Aleksandra Ojdana, 198842
2019
PT-N-17

Wrocław, dn. 29 marca

Prowadzący: dr Piotr Patronik

Laboratorium Architektura Komputerów
Podstawy uruchamiania programów asemblerowych na platformie Linux/x86

1 Treść ćwiczenia

1.1 Zakres i program ćwiczenia:

Celem ćwiczenia było poznanie podstawowych funkcjonalności narzędzia GNU GDB, konstrukcji pliku Makefile, a także stworzenie programu w języku asembler, kompilacja programu oraz uruchomienie stworzonego programu na platformie Linux/x86. Ponadto podczas ćwiczeń należało zapoznać się z następującymi zagadnieniami:

- podstawowe funkcję języka asembler,
- składnia pliku źródłowego
- funkcje systemowe oraz ich wywoływanie
- analiza programów asemblerowych z wykorzystaniem GNU gdb
- podstawowe polecenia systemu Linux oraz narzędzia GNU Binutils

1.2 Zrealizowane zadania podczas ćwiczeń:

- utworzenie pliku sterującego Makefile
- utworzenie i uruchomienie w środowisku 32-bitowym programów:
  * hello - wyświetlanie napisu „hello world!"
  * linijka - wyświetlanie linii z gwiazdek (*) o ustalonej długości
  * balwan – wyświetlanie dwóch rombów z gwiazdek, mniejszego nad większym
  * linijka2 - wyświetlanie linii z gwiazdek (*) o zadanej przez użytkownika długości (program nie został ukończony na ćwiczeniach)
- zapoznanie się z podstawowymi poleceniami systemu Linux: ls, cd, mkdir, touch, rm, cp
- uruchomienie stworzonych programów pod kontrolą gdb oraz przetestowanie poleceń gdb: run, break, step, stepi, continue, delete, quit, help, disassemble, next, print/d, info registers

1.3 Zadania zrealizowane poza ćwiczeniami:

- dokończenie programu linijka2
- dokładniejsze zapoznanie się ze składnią pliku źródłowego m.in. z tym co oznacza dwukropek
- jak poprawnie używać funkcji disassemble

2 Przebieg ćwiczenia

2.1 Uruchomienie programu

Realizację ćwiczenia rozpoczęto od utworzenia pliku sterującego Makefile dla programu **make** [2] sterującego procesem kompilacji. W pliku Makefile umieszczono polecenia odpowiedzialne na kompilowanie pliku źródłowego oraz konsolidację pliku wyjściowego procesu kompilacji. Podczas procesu kompilacji plik źródłowy tłumaczony jest na kod maszynowy i zapisywany do pliku wyjściowego z rozszerzeniem **'.o'** [1]. Wykorzystano do tego polecenie **as** z flagą **'--32'** [1], która informuję kompilator, żeby kod wyjściowy był skompilowany w formacie 32-bitowym [1]. Proces konsolidacji odpowiada za dołączenie do pliku z kodem binarnym odpowiednich bibliotek i plików nagłówkowych oraz stworzenie pliku wykonywalnego. Wykorzystano do tego polecenie **ld** [3] z flagą **–melf_i386** [3], która informuję konsolidator, żeby plik wykonywalny był utworzony pod platformę x86. W załączniku nr 1 umieszczono zawartość pliku Makefile stworzonego podczas zajęć, w załączniku nr 3 umieszczono zmodyfikowany po zajęciach plik Makefile. Podczas ćwiczeń poza poleceniami as or ld należało się zapoznać z poleceniem **gcc** [4]. Polecenie gcc łączy w

sobie funkcjonalność poleceń as oraz ld. Składnia wszystkich trzech poleceń jaką wykorzystano jest następująca: *nazwa_polecenia flaga plik_docelowy plik_źródłowy.* Uruchomienie zawartości pliku Makefile uzyskano poprzez użycie komendy **make** [2] w linii komend w folderze, w którym plik Makefile się znajdował.

Następnie stworzono pliki źródłowe z programami: hello, linijka, balwan oraz linijka2. Kody źródłowe programów zrealizowanych na ćwiczeniach umieszczono w załączniku nr 2, w załączniku nr 3 umieszczono poprawnie działający kod programu linijka2 wykonany po ćwiczeniach. Poniżej opisano konstrukcję plików źródłowych.

2.2 Konstrukcja pliku źródłowego

Plik źródłowy rozpoczyna się od definicji symboli wraz z argumentami dla wybranych funkcji systemowych. Numery wykorzystanych funkcji systemowych pobrano z pliku asm/unistd.h [7]. W stworzonych programach wykorzystano poniższe funkcje systemowe:

RETURN = 0 – wartość umieszczana w rejestrze ebx, zwraca kod poprawnego wykonania programu
EXIT = 1 – wartość umieszczana w rejestrze eax, kończy wykonywanie programu
STDOUT = 1 – wartość umieszczana w rejestrze ebx, wyświetla dane na standardowym wyjściu
STDIN = 0 – wartość umieszczana w rejestrze ebx, pobiera dane ze standardowego wejścia
READ = 3 – wartość umieszczana w rejestrze eax, funkcja odczytu danych
WRITE = 4 – wartość umieszczana w rejestrze eax, funkcja zapisu danych

Następnie po nazwach symbolicznych w pliku źródłowym została umieszczona dyrektywia „ **.align 32**”, która zmusza kompilator do wyrównania kodu programu do granicy słowa maszynowego [10]. Kolejną sekcją programu jest sekcja danych rozpoczynająca się dyrektywą „ **.data**”. Dyrektywy wskazują kompilatorowi jak ma się zachowywać m.in. poprzez wskazywanie granic segmentów, rozkład danych w pamięci [8]. Po dyrektywie „ .data” umieszczono symbole oraz etykiety będące symbolami po których jest dwukropek, czyli nazwy adresów, pod którymi umieszczone są dane [8]. W celu zdefiniowania etykiety po dwukropku podaje się typ danych oraz wartość jaką chcemy zapisać pod danym adresem. Poniżej umieszczono kod sekcji z danymi z programu linijka.s.

```
.section .data

star: .ascii "*"
star_len = . - star
chain: .long 5
new_line: .ascii "\n"
new_line_len = . - new_line
```

W powyższym kodzie zapis „star_len = . - star” rozumie się w następujący sposób: do symbolu „star_len” przypisz wartość odpowiadającą wielkości miejsca od obecnego adresu w pamięci do miejsca, w którym jest początek danych z „star”. Analogicznie jest dla „new_line_len”.

Kolejnym elementem pliku źródłowego jest wskazanie punktu wejścia programu przy użyciu „ **.global _start**” oraz początek funkcji rozpoczynającej program „_start:”. Dyrektywa .global wskazuje jaki symbol, w tym przypadku _start, ma być wykorzystany przez program ładujący. Jest to operacja niezbędna do uruchomienia programu [9]. Jako przykład do omówienia tej sekcji również wykorzystano kod z programu linijka.s.
Kod w sekcji _start można podzielić na cztery elementy. W pierwszym z nich znajduje się umieszczenie wartości 0 w rejestrze edi, który posłużył za licznik oraz umieszczenie w rejestrze edx wartości określającej długość danych jaka będzie wyświetlana.

```
mov $0, %edi
mov $star_len, %edx
```

Drugim elementem jest pętla odpowiedzialna za wyświetlanie gwiazdek na ekranie oraz zwiększanie wartości licznika rejestru edi.

```
loop_start:          # początek pętli
cmpl chain, %edi     # porównanie wartości chain z wartością przechowywaną w rejestrze edi
je loop_end          # instrukcja skoku, jeżeli dane chain i edi są równe, wykonaj skok do etykiety
loop_end
```

```
inc %edi                # zwiększ wartość w rejestrze o jeden
mov $star, %ecx         # wyświetlenie gwiazdki na ekranie
mov $STDOUT, %ebx
mov $WRITE, %eax
int $0x80               # wywołanie przerwania systemowego
jmp loop_start          # skok do miejsca z etykietą loop_start
```

Trzecim elementem jest zakończenie pętli.

```
loop_end:                       # etykieta wskazująca koniec pętli
mov $new_line_len, %edx
mov $new_line, %ecx
mov $STDOUT, %ebx
mov $WRITE, %eax
int $0x80
```

Ostatni czwarty element odpowiada za poprawne zakończenie programu z kodem 0.

```
mov $RETURN , %ebx      # poprawne zakończenie programu
mov $EXIT, %eax
int $0x80
```

Dodatkowo w programie balwan wykorzystano funkcję oraz dyrektywe

2.3 Uruchomienie programu pod kontrolą GDB

W celu uruchomienia programu pod kontrolą gdb w terminalu w miejscu z plikiem wykonywalnym wpisuje się komendę „gdb *nazwa_programu”*. Wcześniej natomiast podczas kompilacji programu warto dodać flagę „-g” do as, która generuję tablicę powiązań symboli z kodem wynikowym. Ułatwia to późniejsze korzystanie z debugera [9].
Po uruchomieniu debugera wyświetla się następujący komunikat:

GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from linijka...done.

W przypadku, gdy nie zostanie użyta opcja -g podczas kompilacji powyższy komunikat będzie się różnił ostatnią linią, która bez opcji -g byłaby następująca:
Reading symbols from line...(no debugging symbols found)...done.
Po uruchomieniu programu pod kontrolą gdb, przetestowano wszystkie komendy gdb wymienione w punkcie 1.2. Opis funkcjonalność testowanych komend pobrano z helpa gdb (*nazwa_funkcji* -help) oraz książki „”Debugging with GDB” [6]:

run – uruchomienie wczytanego programu

break – umieszczenie w programie breakpointa, po poleceniu break można podać nr linii lub nazwę funkcji

step – po zatrzymaniu się programu, funkcja step uruchamia następny krok programu, jeżeli jest to funkcja to nie wchodzi do środka funkcji.

stepi – tak samo jak step, z tym, że w przypadku trafienia na funkcję, wchodzi do niej

continue – kontynuuje debugowanie do następnego breakpointa, przerwania lub końca

delete – usuwa wcześniej ustawione breakpointy w programie

quit – zakończenie pracy debugera

help – sprawdzanie opisu funkcji

disassemble - wykonuje zrzut pamięci wczytanego programu w danym miejscu, po komendzie podaje się nazwę funkcji lub adres w pamięci

next – przejście programu do następnej linii

print/d – wyświetlenie wartości zadanej zmiennej lub rejestru

info registers – wyświetlenie wartości wszystkich rejestrów

3. Wnioski

Przed przystąpieniem do pisania programów w języku asembler wypada zapoznać się szczegółowo ze składnią pliku źródłowego programu asemblerowego, szczególnie ze sposobem tworzenia stałych oraz znaczeniem białych znaków. Mają one istotny wpływ na pracę programu, a niepoprawne tworzenie pliku źródłowego i rozmieszczenia białych znaków może powodować błędy trudne do wyłapania nawet podczas debugowania. Kolejnym istotnym elementem przed przystąpieniem do pracy nad programami asemblerowymi jest zapoznanie się funkcjonalnością debugera GDB, gdyż w znacznej mierze przyspiesza on pracę. Kolejną rzeczą jaka usprawnia pracę nad programami jest wykorzystanie programu make do kontroli procesu kompilacji i konsolidacji programów. Stworzenie pliku Makefile powinno być pierwszą rzeczą rozpoczynającą prace nad programami.

Głównym problemem podczas realizacji ćwiczenia była implementacji funkcjonalności odpowiedzialnej za wczytywanie danych z klawiatury a następnie zamiana tych danych na liczby. Problem pojawił się w programie linijka2. Ostatecznie został rozwiązany.

Programy pisane w języku asembler dają programiście całkowitą kontrolę nie tylko nad przebiegiem programu,ale również nad sposobem alokowanie zasobów komputera. Odbywa się to kosztem czasu oraz większą ilością kodu. Do wyświetlenia napisu „hello world” w języku asembler potrzebne jest wywołanie pięciu funkcji w przypadku języka C lub C++ jest to wywołanie jednej funkcji.

4. Literatura

1. https://sourceware.org/binutils/docs-2.32/as/index.html dn. 18.03.2019
2. https://www.gnu.org/software/make/manual/make.html dn. 18.03.2019
3. https://sourceware.org/binutils/docs-2.32/ld/index.html dn. 18.03.2019
4. https://gcc.gnu.org/onlinedocs/gcc-8.3.0/gcc/ dn. 18.03.2019
5. J. Bartlett “Programming from the Ground Up” GNU Free Documentation License
6. R. Stallman, R. Pesch, S. Shebs, et al. “Debugging with GDB” Free Software Foundation
7. /usr/include/asm/unistd.h, plik nagłówkowy kompilatora gcc z listą kodów funkcjisystemowych systemuLinux
8. Wprowadzenie do laboratorium ze strony: http://zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Wprowadzenie%20do%20laboratorium.pdf
9. Przykładowe sprawozdania ze strony: http://zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2

Załącznik nr 1 – zawartość pliku Makefile

```
lab1: lab1.o linijka.o balwan.o line.o
	ld -melf_i386 lab1.o -o lab1
	ld -melf_i386 linijka.o -o linijka
	ld -melf_i386 balwan.o -o balwan
	ld -melf_i386 line.o -o line

lab1.o: lab1.s linijka.s balwan.s line.s
	as --32 -g lab1.s -o lab1.o
	as --32 -g linijka.s -o linijka.o
	as --32 -g balwan.s -o balwan.o
	as --32 -g line.s -o line.o

all: lab1 linijka balwan line

clean:
	rm -f lab1 lab1.o
	rm -f linijka linijka.o
	rm -f balwan balwan.o
	rm -f line line.o
```

Załącznik nr 2 – programy wykonane podczas zajęć

1. hello

```
EXIT =1
WRITE =4
STDOUT =1
RETURN =0

.align 32

.section .data
hello: .ascii "Hello world!\n"
hello_len = . - hello

.global _start
_start:

mov $hello_len, %edx
mov $hello, %ecx
mov $STDOUT, %ebx
mov $WRITE, %eax
int $0x80

mov $RETURN, %ebx
mov $EXIT, %eax
int $0x80
```

2. balwan

```
EXIT =1
WRITE =4
STDOUT =1
STDIN =0
RETURN =0

.section .data
```

```
star: .ascii "*"
space: .ascii " "
space_len = . - space
star_len = . - star
size_small: .long 3
size_big: .long 5
new_line: .ascii "\n"
new_line_len = . -new_line

.global _start
_start:

mov $1, %edi

print_small_ball:
cmp size_small, %edi
jg print_small_ball_end
push %edi
push %edi
call print_line
add $8, %esp
mov %eax, %edi
inc %edi
inc %edi
jmp print_small_ball

print_small_ball_end:
mov $1, %edi

print_big_ball:
cmp size_big, %edi
jg print_big_ball_end
push %edi
push %edi
call print_line
add $8, %esp
mov %eax, %edi
inc %edi
inc %edi
jmp print_big_ball

print_big_ball_end:
push $1
call print_line
add $8, %esp
mov $RETURN, %ebx
mov $EXIT, %eax
int $0x80

.type print_line, @function
print_line:

push %ebp
mov %esp, %ebp
mov $0, %edi
mov $star_len, %edx

loop_start:
cmp %edi, 8(%ebp)
je loop_end
```

```
inc %edi
mov $star, %ecx
mov $STDOUT, %ebx
mov $WRITE, %eax
int $0x80
jmp loop_start

loop_end:
mov $new_line_len, %edx
mov $new_line, %ecx
mov $STDOUT, %ebx
mov $WRITE, %eax
int $0x80

mov 12(%ebp), %eax
mov %ebp, %esp
pop %ebp
ret
```

3. linijka2

```
EXIT =1
READ =3
WRITE =4
STDOUT =1
STDIN =0
RETURN =0

.section .data

star: .ascii "*"
star_len = . - star
chain: .long 5
new_line: .ascii "\n"
new_line_len = . -new_line

.global _start
_start:

mov $0, %edi
mov $star_len, %edx

loop_start:
cmpl chain, %edi
je loop_end
inc %edi
mov $star, %ecx
mov $STDOUT, %ebx
mov $WRITE, %eax
int $0x80
jmp loop_start

loop_end:
mov $new_line_len, %edx
mov $new_line, %ecx
mov $STDOUT, %ebx
mov $WRITE, %eax
int $0x80

mov $RETURN, %ebx
```

```
mov $EXIT, %eax
int $0x80
```

Załącznik nr 3 – program linijka2

```
EXIT =1
READ =3
WRITE =4
STDOUT =1
STDIN =0
RETURN =0

.section .data

star: .ascii "*"
star_len = . - star
chain: .long 5
new_line: .ascii "\n"
new_line_len = . -new_line

.global _start
_start:

mov $0, %edi
mov $star_len, %edx

loop_start:
cmpl chain, %edi
je loop_end
inc %edi
mov $star, %ecx
mov $STDOUT, %ebx
mov $WRITE, %eax
int $0x80
jmp loop_start


loop_end:

mov $new_line_len, %edx
mov $new_line, %ecx
mov $STDOUT, %ebx
mov $WRITE, %eax
int $0x80

mov $RETURN, %ebx
mov $EXIT, %eax
int $0x80
```