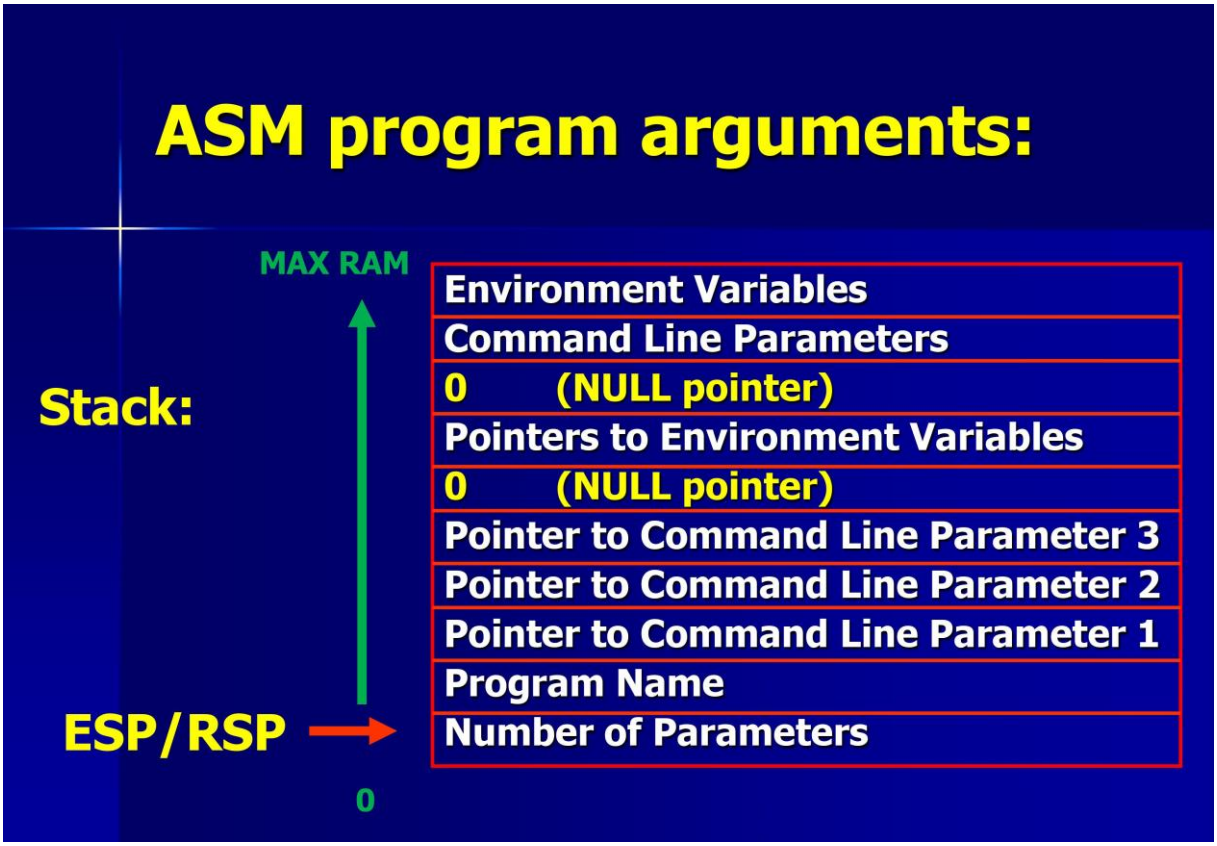
lab_9 - Instrukcja do ćwiczenia

Teoria:

Argumenty programu:

Przed uruchomieniem programu system umieszcza na stosie informacje o liczbie argumentów, ich postaci oraz o zmiennych środowiskowych. Argumenty i zmienne środowiskowe są przekazane w postaci sekwencji wskaźników (adresów) do łańcuchów znaków zakończonych bajtem o wartości **0** (jak łańcuchy znaków w **C**). W systemie 32-bitowym zarówno liczba argumentów jak i wskaźniki mają rozmiar 4 bajtów. W systemie 64-bitowym liczba argumentów i same wskaźniki są danymi 64-bitowymi. Upraszcza to znacznie dostęp do tych danych. Na samym wierzchołku stosu znajduje się liczba argumentów, zaś kolejne dane znajdują się niżej (pod wyższymi adresami). Liczba argumentów jest znana z góry, więc dostęp do kolejnych argumentów może być zrealizowany z użyciem pętli typu **for**. Liczba zmiennych środowiskowych nie jest znana z góry i może być dowolna – w szczególnym przypadku może ich nie być wcale. Wymusza to zastosowanie pętli typu **while**. Struktura danych została pokazana poniżej.



Praktyka (lab_9a.c+lab_9a_asm.s, lab_9b.c+lab_9b_asm.s oraz lab_9c.s):

Działania:

1. Testujemy działanie funkcji **factc** (język **C**) oraz **facta** (asembler) liczących silnię dla argumentów będących liczbami nieujemnymi. Program główny zawiera wywołania obu funkcji, co pozwala porównać uzyskiwane rezultaty. Obliczenia prowadzone są na danych 32-bitowych.
2. CL (Compile&Link) – polecenie: **gcc –o lab_9a lab_9a.c lab_9a_asm.s**
3. R (Run) – polecenie: **./lab_9a**
4. Pojawiają się wyniki dla argumentów od **1** do **6** – obie funkcje dają się identyczne wyniki.
5. Zmieniamy górną granicę licznika pętli w funkcji **main** na **30**:

```
for( i = 1; i <= 30; i++ )
```

6. CLR
7. Dla argumentu równego **13** ostatnią cyfrą wartości silni jest **4**, co świadczy o przepełnieniu rejestru w trakcie mnożenia (wcześniejsze wartości mają **0**).

```
FactC(10) = 3628800 FactA(10) = 3628800
FactC(11) = 39916800 FactA(11) = 39916800
FactC(12) = 479001600 FactA(12) = 479001600
FactC(13) = 1932053504 FactA(13) = 1932053504
FactC(14) = 1278945280 FactA(14) = 1278945280
FactC(15) = 2004310016 FactA(15) = 2004310016
FactC(16) = 2004189184 FactA(16) = 2004189184
```

8. Modyfikujemy kod tak, aby obliczenia były prowadzone na danych 64-bitowych.
9. Kod w **C** należy zmienić w pięciu miejscach:

```
long factc( unsigned int k )
long result = 1;
long facta( unsigned int k );
printf( "FactC(%d) = %ld\n", i, factc(i) );
printf( "FactA(%d) = %ld\n", i, facta(i) );
```

10. Modyfikacja kodu w pliku **lab_9a_asm.s** polega na użyciu wszędzie rejestrów 64-bitowych:

```
mov $1, %rax        # result or multiplicand
cmp $1, %rdi        # k <= 1 ?
mul %rdi            # result * k
dec %rdi            # k--
```

11. CLR
12. Okazuje się, że tym razem przepełnienie pojawia się dla argumentu równego **21** (wartość ujemna).

```
FactC(15) = 1307674368000 FactA(15) = 1307674368000
FactC(16) = 20922789888000 FactA(16) = 20922789888000
FactC(17) = 355687428096000 FactA(17) = 355687428096000
FactC(18) = 6402373705728000 FactA(18) = 6402373705728000
FactC(19) = 121645100408832000 FactA(19) = 121645100408832000
FactC(20) = 2432902008176640000 FactA(20) = 2432902008176640000
FactC(21) = -4249290049419214848 FactA(21) = -4249290049419214848
FactC(22) = -1250660718674968576 FactA(22) = -1250660718674968576
FactC(23) = 8128291617894825984 FactA(23) = 8128291617894825984
```

13. Przechodzimy do programu **lab_9b** – celem jest przetestowanie funkcji wyznaczających wartości kolejnych wyrazów ciągu Fibonacciego. Program składa się z dwóch modułów: w **C** (funkcje **fibc** i **main**) i w asemblerze (funkcja **fiba**).
14. CL (Compile&Link) – polecenie: **gcc –o lab_9b lab_9b.c lab_9b_asm.s**
15. R (Run) – polecenie: **./lab_9b**
16. Pojawiają się wyniki dla argumentów od **0** do **10** – obie funkcje dają się identyczne wyniki.
17. Zmieniamy górną granicę licznika pętli w funkcji **main** na **50**:

```
for( i = 1; i <= 50; i++ )
```

18. CLR
19. Dla argumentu równego **47** wartości są ujemne, co świadczy o przepełnieniu rejestru w trakcie dodawania.

```
FibC(40) = 102334155 FibA(40) = 102334155
FibC(41) = 165580141 FibA(41) = 165580141
FibC(42) = 267914296 FibA(42) = 267914296
FibC(43) = 433494437 FibA(43) = 433494437
FibC(44) = 701408733 FibA(44) = 701408733
FibC(45) = 1134903170 FibA(45) = 1134903170
FibC(46) = 1836311903 FibA(46) = 1836311903
FibC(47) = -1323752223 FibA(47) = -1323752223
FibC(48) = 512559680 FibA(48) = 512559680
FibC(49) = -811192543 FibA(49) = -811192543
FibC(50) = -298632863 FibA(50) = -298632863
```

20. Modyfikujemy kod tak, aby obliczenia były prowadzone na danych 64-bitowych.
21. Kod w **C** należy zmienić w siedmiu miejscach:

```
long fibc( unsigned int k )
long fold;
long fnew;
long sum;
long fiba( unsigned int k );
printf( "FibC(%d) = %ld FibA(%d) = %ld\n", i, fibc(i), i, fiba(i) );
```

22. Modyfikacja kodu w pliku **lab_9a_asm.s** polega na użyciu wszędzie rejestrów 64-bitowych:

```
        mov $0, %rbx        # old
        mov $1, %rcx        # new
        cmp %rbx, %rdi      # k == 0 ?
        cmp %rcx, %rdi      # k == 1 ?
        mov %rbx, %rax      # sum = old
        add %rcx, %rax      # sum += new
        mov %rcx, %rbx      # old = new
        mov %rax, %rcx      # new = sum
        dec %rdi            # k--
        cmp $1, %rdi        # k > 1 ?
f_0:    mov %rbx, %rax      # return 0
f_1:    mov %rcx, %rax      # return 1
```

23. CLR
24. Okazuje się, że tym razem przepełnienie się nie pojawia (dla przetestowanych argumentów).
25. Zmieniamy górną granicę licznika pętli w funkcji **main** na **100**:

```
for( i = 1; i <= 100; i++ )
```

26. CLR
27. Przepełnienie pojawia się dla argumentu równego **93**:

```
FibC(90) = 2880067194370816120 FibA(90) = 2880067194370816120
FibC(91) = 4660046610375530309 FibA(91) = 4660046610375530309
FibC(92) = 7540113804746346429 FibA(92) = 7540113804746346429
FibC(93) = -6246583658587674878 FibA(93) = -6246583658587674878
FibC(94) = 1293530146158671551 FibA(94) = 1293530146158671551
FibC(95) = -4953053512429003327 FibA(95) = -4953053512429003327
FibC(96) = -3659523366270331776 FibA(96) = -3659523366270331776
FibC(97) = -8612576878699335103 FibA(97) = -8612576878699335103
FibC(98) = 6174643828739884737 FibA(98) = 6174643828739884737
FibC(99) = -2437933049959450366 FibA(99) = -2437933049959450366
FibC(100) = 3736710778780434371 FibA(100) = 3736710778780434371
```

28. Przechodzimy do programu **lab_9c.s**.
29. Program zaczyna się od etykiety _start, co uniemożliwia użycie **gcc** w procesie kompilacji i linkowania. Komplikuje to budowę kodu wykonywalnego – konieczne jest użycie następujących poleceń:

```
as -o lab_9c.o lab_9c.s
```

```
ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 -lc -o lab_9c lab_9c.o
```

30. Uruchamiamy program z użyciem dodatkowych argumentów:

```
./lab_9c  Ala  ma  kota
```

31. Program działa poprawnie:

```
buba@buba-pc:~/asm/l9$ ./lab_9c  Ala  ma  kota
Argc = 4
Argv[0] = ./lab_9c
```

```
Argv[1] = Ala
Argv[2] = ma
Argv[3] = kota
----------------------------
Env[0] = CLUTTER_IM_MODULE=xim
Env[1] = LIBVA_DRIVER_NAME=iHD
…
Env[69] = LC_TIME=pl_PL.UTF-8
Env[70] = _=./lab_9c
Env[71] = OLDPWD=/home/buba/asm
```

32. Zamieniamy **_start** na **main** (symbol **_start** występuje w kodzie dwukrotnie!).
33. Teraz możemy użyć **gcc** do budowy programu wykonywalnego:

```
gcc  -no-pie  -o lab_9c  lab_9c.s
```

34. Uruchamiamy program z użyciem dodatkowych argumentów:

```
./lab_9c  Ala  ma  kota
```

35. Program nie działa – wyświetlana jest podejrzana wartość **Argc** i następuje naruszenie ochrony pamięci. Przyczyną jest zmiana nazwy punktu startowego – tym razem kod jest uruchamiany instrukcją **call main**, co oznacza, że na wierzchołku stosu znajduje się adres powrotu. Dane umieszczone przez system operacyjny dalej znajdują się na stosie, ale gdzieś niżej i nie wiadomo dokładnie gdzie. Funkcja **main** ma jednak dostęp do tych samych danych, gdyż są one jej argumentami i jako takie znajdują się w rejestrach: **argc** w **%rdi**, **argv** w **%rsi** oraz **env** w **%rdx**.
36. Modyfikujemy odpowiednie fragmenty kodu źródłowego – na czerwono znaki komentarza, które należy wstawić przed istniejącymi instrukcjami, na niebiesko nowe instrukcje, na czarno istniejące instrukcje:

Początek:

```
main:
#     mov (%rsp), %rax     # argc is here
      mov %rdi, %rax
      mov %rsi, argv
      mov %rdx, env
      mov %rax, argc       # store value of argc
      mov %rax, argc_tmp
```

Okolice środka:

```
#     mov %rsp, %rbx       # use rbx as a pointer
#     add $8, %rbx         # argv[] is here
#     mov %rbx, argv       # store address of argv[]
      mov argv, %rbx
```

Okolice końca:

```
#     add $8, %rbx         # env[] is here - skip zero
```

```
#     mov %rbx, env         # store address of env[]
      mov env, %rbx
next_env:
```

37. Ponownie używamy **gcc** do budowy programu wykonywalnego:

```
gcc  -no-pie  -o lab_9c   lab_9c.s
```

38. Uruchamiamy program z użyciem dodatkowych argumentów:

```
./lab_9c  Ola  ma  psa
```

39. Hurra! Program znowu działa i wyświetla dane:

```
buba@buba-pc:~/asm/l9$ ./lab_9c  Ola  ma  psa
Argc = 4
Argv[0] = ./lab_9c
Argv[1] = Ola
Argv[2] = ma
Argv[3] = psa
----------------------------
Env[0] = CLUTTER_IM_MODULE=xim
Env[1] = LIBVA_DRIVER_NAME=iHD
…
Env[69] = LC_TIME=pl_PL.UTF-8
Env[70] = _=./lab_9c
Env[71] = OLDPWD=/home/buba/asm
```

40. Ciesz się z tego, że Twój komputer wcale nie jest taki wolny – musisz tylko wybierać zadania stosownie do jego możliwości!