시스템 프로그래밍

어셈블리어의 데이터 이동

2016.10.04

황슬아

seula.hwang@cnu.ac.kr

개요

1. 실습명
   - ✓ GDB
   - ✓ 어셈블리어의 데이터 이동
2. 목표
   - ✓ GDB 실행 및 디버깅
   - ✓ 어셈블리 프로그램의 구조 이해
   - ✓ 데이터 이동 명령어의 사용
   - ✓ 어셈블리 함수의 이해와 사용
3. 내용
   - ✓ 실습 1. Hello, CNU!
   - ✓ 실습 2. 출력함수의 사용
   - ✓ 문제 1. gdb로 rax 살펴보기
   - ✓ 문제 2. gcc –S 로 오류 찾기

GDB

1. GDB
   1) 흔히 GDB라고 부르는 GNU 디버거(debugger)는 GNU 소프트웨어 시스템을 위한 기본 디버거이다.
   2) 디버거: 프로그램의 오류를 찾아내기 위한 소프트웨어. 대상으로하는 프로그램을 특정 단위로 실행해, 레지스터 값이나 프로그램 계수기, 플래그 등 중앙 처리장치의 내부 상황을 나타냄.
2. 디버깅을 위해 컴파일 시 옵션을 주어야 한다.
   1) GDB를 사용하기 위해서는 컴파일시에 –g 플래그를 사용 해야한다.
   2) 컴파일 시에 실행파일에 여러 디버깅 정보를 삽입한다.
      - ✓ 내부에 사용된 심볼 문자열과 심볼의 주소, 컴파일에 사용된 소스 파일, 컴파일된 명령어가 어떤 소스 파일의 어떤 행에 해당되는가와 같은 정보
      - ✓ 디버깅 레벨에 따라서 -g0, -g1, -g2, -g3 등과 같이 설정 할수있다.
        - https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html
3. 실행
   - ✓ gdb 실행 파일
4. 종료
   - ✓ q 입력 혹은 ctrl + d

GDB 실행

1. /home/ubuntu/lab04/gdb_test.tar.gz 파일을 다음과 같이 자신의 홈 디렉토리로 복사 한다.

```
sys03@localhost:~$ cp /home/ubuntu/lab04/gdb_test.tar.gz ~
sys03@localhost:~$ ls
gdb_test.tar.gz
```

2. 압축을 푼다.

```
sys03@localhost:~$ tar zxvf gdb_test.tar.gz
gdb_test.c
```

3. 다음과 같이 –g 옵션을 사용해 컴파일 한다.

```
sys03@localhost:~$ gcc -g -o gdb_test.out gdb_test.c
sys03@localhost:~$ ls
gdb_test.c  gdb_test.out  gdb_test.tar.gz
```

GDB 실행

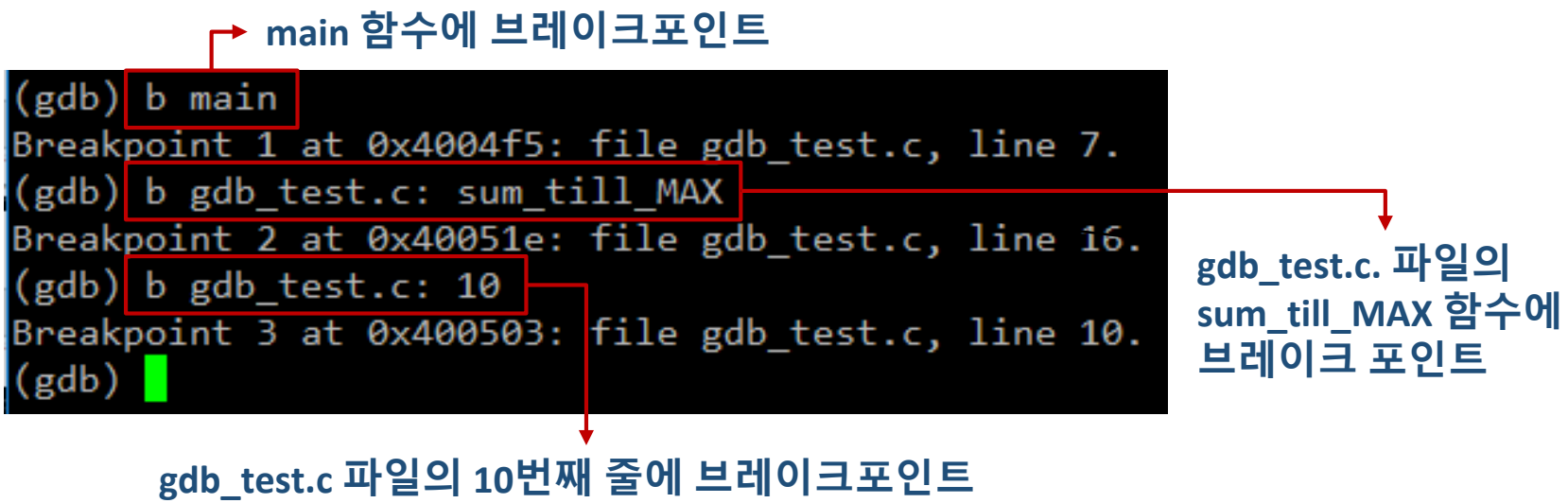
1. 'gdb [실행파일]' 명령어를 사용해 gdb를 실행한다.
   - ✓ gdb의 버전 정보 등이 출력된다.

```
sys03@localhost:~$ gdb gdb_test.out
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 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 gdb_test.out...done.
(gdb)
```

GDB Break point

1. Break Point
   1) 프로그램의 디버그 등에서 검사를 행하기 위하여 실행 중인 처리를 일시적으로 정지하도록 설정
   2) 정지된 상태에서 디버깅을 위한 기능을 사용할 수 있다.
      - 변수 값 및 스택 프레임 출력, break point 재설정, 재실행

2. Break Point 지정
   ✓ b breakpoint 위치



GDB Break point

1. Break Point 설정

✓ 디버깅 시 중단점을 아래와 같이 설정할 수 있다.

Break Point 위치	설 명
함수 명	해당 함수에 break point 설정한다.
행 번호	행에 break point 설정한다.
파일 명 : 함수 명	해당 파일의 함수에 break point 설정한다.
파일 명 : 행 번호	해당 파일의 행에 break point 설정한다.
+ 오프셋	현재 행을 기준으로 오프셋 만큼 + 행에 break point 설정한다.
- 오프셋	현재 행을 기준으로 오프셋만큼 – 행에 break point 설정한다.
*주소	특정 주소에 break point 설정 (어셈블리 디버깅) 한다.
행 if 조건	조건이 만족할 경우 행에 break point 설정한다.

GDB Break point

2. Break Point 제거

✓ cl break point 위치

Break Point 위치	설 명
함수 명	해당 함수의 break point 제거한다.
행 번호	해당 행의 break point 제거한다.
파일 명 : 함수 명	해당 파일의 함수의 break point 제거한다.
파일 명 : 행 번호	해당 파일의 행의 break point 제거한다.

관련 명령	설 명
delete	모든 break point를 제거한다.

GDB Break point

3. Break Point 확인
   1) 'info break' 명령어 사용
   2) Break point의 설정된 위치와 주소를 확인할 수 있다.

```
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004004f5 in main at gdb_test.c:7
2       breakpoint     keep y   0x000000000040051e in sum_till_MAX
                                                   at gdb_test.c:16

(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004004f5 in main at gdb_test.c:7
2       breakpoint     keep y   0x000000000040051e in sum_till_MAX
                                                   at gdb_test.c:16

(gdb) info breakpoint
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004004f5 in main at gdb_test.c:7
2       breakpoint     keep y   0x000000000040051e in sum_till_MAX
                                                   at gdb_test.c:16
```

GDB 디버깅

1. GDB에서 프로그램을 실행
   1) run 명령어 사용
   2) 프로그램을 처음부터 시작 시킨다.
   3) 설정된 break point에서 프로그램이 정지된다.

```
(gdb) b main
Breakpoint 1 at 0x4004f5: file gdb_test.c, line 7.
(gdb) run
Starting program: /home/sys03/sys03/gdb_test.out

Breakpoint 1, main () at gdb_test.c:7
7               int a = 10;
(gdb)
```

GDB 디버깅

1. 디버깅 명령어

실행 명령	설 명
run (r)	gdb에서 프로그램을 시작시킨다.
step (s)	현재 행을 수행하고 정지한다. 함수 호출 시 함수 내부로 들어간다.
next (n)	현재 행을 수행하고 멈춘다. 함수 호출 시 함수 수행 후 다음 행에서 멈춘다.
continue (c)	다음 break point 까지 진행한다.
kill (k)	디버깅 중인 프로그램의 실행을 취소한다.
u	현재 루프를 빠져나간다.
finish	현재 함수를 수행하고 빠져 나간다.
return	현재 함수를 수행하지 않고 빠져 나간다.
return123	현재 함수를 수행하지 않고 나간다. 이때 리턴 값을 123으로 설정한다.

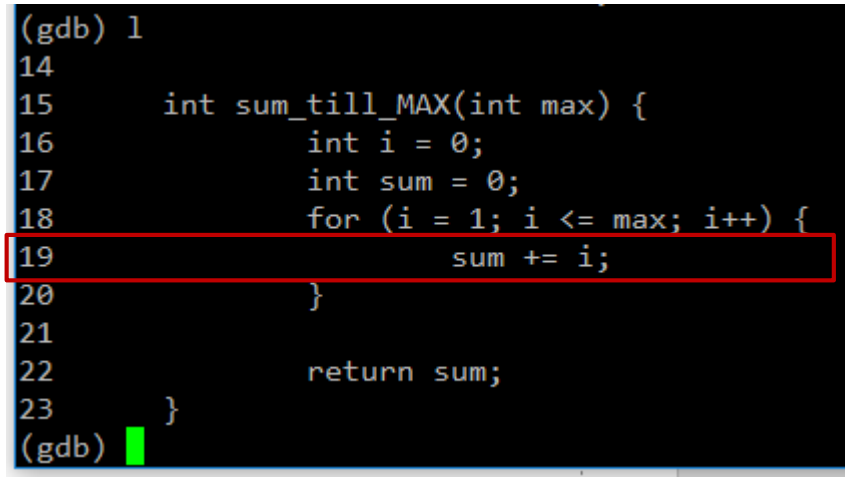
GDB 디버깅

2. 소스 코드 출력

1) 디버깅 중에 현재 실행하고 있는 소스 코드를 볼 수 있다.

2) l (List lines of source code)

3) 소스 코드는 break point를 기준으로 10행을 출력한다.



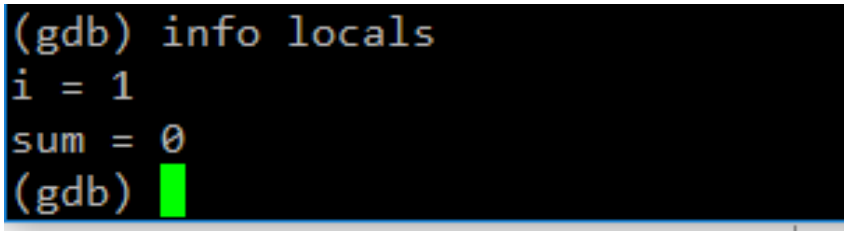
명령	설 명
l	main 함수를 기점으로 소스코드 출력한다.
l 10	10행을 기준으로 출력한다.
l func	func 함수의 소스를 출력한다.
l -	출력된 행의 이전 행을 출력한다.
l file.c:func	file.c 파일의 func 함수를 기준으로 출력한다.
l file.c:1	file.c 파일의 1행을 기준으로 출력한다.
set listsize 20	list 명령의 기본 출력 행의 수를 20행으로 설정한다.

GDB 디버깅

3. 변수 값 확인

1) 전체 지역 변수의 값을 확인 할 수 있다.

2) info locals 명령어



명령	설 명
watch [변수 명]	변수 값이 바뀔 때 마다 값을 확인한다.
info locals	전체 지역 변수를 출력한다.
p [변수 명]	변수의 값을 출력한다.
p *주소	주소가 가리키는 위치의 값을 출력한다.

GDB 디버깅

4. 변수 값 출력 형식 설정

1) p/[출력 형식] [변수 명]

2) p/x var

명령	설 명
t	2진수로 출력한다.
o	8진수로 출력한다.
d	부호가 있는 10진수로 출력(int) 한다.
u	부호가 없는 10진수로 출력(unsigned int) 한다.
x	16진수로 출력한다.
c	최초 1바이트 값을 문자 형으로 출력한다.
f	부동 소수점 값 형식으로 출력한다.
a	가장 가까운 심볼의 오프셋을 출력한다.

GDB 디버깅

5. 변수 값의 변화 확인
   1) display [변수 명]
   2) 매번 변수의 값을 확인

```
(gdb) display sum
1: sum = 0
(gdb) n
18              for (i = 1; i <= max; i++) {
1: sum = 1
(gdb) n
19                      sum += i;
1: sum = 1
(gdb) n
18              for (i = 1; i <= max; i++) {
1: sum = 3
```

명령	설 명
display [변수]	변수 값을 매번 화면에 출력한다.
display/[출력형식] [변수]	변수 값을 설정한 출력 형식으로 출력한다.
undisplay [디스플레이 번호]	디스플레이 설정을 없앤다.
disable display [디스플레이 번호]	디스플레이를 일시 중단한다.
enable display [디스플레이 번호]	디스플레이를 다시 활성화 시킨다.

어셈블리어?

1. **어셈블리어(Assembly Language)**는 0과 1의 이진수 프로그래밍을 좀더 편하게 하기 위해 **비트 패턴을 명령어로 만든 언어**
   - ✓ 하드웨어 디바이스 드라이버, 일반 프로그램의 특정 기능 최적화 등에 사용

2. 어셈블리어는 **시스템의 구조에 따라 문법과 명령어 셋(set) 등이 다르다**
   - ✓ masm(MS), TASM(Borland), NASM(open source), GAS(GNU)

3. 실습시간에는 GNU 소프트웨어인 **GAS 어셈블리어**를 사용한다.
   - ✓ 자세한 내용은 24p '참고 – 어셈블리어' 를 참고

어셈블리어의 장/단점

1. 어세 어셈블리어의 **장점**
   1) **기계와 바로 통신**이 가능함
   2) 명령 실행 **속도가 빠름**
   3) 프로그램 **크기가 작음**

2. 어셈블리어의 **단점**
   1) **배우기 어려움**
   2) 큰 프로그램을 만들기 힘듦
   3) 프로그램 작성 시간과 버그를 잡기 힘듦
   4) **하드웨어 별로 특성이 다름**

기본 레지스터

1. **rsp**
   - ✓ Stack Pointer : Stack의 상위 주소를 가리키는 레지스터

2. **rbp**
   - ✓ Base Pointer : Stack의 Base 주소를 가리키는 레지스터

3. **rip**
   - ✓ Instruction Pointer(Program Counter) : 실행 할 명령의 주소를 가리키는 레지스터.
   - ✓ 각각 명령이 실행될 때, rip에 CPU가 현재 실행하고 있는 주소가 저장됨

어셈블리어 기본 구조

1. **.section**
   - ✓ data, text 등의 메모리 영역을 지정
   - ✓ 일반적으로 코드는 text 영역
   - ✓ 전역 변수, 정적 변수, 배열, 구조체 등은 data 영역에 저장

2. **label**
   - ✓ c언어의 goto 구문과 같이 주소를 저장해 주는 포인터 같은 역할

3. **.global main**
   - ✓ 프로그램의 시작점

```
.section .data
label :

.section .text

.global main
main :


ret
```

실습 1. Hello, CNU!

1. **따라 하기**

✓ 어셈블리어를 이용해서 "Hello, CNU!" 라는 메시지를 출력하는 프로그램을 아래의 과정을 따라 작성

소스코드

```
.section .data
msg :
    .string "Hello, CNU! \n"

.section .text
.global main

main :
    movq    $msg, %rdi
    movq    $0, %rax
    call    printf
    movq    $0, %rax
    ret
```

소스파일명 : ex01.s
실행파일명 : ex01.out

컴파일 및 실행

```
student@localhost:~/test$ vi ex01.s
student@localhost:~/test$ gcc -o ex01.out ex01.s
student@localhost:~/test$ ./ex01.out
Hello, CNU!
student@localhost:~/test$
```

실습 1. Hello, CNU! - 과제

2. **과제**

1) 따라 하기를 참고하여 자신의 학번과 영문이름을 출력하는 프로그램을 작성

2) 컴파일 및 실행 결과

```
student@localhost:~/test$ vi hw01.s
student@localhost:~/test$ gcc -o hw01.out hw01.s
student@localhost:~/test$ ./hw01.out
201500000 Hwang Seula
student@localhost:~/test$
```

실습 2. 출력 함수의 사용법

1. **따라 하기**

   1) 어셈블리어로 아래의 C언어 스타일과 같은 형식의 printf 결과를 보이는 실습을 따라 해 본다.

   2) printf("val1=%d, val2=%d, val3=%d\n", val1, val2, val3);

소스코드

```
.section .data
msg :
    .string "val1 = %d val2 = %d val3 = %d #n"
val1:
    .int 100
val2:
    .int 200
val3:
    .int 300

.section .text
.global main

main :
    movq    $msg, %rdi
    movq    val1, %rsi
    movq    val2, %rdx
    movq    val3, %rcx

    movq    $0, %rax
    call    printf
    movq    $0, %rax
    ret
```

소스파일명 : ex02.s
실행파일명 : ex02.out

컴파일 및 실행

```
student@localhost:~/test$ vi ex02.s
student@localhost:~/test$ gcc -o ex02 ex02.s
student@localhost:~/test$ ./ex02
val1 = 100 val2 = 200 val3 = 300
```

실습 2. 출력 함수의 사용법 - 과제

2. **과제**

1) 앞의 따라 하기를 바탕으로 자신의 키와 나이를 출력하는 어셈블리어 프로그램을 작성하세요. 단, 함수호출을 이용하여 아래와 같이 printf 함수를 호출해서 구현

2) printf("My height is %d cm and my age is %d years old\n", my_height, my_age);

참고 - 어셈블리어

✓ 주의사항

1) 우리는 GAS 어셈블러를 사용하기 때문에 인텔 어셈블리어의 문법규칙이 아니라 AT&T의 문법규칙을 따른다. 그렇기 때문에 위의 사이트에서 알아낸 인스트럭션의 문법을 그래도 써서는 안 된다.

2) AT&T와 인텔의 구문은 source와 destination을 반대로 사용한다. 예를 들면 다음과 같다.
   - 인텔 : mov eax, 4
   - AT&T : movl $4, %eax

3) AT&T 구문에서는 직접 피연산자는 $로 시작한다. 인텔 구문에서는 그렇지 않다. 예를 들면 다음과 같다.
   - 인텔 : push 4
   - AT&T : pushl $4

4) AT&T 구문에서 메모리 피연산자의 크기는 opcode 이름의 마지막 글자로 결정된다. opcode 는 b(8bits), w(16bits), l(32bits)로 각각 정해진 메모리 참조를 나타낸다. 인텔 구문은 메모리 피연산자 접두어(byte ptr, word ptr, dword ptr)로 결정된다. 다음 예를 살펴보자.
   - 인텔 : mov al, byte ptr foo
   - AT&T : movb foo, %al

참고 - 레지스터

https://msdn.microsoft.com/en-us/library/9z1stfyw.aspx

Register	Status	Use
RAX	Volatile	Return value register
RCX	Volatile	First integer argument
RDX	Volatile	Second integer argument
R8	Volatile	Third integer argument
R9	Volatile	Fourth integer argument
R10:R11	Volatile	Must be preserved as needed by caller; used in syscall/sysret instructions
R12:R15	Nonvolatile	Must be preserved by callee
RDI	Nonvolatile	Must be preserved by callee
RSI	Nonvolatile	Must be preserved by callee
RBX	Nonvolatile	Must be preserved by callee
RBP	Nonvolatile	May be used as a frame pointer; must be preserved by callee
RSP	Nonvolatile	Stack pointer
XMM0, YMM0	Volatile	First FP argument; first vector-type argument when **__vectorcall** is used
XMM1, YMM1	Volatile	Second FP argument; second vector-type argument when **__vectorcall** is used
XMM2, YMM2	Volatile	Third FP argument; third vector-type argument when **__vectorcall** is used
XMM3, YMM3	Volatile	Fourth FP argument; fourth vector-type argument when **__vectorcall** is used
XMM4, YMM4	Volatile	Must be preserved as needed by caller; fifth vector-type argument when **__vectorcall** is used
XMM5, YMM5	Volatile	Must be preserved as needed by caller; sixth vector-type argument when **__vectorcall** is used
XMM6:XMM15, YMM6:YMM15	Nonvolatile (XMM), Volatile (upper half of YMM)	Must be preserved as needed by callee. YMM registers must be preserved as needed by caller.

과제

1. 따라 하기와 과제를 진행한 내용을 모두 보고서로 작성하여 사이버캠퍼스와 서면으로 제출(코드는 포함하지 않음)
   1) 파일 제목 : [sys00]HW04_학번_이름
   2) 반드시 파일 제목과 파일 양식을 지켜야 함. (위반 시 감점)
   3) 보고서는 제공된 양식 사용

2. 자신이 실습한 내용을 증명할 것 (결과 화면 – 자신의 학번이 보이도록)

3. 제출 일자
   1) 사이버캠퍼스 : 2016년 10월 11일 08시 59분 59초
   2) 서면 : 2016년 10월 11일 수업시간