Damian Koper, 241292
CZ-N-17

Wrocław, 13.03.2019r.

Lab 1: Podstawy używania dostępnych narzędzi i tworzenia prostych konstrukcji programowych

Cel ćwiczenia

Celem ćwiczenia było zapoznanie się z dostępnymi narzędziami służącymi do kompilacji, linkowania, debugowania i podglądania stanu i struktury programu jak i plików wykonywalnych. Ćwiczenie skupiało się również na tworzeniu prostych konstrukcji programowych - program 'Hello World!' i rysowanie choinki o zadanej szerokości.

Kompilacja i linkowanie - przykład programu `hello`

Polecenie `as hello.s -o hello.o -g` służy do kompilacji programu.

Polecenie `ld hello.s -o hello` służy do przeprowadzenia procesu linkowania.

Połączenie tych funkcji można uzyskać poprzez użycie kompilatora gcc:

```
gcc -g -o hello hello.s
```

Flaga `-g` w obu przypadkach dodaje do pliku informacje ułatwiające korzystanie z debuggera.

Program `hello`

W komentarzach Kodu opisana jest podstawowa struktura programu:

```
# deklarcje zmiennych
.data
  STDOUT = 1 # deskryptor strumienia wyjściowego
  WRITE = 4 # numery funkcji systemowych
  EXIT = 1
  SYSCALL32 = 1

  hello: .string "Hello World!\n"
  helloLen = 13

# część wykonywalna programu
.text
.globl _start # punkt startu programu _start
_start:
  mov $WRITE, %eax
  mov $STDOUT, %ebx
  mov $hello, %ecx
  mov $helloLen, %edx
```

```
  # wywołanie przerwania systemowego WRITE z parametrami wyżej przekazanymi
do rejestrów
  int $0x80

  mov $EXIT, %eax
  mov $0, %ebx
  # wywołanie przerwania systemowego EXIT z numerem 0 - oddanie sterowania
systemowi operacyjnemu
  int $0x80
```

Użycie narzędzia GDB

Uruchomienie programu `hello` za pomocą debuggera GDB:

```
gdb -q hello.out
```

Po wyświetlaniu zachęty terminala `(gdb)`, komendą `r` (run) uruchamiamy program. Uruchomienie GDB i samego programu daje odpowiedź:

```
Reading symbols from ./hello...done.
(gdb) r
Starting program: /home/damian_koper/Documents/GitHub/OiakLab/lab_1/hello
Hello World!
[Inferior 1 (process 7498) exited normally]
```

W celu zatrzymania i podejrzenia stanu wykonywania programy w jego trakcie możemy ustawić breakpoint (pułapkę) komendą :

```
b[reak] wskaźnik(adres instrukcji)|nr linii
```

Następnie po ścieżce wykonywania programu możemy poruszać się używając komend takich jak `step`, `next`, `stepi`, `continue` itp. Niżej przedstawiono proces zatrzymania programu `hello`:

```
Reading symbols from ./hello...done.
(gdb) b 12
Breakpoint 1 at 0x8048074: file hello.s, line 12.
(gdb) b 18
Breakpoint 2 at 0x804808a: file hello.s, line 18.
(gdb) r
Starting program: /home/damian_koper/Documents/GitHub/OiakLab/lab_1/hello

Breakpoint 1, _start () at hello.s:12
12        mov $WRITE, %eax
(gdb) step
```

```
13        mov $STDOUT, %ebx
(gdb) continue
Continuing.
Hello World!

Breakpoint 2, _start () at hello.s:18
18        mov $EXIT, %eax
(gdb) continue
Continuing.
[Inferior 1 (process 8487) exited normally]
```

Ustawiono tu dwa breakpointy w liniach 12 i 13. Program odpowiednio zatrzymał się przed i po wypisaniu tekstu na ekran. Wykonywanie wznowiono poleceniem `continue`.

Komenda `x`

Komenda `x` wyświetla zawartość pamięci odpowiednio ją formatując. Dokumentacja specyfikuje jej użycie następująco:

```
(gdb) x [Address expression]
(gdb) x /[Format] [Address expression]
(gdb) x /[Length][Format] [Address expression]
```

Przykład użycia dla programu `hello` przed wyświetleniem tekstu (13c oznacza wyświetlenie 13 kolejnych wartości sformatywanych jako `char`):

```
(gdb) x/13c &hello
0x8049096:      72 'H'  101 'e' 108 'l' 108 'l' 111 'o' 32 ' '  87 'W'  111
'o'
0x804909e:      114 'r' 108 'l' 100 'd' 33 '!'  10 '\n'
```

Komenda `x` pozwala na wyświetlanie zawartości pamięci na wiele sposobów, które specyfikuje dokumentacja. Niżej ten sam napis wyświetlono jako 13 wartości szesnastkowo (13xb - 13 wartości szesnastkowo o rozmiarze 1 bajta).

```
(gdb) x /13xb &hello
0x8049096:      0x48    0x65    0x6c    0x6c    0x6f    0x20    0x57
0x6f
0x804909e:      0x72    0x6c    0x64    0x21    0x0a
```

Disassemble

Polecenie disassemble wyświetla treść skompilowanego programu w postaci adresów, mnemoników i argumentów. W tym przypadku strzałką zaznaczone jest polecenie, na którym przerwano wykonywanie programu:

```
(gdb) disassemble
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.
```

List

Polecenie list wyświetla kod programu w pobliżu miejsca wykonywania.

Command

Przydatnym poleceniem jest polecenie `command`, które umożliwia wykonanie komendy po natrafieniu na określony breakpoint. Poniżej przedstawiono działanie komendy, która wypisuje zawartość pamięci po zatrzymaniu:

```
(gdb) b 18
Breakpoint 1 at 0x804808a: file hello.s, line 18.
(gdb) comm 1
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
>x/13c &hello
>end
(gdb) r
Starting program:
/home/damian_koper/Documents/GitHub/OiakLab/lab_1/bin/hello.out
Hello World!

Breakpoint 1, _start () at hello.s:18
18        mov $EXIT, %eax
0x8049096:      72 'H'  101 'e' 108 'l' 108 'l' 111 'o' 32 ' '  87 'W'  111
'o'
0x804909e:      114 'r' 108 'l' 100 'd' 33 '!'  10 '\n'
```

Zawartość pliku wykonywalnego

W celu obejrzenia zawartości pliku wykonywalnego możemy użyć polecenia `hexdump`, które wyświetli zawartość pliku, który jest jednowymiarową tablicą, w postaci szesnastkowej. Przykład dla programu hello:

```
$ hexdump hello.out
0000000 457f 464c 0101 0001 0000 0000 0000 0000
```

```
0000010 0002 0003 0001 0000 8074 0804 0034 0000
...
```

W celu wygenerowania kodu assemblera możemy użyć polecenia `objdump` z argumentami `-S` lub `-D` bądź kombinacją:

```
$ objdump -D -S hello.out
hello.out:     file format elf32-i386
Disassembly of section .text:
08048074 <_start>:
  hello: .string "Hello World!\n"
  helloLen = 13
.text
.globl _start
_start:
  mov $WRITE, %eax
 8048074:       b8 04 00 00 00          mov    $0x4,%eax
  mov $STDOUT, %ebx
 8048079:       bb 01 00 00 00          mov    $0x1,%ebx
  mov $hello, %ecx
 804807e:       b9 96 90 04 08          mov    $0x8049096,%ecx
  mov $helloLen, %edx
 8048083:       ba 0d 00 00 00          mov    $0xd,%edx
  int $0x80
 8048088:       cd 80                   int    $0x80

  mov $EXIT, %eax
 804808a:       b8 01 00 00 00          mov    $0x1,%eax
  mov $0, %ebx
 804808f:       bb 00 00 00 00          mov    $0x0,%ebx
  int $0x80
 8048094:       cd 80                   int    $0x80
 ...
```

Programy

Oprócz testowania i poznawania komend **GDB**, wykonałem również dwa zadane programy:

1. Wypisywanie linii z gwiazdek o zadanej długości:
    - Wykorzystałem pętlę z warunkowym poleceniem skoku
    - Wykorzystałem przerwanie systemowe wywołujące funkcję `READ`, która wczytuje znaki z zadanego strumienia - w tym przypadku STDIN - `0`
    - Wykorzystałem przerwanie systemowe wywołujące funkcję `WRITE`, która pisze do zadanego strumienia - w tym przypadku STDOUT - `1`
    - Link do kodu programu: starLine.s
2. Wypisywanie chionki o zadanej szerokości:
    - Wykorzystałem zagnieżdżone pętle z użyciem polecenia `loop` i rejestru `%ecx`
    - Wykorzystałem przerwania systemowe `READ` i `WRITE`

- Stworzyłem własną funkcję, która tworzy ramkę stosu i wypisuje jeden znak. W celu mniejszego skomplikowania programu mogłem użyć makra `.macro foo ... .endm`.
- W sekcji `.bss` zarezerwowałem niezainicjalizowaną pamięć na bufor wejściowy, który przetwarzam na wartość liczbową
- Link do kodu programu: starTree.s

Wnioski

Z moich obserwacji wynika, że nie potrzeba wielkiego nakładu parcy, aby skonfigurować **GDB** do pracy z moim IDE - Visual Studio Code. Można to zrobić poprzez ręczne ustawianie skrótów klawiszowych i potrzebnych makr, a do uzyskania przenośności konfiguracji i wzbogaconego doświadczenia debggowania można napisać własne rozszerzenie (nie znalazłem istniejącego oferującego podobną funkcjonalność). Ważne jest bowiem, aby stworzyć sobie wygodne środowisko pracy z dość niewygodnymi narzędziami.

Sam **GDB** jak i polecenia `objdump` można używać do analizowania kody maszynowego powstałego poprzez kompilację kodu języka wyższego poziomu (C, C++, ...), co daje możliwości szczegółowej analizy działania programu i tym samym późniejszą dokładną optymalizację.

Literatura

1. Wikibooks x86 Assembly - https://en.wikibooks.org/wiki/X86_Assembly
2. Laboratorium AK –ATT asembler (LINUX) - http://zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Linux-AK2-lab-2018%20May.pdf
3. University of Virginia Computer Science - x86 Assembly Guide http://www.cs.virginia.edu/~evans/cs216/guides/x86.html
4. Prezentacja do wykładu
5. Dokumentacja GDB - https://sourceware.org/gdb/current/onlinedocs/gdb/
6. `gdb help`
7. `man command`