Wrocław, dn. 14 maja 2016r

Rafał Pieniążek, 209847
PN-P-8

prowadzący: prof. Janusz Biernat

Laboratorium Architektury Komputerów
(1)Tworzenie prostych konstrukcji programowych z użyciem instrukcji asemblera Linux/x86

1 Treść ćwiczenia

Zakres ćwiczenia:

- Nabycie umiejętności związanych z tworzeniem prostych pętli oraz instrukcji warunkowych z użyciem assemblera.

2 Przebieg ćwiczenia

2.1 Zamiana wielkości liter

Na początku laboratoriów został przygotowany prosty program przetwarzający wprowadzone dane. Program wczytywał dane wprowadzane z klawiatury, zamieniał wszystkie litery na małe. Program był zabezpieczony przed wprowadzaniem niepoprawnych danych. Ciąg znaków z bufora był przetwarzany do momentu napotkania znaku nowej linii.

```
SYSCALL32 = 0x80

EXIT = 1
WRITE = 4
READ = 3
STDOUT = 1
MASK = 0x20  #maska zamieniajaca kazda litere na wielka 0010 0000

.data
bufor: .space BUFOR_SIZE #
bufor_len = . - bufor

.text
.global _start

_start:

 movl $bufor_len, %edx #wczytanie
 movl $bufor, %ecx
 movl $STDOUT, %ebx
 movl $READ, %eax
```

```
 int $SYSCALL32
 #początek przetwarzenia
 xorl %edi, %edi  #inicjalizacja wskaznika

loop:

 movb bufor(,%edi,1), %al #skopiowanie  znaku z bufora do rejestru

 cmpb $'\n',%al  #sprawdzamy, czy przetworzono juz cala linie
 je print

 orb $MASK, %al  #zamiana wszystkich liter na male

 cmpb $'a', %al  #sprawdzenie, czy znak jest literą
 jl break
 cmpb $'z', %al
 jg break

 movb %al,bufor(,%edi,1)#skopiowanie znaku do bufora

 break:
 incl %edi #inkrementacja wskaznika
 jmp loop

 print: #wyswietlenie
 movl $bufor_len, %edx
 movl $bufor, %ecx
 movl $STDOUT, %ebx
 movl $WRITE, %eax

 int $SYSCALL32

 movl $EXIT, %eax
 int $SYSCALL32
```

Program został skompilowany i uruchomiony przy pomocy poniższego polecenia. Zaobserwowano poprawne działanie.

```
as cw1.s
```

2.2 Szyfr Cezara

Kolejnym zadaniem było zaimplementowanie algorytmu szyfrowania szyfrem Cezara. Kluczem był pierwszy znak wczytany z klawiatury. Jeżeli wczytany klucz był wielką literą algorytm szyfrował napis, w przeciwnym przypadku następowało deszyfrowanie.

```
SYSCALL32 = 0x80
```

```
EXIT = 1
WRITE = 4
READ = 3
STDOUT = 1
MASK = 0x20 #maska zamieniajaca rozmiar liter

.data
bufor: .space BUFOR_SIZE
bufor_len = . - bufor

.text
.global _start
#male deszyfrowanie
#wielkie szyfrowanie

_start:

#wczytanie
movl $bufor_len, %edx
movl $bufor, %ecx
movl $STDOUT, %ebx
movl $READ, %eax

int $SYSCALL32

xorl %edi, %edi  #inicjalizacja wskaznika
movb bufor(,%edi,1), %bl  #wczytanie parametru szyfru
incl %edi

or $0x40, %bl  #duze kody mają kody 0x41,0x42....
cmpb $'Z',%bl #jezeli wielka to szyfrujemy
jbe prepare_encrypt
prepare_decrypt: #w przeciwnym przypadku deszyfrujemy
subb $'a', %bl
negb %bl
jmp cipher


prepare_encrypt:
subb $'A', %bl
#movb $' ',bufor(,$0,1)  #ukrycie klucza

cipher:
movb bufor(,%edi,1), %al #pobranie znaku z bufora
cmpb $'\n',%al #sprawdzenie, czy nie nastąpił znak końca nowej linii
je out
```

```
orb $MASK, %al  #zamiana wszystkich liter na male
cmpb $'a', %al  #sprawdzenie, czy znak jest literą
jl hop
cmpb $'z', %al
jg hop

addb %bl, %al #szyfrowanie

cmpb $'z',%al   #koreka szyfru
jle hop
subb $'z', %al
addb $'a', %al

hop:
movb %al,bufor(,%edi,1) #wpisanie przetworzonego napisu do bufora
incl %edi #zwiększenie wskaźnika
jmp cipher

out:
movl $bufor_len, %edx #wyswietlenie
movl $bufor, %ecx
movl $STDOUT, %ebx
movl $WRITE, %eax

int $SYSCALL32

movl $EXIT, %eax
int $SYSCALL32
```

Program wczytuje pierwszy znak i rozpoznaje jej rozmiar. Na tej podstawie obliczony jest klucz, który jest ujemny dla deszyfrowania i dodatni dla szyfrowania. Następnie algorytm przetwarza kolejne znaki wczytane z bufora. Po skopiowaniu pojedynczej litery dodawany jest do niej klucz. Następnie szyfr jest korygowany. Jeżeli po dodaniu, lub odjęciu klucza wartość znajdzie się poza zakresem alfabetu następuje odpowiednia poprawka. Program został uruchomiony przy pomocy poniższych poleceń. Po przetestowaniu stwierdzono poprawne działanie.

```
as cezar.s -o cezar.o
ld cezar.o -o cezar
./cezar
```

2.3 Konwersje liczbowe

2.3.1 Konwersja wejściowa

Kolejnym etapem było przygotowanie programów wczytujących napisy w postaci cyfr i zamieniających je na liczby. Przygotowano uniwersalną funkcję wczytującą ciąg cyfr do bufora,zamieniającą ją na liczbę o podstawie danej stałą zdefiniowaną w programie. W celu zachowania czytelności pominięto powtarzające się elementy programu.

Funkcja jest w stanie dekodować liczby zapisane w systemie o podstawie maksymalnie równej 16. Algorytm jest oparty na schemacie Hornera.

```
...
BASE = 16

.text
.global _start
_start:

#wczytanie
...

mov $0,%edx #wynik
mov $0,%edi #wskaznik
mov $0,%ecx #rejestr do przechowywania aktualnej cyfry

L1:
movb bufor(,%edi,1), %cl

cmpb $'\n',%cl #sprawdzenie, czy nie nastąpił koniec bufora
jne L2
ret
L2:
mov $BASE,%eax #mnozenie kolejnych cyfr
mul %edx
mov %eax,%edx
subb $'0',%cl
add %ecx,%edx

orb $MASK,%cl #zamiana liter na małe
cmp $'a',%cl #sprawdzenie, czy przetwarzana jest litera a-f
jb is_digit
cmp $'f',%cl
ja is_digit
subb $'a',%cl #odjęcie w celu uzyskania cyfry jednosci
add $10, %cl  #dodanie w celu uzyskani cyfry dziesiętnej
jmp hop

is_digit:
subb $'0', %cl #jezeli przetwarzana cyfra, odejmuje kod znaku '0'

hop:
add %ecx,%edx

inc %edi
jmp L1
```

2.3.2 Konwersja wyjściowa

Poniżej przedstawiono fragment programu przetwarzającego liczbę przechowywaną w rejestrze *%eax* na ciąg znaków. Funkcja jest uniwersalna i może wypisywać znaki w systemie o dowolnej podstawie. Algorytm dzieli liczbę przez bazę systemu liczbowego, następnie resztę zamienia na znak danej cyfry i wynik zapisuje w buforze wyjściowym.

```
processOutputNumber:

mov $BASE_OUT,%ebx #podstawa systemu zapisana w programie
mov $outputBuforLen-2, %ecx
L3:
mov $0, %edx #wyzerowanie rejestru danej cyfry
div %ebx #dzielenie pozostałej liczby
addb $'0', %dl #zamiana cyfry na kod znaku a ASCII(reszta z dzielenia)
movb %dl,outputBufor(,%ecx,1)#skopiowanie do bufora
dec %ecx #zmniejszenie wskaznika
cmp $0,%eax #sprawdzenie warunku zakończenia algorytmu
jne L3
ret
```

Powyższe programy były kompilowane i uruchomiane następującymi poleceniami:

```
as z1.s -o z1.o
ld z1.o -o z1
./z1
```

3 Wnioski

Ćwiczenie pozwoliło na zaznajomienie z instrukcjami i składnią asemblera. Początkowo problematyczne było odwoływanie się do bufora, często spotykałem się z błędem *Segmentation fault*.

(2)Utrwalenie wiadomości, użycie stosu oraz innych instrukcji w języku Assembler na platformie Linux

1 Treść ćwiczenia

Zakres ćwiczenia:

- Tworzenie programów wykorzystujących funkcje. Konstrukcja algorytmów rekurencyjnych.

2 Przebieg ćwiczenia

2.1 Funkcja obliczeniowa

Pierwszym zadaniem było zaimplementowanie prostej funkcji obliczeniowej. W tym celu wykorzystano przygotowane na poprzednim laboratorium fragmenty kodu odpowiedzialne za wczytywanie i wypisywanie liczb. Funkcja *multiplyTwoNumbers* pobiera argumenty ze stosu, mnoży przez siebie, a wynik również odkłada na stosie.

```
SYSCALL32 = 0x80

EXIT = 1
WRITE = 4
READ = 3
STDOUT = 1
BUFOR_SIZE = 10
BASE = 10
.data
bufor: .space BUFOR_SIZE
bufor_len = . - bufor

outputBufor: .ascii "wynik:          \n"
outputBuforLen=.-outputBufor
.text
.global _start

_start:

call processInputNumber #wczytaj i przetworz pierwsza liczbe
push %rdx #odłóż wczytaną liczbę na stosie
call processInputNumber
push %rdx
call multiplyTwoNumbers #wywołanie funckji mnozacej dwie liczby
pop %rbx
mov %rbx,%rax
call processOutputNumber
```

```
jmp out

multiplyTwoNumbers:

push %rbp #
mov %rsp,%rbp

mov 16(%rbp),%rax #pobranie pierwszej liczby
mov 24(%rbp),%rdx #pobranie drugiej liczby

  mul %rdx #pomnozenie - wynik w rax

mov %rax,16(%rbp) #zapisanie wyniku dzialania funkcji
mov %rbp,%rsp  #przywrocenie wskaznika stosu szczytu
pop %rbp

ret

processInputNumber: #wczyta i przetworzy napis podany na wejsciu na liczbe dziesiet

#wczytanie
movl $bufor_len, %edx
movl $bufor, %ecx
movl $STDOUT, %ebx
movl $READ, %eax

int $SYSCALL32


mov $0,%edx #wynik
mov $0,%edi
mov $0,%ecx

L1:
movb bufor(,%edi,1), %cl

cmpb $'\n',%cl
jne L2
break1: ret
L2:
mov $BASE,%eax
mul %edx
mov %eax,%edx
subb $'0',%cl
add %ecx,%edx

inc %edi
jmp L1
```

```
processOutputNumber: #przetworzy dane z eax na ciag łańcuchów ascii i wynik przecho

mov $BASE,%ebx
mov $outputBuforLen-2, %ecx
L3:
mov $0, %edx
div %ebx
addb $'0', %dl
movb %dl,outputBufor(,%ecx,1)
dec %ecx
cmp $0,%eax
jne L3
ret

out:
#wyswietlenie
movl $outputBuforLen, %edx
movl $outputBufor, %ecx
movl $STDOUT, %ebx
movl $WRITE, %eax

int $SYSCALL32

movl $EXIT, %eax
int $SYSCALL32
```

Powyższy program został skompilowany i uruchomiony następującymi poleceniami:

```
as z1.s -o z1.o
ld z1.o -o z1
./z1
```

2.2 Obliczanie silni

W celu zachowania czytelności pominięto pewne fragmenty, które zostały przedstawione wcześniej w tym sprawozdaniu.

```
#...
############################################################
.text
.global _start
_start:
call processInputNumber #wczytaj i przetworz pierwsza liczbe
push %rdx
call factorial
  call processOutputNumber
jmp out
```

```
############################################################

factorial:
push %rbp
mov %rsp, %rbp  # nowy wskaznik ramki
mov 16(%rbp), %rax  # pobranie parametru z wnetrza stosu
cmp $1, %rax # Sprawdzenie warunku zatrzymania
je factorial_end

dec %rax # zmniejsz licznik poziomu iteracji
push %rax
call factorial # wywolaj funkcje rekurencyjnie

mov 16(%rbp), %rbx #pobierz aktualny poziom rekurencji
mul %rbx #oblicz iloczyn poziomu rekurencji


factorial_end:
mov %rbp, %rsp # przywroc wskaznik stosu
pop %rbp  #
ret  # Return
```

3 Wnioski

Poprawne operowanie danymi na stosie jest zadaniem nietrywialnym. Początkowo problem sprawiło prawidłowe wyrównywanie stosu. Dopiero dogłębne przeanalizowanie materiałów dotyczących odpowiedniego tworzenia i wykorzystania funckji pozwoliło na zrealizowanie ćwiczenia.

(3) Zapoznanie z technikami pozwalającymi na użycie w tym samym projekcie różnych języków programowania

1 Treść ćwiczenia

Zakres ćwiczenia:

- Użycie języka C w kodzie Assemblera oraz, użycie języka Assemblera w kodzie C. Tworzenie wstawek asemblerowych.

2 Przebieg ćwiczenia

2.1 Wykorzystanie funkcji bibliotecznych języka C w asemblerze

Pierwszym programem zaimplementowanym na laboratorium była aplikacja wykorzystująca funkcję z biblioteki standardowej języka C, mianowicie *printf* i *scanf*. Podczas używania funkcji *scanf* należy zadeklarować łańcuch formatujący i przekazać go funkcji poprzez stos. Program wczytuje liczbę podana przez użytkownika, następnie wyświetla napis złożony z danych wpisanych w programie, oraz z liczby wprowadzonej przez użytkownikiem

```
.data
.align 32
format_string_input:
.string "%d"

format_string:
.ascii "Przykladowy string,Czesc  %s!, Jestem %s i mam   %d lata \n\0"

text1:
.ascii "Krzysiu\0"   # pierwszy parameter %s (łańcuch znaków zakończony \0)
text2:
.ascii "Rafal\0"   # drugi parameter %s (łańcuch znaków zakończony \0)
number:
.long 0           # trzeci parameter %d (liczba dziesiętna)

.global main
.text

main:

push $number
pushl $format_string_input       #łańcuch formatujący
call scanf
```

```
                            # parametery przez stos w odwróconej kolejności
push number             # trzeci liczba dziesiętna (%d)
pushl $text2                # drugi łańcuch (%s)
pushl $text1                # pierwszy łańcuch (%s)
pushl $format_string        #łańcuch formatujący
call printf

call exit                   # funkcja zakończenia programu
```

Powyższy program został skompilowany następującym poleceniem:

```
gcc zad1.s -m32
```

2.2 Użycie funkcji napisanej w asemblerze w programie w języku C

W asemblerze stworzono funkcję przedstawioną na poniższym listingu. Pobiera ona dwa parametry przekazane poprzez stos, następnie dodaje je i wyświetla.

```
SYSCALL32 = 0x80
.data
format_string:
.ascii "Wynik to:   %d \n\0"
.text
.global output

output:
push %ebp
mov %esp, %ebp
mov 8(%ebp), %eax
mov 12(%ebp), %ebx

add %eax, %ebx

push %ebx
push $format_string
call printf

mov %ebp, %esp # przywroc wskaznik stosu
pop %ebp  #

ret
```

Funkcja została wywołana w kodzie napisanym w języku C. Kod został przedstawiony poniżej.

```
#include <stdio.h>

void output(int a, int b);
```

```
int main()
{
    output(2,3);
    return 0;
}
```

Powyższym program został skompilowany poprzez poniższy Makefile:

```
# reguła linkowania
zad2: output.o zad2.o
    ld output.o zad2.o -o zad2
# reguła kompilacji
output.o: output.s
    as output.s -o output.o
zad2.o: zad2.c
    gcc zad2.c -o zad2.o
clean:
    rm -f *.o
run:
    ./zad2
```

Program skompilowano i uruchomiono. Zaobserwowano prawidłowe działanie. Początkowo wystąpiły trudności w prawidłowym stworzeniem funkcji w asemblerze, tak by mogła odbierać parametry poprzez stos. Było to spowodowane nieprawidłowym wyliczaniem adresu przekazywanych parametrów na stosie. Ponadto należało usunąć etykietę *main* z pliku asemblerowego, prowadziało to do niejednoznaczności.

2.3 Użycie funkcji napisanej w C w programie asemblera

Program ma działanie analogiczne do poprzedniego. Tym razem asembler jest odpowiedzialny za wczytanie dwóch liczb, przekazanie ich jako parametrów oraz wywołanie funkcji wykonującej dodawanie i wyświetlenie wyniku. W tym przypadku etykieta *main* musi znajdować się z programie asemblerowym.

```
.data
.align 32
format_string_input:
.string "%d %d"
number1:
    .long 0            # jeden parameter %d (liczba dziesiętna)
number2:
    .long 0            # drugi parameter %d (liczba dziesiętna)

.global main
.text
main:

push $number1
push $number2
```

```
pushl $format_string_input       #łańcuch formatujący
call scanf

push number1
push number2
call dodaj

call exit                   # funkcja zakończenia programu
```

Poniżej przedstawiono kod programu w języku C, który jest wywoływany w programie asemblerowym. Pobiera on dwa argumenty, będące liczbami całkowitymi, następnie dodaje je i wypisuje.

```
#include <stdio.h>
void dodaj(int a, int b){
printf("%d", a+b);
}
```

Program został skompilowany poleceniem:

```
gcc zad2.c add.s -m32
```

2.4 Wstawka asemblerowa

Zaimplementowano wstawkę dodającą dwie liczby.

```
#include <stdio.h>

int main()
{
int a = 3;
int b = 5;
int result = 0;
__asm__(
"mov %1, %%ecx;\n"
        "mov %2, %%ebx;\n"
"add %%ecx,%%ebx\n"
"mov %%ebx,%0\n"
         : "=b" ( result )        /* output */ parametry wyjsciowe
         : "c" ( a ), "b" (b)     /* input *///przekazywnie parametrow do wstawki
         : "%ecx" //informacja o niszczonych rejestrach
);

printf("%d",result);
return 0;
}
```

Kod został skompilowany poniższym poleceniem:

```
gcc zad3.c
```

3 Wnioski

Umiejętność łączenia programów napisanych w C i asemblerze jest praktyczna. Odpowiednie przekazywanie parametrów poprzez stos wymaga dogłębnego zrozumienia prawidłowej obsługi zmiennych odkładanych na stosie.

(4)Zapoznanie z jednostką zmiennoprzecinkową procesorów rodziny x86

1 Treść ćwiczenia

Zakres ćwiczenia:

- Zapoznanie się z działaniem jednostki zmiennoprzecinkowej, jej kodami błędów oraz możliwymi działaniami które można na niej wykonać..

2 Przebieg ćwiczenia

3 Wnioski

1 Treść ćwiczenia

Zakres ćwiczenia:

- Zapoznanie się z podstawami pisania, linkowania i uruchamiania programów w języku assembler.

2 Przebieg ćwiczenia

W celu uruchomienia powyższego programu należało skompilować, a następnie skonsolidować przy pomocy poniższych instrukcji.

```
#kompilacja
as lab.s -o lab.o

#konsolidacja
ld lab.o -o lab

#uruchomienie
./lab
```

2.1 Wczytywanie znaków z klawiatury do bufora

Program miał zapisać znaki wprowadzone przez użytkownika do bufora o nazwie `bufor`.

```
%kod
```

2.2 Operacje na znakach w buforze

Powyższy fragment programu przygotowuje do wykonania i wykonuje funkcję systemową `SYSWRITE`. Użyta na początku dyrektywa `.global` ma za zadanie zdefiniowanie punktu wejścia do programu, czyli poinformowanie systemowego programu ładującego o punkcie, w którym powinno zacząć się jego wykonanie.

Jak widać z powyższego fragmentu kodu, wywołanie funkcji systemowych na platformie *Linux/x86* polega na

1. umieszczeniu numeru funkcji w rejestrze procesora `eax`,
2. umieszczeniu jej argumentów w rejestrach `ebx`, `ecx`, `...`,
3. wykonanie przerwania programowego `0x80`.

Należy tu zauważyć, że składnia poszczególnych instrukcji i trybów adresowania jest odmienna od tej stosowanej na platformie *DOS/WINDOWS* – np. kolejność argumentów instrukcji jest odwrotna (od lewej do prawej).

Ostatni krok, który należy wykonać aby dokończyć program, to wywołanie funkcji systemowej `SYSEXIT` z argumentem `EXIT_SUCCESS`. Operacja ta jest przygotowywana analogicznie.

```
[kberezow@localhost src]$ gdb hello
GNU gdb 5.3-22mdk (Mandrake Linux)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i586-mandrake-linux-gnu"...
(gdb) run
Starting program: /home/kberezow/projects/AK/src/hello
Hello, world!

Program exited normally.
(gdb) quit
[kberezow@localhost src]$
```

Rysunek 1: Przebieg najkrótszej sesji uruchomieniowej dla programu `hello` w środowisku `gdb`.

```
mov $SYSEXIT, %eax      ; funkcja do wywołania - SYSEXIT
mov $EXIT_SUCCESS, %ebx ; 1 arg. -- kod wyjścia z programu
int $0x80               ; wywołanie przerwania programowego -
                        ; wykonanie funcji systemowej.
```

2.3 Uruchomienie programu pod kontrolą `gdb`

Dzięki przygotowanemu plikowi `Makefile` proces kompilacji programu sprowadzał się do wydania powłoce polecenia `make`.

```
[kberezow@localhost src]$ make
as -o hello.o hello.s
ld -o hello hello.o
[kberezow@localhost src]$
```

Uruchomienie go pod kontrolą programu `gdb` następuje poprzez wydanie polecenia `gdb hello`. Przykład kompletnej sesji z programem `gdb` uwidoczniony jest na rys. 1.

Pod kontrolą programu `gdb` można uruchomić program w trybie pracy krokowej bądź ciągłej oraz podejrzeć zarówno tekst programu jak i wartości zawarte w pamięci i rejestrach procesora. W przypadku naszego pierwszego programu możemy uzyskać jedynie wtórnie rozkodowany tekst programu, gdyż nie umieściliśmy żadnej informacji uruchomieniowej w jego treści.

```
(gdb) disassemble _start
Dump of assembler code for function _start:
0x804808e <_start>:     mov    $0x4,%eax
```

```
0x8048093 <_start+5>:   mov    $0x1,%ebx
0x8048098 <_start+10>:  mov    $0x8048080,%ecx
0x804809d <_start+15>:  mov    $0xe,%edx
0x80480a2 <_start+20>:  int    $0x80
0x80480a4 <_start+22>:  mov    $0x1,%eax
0x80480a9 <_start+27>:  mov    $0x0,%ebx
0x80480ae <_start+32>:  int    $0x80
End of assembler dump.
(gdb)
```

Jak widać wyświetlony program odpowiada (z dokładnością do wyliczonych wartości etykiet) zawartości pliku źródłowego.

Program uruchomieniowy możemy również wykorzystać do podejrzenia zawartości pamięci, np. używając polecenia `p[rint]`[1].

```
(gdb) print {char}0x8048080
$6 = 72 'H'
(gdb) print {char}(0x8048080+1)
$8 = 101 'e'
(gdb) print {char}(0x8048080+2)
$9 = 108 'l'
(gdb) print {char}(0x8048080+3)
$10 = 108 'l'
(gdb) print {char}(0x8048080+4)
$11 = 111 'o'
(gdb) print {char}(0x8048080+5)
$12 = 44 ','
(gdb) print {char}(0x8048080+6)
$13 = 32 ' '
(gdb) print {char}(0x8048080+7)
$14 = 119 'w'
(gdb) print {char}(0x8048080+8)
$15 = 111 'o'
(gdb) print {char}(0x8048080+9)
$16 = 114 'r'
(gdb) print {char}(0x8048080+10)
$18 = 108 'l'
(gdb) print {char}(0x8048080+11)
$19 = 100 'd'
(gdb) print {char}(0x8048080+12)
$20 = 33 '!'
(gdb) print {char}(0x8048080+13)
$21 = 10 '\n'
(gdb)
```

[1]Program `gdb` umożliwia skracanie nazw poleceń, jeżeli powstały w wyniku tego procesu skrót jest jednoznaczny w obrębie zbioru jego poleceń. Aby odzwierciadlić tę własność, zachowując równocześnie przejrzystość treści sprawozdania, będziemy wyróżniać część polecenia, którą można bezpiecznie odrzucić stosując nawiasy kwadratowe `[]`.

Ze względu na to, że w pakiecie kompilatorów `GNU gcc` informację pomocniczą dla programu uruchomieniowego kompilatory umieszczają na poziomie kodu asemblerowego, aby przeprowadzić bardziej zaawansowane operacje śledzenia przebiegu wykonania programu, należy skompilować nasz program kompilatorem języka C, w trybie z uzupełnianiem informacji uruchomieniowej (opcja kompilatora `-g`). Oznacza to, że aby precyzyjnie śledzić przebieg programów aseblerowych musieliśmy zmienić nasz plik sterujący `Makefile` np. do następującej postaci:

```
# reguła kompilacji i linkowania
hello: hello.s
  gcc -g -o hello hello.s
```

a punkt wejścia do programu zmienić z `.global _start` do `.global main` tak, aby zapewnić poprawność łączenia z programem startowym języka C.

Dzięki takim zmianom mogliśmy zastosować krokowe śledzenie przebiegu naszego programu, korzystając z następujących poleceń programu `gdb` [6]:

- `b[reak]` – ustawienie pułapki,
- `r[un]` – uruchomienie programu,
- `s[tep], n[ext]` – wykonanie kroku w pracy krokowej (omówienie różnic pomiędzy poleceniami `s` i `n` można znaleźć się w [6]),
- `inf[o]` – wyświetlenie informacji o wykonywanym programie (np. `inf[o] reg[isters]`).

Przebieg typowej sesji pracy krokowej, wykazujący nabyte umiejętności pracy z programem uruchomieniowym, widoczny jest poniżej.

```
(gdb) b main
Breakpoint 1 at 0x804832e: file hello.s, line 20.
(gdb) run
Starting program: /home/kberezow/projects/AK/src/hello
1
Breakpoint 1, main () at hello.s:20
20        mov $SYSWRITE, %eax
Current language:  auto; currently asm
(gdb) list
15      msg_hello_len = . - msg_hello
16
17      .global main
18
19      main:
20        mov $SYSWRITE, %eax
21        mov $STDOUT, %ebx
22        mov $msg_hello, %ecx
23        mov $msg_hello_len, %edx
```

```
24        int $0x80
(gdb) info registers
eax            0x1      1
ecx            0x401517c8       1075124168
edx            0x0      0
ebx            0x40153f50       1075134288
esp            0xbffff50c       0xbffff50c
ebp            0xbffff528       0xbffff528
esi            0x40012780       1073817472
edi            0xbffff554       -1073744556
eip            0x804832e        0x804832e
eflags         0x246    582
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x0      0
fctrl          0x37f    895
fstat          0x0      0
ftag           0xffff   65535
fiseg          0x0      0
fioff          0x0      0
foseg          0x0      0
fooff          0x0      0
fop            0x0      0
mxcsr          0x0      0
orig_eax       0xffffffff       -1
(gdb) s
21        mov $STDOUT, %ebx
(gdb)
22        mov $msg_hello, %ecx
(gdb)
23        mov $msg_hello_len, %edx
(gdb)
24        int $0x80
(gdb)
Hello, world!
27        mov $EXIT_SUCCESS, %ebx
(gdb)
```

3 Podsumowanie i wnioski

Wykonane ćwiczenie pozwoliło nam opanować umiejętności tworzenia i uruchamiania programów asemblerowych na platformie *Linux/x86*. Jego wykonanie uświadomiło nam, iż pomiędzy platformą *Linux/x86*, a znaną nam wcześniej platformą *DOS/WINDOWS* istnieją zarówno podobieństwa jak i różnice.

Pierwszą zaobserwowaną i dotkliwą dla nas różnicą jest odmienna składnia asemblera, niezgodna ze składnią propagowaną przez firmę *INTEL*. Dla programisty, który nabył już pewnych doświadczeń w tworzeniu programów asemblerowych dla systemów *DOS/WINDOWS*, szczególnie niebezpieczna może być odwrotna kolejność argumentów instrukcji asemblera w składni AT&T stosowanej w asemblerze `GNU as`, gdyż może prowadzić do generowania trudnych do wykrycia błędów semantycznych. Drugą istotną różnicą, jest brak możliwości wygenerowania informacji dla programu uruchomieniowego w procesie asemblacji programów, która zmusza do uruchamiania programów asemblerowych w środowisku kompilatora języka C bądź C++. To wszystko powoduje, iż zmiana platformy z *DOS/WINDOWS* na *Linux/x86* raczej nie będzie intuicyjna dla programisty asemblera.

Pomimo występowania różnic, na bazie stworzonego programu, możemy również zauważyć, że występują istotne podobieństwa w sposobie komunikacji pomiędzy programem użytkowym a systemem. Zarówno w systemie *DOS* jak i *Linux* stosuje się instrukcję przerwania programowego. Zwraca uwagę uporządkowany sposób przekazywania argumentów do funkcji systemowych, który, w przeciwieństwie do systemu *DOS*, jest bardzo intuicyjny.

Odmienny, w stosunku do znanego z systemu *DOS* programu *Turbo Debugger* (`td.exe`), jest również sposób pracy z programem uruchomieniowym – choć zestaw dostępnych poleceń służących do analizy programów jest w dużej mierze podobny, a być może nawet szerszy, to praca z tym narzędziem wymaga od programisty większej wiedzy o nim samym. Przede wszystkim, w programie `gdb`, programista jest zmuszony do tego, aby wszystkie informacje związane z przebiegiem uruchamianego programu uzyskiwać wydając właściwe polecenia – w zasadzie żadna informacja, poza całkowicie podstawową, nie jest dostarczana przez program samoczynnie. Jest to związane z tym, że program `gdb` jest przystosowany do pracy z różnymi interfejsami użytkownika, a także może być osadzany w środowisku innych programów [6]. Przykładem mogą być tu programy dostarczające rozbudowany interfejs graficzny dla różnych programów uruchomieniowych, jak *Data Display Debugger* (`ddd`), czy możliwość uruchamiania i analizy programów bezpośrednio w środowisku edytora *EMACS* zademonstrowana na rys. **??**.

Literatura

[1] J. Biernat, *Profesjonalne przygotowanie publikacji*, materiały konferencyjne X Krajowej Konferencji KOWBAN, str. 401–408, Wyd. WTN, Wrocław, 2003.

[2] M. Węgrzyn, *Zastosowanie pakietu MS Word do przygotowania publikacji naukowych*, materiały konferencyjne X Krajowej Konferencji KOWBAN, str. 409–414, Wyd. WTN, Wrocław, 2003.

[3] E. Polański i inni, Nowy słownik ortograficzny PWN z zasadami pisowni i interpunkcji, PWN, Warszawa, 1997

[4] `/usr/include/asm/unistd.h`, plik nagłówkowy kompilatora `gcc` z listą kodów funkcji systemowych systemu *Linux*.

[5] `info gas`, dokumentacja `info` asemblera `GNU as`.

[6] `info gdb`, dokumentacja `info` programu `GNU gdb`.

[7] `info ld`, dokumentacja `info` programu `GNU ld`.

[8] `info make`, dokumentacja `info` programu `GNU make`.

[9] `http://www.linuxassembly.org`, witryna internetowa z informacjami dla programistów asemblera dla platformy *Linux/x86*.

A Wytyczne i uwagi dotyczące dokumentu sprawozdania

A.1 Uwagi ogólne

Zgodnie z ogłoszonymi wcześniej założeniami dotyczącymi sprawozdawczości, sprawozdania nie zawierają wstępów teoretycznych czy omówienia użytych narzędzi w zakresie szerszym niż to wynika z tematu ćwiczenia (*vide*: brak omówienia programu `make` i formatu plików `Makefile` w niniejszym dokumencie).

Sprawozdanie powinno być strony formy wypowiedzi zbliżone do rozprawki. Tym bardziej uczulam Państwa na treści zawarte w podrozdziale 3. Zawarty tam tekst powinien między innymi w syntetyczny sposób posumować nabytą wiedzę. Poza tym, jest tam miejsce na Państwa przemyślenia i obserwacje. Ostrzegam, że ta część sprawozdania – niewłaściwie napisana czy niestarannie przemyślana – może Państwa kosztować najwięcej punktów. Nie oznacza to jednak, że brak sformułowania tez (celu ćwiczenia) we wprowadzeniu ujdzie Państwu na sucho!

A.2 Wymagania składu

Układ strony i akapitów powinien zachowywać następujące własności:

- krój czcionki podstawowej – *Times*, 12pt (*Computer Modern*, 12pt),
- interlinia – 1.5,
- marginesy – lewy, prawy, górny, dolny – 2,5 cm,
- nagłówki numerowane (preferowane cyfry arabskie),
- strony numerowane (preferowane cyfry arabskie) od strony tytułowej, jednakże bez umieszczania numeru na stronie tytułowej,
- wyrównanie tekstu obustronne, dzielenie wyrazów opcjonalne,
- rysunki i tabele opatrzone numerem i tytułem, umieszczane na górze bądź na dole strony (nie w ciągu akapitu),
- odniesienia do rysunków/tabel w tekscie poprzez podanie numeru rysunku,
- cytaty z kodu źródłowego umieszczane w ciągu akapitu lub jako rysunki,
- strona tytułowa zgodna z dostarczonym wzorcem.

Proszę o wprowadzanie i stosowanie konwencji typograficznych związanych z przytaczaniem nazw własnych, symbolicznych, cytowaniem kodu czy wyjścia programów. Konwencje takie przyjmują Państwo i pielęgnują na własną rękę, jednakże konsekwencja ich przestrzegania podlega ocenie.

Szczęśliwi użytkownicy systemu LaTeX 2$_\varepsilon$ mogą posłużyć się plikiem źródłowym niniejszego dokumentu jako wzorcem do konstrukcji własnych sprawozdań i zapomnieć o znakomitej większości problemów składu. Użytkownicy edytora *MS Word* powinni na własną rękę przygotować właściwy szablon dokumentu. Tym ostatnim polecam lekturę artykułów [1, 2], których wersje elektroniczne być może posiada prof. J. Biernat, a do wglądu i odbicia dostępne są również u mnie.

A.3 Język

Przypominam, że językiem urzędowym w naszym kraju jest język polski co oznacza, że należy przestrzegać właściwych mu zasad [3]:

- interpunkcji (!),
- stylu (!!),
- gramatyki (!!!) oraz
- ortografii (!!!!).

Zwracam więc uwagę na to, że z powyższej listy zasad, narzędzia automatycznej korekty zadowalająco radzą sobie jedynie z ortografią.
UWAGA: pozostawianie jednoliterowych spójników (a, i, w, z, ...) na końcu wiersza, choć dopuszczone przez Radę Języka Polskiego, wciąż powszechnie uważa się za brak profesjonalizmu w przygotowywaniu dokumentów.

A.4 Informacje o autorach

Kod zespołu (umieszczany w lewym górnym rogu strony tytułowej) składa się z czterech elementów rozdzielonych znakiem `'/'`. Kolejne pola oznaczają:

1. `XX` – dzień tygodnia (wartości: `PN`, `WT`, `SR`, `CZ`, `PT`,
2. `Y` – „parzystość" tygodnia (wartości: `P`, `N`),
3. `ZZ` – godzina rozpoczęcia zajęć (wartości przykładowe: `08`, `11`, ...)
4. `T` – numer zespołu: wartości przykładowe: `1`, `2`, ...

Dwuelementowa lista autorów umieszczana pod kodem zespołu powinna być rosnąco posortowana ze względu na nazwiska (oczywiście stosujemy sortowanie leksykograficzne).

A.5 Podsumowanie

Przygotowanie całości tego dokumentu, wraz ze stworzeniem i uruchomieniem programu `,,Hello, world!''` oraz przygotowaniem niniejszego komentarza, zajęło mi około czterech godzin, więc nie jest to dla Państwa wysiłek ponad miarę!