GID

Kolejna część (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Opis funkcji przerwania int 80h: 51-100

Jeśli jakaś funkcja zakończy się błędem, w EAX/RAX zwracana jest wartość ujemna z przedziału od -4096 do -1 włącznie.
Z drugiej strony, opisy funkcji na stronach manuala mówią, że zwracane jest -1, a wartość błędu jest zapisywana do zmiennej errno z biblioteki GLIBC. Dzieje się tak tylko w przypadku, gdy korzystamy z interfejsu języka C (czyli deklarujemy i uruchamiamy zewnętrzne funkcje odpowiadające wywołaniom systemowym i linkujemy nasz program z biblioteką języka C), a nie bezpośrednio z wywołań systemowych (czyli przerwania int 80h).

Najaktualniejsze informacje o funkcjach systemowych można znaleźć zazwyczaj w sekcji 2 (lub 3) manuala, np. man 2 open
Najnowsze wersje stron manuala można znaleźć tu: www.kernel.org/pub/linux/docs/man-pages.

Napis ASCIIZ oznacza łańcuch znaków ASCII zakończony znakiem/bajtem Zerowym.
Jeśli potrzeba, przy każdej funkcji jest odnośnik do opisu argumentów i innych dodatkowych informacji: typów danych, wartości błędów, możliwych wartości parametrów itp.

Podstawowe funkcje przewania 80h: 51-100

Numer/ EAX	x86-64 RAX	Opis	Argumenty	Zwraca
51	163	Włącz/wyłącz zapisywanie kończonych procesów (sys_acct)	EBX/RDI = adres nazwy pliku, gdzie ma być zapisywana informacja o kończonych procesach lub NULL, gdy chcemy wyłączyć takie zapisywanie.	EAX = 0 EAX = błąd ENOSYS, ENOMEM, EPERM, EACCES, EIO, EUSERS
52	166	Odmontowanie systemu plików 2 (sys_umount2)	EBX/RDI = adres nazwy zamontowanego pliku specjalnego/katalogu ASCIIZ ECX/RSI = flaga = 1, by siłą odmonotwać, inaczej 0	EAX = 0 EAX = błąd - każdy związany z systemem plików
53	-	Funkcja systemowa sys_lock (porzucone)	--nieużywane od 2.0--	zawsze EAX = -1
54	16	Manipulacja urządzeniem znakowym (sys_ioctl)	EBX/RDI = deskryptor pliku ECX/RSI = kod komendy (man 2 ioctl_list) EDX/RDX = adres zapisywalnego obszaru danych lub innej struktury, zależy od komendy	EAX = 0 EAX = błąd EBADF, EFAULT, EINVAL, ENOTTY

55	72	Kontrola nad deskryptorem pliku (sys_fcntl)	EBX/RDI = deskryptor pliku ECX/RSI = kod komendy EDX/RDX zależy od komendy	EAX zależy od komendy EAX = błąd EACCES, EAGAIN, EBADF, EDEADLK, EFAULT, EINTR, EINVAL, EMFILE, ENOLOCK, EPERM
56	-	Funkcja systemowa sys_mpx (porzucone)	--nieużywane od 2.0--	zawsze EAX = -1
57	109	Ustaw ID grupy procesu (sys_setpgid)	EBX/RDI = ID procesu (PID) ECX/RSI = ID grupy	EAX = 0 EAX = błąd EINVAL, EPERM, ESRCH
58 libc	-	Pobierz/ustaw limity zasobów (sys_ulimit)	--nieużywane (zamiast tego używaj getrlimit, setrlimit, sysconf)-- man 3 ulimit EBX/RDI = komenda, patrz: sys_ulimit ECX/RSI = nowylimit	EAX = aktualny limit EAX = błąd
59	-	Funkcja systemowa sys_oldolduname (porzucone)		
60	95	Ustaw maskę uprawnień przy tworzeniu plików (sys_umask)	EBX/RDI = maska, patrz prawa dostępu / tryb Gdy tworzymy plik o uprawnieniach X, naprawdę ma on uprawnienia X AND (NOT umask)	EAX = poprzednia umask
61	161	Zmień katalog główny (sys_chroot)	EBX/RDI = adres nazwy/ścieżki nowego katalogu głównego	EAX = 0 EAX = błąd - każdy zależny od systemu plików
62 libc	136	Info o zamontowanym systemie plików (sys_ustat)	--zamiast tego, używaj statfs-- EBX:ECX?/RDI = numer główny:poboczny urządzenia / EBX/RDI -> 64 bity numeru urządzenia	EAX = 0 EAX = błąd EFAULT, EINVAL, ENOSYS

			EDX/ECX (odpowiednio) / RSI = adres struktury ustat	
63	33	Zamień deskryptor zduplikowanym deskryptorem pliku (sys_dup2)	EBX/RDI = deskryptor do zduplikowania ECX/RSI = deskryptor, do którego powinien być przyznany duplikat	EAX = zduplikowany deskryptor EAX = błąd EBADF, EMFILE
64	110	Pobierz PID procesu rodzica (sys_getppid)	nic	EAX = PID rodzica
65	111	Pobierz ID grupy procesu rodzica (sys_getpgrp)	nic	EAX = GID rodzica EAX=błąd EINVAL, EPERM, ESRCH
66	112	Stwórz sesję, ustaw ID grupy (sys_setsid)	nic	EAX = ID procesu uruchamiającego EAX=błąd EPERM
67	-	Pobierz/ustal procedurę obsługi sygnału (sys_sigaction)	EBX/RDI = numer sygnału ECX/RSI = adres struktury sigaction opisującą bieżącą procedurę EDX/RDX = adres struktury sigaction opisującą starą procedurę	EAX = 0 EAX=błąd EINVAL, EINTR, EFAULT
68	-	Pobierz maskę synałów procesu (sys_sgetmask)	--przestarzałe (zamiast tego używaj sys_sigprocmask)--	EAX = maska sygnałów bieżącego procesu
69	-	Ustaw maskę synałów procesu (sys_ssetmask)	--przestarzałe (zamiast tego używaj sys_sigprocmask)-- EBX/RDI = nowa maska sygnałów procesu	EAX = poprzednia maska sygnałów
70	113	Ustaw realny i efektywny ID użytkownika (sys_setreuid)	EBX/RDI = realny ID użytkownika (UID) ECX/RSI = efektywny UID	EAX = 0 EAX = błąd EPERM
71	114	Ustaw realny i efektywny ID grupy (sys_setregid)	EBX/RDI = realny ID grupy (GID) ECX/RSI = efektywny GID	EAX = 0 EAX = błąd EPERM

72	-	Zastąpienie dla sigpause - sys_sigsuspend	EBX/RDI = adres nowej maski sygnałowej procesu - struktury sigset_t	EAX = 0 EAX = błąd
73	-	Pobierz trwające blokujące sygnały (sys_sigpending)	EBX/RDI = adres maski sygnałów - struktury sigset_t	EAX = 0 EAX = błąd
74	170	Ustaw nazwę hosta dla systemu (sys_sethostname)	EBX/RDI = adres nazwy hosta ECX/RSI = długość nazwy	EAX = 0 EAX = błąd EFAULT, EINVAL, EPERM
75	160	Ustaw limity zasobów (sys_setrlimit)	EBX/RDI = numer zasobu ECX/RSI = adres struktury rlimit	EAX = 0 EAX = błąd EFAULT, EINVAL, EPERM
76	97	Pobierz limity zasobów (sys_getrlimit)	EBX/RDI = numer zasobu ECX/RSI = adres struktury rlimit	EAX = 0 EAX = błąd EFAULT, EINVAL, EPERM
77	98	Pobierz zużycie zasobów (sys_getrusage)	EBX/RDI = numer użytkownika (who) ECX/RSI = adres struktury rusage	EAX = 0 EAX = błąd EFAULT, EINVAL, EPERM
78	96	Pobierz czas (sys_gettimeofday)	EBX/RDI = adres struktury timeval ECX/RSI = adres struktury timezone	RAX = 0 i wynik zapisany w strukturach EAX = błąd EFAULT, EINVAL, EPERM
79	164	Ustaw czas (sys_settimeofday)	EBX/RDI = adres struktury timeval ECX/RSI = adres struktury timezone	EAX = 0 EAX = błąd EFAULT, EINVAL, EPERM

80	115	Pobierz liczbę dodatkowych grup (sys_getgroups)	EBX/RDI = rozmiar tablicy z ECX/RSI ECX/RSI = adres tablicy, gdzie zostaną zapisane GID-y (DWORDY) grup dodatkowych	EAX = liczba dodatkowych grup procesu EAX = błąd EFAULT, EINVAL, EPERM
81	116	Ustaw liczbę dodatkowych grup (sys_setgroups)	EBX/RDI = rozmiar tablicy z ECX/RSI ECX/RSI = adres tablicy, gdzie zawierającą GID-y (DWORDY)	EAX = 0 EAX = błąd EFAULT, EINVAL, EPERM
82	23	Oczekiwanie zmiany stanu deskryptoru(ów) (sys_select)	EBX/RDI = najwyższy numer spośród deskryptorów + 1 ECX/RSI = adres tablicy deskryptorów sprawdzanych, czy można z nich czytać EDX/RDX = adres tablicy deskryptorów sprawdzanych, czy można do nich pisać ESI/R10 = adres tablicy deskryptorów sprawdzanych, czy nie wystąpił u nich wyjątek EDI/R8 = adres struktury timeval zawierającą maksymalny czas oczekiwania	EAX = całkowita liczba deskryptorów, ktora pozostała w tablicach EAX = 0, gdy skończył się czas EAX = wystąpił błąd
83	88	Stwórz dowiązanie symboliczne do pliku (sys_symlink)	EBX/RDI = adres nazwy pliku ASCIIZ ECX/RSI = adres nazwę linku ASCIIZ	EAX = 0 EAX = błędy związane z uprawnieniami lub systemem plików
84	-	Funkcja systemowa sys_oldlstat (porzucone)		
85	89	Przeczytaj zawartość linku symbolicznego (sys_readlink)	EBX/RDI = adres nazwy dowiązania symbolicznego ASCIIZ ECX/RSI = adres bufora, który otrzyma pzreczytaną informację EDX/RDX = długość bufora	EAX = liczba przeczytanych znaków EAX = błąd
86	134	Wybierz współdzieloną bibliotekę (sys_uselib)	EBX/RDI = adres nazwy biblioteki ASCIIZ	EAX = 0 EAX = błąd

				EACCES, ENOEXEC
87	167	Uruchomienie pliku wymiany (sys_swapon)	EBX/RDI = adres ścieżki do pliku/urządzenia swap ECX/RSI = flagi wymiany	EAX = 0 EAX = błąd
88	169	Reboot systemu (sys_reboot)	EBX/RDI = pierwsza liczba magiczna = 0FEE1DEADh ECX/RSI = druga liczba magiczna = 672274793 lub 85072278 lub 369367448 EDX/RDX = flaga ESI/R10 = adres dodatkowego argumentu (tylko przy RESTART2)	EAX = 0 EAX = błąd
89	-	Czytaj katalog (sys_readdir)	EBX/RDI = deskryptor otwartego katalogu ECX/RSI = adres struktury dirent EDX/RDX = liczba struktur do odczytania (ignorowane, czytana jest 1 struktura)	EAX = 1 EAX = 0 na końcu katalogu EAX = błąd
90	9	Mapuj plik/urządzenie do pamięci (sys_mmap)	-- zgodne z man 2 mmap-- EBX/RDI = proponowany adres początkowy ECX/RSI = długość mapowanego obszaru EDX/RDX = ochrona ESI/R10 = flagi mapowania EDI/R8 = deskryptor mapowanego pliku, jeśli mapowanie nie jest anonimowe EBP/R9 = offset początku mapowanych danych w pliku	EAX = rzeczywisty adres mapowania EAX = błąd
91	11	Odmapuj plik/urządzenie z pamięci (sys_munmap)	EBX/RDI = adres początkowy obszaru ECX/RSI = ilość bajtów	EAX = 0 EAX = błąd
92	76	Skróć plik (sys_truncate)	EBX/RDI = adres nazwy pliku ASCIIZ ECX/RSI = ilość bajtów, do której ma zostać skrócony plik	EAX = 0 EAX = błąd
93	77	Skróć plik (sys_ftruncate)	EBX/RDI = deskryptor pliku otwartego do zapisu ECX/RSI = ilość bajtów, do której ma zostać skrócony plik	EAX = 0 EAX = błąd
94	91	Zmiana uprawnień (sys_fchmod)	EBX/RDI = deskryptor otwartego pliku	EAX = 0 EAX = błąd

			ECX/RSI = nowe prawa dostępu	
95	93	Zmiana właściciela (sys_fchown)	EBX/RDI = deskryptor otwartego pliku ECX/RSI = nowy numer użytkownika EDX/RDX = nowy numer grupy	EAX = 0 EAX = błąd
96	140	Podaj priorytet szeregowania zadań (sys_getpriority)	EBX/RDI = czyj priorytet zmieniamy ECX/RSI = identyfiaktor procesu, grupy procesów lub użytkownika, którego priorytet zmieniamy (0=bieżący)	EAX = aktualny priorytet dla wybranego obiektu (od 1 do 40)
97	141	Ustaw priorytet szeregowania zadań (sys_setpriority)	EBX/RDI = czyj priorytet zmieniamy ECX/RSI = identyfiaktor procesu, grupy procesów lub użytkownika, którego priorytet zmieniamy (0=bieżący) EDX/RDX = nowy priorytet -20...19	EAX = 0 EAX = błąd
98	-	Profilowanie czasu wykonywania (sys_profil)	--man 3 profil-- EBX/RDI = adres tablicy WORDów ECX/RSI = długość tej tablicy, na którą pokazuje EBX/RDI EDX/RDX = offset początkowy ESI/R10 = mnożnik	zawsze EAX = 0
99	137	Pobierz statystyki systemu plików (sys_statfs)	EBX/RDI = adres nazwy dowolnego pliku w zamontowanym systemie plików ECX/RSI = adres struktury statfs	EAX = 0 EAX = błąd
100	138	Pobierz statystyki systemu plików (sys_fstatfs)	EBX/RDI = deskryptor dowolnego otwartego pliku w zamontowanym systemie plików ECX/RSI = adres struktury statfs	EAX = 0 EAX = błąd

Poprzednia część (Alt+3)
Kolejna część (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Opis funkcji przerwania int 80h: 101-150

Jeśli jakaś funkcja zakończy się błędem, w EAX/RAX zwracana jest wartość ujemna z przedziału od -4096 do -1 włącznie.
Z drugiej strony, opisy funkcji na stronach manuala mówią, że zwracane jest -1, a wartość błędu jest zapisywana do zmiennej errno z biblioteki GLIBC. Dzieje się tak tylko w przypadku, gdy korzystamy z interfejsu języka C (czyli deklarujemy i uruchamiamy zewnętrzne funkcje odpowiadające wywołaniom systemowym i linkujemy nasz program z biblioteką języka C), a nie bezpośrednio z wywołań systemowych (czyli przerwania int 80h).

Najaktualniejsze informacje o funkcjach systemowych można znaleźć zazwyczaj w sekcji 2 (lub 3) manuala, np. man 2 open
Najnowsze wersje stron manuala można znaleźć tu: www.kernel.org/pub/linux/docs/man-pages.

Napis ASCIIZ oznacza łańcuch znaków ASCII zakończony znakiem/bajtem Zerowym.
Jeśli potrzeba, przy każdej funkcji jest odnośnik do opisu argumentów i innych dodatkowych informacji: typów danych, wartości błędów, możliwych wartości parametrów itp.

Podstawowe funkcje przewania 80h: 101-150

Numer/ EAX	x86-64 RAX	Opis	Argumenty	Zwraca
101	173	Zmień prawa dostępu do portów (sys_ioperm)	EBX/RDI = początkowy numer portu ECX/RSI = ilość bajtów, które będzie można wysłać/odebrać EDX/RDX = końcowy numer portu	EAX = 0 EAX = błąd
102	41	Funkcje gniazd (sys_socketcall)	EBX/RDI = numer funkcji do uruchomienia ECX/RSI = adres argumentów	EAX = wartość zwrócona przez żądaną funkcję?
103	103	Opcje logowania (sys_syslog)	EBX/RDI = komenda syslog ECX/RSI = adres bufora znakowego EDX/RDX = ilość bajtów (patrz opis EBX/RDI)	EAX = ilość bajtów (patrz opis EBX/RDI) lub 0 EAX = błąd EINVAL, EPERM, ERESTARTSYS, ENOSYS
104	38	Ustaw wartość czasomierza (sys_setitimer)	EBX/RDI = numer czasomierza ECX/RSI = adres struktury itimerval zawierającej nową wartość czasomierza ECX/RSI = adres struktury	EAX = 0 EAX = błąd

			itimerval, która otrzyma starą wartość czasomierza	
105	36	Pobierz wartość czasomierza (sys_getitimer)	EBX/RDI = numer czasomierza ECX/RSI = adres struktury itimerval, która otrzyma wartość czasomierza	EAX = 0 EAX = błąd
106	4	Pobierz status pliku (sys_stat)	EBX/RDI = adres nazwy pliku ASCIIZ. Jeśli plik jest linkiem, to zwracany jest status obiektu docelowego. ECX/RSI = adres struktury stat	EAX = 0 EAX = błąd
107	6	Pobierz status pliku (sys_lstat)	EBX/RDI = adres nazwy pliku ASCIIZ. Jeśli plik jest linkiem, to zwracany jest status linku, a nie obiektu docelowego. ECX/RSI = adres struktury stat	EAX = 0 EAX = błąd
108	5	Pobierz status pliku (sys_fstat)	EBX/RDI = deskryptor otwartego pliku ECX/RSI = adres struktury stat	EAX = 0 EAX = błąd
109	-	Funkcja systemowa sys_olduname (porzucone)		
110	172	Ustaw prawa dostępu do wszystkich portów (sys_iopl)	EBX/RDI = poziom IOPL od 0 (normalny proces) do 3	EAX = 0 EAX = błąd
111	153	Wirtualnie odłącz bieżący terminal (sys_vhangup)	nic	EAX = 0 EAX = błąd EPERM
112	-	Spowoduj bezczynność procesu 0 (sys_idle)	nic	dla procesu nr 0 nigdy nie wraca. Dla pozostałych zwraca EAX = EPERM
113	-	Przejdź w tryb wirtualny 8086 (sys_vm86old)	--to było przed jądrem 2.0.38-- EBX/RDI = adres struktury vm86_struct	EAX = 0 EAX = błąd
114	61	Czekaj na zakończenie procesu (sys_wait4)	EBX/RDI = PID procesu potomnego lub specyfikacja ECX/RSI = NULL lub adres zmiennej DWORD, która otrzyma status	EAX = PID zakończonego procesu EAX = błąd EAX = 0 dla

			EDX/RDX = opcje ESI/R10 = adres struktury rusage	WNOHANG
115	168	Wyłączenie pliku wymiany (sys_swapoff)	EBX/RDI = adres ścieżki i nazwy pliku/urządzenia swap	EAX = 0 EAX = błąd
116	99	Pobierz statystyki systemowe (sys_sysinfo)	EBX/RDI = adres struktury sysinfo	EAX = 0 EAX = błąd
117	-	Komunikacja międzyprocesowa SysV (sys_ipc)	EBX/RDI = numer wywoływanej funkcji ECX/RSI, EDX/RDX, ESI/R10 = parametry 1-3 wywoływanej funkcji EDI/R8 = adres dalszych parametrów, jeśli trzeba EBP/R9 = parametr piąty	zależy od wywoływanej funkcji
118	74	Zapisz pamięć podręczną na dysk (sys_fsync)	EBX/RDI = deskryptor pliku, który ma być zsynchronizowany na dysk	EAX = 0 EAX = błąd
119	-	Powrót z procedury obsługi sygnału (sys_sigreturn)	EBX/RDI = argument zależny od architektury, używany przez jądro	nigdy nie powraca
120	56	Utwórz klon procesu (sys_clone)	EBX/RDI = flagi klonowania ECX/RSI = wkaźnik na oddzielny stos klona EDX/RDX = wskaźnik na strukturę pt_regs lub 0	EAX = numer PID klona lub EAX = błąd EAGAIN, ENOMEM, EINVAL, EPERM
121	171	Ustal nazwę domeny (sys_setdomainname)	EBX/RDI = adres łańcucha znaków, zawierającego domenę ECX/RSI = długość tego łańcucha znaków	EAX = 0 EAX = błąd EINVAL, EPERM, EFAULT
122	63	Pobierz informację o jądrze (sys_uname)	EBX/RDI = adres struktury utsname	EAX = 0 EAX = błąd EINVAL, EPERM, EFAULT
123	154			

		Zmień tablicę LDT (sys_modify_ldt)	EBX/RDI = numer funkcji ECX/RSI = adres miejsca na przechowanie danych EDX/RDX = liczba bajtów obszaru pod [ECX/RSI]	EAX = liczba przeczytanych bajtów lub 0 (gdy zapisywano) EAX = błąd EINVAL, ENOSYS, EFAULT
124	159	Dopasowanie zegara w jądrze (sys_adjtimex)	EBX/RDI = adres struktury timex	EAX = stan zegara (patrz timex) EAX = błąd EINVAL, EPERM, EFAULT
125	10	Kontrola dostępu do obszaru pamięci (sys_mprotect)	EBX/RDI = adres obszaru pamięci (wyrównany do granicy strony) ECX/RSI = długość tego obszaru w bajtach (względem strony pamięci) EDX/RDX = bity włączające ochronę	EAX=0 EAX = błąd EACCES, ENOMEM, EINVAL, EFAULT
126	-	Zmiana listy blokowanych sygnałów (sys_sigprocmask)	EBX/RDI = co zrobić ECX/RSI = adres struktury sigset_t EDX/RDX = adres struktury sigset_t (do przechowania starej maski) lub 0	EAX = 0 EAX = błąd EINVAL, EPERM, EFAULT
127	174	Utwórz wpis ładowalnego modułu jądra (sys_create_module)	EBX/RDI = adres nazwy modułu ECX/RSI = długość nazwy	EAX = adres modułu w jądrze EAX = błąd EINVAL, EPERM, EFAULT, EEXIST, ENOMEM
128	175	Inicjalizacja modułu jądra (sys_init_module)	EBX/RDI = adres nazwy modułu ECX/RSI = adres struktury module	EAX = 0 EAX = błąd EINVAL, EPERM, EFAULT, ENOENT, EBUSY
129	176	Usuń wpis nieużywanego modułu jądra (sys_delete_module)	EBX/RDI = adres nazwy modułu (0 oznacza usunięcie wpisów wszystkich nieużywanych modułów, które można usunąć	EAX = 0 EAX = błąd EINVAL, EPERM, EFAULT, ENOENT,

			automatycznie)	EBUSY
130	177	Pobierz symbole eksportowane przez jądro i moduły (sys_get_kernel_syms)	EBX/RDI = adres struktury kernel_sym (0 oznacza, że chcemy tylko pobrać liczbę symboli)	EAX = liczba symboli EAX = błąd EINVAL, EPERM, EFAULT, ENOENT, EBUSY
131	179	Zarządzanie limitami dyskowymi (sys_quotactl)	EBX/RDI = komenda limitu ECX/RSI = adres nazwy pliku urządzenia blokowego, który ma być zarządzany EDX/RDX = identyfikator UID lub GID ESI/R10 = adres dodatkowej struktury danych (zależy od komendy w EBX/RDI)	EAX = 0 EAX = błąd EINVAL, EPERM, EFAULT, ENOENT, EBUSY, ENOTBLK, ESRCH, EUSERS, EACCES
132	121	Pobierz ID grupy procesów dla danego procesu (sys_getpgid)	EBX/RDI = PID danego procesu	EAX = ID grupy procesów EAX = błąd ESRCH
133	81	Zmień katalog roboczy (sys_fchdir)	EBX/RDI = deskryptor otwartego katalogu	EAX = 0 EAX = błąd EBADF, EACCES i inne
134	-	Demon wypróżniania buforów (sys_bdflush)	EBX/RDI = komenda demona ECX/RSI = dodatkowy parametr, zależny od komendy	EAX=0, gdy sukces i EBX/RDI>0 EAX = błąd EPERM, EFAULT, EBUSY, EINVAL
135	139	Info o systemie plików (sys_sysfs)	EBX/RDI = opcja ECX/RSI, EDX/RDX - zależne od EBX/RDI	EAX zależne od EBX/RDI EAX = błąd EINVAL, EFAULT
136	135	Ustal domenę wykonowania procesu (sys_personality)	EBX/RDI = numer nowej domeny	EAX = numer starej domeny EAX = błąd

137	183	Funkcja systemowa sys_afs_syscall	niezaimplementowane w jądrach 2.4	zawsze EAX = ENOSYS
138	122	Ustal UID przy sprawdzaniu systemów plików (sys_setfsuid)	EBX/RDI = nowy ID użytkownika	EAX = stary UID (zawsze)
139	123	Ustal GID przy sprawdzaniu systemów plików (sys_setfsgid)	EBX/RDI = nowy ID grupy	EAX = stary GID (zawsze)
140	-	Zmiana bieżącej pozycji w dużym pliku (sys_llseek)	EBX/RDI = deskryptor otwartego pliku ECX:EDX/RSI = liczba bajtów, o którą chcemy się przesunąć ESI/RDX = adres QWORDa, który otrzyma nową pozycję w pliku (big endian?) EDI/R10 = odkąd zaczynamy ruch	EAX = 0 EAX = błąd EBADF, EINVAL
141	78	Pobierz wpisy o katalogach (sys_getdents)	EBX/RDI = deskryptor otwartego katalogu ECX/RSI = adres obszaru pamięci na struktury dirent EDX/RDX = rozmiar obszaru pamięci pod [ECX/RSI]	EAX = 0 EAX = błąd EBADF, EFAULT, EINVAL, ENOENT, ENOTDIR
142	-	Oczekiwanie zmiany stanu deskryptoru(ów) (sys_newselect)	EBX/RDI = najwyższy numer spośród deskryptorów + 1 (co najwyżej FILE_MAX) ECX/RSI = adres tablicy deskryptorów (lub 0) sprawdzanych, czy można z nich czytać EDX/RDX = adres tablicy deskryptorów (lub 0) sprawdzanych, czy można do nich pisać ESI/R10 = adres tablicy deskryptorów (lub 0) sprawdzanych, czy nie wystąpił u nich wyjątek EDI/R8 = adres struktury timeval zawierającą maksymalny czas oczekiwania	EAX = całkowita liczba deskryptorów, ktora pozostała w tablicach EAX = 0, gdy skończył się czas EAX = wystąpił błąd EBADF, EINVAL, ENOMEM, EINTR
143	73	Zmień blokowanie plików (sys_flock)	EBX/RDI = deskryptor otwartego pliku ECX/RSI = operacja do wykonania	EAX = 0 EAX = błąd EWOULDBLOCK,

				EBADF, EINTR, EINVAL, ENOLCK
144	26	Synchronizuj mapowany plik z pamięcią (sys_msync)	EBX/RDI = adres do zrzucenia na dysk (zostaną zrzucone zmodyfikowane strony pamięci zawierającej ten adres i co najwyżej ECX/RSI-1 zmienionych następnych) ECX/RSI = ilość bajtów/rozmiar obszaru do zrzucenia na dysk EDX/RDX = 0 lub zORowane flagi	EAX = 0 EAX = błąd EBUSY, EIO, ENOMEM, EINVAL, ENOLCK
145	19	Czytaj wektor (sys_readv)	EBX/RDI = deskryptor otwartego obiektu, z którego będą czytane dane ECX/RSI = adres tablicy struktur iovec EDX/RDX = liczba struktur iovec, do których będą czytane dane	EAX = 0 EAX = błąd EWOULDBLOCK, EBADF, EINTR, EINVAL, ENOLCK
146	20	Zapisz wektor (sys_writev)	EBX/RDI = deskryptor otwartego obiektu, do którego będą zapisane dane ECX/RSI = adres tablicy struktur iovec EDX/RDX = liczba struktur iovec, z których będą czytane dane do zapisania	EAX = 0 EAX = błąd EWOULDBLOCK, EBADF, EINTR, EINVAL, ENOLCK
147	124	Pobierz ID sesji dla procesu (sys_getsid)	EBX/RDI = PID procesu, którego ID sesji chcemy znać	EAX = ID sesji EAX = błąd EPERM, ESRCH
148	75	Zapisz bufory danych pliku na dysk (sys_fdatasync)	EBX/RDI = deskryptor pliku, którego DANE będą zsynchronizowane (ale np. czas dostępu nie będzie zmieniony)	EAX = 0 EAX = błąd EBADF, EIO, EROFS
149	156	Zmień parametry jądra (sys_sysctl)	EBX/RDI = adres struktury sysctl_args	EAX = 0 EAX = błąd EPERM, ENOTDIR, EFAULT
150	149	Zablokowanie stron w pamięci (sys_mlock)	EBX/RDI = adres obszaru pamięci (wyrównany do wielokrotności rozmiaru strony pamięci) ECX/RSI = długość obszaru pamięci	EAX = 0 EAX = błąd EINVAL, EAGAIN, ENOMEM

Poprzednia część (Alt+3)
Kolejna część (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Opis funkcji przerwania int 80h: 151-200

Jeśli jakaś funkcja zakończy się błędem, w EAX/RAX zwracana jest wartość ujemna z przedziału od -4096 do -1 włącznie.
Z drugiej strony, opisy funkcji na stronach manuala mówią, że zwracane jest -1, a wartość błędu jest zapisywana do zmiennej errno z biblioteki GLIBC. Dzieje się tak tylko w przypadku, gdy korzystamy z interfejsu języka C (czyli deklarujemy i uruchamiamy zewnętrzne funkcje odpowiadające wywołaniom systemowym i linkujemy nasz program z biblioteką języka C), a nie bezpośrednio z wywołań systemowych (czyli przerwania int 80h).

Najaktualniejsze informacje o funkcjach systemowych można znaleźć zazwyczaj w sekcji 2 (lub 3) manuala, np. man 2 open
Najnowsze wersje stron manuala można znaleźć tu: www.kernel.org/pub/linux/docs/man-pages.

Napis ASCIIZ oznacza łańcuch znaków ASCII zakończony znakiem/bajtem Zerowym.
Jeśli potrzeba, przy każdej funkcji jest odnośnik do opisu argumentów i innych dodatkowych informacji: typów danych, wartości błędów, możliwych wartości parametrów itp.

Podstawowe funkcje przewania 80h: 151-200

Numer/ EAX	x86-64 RAX	Opis	Argumenty	Zwraca
151	150	Odblokowanie stron pamięci (sys_munlock)	EBX/RDI = adres obszaru pamięci (wyrównany do wielokrotności rozmiaru strony pamięci) ECX/RSI = długość obszaru pamięci	EAX = 0 EAX = błąd EINVAL, ENOMEM
152	151	Zablokowanie całej pamięci procesu (sys_mlockall)	EBX/RDI = flagi blokowania pamięci	EAX = 0 EAX = błąd EINVAL, ENOMEM, EAGAIN, EPERM
153	152	Odblokowanie całej pamięci procesu (sys_munlockall)	nic	EAX = 0 EAX = błąd.
154	142	Ustaw parametry szeregowania zadań (sys_sched_setparam)	EBX/RDI = PID procesu ECX/RSI = adres struktury sched_param, zawierającej dane	EAX = 0 EAX = błąd EINVAL, ESRCH, EPERM
155	143	Pobierz parametry szeregowania zadań (sys_sched_getparam)	EBX/RDI = PID procesu ECX/RSI = adres struktury	EAX = 0 EAX = błąd EINVAL,

			sched_param, która otrzyma wynik	ESRCH, EPERM
156	144	Ustaw parametry/algorytm szeregowania zadań (sys_sched_setsheduler)	EBX/RDI = PID procesu ECX/RSI = polityka ECX/RSI = adres struktury sched_param, zawierającej dane	EAX = 0 EAX = błąd EINVAL, ESRCH, EPERM
157	145	Pobierz parametry/algorytm szeregowania zadań (sys_sched_getsheduler)	EBX/RDI = PID procesu	EAX = polityka EAX = błąd EINVAL, ESRCH, EPERM
158	24	Oddanie procesora innym procesom (sys_sched_yield)	nic	EAX = 0 EAX = błąd.
159	146	Pobierz maksymalny priorytet statyczny (sys_sched_get_priority_max)	EBX/RDI = polityka	EAX = maksymalny priorytet dla tej polityki EAX = błąd EINVAL
160	147	Pobierz minimalny priorytet statyczny (sys_sched_get_priority_min)	EBX/RDI = polityka	EAX = minimalny priorytet dla tej polityki EAX = błąd EINVAL
161	148	Pobierz długość czasu w szeregowaniu cyklicznym (sys_sched_rr_get_interval)	EBX/RDI = PID procesu (0 = ten proces) ECX/RSI = adres strukury timeval, która otrzyma wynik	EAX = 0 EAX = błąd ESRCH, ENOSYS
162	35	Pauza w wykonywaniu programu (sys_nanosleep)	EBX/RDI = adres struktury timespec ECX/RSI = NULL lub adres modyfikowalnej struktury timespec, która otrzyma resztkę czasu, która została	EAX = 0 EAX = sygnał lub błąd EINTR, EINVAL
163	25	Przemapuj adres wirtualny (sys_mremap)	EBX/RDI = stary adres ECX/RSI = rozmiar obszaru do przemapowania EDX/RDX = żądany rozmiar ESI/R10 = zero lub flagi przemapowania EDI/R8 = nowy adres, jeśli dano flagę MREMAP_FIXED	EAX = wskaźnik do nowego obszaru EAX = sygnał lub błąd EFAULT, EAGAIN, ENOMEM, EINVAL

164	117	Ustaw różne ID użytkownika (sys_setresuid)	EBX/RDI = realny UID lub -1 (wtedy jest bez zmian) ECX/RSI = efektywny UID lub -1 (bez zmian) EDX/RDX = zachowany (saved) UID lub -1 (bez zmian)	EAX = 0 EAX = błąd EPERM
165	118	Pobierz różne ID użytkownika (sys_getresuid)	EBX/RDI = adres DWORDa, który otrzyma realny UID ECX/RSI = adres DWORDa, który otrzyma efektywny UID EDX/RDX = adres DWORDa, który otrzyma zachowany UID	EAX = 0 EAX = błąd EFAULT
166	-	Uruchom tryb wirtualny 8086 (sys_vm86)	EBX/RDI = kod funkcji ECX/RSI = adres struktury vm86plus_struct	(zalezy od numeru funkcji) EAX = błąd EFAULT
167	178	Zapytaj o moduł (sys_query_module)	EBX/RDI = adres nazwy modułu lub NULL (jądro) ECX/RSI = numer podfunkcji EDX/RDX = adres bufora ESI/R10 = rozmiar bufora EDI/R8 = adres DWORDa	EAX = 0 EAX = błąd EFAULT, ENOSPC, EINVAL, ENOENT
168	7	Czekaj na zdarzenia na deskryptorze (sys_poll)	EBX/RDI = adres tablicy struktur pollfd ECX/RSI = liczba struktur pollfd w tablicy EDX/RDX = max. czas na oczekiwanie w milisekundach (-1 = nieskończoność)	EAX = liczba odpowiednich deskryptorów EAX = 0, gdy czas upłynął EAX = błąd EFAULT, EINTR, EINVAL
169	180	Interfejs demona NFS (sys_nfsservctl)	EBX/RDI = komenda ECX/RSI = adres struktury nfsctl_arg EDX/RDX = adres unii union nfsctl_res	EAX = 0 EAX = błąd
170	119	Ustaw realny, efektywny i zachowany ID grupy (sys_setresgid)	EBX/RDI = realny GID ECX/RSI = efektywny GID EDX/RDX = zachowany (saved) GID	EAX = 0 EAX = błąd EPERM
171	120	Pobierz realny, efektywny i zachowany ID grupy (sys_getresgid)	EBX/RDI = adres DWORDa, który otrzyma realny GID ECX/RSI = adres DWORDa, który otrzyma	EAX = 0 EAX = błąd EFAULT

			efektywny GID EDX/RDX = adres DWORDa, który otrzyma zachowany (saved) GID	
172	157	Dzialania na procesie (sys_prctl)	EBX/RDI = opcja ECX/RSI, EDX/RDX, ESI/R10, EDI/R8 = argumenty	EAX = 0 lub 1 EAX = błąd EINVAL
173	15	Powrót z procedury obsługi sygnału (sys_rt_sigreturn)	-- funkcja wewnętrzna, nie używać-- EBX/RDI = parametr zależny od architektury	nigdy nie powraca
174	13	Pobierz i zmień procedurę obsługi sygnału (sys_rt_sigaction)	EBX/RDI = numer sygnału ECX/RSI = adres struktury sigaction opisującą bieżącą procedurę EDX/RDX = adres struktury sigaction opisującą starą procedurę ESI/R10 = rozmiar struktury sigset_t	EAX = 0 EAX=błąd EINVAL, EINTR, EFAULT
175	14	Pobierz i zmień blokowane sygnały (sys_rt_sigprocmask)	EBX/RDI = działanie ECX/RSI = adres zestawu sygnałów (tablicy 32 DWORDów) EDX/RDX = adres zestawu sygnałów, który otrzyma starą maskę sygnałów ESI/R10 = rozmiar struktury sigset_t	EAX = 0 EAX=błąd EINVAL
176	127	Pobierz sygnały oczekujące (sys_rt_sigpending)	EBX/RDI = adres zestawu sygnałów, który otrzyma oczekujące sygnały ECX/RSI = rozmiar struktury sigset_t	EAX = 0 EAX=błąd EFAULT
177	128	Synchronicznie czekaj na zakolejkowane sygnały (sys_rt_sigtimedwait)	EBX/RDI = adres zestawu sygnałów, na które czekać ECX/RSI = adres struktury siginfo, która otrzyma informację o sygnale EDX/RDX = adres struktury timespec określającej czas oczekiwania ESI/R10 = rozmiar struktury sigset_t	EAX = numer sygnału EAX=błąd EINVAL, EINTR, EAGAIN
178	129	Zakolejkuj sygnał dla procesu (sys_rt_sigqueueinfo)	EBX/RDI=PID procesu, który ma otrzymać sygnał ECX/RSI=numer sygnału EDX/RDX=adres struktury	EAX = 0 EAX=błąd EAGAIN, EINVAL, EPERM, ESRCH

			siginfo_t do wysłania procesowi razem z sygnałem	
179	130	Czekaj na sygnał (sys_rt_sigsuspend)	EBX/RDI = adres zestawu sygnałów, na które czekać ECX/RSI = rozmiar struktury sigset_t	EAX = -1 EAX=błąd EINTR, EFAULT
180	17	Czytaj z danej pozycji w pliku (sys_pread/sys_pread64)	EBX/RDI = deskryptor otwartego pliku ECX/RSI = adres bufora, który otrzyma dane EDX/RDX = ilość bajtów do odczytania ESI/R10 = pozycja, z której zacząć czytanie	EAX = ilość przeczytanych bajtów (wskaźnik pozycji w pliku pozostaje bez zmian) EAX = błąd (jak w sys_read)
181	18	Zapisuj na danej pozycji w pliku (sys_pwrite/sys_pwrite64)	EBX/RDI = deskryptor otwartego pliku ECX/RSI = adres bufora, z którego pobierać dane do zapisania EDX/RDX = ilość bajtów do zapisania ESI/R10 = pozycja, od której zacząć zapisywanie	EAX = ilość zapisanych bajtów (wskaźnik pozycji w pliku pozostaje bez zmian) EAX = błąd (jak w sys_read)
182	92	Zmiana właściciela pliku (sys_chown)	EBX/RDI=adres ścieżki do pliku ECX/RSI = UID nowego właściciela EDX/RDX = GID nowej grupy	EAX = 0 EAX = błąd np. EPERM, EROFS, EFAULT, ENOENT, ENAMETOOLONG, ENOMEM, ENOTDIR, EACCES, ELOOP
183	79	Pobierz bieżący katalog roboczy (sys_getcwd)	EBX/RDI = adres bufora, który otrzyma ścieżkę ECX/RSI = długość tego bufora	EAX = EBX/RDI EAX=NULL, gdy błąd ERANGE, EACCES, EFAULT, EINVAL, ENOENT
184	125	Pobierz możliwości procesu (sys_capget)	EBX/RDI = adres struktury cap_user_header_t ECX/RSI = adres struktury cap_user_data_t	EAX = EBX/RDI EAX=NULL, gdy błąd EPERM, EINVAL
185	126	Ustaw możliwości procesu (sys_capset)	EBX/RDI = adres struktury cap_user_header_t ECX/RSI = adres struktury	EAX = EBX/RDI EAX=NULL, gdy błąd EPERM, EINVAL

			cap_user_data_t	
186	131	Ustaw alternatywny stos dla procedur obsługi sygnałów (sys_sigaltstack)	EBX/RDI = adres struktury stack_t, opisującej nowy stos ECX/RSI = adres struktury stack_t, opisującej stary stos; lub NULL (ewentualnie EDX/RDX = adres nowego wierzchołka stosu)	EAX = 0 EAX = błąd EPERM, EINVAL, ENOMEM
187	40	Kopiuj dane między deskryptorami plików (sys_sendfile)	EBX/RDI = deskryptor pliku wyjściowego, otwartego do zapisu ECX/RSI = deskryptor pliku wejściowego EDX/RDX = adres 64-bitowej zmiemnnej - numeru bajtu w pliku źródłowym, od którego zacząć kopiować ESI/R10 = liczba bajtów do skopiowania	EAX = liczba zapisanych bajtów EAX = błąd EBADF, EAGAIN, EINVAL, ENOMEM, EIO, EFAULT
188	181	Funkcja systemowa sys_getpmsg	niezaimplementowane w jądrach 2.4, na systemach 64-bitowych zarezerwowane dla LiS/STREAMS	zawsze EAX = ENOSYS
189	182	Funkcja systemowa sys_putpmsg	niezaimplementowane w jądrach 2.4, na systemach 64-bitowych zarezerwowane dla LiS/STREAMS	zawsze EAX = ENOSYS
190	58	Utwórz proces potomny i zablokuj rodzica (sys_vfork)	nic	EAX = PID procesu potomnego EAX = błąd EAGAIN, ENOMEM
191	-	Pobierz limity zasobów (sys_ugetrlimit)	patrz: sys_getrlimit (?)	EAX = 0 EAX = błąd EFAULT, EINVAL, EPERM
192	-	Mapuj urządzenie lub plik do pamięci (sys_mmap2)	EBX/RDI = proponowany adres początkowy ECX/RSI = ilość bajtów pliku do zmapowania EDX/RDX = ochrona	EAX = adres zmapowanego obszaru EAX = błąd (takie same jak w sys_mmap + EFAULT)

			ESI/R10 = flagi mapowania EDI/R8 = deskryptor mapowanego pliku, jeśli mapowanie nie jest anonimowe EBP/R9 = offset początku mapowanych danych w pliku, liczony w jednostkach wielkości strony systemowej zamiast w bajtach	
193	-	Skróć plik, wersja 64-bitowa (sys_truncate64)	EBX/RDI = adres nazwy pliku ASCIIZ ECX/RSI = ilość bajtów, do której ma zostać skrócony plik (niższy DWORD) EDX/RDX = ilość bajtów, do której ma zostać skrócony plik (wyższy DWORD)	EAX = 0 EAX = błąd
194	-	Skróć plik, wersja 64-bitowa (sys_ftruncate64)	EBX/RDI = deskryptor pliku otwartego do zapisu ECX/RSI = ilość bajtów, do której ma zostać skrócony plik (niższy DWORD) EDX/RDX = ilość bajtów, do której ma zostać skrócony plik (wyższy DWORD)	EAX = 0 EAX = błąd
195	-	Pobierz status pliku, wersja 64-bitowa (sys_stat64)	EBX/RDI = adres nazwy pliku ASCIIZ. Jeśli plik jest linkiem, to zwracany jest status obiektu docelowego. ECX/RSI = adres struktury stat64	EAX = 0 EAX = błąd
196	-	Pobierz status pliku, wersja 64-bitowa (sys_lstat64)	EBX/RDI = adres nazwy pliku ASCIIZ. Jeśli plik jest linkiem, to zwracany jest status linku, a nie obiektu docelowego. ECX/RSI = adres struktury stat64	EAX = 0 EAX = błąd
197	-	Pobierz status pliku, wersja 64-bitowa (sys_fstat64)	EBX/RDI = deskryptor otwartego pliku ECX/RSI = adres struktury stat64	EAX = 0 EAX = błąd
198	-	Zmiana właściciela (sys_lchown32)	EBX/RDI = adres nazwy pliku/katalogu ASCIIZ ECX/RSI = nowy numer użytkownika	EAX = 0 EAX = błąd EPERM, EROFS, EFAULT, ENAMETOOLONG,

			EDX/RDX = nowy numer grupy	ENOENT, ENOMEM, ENOTDIR, EACCES, ELOOP i inne
199	-	Pobierz identyfikator użytkownika (sys_getuid32)	nic	EAX = numer UID
200	-	Pobierz ID grupy bieżącego procesu (sys_getgid32)	nic	EAX = ID grupy

Poprzednia część (Alt+3)
Kolejna część (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Opis funkcji przerwania int 80h: 201-250

Jeśli jakaś funkcja zakończy się błędem, w EAX/RAX zwracana jest wartość ujemna z przedziału od -4096 do -1 włącznie.
Z drugiej strony, opisy funkcji na stronach manuala mówią, że zwracane jest -1, a wartość błędu jest zapisywana do zmiennej errno z biblioteki GLIBC. Dzieje się tak tylko w przypadku, gdy korzystamy z interfejsu języka C (czyli deklarujemy i uruchamiamy zewnętrzne funkcje odpowiadające wywołaniom systemowym i linkujemy nasz program z biblioteką języka C), a nie bezpośrednio z wywołań systemowych (czyli przerwania int 80h).

Najaktualniejsze informacje o funkcjach systemowych można znaleźć zazwyczaj w sekcji 2 (lub 3) manuala, np. man 2 open
Najnowsze wersje stron manuala można znaleźć tu: www.kernel.org/pub/linux/docs/man-pages.

Napis ASCIIZ oznacza łańcuch znaków ASCII zakończony znakiem/bajtem Zerowym.
Jeśli potrzeba, przy każdej funkcji jest odnośnik do opisu argumentów i innych dodatkowych informacji: typów danych, wartości błędów, możliwych wartości parametrów itp.

Podstawowe funkcje przewania 80h: 201-250

Numer/ EAX	x86-64 RAX	Opis	Argumenty	Zwraca
201	-	Pobierz efektywne ID użytkownika bieżącego procesu (sys_geteuid32)	nic	EAX = efektywny UID
202	-	Pobierz efektywne ID grupy bieżącego procesu (sys_getegid32)	nic	EAX = efektywny GID
203	-	Ustaw realny i efektywny ID użytkownika (sys_setreuid32)	EBX/RDI = realny ID użytkownika (UID) ECX/RSI = efektywny UID	EAX = 0 EAX = błąd EPERM
204	-	Ustaw realny i efektywny ID grupy (sys_setregid32)	EBX/RDI = realny ID grupy (GID) ECX/RSI = efektywny GID	EAX = 0 EAX = błąd EPERM
205	-	Pobierz liczbę dodatkowych grup (sys_getgroups32)	EBX/RDI = rozmiar tablicy z ECX/RSI ECX/RSI = adres tablicy, gdzie zostaną zapisane GID-y (DWORDY) grup dodatkowych	EAX = liczba dodatkowych grup procesu EAX = błąd EFAULT, EINVAL, EPERM
206	-	Ustaw liczbę dodatkowych grup (sys_setgroups32)	EBX/RDI = rozmiar tablicy z ECX/RSI ECX/RSI = adres tablicy, gdzie	EAX = 0 EAX = błąd EFAULT, EINVAL, EPERM

			zawierającą GID-y (DWORDY)	
207	-	Zmiana właściciela (sys_fchown32)	EBX/RDI = deskryptor otwartego pliku ECX/RSI = nowy numer użytkownika EDX/RDX = nowy numer grupy	EAX = 0 EAX = błąd
208	-	Ustaw różne ID użytkownika (sys_setresuid32)	EBX/RDI = realny UID lub -1 (wtedy jest bez zmian) ECX/RSI = efektywny UID lub -1 (bez zmian) EDX/RDX = zachowany (saved) UID lub -1 (bez zmian)	EAX = 0 EAX = błąd EPERM
209	-	Pobierz różne ID użytkownika (sys_getresuid32)	EBX/RDI = adres DWORDa, który otrzyma realny UID ECX/RSI = adres DWORDa, który otrzyma efektywny UID EDX/RDX = adres DWORDa, który otrzyma zachowany UID	EAX = 0 EAX = błąd EFAULT
210	-	Ustaw realny, efektywny i zachowany ID grupy (sys_setresgid32)	EBX/RDI = realny GID ECX/RSI = efektywny GID EDX/RDX = zachowany (saved) GID	EAX = 0 EAX = błąd EPERM
211	-	Pobierz realny, efektywny i zachowany ID grupy (sys_getresgid32)	EBX/RDI = adres DWORDa, który otrzyma realny GID ECX/RSI = adres DWORDa, który otrzyma efektywny GID EDX/RDX = adres DWORDa, który otrzyma zachowany (saved) GID	EAX = 0 EAX = błąd EFAULT
212	-	Zmiana właściciela pliku (sys_chown32)	EBX/RDI=adres ścieżki do pliku ECX/RSI = UID nowego właściciela EDX/RDX = GID nowej grupy	EAX = 0 EAX = błąd np. EPERM, EROFS, EFAULT, ENOENT, ENAMETOOLONG, ENOMEM, ENOTDIR, EACCES, ELOOP
213	-	Ustaw identyfikator użytkownika (sys_setuid32)	EBX/RDI = nowy UID	EAX = 0 EAX = błąd EPERM
214	-	Ustaw ID grupy bieżącego procesu (sys_setgid32)	EBX/RDI = nowy ID grupy	EAX = 0 EAX = błąd EPERM
215	-		EBX/RDI = nowy ID użytkownika	EAX = stary UID (zawsze)

		Ustal UID przy sprawdzaniu systemów plików (sys_setfsuid32)		
216	-	Ustal GID przy sprawdzaniu systemów plików (sys_setfsgid32)	EBX/RDI = nowy ID grupy	EAX = stary GID (zawsze)
217	155	Zmień główny system plików/katalog (sys_pivot_root)	EBX/RDI = adres łańcucha znaków - nowy główny katalog bieżącego procesu ECX/RSI = adres łańcucha znaków - otrzyma stary główny katalog bieżącego procesu	EAX = 0 EAX = błąd EBUSY, EINVAL, EPERM, ENOTDIR + błędy sys_stat
218	27	Pobierz informację, czy strony pamięci są w rdzeniu procesu (sys_mincore)	EBX/RDI = adres początkowy sprawdzanych bajtów ECX/RSI = liczba sprawdzanych bajtów EDX/RDX = adres tablicy bajtów zdolnej pomieścić tyle bajtów, ile stron pamięci jest sprawdzanych. Najmłodszy bit w każdym bajcie będzie mówił o tym, czy dana strona pamięci jest obecna (=1), czy zrzucona na dysk (=0)	EAX = 0 EAX = błąd EAGAIN, EINVAL, EFAULT, ENOMEM
219	28	Porada dla jądra o uzyciu pamięci (sys_madvise, sys_madvise1)	EBX/RDI = adres początkowy bajtów, których dotoczy porada ECX/RSI = liczba tych bajtów EDX/RDX = porada	EAX = 0 EAX = błąd EAGAIN, EINVAL, EFAULT, ENOMEM
220	217	Pobierz wpisy o katalogach, wersja 64-bitowa (sys_getdents64)	EBX/RDI = deskryptor otwartego katalogu ECX/RSI = adres obszaru pamięci na struktury dirent EDX/RDX = rozmiar obszaru pamięci pod [ECX/RSI]	EAX = 0 EAX = błąd EBADF, EFAULT, EINVAL, ENOENT, ENOTDIR
221	-	Kontrola nad deskryptorem pliku, wersja 64-bitowa (sys_fcntl64)	EBX/RDI = deskryptor pliku ECX/RSI = kod komendy EDX/RDX zależy od komendy	EAX zależy od komendy EAX = błąd EACCES, EAGAIN, EBADF, EDEADLK, EFAULT, EINTR, EINVAL, EMFILE, ENOLOCK, EPERM
222	-	brak danych	-	-
223	185			zawsze EAX = ENOSYS

		Funkcja systemowa sys_security	niezaimplementowane w jądrach 2.4	
224	186	Pobierz identyfikator wątku (sys_gettid)	nic	EAX = id wątku
225	187	Czytaj kilka stron pliku z wyprzedzeniem do pamięci podręcznej (sys_readahead)	EBX/RDI = deskryptor pliku ECX/RSI = miejsce w pliku, od którego zacząć EDX/RDX = ilość bajtów do przeczytania	EAX = -EBADF, gdy błąd
226	188	Ustaw wartość atrybutu rozszerzonego (sys_setxattr)	EBX/RDI = adres ścieżki pliku ECX/RSI = adres nazwy atrybutu EDX/RDX = wartość atrybutu ESI/R10 = długość atrybutu EDI/R8 = flaga (1=utwórz, 2=zamień)	EAX = 0 EAX = błąd
227	189	Ustaw wartość atrybutu rozszerzonego (sys_lsetxattr)	EBX/RDI = adres ścieżki pliku, funkcja nie podąża za dowiązaniami symbolicznymi ECX/RSI = adres nazwy atrybutu EDX/RDX = wartość atrybutu ESI/R10 = długość atrybutu EDI/R8 = flaga (1=utwórz, 2=zamień)	EAX = 0 EAX = błąd
228	190	Ustaw wartość atrybutu rozszerzonego (sys_fsetxattr)	EBX/RDI = deskryptor pliku ECX/RSI = adres nazwy atrybutu EDX/RDX = wartość atrybutu ESI/R10 = długość atrybutu EDI/R8 = flaga (1=utwórz, 2=zamień)	EAX = 0 EAX = błąd
229	191	Pobierz wartość atrybutu rozszerzonego (sys_getxattr)	EBX/RDI = adres ścieżki pliku ECX/RSI = adres nazwy atrybutu EDX/RDX = wartość atrybutu ESI/R10 = długość atrybutu	EAX = 0 EAX = błąd
230	192	Pobierz wartość atrybutu rozszerzonego (sys_lgetxattr)	EBX/RDI = adres ścieżki pliku, funkcja nie podąża za dowiązaniami symbolicznymi ECX/RSI = adres nazwy atrybutu EDX/RDX = wartość atrybutu ESI/R10 = długość atrybutu	EAX = 0 EAX = błąd
231	193	Pobierz wartość atrybutu rozszerzonego (sys_fgetxattr)	EBX/RDI = deskryptor pliku ECX/RSI = adres nazwy atrybutu EDX/RDX = wartość atrybutu ESI/R10 = długość atrybutu	EAX = 0 EAX = błąd
232	194	Pobierz listę nazw atrybutów rozszerzonych pliku (sys_listxattr)	EBX/RDI = adres ścieżki pliku ECX/RSI = adres tablicy na nazwy EDX/RDX = długość tablicy	EAX = 0 EAX = błąd

233	195	Pobierz listę nazw atrybutów rozszerzonych pliku (sys_llistxattr)	EBX/RDI = adres ścieżki pliku, funkcja nie podąża za dowiązaniami symbolicznymi ECX/RSI = adres tablicy na nazwy EDX/RDX = długość tablicy	EAX = 0 EAX = błąd
234	196	Pobierz listę nazw atrybutów rozszerzonych pliku (sys_flistxattr)	EBX/RDI = deskryptor pliku ECX/RSI = adres tablicy na nazwy EDX/RDX = długość tablicy	EAX = 0 EAX = błąd
235	197	Usuń atrybut rozszerzony pliku (sys_removexattr)	EBX/RDI = adres ścieżki pliku ECX/RSI = adres nazwy atrybutu do usunięcia	EAX = 0 EAX = błąd
236	198	Usuń atrybut rozszerzony pliku (sys_lremovexattr)	EBX/RDI = adres ścieżki pliku, funkcja nie podąża za dowiązaniami symbolicznymi ECX/RSI = adres nazwy atrybutu do usunięcia	EAX = 0 EAX = błąd
237	199	Usuń atrybut rozszerzony pliku (sys_fremovexattr)	EBX/RDI = deskryptor pliku ECX/RSI = adres nazwy atrybutu do usunięcia	EAX = 0 EAX = błąd
238	200	Zabij pojedyncze zadanie (sys_tkill)	EBX/RDI = PID zadania (niekoniecznie całego procesu) ECX/RSI = numer sygnału do wysłania	EAX = 0 EAX = błąd EINVAL, ESRCH, EPERM
239	40	Kopiuj dane między deskryptorami plików (sys_sendfile64)	EBX/RDI = deskryptor pliku wyjściowego, otwartego do zapisu ECX/RSI = deskryptor pliku wejściowego EDX/RDX = adres 64-bitowej zmiemnnej - numeru bajtu w pliku źródłowym, od którego zacząć kopiować ESI/R10 = liczba bajtów do skopiowania	EAX = liczba zapisanych bajtów EAX = błąd EBADF, EAGAIN, EINVAL, ENOMEM, EIO, EFAULT
240	202	Szybka funkcja blokowania (sys_futex)	EBX/RDI = sprawdzany adres ECX/RSI = operacja EDX/RDX = wartość ESI/R10 = adres struktury timespec (czas oczekiwania) lub 0	EAX zależy od operacji EAX = błąd EINVAL, EFAULT
241	203	Ustaw maskę procesorów dla procesu (sys_sched_setaffinity)	EBX/RDI = PID procesu, którego maskę ustawiamy (0=bieżący) ECX/RSI = długość maski pod [EDX/RDX] EDX/RDX = adres maski bitowej. Najmłodszy bit maski oznacza,	EAX = 0 EAX = błąd EINVAL, EFAULT, ESRCH, EPERM

			czy proces może być wykonany na pierwszym procesorze logicznym itp...	
242	204	Pobierz maskę procesorów dla procesu (sys_sched_getaffinity)	EBX/RDI = PID procesu, którego maskę pobieramy (0=bieżący) ECX/RSI = długość maski pod [EDX/RDX] EDX/RDX = adres maski bitowej. Najmłodszy bit maski oznacza, czy proces może być wykonany na pierwszym procesorze logicznym itp...	EAX = 0 EAX = błąd EINVAL, EFAULT, ESRCH, EPERM
243	205	Ustaw wpis w obszarze lokalnym wątku TLS (sys_set_thread_area)	EBX/RDI = adres struktury user_desc	EAX = 0 EAX = -EINVAL, -EFAULT, -ESRCH
244	211	Pobierz wpis w obszarze lokalnym wątku TLS (sys_get_thread_area)	EBX/RDI = adres struktury user_desc	EAX = 0 EAX = błąd EINVAL, EFAULT
245	206	Utwórz asynchroniczny kontekst we/wy (sys_io_setup)	EBX/RDI = liczba zradzeń, które kontekst może otrzymać ECX/RSI = adres DWORDa (o wartosci zero), który otrzyma uchwyt do nowego kontekstu	EAX = 0 EAX = błąd -EINVAL, -EFAULT, -ENOSYS, -ENOMEM, -EAGAIN
246	207	Zniszcz asynchroniczny kontekst we/wy (sys_io_destroy)	EBX/RDI = uchwyt do usuwanego kontekstu	EAX = 0 EAX = błąd -EINVAL, -EFAULT, -ENOSYS
247	208	Pobierz zdarzenia we/wy (sys_io_getevents)	EBX/RDI = uchwyt do kontekstu ECX/RSI = minimalna liczba zdarzeń do pobrania EDX/RDX = maksymalna liczba zdarzeń do pobrania ESI/R10 = adres tablicy struktur io_event EDI/R8 = adres struktury timespec (czas oczekiwania) lub 0	EAX = liczba odczytanych zdarzeń EAX = błąd -EINVAL, -EFAULT, -ENOSYS
248	209	Wyślij zdarzenia we/wy do przetworzenia (sys_io_submit)	EBX/RDI = uchwyt do kontekstu ECX/RSI = liczba adresów struktur pod [EDX/RDX] EDX/RDX = adres tablicy adresów struktur iocb opisujących zdarzenia do przetworzenia	EAX = liczba wysłanych bloków we/wy EAX = błąd -EINVAL, -EFAULT, -ENOSYS, -EBADF, -EAGAIN

249	210	Przerwij operację we/wy (sys_io_cancel)	EBX/RDI = uchwyt do kontekstu ECX/RSI = adres struktury iocb, opisującej operację do przerwania EDX/RDX = adres struktury io_event, która otrzyma przerwane działanie	EAX = 0 EAX = błąd -EINVAL, -EFAULT, -ENOSYS, -EBADF, -EAGAIN
250	-	Funkcja systemowa sys_alloc_hugepages	zaimplementowane tylko w jądrach 2.5.36 - 2.5.54, więc nie będę omawiał	zawsze EAX = ENOSYS

Poprzednia część (Alt+3)
Kolejna część (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Opis funkcji przerwania int 80h: 251-300

Jeśli jakaś funkcja zakończy się błędem, w EAX/RAX zwracana jest wartość ujemna z przedziału od -4096 do -1 włącznie.
Z drugiej strony, opisy funkcji na stronach manuala mówią, że zwracane jest -1, a wartość błędu jest zapisywana do zmiennej errno z biblioteki GLIBC. Dzieje się tak tylko w przypadku, gdy korzystamy z interfejsu języka C (czyli deklarujemy i uruchamiamy zewnętrzne funkcje odpowiadające wywołaniom systemowym i linkujemy nasz program z biblioteką języka C), a nie bezpośrednio z wywołań systemowych (czyli przerwania int 80h).

Najaktualniejsze informacje o funkcjach systemowych można znaleźć zazwyczaj w sekcji 2 (lub 3) manuala, np. man 2 open
Najnowsze wersje stron manuala można znaleźć tu: www.kernel.org/pub/linux/docs/man-pages.

Napis ASCIIZ oznacza łańcuch znaków ASCII zakończony znakiem/bajtem Zerowym.
Jeśli potrzeba, przy każdej funkcji jest odnośnik do opisu argumentów i innych dodatkowych informacji: typów danych, wartości błędów, możliwych wartości parametrów itp.

Podstawowe funkcje przewania 80h: 251-300

Numer/ EAX	x86-64 RAX	Opis	Argumenty	Zwraca
251	-	Funkcja systemowa sys_free_hugepages	zaimplementowane tylko w jądrach 2.5.36 - 2.5.54, więc nie będę omawiał	zawsze EAX = ENOSYS
252	231	Zakończ wszystkie wątki procesu (sys_exit_group)	EBX/RDI = status (kod wyjścia)	nigdy nie powraca
253*	212	Pobierz ścieżkę wejścia do katalogu (sys_lookup_dcookie)	EBX:ECX/RDI? = wartość opisująca wpis o katalogu EDX/RSI? = adres bufora, który otrzyma ścieżkę ESI/RDX? = długość tego bufora	EAX = długość ścieżki EAX = błąd ENAMETOOLONG, EPERM, EINVAL, ENOMEM, ERANGE, EFAULT
254	213	Utwórz deskryptor pliku epoll (sys_epoll_create)	EBX/RDI = wstępna ilość deskryptorów	EAX = nowy deskryptor pliku EAX = błąd ENOMEM
255	233	Kontroluj deskryptor pliku epoll (sys_epoll_ctl)	EBX/RDI = deskryptor epoll ECX/RSI = kod operacji EDX/RDX = deskryptor pliku	EAX = 0 EAX = błąd ENOMEM, EBADF,

			ESI/R10 = adres struktury epoll_event	EPERM, EINVAL
256	232	Czekaj na deskryptorze pliku epoll (sys_epoll_wait)	EBX/RDI = deskryptor epoll ECX/RSI = adres tablicy struktur epoll_event EDX/RDX = maksymalna liczba zdarzeń, na które będziemy czekać ESI/R10 = czas czekania w milisekundach (-1 = nieskończoność)	EAX = liczba deskrptorów gotowych EAX = błąd EFAULT, EINTR, EBADF, EINVAL
257	216	Przemapuj strony pamięci / stwórz nieliniowe mapowanie pliku (sys_remap_file_pages)	EBX/RDI = początkowy adres stron pamięci ECX/RSI = rozmiar przemapowywanego obszaru pamięci EDX/RDX = 0 (już nieużywane, musi być 0) ESI/R10 = offset w pliku mierzony w jednostkach strony systemowej EDI/R8 = flagi (znaczenie taie jak w sys_mmap, ale tu tylko MAP_NONBLOCK jest uznawane)	EAX = 0 EAX = błąd EINVAL
258	218	Utwórz wskaźnik do ID wątku (sys_set_tid_address)	EBX/RDI = wskaźnik (adres), na którego wartość ma być ustawiona zmienna clear_child_tid jądra	EAX = PID bieżącego procesu
259	222	Utwórz POSIX-owy licznik czasu (sys_timer_create)	EBX/RDI = ID zegara, który będzie podstawą mierzenia czasu ECX/RSI = 0 lub adres struktury sigevent EDX/RDX = adres zmiennej trzymającej adres DWORDa, który otrzyma ID nowego zegara	EAX = 0 EAX = błąd EAGAIN, EINVAL, ENOTSUPP
260	223	Nastaw POSIX-owy licznik czasu (sys_timer_settime)	EBX/RDI = ID zegara ECX/RSI = flagi (patrz: manual) EDX/RDX = adres struktury itimerspec ESI/R10 = adres struktury itimerspec	EAX = 0 EAX = błąd EINVAL
261	224	Pobierz pozostały czas na POSIX-owym liczniku czasu (sys_timer_gettime)	EBX/RDI = ID zegara ECX/RSI = adres struktury itimerspec, która otrzyma wynik	EAX = 0 EAX = błąd EINVAL
262	225	Pobierz liczbę przekroczeń POSIX-owego licznika czasu (sys_timer_getoverrun)	EBX/RDI = ID zegara	EAX = liczba przekroczeń EAX = błąd EINVAL
263	226	Usuń POSIX-owy licznik czasu (sys_timer_delete)	EBX/RDI = ID zegara	EAX = 0 EAX = błąd EINVAL

264	227	Ustaw czas na danym zegarze (sys_clock_settime)	EBX/RDI = ID zegara ECX/RSI = adres struktury timespec	EAX = 0 EAX = błąd EINVAL, EFAULT, EINVAL
265	228	Pobierz czas na danym zegarze (sys_clock_gettime)	EBX/RDI = ID zegara ECX/RSI = adres struktury timespec	EAX = 0 EAX = błąd EINVAL, EFAULT, EINVAL
266	229	Pobierz precyzję danego zegara (sys_clock_getres)	EBX/RDI = ID zegara ECX/RSI = adres struktury timespec	EAX = 0 EAX = błąd EINVAL, EFAULT, EINVAL
267	230	Przerwa w oparciu o dany zegar (sys_clock_nanosleep)	EBX/RDI = ID zegara ECX/RSI = flagi (TIMER_ABSTIME=1) kontorlujące rodzaj czasu oczekiwania (względny lub nie) EDX/RDX = adres struktury timespec, która zawiera czas czekania ESI/R10 = adres struktury timespec (lub NULL), która otrzyma pozostały czas	EAX = 0 EAX = błąd EINTR, EFAULT, ENOTSUPP
268*	-	Pobierz statystyki systemu plików, wersja 64-bitowa (sys_statfs64)	EBX/RDI = adres nazwy dowolnego pliku w zamontowanym systemie plików ECX/RSI adres struktury statfs64	EAX = 0 EAX = błąd
269*	-	Pobierz statystyki systemu plików, wersja 64-bitowa (sys_fstatfs64)	EBX/RDI = deskryptor dowolnego otwartego pliku w zamontowanym systemie plików ECX/RSI = adres struktury statfs64	EAX = 0 EAX = błąd
270	200	Zabij pojedynczy wątek (sys_tgkill)	EBX/RDI = GID grupy wątku (-1 daje to samo co sys_tkill) ECX/RSI = PID wątku EDX/RDX = numer sygnału do wysłania	EAX = 0 EAX = błąd EINVAL, ESRCH, EPERM
271	235	Zmień czas dostępu do pliku (sys_utimes)	EBX/RDI = adres nazwy pliku (ASCIIZ) ECX/RSI = adres tablicy 2 struktur timeval, NULL gdy chcemy bieżący czas. Pierwsza struktura opisuje czas dostępu, druga - czas zmiany	EAX = 0 EAX = błąd EACCES, ENOENT, EPERM, EROFS
272	221	Zadeklaruj wzorce dostępu (sys_fadvise64_64)	EBX/RDI = deskryptor pliku ECX/RSI = początek obszaru w pliku (offset)	EAX = 0 EAX = błąd EBADF, ESPIPE, EINVAL

			EDX/RDX = długość obszaru pliku ESI/R10 = wzorzec dostępu	
273	236	sys_vserver	niezaimplementowane w jądrach 2.4	zawsze EAX = ENOSYS
274	237	Ustaw politykę dla zakresu pamięci (sys_mbind)	EBX/RDI = adres początku obszaru ECX/RSI = długość obszaru EDX/RDX = polityka ESI/R10 = adres DWORDa zawierającego maskę bitową węzłów EDI/R8 = długość maski w bitach EBP/R9 = flagi polityki	EAX = 0 EAX = błąd EFAULT, EINVAL, ENOMEM, EIO
275	239	Pobierzy politykę pamięci NUMA (sys_get_mempolicy)	EBX/RDI = adres DWORDa, który otrzyma politykę ECX/RSI = NULL lub adres DWORDa, który otrzyma maskę węzłów EDX/RDX = maksymalna długość maski w bitach + 1 ESI/R10 = sprawdzany adres, jeśli potrzebny EDI/R8 = NULL lub MPOL_F_ADDR=2 (wtedy będzie zwrócona polityka dotycząca podanego adersu)	EAX = 0 EAX = błąd
276	238	Ustaw politykę pamięci NUMA (sys_set_mempolicy)	EBX/RDI = polityka ECX/RSI = adres DWORDa zawierającego maskę bitową węzłów EDX/RDX = długość maski w bitach	EAX = 0 EAX = błąd
277	240	Otwórz kolejkę wiadomości (sys_mq_open)	EBX/RDI = adres nazwy, która musi zaczynać się ukośnikiem ECX/RSI = flagi dostępu (EDX/RDX = tryb) (ESI/R10 = adres struktury mq_attr lub NULL)	EAX = deskryptor kolejki EAX = błąd EACCES, EINVAL, EEXIST, EMFILE, ENAMETOOLONG, ENFILE, ENOENT, ENOMEM, ENOSPC
278	241	Usuń kolejkę wiadomości (sys_mq_unlink)	EBX/RDI = adres nazwy, która musi zaczynać się ukośnikiem	EAX = 0 EAX = błąd EACCES, ENAMETOOLONG, ENOENT
279	242	Wyślij wiadomość do kolejki (sys_mq_timedsend)	EBX/RDI = deskryptor kolejki ECX/RSI = adres napisu - treści wiadomości	EAX = 0 EAX = błąd EAGAIN, EBADF, EINTR,

			EDX/RDX = długość wiadomości ESI/R10 = priorytet (<=32768, wiadomości o większym numerze są przed tymi o mniejszym) EDI/R8 = adres struktury timespec, opisującej czas BEZWZGLĘDNY przedawnienia	EINVAL, EMSGSIZE, ETIMEOUT
280	243	Odbierz wiadomość z kolejki (sys_mq_timederceive)	EBX/RDI = deskryptor kolejki ECX/RSI = adres bufora na treśc wiadomości EDX/RDX = długość bufora ESI/R10 = NULL lub adres DWORDa, który otrzyma priorytet wiadomości EDI/R8 = adres struktury timespec, opisującej czas BEZWZGLĘDNY przedawnienia	EAX = 0 EAX = błąd EAGAIN, EBADF, EINTR, EINVAL, EMSGSIZE, ETIMEOUT, EBADMSG
281	244	Powiadamianie o wiadomościach (sys_mq_notify)	EBX/RDI = deskryptor kolejki ECX/RSI = NULL (brak powiadomień) lub adres struktury sigevent	EAX = 0 EAX = błąd EBADF, EBUSY, EINVAL, ENOMEM
282	245	sys_mq_getsetattr	NIE UŻYWAĆ Interfejs do mq_getattr, mq_setattr	brak danych
283	246	sys_kexec_load	brak danych	brak danych
284	247	Czekaj na zmianę stanu innego procesu (sys_waitid)	EBX/RDI = typ identyfikatora (0=czekaj na dowolnego potomka, 1=czekaj na proces o danym PID, 2=czekaj na członka grupy o danym GID) ECX/RSI = identyfikator: PID lub GID (nieważny dla EBX/RDI=0) EDX/RDX = adres struktury siginfo ESI/R10 = opcje opisujące, na jakie zmiany czekamy	EAX = 0, wypełniona struktura siginfo EAX = błąd ECHILD, EINTR, EINVAL
285	-	sys_setaltroot	nieużywane	brak danych
286	248	sys_add_key	brak danych	brak danych
287	249	sys_request_key	brak danych	brak danych
288	250	sys_keyctl	brak danych	brak danych

289	251	Ustaw priorytet kolejkowania We/wy procesu (sys_ioprio_set)	EBX/RDI = typ ECX/RSI = informacja zależna od typu EDX/RDX = nowy priorytet	EAX = 0 EAX = błąd ESRCH, EPERM, EINVAL
290	252	Pobierz priorytet kolejkowania We/wy procesu (sys_ioprio_get)	EBX/RDI = typ ECX/RSI = informacja zależna od typu	EAX = priorytet EAX = błąd ESRCH, EPERM, EINVAL
291	253	Inicjalizacja kolejki zdarzeń inotify (sys_inotify_init)	nic	EAX = deskryptor kolejki EAX = błąd EMFILE, ENOMEM
292	254	Dodaj obiekt obserwowany kolejki zdarzeń inotify (sys_inotify_add_watch)	EBX/RDI = deskryptor kolejki inotify ECX/RSI = adres nazwy pliku ASCIIZ EDX/RDX = flagi inotify	EAX = deskryptor obserwacji danego pliku EAX = błąd EACCES, EBADF, EFAULT, EINVAL, ENOMEM, ENOSPC
293	255	Usuń obserwację obiektu z kolejki zdarzeń inotify (sys_inotify_rm_watch)	EBX/RDI = deskryptor kolejki inotify ECX/RSI = deskryptor obserwacji	EAX = 0 EAX = błąd EBADF, EINVAL
294	256	sys_migrate_pages	brak danych	brak danych
295	257	Otwórz plik względnie do katalogu (sys_openat)	EBX/RDI = deskryptor otwartego katalogu ECX/RSI = adres nazwy pliku ASCIIZ. Jeśli ścieżka jest względna, jest brana jako względna względem podanego katalogu zamiast bieżącego katalogu procesu EDX/RDX = bity dostępu ESI/R10 = prawa dostępu / tryb	EAX = 0 EAX = błąd EBADF, ENOTDIR
296	258	Utwórz katalog względnie do katalogu (sys_mkdirat)	EBX/RDI = deskryptor otwartego katalogu ECX/RSI = adres ścieżki/nazwy ASCIIZ. Jeśli ścieżka jest względna, jest	EAX = 0 EAX = błąd - każdy związany z systemem plików lub prawami

			brana jako względna względem podanego katalogu zamiast bieżącego katalogu procesu EDX/RDX = prawa dostępu / tryb	dostępu
297	259	Utworzenie pliku specjalnego względnie do katalogu (sys_mknodat)	EBX/RDI = deskryptor otwartego katalogu ECX/RSI = adres ścieżki/nazwy ASCIIZ. Jeśli ścieżka jest względna, jest brana jako względna względem podanego katalogu zamiast bieżącego katalogu procesu EDX/RDX = typ urządzenia OR prawa dostępu ESI/R10, EDI/R8 - wynik działania (zmodyfkiowanego) makra makedev	EAX = 0 EAX = błąd EACCES, EEXIST, EFAULT, EINVAL, ELOOP, ENAMETOOLONG, ENOENT, ENOMEM, ENOSPC, ENOTDIR, EPERM, EROFS
298	260	Zmiana właściciela obiektu położonego względnie do katalogu (sys_fchownat)	EBX/RDI = deskryptor otwartego katalogu ECX/RSI = adres ścieżki/nazwy ASCIIZ. Jeśli ścieżka jest względna, jest brana jako względna względem podanego katalogu zamiast bieżącego katalogu procesu EDX/RDX = identyfikator UID nowego właściciela obiektu ESI/R10 = identyfikator GID grupy, która stanie się właścicielem obiektu EDI/R8 = 0 lub wartość AT_SYMLINK_NOFOLLOW=100h, wtedy nie będzie podążał za dowiązaniami symbolicznymi	EAX = 0 EAX = błąd EBADF, ENOTDIR, EINVAL
299	261	Zmiana czasów dostępu i zmian pliku położonego względnie do katalogu (sys_futimesat)	EBX/RDI = deskryptor otwartego katalogu ECX/RSI = adres ścieżki/nazwy ASCIIZ. Jeśli ścieżka jest względna, jest brana jako względna względem podanego katalogu zamiast bieżącego katalogu procesu EDX/RDX = adres tablicy 2 struktur timeval, NULL gdy chcemy bieżący czas. Pierwsza struktura opisuje czas dostępu, druga - czas zmiany	EAX = 0 EAX = błąd EBADF, ENOTDIR
300	262	Pobierz status obiektu położonego względnie do katalogu (sys_fstatat64)	EBX/RDI = deskryptor otwartego katalogu ECX/RSI = adres ścieżki/nazwy ASCIIZ. Jeśli ścieżka jest względna, jest brana jako względna względem podanego katalogu zamiast bieżącego katalogu procesu EDX/RDX = adres struktury stat na dane	EAX = 0 EAX = błąd EBADF, ENOTDIR, EINVAL, błędy sys_stat

ESI/R10 = 0 lub
AT_SYMLINK_NOFOLLOW
(=0x100), gdy nie chcemy
dereferencjonować dowiązań

Poprzednia część (Alt+3)
Kolejna część (Alt+4)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Opis funkcji przerwania int 80h: 301-336

Jeśli jakaś funkcja zakończy się błędem, w EAX/RAX zwracana jest wartość ujemna z przedziału od -4096 do -1 włącznie.
Z drugiej strony, opisy funkcji na stronach manuala mówią, że zwracane jest -1, a wartość błędu jest zapisywana do zmiennej errno z biblioteki GLIBC. Dzieje się tak tylko w przypadku, gdy korzystamy z interfejsu języka C (czyli deklarujemy i uruchamiamy zewnętrzne funkcje odpowiadające wywołaniom systemowym i linkujemy nasz program z biblioteką języka C), a nie bezpośrednio z wywołań systemowych (czyli przerwania int 80h).

Najaktualniejsze informacje o funkcjach systemowych można znaleźć zazwyczaj w sekcji 2 (lub 3) manuala, np. man 2 open
Najnowsze wersje stron manuala można znaleźć tu: www.kernel.org/pub/linux/docs/man-pages.

Napis ASCIIZ oznacza łańcuch znaków ASCII zakończony znakiem/bajtem Zerowym.
Jeśli potrzeba, przy każdej funkcji jest odnośnik do opisu argumentów i innych dodatkowych informacji: typów danych, wartości błędów, możliwych wartości parametrów itp.

Podstawowe funkcje przewania 80h: 301-336

Numer/ EAX	x86-64 RAX	Opis	Argumenty	Zwraca
301	263	Usuń obiekt względnie do katalogu (sys_ulinkat)	EBX/RDI = deskryptor otwartego katalogu ECX/RSI = adres ścieżki/nazwy ASCIIZ. Jeśli ścieżka jest względna, jest brana jako względna względem podanego katalogu zamiast bieżącego katalogu procesu EDX/RDX = 0 lub AT_REMOVEDIR (=0x200), gdy chcemy usuwać katalogi	EAX = 0 EAX = błąd EBADF, ENOTDIR, EINVAL
302	264	Przenieś plik/Zmień nazwę pliku względnie do katalogu (sys_renameat)	EBX/RDI = deskryptor otwartego katalogu źródłowego ECX/RSI = adres starej nazwy (i ewentualnie ścieżki) ASCIIZ EDX/RDX = deskryptor otwartego katalogu docelowego ESI/R10 = adres nowej nazwy (i ewentualnie ścieżki) ASCIIZ	EAX = 0 EAX = błąd EBUSY, EEXIST, EISDIR, ENOTEMPTY, EXDEV (i inne błędy systemu plików), EBADF, ENOTDIR
303	265	Utworzenie twardego dowiązania do pliku względnie do katalogu (sys_linkat)	EBX/RDI = deskryptor otwartego katalogu źródłowego ECX/RSI = adres nazwy istniejącego pliku ASCIIZ EDX/RDX = deskryptor otwartego katalogu docelowego	EAX = 0 EAX=błąd EACCES, EIO, EPERM, EEXIST, EFAULT, ELOOP, EMLINK, ENAMETOOLONG,

			ESI/R10 = adres nazwy nowego pliku ASCIIZ EDI/R8 = 0 (flagi)	ENOENT, ENOMEM, ENOSPC, ENOTDIR, EROFS, EXDEV
304	266	Stwórz dowiązanie symboliczne do pliku względnie do katalogu (sys_symlinkat)	EBX/RDI = adres nazwy pliku ASCIIZ ECX/RSI = deskryptor otwartego katalogu docelowego EDX/RDX = adres nazwy linku ASCIIZ	EAX = 0 EAX = błędy związane z uprawnieniami lub systemem plików
305	267	Przeczytaj zawartość linku symbolicznego względnie do katalogu (sys_readlinkat)	EBX/RDI = deskryptor otwartego katalogu źródłowego ECX/RSI = adres nazwy dowiązania symbolicznego ASCIIZ EDX/RDX = adres bufora, który otrzyma przeczytaną informację ESI/R10 = długość bufora	EAX = 0 EAX = błąd EBADF, ENOTDIR
306	268	Zmiana uprawnień obiektu względnie do katalogu (sys_fchmodat)	EBX/RDI = deskryptor otwartego katalogu źródłowego ECX/RSI = adres nazwy pliku ASCIIZ EDX/RDX = nowe prawa dostępu ESI/R10 = flagi, musi być zero	EAX = 0 EAX = błąd EBADF, ENOTDIR
307	269	Sprawdź uprawnienia dostępu do pliku (sys_faccessat)	EBX/RDI = deskryptor otwartego katalogu źródłowego ECX/RSI = adres nazwy pliku ASCIIZ EDX/RDX = prawa dostępu / tryb (wartości R_OK, W_OK, X_OK) ESI/R10 = flagi: 0 lub AT_SYMLINK_NOFOLLOW=100h (nie podążaj za dowiązaniami symbolicznymi) lub AT_EACCESS=0x200 (sparwdzanie według efektywnych UID i GID)	EAX = 0 EAX = błąd EBADF, ENOTDIR, EINVAL
308	270	sys_pselect6	brak danych	brak danych
309	271	Czekaj na zdarzenia na deskryptorze (sys_ppoll)	EBX/RDI = adres tablicy struktur pollfd ECX/RSI = liczba struktur pollfd w tablicy EDX/RDX = adres struktury timespec - czas oczekiwania lub 0 (nieskończoność) ESI/R10 = adres struktury sigset_t	EAX = liczba odpowiednich deskryptorów EAX = 0, gdy czas upłynął EAX = błąd EFAULT, EINTR, EINVAL, EBADF, ENOMEM
310	272	Cofnij współdzielanie (sys_unshare)	EBX/RDI = CLONE_FILES, CLONE_FS lub CLONE_NEWNS	EAX = 0 EAX=błąd EPERM,

			spośród flag	ENOMEM, EINVAL
311	273	sys_set_robust_list	brak danych	brak danych
312	274	sys_get_robust_list	brak danych	brak danych
313	275	Spleć dane z/do potoku (sys_splice)	EBX/RDI = wejściowy deksryptor pliku ECX/RSI = offset w pliku wejściowym, skąd czytać dane (0 dla potoków) EDX/RDX = wyjściowy deksryptor pliku ESI/R10 = offset w pliku wyjściowym, dokąd zapisać dane (0 dla potoków) EDI/R8 = maksymalna liczba bajtów do przeniesienia EBP/R9 = flagi	EAX = liczba przeniesionych bajtów EAX=błąd EBADF, ESPIPE, ENOMEM, EINVAL
314	277	Synchronizuj segment pliku z dyskiem (sys_sync_file_range)	EBX/RDI = deskryptor pliku EDX:ECX/RSI? = początek zakresu danych do synchronizacji / ECX/RSI -> 64 bity adresu początku danych? EDI:ESI/RDX? = liczba bajtów do zsynchronizowania / EDX/RDX -> 64-bitowa liczba bajtów do zsynchronizowania? EBP/ESI (odpowiednio) / R10 = flagi synchronizacji	EAX = 0 EAX=błąd EBADF, EIO, EINVAL, ENOMEM, ENOSPC, ESPIPE
315	276	Duplikowanie zawartości potoku (sys_tee)	EBX/RDI = wejściowy deksryptor pliku ECX/RSI = wyjściowy deksryptor pliku EDX/RDX = maksymalna liczba bajtów do przeniesienia ESI/R10 = flagi (te same, co dla sys_splice)	EAX = liczba zduplikowanych bajtów EAX=błąd ENOMEM, EINVAL
316	278	Spleć strony pamięci do potoku (sys_vmsplice)	EBX/RDI = wejściowy deskryptor potoku ECX/RSI = adres tablicy struktur iovec EDX/RDX = liczba elementów w tablicy pod [ECX/RSI] ESI/R10 = flagi (te same, co dla sys_splice)	EAX = liczba bajtów przeniesionych do potoku EAX=błąd EBADF, ENOMEM, EINVAL
317	279	Przesuń strony pamięci procesu (sys_move_pages)	EBX/RDI = PID procesu ECX/RSI = liczba stron do przeniesienia EDX/RDX = adres tablicy adresów stron ESI/R10 = adres tablicy liczb całkowitych określających żądane miejsce dla danej strony EDI/R8 = adres tablicy na liczby	EAX = 0 EAX=błąd E2BIG, EACCES, EFAULT, EINVAL, ENODEV, ENOENT, EPERM, ESRCH

			całkowite, które otrzymają status wykonanej operacji EBP/R9 = flagi określające typ stron do przeniesienia (MPOL_MF_MOVE=2 oznacza tylko strony procesu, MPOL_MF_MOVE_ALL=4 oznacza wszystkie strony)	
318	-	Określ procesor, na którym działa wątek (sys_getcpu)	EBX/RDI = 0 lub adres zmiennej, która otrzyma numer procesora ECX/RSI = 0 lub adres zmiennej, która otrzyma numer węzła NUMA EDX/RDX = adres struktury getcpu_cache, która służyła jako pamięć podręczna dla tej funkcji; nieużywane	brak danych
319	281	Czekaj na deskryptorze pliku epoll (sys_epoll_pwait)	EBX/RDI = deskryptor epoll ECX/RSI = adres tablicy struktur epoll_event EDX/RDX = maksymalna liczba zdarzeń, na które będziemy czekać ESI/R10 = czas czekania w milisekundach (-1 = nieskończoność) EDI/R8 = adres zestawu sygnałów (tablicy 32 DWORDów), które też przerwą czekanie	EAX = liczba deskrptorów gotowych EAX = błąd EFAULT, EINTR, EBADF, EINVAL
320	280	Zmień znaczniki czasu pliku (sys_utimensat)	EBX/RDI = deskryptor otwartego katalogu ECX/RSI = adres nazwy pliku EDX/RDX = adres dwuelementowej tablicy struktur timespec, opisujących czas dostępu i zmiany ESI/R10 = flagi: 0 lub AT_SYMLINK_NOFOLLOW=0x100 (nie podążaj za dowiązaniami symbolicznymi)	EAX = 0 EAX=błąd EACCES, EBADF, EFAULT, EINVAL, ELOOP, ENAMETOOLONG, ENOENT, ENOTDIR, EPERM, EROFS, ESRCH
321	282	Stwórz deskryptor pliku do otrzymywania sygnałów (sys_signalfd)	EBX/RDI = deskryptor: -1, gdy tworzymy nowy lub nieujemny, gdy zmieniamy istniejący ECX/RSI = adres maski sygnałów (sigset), które chcemy otrzymywać	EAX = deskryptor pliku EAX=błąd EBADF, EINVAL, EMFILE, ENFILE, ENODEV, ENOMEM
322	283	Stwórz nowy czasomierz (sys_timerfd_create)	EBX/RDI = identyfikator zegara (CLOCK_REALTIME=0 lub CLOCK_MONOTONIC=1) ECX/RSI = flagi: 0 lub zORowane wartości TFD_NONBLOCK (=04000 ósemkowo, operacje mają być nieblokujące), TFD_CLOEXEC (=02000000 ósemkowo, zamknij w czasie	EAX = deskryptor pliku EAX=błąd EINVAL, EMFILE, ENFILE, ENODEV, ENOMEM

			wywołania exec)	
323	284	Stwórz deskryptor pliku do otrzymywania zdarzeń (sys_eventfd)	EBX/RDI = wstępna wartość licznika zdarzeń	EAX = deskryptor pliku EAX=błąd EINVAL, EMFILE, ENFILE, ENODEV, ENOMEM
324	285	Manipulacja przestrzenią pliku (sys_fallocate)	EBX/RDI = deskryptor pliku ECX/RSI = tryb = FALLOC_FL_KEEP_SIZE (alokuje i zeruje przestrzeń na dysku w podanym zakresie bajtów) = 1 EDX/RDX = offset początku zakresu bajtów w pliku ESI/R10 = ilość bajtów w zakresie	EAX = liczba deskrptorów gotowych EAX = błąd EFAULT, EINTR, EBADF, EINVAL
325	286	Uruchom lub zatrzymaj czasomierz (sys_timerfd_settime)	EBX/RDI = deskryptor pliku czasomierza ECX/RSI = flagi (0 uruchamia czasomierz względy, TFD_TIMER_ABSTIME=1 - bezwzględny) EDX/RDX = adres struktury itimerspec, zawierającej początkowy czas ESI/R10 = adres struktury itimerspec, która otrzyma aktualny czas	EAX = 0 EAX=błąd EINVAL, EFAULT, EBADF
326	287	Pobierz czas na czasomierzu (sys_timerfd_gettime)	EBX/RDI = deskryptor pliku czasomierza ECX/RSI = adres struktury itimerspec, która otrzyma aktualny czas	EAX = 0 EAX=błąd EINVAL, EFAULT, EBADF
327	289	Stwórz deskryptor pliku do otrzymywania sygnałów (sys_signalfd4)	EBX/RDI = deskryptor: -1, gdy tworzymy nowy lub nieujemny, gdy zmieniamy istniejący ECX/RSI = adres maski sygnałów (sigset), które chcemy otrzymywać EDX/RDX = flagi: 0 lub zORowane wartości SFD_NONBLOCK (=04000 ósemkowo, operacje mają być nieblokujące), SFD_CLOEXEC (=02000000 ósemkowo, zamknij w czasie wywołania exec)	EAX = deskryptor pliku EAX=błąd EBADF, EINVAL, EMFILE, ENFILE, ENODEV, ENOMEM
328	290	Stwórz deskryptor pliku do otrzymywania zdarzeń (sys_eventfd2)	EBX/RDI = wstępna wartość licznika zdarzeń ECX/RSI = flagi: 0 lub zORowane wartości EFD_NONBLOCK (=04000 ósemkowo, operacje mają być nieblokujące), EFD_CLOEXEC (=02000000 ósemkowo, zamknij w czasie wywołania exec)	EAX = deskryptor pliku EAX=błąd EINVAL, EMFILE, ENFILE, ENODEV, ENOMEM
329	291			

		Utwórz deskryptor pliku epoll (sys_epoll_create1)	EBX/RDI = flagi: 0 lub EPOLL_CLOEXEC (=02000000 ósemkowo, zamknij w czasie wywołania exec)	EAX = nowy deskryptor pliku EAX = błąd ENOMEM, EINVAL, EMFILE, ENFILE
330	292	Zamień deskryptor zduplikowanym deskryptorem pliku (sys_dup3)	EBX/RDI = deskryptor do zduplikowania ECX/RSI = deskryptor, do którego powinien być przyznany duplikat EDX/RDX = flagi: 0 lub O_CLOEXEC (=02000000 ósemkowo, zamknij w czasie wywołania exec)	EAX = zduplikowany deskryptor EAX = błąd EBADF, EMFILE, EBUSY, EINTR, EINVAL
331	293	Utwórz potok (sys_pipe2)	EBX/RDI = adres tablicy dwóch DWORDów ECX/RSI = flagi: 0 lub zORowane wartości O_NONBLOCK (=04000 ósemkowo, operacje mają być nieblokujące), O_CLOEXEC (=02000000 ósemkowo, zamknij w czasie wywołania exec)	EAX = 0 i pod [EBX/RDI]: deskryptor odczytu z potoku fd(0) pod [EBX/RDI], deskryptor zapisu do potoku fd(1) pod [EBX/RDI+4] EAX = błąd EFAULT, EMFILE, ENFILE, EINVAL
332	294	Inicjalizacja kolejki zdarzeń inotify (sys_inotify_init1)	EBX/RDI = flagi: 0 lub zORowane wartości IN_NONBLOCK (=04000 ósemkowo, operacje mają być nieblokujące), IN_CLOEXEC (=02000000 ósemkowo, zamknij w czasie wywołania exec)	EAX = deskryptor kolejki EAX = błąd EMFILE, ENFILE, ENOMEM, EINVAL
333	295	sys_preadv	brak danych	brak danych
334	296	sys_pwritev	brak danych	brak danych
335	297	sys_rt_tgsigqueueinfo	brak danych	brak danych
336	298	sys_perf_counter_open	brak danych	brak danych

Poprzednia część (Alt+3)
Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Dodatkowe typy danych, wartości stałych systemowych.

Należy przyjąć, że "int", "long", "PID", "clock_t", "time_t", "u(int)32" są typu DWORD, zaś "short" jest typu WORD.

Struktura "pt_regs" (funkcja 2) z /usr/include/asm/ptrace.h:

```
struct pt_regs {
        long ebx;
        long ecx;
        long edx;
        long esi;
        long edi;
        long ebp;
        long eax;
        int  xds;
        int  xes;
        long orig_eax;
        long eip;
        int  xcs;
        long eflags;
        long esp;
        int  xss;
};
```

Bity dostępu (funkcje 5 i 277) z /usr/include/asm/fcntl.h

nazwa	ósemkowo	komentarz
O_ACCMODE	3	Pełne prawa dostępu
O_RDONLY	0	Otwieranie tylko do odczytu. Dostępne dla sys_mq_open.
O_WRONLY	1	Otwieranie tylko do zapisu. Dostępne dla sys_mq_open.
O_RDWR	2	Otwieranie do odczytu i zapisu. Dostępne dla sys_mq_open.
O_CREAT	100	Utworzenie pliku. Dostępne dla sys_mq_open.
O_EXCL	200	Nie twórz pliku, jeśli już istnieje. Dostępne dla sys_mq_open.
O_NOCTTY	400	Jeśli podana nazwa pliku to terminal, to NIE zostanie on terminalem kontrolnym procesu.
O_TRUNC	1000	Obcięcie pliku
O_APPEND	2000	Dopisywanie do pliku
O_NONBLOCK	4000	Nie otwieraj, jeśli spowoduje to blokadę. Dostępne dla sys_mq_open.
O_NDELAY	O_NONBLOCK	jak wyżej
O_SYNC	10000	specyficzne dla ext2 i urządzeń blokowych
FASYNC	20000	fcntl, dla zgodności z BSD
O_DIRECT	40000	podpowiedź bezpośredniego dostępu do dysku, obecnie ignorowana
O_LARGEFILE	100000	Pozwól na otwieranie plików >4GB

O_DIRECTORY	200000	musi być katalogiem
O_NOFOLLOW	400000	nie podążaj za linkami

Prawa dostępu / tryb (funkcje 5, 8, 14, 15 i 277) z /usr/include/linux/stat.h

nazwa	ósemkowo	komentarz
S_ISUID	4000	ustaw ID użytkownika przy wykonywaniu (suid)
S_ISGID	2000	ustaw ID grupy przy wykonywaniu (sgid)
S_ISVTX	1000	"sticky bit" - usuwać z takiego katalogu może tylko właściciel
S_IRUSR	400	czytanie przez właściciela (S_IREAD)
S_IWUSR	200	zapis przez właściciela (S_IWRITE)
S_IXUSR	100	wykonywanie/przeszukiwanie katalogu przez właściciela (S_IEXEC)
S_IRGRP	40	czytanie przez grupę
S_IWGRP	20	zapis przez grupę
S_IXGRP	10	wykonywanie/przeszukiwanie katalogu przez grupę
S_IROTH	4	czytanie przez innych (R_OK)
S_IWOTH	2	zapis przez innych (W_OK)
S_IWOTH	1	wykonywanie/przeszukiwanie katalogu przez innych (X_OK)
S_IRWXUGO	(S_IRWXU | S_IRWXG | S_IRWXO)	czytanie, pisanie i wykonywanie przez wszystkich
S_IALLUGO	(S_ISUID | S_ISGID | S_ISVTX | S_IRWXUGO)	czytanie, pisanie i wykonywanie przez wszystkich + suid + sgid + sticky bit
S_IRUGO	(S_IRUSR | S_IRGRP | S_IROTH)	czytanie dla wszystkich
S_IWUGO	(S_IWUSR | S_IWGRP | S_IWOTH)	zapis dla wszystkich
S_IXUGO	(S_IXUSR | S_IXGRP | S_IXOTH)	wykonywanie/przeszukiwanie katalogu dla wszystkich

---

Flagi montowania (funkcja 21) z /usr/include/linux/fs.h

nazwa	wartość	komentarz
MS_MGC_MSK	0xC0ED	in m.s. 16-bit; 'magic', niezbędne przed 2.4.0-t9 (????)
MS_RDONLY	1	Zamontuj tylko do odczytu
MS_NOSUID	2	Ignoruj bity suid i sgid
MS_NODEV	4	Zabroń dostępu do specjalnych plików/urządzeń
MS_NOEXEC	8	Zabroń wykonywania programów
MS_SYNCHRONOUS	16	Zapisy są synchronizowane natychmiast
MS_REMOUNT	32	

		Zmień flagi zamontowanego systemu plików (przemontuj z innymi atrybutami)
MS_MANDLOCK	64	Pozwól na nakazane blokady na systemie plików (???)
MS_NOATIME	1024	Nie zmieniaj czasów dostępu
MS_NODIRATIME	2048	Nie zmieniaj czasów dostępu do katalogów
MS_BIND	4096	Montowanie bindowane - widoczne w innym miejscu systemu plików
MS_REC	16384	
MS_VERBOSE	32768	
MS_ACTIVE	(1<<30)	
MS_NOUSER	(1<<31)	

Przechodzenie do innego miejsca w pliku (funkcja 19 i 140) z /usr/include/fcntl.h

nazwa	wartość	znaczenie
SEEK_SET	0	Przesuwanie zaczyna się od początku pliku
SEEK_CUR	1	Przesuwanie zaczyna się od bieżącej pozycji
SEEK_END	2	Przesuwanie zaczyna się od końca pliku

Tak zwana specyfikacja procesu (funkcja 7 i 37)

PID	dany sygnał zostanie wysłany do
> 0	procesu potomnego o tym PID
0	każdego procesu potomnego w grupie procesów wywołującego
-1	wszystkich procesów potomnych z wyjątkiem pierwszego
< -1	wszystkich procesów potomnych w grupie |{gid}|

Żądane działanie (funkcja 26) z /usr/include/linux/ptrace.h

PTRACE_...	wartość	argumenty	zwraca
PEEKTEXT	0	PID, adres, wskaźnik na dane	czytaj (d)word pod podanym adresem
PEEKDATA	1	PID, adres, wskaźnik na dane	czytaj (d)word pod podanym adresem w obszarze pamięci Użytkownika
PEEKUSR	2	PID, adres, wskaźnik na dane	czytaj (d)word pod podanym adresem w obszarze pamięci Użytkownika
POKETEXT	3	PID, adres, wskaźnik na dane	zapisz (d)word pod podanym adresem
POKEDATA	4	PID, adres, wskaźnik na dane	zapisz (d)word pod podanym adresem
POKEUSR	5	PID, adres, wskaźnik na dane	zapisz (d)word pod podanym adresem w obszarze pamięci Użytkownika

TRACEME	6	-	Ustaw bit PTRACE we flagach procesu
CONT	7	PID, - , numer sygnału	Uruchom ponownie po sygnale
KILL	8	PID	wyślij sygnał SIGKILL do procesu potomnego, kończąc go
SINGLESTEP	9	PID	Ustaw flagę TRAP, sygnał SIGTRAP
GETREGS	12	PID, -, wskaźnik na dane	pobierz wartości rejestrów procesu potomnego
SETREGS	13	PID, -, wskaźnik na dane	ustaw wartości rejestrów procesu potomnego
GETFPREGS	14	PID, -, wskaźnik na dane	pobierz stan FPU procesu potomnego
SETFPREGS	15	PID, -, wskaźnik na dane	ustal stan FPU procesu potomnego
ATTACH	16	PID, -, -	przyłącz proces do już uruchomionego procesu.
DETACH	17	PID, -, -	odłącz wcześniej przyłączony proces
(K4)GETFPXREGS	18	PID, -, wskaźnik na dane	pobierz rozszerzony stan FPU procesu potomnego
(K4)SETFPXREGS	19	PID, -, wskaźnik na dane	ustal rozszerzony stan FPU procesu potomnego
(K4)SETOPTIONS	21	PID, -, dane	Opcja PTRACE_O_TRACESYSGOOD lub żadna
SYSCALL	24	PID, -, numer sygnału	Kontynuuj i zatrzymaj się na następnym powrocie z danego sygnału

Rejestry są ustawione tak:
EBX, ECX, EDX, ESI, EDI, EBP, EAX, DS, ES, FS, GS, ORIG_EAX, EIP, CS, EFL, UESP, SS, FRAME_SIZE.

Wartości błędów zwracanych przez funkcje systemowe Linuksa. Te numery można znaleźć także w:

- mojej bibliotece - załącznik "bibl/incl/..../...._system.inc"
- /usr/src/linux/include/asm/errors.h
- man errno (tylko część)

"Prawdziwe" zwracane wartości błędów są *przeciwnego znaku* (np. EIO = -5).

```
%define EPERM            1      ; Operacja niedozwolona
%define ENOENT           2      ; Nie ma takiego pliku/katalogu
%define ESRCH            3      ; Nie ma takiego procesu
%define EINTR            4      ; Przerwana funkcja systemowa
%define EIO              5      ; Błąd I/O
%define ENXIO            6      ; Nie ma takiego urządzenia/adresu
%define E2BIG            7      ; Za długa lista argumentów
%define ENOEXEC          8      ; Błąd formatu wykonywalnego
%define EBADF            9      ; Zły numer pliku
%define ECHILD          10      ; Nie ma procesów potomnych
%define EAGAIN          11      ; Zasoby chwilowo niedostępne.
%define ENOMEM          12      ; Brak pamięci
%define EACCES          13      ; Odmowa dostępu
```

```
%define EFAULT          14      ; Zły adres
%define ENOTBLK         15      ; Wymagane jest urządzenie blokowe
%define EBUSY           16      ; Urządzenie/zasób zajęty
%define EEXIST          17      ; Plik istnieje
%define EXDEV           18      ; Dowiązanie międzyurządzeniowe
%define ENODEV          19      ; Nie ma takiego urządzenia
%define ENOTDIR         20      ; To nie jest katalog
%define EISDIR          21      ; To jest katalog
%define EINVAL          22      ; Nieprawidłowy argument
%define ENFILE          23      ; Przepełnienie tablicy plików
%define EMFILE          24      ; Za dużo otwartych plików
%define ENOTTY          25      ; Nieodpowiednia operacja kontroli
                                ; wejścia/wyjścia (Not a typewriter)
%define ETXTBSY         26      ; Plik tekstowy zajęty
%define EFBIG           27      ; Plik za duży
%define ENOSPC          28      ; Brak miejsca na urządzeniu
%define ESPIPE          29      ; Nieprawidłowa zmiana pozycji w pliku
%define EROFS           30      ; System plików tylko do odczytu
%define EMLINK          31      ; Za dużo linków
%define EPIPE           32      ; Zły potok
%define EDOM            33      ; Argument poza dziedziną funkcji matemat.
%define ERANGE          34      ; Wyniku nie da się przedstawić
%define EDEADLK         35      ; Uniknięto zakleszczenia zasobów
                                ; (Resource deadlock would occur)
%define ENAMETOOLONG    36      ; Zbyt długa nazwa pliku
%define ENOLCK          37      ; Brak dostępnych blokad
%define ENOSYS          38      ; Funkcja nie zaimplementowana
%define ENOTEMPTY       39      ; Katalog nie jest pusty
%define ELOOP           40      ; Zbyt dużo linków symbolicznych
%define EWOULDBLOCK     EAGAIN  ; Zasoby chwilowo niedostępne (operacja
                                ;   zablokowałaby program)
%define ENOMSG          42      ; Nie ma wiadomości żądanego typu
%define EIDRM           43      ; Identyfikator usunięty
%define ECHRNG          44      ; Numer kanału spoza zasięgu
%define EL2NSYNC        45      ; Poziom 2 nie zsynchronizowany
%define EL3HLT          46      ; Poziom 3 zatrzymany
%define EL3RST          47      ; Poziom 3 ponownie uruchomiony
%define ELNRNG          48      ; Za duża liczba linków/numer linku
%define EUNATCH         49      ; Niepodłączony sterownik protokołu
%define ENOCSI          50      ; Brak wolnych struktur CSI
%define EL2HLT          51      ; Poziom 2 zatrzymany
%define EBADE           52      ; Nieprawidłowa wymiana
%define EBADR           53      ; Nieprawidłowy deskryptor żądania
%define EXFULL          54      ; Wymiana pełan (bufor?)
%define ENOANO          55      ; No anode
%define EBADRQC         56      ; Nieprawidłowy kod zadania
%define EBADSLT         57      ; Invalid slot

%define EDEADLOCK       EDEADLK

%define EBFONT          59      ; Nieprawidłowy format pliku czcionki
%define ENOSTR          60      ; Urządzenie nie jest strumieniem
%define ENODATA         61      ; Nie ma danych
%define ETIME           62      ; Przekroczony limit czasu
%define ENOSR           63      ; Braz zasobów strumieniowych
%define ENONET          64      ; Komputer nie jest w sieci
%define ENOPKG          65      ; Pakiet nie zainstalowany
%define EREMOTE         66      ; Obiekt jest zdalny
%define ENOLINK         67      ; Link has been severed
%define EADV            68      ; Advertise error
%define ESRMNT          69      ; Srmount error
%define ECOMM           70      ; Błąd komunikacji przy wysyłaniu
```

```
%define EPROTO          71      ; Błąd protokołu
%define EMULTIHOP       72      ; Multihop attempted
%define EDOTDOT         73      ; Błąd specyficzny dla RFS
%define EBADMSG         74      ; To nie jest wiadomość z danymi
%define EOVERFLOW       75      ; Wartość za duża dla okresłonego typu
%define ENOTUNIQ        76      ; Nazwa nie jest unikalna w sieci
%define EBADFD          77      ; Deskryptor pliku w złym stanie
%define EREMCHG         78      ; Zmiana adresu zdalnego
%define ELIBACC         79      ; Nie mozna sie dostać do wymaganej bibl.
%define ELIBBAD         80      ; Dostęp do zepsutej
                                ;     biblioteki współdzielonej
%define ELIBSCN         81      ; Zepsuta sekcja .lib w a.out
%define ELIBMAX         82      ; Próba podłączenia zbyt wielu bibliotek
                                ;     współdzielonych
%define ELIBEXEC        83      ; Nie można bezpośrednio uruchamiać
                                ;     biblioteki współdzielonej
%define EILSEQ          84      ; Nieprawidłowa sekwencja bajtów
%define ERESTART        85      ; Przerwana funkcja systemowa powinna zostać
                                ;     uruchomiona ponownie
%define ESTRPIPE        86      ; Streams pipe error
%define EUSERS          87      ; Za dużo użytkowników
%define ENOTSOCK        88      ; Operacja gniazdowa na nie-gnieździe
%define EDESTADDRREQ    89      ; Wymagany adres docelowy
%define EMSGSIZE        90      ; Wiadomość za długa
%define EPROTOTYPE      91      ; Zły typ protokołu dla gniazda
%define ENOPROTOOPT     92      ; Protokół niedostepny
%define EPROTONOSUPPORT 93      ; Protokół nieobsługiwany
%define ESOCKTNOSUPPORT 94      ; Typ gniazda nieobsługiwany
%define EOPNOTSUPP      95      ; Operacja nie obsługiwana po drugiej
                                ;     stronie transportu
%define EPFNOSUPPORT    96      ; Nieobsługiwana rodzina protokołów
%define EAFNOSUPPORT    97      ; Rodzina adresów nie obsługiwana przez
                                ;     ten protokół
%define EADDRINUSE      98      ; Adres już jest używany
%define EADDRNOTAVAIL   99      ; Nie można przydzielić żądanego adresu
%define ENETDOWN        100     ; Sieć nie działa
%define ENETUNREACH     101     ; Sieć jest niedostępna
%define ENETRESET       102     ; Brak sieci z powodu resetu
%define ECONNABORTED    103     ; Przerwanie połączenia przez program
%define ECONNRESET      104     ; Reset połączenia przez drugą stronę
%define ENOBUFS         105     ; Brak miejsca w buforze
%define EISCONN         106     ; Druga strona transportu już jest
                                ;     podłączona
%define ENOTCONN        107     ; Druga strona transportu nie jest
                                ;     podłączona
%define ESHUTDOWN       108     ; Nie można wysyłać po wyłączeniu z drugiej
                                ;     strony transportu
%define ETOOMANYREFS    109     ; Too many references: cannot splice
%define ETIMEDOUT       110     ; Przekroczony limit czasu połączenia
%define ECONNREFUSED    111     ; Odmowa połączenia
%define EHOSTDOWN       112     ; Host jest wyłączony
%define EHOSTUNREACH    113     ; Nie ma drogi do hosta
%define EALREADY        114     ; Operacja już trwa
%define EINPROGRESS     115     ; Operation trwa teraz
%define ESTALE          116     ; Stale NFS file handle
%define EUCLEAN         117     ; Struktura wymaga porządkowania
%define ENOTNAM         118     ; Not a XENIX named type file
%define ENAVAIL         119     ; No XENIX semaphores available
%define EISNAM          120     ; Is a named type file
%define EREMOTEIO       121     ; Błąd zdalnego I/O
%define EDQUOT          122     ; Przekroczony limit dyskowy (Quota)
```

```
%define ENOMEDIUM       123     ; Brak nośnika
%define EMEDIUMTYPE     124     ; Zły typ nośnika
%define ECANCELED       125     ; Zrezygnowano z operacji
%define ENOKEY          126     ; Wymagany klucz niedostępny
%define EKEYEXPIRED     127     ; Klucz przedawniony
%define EKEYREVOKED     128     ; Klucz nieważny
%define EKEYREJECTED    129     ; Klucz odrzucony

%define EOWNERDEAD      130     ; Właścicel zginął
%define ENOTRECOVERABLE 131     ; State not recoverable


%define ERESTARTSYS     512
%define ERESTARTNOINTR  513
%define ERESTARTNOHAND  514     ; restart if no handler..
%define ENOIOCTLCMD     515     ; No ioctl command
%define ERESTART_RESTARTBLOCK 516 ; restart by calling sys_restart_syscall

; NFS v3
%define EBADHANDLE      521     ; Illegal NFS file handle
%define ENOTSYNC        522     ; Update synchronization mismatch
%define EBADCOOKIE      523     ; Cookie is stale
%define ENOTSUPP        524     ; Operacja nieobsługiwana
%define ETOOSMALL       525     ; Bufor lub żądanie za małe
%define ESERVERFAULT    526     ; An untranslatable error occurred
%define EBADTYPE        527     ; Typ nieobsługiwany przez serwer
%define EJUKEBOX        528     ; Request initiated, but will not
                                ;     complete before timeout
%define EIOCBQUEUED     529     ; iocb queued, will get completion event
%define EIOCBRETRY      530     ; iocb queued, will trigger a retry

#define EWOULDBLOCKIO   530     ; Would block due to block-IO
```

Struktura "tms" (funkcja 43) z /usr/include/linux/times.h:

```
        struct tms {
                clock_t tms_utime;
                clock_t tms_stime;
                clock_t tms_cutime;
                clock_t tms_cstime;
            };
```

Struktura "flock" z /usr/include/asm/fcntl.h:

```
        struct flock {
                short l_type;   // rodzaj blokady pliku (WORD)

                short l_whence; // patrz SEEK_* powyżej? (WORD)

                off_t l_start;  // adres miejsca, do którego odnosi
                                // się blokada, w bajtach liczonych
                                // od pozycji określonej
                                // przez l_whence (DWORD)

                off_t l_len;    // długość zablokowanego obszaru.
                                // Zero oznacza do końca pliku (DWORD)

                pid_t l_pid;    // otrzymane komendą F_GETLK,
```

```
                        // ignorowane przy włączaniu blokady (DWORD)
};
```

Argumenty funkcji sys_fcntl (numer 55) z /usr/include/bits/fcntl.h

ECX (komenda)	Wartość	Opis
F_DUPFD	0	EDX staje się kopią deskryptora z EBX
F_GETFD	1	Odczytaj flagę close-on-exec. Gdy bit0=0, plik zostanie otwarty pomimo exec, inaczej plik zostanie zamknięty.
F_SETFD	2	Ustaw flagę close-on-exec na wartość ostatniego bitu w EDX
F_GETFL	3	Zwróć takie flagi deskryptora, jakie były ustawione przez funkcję open
F_SETFL	4	Ustaw flagi deskryptora na wartość EDX. Można ustawić tylko O_APPEND i O_NONBLOCK.
F_GETLK	5	[Jeśli uruchamiasz sys_fcntl64, użyj wersji 64-bitowej] EDX ma adres struktury flock. Zwróć strukturę flock, która chroni nas przed uzyskaniem blokady lub ustaw pole l_type blokady na F_UNLCK jeśli możliwe
F_SETLK	6	[Jeśli uruchamiasz sys_fcntl64, użyj wersji 64-bitowej] EDX ma adres struktury flock. Blokada jest ustawiana jeśli l_type jest F_RDLCK lub F_WRLCK albo usuwana, gdy jest F_UNLCK. Jeśli blokada jest przechowywana przez kogoś innego, zwraca -1 i EACCES lub EAGAIN.
F_SETLKW	7	[Jeśli uruchamiasz sys_fcntl64, użyj wersji 64-bitowej] Podobnie jak F_SETLK, ale zamiast zwracać błąd, czeka na zwolnienie blokady.
F_SETOWN	8	Ustaw proces, będący właścicielem gniazda (socket). Tutaj i w poniższej funkcji grupy procesów zwracane są jako wartości ujemne.
F_GETOWN	9	Pobierz ID procesu, będącego właścicielem gniazda.
F_SETSIG	10	Ustaw numer sygnału, który ma zostać wysłany.
F_GETSIG	11	Pobierz numer sygnału, który ma zostać wysłany.
F_GETLK64	12	EDX ma adres struktury flock. Zwróć strukturę flock, która chroni nas przed uzyskaniem blokady lub ustaw pole l_type locka na F_UNLCK jeśli możliwe
F_SETLK64	13	EDX ma adres struktury flock. Blokada jest ustawiana jeśli l_type jest F_RDLCK lub F_WRLCK albo usuwana, gdy jest F_UNLCK. Jeśli blokada jest przechowywana przez kogoś innego, zwraca -1 i EACCES lub EAGAIN.
F_SETLKW64	14	Podobnie jak F_SETLK, ale zamiast zwracać błąd, czeka na zwolnienie blokady.
F_SETLEASE	1024	Ustaw dzierżawę.
F_GETLEASE	1025	Pobierz informację, jaka dzierżawa jest aktywna.
F_NOTIFY	1026	Żądaj powiadomień na danym katalogu.

Struktura sigaction (funkcja 67) z /usr/include/asm/signal.h:

```
struct sigaction {
        __sighandler_t sa_handler;      //procedura obsługi sygnału
        unsigned long sa_flags;
        void (*sa_restorer)(void);      // procedura przywracająca?
```

```
        sigset_t sa_mask;               // maska utrzymywana dla
                                        // rozszerzalności
    };

typedef void (*__sighandler_t)(int);    // typ procedury
                                        // obsługi sygnału

#define _NSIG           64
#define _NSIG_BPW       32
#define _NSIG_WORDS     (_NSIG/_NSIG_BPW)

typedef struct {
        unsigned long sig[_NSIG_WORDS];
    } sigset_t;                         // definicja typu tej
                                        // maski powyżej
```

Struktura rlimit (funkcje 75 i 76) z /usr/include/linux/resource.h:

```
struct rlimit {
        unsigned long rlim_cur; // limit "miękki" lub
                                // RLIM_INFINITY=-1 jeśli brak
        unsigned long rlim_max; // maksymalny limit lub
                                // RLIM_INFINITY jeśli brak
    };
```

Tabela numerów zasobów z /usr/include/bits/resource.h

nazwa	wartość	co oznacza
RLIMIT_CPU	0	limit czasu procesora w sekundach
RLIMIT_FSIZE	1	rozmiar w bajtach największego pliku możliwego do utworzenia
RLIMIT_DATA	2	maksymalny rozmiar w bajtach wszystkich segmentów danych
RLIMIT_STACK	3	maksymalny rozmiar stosu w bajtach
RLIMIT_CORE	4	maksymalny rozmiar rdzenia (core) w bajtach
RLIMIT_RSS	5	maksymalny rozmiar kodu rezydentnego
RLIMIT_NPROC	6	maksymalna liczba procesów dla danego rzeczywistego PID
RLIMIT_OFILE, RLIMIT_NOFILE	7	maksymalna liczba otwartych deskryptorów + 1
RLIMIT_MEMLOCK	8	maksymalna liczba bajtów pamięci, którą można zablokować (sys_mlock)
RLIMIT_AS	9	limit przestrzeni adresowej
RLIMIT_LOCKS	10	limit blokad plików (sys_flock itp.)
RLIMIT_SIGPENDING	11	maksymalna liczba czekających sygnałów
RLIMIT_MSGQUEUE	12	maksymalna liczba bajtów w kolejkach wiadomości
RLIMIT_NICE	13	maksymalny priorytet sys_nice, jaki można ustawić
RLIMIT_RTPRIO	14	maksymalny priorytet czasu rzeczywistego, jaki można ustawić dla nieuprzywilejowanego procesu

---

Struktura rusage (funkcja 77) z /usr/include/linux/resource.h:

```
#define     RUSAGE_SELF     0
#define     RUSAGE_CHILDREN (-1)
#define     RUSAGE_BOTH     (-2)    // sys_wait4() używa tego

struct      rusage {
    struct timeval ru_utime; // wykorzystany czas użytkownika
    struct timeval ru_stime; // wykorzystany czas systemu
    long    ru_maxrss;      // rozmiar maksymalnego
                            // rezydentnego zbioru?
    long    ru_ixrss;       // rozmiar współdzielonej pamięci
    long    ru_idrss;       // rozmiar niewspółdzielonej pamięci
    long    ru_isrss;       // rozmiar niewspółdzielonego stosu
    long    ru_minflt;      // ilość odzyskanych stron pamięci
    long    ru_majflt;      // ilość błędów stron pamięci
    long    ru_nswap;       // wymiany (swaps)
    long    ru_inblock;     // blokujące operacje wejścia
    long    ru_oublock;     // blokujące operacje wyjścia
    long    ru_msgsnd;      // wysłane wiadomości
    long    ru_msgrcv;      // odebrane wiadomości
    long    ru_nsignals;    // odebrane sygnały
    long    ru_nvcsw;       // dobrowolne zmiany kontekstu
    long    ru_nivcsw;      // niedobrowolne zmiany kontekstu
};
```

Struktura timeval (funkcja 78) z /usr/include/linux/time.h:

```
struct timeval {
        time_t          tv_sec;         // dword, sekundy
        suseconds_t     tv_usec;        // dword, mikrosekundy
    };
```

Struktura timezone (funkcja 78) z /usr/include/linux/time.h:

```
struct timezone {
        int     tz_minuteswest; // minuty na zachód od Greenwich
        int     tz_dsttime;     // typ poprawki punktu docelowego?
        };
```

Struktura sigset_t (funkcje 72 i 73) z /usr/include/asm/signal.h:

```
#define _NSIG           64
#define _NSIG_BPW       32
#define _NSIG_WORDS     (_NSIG/_NSIG_BPW)

typedef struct {
        unsigned long sig[_NSIG_WORDS];
    } sigset_t;
```

Komendy dla funkcji sys_ulimit (numer 58) z /usr/include/ulimit.h

nazwa	wartość	komenda
UL_GETFSIZE	1	pobierz limit rozmiaru pliku, w 512-bajtowych jednostkach
UL_SETFSIZE	2	ustaw limit rozmiaru pliku, w 512-bajtowych jednostkach
__UL_GETMAXBRK	3	(podobno nie w Linuksie) podaj najwyższy możliwy adres w segmencie danych
__UL_GETOPENMAX	4	podaj największą liczbę plików, którą ten proces może otworzyć

Typ urządzenia dla funkcji sys_mknod (numer 14) z /usr/include/linux/stat.h

nazwa	ósemkowo	typ urządzenia
S_IFREG	0100000	normalny plik
S_IFCHR	0020000	urządzenie znakowe
S_IFBLK	0060000	urządzenie blokowe
S_IFIFO	0010000	nazwany potok (named pipe)

Struktura "utimbuf" (funkcja 30) z /usr/include/linux/utime.h:

```
struct utimbuf {
        time_t actime;  /* czas dostępu */
        time_t modtime; /* czas ostatniej zmiany */
};
```

Struktura "timeb" (funkcja 35) z /usr/include/sys/timeb.h:

```
struct timeb {
        time_t   time;          /*liczba sekund od początku epoki*/
        unsigned short millitm; /* liczba milisekund od chwili
                                   time sekund */
        short    timezone;      /* przesuniecie czasu dla strefy
                                   lokalnej w minutach na zachód
                                    od Greenwich */
        short    dstflag;       /* różne od zera oznacza, że w
                                   danej części roku obowiązuje
                                   czas letni */
         };
```

Struktura "ustat" (funkcja 62) z /usr/include/bits/ustat.h:

```
struct ustat {
        __daddr_t f_tfree; /* całkowita liczba wolnych bloków */
```

```
    __ino_t f_tinode;  /* liczba wolnych węzłów i-node */
    char f_fname[6];   /* nazwa systemu plików (nieużywane) */
    char f_fpack[6];   /* nazwa paczki(?) systemu plików
                          (nieużywane) */
      };
```

Flagi dla funkcji swapon (numer 87) z /usr/include/sys/swap.h:

```
/* Priorytet przestrzeni wymiany swap jest kodowany tak:
   (prio << SWAP_FLAG_PRIO_SHIFT) & SWAP_FLAG_PRIO_MASK
*/
#define SWAP_FLAG_PREFER        0x8000  /* Ustawiony, jeśli jest określony
                                           priorytet. Ustawienie tego bitu
                                           powoduje to, że nowa przestrzeń
                                           wymiany będzie miała wyższy
                                           priorytet */
#define SWAP_FLAG_PRIO_MASK     0x7fff
#define SWAP_FLAG_PRIO_SHIFT    0
```

Flagi dla funkcji reboot (numer 88) z /usr/include/linux/reboot.h

nazwa	wartość	co oznacza
LINUX_REBOOT_CMD_RESTART	0x01234567	reset systemu
LINUX_REBOOT_CMD_HALT	0xCDEF0123	zatrzymanie systemu
LINUX_REBOOT_CMD_CAD_ON	0x89ABCDEF	włączenie obsługi Ctrl-Alt-Del
LINUX_REBOOT_CMD_CAD_OFF	0x00000000	wyłączenie obsługi Ctrl-Alt-Del
LINUX_REBOOT_CMD_POWER_OFF	0x4321FEDC	wyłączenie zasilania
LINUX_REBOOT_CMD_RESTART2	0xA1B2C3D4	reset z użyciem komendy podanej jako dodatkowy argument

---

Struktura "dirent" (funkcja 89 i 141) z /usr/include/linux/dirent.h:

```
        struct dirent {
                long d_ino;                 /* numer i-węzła */
                off_t d_off;                /* offset od początku katalogu
                                            do następnej struktury dirent
                                            (do bieżącego dirent?) */
                unsigned short d_reclen;    /* długość tego dirent
                                                (d_name?) */
                char d_name [NAME_MAX+1];   /* nazwa pliku (zakończona
                                                znakiem zerowym) */
              };
```

Rodzaje ochrony mapowanych danych (funkcja 90) z /usr/include/bits/mman.h

nazwa	wartość	co oznacza
PROT_READ	1	strona pamięci może być czytana
PROT_WRITE	2	strona może być zapisywana
PROT_EXEC	4	strona może być wykonywana
PROT_NONE	0	nie ma dostępu do strony

Można użyć OR, by połączyć więcej flag.

Flagi mapowania danych (funkcja 90) z /usr/include/bits/mman.h

nazwa	wartość	co oznacza
MAP_FIXED	0x10	Koniecznie użyj adresu podanego jako parametr
MAP_GROWSDOWN	0x0100	Segment typu stosowego ("rośnie" w dół)
MAP_DENYWRITE	0x0800	Ignorowane
MAP_EXECUTABLE	0x1000	Wykonywalny (ignorowane)
MAP_LOCKED	0x2000	Zablokuj mapowanie. Ignorowane.
MAP_NORESERVE	0x4000	Nie rezerwuj stron wymiany swap dla tego mapowania.
MAP_POPULATE	0x8000	Rozmnóż tablice stron?
MAP_NONBLOCK	0x10000	Nie blokuj w czasie we/wy.
MAP_FILE	0	użyj pliku? Ignorowane.
MAP_ANONYMOUS, MAP_ANON	0x20	Nie używaj pliku. Ignorowane są deskryptor pliku i parametr offset. Zaimplementowany od 2.4

Wybrać tylko 1 spośród tych:
Rodzaje współdzielenia

nazwa	wartość	co oznacza
MAP_SHARED	0x01	współdzielenie zmian
MAP_PRIVATE	0x02	zmiany są prywatne

Parametry funkcji 96 i 97 (z /usr/include/bits/resource.h)

EBX = czyj priorytet pobieramy/zmieniamy	wartość w EBX	co oznacza
PRIO_PROCESS	0	ECX to ID procesu
PRIO_PGRP	1	ECX to ID grupy procesów
PRIO_USER	2	ECX to ID użytkownika

Struktura "statfs" (funkcja 99) z /usr/include/asm/statfs.h:

```
struct statfs {
        long    f_type;     /* typ systemu plików (patrz niżej) */
```

```
    long    f_bsize;    /* optymalny rozmiar bloku transferu */
    long    f_blocks;   /* całkowita ilość bloków danych */
    long    f_bfree;    /* wolne bloki */
    long    f_bavail;   /* wolne bloki dostępne dla
                           nie-superużytkowników */
    long    f_files;    /* całkowita ilość węzłów plików
file nodes) */           (
    long    f_ffree;    /* wolne węzły */
    fsid_t  f_fsid;     /* ID systemu plików */
    long    f_namelen;  /* maks. długość nazwy pliku */
    long    f_spare[6]; /* zapasowe na później */
  };

    struct
  {
    int __val[2];
  } __fsid_t;               /* typ struktury fsid_t powyżej */
```

Typ systemu plików

nazwa	wartość
AFFS_SUPER_MAGIC	0xADFF
EXT_SUPER_MAGIC	0x137D
EXT2_OLD_SUPER_MAGIC	0xEF51
EXT2_SUPER_MAGIC	0xEF53
HPFS_SUPER_MAGIC	0xF995E849
ISOFS_SUPER_MAGIC	0x9660
MINIX_SUPER_MAGIC	0x137F (oryg. minix)
MINIX_SUPER_MAGIC2	0x138F (30-znakowy minix)
MINIX2_SUPER_MAGIC	0x2468 (minix V2)
MINIX2_SUPER_MAGIC2	0x2478 (minix V2, 30-znakowy)
MSDOS_SUPER_MAGIC	0x4d44
NCP_SUPER_MAGIC	0x564c
NFS_SUPER_MAGIC	0x6969
PROC_SUPER_MAGIC	0x9fa0
SMB_SUPER_MAGIC	0x517B
XENIX_SUPER_MAGIC	0x012FF7B4
SYSV4_SUPER_MAGIC	0x012FF7B5
SYSV2_SUPER_MAGIC	0x012FF7B6
COH_SUPER_MAGIC	0x012FF7B7
UFS_MAGIC	0x00011954
_XIAFS_SUPER_MAGIC	0x012FD16D

Komenda dla funkcji sys_syslog (numer 103) z /usr/src/linux/kernel/printk.c

wartość	komentarz
0	Zamknij log. Nic nie robi.
1	Otwórz log. Nic nie robi.
2	Czytaj z logu co najwyżej EDX bajtów do [ECX]. Zwraca w EAX ilość odczytanych bajtów.
3	Przeczytaj wszystkie (ostatnie EDX bajtów) wiadomości pozostałe w buforze. Zwraca w EAX ilość odczytanych bajtów.
4	Przeczytaj i wyczyść wszystkie (ostatnie EDX bajtów) wiadomości pozostałe w buforze. Zwraca w EAX ilość odczytanych bajtów.
5	Wyczyść bufor.
6	Wyłącz funkcję printk() na konsolę.
7	Włącz funkcję printk() na konsolę.
8	Ustal poziom logowania wiadomości wysyłanych na konsolę.

Numer czasomierza (funkcja 104) z /usr/include/linux/time.h

nazwa	wartość	komentarz
ITIMER_REAL	0	odlicza czas rzeczywisty
ITIMER_VIRTUAL	1	odlicza czas wykonywania się procesu
ITIMER_PROF	2	odlicza oba czasy

Struktura "itimerval" (funkcja 104) z /usr/include/linux/time.h:

```
struct itimerval {
        struct timeval it_interval; /* następna wartość? */
        struct timeval it_value;    /* obecna wartość */
    };

struct timeval {
        long tv_sec;                /* sekundy */
        long tv_usec;               /* mikrosekundy */
    };
```

Struktura "stat" (funkcja 104) z man 2 stat (i /usr/include/asm/stat.h):

```
struct stat {
    dev_t         st_dev;      /* urządzenie */
    ino_t         st_ino;      /* i-węzeł (inode) */
    umode_t       st_mode;     /* ochrona */
    nlink_t       st_nlink;    /* liczba dowiązań stałych
                                  (hardlinks) */
    uid_t         st_uid;      /*ID użytkownika właściciela*/
    gid_t         st_gid;      /* ID grupy właściciela */
    dev_t         st_rdev;     /* typ urządzenia (jeśli
                                  urządzenie inode) */
    off_t         st_size;     /* całkowity rozmiar
                                  w bajtach */
    unsigned long st_blksize;  /* wielkość bloku dla I/O
                                  systemu plików */
```

```
    unsigned long st_blocks;   /*ilość zaalokowanych bloków*/
    time_t        st_atime;    /* czas ostatniego dostępu */
    time_t        st_mtime;    /*czas ostatniej modyfikacji*/
    time_t        st_ctime;    /* czas ostatniej zmiany */
};
```

Możliwe wartości pola st_mode

nazwa	ósemkowo	co oznacza
S_IFMT	00170000	maska bitowa dla pól bitowych typu pliku
S_IFSOCK	0140000	gniazdo
S_IFLNK	0120000	dowiązanie symboliczne (symbolic link)
S_IFREG	0100000	plik regularny
S_IFBLK	0060000	urządzenie blokowe
S_IFDIR	0040000	katalog
S_IFCHR	0020000	urządzenie znakowe
S_IFIFO	0010000	fifo
S_ISUID	0004000	bit 'set UID'
S_ISGID	0002000	bit 'set GID'
S_ISVTX	0001000	bit 'sticky'
S_IRWXU	00700	użytkownik (właściciel pliku) ma prawa odczytu, zapisu i wykonania
S_IRUSR (S_IREAD)	00400	użytkownik ma prawa odczytu
S_IWUSR (S_IWRITE)	00200	użytkownik ma prawa zapisu
S_IXUSR (S_IEXEC)	00100	użytkownik ma prawa wykonania
S_IRWXG	0070	grupa ma prawa odczytu, zapisu i wykonania
S_IRGRP (S_IREAD)	0040	grupa ma prawa odczytu
S_IWGRP (S_IWRITE)	0020	grupa ma prawa zapisu
S_IXGRP (S_IEXEC)	0010	użytkownik ma prawa wykonania
S_IRWXO	007	inni mają prawa odczytu, zapisu i wykonania
S_IROTH (S_IREAD)	004	inni mają prawa odczytu
S_IWOTH (S_IWRITE)	002	inni mają prawa zapisu
S_IXOTH (S_IEXEC)	001	inni mają prawa wykonania

Struktura "vm86_struct" (funkcja 113) z /usr/include/asm/vm86.h:

```
struct vm86_struct {
        struct vm86_regs regs;
        unsigned long flags;
        unsigned long screen_bitmap;
        unsigned long cpu_type;
        struct revectored_struct int_revectored;
        struct revectored_struct int21_revectored;
};
```

```
struct vm86_regs {
/*
 * normalne rejestry, ze specjalnym znaczeniem dla
        rej.segmentowych i deskryptorów
 */
        long ebx;
        long ecx;
        long edx;
        long esi;
        long edi;
        long ebp;
        long eax;
        long __null_ds;
        long __null_es;
        long __null_fs;
        long __null_gs;
        long orig_eax;
        long eip;
        unsigned short cs, __csh;
        long eflags;
        long esp;
        unsigned short ss, __ssh;
/*
 * te są specyficzne dla trybu v86:
 */
        unsigned short es, __esh;
        unsigned short ds, __dsh;
        unsigned short fs, __fsh;
        unsigned short gs, __gsh;
};

struct revectored_struct {
        unsigned long __map[8];            /* 256 bitów */
};
```

Opcje dla funkcji typu "wait" (numer 7 i 114) to wartość 0 lub jedna lub więcej tych opcji (z /usr/include/bits/waitflags.h):

Opcje dla funkcji typu "wait"

nazwa	wartość	co oznacza
WNOHANG	1	nie blokuj czekania, wraca natychmiast, gdy żaden potomek się nie zakończył. Tylko dla sys_wait/pid
WUNTRACED	2	podaj status zatrzymanych procesów potomnych. Tylko dla sys_wait, sys_waitpid
WCONTINUED	8	czekaj na zatrzymanych potomków, wznowionych sygnałem SIGCONT
WSTOPPED	2	czekaj na potomków zatrzymanych przez sygnał. Tylko dla sys_waitid
WEXITED	4	czekaj na zakończenie potomków. Tylko dla sys_waitid
WNOWAIT	0x01000000	tylko pobierz status. Tylko dla sys_waitid
__WNOTHREAD	0x20000000	nie czekaj na potomków innych wątków grupy. Tylko dla sys_waitid
__WALL	0x40000000	czekaj na któregokolwiek z potomków. Tylko dla sys_waitid
__WCLONE	0x80000000	czekaj na sklonowane procesy. Tylko dla sys_waitid

Struktura sysinfo (funkcja 116) z /usr/include/linux/kernel.h:

```
(pre 2.3.16, wszystkie rozmiary w bajtach):

struct sysinfo {
           long uptime;             /* ilość sekund od startu
                                       systemu */
           unsigned long loads[3];  /* średnie obciążenie w ciągu
                                       1, 5 i 15 minut */
           unsigned long totalram;  /* ilość pamięci */
           unsigned long freeram;   /* ilość wolnej pamięci */
           unsigned long sharedram; /* ilość pamięci wspólnej */
           unsigned long bufferram; /* pamięć wykorzystywana
                                       przez bufory */
           unsigned long totalswap; /* ilość pamięci wymiany */
           unsigned long freeswap;  /* ilość wolnej
                                       pamięci wymiany */
           unsigned short procs;    /* ilość procesów */
           char _f[22];             /* dopełnienie do 64 bajtów */
      };

(od 2.3.48, rozmiary w krotnościach mem_unit?)

struct sysinfo {
           long uptime;             /* ilość sekund od startu
                                       systemu */
           unsigned long loads[3];  /* średnie obciążenie w ciągu
                                       1, 5 i 15 minut */
           unsigned long totalram;  /* ilość pamięci */
           unsigned long freeram;   /* ilość wolnej pamięci */
           unsigned long sharedram; /* ilość pamięci wspólnej */
           unsigned long bufferram; /* pamięć wykorzystywana
                                        przez bufory */
           unsigned long totalswap; /* ilość pamięci wymiany */
           unsigned long freeswap;  /* ilość wolnej
                                       pamięci wymiany */
           unsigned short procs;    /* ilość procesów */
           unsigned long totalhigh; /* ilość pamięci wysokiej */
           unsigned long freehigh;  /* ilość wolnej
                                       pamięci wysokiej */
           unsigned int mem_unit;   /* wielkość jednostki pamięci
                                       w bajtach */
                       /* dopełnienie dla libc5 */
            char _f[20-2*sizeof(long)-sizeof(int)];
      };
```

Flagi dla funkcji 120 (dla jądra 2.4.18?) z /usr/include/linux/sched.h:

Flagi klonowania

nazwa	wartość	co oznacza
CSIGNAL	0x000000ff	maska sygnałów do wysłania przy wychodzeniu

CLONE_VM	0x00000100	gdy VM jest dzielone między procesy
CLONE_FS	0x00000200	gdy info o systemie plików jest dzielone między procesy
CLONE_FILES	0x00000400	gdy otwarte pliki są dzielone między procesy
CLONE_SIGHAND	0x00000800	gdy dzielone są procedury obsługi sygnałów i blokowane sygnały
CLONE_PID	0x00001000	gdy PID jest dzielony między procesy
CLONE_PTRACE	0x00002000	jeśli chcemy, aby klon też mógł być śledzony
CLONE_VFORK	0x00004000	jeśli klonujący chce, by proces potomny go obudził przy mm_release
CLONE_PARENT	0x00008000	jeśli klon ma mieć tego samego rodzica, co klonujący
CLONE_THREAD	0x00010000	Ta sama grupa wątków?
CLONE_NEWNS	0x00020000	Nowa grupa przestrzeni nazw??
CLONE_SIGNAL	(CLONE_SIGHAND | CLONE_THREAD)	Połączenie tych dwóch: ta sama grupa wątków oraz gdy dzielone są procedury obsługi sygnałów i blokowane sygnały

---

Struktura utsname (funkcja 122) z /usr/include/sys/utsname.h:

```
#define _UTSNAME_LENGTH 65      /* wszystkie tablice poniżej
                                        są tej długości */

struct utsname  {
        /* Nazwa implementacji systemu operacyjnego.  */
        char sysname[_UTSNAME_SYSNAME_LENGTH];

        /* Nazwa tego komputera w sieci.  */
        char nodename[_UTSNAME_NODENAME_LENGTH];

        /* Wydanie (release) tej implementacji.  */
        char release[_UTSNAME_RELEASE_LENGTH];
        /* Wersja tego wydania.  */
        char version[_UTSNAME_VERSION_LENGTH];

        /* Nazwa sprzętu, na który system pracuje.  */
        char machine[_UTSNAME_MACHINE_LENGTH];

#if _UTSNAME_DOMAIN_LENGTH - 0
    /* Nazwa domeny tego komputera w sieci.  */
# ifdef __USE_GNU
    char domainname[_UTSNAME_DOMAIN_LENGTH];
# else
    char __domainname[_UTSNAME_DOMAIN_LENGTH];
# endif
#endif
  };
```

Numery funkcji dla modify_ldt (funkcja 123)

Funkcje zmiany Lokalnej Tablicy Deskryptorów

wartość	co oznacza
0	Czytaj LDT do [ECX], EDX bajtów
1	Zmień 1 wpis w LDT. ECX ma adres struktury modify_ldt_ldt_s, a EDX - jej rozmiar
2	Czytaj domyślne LDT do [ECX], EDX bajtów
17	Zmień 1 wpis w LDT. ECX ma adres struktury modify_ldt_ldt_s, a EDX - jej rozmiar (?)

Struktura "modify_ldt_ldt_s" (funkcja 123) z /usr/include/asm/ldt.h:

```
struct modify_ldt_ldt_s {
        unsigned int  entry_number;
        unsigned long base_addr;
        unsigned int  limit;
        unsigned int  seg_32bit:1;
        unsigned int  contents:2;
        unsigned int  read_exec_only:1;
        unsigned int  limit_in_pages:1;
        unsigned int  seg_not_present:1;
        unsigned int  useable:1;
};
```

Struktura "timex" (funkcja 124) z man 2 adjtimex (w /usr/include/linux/timex.h jest trochę większa):

```
struct timex {
        int modes;           /* przełącznik trybu */
        long offset;         /* offset czasu (mikrosekundy) */
        long frequency;      /* offset częstotliwości
                                (skalowany ppm) */
        long maxerror;       /* maksymalny błąd (mikrosekundy) */
        long esterror;       /* obliczony błąd (mikrosekundy) */
        int status;          /* komenda/status zegara */
        long constant;       /* stała czasu pll */
        long precision;      /* dokładność zegara (mikrosekundy)
                                (tylko do odczytu) */
        long tolerance;      /* tolerancja częstotliwości
                                zegara (ppm) (tylko do odczytu) */
        struct timeval time; /* aktualny czas (tylko do odczytu) */
        long tick;           /* czas miedzy tyknięciami
                                zegara (mikrosekundy) */
      };
```

Pole "modes" określa, które parametry (jeśli w ogóle) ustawić. Może ono zawierać bitowe OR kombinacji zera lub więcej spośród następujących wartości:

Wartości pola "moeds"

nazwa	wartość	opis
ADJ_OFFSET	0x0001	offset czasu
ADJ_FREQUENCY	0x0002	offset częstotliwości

ADJ_MAXERROR	0x0004	maksymalny błąd czasu
ADJ_ESTERROR	0x0008	obliczany błąd czasu
ADJ_STATUS	0x0010	status zegara
ADJ_TIMECONST	0x0020	stała czasu pll
ADJ_TICK	0x4000	wartość tyknięcia
ADJ_OFFSET_SINGLESHOT	0x8001	staromodne adjtime

Zwyczajni użytkownicy są ograniczeni do wartości zero dla "modes". Jedynie superużytkownik może ustawiać jakiekolwiek parametry. Jeśli nie wystąpił błąd, zwracane jest:
Możliwe wyniki działania i stan zegara

nazwa	wartość	opis
TIME_OK	0	zegar zsynchronizowany
TIME_INS	1	dodaj sekundę przestępną
TIME_DEL	2	skasuj sekundę przestępną
TIME_OOP	3	sekunda przestępna trwa
TIME_WAIT	4	wystąpiła sekunda przestępna
TIME_BAD	5	błąd, zegar nie zsynchronizowany

Więcej informacji w /usr/include/linux/timex.h.

Akcja do wykonania (funkcja 126) z /usr/include/asm/signal.h:
Możliwe zmiany bieżącego zestawu sygnałów blokowanych

nazwa	wartość	opis
SIG_BLOCK	0	Do aktualnego zestawu sygnałów blokowanych dodaj te spod [ECX].
SIG_UNBLOCK	1	Od aktualnego zestawu sygnałów blokowanych usuń te spod [ECX].
SIG_SETMASK	2	Aktualny zestaw sygnałów blokowanych zamień na ten spod [ECX].

Struktura "module" (funkcja 128) z man module:

```
struct module {
        unsigned long size_of_struct;
        struct module *next;
        const char *name;
        unsigned long size;
        long usecount;
        unsigned long flags;
        unsigned int nsyms;
        unsigned int ndeps;
        struct module_symbol *syms;
        struct module_ref *deps;
        struct module_ref *refs;
        int (*init)(void);
        void (*cleanup)(void);
        const struct exception_table_entry *ex_table_start;
        const struct exception_table_entry *ex_table_end;
      #ifdef __alpha__
        unsigned long gp;
      #endif
      };
```

Więcej w /usr/include/linux/module.h.

Struktura "kernel_sym" (funkcja 130) z /usr/include/linux/module.h:

```
struct kernel_sym {
        unsigned long value;
        char name[60];
};
```

Komenda do wykonania (funkcja 131) z /usr/include/sys/quota.h (w nawiasach z /usr/include/linux/quota.h): Opcje limitów dyskowych

nazwa	wartość	opis
Q_QUOTAON	0x0100 (0x800002)	Włącz limity dyskowe. ESI = adres nazwy pliku zawierającego limity.
Q_QUOTAOFF	0x0200 (0x800003)	Wyłącz limity. EDX i ESI ignorowane.
Q_GETQUOTA	0x0300 (0x800007)	Pobierz limity i bieżące zapełnienie dla użytkownika/grupy EDX. ESI = adres struktury mem_dqblk.
Q_SETQUOTA	0x0400 (0x800008)	Ustaw limity i bieżące zapełnienie dla użytkownika/grupy EDX. ESI = adres struktury mem_dqblk.
Q_SETQLIM	0x0700 (brak)	Ustaw limity dla użytkownika/grupy EDX. ESI = adres struktury mem_dqblk.
Q_SETUSE	0x0500 (brak)	Ustaw bieżące zapełnienie dla użytkownika/grupy EDX. ESI = adres struktury mem_dqblk.
Q_SYNC	0x0600 (0x800001)	Aktualizuj kopię quot dla systemu plików. Jeśli ECX=0, synchronizowane są wszystkie systemy plików z włączoną quotą. EDX i ESI ignorowane.
Q_GETSTATS	0x0800	Pobierz statystyki i ogólne informacje o quocie. ESI = adres struktury dqstats. ECX i EDX ignorowane.
Q_GETINFO	brak (0x800005)	Pobierz info o pliku z quotami?. ESI = adres struktury mem_dqinfo. EDX ignorowane.
Q_SETINFO	brak (0x800006)	Ustal info o pliku z quotami?. ESI = adres struktury mem_dqinfo. EDX ignorowane.
Q_SETGRACE	brak (brak)	Ustal "grace times" w pliku z quotami?. ESI = adres struktury mem_dqinfo. EDX ignorowane.
Q_SETFLAGS	brak (brak)	Ustal flagi w informacji o pliku z quotami?. ESI = adres struktury mem_dqinfo. EDX ignorowane.

Na systemie plików XFS komendy są inne.

```
struct mem_dqblk {
        __u32 dqb_bhardlimit;   /* bezwzględny limit zajętych
                                        bloków na dysku */
        __u32 dqb_bsoftlimit;   /* preferowany limit zajętych
                                        bloków na dysku */
        qsize_t dqb_curspace;   /* bieżący rozmiar zajmowanej
                                        przestrzeni */
        __u32 dqb_ihardlimit;   /* bezwzględny limit zajętych
                                       węzłów (i-nodes) na dysku*/
```

```
        __u32 dqb_isoftlimit;   /* preferowany limit zajętych
                                        węzłów na dysku */
        __u32 dqb_curinodes;    /*bieżąca liczba zajętych węzłów*/
        time_t dqb_btime;       /* limit czasu nadmiernego
                                        użycia dysku */
        time_t dqb_itime;       /* limit czasu nadmiernego
                                        użycia węzła */
};

struct mem_dqinfo {
        struct quota_format_type *dqi_format;
        int dqi_flags;
        unsigned int dqi_bgrace;
        unsigned int dqi_igrace;
        union {
                struct v1_mem_dqinfo v1_i;
                struct v2_mem_dqinfo v2_i;
        } u;
};

struct dqstats {
        int lookups;
        int drops;
        int reads;
        int writes;
        int cache_hits;
        int allocated_dquots;
        int free_dquots;
        int syncs;
};
```

Komenda do wykonania (funkcja 134):
Komendy demona bdflush

wartość EBX	opis
<= 0	jeśli demon nie był uruchomiony, to funkcja wchodzi w kod demona i nigdy nie powraca.
1	Niektóre bufory są zapisywane na dysk.
>=2 i jest parzyste	ECX = adres DWORDa, pod [ECX] zostaje zwrócony parametr dostrajający równy (EBX-2)/2
>=3 i jest nieparzyste	ECX = DWORD, jądro nadaje tę wartość parametrowi dostrajającemu o numerze (EBX-3)/2

Opcje dla sysfs (funkcja 135):
Operacje na nazwach systemów plików

EBX	opis	ECX i EDX	co zwraca
1	Tłumacz nazwę systemu plików na numer	ECX = adres łańcucha znaków zawierającego nazwę.	EAX = numer systemu plików
2	Tłumacz numer systemu plików na nazwę	ECX = numer systemu plików EDX = adres bufora na nazwę.	EAX = 0

3	Zwróć ogólną liczbę systemów plików aktualnie obecnych w jądrze.	ignorowane	EAX = liczba systemów plików

---

Operacja dla sys_flock (funkcja 143) z /usr/include/asm/fcntl.h:
Opcje blokad plików

nazwa	wartość	opis
LOCK_SH	1	Założenie blokady współdzielonej.
LOCK_EX	2	Założenie blokady wyłącznej.
LOCK_UN	8	Usunięcie blokady założonej przez ten proces

Po zORowaniu z wartością LOCK_NB=4, funkcja nie zablokuje działania programu.

Flagi dla sys_msync (funkcja 144) z /usr/include/asm/mman.h:
Możliwości synchronizacji zapisu pamięci

nazwa	wartość	opis
MS_ASYNC	1	Wykonaj zapisy asynchroniczne.
MS_INVALIDATE	2	Zaznacz dane jako nieważne po zapisaniu
MS_SYNC	4	Wykonaj zapisy synchroniczne.

Struktura iovec (funkcja 145) z /usr/include/bits/uio.h:

```
struct iovec {
        void *iov_base;      /* adres danych */
        size_t iov_len;      /* długość danych */
};
```

Struktura sysctl_args (funkcja 149) z man 2 sysctl:

```
struct __sysctl_args {
        int *name;          /* wektor liczb całkowitych
                               opisujący zmienną */
        int nlen;           /* długość tego wektora */
        void *oldval;       /* 0 lub adres, gdzie zachować
                               starą wartość */
        size_t *oldlenp;    /* ilość miejsca na starą wartość
                               nadpisywana przez rzeczywisty
                               jej rozmiar */
        void *newval;       /* 0 lub adres nowej wartości */
        size_t newlen;      /* rozmiar nowej wartości */
};
```

Flagi dla sys_mlockall (funkcja 152) z /usr/include/bits/mman.h:
Blokowanie wszystkich stron pamięci procesu

nazwa	wartość	opis
MCL_CURRENT	1	Zablokuj wszystkie strony pamięci w przestrzeni adresowej procesu.
MCL_FUTURE	2	Zablokuj wszystkie strony pamięci w przestrzeni adresowej procesu w przyszłości, w chwili mapowania ich do przestrzeni procesu.

Struktura sched_param (funkcja 154) z /usr/include/bits/sched.h:

```
struct sched_param {
        int __sched_priority;
};
```

Polityka dla szeregowania zadań (funkcje 156,157,159,160) z /usr/include/bits/sched.h:
Sposoby szeregowania zadań

nazwa	wartość	opis
SCHED_OTHER	0	Domyślny sposób szeregowania zadań
SCHED_FIFO	1	Pierwszy na wejściu, pierwszy na wyjściu
SCHED_RR	2	Szeregowanie cykliczne

Struktura timespec (funkcja 162 i inne) z man nanosleep:

```
struct timespec {
        time_t  tv_sec;         /* sekundy */
        long    tv_nsec;        /* nanosekundy */
};
```

Flagi dla funkcji sys_mremap (numer 163) z /usr/include/linux/mman.h:
Możliwości remapowania pamięci

nazwa	wartość	opis
MREMAP_MAYMOVE	1	Można przenosić te strony
MREMAP_FIXED	2	Nie można przenosić stron.

Kody funkcji dla funkcji sys_vm86 (numer 166) z /usr/include/asm/vm86.h:
Funkcje Trybu wirtualnego 8086

nazwa	wartość
VM86_PLUS_INSTALL_CHECK	0
VM86_ENTER	1
VM86_ENTER_NO_BYPASS	2
VM86_REQUEST_IRQ	3

VM86_FREE_IRQ	4
VM86_GET_IRQ_BITS	5
VM86_GET_AND_RESET_IRQ	6

Struktura "vm86plus_struct" (funkcja 113) z /usr/include/asm/vm86.h:

```
struct vm86plus_struct {
        struct vm86_regs regs;
        unsigned long flags;
        unsigned long screen_bitmap;
        unsigned long cpu_type;
        struct revectored_struct int_revectored;
        struct revectored_struct int21_revectored;
        struct vm86plus_info_struct vm86plus;
};

struct vm86plus_info_struct {
        unsigned long force_return_for_pic:1;
        unsigned long vm86dbg_active:1;       /* dla debuggera */
        unsigned long vm86dbg_TFpendig:1;     /* dla debuggera */
        unsigned long unused:28;
        unsigned long is_vm86pus:1;           /* do użytku
                                              wewnętrznego trybu vm86*/
        unsigned char vm86dbg_intxxtab[32];   /* dla debuggera */
};
```

Numery podfunkcji dla funkcji sys_query_module (numer 167) z /usr/include/linux/module.h:
Odpytywanie modułów jądra

nazwa	wartość	co zwraca
brak	0	zawsze sukces
QM_MODULES	1	bufor: nazwy oddzielone znakiem zerowym [EDI] = liczba modułów
QM_DEPS	2	w buforze: nazwy modułów używane przez podany moduł, [EDI] = ilość takich modułów
QM_REFS	3	w buforze: nazwy modułów używające podanego modułu, [EDI] = ilość takich modułów
QM_SYMBOLS	4	bufor: eksportowane symbole i wartości. Format: struktury module_symbol (patrz niżej) i nazwy oddzielone znakiem zerowym. [EDI] = liczba symboli
QM_INFO	5	Format bufora: struktury module_info (patrz niżej) [EDI] = rozmiar struktury module_info

```
struct module_symbol    {
        unsigned long value;
        unsigned long name;     /* adres łańcucha znaków od
                                   początku bufora */
};

struct module_info      {
        unsigned long address;  /* adres modułu */
```

```
        unsigned long size;     /* zajmowana pamięć */
        unsigned long flags;    /* stan: MOD_RUNNING,
                                   MOD_AUTOCLEAN, ... */
};
```

Struktura "pollfd" dla funkcji sys_poll (numer 168) z man poll:

```
struct pollfd {
        int fd;         /* deskryptor otwartego pliku */
        short events;   /* maska zdarzeń (patrz niżej) do
                           monitorowania */
        short revents;  /* powrotna maska zdarzeń znalezionych
                           (patrz niżej) */
};
```

Zdarzenia z /usr/include/sys/poll.h:

Zdarzenia dla funkcji sys_poll

nazwa	wartość	co oznacza
POLLIN	0x0001	mogą być czytane (bez blokowania) dane o priorytecie innym niż wysoki
POLLPRI	0x0002	mogą być czytane dane o priorytecie wysokim
POLLOUT	0x0004	mogą być zapisywane dane normalne
POLLWRNORM	POLLOUT	jak POLLOUT.
POLLERR	0x0008	błąd
POLLHUP	0x0010	rozłączenie
POLLNVAL	0x0020	deskryptor jest nieprawidłowy
POLLRDNORM	0x0040	mogą być czytane dane normalne
POLLNORM	POLLRDNORM	jak POLLRDNORM.
POLLRDBAND	0x0080	mogą być czytane dane priorytetowe
POLLWRBAND	0x0100	mogą być zapisywane dane priorytetowe

Komendy w funkcji sys_nfsservctl (numer 169) z man nfsservctl i /usr/include/linux/nfsd/syscall.h:
Komendy kontroli serwera NFS

nazwa	wartość	co oznacza
NFSCTL_SVC	0	to jest proces serwera
NFSCTL_ADDCLIENT	1	dodanie klienta NFS
NFSCTL_DELCLIENT	2	usunięcie klienta NFS
NFSCTL_EXPORT	3	eksportowanie systemu plików
NFSCTL_UNEXPORT	4	zaprzestanie eksportowania systemu plików
NFSCTL_UGIDUPDATE	5	uaktualnienie mapy uid/gid klienta
NFSCTL_GETFH	6	otrzymanie fh przez ino (używane przez mountd)
NFSCTL_GETFD	7	otrzymanie fh przez ścieżkę (używane przez mountd)

NFSCTL_GETFS	8	otrzymanie fh przez ścieżkę z maksymalną długością FH
NFSCTL_FODROP	50	odrzuć żądania w czasie awarii
NFSCTL_STOPFODROP	51	przestań odrzucać żądania
NFSCTL_FOLOCKS	52	porzuć blokady w czasie awarii
NFSCTL_FOGRACE	53	set grace period for failover
NFSCTL_FOSERV	54	remove service mon for failover

Struktura "nfsctl_arg" i unia "nfsctl_res" dla funkcji sys_nfsservctl (numer 169) z man nfsservctl:

```
struct nfsctl_arg {
        int                     ca_version;     /*zabezpieczenie*/
        union {
                struct nfsctl_svc       u_svc;
                struct nfsctl_client    u_client;
                struct nfsctl_export    u_export;
                struct nfsctl_uidmap    u_umap;
                struct nfsctl_fhparm    u_getfh;
                struct nfsctl_fdparm    u_getfd;
                struct nfsctl_fsparm    u_getfs;
                struct nfsctl_fodrop    u_fodrop;
        } u;
}

union nfsctl_res {
        __u8                    cr_getfh[NFS_FHSIZE];
        struct knfsd_fh         cr_getfs;
};

/* SVC */
struct nfsctl_svc {
        unsigned short          svc_port;
        int                     svc_nthreads;
};

/* ADDCLIENT/DELCLIENT */
struct nfsctl_client {
        char                    cl_ident[NFSCLNT_IDMAX+1];
        int                     cl_naddr;
        struct in_addr          cl_addrlist[NFSCLNT_ADDRMAX];
        int                     cl_fhkeytype;
        int                     cl_fhkeylen;
        unsigned char           cl_fhkey[NFSCLNT_KEYMAX];
};

/* EXPORT/UNEXPORT */
struct nfsctl_export {
        char                    ex_client[NFSCLNT_IDMAX+1];
        char                    ex_path[NFS_MAXPATHLEN+1];
        __kernel_dev_t          ex_dev;
        __kernel_ino_t          ex_ino;
        int                     ex_flags;
        __kernel_uid_t          ex_anon_uid;
        __kernel_gid_t          ex_anon_gid;
};

/* UGIDUPDATE */
struct nfsctl_uidmap {
        char *                  ug_ident;
        __kernel_uid_t          ug_uidbase;
        int                     ug_uidlen;
```

```
        __kernel_uid_t *        ug_udimap;
        __kernel_gid_t          ug_gidbase;
        int                     ug_gidlen;
        __kernel_gid_t *        ug_gdimap;
};

/* GETFH */
struct nfsctl_fhparm {
        struct sockaddr         gf_addr;
        __kernel_dev_t          gf_dev;
        __kernel_ino_t          gf_ino;
        int                     gf_version;
};

/* GETFD */
struct nfsctl_fdparm {
        struct sockaddr         gd_addr;
        char                    gd_path[NFS_MAXPATHLEN+1];
        int                     gd_version;
};

/* GETFS - Pobierz uchwyt do pliku wraz z rozmiarem */
struct nfsctl_fsparm {
        struct sockaddr         gd_addr;
        char                    gd_path[NFS_MAXPATHLEN+1];
        int                     gd_maxlen;
};

/* FODROP/STOPFODROP */
struct nfsctl_fodrop {
        char                    fo_dev[NFS_MAXPATHLEN+1];
        __u32                   fo_timeout;
};
```

Opcje w funkcji sys_prctl (numer 172) z man prctl i /usr/include/linux/prctl.h:
Operacje na procesie

nazwa	wartość	co oznacza
PR_SET_PDEATHSIG	1	ECX=numer sygnału, który otrzyma proces potomny po zakończeniu rodzica
PR_GET_PDEATHSIG	2	wczytaj bieżący numer sygnału, który otrzyma proces potomny po zakończeniu rodzica do [ECX]
PR_GET_DUMPABLE	3	pobranie informacji, czy program ma zrzucać rdzeń (core dump), zwraca w EAX
PR_SET_DUMPABLE	4	ustawienie, czy program ma zrzucać rdzeń (core dump) ECX=0 (nie) ECX=1 (tak)
PR_GET_UNALIGN	5	pobierz bity kontroli dostępu do nieułożonych danych? (unaligned access control bits), wynik w EAX?
PR_SET_UNALIGN	6	ustaw bity kontroli dostępu do nieułożonych danych (unaligned access control bits) ECX=1 (nie rób nic), ECX=2 (generuj sygnał SIGBUS)
PR_GET_KEEPCAPS	7	zachowanie możliwości procesu (keep capabilities), zwraca w EAX
PR_SET_KEEPCAPS	8	zachowanie możliwości procesu (keep capabilities), ECX=1 (tak) ECX=0 (nie)

PR_GET_FPEMU	9	pobierz bity kontroli emulacji FPU, zwraca w EAX?
PR_SET_FPEMU	10	ustaw bity kontroli emulacji FPU, ECX=1 (emulacja włączona) ECX=2 (generuj sygnał SIGFPE)
PR_GET_FPEXC	11	pobierz tryb wyjątków FPU, zwraca w EAX?
PR_SET_FPEXC	12	ustaw tryb wyjątków FPU, ECX=0 (wyłączone) ECX=1 (async non-recoverable exc. mode), ECX=2 (async recoverable exception mode), ECX=3 (precise exception mode)
PR_GET_TIMING	13	pobierz tryb mierzenia czasu procesu, zwraca w EAX?
PR_SET_TIMING	14	pobierz tryb mierzenia czasu procesu, ECX=0 (normalny) ECX=1 (dokładny)

Struktury dla funkcji sys_capget (numer 184) i sys_capset (numer 185) z /usr/include/linux/capability.h:

```
typedef struct __user_cap_header_struct {
        __u32 version;
        int pid;
} *cap_user_header_t;

typedef struct __user_cap_data_struct {
        __u32 effective;
        __u32 permitted;
        __u32 inheritable;
} *cap_user_data_t;
```

Struktura "stack_t" dla funkcji sys_sigaltstack (numer 186) z /usr/include/asm/signal.h:

```
SS_ONSTACK      0x0001
SS_DISABLE      0x0004

typedef struct sigaltstack {    // stack_t
        void *ss_sp;            // int (dword)
        int ss_flags;   // int (dword), SS_ONSTACK lub SS_DISABLE
        size_t ss_size;         // int (dword)
} stack_t;
```

Struktura "stat64" dla funkcji sys_*stat64 (numer 195, 196, 197) z /usr/include/asm/stat.h:

```
struct stat64 {
        unsigned long long      st_dev;
        unsigned char   __pad0[4];

        unsigned long   __st_ino;

        unsigned int    st_mode;
        unsigned int    st_nlink;

        unsigned long   st_uid;
        unsigned long   st_gid;

        unsigned long long      st_rdev;
```

```
        unsigned char   __pad3[4];

        long long       st_size;
        unsigned long   st_blksize;

        /* Liczba zaalokowanych 512-bajtowych bloków. */
        unsigned long long      st_blocks;

        unsigned long   st_atime;
        unsigned long   st_atime_nsec;

        unsigned long   st_mtime;
        unsigned int    st_mtime_nsec;

        unsigned long   st_ctime;
        unsigned long   st_ctime_nsec;

        unsigned long long      st_ino;
};
```

Informacja dla jądra o korzystaniu z pamięci dla funkcji sys_madvise (numer 219) z /usr/include/bits/mman.h:
Informowanie o sposobach korzystania z pamięci

nazwa	wartość	co oznacza
MADV_NORMAL	0	Żadnego specjalnego traktowania
MADV_RANDOM	1	Można oczekiwać losowych dostępów do tej pamięci
MADV_SEQUENTIAL	2	Można oczekiwać sekwencyjnego dostępu do tej pamięci
MADV_WILLNEED	3	Nasz proces będzie potrzebował tych stron pamięci
MADV_DONTNEED	4	Nasz proces nie potrzebuje tych stron pamięci

---

Makra "makedev" w składni FASM dla funkcji sys_mknod (numer 14) z /usr/include/sys/sysmacros.h:

```
; maj = numer główny urządzenia
; min = numer poboczny urządzenia

; dla jądra 2.4
macro   makedev24       maj, min
{
        xor     esi, esi
        mov     edx, maj
        shl     edx, 8
        or      edx, min
}

; dla jądra 2.6
macro   makedev26       maj, min
{

        mov     edx, min
        and     edx, 0xff

        mov     esi, maj
        and     esi, 0xfff
        shl     esi, 8
```

```
            or      edx, esi

            xor     eax, eax
            mov     esi, min
            and     esi, not 0xff
            shld    eax, esi, 12
            shl     esi, 12

            or      edx, esi

            mov     esi, maj
            and     esi, not 0xfff

            or      esi, eax

    }
```

Numery sygnałów (funkcje 37, 48 i 238) z /usr/include/bits/signum.h:
Sygnały

nazwa	wartość	co oznacza
SIGHUP	1	"Rozłącz się" (hangup)
SIGINT	2	Przerwanie (na przykład naciśnięto Ctrl+C)
SIGQUIT	3	Wyjście
SIGILL	4	Procesor wykonał nieprawidłową instrukcję
SIGTRAP	5	Pułapka (przy śledzeniu wykonywania)
SIGABRT	6	Przerwanie działania
SIGIOT	6 (też!)	Pułapka IOT
SIGBUS	7	Błąd szyny (złe ustawienie danych - np. adres niepodzielny przez 4)
SIGFPE	8	Wyjątek koprocesora (wynik typu NaN, ale też dzielenie przez zero lub przepełnienie w dzieleniu)
SIGKILL	9	Zabicie procesu
SIGUSR1	10	Sygnał definiowany przez użytkownika
SIGSEGV	11	Naruszenie ochrony pamięci (segmentation fault)
SIGUSR2	12	Drugi sygnał definiowany przez użytkownika
SIGPIPE	13	Nieprawidłowy potok
SIGALRM	14	Budzik
SIGTERM	15	Żądanie zakończenia programu
SIGSTKFLT	16	Błąd stosu (koprocesora?)
SIGCHLD, SIGCLD	17	Zmienił się stan procesu potomnego
SIGCONT	18	Kontynuacja
SIGSTOP	19	Żądanie zatrzymania programu
SIGTSTP	20	Zatrzymanie (z) klawiatury (?)
SIGTTIN	21	Odczyt z terminala w tle
SIGTTOU	22	Zapis do terminala w tle

SIGURG	23	Pilne zdarzenie na gnieździe
SIGXCPU	24	Przekroczony limit procesora
SIGXFSZ	25	Przekroczony limit rozmiaru pliku
SIGVTALRM	26	Wirtualny budzik
SIGPROF	27	Budzik profilujący
SIGWINCH	28	Zmiana rozmiaru okna
SIGIO, SIGPOLL	29	Można wykonywać I/O
SIGPWR	30	Restart po awarii zasilania (?) / Błąd zasilania
SIGSYS	31	Nieprawidłowa funkcja systemowa
SIGUNUSED	31 (też!)	(nieużywane)

Operacje futex (funkcja 240) z /usr/include/linux/futex.h:
Operacje na futeksach

nazwa	wartość	co oznacza	zwraca w EAX
FUTEX_WAIT	0	Sprawdza, czy wartość futeksu wynosi tyle, ile podano i czeka	0, gdy proces obudzono przez FUTEX_WAKE
FUTEX_WAKE	1	Budzi co najwyżej ECX procesów czekających na danym adresie	liczba obudzonych procesów
FUTEX_FD	2	Przyporządkuje futeksowi deskryptor pliku	nowy deskryptor pliku

---

Struktura user_desc (funkcja 243) z /usr/src/?/include/asm/ldt.h:

```
struct user_desc {
        unsigned int  entry_number;     /* numer zmienianego lub
                                           pobieranego wpisu w TLS */
        unsigned long base_addr;        /* adres bazowy */
        unsigned int  limit;            /* limit */
        unsigned int  seg_32bit:1;      /* segment */
        unsigned int  contents:2;
        unsigned int  read_exec_only:1; /* tylko RX */
        unsigned int  limit_in_pages:1;
        unsigned int  seg_not_present:1;/* czy nieobecny */
        unsigned int  useable:1;        /* można używać */
};
```

Struktura io_event (funkcja 247) z /usr/src/?/include/linux/aio_abi.h:

```
struct io_event {
        __u64   data;           /* pole danych */
        __u64   obj;            /* skąd przyszło zdarzenie */
        __s64   res;            /* kod wynikowy zdarzenia */
        __s64   res2;           /* wynik drugorzędny */
};
```

Struktura iocb (funkcja 247) z /usr/src/?/include/linux/aio_abi.h:

```
struct iocb {
        /* do wewnętrznego użytku jądra/libc. */
        __u64   aio_data; /*dane do zwrócenia jako dane zdarzenia*/
        __u32   aio_key, aio_reserved1;
                        /* jądro ustawia aio_key na żądany numer */

        /* pola wspólne */
        __u16   aio_lio_opcode; /* zobacz: IOCB_CMD_ */
        __s16   aio_reqprio;
        __u32   aio_fildes;

        __u64   aio_buf;
        __u64   aio_nbytes;
        __s64   aio_offset;

        /* parametry dodatkowe */
        __u64   aio_reserved2;  /*w przyszłości będzie to wskaźnik
                                  na strukturę sigevent */
        __u64   aio_reserved3;
};
```

Struktura epoll_event (funkcja 255 i 256) z man 2 epoll_ctl:

```
typedef union epoll_data {
        void *ptr;
        int fd;
        __uint32_t u32;
        __uint64_t u64;
} epoll_data_t;

struct epoll_event {
        __uint32_t events;  /* zdarzenia Epoll */
        epoll_data_t data;  /* Zmienna danych użytkownika */
};
```

Operacje epoll (funkcja 255) z /usr/include/sys/epoll.h:
Operacje na deskryptorze epoll

nazwa	wartość	co oznacza
EPOLL_CTL_ADD	1	Dodaj deskryptor EDX do deskryptora "epoll" w EBX
EPOLL_CTL_DEL	2	Usuń deskryptor EDX do deskryptora "epoll" w EBX
EPOLL_CTL_MOD	3	Zmień zdarzenie (struktura epoll_event) związane z deskryptorem EDX

Struktura sigevent (funkcja 259) z /usr/include/asm/siginfo.h:

```
#define SIGEV_MAX_SIZE  64
#define SIGEV_PAD_SIZE  ((SIGEV_MAX_SIZE/sizeof(int)) - 3)
```

```
typedef struct sigevent {
        sigval_t sigev_value;
        int sigev_signo;
        int sigev_notify;
        union {
                int _pad[SIGEV_PAD_SIZE];

                struct {
                        void (*_function)(sigval_t);
                        void *_attribute;   /* pthread_attr_t */
                } _sigev_thread;
        } _sigev_un;
} sigevent_t;

/* Dla funkcji mq_notify podana jest taka definicja
man mq_notify:w*/

 union sigval {                /* Przekazane dane */
     int     sival_int;        /* Wartość całkowita */
     void   *sival_ptr; /* Wskaźnik (tak dosłownie mówi manual) */
 };

 struct sigevent {
     int    sigev_notify;      /* Sposób powiadomienia 0=sygnał,
                                  1=nic, 2=utwórz wątek*/
     int    sigev_signal;      /* Numer sygnału powiadomienia */
     union sigval sigev_value; /* Przekazane dane */
     void (*sigev_notify_function) (union sigval);
                               /* Funkcja powiadamiania wątku */
     void  *sigev_notify_attributes;
                               /* Atrybuty funkcji wątku */
 };
```

Struktura itimerspec (funkcja 260) z /usr/include/time.h:

```
struct itimerspec {
        struct timespec it_interval;
        struct timespec it_value;
};
```

Identyfikatory zegara (funkcje 264-267) z /usr/include/bits/time.h:
Identyfikatory zegarów w systemie

nazwa	wartość	co oznacza
CLOCK_REALTIME	0	Systemowy zegar czasu rzeczywistego
CLOCK_MONOTONIC	1	Systemowy zegar monotoniczny
CLOCK_PROCESS_CPUTIME_ID	2	Wysokiej rozdzielczości zegar CPU (dla procesu)
CLOCK_THREAD_CPUTIME_ID	3	Wysokiej rozdzielczości zegar CPU (dla wątku)

Struktura statfs64 (funkcje 268 i 269) z /usr/include/bits/statfs.h:

```
struct statfs64  {
        __SWORD_TYPE f_type;            /* 32 bity */
        __SWORD_TYPE f_bsize;
        __fsblkcnt64_t f_blocks;        /* 64 bity */
        __fsblkcnt64_t f_bfree;
        __fsblkcnt64_t f_bavail;
        __fsfilcnt64_t f_files;
        __fsfilcnt64_t f_ffree;
        __fsid_t f_fsid;        /* struct { int __val[2]; } */
        __SWORD_TYPE f_namelen;
        __SWORD_TYPE f_frsize;
        __SWORD_TYPE f_spare[5];
};
```

Polityka dla pamięci (funkcja 274) z /usr/include/numaif.h i man 2 mbind:
Rodzaje polityki odnośnie pamięci

nazwa	wartość	co oznacza
MPOL_DEFAULT	0	Użyj domyślnej polityki procesu
MPOL_PREFERRED	1	Ustal preferowany węzeł do alokacji
MPOL_BIND	2	Ogranicz alokację pamięci tylko do podanych węzłów
MPOL_INTERLEAVE	3	Optymalizacja przepustowości na rzecz czasu trwania

---

Flagi dla pamięci (funkcja 274) z /usr/src/kernels/.../include/linux/mempolicy.h:
Flagi dla polityki odnośnie pamięci

nazwa	wartość	co oznacza
MPOL_MF_STRICT	(1<<0)	Sprawdź, czy strony pamięci odpowiadają polityce. Jeśli nie odpowiadają polityce domyślnej lub nie mogą zostać przesunięte MPOL_MF_MOVE*, zwracany jest błąd EIO.
MPOL_MF_MOVE	(1<<1)	Przesuń strony pamięci tego procesu, by odpowiadały polityce
MPOL_MF_MOVE_ALL	(1<<2)	Przesuń wszystkie strony pamięci, by odpowiadały polityce

---

Struktura mq_attr (funkcja 277) z man 3 mq_getattr:

```
struct mq_attr {
    long mq_flags;    /* Flagi: 0 lub O_NONBLOCK */
    long mq_maxmsg;   /* Max. liczba wiadomości w kolejce */
    long mq_msgsize;  /* Max. rozmiar wiadomości w bajtach */
    long mq_curmsgs;  /* Liczba wiadomości aktualnie w kolejce */
};
```

Struktura siginfo (funkcja 277) z /usr/include/asm/siginfo.h:

```
typedef struct siginfo {
   int si_signo;
```

```
    int si_errno;
    int si_code;

    union {
            int _pad[SI_PAD_SIZE];

            /* kill() */
            struct {
                    pid_t _pid;             /* pid wysyłającego */
                    uid_t _uid;             /* uid wysyłającego */
            } _kill;

            /* czasomierze POSIX.1b */
            struct {
                    unsigned int _timer1;
                    unsigned int _timer2;
            } _timer;

            /* sygnały POSIX.1b */
            struct {
                    pid_t _pid;             /* pid wysyłającego */
                    uid_t _uid;             /* uid wysyłającego */
                    sigval_t _sigval;
            } _rt;

            /* SIGCHLD */
            struct {
                    pid_t _pid;             /* który potomek */
                    uid_t _uid;             /* uid wysyłającego */
                    int _status;            /* kod wyjścia */
                    clock_t _utime;
                    clock_t _stime;
            } _sigchld;

            /* SIGILL, SIGFPE, SIGSEGV, SIGBUS */
            struct {
                    void *_addr; /* instrukcja, która wywołała błąd */
            } _sigfault;

            /* SIGPOLL */
            struct {
                    int _band;      /* POLL_IN, POLL_OUT, POLL_MSG */
                    int _fd;
            } _sigpoll;
    } _sifields;
} siginfo_t;
```

Flagi dla inotify (funkcja 292) z /usr/src/kernels/.../include/linux/inotify.h:
Flagi dla powiadamiania o zdarzeniach na obserwowanym obiekcie

nazwa	wartość	co oznacza
IN_ACCESS	0x00000001	Dostęp do obiektu
IN_MODIFY	0x00000002	Obiekt został zmodyfikowany
IN_ATTRIB	0x00000004	Zmiana atrybutów
IN_CLOSE_WRITE	0x00000008	Zamknięcie pliku otwartego do zapisu
IN_CLOSE_NOWRITE	0x00000010	Zamknięcie pliku nie otwartego do zapisu

IN_OPEN	0x00000020	Obiekt został otwarty
IN_MOVED_FROM	0x00000040	Z obserwowanego katalogu przeniesiono plik
IN_MOVED_TO	0x00000080	Do obserwowanego katalogu przeniesiono plik
IN_CREATE	0x00000100	W obserwowanym katalogu utworzono plik
IN_DELETE	0x00000200	W obserwowanym katalogu skasowano plik
IN_DELETE_SELF	0x00000400	Obserwowany obiekt został usunięty
IN_MOVE_SELF	0x00000800	Obiekt został przeniesiony
IN_UNMOUNT	0x00002000	System plików został odmontowany
IN_Q_OVERFLOW	0x00004000	Przepełnienie kolejki zdarzeń
IN_IGNORED	0x00008000	Plik został zignorowany
IN_ONLYDIR	0x01000000	Obserwuj ścieżkę tylko gdy jest katalogiem
IN_DONT_FOLLOW	0x02000000	Nie podążaj za dowiązaniami symbolicznymi
IN_MASK_ADD	0x20000000	Jeśli ten obiekt już jest obserwowany, to dopisz dane zdarzenia do obserwacji
IN_ISDIR	0x40000000	Zaszło zdarzenie na katalogu
IN_ONESHOT	0x80000000	Obserwuj daną ścieżkę tylko do pierwszego zdarzenia

---

Wartości dla funkcji sys_ioprio (numer 289 i 290) z /usr/src/.../include/linux/fcntl.h

nazwa	wartość	znaczenie
IOPRIO_WHO_PROCESS	0	ECX to numer pojedynczego procesu
IOPRIO_WHO_PGRP	1	ECX to identyfikator grupy procesów
IOPRIO_WHO_USER	2	ECX to identyfikator użytkownika

Flagi synchronizacji dla funkcji sys_sync_file_range (numer 314) z /usr/src/.../include/linux/fs.h

nazwa	wartość	znaczenie
SYNC_FILE_RANGE_WAIT_BEFORE	1	Czekaj na synchronizację zmienionych stron, które są zaznaczone do zapisania, przed jakimkolwiek zapisem
SYNC_FILE_RANGE_WRITE	2	Zacznij synchronizację zmienionych stron, które nie są zaznaczone do zapisania
SYNC_FILE_RANGE_WAIT_AFTER	4	Czekaj na synchronizację zmienionych stron, po jakimkolwiek zapisie

---

Opcje dla funkcji sigprocmask (numer 175) z /usr/include/asm/signal.h

nazwa	wartość	znaczenie
SIG_BLOCK	0	Zestaw blokowanych sygnałów jest sumą bieżącego zestawu i zestawu w [ECX]
SIG_UNBLOCK	1	Sygnały z [ECX] zostają odblokowane

SIG_SETMASK	2	estaw blokowanych sygnałów jest ustawiany na [ECX]

Opcje dla funkcji fadvice (numer 272) z /usr/include/bits/fcntl.h

nazwa	wartość	znaczenie
POSIX_FADV_NORMAL	0	Domyślny dostęp
POSIX_FADV_RANDOM	1	Dostęp w losowej kolejności
POSIX_FADV_SEQUENTIAL	2	Dostęp sekwencyjny
POSIX_FADV_WILLNEED	3	Te dane będą potrzebne w najbliższej przyszłości
POSIX_FADV_DONTNEED	4	Te dane nie będą potrzebne w najbliższej przyszłości
POSIX_FADV_NOREUSE	5	Dane będą potrzebne tylko raz

Flagi dla funkcji splice (numer 313), vmsplice (numer 316) i tee (numer 315) z /usr/include/bits/fcntl.h

nazwa	wartość	znaczenie
SPLICE_F_MOVE	1	Spróbuj przenieść strony pamięci zamiast kopiowania. Nic nie robi w sys_tee. Nieużywane w sys_vmsplice.
SPLICE_F_NONBLOCK	2	Nie blokuj w czasie operacji wejscia-wyjścia
SPLICE_F_MORE	4	W kolejnych wywołaniach będą dalsze dane. Nic nie robi w sys_tee i sys_vmsplice.
SPLICE_F_GIFT	8	Nieużywane w sys_splice i sys_tee. W sys_vmsplice oznacza darowanie tych strona pmieci dla jądra.

Flagi dla funkcji shmget (numer 29 w x86-64) z /usr/include/linux/ipc.h i /usr/include/bits/shm.h

nazwa	wartość ósemkowo	znaczenie
IPC_CREAT	00001000	Stwórz nowy segment
IPC_EXCL	00002000	Wyłączny dostęp do segmentu
SHM_HUGETLB	04000	Alokuj używając "wielkich stron" pamięci.
SHM_NORESERVE	010000	Nie reserwuj przestrzeni wymiany dla tego segmentu
tryb dostępu	000-777	Takie samo znaczenie, jak we flagach dostępu

---

Flagi dla funkcji shmat (numer 30 w x86-64) z /usr/include/bits/shm.h

nazwa	wartość ósemkowo	znaczenie
SHM_RDONLY	010000	Podłącz segment tylko do odczytu.
SHM_RND	020000	Zaokrąglaj adres w dół do wielokrotności SHMLBA.

SHM_REMAP	040000	Zmień wszystkie mapowania w segmencie

Rozkazy dla funkcji shmctl (numer 31 w x86-64) z /usr/include/linux/ipc.h i /usr/include/bits/shm.h

nazwa	wartość	znaczenie i wartość zwracana
IPC_RMID	0	Zaznacz segment do usunięcia. Zwraca 0.
IPC_SET	1	Zapisz niektóre elementy podanej struktury do jądra. Zwraca 0.
IPC_STAT	2	Skopiuj dane z jądra o podanym segmencie do podanej struktury. Zwraca 0.
IPC_INFO	3	Zwróć informacje o limitach i parametrach współdzielonej pamięci. Zwraca ostatni indeks w tablicy jądra do współdzielonej pamięci.
SHM_LOCK	11	Zapobiega wymianie (swapowaniu) segmentu. Zwraca 0.
SHM_UNLOCK	12	Umożliwia wymianę (swapowanie) segmentu. Zwraca 0.
SHM_STAT	13	Podobne do IPC_STAT, ale identyfikator oznacza numer w tablicy jądra. Zwraca identyfikator segmentu o danym numerze.
SHM_INFO	14	Zwróć informacje o zasobach używanych przez współdzieloną pamięć. Zwraca ostatni indeks w tablicy jądra do współdzielonej pamięci.

Struktura shmid_ds (funkcja 31 w x86-64) z /usr/include/bits/shm.h:

```
struct shmid_ds {
        struct ipc_perm shm_perm;    /* Właściciel i uprawnienia */
        size_t          shm_segsz;   /* Rozmiar segmentu w bajtach */
        time_t          shm_atime;   /* Czas ostatniego dołączenia */
        time_t          shm_dtime;   /* Czas ostatniego odłączenia */
        time_t          shm_ctime;   /* Czas ostatniej zmiany */
        pid_t           shm_cpid;    /* PID twórcy */
        pid_t           shm_lpid;    /* PID ostatniej operacji shmat
                                        lub shmdt */
        shmatt_t        shm_nattch;  /* Bieżąca liczba podłączeń */
        ...
};
```

Domeny dla gniazd (funkcja 41 w x86-64) z /usr/include/bits/socket.h

nazwa	wartość	znaczenie
AF_UNIX, AF_LOCAL	1	Lokalna komunikacja
AF_INET	2	Protokoły IPv4
AF_AX25	3	Protokół AX.25 amatorskiego radia
AF_IPX	4	Protokoły Novell IPX
AF_APPLETALK	5	Appletalk
AF_NETROM	6	Amatorskie radio NetROM
AF_BRIDGE	7	Mostek wieloprotokołowy
AF_ATMPVC	8	Dostęp do surowych ATM PVC

AF_X25	9	Protokół ITU-T X.25 / ISO-8208
AF_INET6	10	Protokoły IPv6
AF_ROSE	11	Amatorskie radio X.25 PLP
AF_DECnet	12	Zarezerwowane dla projektu DECnet
AF_NETBEUI	13	Zarezerwowane dla projektu 802.2LLC
AF_SECURITY	14	Pseudo-domena dla wywołania zwrotnego zabezpieczeń
AF_KEY	15	Interfejs zarządzania kluczami
AF_NETLINK	16	Urządzenie interfejsu do jądra
AF_PACKET	17	Niskopoziomowy interfejs pakietowy
AF_ASH	18	Ash
AF_ECONET	19	Acorn Econet
AF_ATMSVC	20	ATM SVC
AF_SNA	22	Projekt Linux SNA
AF_IRDA	23	Gniazda IrDA
AF_PPPOX	24	Gniazda PPPoX
AF_WANPIPE	25	Interfejs do gniazd Wanpipe
AF_BLUETOOTH	31	Gniazda Bluetooth

Typy gniazd (funkcja 41 w x86-64) z /usr/include/bits/socket.h

nazwa	wartość	znaczenie
SOCK_STREAM	1	Sekwencjonowany, wiarygodny, dwukierunkowy, opary na połączeniu strumień bajtów
SOCK_DGRAM	2	Obsługuje datagramy (bez połączenia, niewiarygodny)
SOCK_RAW	3	Dostęp bezpośredni do protokołów sieciowych
SOCK_RDM	4	Wiarygodna warstwa datagramów bez gwarancji kolejności.
SOCK_SEQPACKET	5	Sekwencjonowany, wiarygodny, dwukierunkowy, opary na połączeniu strumień bajtów. Odbiorca musi przeczytać cały pakiet za każdym czytaniem.
SOCK_PACKET	10	Przestarzałe, nie używać
SOCK_NONBLOCK	04000 ósemkowo	Ustaw tryb nieblokujący.
SOCK_CLOEXEC	02000000 ósemkowo	Ustaw flagę zamknij-podczas-exec.

Flagi dla funkcji sendto (numer 44 w x86-64) i recvfrom (numer 45 w x86-64) z /usr/include/bits/socket.h

nazwa	wartość	znaczenie
MSG_CONFIRM	0x800	(sendto) Potwierdzenie otrzymania odpowiedzi
MSG_DONTROUTE	0x04	(sendto) Nie używaj bramki do wysyłania, wyślij bezpośrednio

MSG_DONTWAIT	0x40	(sendto, recvfrom) Włącz tryb nieblokujący
MSG_EOR	0x80	(sendto) Koniec rekordu danych
MSG_MORE	0x8000	(sendto) Uruchamiający ma więcej danych do wysłania
MSG_NOSIGNAL	0x4000	(sendto) Nie wysyłaj sygnałów
MSG_OOB	0x01	(sendto, recvfrom) Wyślij dane poza kolejnością
MSG_CMSG_CLOEXEC	0x40000000	(recvmsg) Ustaw flagę zamknij-podczas-exc na deskryptorze otrzymanym podczas operacji SCM_RIGHTS
MSG_ERRQUEUE	0x2000	(recvfrom) Błędy powinny być odbierane przez kolejkę błędów gniazda
MSG_PEEK	0x02	(recvfrom) Pobierz dane z kolejki bez usuwania ich z kolejki
MSG_TRUNC	0x20	(recvfrom) Zwróć prawdziwą długość danych, nawet gdy bufor był mniejszy
MSG_WAITALL	0x100	(recvfrom) Czekaj na pełne zakończenie operacji

Struktura msghdr (funkcja 46 w x86-64) z man 2 sendmsg:

```
struct msghdr {
        void         *msg_name;       /* opcjonalny adres */
        socklen_t     msg_namelen;    /* rozmiar adresu */
        struct iovec *msg_iov;        /* tablica wysyłania i
                                         zbierania */
        size_t        msg_iovlen;     /* liczba elementów w
                                          msg_iov */
        void         *msg_control;    /* dane pomocnicze */
        socklen_t     msg_controllen; /* długość bufora danych
                                         pomocniczych */
        int           msg_flags;      /* flagi na odebranej
                                         wiadomości: MSG_EOR,
                                         MSG_TRUNC, MSG_CTRUNC,
                                         MSG_OOB, MSG_ERRQUEUE*/

        };
```

Struktura sembuf (funkcja 65 w x86-64) z /usr/include/sys/sem.h:

```
struct sembuf
{
        unsigned short int sem_num;   /* liczba semaforów */
        short int sem_op;             /* operacja na semaforze:
                                         liczba dodatnia jest dodawana
                                         zero oznacza czekanie na zero
                                       liczba ujemna jest odejmowana*/
        short int sem_flg;            /* flaga operacji:
                                         IPC_NOWAIT=04000 ósemkowo
                                         lub SEM_UNDO=0x1000 */
};
```

Rozkazy dla semaforów (funkcja 66 w x86-64) z /usr/include/bits/sem.h i /usr/include/bits/ipc.h

nazwa	wartość	znaczenie i wartość zwracana
IPC_RMID	0	Usuń zestaw semaforów. Zwraca 0.
IPC_SET	1	Kopiuj dane z tablicy buf o adresie podanym w R10 do struktur jądra. Zwraca 0.
IPC_STAT	2	Kopiuj dane ze struktur jądra do tablicy buf o adresie podanym w R10. Zwraca 0.
IPC_INFO	3	Zwróć w buf w R10 informacje o systemowych limitach i parametrach semaforów. Zwraca numer ostatniego używanego elementu w tablicy jądra.
GETPID	11	Zwraca PID procesu, który wykonał ostatnią operację na tym semaforze
GETVAL	12	Zwraca wartość podanego semafora w zestawie
GETALL	13	Do tablicy array o adresie podanym w R10 wpisuje wartości wszystkich semaforów w systemie. Zwraca 0.
GETNCNT	14	Zwraca liczbę procesów czekających na zwiększenie się podanego semaforu w zestawie
GETZCNT	15	Zwraca liczbę procesów czekających na wyzerowanie się podanego semaforu w zestawie
SETVAL	16	Ustaw wartości podane semafora w zestawie na tą podaną w R10. Zwraca 0.
SETALL	17	Ustaw wartości wszystkich semaforów na te podane w tablicy array o adresie podanym w R10. Zwraca 0.
SEM_STAT	18	Podobne do IPC_STAT, lecz identyfikator zestawu semaforów jest numerem semaforu w tablicy jądra. Zwraca identyfikator zestawu semaforów o podanym numerze.
SEM_INFO	19	Podobne do IPC_INFO, zwraca inne wartości w niektórych polach (man semctl) Zwraca numer ostatniego używanego elementu w tablicy jądra.

Unia semun (funkcja 66 w x86-64) z man 2 semctl:

```
union semun {
        int              val;    /* Wartość dla SETVAL */
        struct semid_ds *buf;    /* Bufor na IPC_STAT, IPC_SET */
        unsigned short  *array; /* Tablica dla GETALL, SETALL */
        struct seminfo  *__buf; /* Bufor dla IPC_INFO */
};
```

Struktura msgp (funkcja 69 w x86-64) z man 2 msgsnd:

```
struct msgbuf {
        long mtype;       /* typ wiadomości, musi być > 0 */
        char mtext[1];    /* dane wiadomości */
};
```

Flagi dla funkcji msgrcv (numer 70 w x86-64) z /usr/include/bits/msq.h

nazwa	wartość ósemkowo	znaczenie
IPC_NOWAIT	04000	Nie czekaj na wiadomości
MSG_EXCEPT	020000	Odbierz pierwszą wiadomość NIE będącą podanego typu

MSG_NOERROR	010000	Obcinaj wiadomości do podanego rozmiaru

Struktura msqid_ds (funkcja 71 w x86-64) z man 2 msgctl:

```
struct msqid_ds {
        struct ipc_perm msg_perm;     /* Właściciel i uprawnienia */
        time_t          msg_stime;    /* Czas ostatniego msgsnd */
        time_t          msg_rtime;    /* Czas ostatniego msgrcv */
        time_t          msg_ctime;    /* Czas ostatniej zmiany */
        unsigned long   __msg_cbytes; /* Aktualna liczba bajtów w
                                         kolejce */
        msgqnum_t       msg_qnum;     /* Aktualna liczba wiadomości w
                                         kolejce */
        msglen_t        msg_qbytes;   /* Maksymalna liczba bajtów
                                         dozwolona w kolejce */
        pid_t           msg_lspid;    /* PID ostatniego msgsnd */
        pid_t           msg_lrpid;    /* PID ostatniego msgrcv */
};

struct ipc_perm {
        key_t          __key;       /* Klucz podany msgget */
        uid_t          uid;         /* Efektywny UID of właściciela */
        gid_t          gid;         /* Efektywny GID of właściciela */
        uid_t          cuid;        /* Efektywny UID of twórcy */
        gid_t          cgid;        /* Efektywny GID of twórcy */
        unsigned short mode;        /* Uprawnienia */
        unsigned short __seq;       /* Numer sekwencyjny */
};
```

Rozkazy dla kolejek (funkcja 71 w x86-64) z /usr/include/bits/msq.h i /usr/include/bits/ipc.h

nazwa	wartość	znaczenie i wartość zwracana
IPC_RMID	0	Usuń kolejkę. Zwraca 0.
IPC_SET	1	Kopiuj dane z tablicy buf o adresie podanym w R10 do struktur jądra. Zwraca 0.
IPC_STAT	2	Kopiuj dane ze struktur jądra do tablicy buf o adresie podanym w R10. Zwraca 0.
IPC_INFO	3	Zwróć w buf w R10 informacje o systemowych limitach i parametrach kolejek. Zwraca numer ostatniego używanego elementu w tablicy jądra.
MSG_STAT	11	Podobne do IPC_STAT, lecz identyfikator zestawu semaforów jest numerem semaforu w tablicy jądra. Zwraca identyfikator zestawu semaforów o podanym numerze.
MSG_INFO	12	Podobne do IPC_INFO, zwraca inne wartości w niektórych polach (man semctl) Zwraca numer ostatniego używanego elementu w tablicy jądra.

Podfunkcje dla arch_prctl (numer 158 w x86-64) z linux/arch/x86/include/asm/prctl.h

nazwa	wartość	znaczenie i wartość zwracana

ARCH_SET_FS	0x1002	Ustaw adres bazowy deskryptora FS na podany adres
ARCH_GET_FS	0x1003	Pobierz adres bazowy deskryptora FS do zmiennej pod podanym adresem
ARCH_SET_GS	0x1001	Ustaw adres bazowy deskryptora GS na podany adres
ARCH_GET_GS	0x1004	Pobierz adres bazowy deskryptora GS do zmiennej pod podanym adresem

---

Struktura getcpu_cache (funkcja 318) z linux/include/linux/getcpu.h:

```
struct getcpu_cache {
        unsigned long blob[128 / sizeof(long)];
};
```

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Różnice składni AT&T i Intel

Różnice między składnią Intela a AT&T

Cecha	Intel	AT&T
Liczby w instrukcjach (z wyjątkiem adresów)	12345678	$12345678
Rejestry	eax, ebx, cs, cr0, dr0, tr6, st(0)/st0	%eax, %ebx, % cs, %cr0, %db0, %tr6, %st(0)
Adresy bezwzględne w skokach	12345678	*12345678 brak gwiazdki oznacza adres względem bieżącego EIP
Kolejność argumentów w instrukcjach	mov eax, ebx imul ebx, eax, 69	movl %ebx, %eax imul $69, %eax, %ebx

Przyrostki rozmiarowe	mov al, byte ptr [abcd]
	mov bx, word ptr [efgh]
	mov ecx, dword ptr [ijkl]
	mov eax, 1
	int 80h
		movb abcd, %al
		movw efgh, %bx
		movl ijkl, %ecx
		movl $1, %eax
		int $0x80

Skoki pod dany adres	jmp/call far seg:off	ljmp/lcall $seg, $off
Instrukcje powrotu	retf n ret n	lret $n ret $n

Rozszerzanie wartości do większych rozmiarów	movsx ax, bl
	movzx eax, bl
	movsx eax, bx
	cbw
	cwde
	cwd
	cdq
		movsbw %bl, %ax
		movzbl %bl, %eax
		movswl %bx, %eax
		cbtw
		cwtl
		cwtd
		cltd

Adresowanie złożone	seg : [baza + index*skala + liczba]
	[ebp-4]
	[cos+eax*2]
	gs : gdzies
	mov eax, [ecx]
	sub eax, [ebx+ecx*4-20h]
	call [eax*4 + zmienna]
		%seg : liczba(%baza, %index, skala)
		-4(%ebp)
		cos(,%eax,2)
		%gs : gdzies
		movl (%ecx), %eax
		subl -0x20(%ebx,%ecx,0x4), %eax
		call *zmienna(,%eax,4)

Adresowanie pojedynczych zmiennych	mov al, byte [costam]	movb costam(,1), %al movb costam, %al
Pobieranie adresu zmiennych	mov eax, offset costam (TASM, MASM) mov eax, costam (NASM, FASM)	movl $costam, %eax
Przyrostki instrukcji FPU	fld dword [a] fld qword [b] fld tbyte/tword [c] fild qword [d] fild dword [e] fild word [f]	flds a fldl b fldt c fildq d fildl e filds f

Polecam też (po angielsku) porównanie składni AT&T ze składnią Intela oraz wstęp do wstawek asemblerowych (w GCC) na stronach DJGPP, podręcznik GCC (sekcje: 5.34 i 5.35), oraz (w języku polskim) stronę pana Danileckiego.

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

14.02.2010

Skankody i kody ASCII klawiszy

Informacje te pochodzą z Ralf Brown's Interrupt List oraz ze znakomitej książki Art of Assembly Language Programming (wersja dla DOS-a) autorstwa Randalla Hyde'a. Książkę można za darmo ściągnąć z Webstera

(przeskocz skankody)

Skankody (scan codes) wysyłane przez klawiaturę

Klawisz	Naciśnięcie	Zwolnienie	Kl	Nac	Zwol	Kl	Nac	Zwol	Kl	Nac	Zwol
Esc	01	81	] }	1B	9B	. >	34	B4	END	4F	CF
1 !	02	82	ENTER	1C	9C	/ ?	35	B5	DÓŁ	50	D0
2 @	03	83	Ctrl	1D	9D	PShift	36	B6	PGDN	51	D1
3 #	04	84	A	1E	9E	* (num)	37	B7	INS	52	D2
4 $	05	85	S	1F	9F	alt	38	B8	DEL	53	D3
5 %	06	86	D	20	A0	spacja	39	B9	SysRq	54	D4
6 ^	07	87	F	21	A1	CAPS	3A	BA	/ (num)	E0 35	B5
7 &	08	88	G	22	A2	F1	3B	BB	enter (num)	E0 1C	9C
8 *	09	89	H	23	A3	F2	3C	BC	F11	57	D7
9 (	0A	8A	J	24	A4	F3	3D	BD	F12	58	D8
0)	0B	8B	K	25	A5	F4	3E	BE	LWin	5B	DB
- _	0C	8C	L	26	A6	F5	3F	BF	PWin	5C	DC
+ =	0D	8D	; :	27	A7	F6	40	C0	Menu	5D	DD
BkSp	0E	8E	‘ "	28	A8	F7	41	C1	ins (num)	E0 52	D2
Tab	0F	8F	~ `	29	A9	F8	42	C2	del (num)	E0 53	D3
Q	10	90	LShift	2A	AA	F9	43	C3	home (num)	E0 47	C7
W	11	91	\ |	2B	AB	F10	44	C4	end (num)	E0 4F	CF
E	12	92	Z	2C	AC	NUM	45	C5	pgup (num)	E0 49	C9
R	13	93	X	2D	AD	SCRLCK	46	C6	pgdn (num)	E0 51	D1
T	14	94	C	2E	AE	HOME	47	47	lewo (num)	E0 4B	CB
Y	15	95	V	2F	AF	GÓRA	48	C8	prawo (num)	E0 4D	CD
U	16	96	B	30	B0	PGUP	49	C9	góra (num)	E0 48	C8
I	17	97	N	31	B1	- (num)	4A	CA	dół (num)	E0 50	D0
O	18	98	M	32	B2	5 (num)	4C	CC	Palt	E0 38	B8
P	19	99	LEWO	4B	CB	PRAWO	4D	CD	Pctrl	E0 1D	9D
[{	1A	9A	, <	33	B3	+ (num)	4E	CE	Pauza	E1 1D 45 E1 9D C5	(brak)

Na żółto, małymi literami i napisem num oznaczyłem klawisze znajdujące się (moim zdaniem) na klawiaturze numerycznej.

Kody ASCII klawiszy z modyfikatorami

Klawisz	Skankod	kod ASCII	z Shift	z Control	z Alt	z NumLock	z CapsLock	z Shift+CapsLock	z Shift+NumLock
Esc	01	1B	1B	1B	(brak)	1B	1B	1B	1B
1 !	02	31	21	(brak)	7800	31	31	31	31
2 @	03	32	40	0300	7900	32	32	32	32
3 #	04	33	23	(brak)	7A00	33	33	33	33
4 $	05	34	24	(brak)	7B00	34	34	34	34
5 %	06	35	25	(brak)	7C00	35	35	35	35
6 ^	07	36	5E	1E	7D00	36	36	36	36
7 &	08	37	26	(brak)	7E00	37	37	37	37
8 *	09	38	2a	(brak)	7F00	38	38	38	38
9 (	0A	39	28	(brak)	8000	39	39	39	39
0)	0B	30	29	(brak)	8100	30	30	30	30
- _	0C	2D	5F	1F	8200	2D	2D	5F	5F
+ =	0D	3D	2B	(brak)	8300	3D	3D	2B	2B
BkSp	0E	08	08	7F	(brak)	08	08	08	08
Tab	0F	09	0F00	(brak)	(brak)	09	09	0F00	0F00
Q	10	71	51	11	1000	71	51	71	51
W	11	77	57	17	1100	77	57	77	57
E	12	65	45	05	1200	65	45	65	45
R	13	72	52	12	1300	72	52	72	52
T	14	74	54	14	1400	74	54	74	54
Y	15	79	59	19	1500	79	59	79	59
U	16	75	55	15	1600	75	55	75	55
I	17	69	49	09	1700	69	49	69	49
O	18	6F	4F	0F	1800	6F	4F	6F	4F
P	19	70	50	10	1900	70	50	70	50
[{	1A	5B	7B	1B	(brak)	5B	5B	7B	7B
] }	1B	5D	7D	1D	(brak)	5D	5D	7D	7D
ENTER	1C	0D	0D	0A	(brak)	0D	0D	0A	0A
CTRL	1D	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)
A	1E	61	41	01	1E00	61	41	61	41
S	1F	73	53	13	1F00	73	53	73	53
D	20	64	44	04	2000	64	44	64	44
F	21	66	46	06	2100	66	46	66	46
G	22	67	47	07	2200	67	47	67	47
H	23	68	48	08	2300	68	48	68	48
J	24	6A	4A	0A	2400	6A	4A	6A	4A
K	25	6B	4B	0B	2500	6B	4B	6B	4B
L	26	6C	4C	0C	2600	6C	4C	6C	4C

; :	27	3B	3A	(brak)	(brak)	3B	3B	3A	3A
‘ "	28	27	22	(brak)	(brak)	27	27	22	22
~ `	29	60	7E	(brak)	(brak)	60	60	7E	7E
LShift	2A	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)
\ |	2B	5C	7C	1C	(brak)	5C	5C	7C	7C
Z	2C	7A	5A	1A	2C00	7A	5A	7A	5A
X	2D	78	58	18	2D00	78	58	78	58
C	2E	63	43	03	2E00	63	43	63	43
V	2F	76	56	16	2F00	76	56	76	56
B	30	62	42	02	3000	62	42	62	42
N	31	6E	4E	0E	3100	6E	4E	6E	4E
M	32	6D	4D	0D	3200	6D	4D	6D	4D
, <	33	2C	3C	(brak)	(brak)	2C	2C	3C	3C
. >	34	2E	3E	(brak)	(brak)	2E	2E	3E	3E
/ ?	35	2F	3F	(brak)	(brak)	2F	2F	3F	3F
PShift	36	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)
* (num)	37	2A	(brak?)	10	(brak)	2A	2A	(brak?)	(brak?)
alt	38	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)
spacja	39	20	20	20	(brak)	20	20	20	20
caps lock	3A	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)
F1	3B	3B00	5400	5E00	6800	3B00	3B00	5400	5400
F2	3C	3C00	5500	5F00	6900	3C00	3C00	5500	5500
F3	3D	3D00	5600	6000	6A00	3D00	3D00	5600	5600
F4	3E	3E00	5700	6100	6B00	3E00	3E00	5700	5700
F5	3F	3F00	5800	6200	6C00	3F00	3F00	5800	5800
F6	40	4000	5900	6300	6D00	4000	4000	5900	5900
F7	41	4100	5A00	6400	6E00	4100	4100	5A00	5A00
F8	42	4200	5B00	6500	6F00	4200	4200	5B00	5B00
F9	43	4300	5C00	6600	7000	4300	4300	5C00	5C00
F10	44	4400	5D00	6700	6100	4400	4400	5D00	5D00
num lock	45	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)
scroll lock	46	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)	(brak)
home	47	4700	37	7700	(brak)	37	4700	37	4700
góra	48	4800	38	(brak)	(brak)	38	4800	38	4800
pgup	49	4900	39	8400	(brak)	39	4900	39	4900
- (num)	4A	2D	2D	(brak)	(brak)	2D	2D	2D	2D
lewo	4B	4B00	34	7300	(brak)	34	4B00	34	4B00
5 (num)	4C	4C00	35	(brak)	(brak)	35	4C00	35	4C00
prawo	4D	4D00	36	7400	(brak)	36	4D00	36	4D00

+ (num)	4E	2B	2B	(brak)	(brak)	2B	2B	2B	2B
end	4F	4F00	31	7500	(brak)	31	4F00	31	4F00
dół	50	5000	32	(brak)	(brak)	32	5000	32	5000
pgdn	51	5100	33	7600	(brak)	33	5100	33	5100
ins	52	5200	30	(brak)	(brak)	30	5200	30	5200
del	53	5300	2E	(brak)	(brak)	2E	5300	2E	5300

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

14.02.2010

Porównanie dyrektyw FASMa i NASMa

(przeskocz różnice składni)

Najważniejsze różnice między dyrektywami FASMa i NASMa

typ dyrektywy	NASM	FASM
deklaracje danych	NASM: db, dw, dd, dq, dt	FASM: db, dw/du, dd, dp/df, dq, dt
rezerwacja niezainicjalizowanej pamięci	NASM: resb, resw, resd, resq, rest	FASM: rb, rw, rd, rp/rf, rq, rt
deklaracje tablic	NASM: TIMES 10 db "$" TIMES 25*34 db/dw/dd/dp/df/dq/dt 0 resb/resw/resd/resq/rest 25*34	FASM: TIMES 10 db "$" TIMES 25*34 db/dw/dd/dp/df/dq/dt 0 rb/rw/rd/rp/rf/rq/rt 25*34
operacje logiczne	NASM: +, -, *, /, %, |, ^, &, <<, >>, ~	FASM: +, -, *, /, mod, or, xor, and, shl, shr, not
deklaracje stałych	NASM: %define, %idefine, %xdefine, %xidefine, equ	FASM: =, equ
etykiety anonimowe	NASM: nie ma	FASM: @@, @b/@r, @f
makra	NASM: %macro, %imacro nazwa ilosc_arg ... %endm	FASM: macro nazwa arg {...}
kompilacja warunkowa	NASM: %if, %if(n)def, %elif, %else, %endif	FASM: if, else if, else, end if
struktury	NASM: struc nazwa ... endstruc	FASM: struc nazwa {...}
symbole zewnętrzne	NASM: extern, global	FASM: extrn, public
segmenty	NASM: segment nazwa, section nazwa	FASM: ELF: section "nazwa" executable/writeable ELF executable: segment readable/writeable/executable
typowy początek linkowanego programu linuksowego	NASM: section .text global _start _start:	FASM: format ELF section ".text" executable public _start _start:

typowy początek nielinkowanego programu linuksowego	NASM: nie ma, zawsze trzeba linkować	FASM: format ELF executable segment readable executable

Sposoby kompilacji w kompilatorach FASM i NASM

typ programu	NASM	FASM
binarny	nasm -f bin -o prog.bin prog.asm	fasm prog.asm prog.bin
obiekt ELF	nasm -f elf -o prog.o prog.asm	fasm prog.asm prog.o
wykonywalny ELF	nasm -f elf -o prog.o prog.asm ld -s -o prog prog.o	fasm prog.asm prog

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)

Odnośniki do innych źródeł:

1. (DOBRY) Spis przerwań Ralfa Browna (Ralf Brown's Interrupt List, RBIL)
   (przeskocz RBIL)
   Jeśli zaczynasz programować dla DOS-a (i nie chcesz na razie pisać aplikacji okienkowych dla Windowsa), to nie pożałujesz, jeśli ściągniesz! Zawiera opis wszystkich funkcji DOSa, BIOS-u, i wiele innych informacji. Bez tego ani rusz! Do ściągnięcia tu: RBIL

2. Kompilatory języka asembler:
   (przeskocz kompilatory)
   - (DOBRY) NASM (The Netwide Assembler - DOS, Windows, Linux, 16-bit, 32-bit, 64-bit) - prosty w obsłudze kompilator języka asembler z pełną dokumentacją: strona NASMa.

     W sam raz do pisania programów typu .COM. Do pisania programów .EXE potrzebować będziesz linkera. Polecam Alink (darmowy program służący za DPMI znajdziecie na stronach, z których można pobrać NASMa - nazywa się CWSDPMI) lub VAL

   - Napisany przez Polaka FASM (The Flat Assembler - DOS, Windows, Linux, 16-bit, 32-bit, 64-bit): strona FASMa
     Absolutnie fantastyczne narzędzie do pisania programów okienkowych! Żadnych zbędnych śmieci, nie potrzebujesz zewnętrznych linkerów, bibliotek, niczego. FASM ma to wszystko w załącznikach, a wersja GUI dla Windows to kompilator ze środowiskiem, całość tylko w 1 pliku .exe!
     Całkiem nieźle radzi sobie też w Linuksie.

   - YASM (DOS, Linux, Windows, 16-bit, 32-bit, 64-bit): strona YASMa
     Prawie całkowicie zgodny ze składniami NASMa i GNU asemblera.

   - Napisany przez Polaka SB86 (dawniej SASM) - DOS, Windows, Linux, 16-bit, 32-bit: sb86.way.to
     Składnia różni się nieco od innych - przypomina nieco język C, ale z instrukcji wynika, że kompilator ten ma całkiem duże możliwości.

   - LZASM (Lazy Assembler - DOS/Windows, zgodny z TASMem): lzasm.hotbox.ru

   - JWasm (DOS/Windows, 16-bit, 32-bit, zgodny z MASMem w wersji 6): japheth.de/JWasm.html

   - A86 (DOS, darmowy tylko 16-bit + debugger 16-bit): eji.com

   - MASM (Microsoft Macro Assembler - DOS/Windows, 16-bit, 32-bit): www.masm32.com oraz webster.cs.ucr.edu
     16-bitowy linker znajduje się na stronach Microsoft

   - HLA (High-Level Assembler - Windows/Linux, 32-bit): webster.cs.ucr.edu

   - Jeremy Gordon's GoAsm + dobry debugger 32-bit GoBug i wiele innych (tylko Windows): www.godevtool.com

   - Odnośniki do innych kompilatorów: Forever Young Software - linki

3. Kursy, książki:
   (przeskocz kursy)
   - (DOBRY) The Art of Assembly Language Programmnig (Art of Assembler, AoA): webster.cs.ucr.edu
     (PL) Książka została przetłumaczona na język polski przez Kremika: www.rag.kgb.pl/aoapl.php
   - PC-Asm

   - Kursy programowania w trybie chronionym
   - Assembler Programming

   - Tutorial dla początkujących - Ready to start!
   - Atrevida PC Game Programming Tutorials: atrevida.comprenica.com
   - (PL) Kurs asemblera by Skowik: www.republika.pl/skowi_magik
   - (PL) Kursy asemblera: www.pieciuk.terramail.pl/assembler.htm
   - (PL) Asembler - szybkie wprowadzenie: www.asembler.host.sk

   - (PL) Jeszcze jeden kurs asemblera w połączeniu z Pascalem: www.zsme.tarnow.pl/killer/asm/asm.htm
   - (PL) Kopia kursu Grzegorza Złotowicza: www.shitsoft.net/programowanie/asm/index2.htm oraz kilka innych arytkułów: www.shitsoft.net/biblioteka/bib_prog.htm

   - (PL) Trochę artykułów o różnej tematyce: http://coders.shnet.pl/legacy/coders/main/assembler.html

   - (PL) Asembler Programowanie
   - (PL, wersja papierowa) Ryszard Goczyński, Michał Tuszyński (Wydawnictwo HELP): *Mikroprocesory 80286, 80386 i i486* (o programowaniu ogólnym) oraz *Koprocesory Arytmetyczne 80287 i 80387 oraz jednostka arytmetyki zmiennoprzecinkowej i486* (o programowaniu koprocesora)

4. Polskie fora o programowaniu:
   (przeskocz fora)
   - Vademecum Programisty
   - Forum koder.org

5. Dokumentacja procesorów (ich wszystkie instrukcje, rejestry, technologie):
   (przeskocz dokumentację)
   - AMD
   - Intel
   - DDJ Microprocessor Center: www.x86.org
   - Transmeta

   - Ogólna, wiele firm, wiele procesorów (ale tylko te zgodne z Intel/AMD): Sandpile

   - Spis instrukcji według kodu rozkazu: X86Asm
   - Kolejny spis instrukcji

6. Pisanie w asemblerze pod Linuksa:
   (przeskocz asm w Linuksie)
   - Kursy, porady, dużo różnych informacji - Linux Assembly: linuxassembly.org (alternatywny adres: asm.sourceforge.net)
   - Kursy dla FreeBSD - int80h.org: www.int80h.org
   - Debugger pob Linuksa: PrivateICE
   - Linux Assembly Tutorial
   - inny tutorial
   - Przykładowe maleńkie programiki
   - (PL) Wstawki asemblerowe w GCC - krótki kurs w języku polskim
   - Porównanie składni AT&T ze składnią Intela oraz wstęp do wstawek asemblerowych (w GCC)
   - Opis wstawek asemblerowych w GCC prosto z podręcznika GCC (sekcje: 5.34 i 5.35)
   - Program przerabiający składnię AT&T na składnię NASMa
   - (PL) RAG
   - Kopia mojego opisu przerwania int 80h

   - Książka Programming from the Ground Up
   - Desktop Linux Asm
   - kolejny opis wstawek asemblerowych
   - Różne narzędzia
   - Debuger do asemblera
   - Pisanie pierwszych programów w NASMie
   - Specyfikacja funkcji systemowych

7. Pisanie w asemblerze pod Windowsa:
   (przeskocz asm w Windowsie)
   - (PL)(DOBRY) Asembler dla Windows (kopia kursu Iczeliona)
   - (DOBRY) Programowanie pod Windows'a: kurs Iczeliona
   - Tom Cat's Win32 Asm page

   - Olly Debugger
   - NaGoA - Nasm + GoRC (Go Resource Compiler) + Alink
   - GoAsm (+dobry debugger 32-bit GoBug, GoRC i wiele innych): www.godevtool.com
   - strona Hutch'a: www.movsd.com

   - (PL) RAG

8. Portale programistyczne:
   (przeskocz portale)
   - (PL) 4programmers.net

   - (PL) Programik.com
   - Programmers' Heaven

   - The Free Country
   - Free Programming Resources

   - CodeWiki - wiki z różnymi wycinkami kodu

9. Strony poświęcone pisaniu systemów operacyjnych:
   (przeskocz OS)
   - (DOBRY) Bona Fide OS Development
   - (DOBRY) Operating System Resource Center
   - Kursy programowania w trybie chronionym

   - Dokumentacja na różne tematy: strona systemu O3one
   - OSDev.org
   - Zakątek Boba
   - OSDev.pl
   - alt.os.development - najczęściej zadawane pytania

10. Środowiska programistyczne:
    (przeskocz IDE)
    - RadASM - środowisko programistyczne obsługujące wiele kompilatorów (MASM, TASM, NASM, FASM, GoAsm, HLA)
    - NasmIDE
    - TasmIDE
    - Środowisko dla FASMa (wbudowane w kompilator w wersji GUI): flatassembler.net oraz Fresh
    - WinAsm Studio
    - AsmEdit (dla MASMa)
    - Lizard NASM IDE

11. Edytory i hex-edytory/disassemblery:
    (przeskocz edytory)
    - (DOBRY) Programmer's File Editor
    - Quick Editor
    - The Gun
    - HTE
    - Dużo więcej na stronach The Free Country - edytory
    - (DOBRY) XEdit
    - b2hedit
    - Biew
    - Dużo więcej na stronach The Free Country - disassemblery

12. Inne:
    (przeskocz inne linki)
    - (PL)(DOBRY) Mnóstwo różnych dokumentacji: mediaworks.w.interia.pl/docs.html
    - (PL) Kursy, linki, sporo o FASMie: Decard.net

    - (PL) Architektura procesorów firmy Intel: domaslawski.fm.interia.pl
    - Forever Young Software
    - Spis instrukcji procesora i koprocesora, czasy ich wykonywania, sztuczki optymalizacyjne: www.emboss.co.nz/pentopt/freeinfo.html
    - Strona poświęcona opisom foramtów plików różnego typu (graficzne, dźwiękowe): www.wotsit.org
    - Optymalizacja, dużo linków, makra dla kompilatorów: www.agner.org/assem

    - (PL) RAG

- (PL) Wojciech Muła
- (PL) Programowanie - KODER
- Tabela kodów ASCII

- Informacje o dyskach twardych itp.: www.ata-atapi.com
- Brylanty asemblera
- Linki, źródła, informacje: grail.cba.csuohio.edu/~somos/asmx86.html
- Christopher Giese
- Laura Fairhead
- Jim Webster
- LadSoft
- Paul Hsieh

- Whiz Kid Technomagic

- Koms Bomb Assembly World: http://www.mujweb.cz/www/komsbomb/

- Comrade's homepage: comrade64.cjb.net, comrade.win32asm.com, comrade.ownz.com
- Ciekawe operacje na bitach (w C)

- Sztuczki optymalizacyjne: www.mark.masmcode.com.
- FASMLIB - biblioteka procedur, nie tylko dla FASMa: fasmlib.x86asm.net
- Strona domowa Franka Kotlera
- Projekt NASMX - zestaw makr, plików nagłówkowych i przykładów dla NASMa
- Biblioteka FXT - www.jjj.de/fxt - funkcje różnego typu
- x86 Machine Code

Spis treści off-line (Alt+1)
Spis treści on-line (Alt+2)
Ułatwienia dla niepełnosprawnych (Alt+0)