LINUX – sesja

```
svn://lak.ict.pwr.wroc.pl/...  albo
ssh lak.ict.pwr.wroc.pl    # (lak: 156.17.40.28)
      login: nazwa         # konto (ogólne lokalne:"lstudent", zdalne: "student")
      passwd: hasło        # hasło (ogólne: „stud013")
```

LINUX – użyteczne komendy ([..] – opcja)

```
ls [-al]                   # drukuj zawartość kartoteki [all, long - szczegółowo]
ls [-al] ab*/*ab           # lista plików, których nazwa zaczyna/kończy się od ab
find filename              # wyszukaj plik o nazwie filename (dozwolone *)
pwd                        # wypisz nazwę katalogu roboczego
cd                         # zmiana katalogu (cd .. - zmień na katalog nadrzędny)
cat filename               # drukuj zawartość pliku tekstowego filename
cp filename fcopy          # kopiuj zawartość pliku filename do pliku fcopy
rm filename                # usuń plik filename
clear                      # wyczyść ekran
history [n]                # wyświetl historię [n] ostatnich poleceń
history -c                 # wyczyść historię poleceń
↑/↓ (NumPad)               # nawigacja po historii komend

help command               # pomoc dla komendy zewnętrznej (powłoki .bash)
man command                # wywołanie instrukcji obsługi polecenia (manual-a)
   command --help          # wywołanie listy opcji polecenia
 „Ctrl"+D/„Ctrl"+C         # przerwanie procesu
exit                       # zamknięcie sesji i wylogowanie

mc                         # Midnight Commander (podobny do Norton Commandera)
mcedit [filename]          # (niezalecany) edytor tekstowy mc (nakładka na vim),
dc/bc                      # (UNIX) kalkulator ekranowy …
nroff/troff                # formatowanie tekstu do wydruku…
lp plik                    # ustaw plik w kolejce do wydruku
```

Tworzenie i uruchamianie programu

```
vim plik.s                 # utwórz plik z tekstem programu „plik.s"
as [..] plik.s -o plik.o   # kompiluj plik źródłowy - wynik w pliku „plik.o"
ld [..] plik.o -o plik     # konsoliduj - wynik w pliku wykonalnym „plik"
./plik                     # uruchom program „plik" z katalogu bieżącego (./)
./plik par1 par2 ...       # uruchom program z parametrami (dostępne na stosie,
                           # tak jak parametry wywołania →→ funkcji)
```

```
                           # przykład pliku wsadowego (do uruchamiania poleceniem make)
plik: plik.o               # reguła konsolidacji (linkowania)
    ld -o plik plik.o      # „Tab" konieczny na początku linii komendy
plik.o: plik.s             # reguła kompilacji (asemblacji)
    as -o plik.o plik.s
```

Edytor vim (wersja rozszerzona `vi`) **[3]**

`vim` [*plik*] # uruchomienie edytora [z nadaną nazwą pliku źródłowego]

tryby: (1) edycja tekstu; [Esc]+ZZ – zakończenie edycji z zapisem do pliku otwarcia
(2) tryb komend (startowy) – przyjmowanie poleceń z klawiatury
(3) polecenie edytowane (zapis, odczyt, wyszukanie, wyjście z vi)) [Esc] - powrór do (2)

`vim` [*plik*]: → (2) → [a] (dołącz) / [i] (wstaw) /… / [p] (wkopiuj) → (1) …(*edycja*)… (1) [Esc] → (2)
(..) [Esc] → (2) ((2) [Esc] →„beep"→ (2) – bez efektu, sygnał dźwiękowy błędnej sekwencji)
(2) [:] / [/] / [?] → (3) – przejście do edycji wiersza poleceń … [Esc] → (2)
(3) :[w]q! [enter] = **w**rite and **q**uit ([zapisz zmiany,] zakończ i wróć do systemu)

(2) Tryb edycji tekstu

((2)[polecenie] → (1) edycja tekstu → [Esc] → (2))

i / **I** – **i**nsert, *wstaw przed kursorem* / *od początku wiersza* (*shift & insert*)

a / **A** – **a**ppend, *wstaw za kursorem* / *na końcu wiersza* (*shift & append*)

x / X – e**x**tract, *usuń znak na pozycji kursora* / *przed kursorem*

p / P – **p**aste *wstaw zawartość bufora za /przed kursorem lub do kolejnego /poprzedniego wiersza*

r / R – **r**eplace, *nadpisz znak na pozycji kursora* / *nadpisz tekst od pozycji kursora*

o / **O** – *wstaw wiersz poniżej* / *powyżej*

(polecenie → akcja (1) → (2))

dd/dw – **d**elete, *usuń cały wiersz* / *słowo wskazane przez kursor*

d0 / **d$** – *usuń od początku wiersza do kursora* / *od kursora do końca wiersza* (także **D**)

yy/yw – **y**ank, *kopiuj do bufora*: *aktualny wiersz* / *słowo wskazane przez kursor*

y0 / y$ – *kopiuj od początku wiersza do kursora* / *od kursora do końca wiersza*

c.. – **c**hange, *zamień (usuń poprzedni tekst i wpisz nowy*) (także **d..i** lub **d..I**)

cc – *zamień aktualny wiersz* (także [**ddI**]), **cw** – *zamień słowo wskazane przez kursor* (także **dwi**)

c0 / c$ – *zamień od początku wiersza do kursora* (także **d0i**) / *od kursora do końca wiersza* (także **d$i**)

J – **j**oin, *przyłącz następny wiersze* (usuń znak NewLine)…

u – **u**ndo, *anuluj*

^r – **r**edo, *powtórz*

Manewrowanie kursorem

Strzałkami: (znak) ← → ↑ ↓ (słowo) Ctrl+← Ctrl+→ albo z klawiatury

(3) Polecenia edytowane (z parametrami)

:r *plik* – wstaw (czytaj) plik

:w *plik* – **w**rite, zapisz do pliku (**:w!** *nazwa* – nadpisanie istniejącego pliku *nazwa*)

:q – **q**uit, bez aktualizacji po ostatnim zapisie zmian (**:q!** – **q**uit bezwarunkowo, bez zapisu zmian)

:wq [*plik*] – **w**rite & **q**uit, zapisz do pliku otwarcia lub wskazanego (*plik*) i zakończ (także **:x**)

ZZ – bezwarunkowe zakończenie z zapisem zmian (jeśli plik ma nazwę) (= **:wq!**)

:help – wezwanie pomocy (**F1** – ekran pomocy) wyjście przez **:q**

:s/wzorzec/zamiennik[/**g**] – szukaj i zastąp (/**g** wszystkie wystąpienia w wierszu)

:1,$s/wzorzec/zamiennik[/**g**] – szukaj i zastąp w całym pliku (od wiersza 1 do ostatniego)

:j,k **m** t /**:**j,k **co** t – przesuń (**m**ove) /kopiuj (**co**py) wiersze od j do k począwszy od linii t

/*wzorzec* | **?***wzorzec* – szukaj w pliku (tekście) wzorca od kursora do końca | początku

n / **N** – *powtórz ostatnie wyszukiwanie w tym samym / przeciwnym kierunku*

Opcje **vim** *(w trybie 2)*, niedostępne w edytorze **vi**

– klawisze **Del** i **Backspace**

– Shift+Ins (w trybie wstawiania tekstu: **a**/**i** …) – wstaw zawartość bufora

umożliwia kopiowanie tekstu ASCII z Win XP (Notepad) do edytora Ubuntu LINUX:

Notepad (*zaznacz*: Ctrl-A → *wytnij do bufora*: Ctrl-C) → `vim` / `mcedit` (otwórz plik → Shift-Ins)

– zaznaczenie myszą tekstu w oknie terminala (np. przez PuTTy) i skopiowanie do bufora:

umożliwia kopiowanie tekstu ASCII z okna LINUX (`mcedit`) do Win XP (Notepad):

vim / `mcedit` „plik" → okno terminala - mysz-R: „Copy All to Clipboard" → Notepad (Ctrl-V)

Uwaga: plik LINUX musi kończyć się pustą linią

Tryb poleceń – manewrowanie kursorem z klawiatury

[**l**] – →[1] (1 znak w prawo, także [space])

[**h**] – ←[1] (1 znak w lewo, także [backspace])

[**j**] / – ↓[1] (1 linia w dół)

[**k**] – ↑[1] (1 linia w górę)

[**-**] – ←←↑[1] (początek wiersza, linia w górę)

[**+**] – ←←↓[1] (początek wiersza, linia w dół = ↵)

[**0**] – ←← (początek wiersza)

[**$**] – →→ (koniec wiersza)

[**w**] – początek kolejnego słowa

[**b**] – najbliższy początek słowa

[Ctrl+**f**] / [Ctrl+**d**] – ekran / pół ekranu w dół

[Ctrl+**b**] / [Ctrl+**u**] – ekran / pół ekranu w górę

#[**G**] – początek wiersza o numerze #; 1[G] – początek pliku, [G] – koniec pliku

[**H**] – początek bieżącego ekranu

Kompilacja (as), konsolidacja (ld) i uruchamianie programu

```
!! Utwórz kopię zapasową pliku źródłowego (:w plik, :w! kopia) lub  cp plik kopia
as -o plik.o plik.s        # plik wynikowy: plik.o, plik źródłowy: plik.s
  as plik.s                # domniemany plik wynikowy: a.out
as -g -o prode.o prode.s   # asemblacja dla gdb (generuje tablice dla debuggera)
inne opcje asemblera       (--help    # pokaż listę opcji)
  --32                     # generuj kod 32 bitowy (na maszynę 64b AMD)
  --gstabs+                # opcja gdb dla kompilacji z optymalizacją
                                    sourceware.org/binutils/docs2.17/as/index.html

acc –s –m32 a.c            gcc –S [-m32] a.c  (mv  a.s a32.s  - kopia 32-b)

ld -o plik plik.o          # plik wykonalny: plik,       plik wynikowy: plik.o
opcje konsolidatora        (--help    # pokaż listę opcji)
   -m EMULATION...         # wskazanie emulacji (elf_i386/ i386linux/ elf_x86_64)
   -m elf_i386             # generuj kod 32-b na maszynę 64b AMD # w emulacji i386
```

Użycie kompilatora C++ (etykieta startowa: `_main` zamiast `_start`)

```
gcc -g -o plik plik.s      # plik wykonywalny: plik, plik źródłowy: plik.s

UWAGA: Konieczne jest podanie ścieżki dostępu do programu (./ - bieżący)
   ./plik      # uruchomienie program z bieżącego (./) katalogu
   echo $?     # 8-bitowy status zakończenia programu z %bl (to nie jest wynik!!)
```

Typowe błędy kompilacji:

- błędy składni (niepoprawne instrukcje, zła kolejność argumentów)
- brak symboli specjalnych (% przed nazwą rejestru, $ przed wartością stałej)
- „unexpected end of file" – brak nowej linii w pliku źródłowym `filename.s`
- błędny kod znaku (podczas transferu z innego środowiska, np. Windows)

Typowe błędy konsolidacji:

- wadliwa struktura programu (brak dyrektywy **glob(a)l**, brak otwarcia sekcji)
- brak definicji używanych symboli (domyślnie zewnętrzne dla kompilatora)

Typowe błędy wykonania:

„segmentation fault" (podczas wykonania)

- wskaźnik poza obszarem zmiennych (błąd indeksowania zmiennej)
- zmienne w sekcji **.text** (chroniona przed zapisem)
- brak dyrektywy **.data** przed deklaracją danych (deklaracja danych poza sekcją)
- **stos** programowy – przekroczenie dna stosu lub nadmierna rozbudowa
- błędne użycie funkcji systemowej (niepoprawne parametry)
- brak wywołania funkcji **exit** (!) (błędny nr lub argument **int $syscall**)

„floating point error" – błąd obliczeń zmiennoprzecinkowych lub dzielenie przez 0
„arithmetic error" – błąd obliczeń stałoprzecinkowych (za duży iloraz)

Użycie debuggera gdb (linux-asm dla x86/Pentium)

```
## plik prode.s oraz prode powinny być w tym samym katalogu bieżacym ./##
gdb ./plik              # program uruchomieniowy (debugger) z nazwą pliku
                        # wyświetlenie nagłówka: oczekiwanie na akcję: (gdb) ...
(gdb) break xyz         # ustawienie pułapki w miejscu xyz
                        # xyz = etykieta albo nazwa_funkcji albo nr linii
(gdb) clear xyz         # skasowanie pułapki w miejscu xyz
(gdb) disable x y …     # zamaskowanie pułapki w miejscu x, y, … (lub wszystkie)
(gdb) enable x y …      # odmaskowanie pułapki w miejscu x, y, … (lub wszystkie)

(gdb) run               # uruchomienie załadowanego programu z poziomu (gdb)
                        # komunikat: „Starting program …" – w razie zablokowania:
  …                     # Ctrl-C; „Program received signal SIGINT, Interrupt."#
(gdb) step              # praca krokowa - Wynik: (nr_linii    instrukcja)
                          (gdb) # oczekiwanie na akcję
(gdb) next              # praca krokowa – funkcja jako pojedyncza instrukcja!
(gdb) finish            # wyjście z pracy krokowej
(gdb) continue          # wznów wykonanie po zatrzymaniu w pułapce
(gdb) kill              # przerwij wykonanie programu

(gdb) print/w $reg      # wyświetl rejestr %reg (parametr $reg! w formacie
                        w: d – dziesiętnie, x – szesnastkowo (domyślny),
                        c – znakowo, o – ósemkowo, f – zmiennoprzecinkowo)
(gdb) print zmienna     # wyświetl wartość zmiennej
(gdb) x/rsf &addr       # wyświetl stan r jednostek pamięci o rozmiarze s (b –
                        bajt, h – półsłowo, w – słowo, g – dwusłowo) w formacie f
                        jak wyżej, a także i = instrukcja (r=1, s=w można pominąć)
(gdb) x &addr           # wyświetl stan r jednostek pamięci o rozmiarze s (b –
(gdb) display/w arg     # w – jak print, ale po każdej pułapce (stopie)
(gdb) display n &addr   # n komórek od adresu addr
(gdb) info              # informacja o: rejestrach, pamięci,
(gdb) info reg          # informacja o rejestrach
(gdb) info breakpoints  # informacja o ustawionych pułapkach

(gdb) list              # listing programu
(gdb) help [cmd]        # lista poleceń lub opis polecenia debugera [cmd}
(gdb) <ENTER>           # powtórz poprzednią akcję, jeśli jest to sensowne
(gdb) quit              # wyjście z programu debuggera

$reg – rejestr
*nazwa – wskaźnik
&zmienna – wartość zmiennej
```

Konwencje asemblera

% - wskazanie rejestru (np. %esi, %ah, %esp, %eax,…)
$ - wskazanie stałej w treści instrukcji (np. $0, $wart, $'s'- kod ASCII litery s)

Zapis wartości liczbowych (znak liczby opcjonalny) i kodów

```
[-]zc..c              # liczba dziesiętna, z=1,…,9, c=0,1,...,9, np. -32, 15
0d[-]c..c             # liczba dziesiętna (decimal), c=0,1,…,9, np. -32, 0d713
0x[-]h..h             # liczba szesnastkowa (hexadecimal), h=0,1,…,9,a,b,…,f
0q..q                 # liczba ósemkowa, q=0,1,…,7, np. 031,
0bt..t                # liczba dwójkowa (bin[ary]), t=0,1, np. 0b1011
0f[-]i,fE[-]e         # liczba float, i,f,e dziesiętnie np. -314,59 x10-2 –
'β'                   # kod ASCII znaku alfanumerycznego β
"tekst"               # ciąg kodów ASCII znaków alfanumerycznych
```

UWAGA: specyfikacja każdej stałej jako argumentu instrukcji musi być poprzedzona znakiem $

Zapis znaków specjalnych ASCII w tekście (\ ="escape" – następny to specjalny) – konwencja LINUX

```
\ddd  # 3 cyfry ósemkowe (kod ASCII)   \xDD # 2 cyfry szesnastkowe
\n    # (\x0A=\012) LF,NL, new line    \0   # (\x00=\000) NUL, koniec rekordu
\b    # (\x08=\010) BS, backspace      \t   # (\x09=\011) HT,TAB, tabulation
\f    # (\x0C=\014) FF, form feed      \r   # (\x0D=\015) CR, carriage return
\\, \", \?              # ukośnik, apostrof, znak zapytania (znaki specjalne)
```

Obsługa kodów ASCII

Konwencja asemblera **as** (AT&T) [także ASM/TASM/MASM (Borland) z dokładnością do składni]

'znak' – 8-bitowy kod ASCII symbolu, np '0'= 0x30 (48, 0d48, 060), 'A'= 0x41 (65, 0d65, 0101)

obliczenie wartość znaku hex

```
slot_c = 'A'-'9'-1    # odległość cyfr hex od dec = 7, więc lit_hex ma wartość 'lit_hex' – 0x37
stand = 'a'-'A'       # 'a'-'A' = 0x20=0b00100000, 'a'XOR'A'  # standaryzacja kodu litery na małe
mask = 0xDF           # NOT 0x20 = 0xDF = 0b1101111 – maska „małe na wielkie"
key: .ascii "... "    # uniwersalne przetwarzanie kodu litery dużej lub małej jako cyfry hex

movb key(...), %al    # kopia znaku z bufora key do rejestru al
cmpb $'0',%al         # sprawdź, czy to kod cyfry dziesiętnej
jlt out               # wyjdź jeśli to nie jest kod cyfry
cmpb $'9',%al         # sprawdź, czy to kod cyfry dziesiętnej
jle hop               # jeśli to kod cyfry przejdź do obliczenia
andb $mask, %al       # nie-cyfra dziesiętna, zamiana litery na wielką
cmpb $'F',%al         # sprawdź, czy to kod cyfry hex
jgt out               # wyjdź jeśli to nie jest kod cyfry hex
subb $slot, %al,      # redukcja kodu wielkiej hex, 'A' ... 'F' uzyskuje  kod  0x3A ... 0x3F
hop:                  #
andb $0xF, %al        # obliczenie wartości cyfry dziesiętnej lub hex w bajcie
```

Struktura programu w języku asemblera ATT – linux-asm dla IA-32 (80386+/Pentium)

Dyrektywy organizacyjne

```
.globl _start|_main      # etykieta startowa (_start – dla as, _main – dla gcc)
.data /.section data     # sekcja danych programu, zawiera deklaracje zmiennych
.code32                  # wymusza generowanie wpasowanego kodu danych (32-b)
.text /.section text     # sekcja algorytmu (tekst programu) – sekcja obowiązkowa
.bss /.section bss       # bufor nieinicjowanych danych globalnych (s.55/61 [1])

.type nf @function       # deklaracja funkcji nf udostępnianej poza plikiem
.globl etykieta          # dyrektywa udostępnienia etykiety poza plikiem
(.extern etykieta)       # wskazanie definicji etykiety poza plikiem (domniemane)
.include FILE            # wstawia specyfikowany plik jako część tekstu programu.
.section NAME            # deklaracja początku sekcji NAME
```

Stała, łańcuch, zmienna, jej adres i rozmiar (inicjalizacja obowiązkowa)

```
.equ VAL, stala          # lokalne przypisanie wartości (.data)
VAL = stala              # globalne przypisanie wartości (.data, .text)
napis: .ascii "Hi\n"     # łańcuch znaków ASCII (txt: .ascii = txt: .ascii „\n")
dane:                    # deklaracja zmiennej (bloku danych) o adresie „dane":
.typ, lista [,lista]     # inicjalizacja obowiązkowa, typ=.byte|.ascii|.long|.float
.rept  (ile_razy)        # liczba powtórzeń wartości podanej niżej
.(typ) value             # wartość powtarzana
.endr                    # koniec powtórzenia
dane_size =.-dane        # bezpośrednio za deklaracją zmiennej: obliczany przez as
                         # rozmiar zmiennej w bajtach (nazwa zmiennej bez znaku $)
   dane                  # wartość pierwszego elementu zmiennej „dane"
   $dane                 # adres pierwszego bajtu zmiennej „dane" (wskaźnik)
   3145                  # wartość zmiennej spod adresu 3145 (ósemkowo!)!!!
```

Rezerwacja bufora w pamięci (inicjalizacja opcjonalna)

```
blok:                    # deklaracja bufora pamięci o nazwie blok w sekcji danych
.space num [, fill]      # num – rozmiar [B], fill – wartość wypełnienia (dom. 0)
(.skip num)              # rezerwacja bloku bez inicjalizacji
```

Rezerwacja bufora danych nieinicjowanych – sekcja bss

```
.equ B_SIZE, num         # z wielu powodów num nie powinno przekraczać 16000.
.bss                     # początek sekcji bufora (bss) poza sekcją danych
.lcomm ADDR, B_SIZE      # alokuje w pamięci bufor ADDR o rozmiarze B_SIZE bajtów
```

Funkcje standardowe – syscall32 (linux-asm dla IA-32: x86/Pentium) – tryb 32-bitowy

```
movl $FUNC_ID, %eax     # nr funkcji do %eax
movl $FILE_ID, %ebx     # identyfikator (pliku) do %ebx
movl $PAR1, %ecx        # parametr 1, np adres bufora, do %ecx
movl $PAR2, %edx        # parametr 2, np. rozmiar bufora, do %edx
int $SYSCALL32          # ogólne wywołanie funkcji

SYSCALL32 = 0x80        # sysfun: nr funkcji w %eax, parametry: %ebx, %ecx, %edx
EXIT = 1                # nr funkcji restartu (=1) – zwrot sterowania do s.o.
STDIN = 0               # nr wejścia standardowego (klawiatura) do %ebx
READ = 3                # nr funkcji odczytu wejścia (=3)
STDOUT = 1              # nr wyjścia standardowego (ekran tekstowy) do %ebx
WRITE = 4               # nr funkcji wyjścia (=4)
RWX = (0666)            # kod zezwoleń: (0666 (ósemkowo) - „wszystko”)
MODE =                  # tryb dostępu (w %ecx)
OPEN = 5                # nr funkcji otwierania pliku (=5)
CLOSE = 6               # nr funkcji zamykania pliku (=6)
.section .data          # przygotowanie bufora wejścia
BUFOR: .space BUF_SIZE  # deklaracja bufora wejścia
[BUF_SIZE =.- BUFOR]    # obliczony rozmiar bufor (w bajtach) jeśli trzeba
.section .text          # użycie funkcji EXIT, READ, WRITE, OPEN w programie
...
movl $EXIT, %eax        # restart – obowiązkowe zakończenie programu
[movl $num, %ebx]       # w %ebx kod stanu, „echo $” zwraca %bl (8 niższych bitów)
int $SYSCALL32          # ogólne wywołanie funkcji

                        # czytanie wejścia do zapełnienia bufora lecz najwyżej
movl $READ, %eax        # do znaku „ENTER” (końca pliku - znaku 0)
movl $STDIN, %ebx       # identyfikator pliku (STDIN=0) do %ebx
movl $BUFOR, %ecx       # adres początku bufora ($BUFOR) do %ecx
movl $BUF_SIZE, %edx    # rozmiar bufora w bajtach ($BUF_SIZE) do %edx
int $SYSCALL32          # w %eax zwraca liczbę wczytanych znaków

movl $WRITE, %eax       # wyprowadzenie zawartości bufora na wyjście
movl $STDOUT, %ebx      # identyfikator pliku (STDOUT=0) do %ebx
movl $BUFOR, %ecx       # adres początku bufora ($BUFOR) do %ecx
movl $BUF_SIZE, %edx    # rozmiar bufora w bajtach ($BUF_SIZE) do %edx
int $SYSCALL32          # w %eax zwraca liczbę przesłanych znaków

movl $RWX, %edx         # kod zezwoleń do %edx
movl $MODE, %ecx        # kod trybu dostępu do %ecx
movl $NAME, %ebx        # pierwsze litery nazwy pliku do %ebx
movl $OPEN, %eax        # otwarcie pliku (do zapisu/odczytu) – zwraca deskryptor
int $SYSCALL32          # deskryptor pliku w %eax
```

Najważniejsze funkcje systemu Linux (Table C-1. Important Linux System Calls [1])

nazwa	%eax	%ebx	%ecx	%edx	Uwagi
exit	1	status (int)			restart – zwrot sterowania do s.o.
read	3	numer pliku[1)]	adres bufora	rozmiar bufora	odczyt pliku – zwraca w %eax l.bajtów
write	4	numer pliku[1)]	adres bufora	rozmiar bufora	zapis do pliku – zwraca w %eax l.bajtów
open	5	nazwa pliku[2)]	lista opcji[3)]	kod zezwoleń (lub 0666)	zwraca w %eax deskryptor pliku o danej nazwie lub kod błędu.
close	6	deskryptor pliku			Zamyka plik[1)] o danym numerze.
chdir	12	nazwa katalogu[4)]			Przełącza do wskazanego katalogu.
getpid	20				Identyfikator ID bieżącego procesu.
mkdir	39	nazwa katalogu[4)]	kod zezwoleń		Tworzy katalog, zakładając, że katalogi nadrzędne (ścieżka) już istnieją.
rmdir	40	nazwa katalogu[4)]			Usuwa katalog.
brk	45	adres pułapki			Ustanawia pułapkę (*system break*) (ostatni adres sekcji danych). Jeśli %ebx=0, zwraca bieżący adres pułapki

[1)] nadawany przez LINUX deskryptor (numer) pliku (*file descriptor*), [2)] pierwsze litery nazwy pliku
[3)] opcje dostępu – odczyt lub/i zapis, [4)] pierwsze litery nazwy katalogu, [5)] 0 – absolutny, 1 – względny

Niektóre funkcje zwracają **wynik** w rejestrze **eax**. Prawie kompletna lista funkcji systemu jest na stronie http://www.lxhp.in-berlin.de/lhpsyscal.html. Informacje o funkcjach systemowych z części 2 UNIX manual, z odwołaniami do wywołań funkcji z poziomu jezyka C, zwraca polecenie `man 2 SYSCALLNAME`. Na stronie http://www.faqs.org/docs/kernel_2_4/lki-2.html#ss2.11 (Linux Kernel 2.4 Internals, section on how system calls are implemented) są informacje o implementacji Linux-owej.
Kod ASCII jest wypierany przez standard Unicode. UTF-8 jest częściowo kompatybilny z ASCII (zgodność dla znaków łacińskich, rozszerzenia wielobajtowe dla innych). UTF-32 wymaga zawsze 4 bajtów na znak. Windows® używa UTF-16, który jest kodem o zmiennej długości (najmniej 2 bajty na znak, znaki ASCII poprzedza bajt zerowy). Dobrym podręcznikiem Unicode jest: Joe Spolsky, "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)" (http://www.joelonsoftware.com/articles/Unicode.html.)

Funkcje standardowe – syscall (linux-asm dla IA-32e x86-64) – tryb 64-bitowy

```
STDIN = 0               # nr wejścia standardowego (klawiatura) do %ebx
STDOUT = 1              # nr wyjścia standardowego (ekran tekstowy) do %ebx
READ = 0                # nr funkcji wejścia (=0) – odczyt z pliku
WRITE = 1               # nr funkcji wyjścia (=1) – zapis do pliku
OPEN = 2                # nr funkcji otwierania pliku (=2)
CLOSE = 3               # nr funkcji zamykania pliku (=3)
                        # ogólna struktura dla x86-64 (IA-32e)–funkcje podstawowe
movl $FUNC_ID, %rax     # numer funkcji do %rax (%eax w IA-32)
movl $FILE_ID, %rdi     # identyfikator pliku (STDIN=0, STDOUT=1) (%ebx w IA-32)
movl $PAR1, %rsi        # parametr 1, np. adres bufora we/wy (%ecx w IA-32)
movl $PAR2, %rdx        # parametr 2, np. rozmiar bufora (%edx w IA-32)
syscall                 # ogólne wywołanie funkcji (nowa instrukcja IA-32e)

sysexit                 # zwrot sterowania (nowa instrukcja IA-32e)
```

Pierwsze programy

/* wyświetlenie komunikatu */

```
SYSCALL32 = 0x80          # nr wywołania systemowego
EXIT = 1                  # nr funkcji restartu (=1)
WRITE = 4                 # nr funkcji „pisz”
STDOUT = 1                # nr wejścia standardowego
.data
komunikat: .ascii “Hello\n”        # tekst komunikatu,
rozmiar = . – komunikat   # obliczenie liczby znaków komunikatu
.text
.globl _start
_start:
movl $rozmiar, %edx       # rozmiar bufora w bajtach ($rozmiar) do %edx
movl $komunikat, %ecx     # adres startowy bufora ($komunikat) do %ecx
movl $STDOUT, %ebx        # nr wejścia do %ebx (STDOUT=1)
movl $WRITE, %eax         # nr funkcji do %eax (=4)
int $SYSCALL32            # syscall – ogólne wywołanie funkcji
movl $EXIT, %eax          # numer funkcji (=1) do %eax
int $SYSCALL32            # syscall – ogólne wywołanie funkcji
```

/* wczytanie tekstu – przetworzenie – wyświetlenie tekstu przetworzonego */

```
SYSCALL32 = 0x80        # sysfun: nr funkcji w %eax, parametry: %ebx, %ecx, %edx
EXIT = 1                # nr funkcji restartu (=1) – zwrot sterowania do s.o.
STDIN = 0               # nr wejścia standardowego (klawiatura) do %ebx
READ = 3                # nr funkcji odczytu wejścia (=3)
STDOUT = 1              # nr wyjścia standardowego (ekran tekstowy) do %ebx
WRITE = 4               # nr funkcji wyjścia (=4)
BUF_SIZE = 254          # rozmiar bufora (w bajtach/znakach ASCII) – max 254
DISTANCE = ’z’-’a’+1
COMPL =’z’+’a’
.data                   # przygotowanie bufora wejścia
TEXT_SIZE .long 0
BUFOR: .space BUF_SIZE  # deklaracja bufora wejścia
[BUF_SIZE =.- BUFOR]    # obliczony rozmiar bufor (w bajtach) jeśli trzeba
ERR_MSG .asci „Niepoprawny znak\n”
ER_LEN=.- ERR_MSG
.text                   # użycie funkcji EXIT, READ, WRITE, OPEN w programie
.globl _start
_start:
movl $READ, %eax        # do znaku „ENTER” (końca pliku - znaku 0)
movl $STDIN, %ebx       # identyfikator pliku (STDIN=0) do %ebx
movl $BUFOR, %ecx       # adres początku bufora ($BUFOR) do %ecx
movl $BUF_SIZE, %edx    # rozmiar bufora w bajtach ($BUF_SIZE) do %edx
int $SYSCALL32          # ogólne wywołanie funkcji
```

```
movl %eax, TEXT_SIZE    # !!  w %eax zwraca liczbę wczytanych znaków

CALL ENCRYPT

movl $WRITE, %eax       # wyprowadzenie zawartości bufora na wyjście
movl $STDOUT, %ebx      # identyfikator pliku (STDOUT=0) do %ebx
movl $BUFOR, %ecx       # adres początku bufora ($BUFOR) do %ecx
movl TEXT_SIZE, %edx    # faktyczna liczba znaków w bajtach do %edx
int $SYSCALL32          # ogólne wywołanie funkcji

movl $EXIT, %eax        # restart – obowiązkowe zakończenie programu
[movl $num, %ebx]       # w %ebx kod stanu, „echo $" zwraca %bl (8 niższych bitów)
int $SYSCALL32          # ogólne wywołanie funkcji

ENCRYPT:                # wielka litera – szyfrowanie, mała litera – deszyfrowanie
movl $0, %edi           # pierwszy znak jest kluczem
movb BUFOR(,%edi,1), %bl   # klucz (pierwszy znak) do %bl
or $0x40, %bl           # duże mają kody 0x41, 0x42,   0x5A
cmpb $'Z', %bl          # czy wielka?
jbe szyfruj             # jeśli duża szyfowanie
subb $COMPL, %bl        # obliczenie x='key'-('a'+'z')
negb %bl                # dopełnienie: klucz = 0 – x = 'a'+'z'-'key'
szyfruj:
incl %edi               # indeksacja wskaźnika znaku w buforze
movb BUFOR(,%edi,1), %al
orb $0x20, %al          # ujednolicenie – wszystkie litery duże
cmpb $'Z', %al          # czy znak jest literą
ja error
subb $'A', %al          # czy znak jest literą
jb error
addb %bl, %al           # szyfrowanie – dodanie klucza
cmpb $'Z', %al
jbe cykl
subb $DISTANCE, %al     # korekta cyklu
cmpl TEXT_SIZE, %edi
jbe cykl
ret
error:                  # znak nie jest literą
movl $WRITE, %eax       # wyprowadzenie zawartości bufora na wyjście
movl $STDOUT, %ebx      # identyfikator pliku (STDOUT=0) do %ebx
movl $ERR_MSG, %ecx     # adres początku bufora ($BUFOR) do %ecx
movl $ER_LEN, %edx      # faktyczna liczba znaków w bajtach do %edx
int $SYSCALL32          # ogólne wywołanie funkcji
ret
```

Koncepcja pliku w systemie UNIX / LINUX – obsługa plików z poziomu asemblera

Pliki UNIX /LINUX, niezależnie od rodzaju i sposobu wytworzenia, są dostępne jako łańcuch bajtów. Dostęp do pliku rozpoczyna jego otwarcie przez podanie nazwy. Wtedy system operacyjny podaje (tymczasowy) numer, zwany deskryptorem pliku (*file descriptor*), używany jako odsyłacz do pliku podczas jego użycia. Po zapisie lub odczycie plik należy zamknąć, co unieważnia deskryptor.

Postępowanie z plikami (dealing with files)

1. Podaj do systemu Linux nazwę pliku i żądany tryb otwarcia (odczyt, zapis, odczyt i zapis, utwórz go jeśli nie istnieje, itd.). Wykonuje to funkcja `open` (`%eax` = 5), która pobiera nazwę pliku, kod trybu oraz zbiór zezwoleń (*permissions set*) jako parametry. Adres pierwszego znaku nazwy pliku powinien być w `%ebx`. W `%ecx` należy wpisać kod trybu użycia (0 dla plików, które będą tylko odczytywane, **0**3101 dla plików, które będą zapisywane (! zero wiodące jest konieczne). Zbiór zezwoleń ma być wpisany do `%edx`. Jeśli nie znasz kodów zezwoleń UNIX / LINUX, wpisz kod 0666 (! zero wiodące jest konieczne – patrz [1: rozdział 10, sekcja *Truth, Falsehood, and Binary Numbers*].
2. LINUX zwróci w `%eax` deskryptor pliku (*file descriptor*), który jest odsyłaczem do tego pliku.
3. Teraz można wykonać funkcje read / write (`%eax` = 3/4), wpisując deskryptor pliku do `%ebx`, adres bufora danych do `%ecx`, rozmiar bufora do `%edx`. Funkcja (`read/write`) zwróci (w `%eax`) liczbę znaków przeczytanych z pliku, albo kod błędu (*error code*), który jest liczbą ujemną w systemie U2).
4. Po zakończeniu operacji plik należy zamknąć za pomocą funkcji `close` (`%eax` = 6), której jedynym parametrem jest deskryptor pliku (w `%ebx`). Deskryptor jest odtąd unieważniony.

Bity kodu opcji dla funkcji lub *system call* nazywa się *flags*. Parametrem funkcji systemowej `open` jest lista flag tworząca kod zezwoleń zapisany w rejestrze %edx. Niektóre z tych kodów (ósemkowo) to:

`O_RDONLY`	– 00	– tylko odczyt (read-only mode)
`O_WRONLY`	– 01	– tylko zapis (write-only mode)
`O_RDWR`	– 02	– zapis i odczyt (both reading and writing)
`O_CREAT`	– 0100	– utworzenie pliku, jeśli nie istnieje (create the file if it doesn't exist)
`O_TRUNC`	– 01000	– skasuj zawartość pliku, jeśli istnieje (erase the contents of the file)
`O_APPEND`	– 02000	– Dopisywanie na końcu pliku (start writing at the end of the file)

Flagi można logicznie sumować (OR), np. `O_WRONLY` (01) OR `O_CREAT` (0100) daje kod 0101.

Pliki standardowe i specjalne

W filozofii Linux/UNIX każde źródło danych (połączenie sieciowe, urządzenia) jest plikiem.
Komunikacja jest realizowana za pomocą specjalnych plików, zwanych rurami (*pipes*). Niektóre z nich wymagają specyficznych sposobów tworzenia i otwierania (nie używa się funkcji open), ale mogą być czytane i zapisywane normalnie (read / write z system calls).
W systemie Linux programy startują zwykle z trzema deskryptorami plików standardowych:
STDIN – file descriptor 0
Wejście standardowe (*standard input*), zawsze w trybie read-only, zwykle klawiatura.
STDOUT – file descriptor 1
Wejście standardowe (*standard output*), zawsze w trybie write-only file, zwykle monitor.
STDERR – file descriptor 2
Błąd standardowy (*standard error*), plik write-only, zwykle monitor. Wyjścia przetwarzania wędrują do STDOUT komunikaty o błędach do STDERR. Można je rozdzielić na 2 osobne miejsca.
Każdy standardowy strumień danych można w Linux-ie przekierować do innego pliku.

Obsługa plików

```
/* Nazwy plików przekazywane przez stos przed wywołaniem procedury */
SYSCALL32 = 0x80          # syscall - parametry: %eax, %ebx, %ecx, %edx
EXIT = 1                 # nr funkcji restartu (=1) - zwrot sterowania do s.o.
STDIN = 0                # nr wejścia standardowego (klawiatura) do %ebx
READ = 3                 # nr funkcji odczytu wejścia (=3)
STDOUT = 1               # nr wyjścia standardowego (ekran tekstowy) do %ebx
WRITE = 4                # nr funkcji wyjścia (=4)
OPEN = 5                 # (opcje otwarcia: /usr/include/asm/fcntl.h)
CLOSE = 6                # nr funkcji zamknięcia pliku
CR_WRONLY_TR = 03101     # flaga: tylko zapis (notacja ósemkowa!)
RDONLY = 0               # flaga: tylko odczyt, składanie opcji - OR

.section .bss            # bufor danych
.lcomm BUFFER, 500       # (rozmiar musi być <16000)

.text                    # wywołanie z nazwami plików: ./prog file-in file-out
.equ FD_IN, -4           # lokalizacja deskryptora pliku we (par1)
.equ FD_OUT, -8          # lokalizacja deskryptora pliku wy (par1)
.equ ARG_1, 8            # lokalizacja nazwy pliku we (par1)
.equ ARG_2, 12           # lokalizacja nazwy pliku wy (par1)

.globl _start            ###START PROGRAMU###
start:
movl %esp, %ebp          # przechowanie wskaźnika stosu
subl $8, %esp            # miejsce na stosie na deskryptory plików (2*4 bajty)

movl $OPEN, %eax         # otwarcie pliku wejściowego
movl ARG_1(%ebp), %ebx   # nazwa pliku we (file-in) do %ebx
movl $RDONLY, %ecx       # flaga: tylko do odczytu
movl $0666, %edx         # bez znaczenia podczas otwierania
int $SYSCALL32
movl %eax, FD_IN(%ebp)   # zwrócony w %eax deskryptor pliku we do ramki stosu

movl $OPEN, %eax         # otwarcie pliku wyjściowego
movl ARG_2(%ebp), %ebx   # nazwa pliku wy (file-out) do %ebx
movl $CR_WRONLY_TR, %ecx # flaga: tylko do zapisu
movl $0666, %edx         # tryb dla tworzonego pliku (jeśli nowy)
int $SYSCALL32
movl %eax, FD_OUT(%ebp)  # zwrócony w %eax deskryptor pliku wy do ramki stosu

read_loop_begin:         ###ODCZYT BLOKU Z PLIKU WEJŚCIOWEGO###
movl $READ, %eax
movl FD_IN(%ebp), %ebx   # pobranie z ramki deskryptora pliku odczytywanego
movl $BUFFER, %ecx       # adres bufora odczytu
movl $B_SIZE, %edx       # rozmiar bufora odczytu
int $SYSCALL32           # rozmiar bufora odczytu zwracany w %eax
```

```
cmpl $0, %eax              # sprawdzenie, czy osiągnięto koniec pliku EOF
jle end_loop               # jeśli wykryto EOF lub w razie błędu koniec

                           # argumenty wywołania przekazywaneprzez stos
pushl $BUFFER              # adres bufora na stos
pushl %eax                 # rozmiar bufora (zwrócony w %eax) na stos
call convert
popl %edx                  # rozmiar bufora ze stosu do %edx
addl $4, %esp              # przywrócenie wskaźnika stosu %esp

movl $WRITE, %eax          # blok po konwersji do pliku wyjściowego
movl FD_OUT(%ebp), %ebx    # pobranie z ramki deskryptora pliku wynikowego
movl $BUFFER, %ecx         # adres bufora (pliku zapisywanego)
int $SYSCALL32
jmp read_loop_begin        # kontynuacja – następna porcja pliku

end_loop:
movl $CLOSE, %eax          ###ZAMYKANIE PLIKÓW – nie ma potrzeby kontroli
movl FD_OUT(%ebp), %ebx    # błędu, nie ma to tutaj istotnego znaczenia
int $SYSCALL32             # deskryptor pliku wynikowego
movl $CLOSE, %eax
movl FD_IN(%ebp), %ebx
int $SYSCALL32
movl $EXIT, %eax           ###EXIT###
movl $0, %ebx
int $SYSCALL32

convert:                   # jeśli bufor ma długość zero, funkcja nie jest użyta
movl 12(%ebp), %eax        # adres bufora
movl 8(%ebp), %ecx         # rozmiar bufora
decl %eax                  #
convert_loop:
movb -1(%eax,%ecx,1), %dl  # element N jest pod adresem (%eax)+N-1
cmpb $'a', %dl             # kolejny bajt (znak ASCII)
jb next_byte
cmpb $'z', %dl
ja next_byte               # jeśli znak poza ('a' do 'z') weź kolejny
andb $0xDF, %dl            # zamień na dużą i zapisz zwrotnie do bufora
movb %dl, -1(%eax,%ecx,1)
next_byte:
loop convert_loop          # kontynuuj jeśli nie osiągnięto końca bufora (ecx--)
movl %ebp, %esp
pop %ebp
ret
```

Makra

Makro tworzy makrodefinicja (*macrodefinition*) i makrowywołanie (*macrocall*).
Makrodefinicja jest tekstowym opisem treści podobnych fragmentów kodu źródłowego w formie, która umożliwia automatyczne tworzenie podobnych fragmentów kodu, na przykład sekwencji instrukcji, które różnią się tylko argumentami. Makrodefinicję rozpoczyna dyrektywa **.macro** po której następuje **nazwa makra** i lista parametrów formalnych, a kończy dyrektywa **.endm**.
Makrowywołanie to użycie **nazwy makra**, po której następuje specyfikacja parametrów w kolejności podanej w makrodefinicji.

```
#  makrodefinicje #
.macro write str, str_size, dest    # makro o nazwie  „write”
   movl $WRITE, %eax
   movl \dest, %ebx
   movl \str, %ecx
   movl \str_size, %edx
   int $0x80
.endm                               # zakończenie makrodefinicji - dyrektywa .endm

.macro read buf, buf_size, source   # makro o nazwie  „read”
   movl $READ, %eax
   movl \source, %ebx
   movl \buf, %ecx
   movl \buf_size, %edx
   int $0x80
.endm

#     wywołanie     #
write $msg1, $msg1_size, $STDOUT      # prinf("%s", txt_msg1)
                                      # wyświetla txt_msg1 na monitorze (STDOUT)
read $in-buff, $in-buff_size, $STDIN  # scanf("%s", bufor_wej)
                                      # odczytuje znaki z klawiatury (STDIN)
```

Funkcje

stos programowy – dynamiczna struktura danych wspomagająca użycie funkcji
adres powrotu (*return address*) – parametr automatyczny, tworzony na stosie przez wywołanie (call)
zmienne globalne – dostępne i zarządzane na zewnątrz funkcji
zmienne lokalne – używane tylko wewnątrz funkcji, ignorowane po zakończeniu
zmienne statyczne – dostępne tylko wewnątrz funkcji, pamiętane do kolejnego jej wywołania

Konwencje wywołania funkcji w języku C (*calling convention*)

Wskaźnik stosu %esp wskazuje lokalizację bajtu zajmującego szczyt stosu (konwencja Little Endian):

- operacja pushl wykonywana jako przesłanie słowa pod adres –4(%esp), czyli esp–4
- operacja popl wykonywana jako przesłanie słowa spod adresu (%esp)
- dostęp do słów poniżej szczytu stosu – adres N*4(%esp) – N to numer kolejny parametru

Wartość wskaźnika stosu może się zmienić podczas wykonania funkcji (np. wskutek przerwania), więc dostęp do struktur danych funkcji wymaga użycia wskaźnika powiązania dynamicznego (%ebp w konwencji C dla IA-32). Pierwszą instrukcją funkcji musi więc być zachowanie „starego" wskaźnika i załadowanie %ebp nową wartością (którą jest aktualna wartość wskaźnika stosu %esp). Przed zakończeniem funkcji trzeba odtworzyć "stary" %ebp.

Jeśli kod asemblerowy jest wstawiany do programu w języku C należy także pamiętać, że kompilator gcc dla IA-32 [5] przypisuje rejestry %ebp, %esp, %ebx, %edi, %esi funkcji wywołującej, więc funkcja wywoływana powinna chronić nie tylko %ebp, %esp ale też %ebx, %edi i %esi (zachować i odtworzyć przy zakończeniu). Rejestry %eax, %ecx, %edx są przypisane funkcji wywoływanej, więc funkcja wywołująca powinna je przechować (na stosie), jeśli ich używa w chwili wywołania [6].

Funkcja w programie - schemat wywołania

```
.text
.globl _start
_start:
.......                     # wcześniejsza część programu
pushl PAR-N                 # przekazanie parametru nr N
…
pushl PAR-1                 # przekazanie parametru nr 1
call funkcja                # wywołanie, adres powrotu na szczyt stosu
   (addl $N*4, %esp)        # oczyszczenie stosu (jedna z możliwości)
.......                     # obsługa wyników funkcji
```

Struktura funkcji (konwencja C/C++)

```
.type funkcja @function     # deklaracja funkcji
funkcja:
[„push …"]                  # rejestry f. wywołującej na stos (gcc)
pushl %ebp                  # wskaźnik kontekstu poziomu wywołania (powiąz. dynam.)
movl %esp, %ebp             # wskaźnik kontekstu funkcji wywołanej: szczyt stosu
…                           # „ciało funkcji" (dostęp swobodny do kontekstu)
movl %ebp, %esp             # przywrócenie wskaźnika szczytu stosu
popl %ebp                   # odtworzenie "starego" wskaźnik powiązania
[„pop …"]                   # rejestry f. wywołującej ze stosu (gcc)
ret                         # zwrot sterowania do miejsca wywołania, odtąd zmienne
                            # lokalne są niedostępne, na stosie pozostały parametry
```

Funkcja „wyświetl dziesiętnie zawartość rejestru 32-b"

```
SYSCALL32 = 0x80           # nr wywołania systemowego
WRITE = 4                  # nr funkcji „pisz"
STDOUT = 1                 # nr wejścia standardowego
.data
BUFWY: .ascii „          \n"     # miejsce na 10 cyfr
BWY_LEN=.-BUFWY            # rozmiar bufora
.equ PODSTAWA, 10
.equ NUMB_LEN, 10          # 10**9<2**32<10*10 (liczba max 10-cyfrowa w rej. 32-b)

.type p_reg_dec @function  # wyświetlenie zawartości rejestru dziesiętnie
p_reg_dec:
[„push …"]                 # rejestry f. wywołującej na stos (gcc)
pushl %ebp                 # "stary" wskaźnik powiązania dynamicznego na stos
movl %esp, %ebp            # „nowy" wskaźnik (%esp)
movl $PODSTAWA, %ebx
movl $NUMB_LEN-1, %ecx     #
konwert:
movl $0, %edx              # w %edx jest „stara" reszta, trzeba ją wyzerować
div %ebx                   # (%edx) %dl – kolejna cyfra, %eax=iloraz
orb '0', %dl               # kod ASCII cyfry 0 (albo $ZERO jeśli .equ ZERO, 0x30)
movb %dl, BUFWY(%ecx)
dec %ecx
andl %eax, %eax            # iloraz=0 – koniec konwersji
jnz konwert
end_konw:                  # teraz wyprowadź liczbę (kody ASCII z bufora)
movl $BWY_LEN, %edx        # rozmiar bufora w bajtach ($BWY_LEN) do %edx
movl $BUFWY, %ecx          # adres startowy bufora ($BUFWY) do %ecx
movl $STDOUT, %ebx         # nr wejścia do %ebx (STDOUT=1)
movl $WRITE, %eax          # nr funkcji do %eax (WRITE=4)
int $SYSCALL32             # syscall
movl %ebp, %esp            # przywrócenie wskaźnika szczytu stosu (i oczyszczenie)
popl %ebp                  # odtworzenie "starego" wskaźnik powiązania
[„pop …"]                  # rejestry f. wywołującej ze stosu (gcc)
ret                        # zwrot sterowania do miejsca wywołania
```

Funkcje rekurencyjne

Ciało funkcji rekurencyjnej zawiera:

- przekazywanie parametrów na kolejny poziom zagniedżenia (przez stos)
- sprawdzenie warunku zakończenia rekurencji
- wywołanie funkcji lub ominięcie jeśli koniec rekurencji
- wykonanie kolejnych obliczeń
- odtworzenie stosu poprzedniego poziomu
- przekazanie wyniku do poziomu poprzedniego wywołania

```
...                     # wywołanie funkcji
[pushl $PAR]            # parametry funkcji na stos (zwykle statyczne)
pushl $ARG_REK          # argument funkcji na stos
call frecursive         # wywołanie
addl $ST_SIZE, %esp     # oczyszczenie stosu (%esp poniżej parametru wywołania)
...                     # dalszy ciąg programu

.type frecursive @function # deklaracja funkcji
frecursive:
pushl %ebp              # “stary” wskaźnik powiązania dynamicznego na stos
movl %esp, %ebp         # nowy wskaźnik: szczyt stosu – tu się rozpoczyna nowy
movl 8(%ebp), [%eax]    # bieżący argument funkcji (z wnętrza stosu) do %eax
                        ((%ebp):„stary ebp, 4(%ebp): adres powrotu, 8(%ebp):ARG)
                        (w IA-32 słowo=4bajty; w IA-32e słowo=8bajtów)

[cmpl $LAST, %eax]      # warunek końca [w %eax jest bieżący argument funkcji]
[je end_fact]           # koniec sekwencji wywołań rekurencyjnych

[decl %eax]             # obliczenie kolejnego argumentu rekurencji
[pushl ARG]             # kolejny argument rekurencji na stos [ARG=%eax]

call frecursive

[movl 8(%ebp), %ebx]    # obliczenie etapowe
[mull %ebx]             # kolejne obliczenie (ew. imull %ebx, %eax)

end_fact:               # powrót na poprzedni poziom wywołania
movl %ebp, %esp         # przywrócenie wskaźnika szczytu stosu (i oczyszczenie)
popl %ebp               # odtworzenie “starego” wskaźnik powiązania
ret                     # zwrot sterowania do miejsca wywołania
```

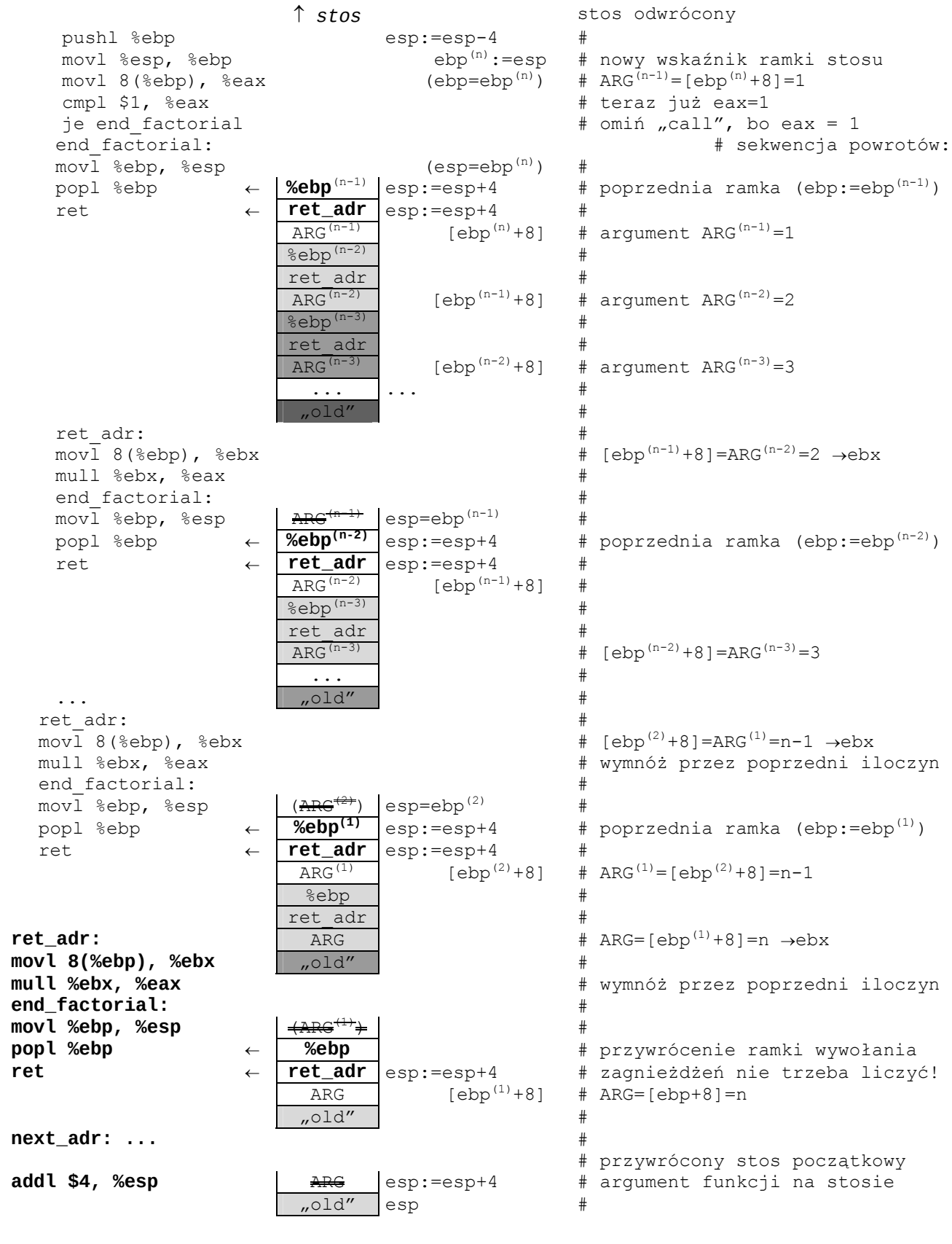
Funkcja rekurencyjna (obliczanie silni)

```
.type factorial @function  # deklaracja funkcji
factorial:
pushl %ebp                 # "stary" wskaźnik powiązania dynamicznego na stos
movl %esp, %ebp            # nowy wskaźnik: szczyt stosu – tu się rozpoczyna nowy
movl 8(%ebp), %eax         # parametr (z wnętrza stosu) do %eax
cmpl $1, %eax              # warunek końca (1!=1), 1 to jednocześnie wartość pocz
je end_factorial           # koniec sekwencji wywołań rekurencyjnych
decl %eax                  # obliczenie kolejnego argumentu rekurencji
pushl %eax                 # kolejny argument rekurencji na stos
call factorial
ret_adr:
movl 8(%ebp), %ebx         # parametr rekurencji do %ebx, bo %eax zawiera wynik
mull %ebx                  # kolejne obliczenie (ew. imull %ebx, %eax)
end_factorial:
movl %ebp, %esp            # przywrócenie wskaźnika szczytu stosu (i oczyszczenie)
popl %ebp                  # odtworzenie "starego" wskaźnik powiązania
ret                        # zwrot sterowania do miejsca wywołania
```

– zagnieżdżanie (budowa stosu)

		stos ↓		komentarz
		„old"	esp	
pushl %eax	→	**ARG**	esp:=esp-4	# argument (ARG=eax) na stos
call factorial	→	**nxt_adr**	esp:=esp-4	# adresu powrotu na stos
nxt_adr: ...				#
		„old"		#
		ARG	[ebp(1)+**8**]	# argument: 8-11 bajt od szczytu
factorial:		nxt_adr	[ebp(1)+4]	# wykonanie 1 kroku rekurencji
pushl %ebp	→	**%ebp**	esp:=esp-4	#
movl %esp, %ebp			ebp(1):=esp	# nowy wskaźnik ramki stosu
				#
movl 8(%ebp), %eax			eax:=[ebp(1)+8]	# argument ARG z ramki do eax
cmpl $1, %eax				#
je end_factorial		„old"		# jeśli eax=1, koniec rekurencji
		ARG	[ebp(1)+8]	#
		nxt_adr	[ebp(1)+4]	#
decl %eax		%ebp		# dopóki eax≠1 zagnieżdżaj
pushl %eax	→	**ARG(1)**	[ebp(2)+8]	# parametr rekurencji na stos
call factorial	→	**ret_adr**	[ebp(2)+4]	# adres powrotu
pushl %ebp	→	**%ebp**(1)	esp:=esp-4	#
movl %esp, %ebp			ebp(2):=esp	# nowy wskaźnik ramki stosu
				#
movl 8(%ebp), %eax		„old"	eax:=[ebp(2)+8]	# argument ARG(1) z ramki do eax
cmpl $1, %eax		ARG		#
je end_factorial		nxt_adr		#
		%ebp		#
		ARG(1)	[ebp(2)+8]	#
		ret_adr	[ebp(2)+4]	#
decl %eax		%ebp(1)		# dopóki eax≠1 zagnieżdżaj
pushl %eax	→	ARG(2)	[ebp(3)+8]	# parametr rekurencji na stos
call factorial	→	**ret_adr**	[ebp(3)+4]	# adres powrotu
pushl %ebp	→	**%ebp**(2)	esp:=esp-4	#
movl %esp, %ebp			ebp(3):=esp	# nowy wskaźnik ramki stosu
movl 8(%ebp), %eax			eax:=[ebp(3)+8]	# argument ARG(2) z ramki do eax
... ...				# kolejne wywołania dopóki eax≠1

– powroty (zwalnianie stosu)



Funkcja wariacje (ang. *variations*) – iteracyjna i rekurencyjna

```
                        # argumenty: zmienne nn i kk; V(nn,kk)=V(n,k)=n!/(n-k)!
                        # V(n,k+1)=(n-k)V(n,k) V(n,n)=P(n)=n!
movl nn, %ebx           # argument nn do %ebx
movl kk, %ecx           # argument kk do %ecx, także licznik iteracji
movl $1, %eax           # V(n,0)=1, V(n,1)=n
cmpl $0, %ecx           # sprawdzenie, czy k=0
jz end
   variter:             # iteracyjne obliczenie wariacji
   mull %ebx            # wymnóż przez kolejny czynnik
   decl %ebx            # kolejny mnożnik k-1
   loop variter         # koniec obliczeń gdy ecx=[kk]

                        # rekurencyjne obliczanie wariacji
pushl kk                # albo „pushl %ecx - argument kk na stos
pushl nn                # albo „pushl %ebx - argument nn na stos
call var                # wywołanie (w rejestrze %eax jest 1)
addl $8, %esp           # zwolnienie stosu, %esp poniżej parametru wywołania
end:
movl %eax, wynik        # wynik do zmniennej wynik

.type var @function     /* *) alternatywne zestawy instrukcji, wymaganie początkowe %eax=1 */
var:
[„push …”]              # rejestry f. wywołującej na stos (gcc)
pushl %ebp              # “stary” wskaźnik powiązania dynamicznego na stos
movl %esp, %ebp         # nowy wskaźnik: szczyt stosu) do %ebx
movl 12(%ebp), %ecx     # parametr k* (z wnętrza stosu) do %eax
movl 8(%ebp), %eax      # parametr n* (z wnętrza to jednocześnie wartość #)
                        # *) movl 8(%ebp), %eax
cmpl $1, %ecx           # warunek końca #)
                        # *) cmpl $0, %ecx
je end_var              # koniec sekwencji wywołań rekurencyjnych
decl %ecx               # obliczenie kolejnego argumentu rekurencji
decl %eax               # obliczenie kolejnego argumentu rekurencji
                        # *) decl %ebx
pushl %ecx              # kolejny argument rekurencji (k*) na stos
pushl %eax              # kolejny argument rekurencji (n*) na stos
                        # *) pushl %ebx
call var
movl 8(%ebp), %ebx      # parametr rekurencji do %ebx, bo %eax zawiera wynik
mull %ebx               # kolejne obliczenie (ew. imull %ebx, %eax)
end_var:                # pierwsze wejście: %eax=n-k+1
movl %ebp, %esp         # przywrócenie wskaźnika szczytu stosu (i oczyszczenie)
popl %ebp               # odtworzenie “starego” wskaźnik powiązania
[„pop …”]               # rejestry f. wywołującej ze stosu (gcc)
ret                     # zwrot sterowania do miejsca wywołania
```

Funkcja kombinacje (ang. *variations*) – iteracyjna i rekurencyjna

```
/* C(n,k)=n!/(k!(n-k)!)= C(n-1,k-1)*n/k*/
/* C(n,k)= {...([n·(n-1)/2]·(n-2)/3) ….·(n-k+1)}/k , albo w odwrotnej kolejności*/
/* C(n,k)= {...([(n-k+1)(n-k+2)/2]·(n-k+3)/3) …· n}/k */
/* max C(n,k)=C(n,n/2). Także 2^k<C(2k+1,k)<k^k, co pozwala oszacować zakres */

...                          # argumenty: zmienne nn i kk; V(nn,kk)=V(n,k)=n!/(n-k)!
movl $0, %eax                # C(n<k,k)=df 0
movl kk, %ecx
movl nn, %ebx
cmpl %ebx, %ecx              #
jl end_combine               # jeśli n<k, to (def) C(n<k,k)=0
subl %ecx, %ebx              # %ebx:= %ebx - %ecx (n:n-k)
cmpl %ebx, %ecx              # n-k>k?
jg hop:                      # jeśli n<k, to (def) C(n<k,k)=0
movl %ebx, kk                # n-k zamiast k
hop:
movl $1, %eax                # C(n,0)=1 (pierwszy iloczyn)
cmpl $0, kk                  #
je end_combine               # jeśli k=0 (lub k=n), to C(n,0)=1
movl nn, %ebx                # przywrócenie n (nie zmienia flag!)
movl $1, %ecx                # dzielnik (1,2,…,k)

   combine:                  # iteracyjne obliczenie kombinacji
   mull %ebx                 # wymnóż przez kolejny czynnik
   divl %ecx                 # wynik w %edx:%eax
   decl %ebx                 # kolejny mnożnik (nn--)
   incl %ecx                 # kolejny dzielnik (licznik++ until kk)
   cmpl %ecx, kk             # powtarzaj, dopóki licznik (%ecx) nie przekroczy k
   jl combine
                             # rekurencyjne obliczanie kombinacji
pushl kk                     # argument k na stos
pushl nn                     # argument n na stos
call combine                 # wywołanie (combine = newton lub pascal)
addl $8, %esp                # zwolnienie stosu, %esp poniżej parametru wywołania
end_combine:
movl %eax, wynik             # wynik

.type pascal @function       # rekurencyjne obliczanie kombinacji wg trójkata Pascala
pascal:                      # C(n,k)= C(n-1,k)+ C(n-1,k-1)
pushl %ebp                   # "stary" wskaźnik powiązania dynamicznego na stos
movl %esp, %ebp              # nowy wskaźnik: szczyt stosu do %ebx
…
call pascal
movl %ebp, %esp              # przywrócenie wskaźnika szczytu stosu (i oczyszczenie)
popl %ebp                    # odtworzenie "starego" wskaźnik powiązania
ret                          # zwrot sterowania do miejsca wywołania
```

```
.type newton @function     # rekurencyjne obliczanie kombinacji
newton:                    # C(n,k)=n!/(k!(n-k)!)= C(n-1,k-1)*n/k
[„push …”]                 # rejestry f. wywołującej na stos (gcc)
pushl %ebp                 # “stary” wskaźnik powiązania dynamicznego na stos
movl %esp, %ebp            # nowy wskaźnik: szczyt stosu stosu) do %ebx
movl 12(%ebp), %ecx        # parametr k* (z wnętrza stosu) do %ecx
movl 8(%ebp), %eax         # parametr n* (z wnętrza stosu) do %ebx
cmpl $1, %ecx              # warunek końca
je end_comb                # koniec sekwencji wywołań rekurencyjnych
decl %ecx                  # obliczenie kolejnego argumentu rekurencji
decl %eax                  # obliczenie kolejnego argumentu rekurencji
pushl %ecx                 # kolejny argument rekurencji (k*) na stos
pushl %eax                 # kolejny argument rekurencji (n*) na stos
call newton
movl 8(%ebp), %ebx         # parametr rekurencji do %ebx, bo %eax zawiera wynik
movl 12(%ebp), %ecx        # parametr rekurencji do %ecx
mull %ebx                  # kolejne obliczenie (ew. imull %ebx, %eax)
div %ecx
end_comb:                  # pierwsze wejście: %eax=n-k+1, %ecx=1
movl %ebp, %esp            # przywrócenie wskaźnika szczytu stosu (i oczyszczenie)
popl %ebp                  # odtworzenie “starego” wskaźnik powiązania
[„pop …”]                  # rejestry f. wywołującej ze stosu (gcc)
ret                        # zwrot sterowania do miejsca wywołania
```

Łączenie as z C/C++

Użycie funkcji bibliotecznych C z poziomu asemblera,

Dostęp do funkcji uzyskamy kompilując program za pomocą **gcc** a nie **as**, co wymaga użycia etykiety startowej `main` zamiast `_start` – kompilator *GNU gcc* inicjuje procedury ładujące biblioteki systemu. Alternatywą jest (z etykietą `_start`) wymuszone dołączenie bibliotek przez linker:

```
ld prog.o -o prog -lc -dynamic-linker /lib/ld-linux.so.2
```

```
# int puts(const char *s) #
# wysyła na STDOUT znaki od adresu złożonego na stosie i dodaje znak NL (newline)#
                        # kompilator gcc dołącza potrzebną bibliotekę
.data                   # dyrektywa .align 32 zbędna dane bajtowe/ascii
napis: .ascii "Napis\0" # napis musi być zakończony terminatorem '\0' (koniec)
.text
.globl main             # dyrektywa eksportowania nazwy do linkera
main:                   # etykieta startowa jak dla C, kompilacja gcc
push $napis             # adres łańcucha na stos (parametr funkcji puts)
call puts               # wywołanie funkcji języka C
call exit               # albo jak w as: mov $1, %eax - mov $0, %ebx - int $0x80
```

```
# printf(format-a, arg1, arg2,...) #
.data                       # Łańcuch formatujący jest pierwszym argumentem printf
format_string:              # – określa liczbę i rodzaj parametrów – tu są 3 param.
.ascii "Hello! %s is a %s who loves the number %d\n\0"
text1: .ascii "Tekst1\0"    # pierwszy parameter %s (łańcuch znaków zakończony \0)
text2: .ascii "Tekst2\0"    # drugi parameter %s (łańcuch znaków zakończony \0)
number: .long 3             # trzeci parameter %d (liczba dziesiętna)
.text
.globl _start
_start:                     # konieczny linkera w opcji dynamicznej lub main
                            # parametery przez stos w odwróconej kolejności
pushl number                # trzeci – liczba dziesiętna (%d)
pushl $text2                # drugi – łańcuch (%s)
pushl $text1                # pierwszy – łańcuch (%s)
pushl $format_string        # łańcuch formatujący
call printf
#pushl $0                   .. zwracany kod błędu #
call exit                   # funkcja zakończenia programu
```

Funkcja zlicza ilość znaków % w łańcuchu formatującym, co pozwala określić liczbę parametrów na stosie. Dodatkowo musi wykonywać konwersję liczb do ciągów znaków odpowiadających zapisowi pozycyjnemu. Funkcja ma też wiele innych opcji formatowania (%o, %x, %e, %f, %g).

Wstawki asemblerowe w języku C (inline assembly)

Główne problemy przy łączeniu asemblera z kodem w języku wyższego poziomu to przekazanie danych i wyników oraz zapobieganie wzajemnemu niszczeniu zawartości rejestrów. W języku C dostępna jest konstrukcja składniowa, umożliwiająca wstawkę asemblerową. Ma ona postać łańcucha znaków przekazywanego do asemblera po wstępnym przetworzeniu przez kompilator C:

```
__asm__ {
        "instrukcja asemblera\n"
        "następna instr    "ostatnia instrukcja\n"
        : zmienne wyjściowe (opcjonalne)
        : wartości wejściowe (opcjonalne)
        : niszczone rejestry (opcjonalne)
        };
```

Instrukcje można też zapisać jako jeden łańcuch używając symbolu kontynuacji '\' w kolejnej linii. Symbole poprzedzone znakiem % są traktowane jako argumenty instrukcji zapisanych w kolejnych łańcuchach, albo nazwy rejestrów (%eax, %ebc,...) albo numery porządkowe zmiennych (0, 1, 2, ...):

```
/* zamiana wartości zmiennych x i y (typu int) przy użyciu rejestrów */
__asm__ {
        "movl %2, %%eax\n"          // "x we" do eax
        "movl %3, %0\n"             // "y we" do "x wy"
        "movl %%eax, %1\n"          // eax do "y wy"
        : "=r"(x),"=r"(y)           // zmienne wyjściowe (nr 0 i 1)
        : "r"(x),"r"(y)             // wartości wejściowe (nr 2 i 3)
        : "%eax"                    // rejestr niszczony (eax)
        };
```

Elementy list oddzielają przecinki. Każdy element ma określony sposób przekazywania.

- "r" - za pomocą dowolnego rejestru
- "m" - poprzez adres w pamięci
- "a", "b", "c", "d", "S", "D" - w rejestrach eax, ebx, ecx, edx, esi lub edi

Znak = bezpośrednio przed symbolem (r,m,...) oznacza, że lokalizacja dotyczy zmiennej wyjściowej. Znaki =& przed symbolem (r,m,...) oznaczają użycie innego rejestru dla zmiennej na wyjściu.

W instrukcji asemblera można bezpośrednio wskazać statyczną zmienną globalną, pisząc jej nazwę poprzedzoną znakiem $. Na przykład: __asm__("movl $xxx, %%eax":::"%eax");

Aby zablokować optymalizację kodu asemblerowego przydatne jest pisanie __volatile__ po __asm__,Bez tego **gcc** może uznać, że nasz asembler tutaj nic istotnego nie wnosi (czytaj: nie zmienia wartości zmiennej, ani nie wywołuje funkcji) i usunąć go z końcowego kodu.

Kompilacja z opcjami –s oraz –fverbose-asm pozwala wygenerować kod z kompilatora wyższego poziomu do asemblera. Kod kod generowany przez kompilator **gcc** będzie wtedy wyraźnie oddzielony komentarzami od kodu wstawki asm.

http://students.mimuw.edu.pl/SO/Projekt03-04/temat2-g6/book1.html

Przekazywanie nazw (zmiennych, funkcji) i danych pomiędzy modułami C i as

- korzystanie ze zmiennych zdefiniowanych w asemblerze z poziomu C i odwrotnie

- wywołanie własnej funkcji napisanej w C z poziomu asemblera
- wywołanie własnej funkcji argumentów zmiennoprzecinkowych
- korzystanie z funkcji napisanych w asemblerze z poziomu języka C

Funkcja `main()` w pliku z kodem w C zawiera wywoływanie funkcji asemblerowej, która wywołuje kolejno funkcje napisane w C:
– funkcja `suma`, dodająca wartość lokalną z as oraz zmienną globalną zdefiniowaną w C; wynik wypisuje funkcja biblioteczna `printf`;
– funkcja `iloraz` argumentów zmiennoprzecinkowych (wskazanych za pomocą dyrektywy `.float`) przesyłanych przez stos do FPU – wynik jest na szczycie stosu `st(0)` koprocesora (FPU)
– wypisanie wyniku z FPU (`st(0)`) za pomoca `printf` – należy go przesłać go ze stosu FPU (`st(.)`) na stos programowy po uprzedniej konwersji przez stos FPU na format `double` (standard w C).

```
#include <stdio.h>
extern void funkcja_asm();  / nazwa funkcji zewnętrznej (z innego pliku)
extern int globalna_z_asm;  / nazwa zmiennej zewnętrznej (z innego pliku)
int globalna_z_C = 777;
void moja_fun(char *arg)    / deklaracja własnej funkcji w C
{
printf("Wywolanie z C: %s", arg);
}
int suma(int a, int b)      /  deklaracja własnej funkcji w C
{
return a+b;
}
float iloraz(float a, float b)   /  deklaracja własnej funkcji w C
{
if(b==0.0) return 0.0;
return a/b;
}
int main()                       /  funkcja główna w C
{
funkcja_asm();                   / wywołanie funkcji zewnetrznej
printf("Zmienna z asemblera: %d\n", globalna_z_asm);
return 0;
}
.globl globalna_z_asm      # deklaracja nazwy zmiennej jako globalnej
[.extern globalna_z_C]     # zbędne, symbol niezdefiniowany jest uznany za extern
.data
napis: .ascii "Argument z asemblera, wynik funkcji z C = %d\n\0"
napis2:.ascii "Argument z asemblera, wynik float z C = %f\n\0"
liczba1: .float 3
liczba2: .float 4
.type globalna_z_asm, @object    # zadeklaruj zmienna z C jako obiekt
.size globalna_z_asm, 4
globalna_z_asm: .long 444
```

```
.text
.globl funkcja_asm              # deklaracja nazwy funkcji jako globalnej
.type funkcja_asm, @function    # definicja funkcji
funkcja_asm:
push %ebp
mov %esp, %ebp            # utworz ramke stosu
push $4                   # na stos liczbe 4
push globalna_z_C         # na stos wartosc zmiennej globalne zdefiniowanej w C
call suma                 # wywolaj funkcje z C
add $8, %esp              # przesun stos
push %eax                 # wynik sumowania na stos
push $napis               # adres napisu jako ciag formatujacy dla printf
call printf
add $4, %esp
mov liczba1, %eax         # zaladuj na stos zmienne float
mov %eax, 4(%esp)
mov liczba2, %eax
mov %eax, (%esp)
call iloraz               # wywolaj funkcje z C operujaca na zmiennych float
fstps -8(%ebp)            # zapisz wynik z pamieci (ze stosu FPU st(0) )
flds -8(%ebp)             # zaladuj go ponownie do stosu FPU jako double
fstpl (%esp)              # ze szczytu stosu FPU na szczyt sotsu programowego
push $napis2              # adres napisu jako ciag formatujący dla printf
call printf               # wypisz informacje
add $4, %esp
leave                     # usuń ramke stosu (mov %ebp, %esp / pop %ebp)
ret
```

Użycie profilera do optymalizacji kodu – program w języku C działający na dużych zbiorach danych.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 5000000
void fun(int *dane)
{
int i;
for(i=0; i<SIZE; ++i)
dane[i] = rand(); //wypelnij tablice losowymi danymi
for(i=0; i<SIZE; ++i)
{
//dane[i] = 5*dane[i] + dane[i]/3 + dane[i]*3; // waskie gardlo zastąpione przez:
__asm(                              / wstawka asm zamiast poprzedniej linii/
"movl (%%edi,%%ecx,4), %%eax \n\
mov %%eax, %%ebx \n\
mov %%eax, %%edx \n\
```

```
shl $3, %%eax \n\
shr $1, %%ebx \n\
sub %%edx, %%ebx \n\
add %%ebx, %%eax \n\
movl %%eax, (%%edi,%%ecx,4)"
: // no output
:"c"(i), "D"(dane)              / modyfikatory _asm – przypisanie do rejestrów
);                              / koniec wstawki
}
}
int main()
{
int *dane = (int*) malloc(SIZE * sizeof(int)); //rezerwuj pamiec
srand(time(NULL)); //zainicjuj generator pseudolosowy
fun(dane); //wywolaj badana funkcje
free(dane); //zwolnij pamiec
return 0;
}
```

Wywołanie funkcji `__asm()` pozwala na wstawienie kodu asemblerowego w zadane miejsce w programie w C. Ma ono specyficzną budowę. Wewnątrz wstawki można dostać się do zmiennej iteracyjnej pętli „`i`" za pomocą rejestru `ecx` oraz adresu początku tablicy przez rejestr `edi`. Aby to było możliwe należy dodać modyfikatory do parametru wywołania instrukcji `__asm()`, a mianowicie `:"c"(i), "D"(dane)`. Informuje to kompilator, że do rejestru `ecx` ma wstawić wartość zmiennej „`i`" a do rejestru `edi` – adres tablicy (wart. zmiennej `dane`). Wyniki pomiaru czasu wykonania programu niezoptymalizowanego (bez wstawki) uzyskujemy za pomocą aplikacji `gprof` (do kompilacji należy użyć modyfikatora `-pg`). Przykładowy program skompilowano z domyślnymi ustawieniami optymalizacji automatycznej kompilatora.

(przykładowy komunikat)

```
Each sample counts as 0.01 seconds.
% cumulative self self total
time   seconds  seconds  calls  ms/call  ms/call  name
100.00 0.22     0.22     1      220.00   220.00   fun     # przed optymalizacją
100.00 0.18     0.18     1      180.00   180.00   fun     # po optymalizacji (asm)
```

Do pomiarów można też wykorzystać instrukcję **`rdtsc`**, która zapisuje stan licznika cykli procesora (liczonych od restartu procesora – zwykle włączenia komputera) do pary rejestrów `edx:eax`.
Aby uzyskać dostęp do stanu licznika TSC (*Time Stamp Counter*) użyto makra preprocesora:

```
#define rdtsc() ({int64_t x; asm volatile("rdtsc" : "=A" (x)); x; })
```

Makro to jest wywoływane w kodzie jak funkcja i zwraca wartość typu `int64_t` – jest to rozszerzona do 64-b zmienna `int`. Aby jej użyć należy dołączyć plik nagłówkowy: `#include <inttypes.h>`. Czas działania funkcji jest różnicą stanów licznika przed wywołaniem, podzieloną przez częstotliwość procesora (jeśli w kHz – wtedy czas jest w ms).

Odwzorowanie programu, danych i stosu w pamięci [1]

Każda sekcja jest ładowana do osobnego obszaru w pamięci (początek bloku: adres 0x BFFF FFFF). Kody instrukcji (`.section .text`) są ładowane począwszy od adresu 0x08048000, kody danych (`.section .data`) bezpośrednio po nich, a następnie bufor dynamiczny (`.section .bss`). Ostatni bajt nie może być wyżej niż w lokacji 0xBFFFFFFF. W tym miejscu Linux zaczyna tworzyć swój stos, który jest rozbudowywany w kierunku adresów rosnących, aż do swej kolejnej sekcji.
Na dnie stosu (ang. *the bottom of the stack*), którego adres jest najwyższy (ang. *the top address of memory*) jest początkowo umieszczone *słowo zerowe* (wszystkie bity są zerami). Po nim następuje zakończona zerem (ang. *the null-terminated*) nazwa programu w kodzie ASCII, a po niej zmienne środowiskowe programu (ang. *program's environment variables*). Dalej są się argumenty wywołania (ang. *command-line arguments*), czyli parametry (wartości) wpisane do linii polecenia podczas wywołania programu. Na przykład, uruchamiając `as`, podajemy jako argumenty: `as`, `sourcefile.s`, `-o` i `objectfile.o`. Po nich następują używane argumenty, umieszczone na początku bloku stosu wskazanego przez wskaźnik stosu (ang. *stack pointer*) `%esp`. Kolejne operacje na stosie zmieniają ten wskaźnik – złożenie danych na stos powoduje zmniejszenie `%esp`.

UWAGA: podobne, ale nierównoważne działanie (pushl i popl są nierozdzielne wobec przerwania):

```
movl %eax, (%esp)              # „pushl %eax”
subl $4, %esp
movl (%esp), %eax              # „popl %eax”
addl $4, %esp
```

Obszar danych programu rozpoczyna się na dnie pamięci (ang. *the bottom of memory*) i jest budowany wzwyż (kolejne lokacje mają coraz wyższe adresy). Stos rozpoczyna się na szczycie pamięci (ang. *the top of memory*) i jest rozbudowywany w dół (w kierunku adresów malejących) po każdym wykonaniu instrukcji `push`. Obszar pomiędzy stosem a obszarem danych jest z zasady niedostępny z poziomu programu. Próba dostępu (użycie adresu z tego obszaru) kończy się sygnalizacją błędu, zwykle jako "segmentation fault". To samo zdarzy się podczas próby zaadresowania obszaru poniżej adresu 0x08048000. Ten najniższy dostępny adres jest nazywany *system break* (albo [*current*] *break*).

Zmienne środowiskowe (environment variables)		0x BFFF FFFF
…		
Arg 2		
Arg 1		
Nazwa programu (program name)		
Liczba argumentów (# of arguments)	%esp	
Pamięć dostępna dla programu (Unmapped memory)		
Kod i dane programu (Program Code and Data)	break	0x 0804 8000

Dostęp do obszaru zakazanego, z ograniczeniami wynikającymi ze struktury systemu operacyjnego) można uzyskać za pomocą komunikatu przekazanego funkcjom jądra (kernel) systemu operacyjnego.

System plików UNIX/Linux

/ – katalog główny (root) –

/root – pliki prywatne superużytkownika

/usr – udostępnione dane i programy wspólnego użytku (**usr**)

/dev – sterowniki urządzeń (**dev**ices)

/bin – podstawowe polecenia systemowe (**bin**ary)

/etc – pliki konfiguracyjne systemu (**et c**etera)

/tmp – pliki tymczasowe (**tmp**orary)

/home – pliki prywatne (domowe) użytkowników

Procesor poleceń / powłoka (ang. shell) oraz pliki konfiguracyjne i znak zachęty (ang. prompt)

($) **Bourne shell** (oficjalny dostarczany z systemem UNIX) – autoexec: **.profile**

(%) **C shell** (bardziej elestyczny, wolniejszy) – autoexec: **.profile** oraz **.login**

($) **Korn shell** (to co najlepsze z Bourne & C shells)

plik **.profile** zawiera zmienne środowiskowe i polecenia, np.

HOME=/usr/john nazwa katalogu użytkownika

PATH=/bin:usr/bin:$HOME/bin ścieżki wyszukiwania programów

MAIL=/usr/spool/mail/'basename $HOME' ścieżka poczty

TERM tv950 typ terminala

export HOME PATH MAIL TERM polecenie wyeksportowania do systemu

w **C shell** zamiast export jest polecenie **setenv** dla każdej zmiennej np. setenv HOME ..

Literatura

[1] Programming from the Ground Up . (www….)

[2] J. Biernat, Architektura komputerów, Oficyna Wyd. PWr, Wrocław, 2005 (wyd. IV)

[3] L. Madeja, Ćwiczenia z systemu Linux, MIKOM, 1999 (rozdz. 4).

[4] D. Elsner, J. Fenlason & friends, Using as. The GNU assembler, Free Software Foundation Inc., 1994, (www….)

[5] http://www.linux-foundation.org/spec/refspecs/LSB_3.1.0/LSB-Core-IA32/LSB-Core-IA32.pdf

[6] http://www.caldera.com/developers/devspecs/abi386-4.pdf

[7] http://students.mimuw.edu.pl/SO/Projekt03-04/temat2-g6/book1.html