lab_7 - Instrukcja do ćwiczenia

Teoria (params_64.pdf):

Przekazywanie parametrów:

Zadanie polega na użyciu w kodzie asemblerowym funkcji znajdujących się w bibliotece standardowej LIBC. Wymaga to zrozumienia zasad dotyczących przekazywania parametrów do wywoływanych funkcji oraz zwracania przez te funkcje ewentualnych rezultatów. Szczegółowe informacje na ten temat można znaleźć w dokumencie „System V Application Binary Interface - AMD64 Architecture Processor Supplement" (**abi.pdf**).

Aby uprościć proces tworzenia kodu wykonywalnego, w programie użyto jako punktu startowego etykiety **main** (a nie **_start**) – pozwala to na wykorzystanie do kompilacji i linkowania programu **gcc** oraz zwalnia nas od konieczności podawania dokładnych nazw bibliotek i ich lokalizacji. Dodatkową zaletą takiego podejścia jest też możliwość prostego i łatwego dostępu do danych przekazywanych standardowo do funkcji **main** (**argc**, **argv** i **env**).

Do przekazywania parametrów będących liczbami całkowitymi (dane liczbowe i adresy) wykorzystywane są następujące rejestry procesora:

%rdi – parametr nr 1
%rsi – parametr nr 2
%rdx – parametr nr 3
%rcx – parametr nr 4
%r8 – parametr nr 5
%r9 – parametr nr 6

Jeśli funkcja wymaga większej liczby parametrów, to muszą one być przekazane poprzez stos.

Funkcja nie może zmienić wartości rejestrów **%rbp**, **%rbx**, **%r12**, **%r13**, **%r14**, **%15** oraz **%rsp** – co oznacza, że jeśli chcemy ich użyć, to konieczne jest zapisanie ich na stosie (np. na początku kodu) a później odtworzenie ich oryginalnej zawartości (np. na końcu kodu). Rejestry wykorzystywane do przekazywania parametrów mogą być modyfikowane (ich wyjściowa zawartość może być różna od wejściowej). Funkcje o zmiennej liczbie parametrów (np. **printf**, **scanf**, itp.) wymagają dodatkowego parametru: rejestr **%al** musi zawierać liczbę wykorzystywanych rejestrów wektorowych (**xmm**, **ymm** – są konieczne do przekazywania liczb zmiennoprzecinkowych (**float**, **double**) jako parametrów) – nawet jeśli nie używamy żadnych rejestrów wektorowych, to musimy do rejestru **%al** wstawić liczbę **0**.

Zwracany rezultat musi znaleźć się w rejestrze **%rax**.

Przykład:

język C – `printf(„Value=%d\n", value);`

Aby użyć funkcji **printf** w języku asemblera konieczne jest zdefiniowanie odpowiednich danych:

```
value:    .long 6
fmt:      .asciz „Value=%d\n”
```

a następnie przekazanie w rejestrach parametrów dla funkcji **printf** i jej wywołanie:

```
mov value, %esi # printf(char *fmt,long num) – 2nd argument to %rsi/%esi
mov $fmt, %rdi  # printf(char *fmt,long num) – 1st argument to %rdi
xor %rax, %rax  # printf - number of vector regs to %al
call printf
```

Funkcja **printf** zwraca liczbę danych wyświetlonych zgodnie z zawartością łańcucha formatującego (w powyższym przykładzie będzie to **1**), ale zwykle można ten rezultat pominąć.

Parametry funkcji main:

Zmiana nazwy punktu startowego z **_start** na **main** i wykorzystanie programu **gcc** w procesie kompilacji i linkowania skutkuje tym, że instrukcje programu stanowią faktycznie zawartość funkcji main (głównej funkcji programu). Funkcja ta jest wywoływane instrukcją **CALL**, co pozwala na zakończenie działania poprzez użycie instrukcji **RET**. Bardziej przydatną właściwością funkcji **main** jest możliwość prostego dostępu do danych przekazywanych z zewnątrz do programu. Deklaracja funkcji **main** w języku **C** może mieć następującą postać:

```
int/void main( int argc, char *argv[], char *env[] );
int/void main( int argc, char **argv, char **env );
```

Ponieważ funkcja **main** jest taką samą funkcją jak np. **printf** (w sensie przekazywania parametrów), to parametry funkcji **main** można znaleźć w odpowiednich rejestrach:

- **argc** – w **%rdi** (bo to pierwszy argument)
- **argv** – w **%rsi** (bo to drugi argument)
- **env** – w **%rdx** (bo to trzeci argument)

Celowe jest przeniesienie tych parametrów do zmiennych, bo te same rejestry będą używane przy wywołaniach innych funkcji wewnątrz funkcji **main** – najlepiej jest to zrobić na samym początku programu.

Liczba argumentów programu (**argc**) jest większa lub równa **1** (**1**, jeśli nie podano żadnych argumentów – wtedy **argv[0**] jest nazwą programu (polecenia użytego do uruchomienia programu) – może być wyświetlona tak, jak każda inna liczba całkowita. Argumenty programu (**argv**) są wskaźnikami do łańcuchów znaków (zgodnymi z językiem **C**), więc do ich

wyświetlenia konieczne jest użycie formatu **%s**, a samo wyświetlanie można zrealizować w pętli typu **for** (liczba elementów jest znana z góry). Przykładowy kod ma następującą postać:

```
fmt_str:    .string "%s\n"

      mov argv, %rbp          # %rbp = argv;
next_argv:
      mov (%rbp), %rsi        # printf( fmt, str ) – 2nd argument to %rsi;
      mov $fmt_str, %rdi      # printf( fmt, str ) – 1st argument to %rdi;
      xor %rax, %rax          # printf - number of vector regs to %al
      call printf
      add $8, %rbp            # next argv
      decl argc               # argc;
      jnz next_argv
```

Rejestr **%rbp** wykorzystany jest do adresowania kolejnych elementów tablicy **argv** (zmienne **argv** i **argc** powinny zostać zainicjalizowane na początku programu na podstawie parametrów przesłanych do funkcji **main**). Tablica **argv** jest tablicą wskaźników, więc do dostępu do samego wskaźnika, konieczne jest użycie adresowania pośredniego: `mov (%rbp), %rsi`. Po tej operacji **%rsi** zawiera wskaźnik do łańcuch znaków (zgodnego z **C**), będącego kolejnym argumentem programu (**argv[0]**, **argv[1]**, ..., itd.). Licznikiem pętli jest zmienna **argc**, co oznacza, że jej oryginalna wartość jest niszczona – gdyby była ona jeszcze potrzebna, konieczne jest użycie jakiegoś rejestru jako licznika pętli, albo innej zmiennej, będącej kopią **argc**. Każdy element tablicy **argv** jest wskaźnikiem (adresem) i zajmuje w pamięci **8** bajtów, więc o tyle modyfikowana jest zawartość rejestru **%rbp** przy przejściu do kolejnego elementu tablicy.

Ponieważ liczba elementów tablicy **env** nie jest znana z góry, konieczne jest zastosowanie pętli typu **while**. Warunkiem stopu jest znalezienie wskaźnika o wartości **0** (**NULL**). Przykładowy kod ma następująca postać:

```
      mov env, %rbp           # %rbp = env;
next_env:
      cmp $0,(%rbp)           # while( env[i] != NULL )
      jz no_more_env
      mov (%rbp), %rsi        # printf( fmt, str ) – 2nd argument to %rsi;
      mov $fmt_str, %rdi      # printf( fmt, str ) - 1st argument to %rdi;
      xor %rax, %rax          # printf - number of vector registers to %al
      call printf
      add $8, %rbp            # next env
      jmp next_env
no_more_env:
```

Konwersja łańcuchów znaków na liczby:

Ponieważ argumenty programu są dostępne w formie łańcuchów znaków, tam gdzie niezbędne są ich wartości liczbowe, konieczne jest dokonanie odpowiedniej konwersji. Można do tego celu wykorzystać np. funkcję atoi – jej deklaracja jest następująca:

```
int atoi( const char *string );
```

Funkcja zwraca wartość liczby zapisanej w postaci ciągu znaków lub 0, jeśli znaki nie dają się zinterpretować jako liczba. Przykładowe zastosowanie może wyglądać następująco:

```
num_str:  .string   "15"
num_val:  .long     0

mov $num_str, %rdi  # 1st and only argument
call atoi           # call function
mov %eax, num_val   # store converted value in variable
```

Praktyka (lab_7a.s oraz lab_7b.c i lab_7b_asm.s):

Działania:

1. Testujemy użycie funkcji **printf** do wyświetlenia wartości liczby **val_1** i funkcji **exit** do zakończenia działania programu – inne instrukcje są zakomentowane.
2. CL (Compile&Link) – polecenie: **gcc –no-pie –o lab_7a lab_7a.s**
3. R (Run) – polecenie: **./lab_7a**
4. Pojawia się napis: „Value = 6” i program kończy swoje działanie, co świadczy o prawidłowym przebiegu przekazywania parametrów i wywoływania funkcji z biblioteki C.
5. Komentujemy linie dotyczące wywołania funkcji **exit**:

```
#  xor %rdi, %rdi          # exit( code ) - first argument to %rdi
#  call exit
```

6. CLR
7. Efekt jest taki sam jak wcześniej – funkcja **main** kończy swoje działanie poprzez wykonanie instrukcji **RET**.
8. Usuwamy komentarze w kolejnych liniach programu – celem jest przetestowanie wyświetlania dwóch napisów przy pomocy funkcji **printf**, stanowiących otoczenie wywołania funkcji scanf (tzw. prompt oraz znak końca linii) - aby poprawić wygląd ekranu po użyciu funkcji **scanf**, wyświetlamy znak **\n** (w ten sposób wymusimy wyświetlanie kolejnych informacji od początku kolejnej linii ekranu niezależnie od tego jak przebiegało wprowadzanie danych).:

```
#  mov $fmt_prompt, %rdi   # printf( fmt ) - 1st argument to %rdi;
#  xor %rax, %rax          # printf - number of vector registers to %al
#  call printf

#  mov $fmt_lf, %rdi # printf( fmt ) - 1st argument to %rdi;
#  xor %rax, %rax          # printf - number of vector registers to %al
#  call printf
```

9. CLR
10. Tym razem na ekranie powinie pojawić się następujący rezultat:

```
buba@buba-pc:~/asm/l7$ ./lab_7a
Value = 6
```

11. Usuwamy kolejne komentarze – tym razem chodzi o przetestowanie działania funkcji **scanf**, której chcemy użyć do wprowadzania danych przez użytkownika. Funkcja **scanf** ma wczytać jedną liczbę, więc przekazujemy do niej dwa parametry: adres łańcucha formatującego (podobnie jak dla funkcji **printf**) oraz adres zmiennej, w której zostanie umieszczona wczytana dana: **$val_1**. Dodatkowym parametrem zawartym w rejestrze **%al** jest liczba wykorzystanych rejestrów wektorowych (**scanf** jest funkcją o zmiennej liczbie argumentów) – testy wskazują, że parametr ten może być pominięty, bo tak naprawdę wszystkie argumenty funkcji **scanf** są adresami. Funkcja **scanf** zwraca liczbę danych jakie udało się wczytać – zapamiętujemy zwróconą liczbę w zmiennej **ok_num**. Poprawne wprowadzenie liczby będzie skutkować jej wyświetleniem. Jeśli dana zostanie wprowadzona nieprawidłowo (nie podano żadnej liczby), to nastąpi przejście do etykiety no_more_numbers.

```
#     mov $fmt_scanf, %rdi
#     mov $val_1, %rsi
#     mov $0, %al
#     call scanf
#     mov %eax, ok_num


#     cmp $1, ok_num
#     jnz no_more_numbers
```

12. CLR
13. Program nie działa prawidłowo – po dojściu do wywołania funkcji **scanf** następuje naruszenie ochrony pamięci. Przyczyną jest wrażliwość (wręcz nadwrażliwość bo funkcja **printf** w identycznych warunkach działa poprawnie) funkcji **scanf** na kwestię wyrównania stosu do wielokrotności 16 bajtów (po wywołaniu funkcji **main** na dotychczas wyrównanym stosie znajduje się tylko adres powrotu (**8** bajtów)). Wyrównujemy stos poprzez umieszczenie na nim dodatkowej danej o wielkości **8** bajtów – zawartości rejestru **%rbp**.
14. Wyszukujemy w kodzie (na początku i na końcu) poniższe linie i usuwamy w nich znaki komentarza:

```
main:
#     push %rbp


#     pop %rbp
```

15. CLR
16. Teraz program działa prawidłowo – możemy wprowadzić dowolną liczbę i następnie ją wyświetlić. Aby pominąć etap wyświetlania wartości liczby należy wprowadzić znaki, które nie dadzą się zinterpretować jako liczba – np. literę/litery.
17. Teraz zajmiemy się wykorzystaniem danych przekazanych do funkcji **main** z zewnątrz. Ponieważ argumenty funkcji **main** znajdują się w rejestrach, które będą też używane przy wywołaniach innych funkcji (np. **printf**, **scanf**, itp.), należy przenieść je z rejestrów

do zmiennych na początku funkcji **main** (zanim zostaną użyte do innych celów). W tym celu musimy usunąć komentarze w następujących liniach programu:

```
#      mov %edi, argc
#      mov %edi, argc_tmp
#      mov %rsi, argv
#      mov %rdx, env
```

Aby wyświetlić zawartość zmiennej **argc** niezbędne jest odkomentowanie następujących instrukcji:

```
no_more_numbers:
#      mov argc, %rsi          # printf(fmt,num) – 2nd argument to %rsi;
#      mov $fmt_argc, %rdi     # printf(fmt,num) - 1st argument to %rdi;
#      xor %rax, %rax          # printf - number of vector regs to %al
#      call printf
```

18. CLR
19. W normalnych warunkach wartość **argc** jest równa **1** – aby sprawdzić czy wszystko działa prawidłowo, należy kilkukrotnie uruchomić program, podając w linii poleceń dodatkowe argumenty - np. w poniższy sposób:

    `./lab_7a Ala ma kota`

20. Dla powyższego polecenia powinien pojawić się napis: „**Argc=4**” - jeśli program reaguje na liczbę podawanych argumentów, to możemy przejść do ich wyświetlenia. Usuwamy znaki komentarza w następujących liniach:

```
#      mov argv, %rbp          # %rbp = argv;

next_argv:

#      mov (%rbp), %rdx    # printf(fmt,num,str)- 3rd argument to %rdx;
#      mov argc, %esi
#      sub argc_tmp, %esi  # printf(fmt,num,str)- 2nd argument to %rsi;
#      mov $fmt_argv, %rdi # printf(fmt,num,str)- 1st argument to %rdi;
#      xor %rax, %rax      # printf - number of vector registers to %al
#      call printf
#      add $8, %rbp        # next argv
#      decl argc_tmp       # argc_tmp--;
#      jnz next_argv
```

Jako licznik pętli wykorzystujemy zmienną **argc_tmp** (kopia **argc**) – kolejne wartości licznika dla wcześniejszego przykładu to: **4**, **3**, **2**, **1**. Aby wyświetlić numery argumentów w sposób właściwy (**0**, **1**, **2**, **3**) wykorzystujemy wyrażenie **argc – argc_tmp** (**4-4-0**, **4-3=1**, **4-2=2**, **4-1=3**). Tablica **argv** zawiera wskaźniki/adresy – każdy o rozmiarze **8** bajtów, więc o taką wartość jest modyfikowany rejestr **%rbp** (użyty jako wskaźnik do kolejnych elementów) przy przejściu do kolejnego elementu tablicy **argv**.

21. CLR
22. Sprawdzamy działanie programu uruchamiając go kilka razy jednocześnie zmieniając podając postać argumentów i ich liczbę.
23. Przykładowy efekt uzyskany w trakcie testowania wygląda następująco:

```
buba@buba-pc:~/asm/l7$ ./lab_7a  Ala  ma  kota
Number: 10
Value = 10
Argc=4
Argv[0]=./lab_7a
Argv[1]=Ala
Argv[2]=ma
Argv[3]=kota
buba@buba-pc:~/asm/l7$
```

24. Ostatnim etapem jest wyświetlenie wartości zmiennych środowiskowych (zawartości tablicy **env**). Dokonujemy tego przez usunięcie znaków komentarza w następujących liniach:

```
#       mov env, %rbp     # %rbp = env;

next_env:

#       cmp $0,(%rbp)     # while( env[i] != NULL )
#       jz no_more_env
#       mov (%rbp), %rdx  # printf(fmt,num,str) - 3rd argument to %rdx;
#       mov argc_tmp,%esi # printf(fmt,num,str) - 2nd argument to %rsi;
#       mov $fmt_env,%rdi # printf(fmt,num,str) - 1st argument to %rdi;
#       xor %rax, %rax    # printf - number of vector registers to %al
#       call printf

#       add $8, %rbp      # next env
#       incl argc_tmp     # argc_tmp++;
#       jmp next_env

no_more_env:
```

Ponieważ liczba elementów tablicy **env** nie jest znana (w szczególności może być równa **0**), to konieczne jest zastosowanie pętli **while** (warunek sprawdzany na początku) – warunkiem stopu jest element o wartości **0** (wskaźnik **NULL**). Do numerowania elementów wykorzystano zmienną **argc_tmp** – wcześniejsze działania spowodowały, że ma ona wartość **0** i może być użyta wprost. Wartość tej zmiennej jest zwiększana o **1** przy każdym obiegu pętli, natomiast rejestr **%rbp** będący wskaźnikiem do kolejnych elementów tablicy **env**, jest zwiększany o **8** bo każdy element tablicy **env** zajmuje **8** bajtów.

25. CLR
26. Wszystkie dane przekazane do programu z zewnątrz są już wyświetlane prawidłowo.
27. Przechodzimy do programu **lab_7b** – celem tej części ćwiczenia jest przetestowanie mechanizmu przekazywanie parametrów z programu głównego (napisanego w języku C) do funkcji (napisanej w asemblerze) i zwracania przez funkcję rezultatu.
28. Program **lab_7b** składa się z dwóch modułów: **lab_7b.c** napisanego w języku C oraz **lab_7b_asm.s** napisanego w asemblerze. Moduł w asemblerze zawiera kod funkcji **sum3a** (zwracającej sumę trzech argumentów), zaś w module napisanym w języku C

znajduje się kod funkcji **sum3c** (odpowiednik funkcji sum3a) oraz kod funkcji **main**, w którym umieszczono wywołania obu funkcji.

29. CL (Compile&Link) – polecenie: **gcc –o lab_7b lab_7b.c lab_7b_asm.s**
30. R (Run) – polecenie: **./lab_7b**
31. Efekt działania programu powinien wyglądać następująco:

```
buba@buba-pc:~/asm/l7$ ./lab_7b
Sum3c(5, 2, 1) = 8
Sum3a(5, 2, 1) = 8
buba@buba-pc:~/asm/l7$
```

32. Obie funkcje działają w identyczny sposób.
33. Zmieniamy w kodzie modułu **lab_7b.c** wartość zmiennej **a** z **5** na **-5**.
34. CLR
35. Tym razem pojawiają się różnice w uzyskanych rezultatach:

```
buba@buba-pc:~/asm/l7$ ./lab_7b
Sum3c(-5, 2, 1) = -2
Sum3a(-5, 2, 1) = 4294967294
buba@buba-pc:~/asm/l7$
```

36. Przyczyną jest wykorzystanie w kodzie funkcji **sum3a** rejestrów 32-bitowych – wartość zwrócona w ten sposób interpretowana jest jako duża liczba dodatnia, a nie mała liczba ujemna.
37. Zmieniamy nazwy użytych rejestrów na ich wersje 64-bitowe:

```
mov %rdi, %rax          # 1st parameter to %rax
add %rsi, %rax          # add 2nd parameter
add %rdx, %rax          # add 3rd parameter
```

38. CLR
39. Obie funkcje zwracają ten sam rezultat:

```
buba@buba-pc:~/asm/l7$ ./lab_7b
Sum3c(-5, 2, 1) = -2
Sum3a(-5, 2, 1) = -2
buba@buba-pc:~/asm/l7$
```

40. “You're the best, better than all the rest”