Wrocław, dn. 2 marca 2017

Hanna Grodzicka, 226154

Podstawy uruchamiania programów asemblerowych na platformie Linux/x86

sprawozdanie z laboratorium przedmiotu „Architektura Komputerów 2”

Rok. akad. 2016/2017, kierunek: INF

Prowadzący:

mgr inż. Aleksandra Postawka

1. Cel ćwiczenia

   Celem ćwiczenia było napisanie pierwszego programu w języku asemblera na platformie *Linux/x86.* Program miał wypisywać „`Hello world!`” na monitorze. W tym celu konieczne było nabycie nowych umiejętności takich jak:

- korzystanie z terminala; poruszanie się po systemie za pomocą komend np. **man**, **cd**, **ls**, **rm**, **cat;**
- poznanie podstaw składni programów dla asemblera GNU;
- kompilacja, konsolidacja komendami **as**, **ld**, **gcc**;
- znajomość funkcji standardowych i metody ich wywoływania tj. **syscall**;
- zdalne logowanie się do systemów unixopodobnych przy pomocy **ssh**, **putty**, **winscp** itp.;
- obsługa debuggera.

  Dalsza część ćwiczenia wymagała modyfikacji programu, tak aby można było do niego wpisać i wypisać ciąg znaków. Kolejny krokiem w tej modyfikacji było dopisanie funkcji, która przy wypisywaniu wprowadzonych znaków zamienia wszystkie małe litery na duże, zaś duże na małe.

2. Przebieg ćwiczenia

   Na wprowadzeniu do zajęć mieliśmy uruchomić terminal i opanować:

- tworzenie katalogu: `mkdir kat1`
- poruszanie się po systemie: `cd kat1`, `cd ../kat2`, `cd ~`

- rozróżnianie ścieżek względnych i bezwzględnych
- tworzenie pliku: `touch plik`
- korzystanie z edytorów np. **nano**, **vim**, **mcedit**, **gedit**
- wyświetlanie informacji o komendach (manual): `man ls`
- usuwania plików i katalogów: `rm plik` i `rm -R katalog`
- kopiowanie plików i katalogów: `cp kat1/plik1 kat2/` i `cp -R kat1/ kat2/`
- wyświetlanie obecnej lokalizacji: `pwd`
- wyświetlanie id użytkownika: `whoami`
- wyświetlanie zawartości plików/łączenie zawartości plików w jeden plik: `cat plik1 plik2`
- wyświetlanie zawartości obecnej lub podanej po komendzie lokalizacji: `ls`
- przenoszenie plików między lokalizacjami: `mv a b`
- logowanie się na konto przez: `ssh XXXXXX@lak.iiar.pwr.wroc.pl`

Po wstępie teoretycznym dotyczącym rejestrów ogólnego przeznaczenia, mnemoników, niektórych sufiksów, podziału programu na sekcje i wywoływania funkcji systemowych w systemach 32b i 64b, utworzyliśmy plik `hello.s`, w którym napisaliśmy pierwszy program wypisujący na ekranie „`Hello world!`”.

1) Konstrukcja programu `hello.s`

```
.data
SYSREAD = 0
SYSWRITE = 1
SYSEXIT = 60
STDOUT = 1
STDIN = 0
EXIT_SUCCESS = 0
```

W sekcji `.data`, służącej do inicjalizowania danych lub stałych, zadeklarowaliśmy symboliczne nazwy używane później jako argumenty do wywołania funkcji systemowych. Potrzebne numery funkcji systemowych można odtworzyć na podstawie pliku nagłówkowego `asm/uni-std.h`.

```
buf: .ascii „Hello world!\n”
buf_len = .-buf
```

W tej samej sekcji zadeklarowaliśmy jeszcze `buf` i `buf_len`. Pierwsze z nich jest etykietą, która wskazuje na napis, który ma zostać wypisany. W dalszej części programu napis będzie wyprowadzony na strumień wyjściowy przez funkcję systemową `SYSWRITE`, która przyjmuje trzy argumenty: liczbę określającą strumień wyjściowy, wskaźnik na miejsce, od którego zaczyna się wypisanie oraz długość napisu. Aby uzyskać ostatni argument stworzona została zmienna `buf_len`, przy deklaracji której „`.`” oznacza bieżącą lokalizację (tj. koniec ostatni znak z napisu), a `buf` to wskaźnik na początek ciągu znaków. Ich różnica daje długość łańcucha znaków, który zostaje zapisany do `buf_len`.

```
.text
.globl _start
```

Sekcja `.text` zawiera wykonywany kod. Dyrektywa `.globl` wyznacza etykietę, od której kod będzie wykonywany - w naszym przypadku jest to `_start`. Etykieta ta jest dostępna też poza plikiem (nie jest prywatna), co jest przydatne, kiedy chcemy połączyć kilka plików razem.

```
_start:

movq $SYSWRITE, %rax
movq $STDOUT, %rdi
movq $buf, %rsi
movq $buf_len, %rdx
syscall
```

Powyższy kod jest wykonaniem funkcji systemowej `SYSWRITE`. Do rejestru `rax` wpisywana jest funkcja, która ma zostać wykonana, zaś do rejestrów `rdi`, `rsi` i `rdx` argumenty tej funkcji (odpowiednia kolejność argumentów musi być zachowana).

Po `syscall` (przerwanie programowe) funkcja zostaje wykonana.
Mnemonik `mov` służy do kopiowania danych z jednej lokalizacji (po lewej - *source*), do drugiej (po prawej - *destination*). Dodawane są do niej sufiksy: `b`, `w`, `l`, `q`, które określają dla jak dużych dwóch liczb instrukcja się odbywa. Podane sufiksy odnoszą się odpowiednio do liczb: 8-bitowych, 16-bitowych, 32-bitowych, 64-bitowych.
Znak `$` oznacza, że podany operand ma być odczytywany jako wartość, a znak `%` jest stawiany przed nazwami rejestrów.

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

Funkcja systemowa `SYSEXIT` to ostatnia część programu. Przyjmuje ona tylko jeden argument (`EXIT_SUCCESS`), który jest kodem wyjścia programu. Jeśli ten argument jest równy `0` (tak jak określiliśmy na początku pisania programu w sekcji `.data`), to oznacza poprawne wykonanie się programu.

2) Wywołanie napisanego programu
W celu wykonania programu należy wpisać w terminal odpowiednie komendy:

```
as -o hello.o hello.s
```

W tym kroku instrukcje z pliku `hello.s` (*source file*) są transformowane na język maszynowy (*assembling*). Tłumaczenie to zapisywane jest do pliku `hello.o` (*object file*), który się tworzy lub nadpisuje po wykonaniu tej komendy. Ta komenda nie wystarczy do złożenia całego programu. W większych programach, zawierających kilka plików z kodem źródłowym, dla każdego z nich tworzony jest *object file*.

```
ld -o hello hello.o
```

Następnym krokiem jest złączenie wszystkich przetłumaczonych plików (*linking*) i dodanie odpowiednich informacji, aby kernel wiedział jak je załadować i

uruchomić.

3) Utworzenie pliku `Makefile`

   Aby uniknąć potrzeby każdorazowego wywoływania po sobie dwóch komend opisanych w poprzednim punkcie (kompilacji i konsolidacji), używany jest plik `Makefile`. Jest to plik bez rozszerzenia, wywoływany przy pomocy komendy `make`. Dla naszego programu plik wyglądał następująco:

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

   Komendy zapisujemy w odwrotnej kolejności niż wykonywaliśmy w punkcie poprzednim. Konieczne jest dodanie etykiet, których nazwy są nazwami plików, które otrzymujemy po wykonaniu ich zawartości. Po nazwie etykiety powinien pojawić się tabulator, a za nim nazwa pliku z rozszerzeniem, który jest naszym źródłem. Przed wpisaniem komendy też powinien pojawić się tabulator - inaczej przy wywołaniu `Makefile`, po komendzie `make` dostaniemy błąd.

4) Modyfikacja programu na program wczytaj-wypisz z zamianą wielkości liter

```
BUFLEN = 512
```

   Dla modyfikacji programu w sekcji `.data` dodaliśmy stałą `BUFLEN`, do której przypisujemy wartość 512.

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

   W sekcji `.bss` przy pomocy dyrektywy `.comm` zarezerwowaliśmy dwa obszary pamięci (bufory) `textin` oraz `textout`, oba na 512 bajtów. Taka rezerwacja nie wymaga inicjalizacji danych.

```
_start:
movq $SYSREAD, %rax
movq $STDIN, %rdi
movq $textin, %rsi
movq $BUFLEN, %rdx
syscall
```

Na początku programu wywołujmy funkcję systemową `SYSREAD`, która pobiera trzy argumenty: strumień wejścia, wskaźnik na początek bufora, długość bufora. Jak każdą funkcję systemową wywołujemy ją przy pomocy `syscall`. Po wykonaniu funkcji, w rejestrze `rax` zwracana jest liczba wczytanych znaków.

```
dec %rax          #'\n'
movl $0, %edi     #licznik
```

Dekrementacja rejestru `rax` przy pomocy operandu `dec` dokonywana jest, ponieważ w dalszej części programu chcemy dokonywać zmian na wszystkich wczytanych znakach, oprócz znaku nowej linii, która jest wliczana do wartości zwrotnej (liczby wszystkich wczytanych znaków) po wykonaniu funkcji `SYSREAD`.

```
zamien_wielkosc_liter:
```

Od powyższej etykiety zaczyna się część programu odpowiedzialna za zamianę małych liter na wielkie i wielkich na małe.

```
movb textin(, %edi, 1), %bh
```

1 bajt zostaje odczytany z bufora `texin`. Rejestr `edi` pomaga w indeksowaniu kolejnych wczytywanych bajtów. Ten bajt zostaje skopiowany do rejestru `bh`.

```
movb $0x20, %bl
```

W tym kroku wartość `0x20`, która jest szesnastkowym zapisem dziesiętnej

liczby 32, zostaje przeniesiona do rejestru `bl`. W zapisie binarnym jest to: `0010 0000`.

```
xor %bh, %bl
```

`xor` jest operacją logiczną, w wyniku której dostajemy 1, jeśli na wejściu były dwa różne bity, albo 0, gdy bity wejściowe były takie same. Tej operacji poddawany jest każdy wczytany bajt (znak) z liczbą 32. Wczytany znak jest interpretowany jako kod z tablicy ascii. W zapisie binarnym znaków z tej tablicy wielkie litery mają 0 na bicie 2^5, a małe litery mają 1 w tym samym miejscu. Ta operacja dla dużych liczb ustawia 1 na bicie 2^5, a dla małych ustawia 0. W jej wyniku dostajemy kody ascii liter małych na wielkie i odwrotnie.
Wynik operacji zapisywany jest do argumentu po prawej stronie (*destination*), czyli do rejestru `bl`.

```
movb %bl, textout(, %edi, 1)
```

Zamieniony bajt pojedynczego znaku zostaje przeniesiony do bufora `textout` na pozycję wskazywaną przez licznik - rejestr `edi`.

```
inc %edi
```

Zwiększamy licznik, żeby w kolejnych krokach móc pobrać, przekonwertować i zapisać do bufora wyjściowego kolejny znak wskazywany przez ten rejestr.

```
cmp %eax, %edi
jl zamien_wielkosc_liter
```

Zawartość dwóch rejestrów `eax` i `edi` jest porównywana przez operand `cmp`, a następnie ustawiana jest flaga. `jl` oznacza skok warunkowy, jeśli zawartości rejestru `eax` jest większa niż `edi`. Zatem cała instrukcja począwszy od `zamien_wielkosc_liter`, aż do tego miejsca, będzie się wykonywać dopóki zawartość obu rejestrów nie będzie nową linią (`,\n'`).

```
movb $'\n', textout(, %edi, 1)
```

Na koniec wczytywania wszystkich znaków do bufora `textout` dodany zostaje znak nowej linii.

```
movq $SYSWRITE, %rax
movq $STDOUT, %rdi
movq $textout, %rsi
movq $BUFLEN, %rdx
syscall
```

Wywoływana jest funkcja systemowa, która wypisuje zawartość bufora `textout`.

Na koniec wywoływana jest funkcja `SYSEXIT`, tak samo jak w poprzednim programie (`hello.s`).

3. Podsumowanie i wnioski

Przy wywoływaniu funkcji systemowych wczytywania (`SYSREAD`) i wypisywania (`SYSWRITE`) w systemie 64b należy:

- w `%rax` podać numer wywoływanej funkcji (dla `SYSREAD` jest to `0`, a dla `SYSWRITE` `1`)
- w `%rdi`, `%rsi` i `%rdx` wpisać argumenty funkcji:
  - `%rdi` przyjmuje numer wejścia standardowego (`STDIN = 0`, `STDOUT = 1`);
  - `%rsi` pobiera wskaźnik na miejsce w pamięci, od którego będą wczytywane dane do wypisania lub od którego dane będą alokowane w pamięci;
  - `%rdx` dostaje informację o liczbie bajtów, które będą wczytane do pamięci lub wypisane;
- wywołanie funkcji zainicjować przez `syscall`.

Po wywołaniu `SYSREAD` w rejestrze `%rax` można znaleźć liczbę wczytanych znaków

4. Bibliografia

- https://www.tutorialspoint.com/assembly_programming/assembly_basic_syntax.htm
- https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/dataseg.html
- https://pl.wikipedia.org/wiki/X86-64
- https://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax
- https://en.wikibooks.org/wiki/X86_Assembly/Arithmetic
- J.Barlett *Programming from the ground up*, 2003
- http://sticksandstones.kstrom.com/appen.html