

*Środowisko programistyczne laboratorium*  
*Architektury komputerów*  
sprawozdanie z laboratorium przedmiotu „Organizacja i Architektura Komputerów”

Rok akad. 2018/2019, kierunek: INF

PROWADZĄCY:  
dr inż. Piotr Patronik

## Spis treści

<b>1</b>	<b>Cel ćwiczenia</b>	<b>2</b>
<b>2</b>	<b>Przebieg ćwiczenia</b>	<b>2</b>
2.1	Konstrukcja pliku źródłowego hello.s . . . . .	2
2.2	Konstrukcja pliku Makefile . . . . .	3
2.3	Kompilacja programu kompilatorem gcc . . . . .	4
2.4	Debugowanie - gdb . . . . .	4
<b>3</b>	<b>Podsumowanie i wnioski</b>	<b>6</b>
	<b>Bibliografia</b>	<b>6</b>
<b>4</b>	<b>Listing Kodu</b>	<b>6</b>
4.1	Plik hello.s . . . . .	6
4.2	Plik stars.s . . . . .	7
4.3	Plik tree.s . . . . .	7
4.4	Niektóre nieopisane wyżej działania . . . . .	10

# 1 Cel ćwiczenia

Celem ćwiczenia było zapoznanie się z podstawami programowania w języku assemblera AT&T na platformie Linux/x86. Skompilowanie, linkowanie, i debugowanie programu napisanego w trakcie laboratorium. Wykonując ćwiczenia zapoznano się z następującymi zagadnieniami:

- assemblowanie programu używając assemblera **GNU as**, poprzez użycie komendy **as**.
- linkowanie programu używając linkera **GNU linker**, poprzez użycie komendy **ld**.
- debugowanie programu - analiza, podglądanie wartości zapisanych w używanych rejestrach i adresach pamięci, za pomocą debugera **GNU Debugger** - polecenie **gdb**.
- kompilacja programu za pomocą kompilatora **GNU Compiler Collection** - polecenie **gcc**
- napisanie pliku automatyzującego proces kompilacji programu - **Makefile**, uruchamianie go za pomocą polecenia **make**.

## 2 Przebieg ćwiczenia

### 2.1 Konstrukcja pliku źródłowego hello.s

Zgodnie z wcześniej napisanymi na tablicy zadaniami/ćwiczeniami, został utworzony pierwszy plik źródłowy **hello.s**, który edytowany był edytorem **vim**. Następnie przystąpiono do pisania programu wyświetlającego napis "Hello World". Program został podzielony na sekcje, poprzez napisanie dyrektyw: `'.section .data'`, `'.section .text'`. W miejscu pod `'.section .data'` zostały zadeklarowane zmienne:

- `msg: .string "Hello World"` - napis, który będzie wyświetlany,
- `msg_len = . - msg` - długość wyświetlanego napisu.

Miejsce między `'.section .text'` a dyrektywą `'.globl _start'` zostało puste. Następnie pod spodem etykiety `._start:` zostało napisane "ciało" programu. Aby wyświetlić na ekranie napis "Hello World", należy wykonać odpowiednie wywołanie systemowe. W celu wyświetlenia napisu skorzystano z wywołania systemowego **write**, aby wywołać polecenie systemowe drukowania na ekran należy:

- wprowadzić wartość 4 do rejestru `%eax` - 4 oznacza wywołanie systemowego polecenia **write**,
- wprowadzić wartość 1 do rejestru `%ebx` - 1 oznacza **standard output**.

Do wprowadzenia wartości do rejestrów użyto trybu adresowania nazywanego **immediate mode**, tryb ten załadowuje bezpośrednio podane wartości do rejestrów, bądź adresów pamięci. Należy pamiętać o znaku `$` w tym trybie adresowania, gdyby go nie było załadowalibyśmy do rejestru to, co się znajduje w pamięci pod adresem np. 4.  
*przykład:* `movl $4, %eax` - załadownie wartości 4 do rejestru `%eax`.

- wprowadzić do rejestru `%ecx` początek buffora, który przechowuje dane do wyświetlenia, zostało to wykonane poleceniem: `movl $msg, %ecx`,
- wprowadzić do rejestru `%edx` wielkość wyświetlanego bufforu, zostało to wykonane poleceniem: `movl $msg_len, %edx`.

Po załadowaniu odpowiednich danych do, wymagających tego, rejestrów wykonano wywołanie polecenia systemowego: `int $0x80`. Następnie aby zakończyć program wykonano wywołanie systemowe - `exit`. Aby tego dokonać, do rejestru `%eax`, wpisano wartość 1, zostało to wykonane poleceniem: `movl $1, %eax`.

Wywołując polecenie `exit`, wpisano również do rejestru `%ebx` wartość liczby zwracanej do systemu - 0, dokonano tego poleceniem: `movl $0,%ebx`. Po tych zabiegach wykonano wywołanie polecenia systemowego: `int $0x80`.

Program zassemblewano poleceniem `as` oraz zlinkowano poleceniem `ld`. Wykonania tych operacji i wykonanie programu widać na zrzucie ekranu:

```
michal@michal-VirtualBox: ~/Desktop/241130/hello
michal@michal-VirtualBox:~/Desktop/241130/hello$ as hello.s -o hello.o
michal@michal-VirtualBox:~/Desktop/241130/hello$ ld hello.o -o hello
michal@michal-VirtualBox:~/Desktop/241130/hello$ ./hello
Hello World
michal@michal-VirtualBox:~/Desktop/241130/hello$
```

## 2.2 Konstrukcja pliku Makefile

Aby zautomatyzować assemblewanie i linkowanie (w razie zmian w programie, oraz w celu przećwiczenia tworzenia pliku Makefile) pliku źródłowego `hello.s` został utworzony plik Makefile pozwalający na korzystanie z programu sterującego procesem kompilacji `make`. Folder przechowujący pliki programu zawiera następujące pliki:

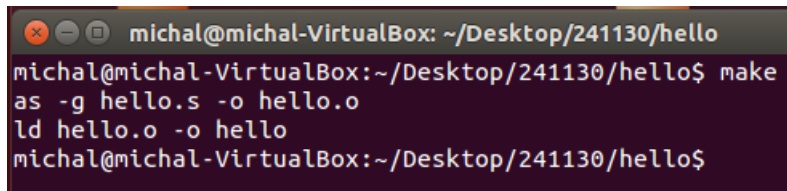
- `hello.s` - plik z kodem źródłowym programu,
- `hello.o` - object file, plik obiektowy zawierający skompilowany kod programu,
- `hello` - plik wykonywalny programu.

Plik `hello.o` jest tworzony przez narzędzie `as`, w trakcie assemblewania pliku źródłowego `hello.s`, więc można powiedzieć, że zawartość tego pliku jest zależna od procesu assemblewania pliku `hello.s`. Podobnie w przypadku pliku `hello`, który zależy od procesu linkowania pliku `hello.o`. Narzędzie `make` pozwala stworzyć plik, który opisuje te zależności i pozwala na wykonywanie operacji gdy, pliki są nieaktualne. Plik Makefile edytowany był edytorem `vim`,

```
michal@michal-VirtualBox: ~/Desktop/241130/hello
hello: hello.o
    ld hello.o -o hello

hello.o: hello.s
    as -g hello.s -o hello.o
```

W pliku Makefile, zostały opisane zależności pomiędzy plikami. Pierwsza linijka tekstu na zrzucie ekranu mówi o tym, że plik `hello`, jest zależny od pliku `hello.o`, bliźniaczo trzecia linijka tekstu opisuje zależność pliku `hello.s` od pliku `hello.o`. Druga i trzecia linijka tekstu, są poleceniami, które należy wykonać w przypadku gdy data edycji pliku zależnego jest starsza od daty edycji pliku, od którego ten plik zależy. Komenda `as`, została wzbogacona o opcję `-g`, która generuje informacje dla debuggera. Należało pamiętać również, że polecenia, które mają się wykonać w przypadku, gdy któryś z plików jest nieaktualny, muszą być poprzedzone tabulatorem. Plik `Makefile`, został zapisany, a plik `hello.s` zmodyfikowany. Następnie przetestowano działanie komendy `make`.



```
michal@michal-VirtualBox: ~/Desktop/241130/hello$ make
as -g hello.s -o hello.o
ld hello.o -o hello
michal@michal-VirtualBox: ~/Desktop/241130/hello$
```

Program został automatycznie zassembledowany i zlinkowany.

## 2.3 Kompilacja programu kompilatorem gcc

Kompilator gcc, różni się od narzędzi `as`, `ld` tym, że posiada on w sobie narzędzie assemblujące program i linkujące jednocześnie. Jest to wygodne narzędzie, jego działanie przetestowano w trakcie trwania laboratorium. Aby użyć narzędzia gcc zmieniono nazwę etykiety `_start` na `main`. Kompilator gcc wymaga zastąpienia etykiety `_start` na etykietę `main`. Po dokonanej zmianie etykiety, przystąpiono do kompilacji programu, dokonano tego komendą: `gcc hello.s`. Program został skompilowany bez żadnych błędów. Następnie wykonano polecenie `./hello` w celu sprawdzenia poprawności działania programu. Program zadziałał zgodnie z oczekiwaniami.

## 2.4 Debugowanie - gdb

Aby przećwiczyć działanie debuggera gdb, program `hello` został uruchomiony za pomocą tego narzędzia. W terminalu wykonano polecenie: `gdb hello`, które uruchomiło debugger. Następnie wedle ustaleń zaczęto testować odpowiednie komendy:

- aby zobaczyć listing kodu programu, wykonano polecenie `list`,
- ustawiono dwa breakpointy (miejsca zatrzymania się debuggera) poleceniem `break 19` oraz `break 24` - 19, 24 to linie programu w, których debugger ma się zatrzymać,
- program pod kontrolą debuggera, uruchomiono poleceniem `run`.

Breakpoint'y ustawione zostały tuż, przed wywołaniami systemowymi, pierwszy przed drukującym napis `Hello World`, a drugi przed wywołaniem kończącym program.

- Aby upewnić się, że w rejestrach znajdują się pożądane wartości wykonano polecenie `info registers`, które powoduje wyświetlenie listy rejestrów wraz z ich zawartością.

Interesują nas tylko rejestry %eax, %ebx, %ecx, %edx, dlatego nie zamieszczam zrzutu ekranu wszystkich rejestrów.

```
(gdb) info registers
eax             0x4          4
ecx             0x8049096      134516886
edx             0xd          13
ebx             0x1          1
```

Zauważono, że polecenia wpisania wartości 4 do rejestru %eax, wartości 1 do rejestru %ebx i długości napisu Hello World wraz ze znakiem nowej linii - \n) zostały umieszczone poprawnie w rejestrach. Do rejestru %ecx został wpisany adres pierwszego znaku zmiennej, przechowującej napis.

- Numer adresu zmiennej sprawdzono komendą: **print &msg**, zgadzał się z tym który został wpisany do rejestru.
- Aby upewnić się, że rzeczywiście pod tym adresem znajduje się litera 'H', zastosowano polecenie: **x/c 0x8049096**, które z opcją "**\c**" powoduje wyświetlenie zawartości rejestru jako znak **char** (poleceniem **print \*(ADDRESS)**, również możemy sprawdzić jaka wartość znajduje się w danej komórce pamięci, jednak bez formatowania w postaci char).

```
(gdb) x/c 0x8049096
0x8049096:      72 'H'
(gdb) █
```

Zauważono, że w istocie adres ten przechowuje liczbę, która w kodzie ASCII oznacza literę 'H'.

- Następnym poleceniem jakie przetestowano, było polecenie: **disass**, które rozkłada aktualnie wykonywaną funkcję i pokazuje wykonywane po kolei operacje.

```
(gdb) disass
Dump of assembler code for function _start:
0x08048074 <+0>:    mov     $0x4,%eax
0x08048079 <+5>:    mov     $0x1,%ebx
0x0804807e <+10>:   mov     $0x8049096,%ecx
0x08048083 <+15>:   mov     $0xd,%edx
=> 0x08048088 <+20>:  int     $0x80
0x0804808a <+22>:   mov     $0x1,%eax
0x0804808f <+27>:   mov     $0x0,%ebx
0x08048094 <+32>:   int     $0x80
End of assembler dump.
```

Zauważono, że operacje zostały wykonane zgodnie z ich kolejnością w pliku źródłowym oraz, że ustawiony breakpoint spowodował zatrzymanie wykonywania się instrukcji w miejscu w którym tego oczekiwano.

- Następnie wykonano polecenie **step**, na ekranie zgodnie z oczekiwaniami pojawił się napis Hello World, oraz gdb poinformował o instrukcji, która zostanie wykonana następnie.

- Aby przejść do drugiego ustawionego breakpointa, wykorzystano komendę: **continue 2**, która powoduje przejście programu, do drugiego ustawionego breakpoint'a (można również skorzystać z komendy **cont**, która powoduje przejście do następnego breakpointa z kolei). Po powtórnym użyciu polecenia **disass**, gdb wskazuje zgodnie z oczekiwaniami, że następną wykonywaną operacją będzie wykonanie polecenia systemowego kończącego program.

### 3 Podsumowanie i wnioski

Wykonując ćwiczenia zadane na laboratoriach udało się napisać program wyświetlający napis `Hello World` na ekranie, następnie zrozumiano jego działanie. Przecwiczone działania wykonywane przy pomocy rejestrów - wpisywanie do nich wartości, wykonywanie poleceń systemowych. Za pomocą narzędzia gdb upewniono się, że napisane operacje w pliku źródłowym są wykonywane z zaplanowanym skótkiem - wpisywanie różnych wartości do rejestrów, sprawdzanie czy pod danym adresem pamięci rzeczywiście znajduje się pożądana wartość. Opanowano proces assemblowania i linkowania, przecwiczone tworzenie plików pomagających zautomatyzować ten proces - Makefile.

### Literatura

- [1] Jonathan Bartlett, *Programming from the Ground Up* str. 1–52, 2004.
- [2] Vim narzędzie do przeglądania dokumentacji, pomocy, *vimtutor*
- [3] gdb wbudowana w narzędzie opcja pomocy, *help*

## 4 Listing Kodu

### 4.1 Plik `hello.s`

```
EXIT = 1
WRITE = 4
STDOUT = 1
SYSCALL32 = 0x80
.section .data
msg: .string "Hello World"
msg_len = . - msg
.section .text
.globl _start
_start:
movl $WRITE, %eax
movl $STDOUT, %ebx
movl $msg, %ecx
movl $msg_len, %edx
int $SYSCALL32
movl $EXIT, %eax
```

```
movl $0, %ebx
int $SYSCALL32
```

## 4.2 Plik stars.s

```
SYSCALL32 = 0x80
EXIT = 1
STDIN = 0
READ = 3
STDOUT = 1
WRITE = 4
.section .data
newline: .ascii "\n"
star: .ascii "*"
number: .long 0
.section .text
.globl _start
_start:
movl $READ, %eax
movl $STDIN, %ebx
movl $number, %ecx
movl $1, %edx
int $SYSCALL32
subl $48, number
movl number, %edi
loop_start:
cmpl $0, %edi
je loop_end
movl $WRITE, %eax
movl $STDOUT, %ebx
movl $star, %ecx
movl $1, %edx
int $SYSCALL32
subl $1, %edi
jmp loop_start
loop_end:
movl $WRITE, %eax
movl $STDOUT, %ebx
movl $newline, %ecx
movl $1, %edx
int $SYSCALL32
movl $EXIT, %eax
movl $0, %ebx
int $SYSCALL32
```

## 4.3 Plik tree.s

```
EXIT = 1
```

```

READ = 3
STDIN = 0
WRITE = 4
STDOUT = 1
SYSCALL32 = 0x80
.section .data
star: .ascii "*"
space: .ascii " "
newline: .ascii "\n"
number: .long
.section .text
.globl _start
_start:
movl $READ, %eax
movl $STDIN, %ebx
movl $number, %ecx
movl $1, %edx
int $SYSCALL32
movl number, %ebx
subl $48, %ebx
movl %ebx, %edi
pushl %edi #wysokosc choinki
pushl %ebp #zawartosc starego ebp
movl %esp, %ebp
pushl %edi #wysokosc choinki jeszcze raz
movl $1, %eax #licznik
pushl %eax #licznik
movl %edi, %eax
subl $1, %eax
pushl %eax #wysokosc do spacji na koniec
first_loop_start:
movl 4(%ebp), %edi #wysokosc choinki jako indeks
cmpl $0, %edi
je first_loop_end
movl -4(%ebp), %edi
subl $1, %edi
movl %edi, -4(%ebp)
second_loop_start:
cmpl $0, %edi
je second_loop_end
movl $WRITE, %eax
movl $STDOUT, %ebx
movl $space, %ecx
movl $1, %edx
int $SYSCALL32
subl $1, %edi
cmpl $0, %edi
je second_loop_end

```



```

jmp second_loop_start
second_loop_end:
movl -8(%ebp), %edi
third_loop_start:
movl $WRITE, %eax
movl $STDOUT, %ebx
movl $star, %ecx
movl $1, %edx
int $SYSCALL32
subl $1, %edi
cmpl $0, %edi
je third_loop_end
jmp third_loop_start
third_loop_end:
movl -8(%ebp), %edi
addl $2, %edi
movl %edi, -8(%ebp)
movl 4(%ebp), %edi
subl $1, %edi
movl %edi, 4(%ebp)
movl $WRITE, %eax
movl $STDOUT, %ebx
movl $newline, %ecx
movl $1, %edx
int $SYSCALL32
jmp first_loop_start
first_loop_end:
movl -12(%ebp), %edi
space_loop_start:
cmpl $0, %edi
je space_loop_end
movl $WRITE, %eax
movl $STDOUT, %ebx
movl $space, %ecx
movl $1, %edx
int $SYSCALL32
subl $1, %edi
jmp space_loop_start
space_loop_end:
movl $WRITE, %eax
movl $STDOUT, %ebx
movl $star, %ecx
movl $1, %edx
int $SYSCALL32
movl $WRITE, %eax
movl $STDOUT, %ebx
movl $newline, %ecx
movl $1, %edx

```

```
int $SYSCALL32
movl $1, %eax
movl $0, %ebx
int $SYSCALL32
```

## 4.4 Niektóre nieopisane wyżej działania

Listingi kodów zostały zamieszczone informacyjnie, w celu pokazania, że wszystkie programy zostały napisane. Programy zostały przetestowane i działają poprawnie, zgodnie z początkowymi założeniami.

W programach **stars.s** oraz **tree.s**, użyto nieopisanego wcześniej wywołania systemowego - **Read**, które to wywołanie powoduje zczytanie zadanej wielkości bufora do wyznaczonego adresu. Ponadto w programie **tree.s** użyto operacji działającej na stosie (**pushl** - 'wrzucenie', na stos adresu pamięci), oraz użyto trybu adresowania zwanego - "base pointer addressing mode". Program **tree.s** dodatkowo był asemblowany i linkowany z wykorzystaniem następujących opcji:

- **as --32 -g tree.s -o tree.o** - opcja **-32** generuje kod 32 bitowy
- **ld -melf\_i386 tree.o -o tree** - opcja **-melf\_i386** generuje kod 32 bitowy na maszynę 64 bitową w emulacji i386