Marta Głowacka, 234999 Wrocław, 11.03.2018r.

WT-N-10

LABOLATORIUM ARCHITEKTURY KOMPUTERÓW

0. Środowisko programistyczne laboratorium Architektury komputerów

Podstawy uruchomiania programu w assemblerze na platformie Linux/x64

1. Treść ćwiczenia

Zakres i program ćwiczenia:

- Podstawowe komendy w terminalu Linuxa/x64. Użycie poleceń systemu Linux w terminalu (cd, ls, rm, cat, mv…)
- Zapoznanie się z pojęciem rejestru, mnemonika, dyrektywy – wykorzystanie praktyczne
- Poznanie składni AT&T
- Utworzenie, uruchomienie i zrozumienie pierwszego programu w assemblerze na platformie Linux/x64
- Poznanie edytora tekstu Nano
- Uruchomienie w środowisku 64-bitowym programu poprzez wcześniejsze kompilowanie poleceniem **as** oraz linkowanie poleceniem **ld**
- Utworzenie pliku sterującego makefile i użycie polecenia **make**

Zrealizowane zadania:

- Utworzenie pliku sterującego makefile
- Pierwszy program helloWorld – kompilacja i omówienie kodu źródłowego
- Zamiana małych liter na wielkie

2. Przebieg ćwiczenia

Uruchomienie programu „Hello World!" napisanego w assemblerze na platformie Linux/x64

I. Sprawne operowanie komendami w terminalu linuxowym pozwala m.in. na stworzenie folderu oraz pliku. Wyróżnione poniżej komendy będą potrzebne do sprawnego poruszania się w katalogach, utworzenia pliku makefile oraz .s.

Komenda	Znaczenie
cd	Zmiana katalogu
touch [plik]	Tworzenie pliku
cat [plik]	Wyświetlenie zawartości pliku tekstowego
man [komenda]	Wywołanie manuala dla komendy
mov a b	Przenoszenie z miejsca a do b
rm [plik]	Usuwanie pliku
cp a b	Kopiuj zawartość pliku a do b
pwd	Wypisanie nazwy katalogu, w którym aktualnie jesteśmy
exit	Zamknięcie terminala, wylogowanie

Tabela 1: Komendy linuxowe

II. Korzystając z wypunktowanych poleceń, stworzono plik źródłowy helloWorld.s oraz plik sterujący **makefike**, aby ułatwić proces wielokrotnego kompilowania i linkowania programu. Ów proces wygląda następująco:

```
as –o helloWorld.o helloWorld.s
ld –o helloWorld helloWorld.o
```

W pierwszej linii kompilacja pliku źródłowego i zapisanie wyniku w pliku z rozszerzeniem *.o (tutaj o nazwie helloWorld.o). W drugiej, konsolidowanie (linkowanie) i zapisanie wyniku w pliku wykonywalnym (tutaj o nazwie helloWorld).

Uruchomienie programu z katalogu, w którym aktualnie jesteśmy, odbywa się poprzez wywołanie komendy:

```
./helloWorld
```

Ten program nie przyjmuje żadnych parametrów, więc nie piszemy nic więcej po powyższej komendzie.

W książce *Assembly Language*, Richarda Bluma podano również inny sposób kompilacji i linkowania (kompilator gcc):

```
gcc -o cpuid cpuid.s
./cpuid
```

Wiedząc już, jak wygląda procedura kompilacji i konsolidacji, możemy przystąpić do omówienia pliku sterującego makefile:

```
#reguła linkowania
helloWorld:    helloWorld.o
               ld –o helloWorld helloWorld.o
#reguła kompilowania
helloWorld.o:  helloWorld.s
               as –o helloWorld.o helloWorld.s
```

Pierwsza fraza wskazuje na to, co chcemy uzyskać (helloWorld.o), po dwukropku wskazujemy, na podstawie czego, a poniżej za pomocą jakiej formuły.

Tak przygotowany plik **makefile** możemy podać jako argument komendzie **make**. Wygląda to teraz tak:

```
make helloworld
```

W ten sposób przyspieszyliśmy cały proces uruchamiania.

III. Utworzenie pliku źródłowego .s

Hello World!

Korzystając z poleceń linuxowych **touch** oraz programu **Nano**, utworzyliśmy plik helloWorld.s. Taki kod będzie składał się z odpowiednich sekcji (Rys.1), zawierał komentarze poprzedzone znakiem #, mnemoniki, deklaracje zmiennych oraz buforów.

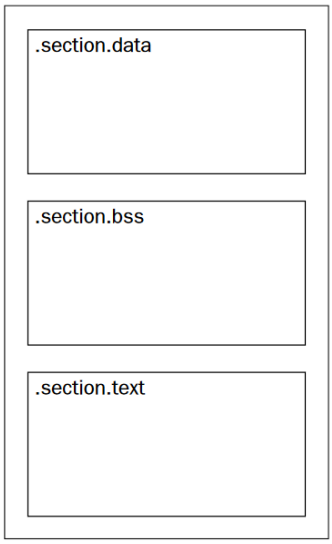

Figure 4-1

Rys.1 Definiowanie sekcji. Źródło: *Proffesional Assembly Language*, Richard Blum

W sekcji **.data** dokonujemy definicji nazw symbolicznych, które znacząco uczytelnią kod assemblerowy. Numery funkcji przypisano, posiłkując się plikiem unistd.h.

```
.data
SYSREAD = 3             # nr funkcji odczytu wejścia
SYSWRITE = 4            # nr funkcji wyjścia
SYSEXIT = 1             # nr funkcji restartu
STDOUT = 1              # nr funkcji wyjścia standarowego
STDIN = 0               # nr funkcji wejścia standarowego
EXIT_SUCCESS = 0
```

Napis “Hello World!” umieszczamy również w sekcji data (wyjątkowo), zakładając możliwe późniejsze modyfikacje. Ogólnie nie zaleca się jednak tego robić.

```
buf: .ascii “Hello World!\n”
buf_len = .-buf
```

Dyrektywa .ascii jest potrzebna, by zadeklarować tekst typu string, używając znaków ASCII.

Długość bufora jest nam potrzebna do wywołania funkcji wypisującej na ekranie – u nas nazwanej SYSWRITE, gdyż będzie jego argumentem.

Długość obliczamy w następujący sposób: od bieżącego miejsca (oznaczanego kropką) odejmujemy początek ciągu znaków (buf). Musimy to zrobić bezpośrednio poniżej deklaracji napisu, aby kropka oznaczała to miejsce, o które nam chodzi.

Poniższy kod wywołuje funkcję systemową SYSWRITE.

```
.text
.globl _start           #wskazujemy od jakiej etykiety
                        zaczynamy wejście do    programu


_start:                 #używamy etykiety start


movq $SYSWRITE, %rax    #funkcja do wywołania to SYSWRITE
movq $STDOUT, %rdi      #1 arg - out
movq $buf, %rsi         #2 arg – adres początkowy napisu
movq $buf_len, %rdx     #3 arg – długość łańcucha
syscall                 #wywołanie przerwania programowego
                        #wykonanie funkcji systemowej
```

Dyrektywa **.global** wskazuje etykietę, od której zaczyna się program – jest ona udostępniana przez program ładujący ld. Jest to konieczne, żeby było wiadomo, gdzie zacząć pierwszą instrukcję.

W Linuxie 64-bitowym operujemy na rejestrach 64-bitowych m.in. RAX, RBX, RCX, RDX, RDI, RSI. Dwa ostatnie wykorzystywane są najczęściej jako liczniki.

W rejestrze RAX umieszczamy numer funkcji (u nas SYSWRITE = 4). W rejestrach RDI, RSI, RDX umieszczamy kolejne argumenty. Wywołujemy przerwanie systemowe poleceniem **syscall**, by funkcja mogła się wykonać.

Aby na końcu doszło do wyjścia z programu, należy wywołać funkcję systemową SYSEXIT w następujący sposób:

```
movq $SYSEXIT, %rax      #funkcja do wywołania to SYSEXIT
movq $EXIT_SUCCESS, %rdi #1 arg – EXIT_SUCCESS
syscall                  #wykonanie funkcji
```

Wpisz – wypisz

Aby móc pobierać tekst od użytkownika i wyświetlać go na ekranie, należy wprowadzić do poprzedniego programu pewne poprawki.

W sekcji .data należy wprowadzić długość bufora.

```
BUFLEN = 512
```

W sekcji .bss znajdują się niezainicjalizowane dane. Tutaj wprowadzimy bufory textin i textout, gdyż nie możemy ich oczywiście zainicjalizować:

```
.comm textin, 512
.comm textout, 512
```

Po _start: zamiast wypisywania „HelloWorld!” na ekranie, wywołujemy funkcje wczytania tekstu, czyli SYSREAD.

```
movq $SYSREAD, %rax      #funkcja do wywołania to SYSREAD
movq $STDIN, %rdi        #1 arg – systemowy deskryptor stdin
movq $textin, %rsi       #2 arg – bufor do zapisania tego, co
                         wpisał użytkownik
movq $BUFLEN, %rdx       #3 arg – długość łańcucha
syscall                  #wywołanie funkcji
```

Analogicznie odbywa się wypisanie zmiennej na ekranie:

```
movq $WRITE, %rax        #funkcja do wywołania to SYSWRITE
movq $STDOUT, %rdi       #1 arg – systemowy deskryptor stdout
movq $textin, %rsi       #2 arg – adres początkowy napisu
movq $BUFLEN, %rdx       #3 arg – długość łańcucha
syscall                  #wywołanie funkcji
```

Kod zamknięcia programu wygląda dalej tak samo jak w przykładzie Hello World!.

To Upper Case – zamiana podanych liter z małych na wielkie

Na początku w sekcji .data podajemy definicje systemowych wywołań:

```
.data
SYSREAD = 3             # nr funkcji odczytu wejścia
SYSWRITE = 4            # nr funkcji wyjścia
SYSEXIT = 1             # nr funkcji restartu
STDOUT = 1              # nr funkcji wyjścia standarowego
STDIN = 0               # nr funkcji wejścia standarowego
EXIT_SUCCESS = 0
```

W sekcji .data ustalamy również rozmiar bufora dla wprowadzanego i skonwertowanego tekstu.

```
#Rozmiar bufora dla tekstu wprowadzanego
bufferSize = 512
```

W sekcji .bss tworzymy bufory – jeden z nich posłuży nam do tekstu wprowadzanego, a drugi do konwertowanego

```
#Bufory do wprowadzanego tekstu i przerobionego na wielkie
litery tekstu
.bss
.comm input, bufferSize
.comm converted, bufferSize
```

Po omówionym wcześniej fragmencie kodu, nastąpi wczytanie.

```
.text
  .globl _start
_start:
```

Teraz wywołujemy funkcje wczytania tekstu, czyli SYSREAD, podając jej trzy argumenty.

1. arg – systemowy deskryptor stout, 2. arg – adres początkowy napisu, 3.arg – długość łańcucha

```
  movq $SYSREAD, %rax
  movq $STDIN, %rdi
  movq $input, %rsi
  movq $bufferSize, %rdx
  syscall
```

```
#‘/n’
dec %rax
```

Umieszczamy 0 w rejestrze, który będzie użyty do indeksowania aktualnej litery.

```
movq $0, %rdi
```

Tworzymy pętle, która będzie zamieniała litery małe na duże. Musimy ustawić etykietę zamien_wielkosc_liter, żeby instrukcją jl zamien_wielkosc_liter wracać na początek pętli.

```
zamien_wielkosc_liter:
```

Wiedząc o tym, że małe i wielkie litery różnią się na piątym bicie od końca, dokonujemy operacji XOR na wczytanych literach i na „masce” typu …010000.

Korzystając z:

offset (%base, %index, multiplier) base+ index* multiplier+ offset

widzimy, że w tej linii kodu przemieszczamy pojedynczą literę z bufora nazwanego input to rejestru AH.

```
movb input(, %rdi, 1), %ah
```

Wpisujemy 0x20 (5), a potem wykonujemy operację XOR na zawartości rejestrów AH I AL.

```
movb $0x20, %al
xor %ah, %al
```

Suma logiczna wyłączająca działa w następujący sposób:

Wynik operacji XOR na każdej parze bitów rejestrów ebx oraz ecx ($ecx_i := ecx_i \oplus ebx_i$)*

Przypisano wynik operacji do AL. Teraz do bufora converted kopiujemy zawartość tego rejestru.

```
movb %al, converted(,%rdi,1)
```

Po czym wykonujemy operację inkrementacji poleceniem inc, które działa tak:

incl %eax - zwiększ o 1 zawartość rejestru eax, nie zmienia CF*

```
inc %rdi
```

W ten sposób zwiększyliśmy indeks.

Porównujemy zawartość rejestru rax (‘/n’) z literą. Jeśli wartość wyjdzie mniejsza, kontynuujemy pętlę. Jeśli nie, zatrzymujemy. Znaczy to, że doszliśmy do końca ciągu znaków.

```
cmp %rax, %rdi
```

```
  jl zamien_wielkosc_liter
```

Zakańczamy łańcuch znaków, kopiując do bufora z wielkimi literami ‘/n’.

```
  movb $'/n', converted(, %rdi,1)
```

Na koniec drukujemy słowo z bufora converted na STDOUT

```
  movq $SYSWRITE, %rax
  movq $STDOUT, %rbx
  movq $converted, %rcx
  movq BUFLEN, %rdx
  syscall
```

Oczywiście na końcu należy zakończyć program podaną niżej instrukcją, omówioną już wcześniej.

```
  movq $SYSEXIT, %rax
  movq $EXIT_SUCCES, %rdi
  syscall
```

Wnioski

Program zamiany liter z małych na duże opierał się na pomyśle XORowania bitu z bitem maski …00010000. Można to zadanie zrealizować również inaczej. Najpierw w sekcji .data obliczylibyśmy różnicę między małą a wielką literą, np.:

```
distance = ‘a’ – ‘A’
```

Musielibyśmy określić również pierwszą i ostatnią literę oraz rozmiar tekstu wprowadzanego przez użytkownika.

```
firstLetter = ‘a’
lastLetter = ‘z’
textLength: .int 0
```

Pętla wyglądałaby natomiast tak:

```
loop:
    # Porównanie indexu z długością tekstu
    cmpl %rdi, textLength
    # Jeżeli równe, skocz do etykiety – wypisywanie slowa
    je wypisz_zmieniona_litere
```

```
    # Jeżeli nie, idziemy dalej

    # Jak w poprzednim przykładzie, pojedyncza litera
kopiowana z bufora wejściowego do AL

    movb input(, %rdi, 1), %al

    # Porównanie litery z rejestru AL (wprowadzonej przez
użytkownika) z wartością litery ‘a’

    cmpb $firstLetter, %al

    # Jeżeli wartość jest mniejsza, skocz do wypisania litery

    jb wypisz_zmieniona_litere

    # Porównanie litery z rejestru AL (wprowadzonej przez
użytkownika) z wartością litery ‘z’

    cmpb $lastLetter, %al

    # Jeżeli wartość większa, skocz do wypisania litery

    ja wypisz_zmieniona_litere

    # Jeżeli wartość nie jest małą literą, to zmień ją na dużą
literę, odejmując dystans litery

    subb $distance, %al

    # Wypisz słowo po przeróbce

    jmp wypisz_zmieniona_litere
```

Litera jest w rejestrze AL – kopiujemy ją do bufora i dalej przeskakujemy do pętli poprzedniej.

```
wypisz_zmieniona_litere:

  # Kopiujemy do bufora wyjściowego

  movb %al, converted(, %rdi, 1)

  # Zwiększamy indeks

  incq %rdi

  # Powracamy do pętli

  jmp loop
```

Źródła:

http://zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Dokumentacja/Programowanie/Linux-asm-lab-2015.pdf

Proffesional Assembly Language, Richard Blum

Kod zmiany liter na duże wykonany na podstawie programu w prezentacji ze strony Zakład Architektury Komputerów: http://zak.ict.pwr.wroc.pl

* Prezentacja profesora Janusza Biernata
http://zak.ict.pwr.wroc.pl/materials/architektura/wyklad%20AK2/AK2-1%20-programowanie'17.pdf

IA-32 Intel ® Architecture Software Developer's Manual