시스템 프로그래밍

GDB & 함수의 이용

2016.10.18

황슬아

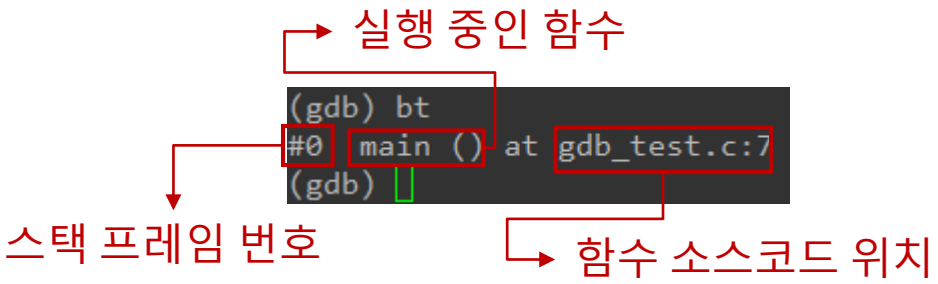
seula.hwang@cnu.ac.kr

개요

1. 실습명
   - ✓ GDB & 함수의 이용
2. 목표
   - ✓ 함수를 사용할 수 있다.
   - ✓ GDB를 사용하여 프로그램을 디버깅할 수 있다.
   - ✓ 디버깅 제어 명령을 사용할 수 있다.
3. 내용
   - ✓ GDB
   - ✓ 함수의 이용

GDB – Stack 디버깅

1. 디버깅 시 현재 스택의 상태를 확인 할 수 있다.
   1) bt(Back Trace)
   2) Back Trace는 Stack Trace라는 의미이다. 프로그램의 실행 중에 현재 동작중인 스택 프레임을 보고한다.



GDB – Stack 디버깅

2. 다음 break point 까지 이동 한 후 스택을 확인한다.
   1) c (continue) 명령을 통해서 다음 break point 까지 이동.
   2) bt 명령으로 현재 스택의 상태를 확인한다.

```
(gdb) c
Continuing.

Breakpoint 2, sum_till_MAX (max=10) at gdb_test.c:16
16              int i = 0;
(gdb) bt
#0  sum_till_MAX (max=10) at gdb_test.c:16
#1  0x000000000040050d in main () at gdb_test.c:10
(gdb)
```

스택 프레임이 증가한 것을 볼 수 있다.

GDB – Stack 디버깅

3. 스택 프레임 내의 지역 변수 확인

1) bt full

```
(gdb) bt full
#0  sum_till_MAX (max=10) at gdb_test.c:16
        i = 0
        sum = 0
#1  0x000000000040050d in main () at gdb_test.c:10
        a = 10
        b = 0
(gdb)
```

명령	설 명
bt full N	최초 N개의 프레임의 bt와 지역 변수를 출력한다.
bt full -N	마지막 N개의 프레임의 bt와 지역 변수를 출력한다.
frame N	N번 스택 프레임으로 변경한다.
up	상위 스택 프레임으로 변경한다.
up N	N번 상위 프레임으로 변경한다.
down	하위 스택 프레임으로 변경한다.
down N	N번 하위 프레임으로 변경한다.

GDB – 메모리 상태 검사

1. 메모리 상태 검사
   1) -g 옵션을 사용하지 않고 컴파일 한 실행 파일의 디버깅에 사용한다.
   2) x/[출력 횟수] [출력 형식] [출력 단위] [출력 위치]

```
(gdb) x/10xb main
0x4004ed <main>:        0x55    0x48    0x89    0xe5    0x48    0x83    0xec    0x10
0x4004f5 <main+8>:      0xc7    0x45
(gdb) x/8xw main
0x4004ed <main>:        0xe5894855      0x10ec8348      0x0af845c7      0xc7000000
0x4004fd <main+16>:     0x0000fc45      0x458b0000      0xe8c789f8      0x0000000a
(gdb)
```

main의 주소 부터 16진수로 1바이트 씩 10개를 출력

main의 주소 부터 16진수로 4바이트 씩 8개를 출력

GDB – 메모리 상태 검사

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

GDB – 메모리 상태 검사

출력 단위	설 명
b	1 바이트 단위 (byte)
h	2 바이트 단위 (half word)
w	4 바이트 단위 (word)
g	8 바이트 단위 (giant word)

GDB – 어셈블리 코드 보기

1. gdb에서 어셈블리 코드를 볼 수 있다.
   1) -g 옵션으로 컴파일 되지 않은 실행 파일 디버깅에 사용한다.
   2) disas [함수 명]: 함수의 어셈블리 코드를 출력한다.
   3) disas [시작 주소] [끝 주소]: 주소 범위의 어셈블리 코드를 출력한다.

```
(gdb) disas main
Dump of assembler code for function main:
   0x00000000004004ed <+0>:     push   %rbp
   0x00000000004004ee <+1>:     mov    %rsp,%rbp
   0x00000000004004f1 <+4>:     sub    $0x10,%rsp
   0x00000000004004f5 <+8>:     movl   $0xa,-0x8(%rbp)
   0x00000000004004fc <+15>:    movl   $0x0,-0x4(%rbp)
   0x0000000000400503 <+22>:    mov    -0x8(%rbp),%eax
   0x0000000000400506 <+25>:    mov    %eax,%edi
   0x0000000000400508 <+27>:    callq  0x400517 <sum_till_MAX>
   0x000000000040050d <+32>:    mov    %eax,-0x4(%rbp)
   0x0000000000400510 <+35>:    mov    $0x0,%eax
   0x0000000000400515 <+40>:    leaveq
   0x0000000000400516 <+41>:    retq
End of assembler dump.
(gdb)
```

어셈블리어 함수

1. 어셈블리어에서 함수는 아래와 같이 선언한다.
   - ✓ .type 함수 명, @function

2. 함수의 호출은 아래와 같다.
   - ✓ call 함수 명

3. 함수의 인자는 정해진 레지스터에 순서대로 저장된다.
   - ✓ 인자가 너무 많아 레지스터를 다 사용하면 어떻게 저장될까?

따라하기 1. 함수 이용

```
.section .data
message :
    .string "%d + %d = %d ₩n"
val1 :
    .int 100
val2 :
    .int 200

.section .text
.global main
main :
    movq    val1, %rsi
    movq    val2, %rdx
    call    add_func

    movq    %rax, %rcx
    movq    val1, %rsi
    movq    val2, %rdx
    movq    $message, %rdi
    movq    $0, %rax
    call    printf

    ret

.type add_func, @function
add_func:
    movq    %rsi, %rax
    addq    %rdx, %rax
    ret
```

1. add_func 함수를 호출 후 결과 값 출력
2. main 함수에서 다음과 같이 add_func 함수 호출

```
movq    val1, %rsi
movq    val2, %rdx
call    add_func
```

3. add_func의 내용은 다음 코드와 같다.

```
int add_func(int a, int b){
        return (a + b);
}
```

4. 인자의 값은 사전에 정의된 레지스터 순서를 따른다
   - ✓ rdi, rsi, rdx, rcx 등등
5. %rax 에 결과 값을 넣으면 결과가 반환된다.

따라하기 2. 함수 이용

```
.section .data
message :
    .string "%d, %d ₩n"
val1 :
    .int 100
val2 :
    .int 200

.section .text
.globl main
main :
    movq    val1, %rsi
    movq    val2, %rdx

    movq    $message, %rdi
    movq    $0, %rax
    call    printf

    movq    val1, %rsi
    movq    val2, %rdx

    call    swap

    movq    val1, %rsi
    movq    val2, %rdx
    movq    $message, %rdi
    movq    $0, %rax
    call    printf

    ret

.type swap, @function
swap:
    movq    val1, %rcx  # temp = val1
    movq    %rdx, val1  # val1 = val2
    movq    %rcx, val2  # val2 = temp

    ret
```

1. swap 함수를 호출 후 결과를 출력
2. 다음과 같은 명령어를 통해 값이 변경된다

```
movq    val1, %rcx
movq    %rdx, val1
movq    %rcx, val2
```

3. 실행 결과

```
sys00@localhost:~/6lab$ ./ex02
100, 200
200, 100
```

과제 1, 2

```
.section .data
i :
    .int 1
result :
    .int 1
msg :
    .string "result : %d ₩n"

.section .text

.globl main
main:
    movl    i, %ecx
    movq    result, %rsi

loop:
    call    factorial
    incl    %ecx
    cmpl    $5, %ecx
    jle     loop

    movq    $msg, %rdi
    movq    $0, %rax
    call    printf
    ret

.type factorial, @function
factorial:
    imulq   %rcx, %rsi
    ret
```

(1)

(2)

(3)

- /home/ubuntu/lab06/factorial 을 자신의 홈 디렉토리로 복사한다.

과제1.

GDB 를 이용해 화살표 (1), (2), (3) 시점 즉, 함수 호출 전, 함수 호출 후, 함수 종료 시점에서의 레지스터 rsp, rcx, rsi 를 모두 비교하여 표로 작성한다.

과제 2.

1) 좌측 코드를 바탕으로 10개의 피보나치 수열을 출력하는 프로그램을 작성하여라

2) 과제 1 에서와 같이 rsp, rcx, rsi 의 변화를 표로 작성하여라

따라하기 3. switch 문의 구현

1. switch 문은 if…else 형태 외에도 jump table 형태로도 구현이 가능하다.

```c
#include <stdio.h>

int main(){

    int x = 1;
    int ch = 0;

    switch(x){
        case 0:
            ch = 65;
            break;
        case 1:
            ch = 66;
            break;
        default:
            ch = 0;
    }

    printf("result: %d \n",  ch);
}
```

C 언어의 switch 문

```
.section .data
printf_str:
    .string "result : %d \n"
x:
    .int 1


JUMP_TABLE:
    .quad .L0
    .quad .L1

.section .text
.globl  main

main:
    movl    x, %ecx
    cmpl    $2, %ecx
    jg      DEFAULT
    jmp     *JUMP_TABLE(, %rcx, 8)
.L0:
    movl    $65, %esi
    jmp     END

.L1:
    movl    $66, %esi
    jmp     END

    DEFAULT:
    movl    $0, %esi

END:
    movq    $printf_str, %rdi
    movq    $0, %rax
    call    printf

    ret
```

jump table

과제 3

1. 왼쪽의 c 코드와 기능이 같은 어셈블리어 코드를 작성하세요. **(jump table 형태로)**

```c
#include <stdio.h>

void main()
{
    int x = 0;
    printf("input number 0~4 : ");
    scanf("%d", &x);

    switch(x){
        case 0:
            printf("linux ₩n");
            break;
        case 1:
            printf("gcc ₩n");
            break;
        case 2:
            printf("switch ₩n");
        case 3:
            printf("asm ₩n");
            goto def;
        case 4:
            printf("gdb ₩n");
            break;
        default:
            def:    printf("example ₩n");
    }
}
```

실행 결과

```
sys00@localhost:~/lab05/TA$ ./hw_jump
input number 0~4 : 0
linux
sys00@localhost:~/lab05/TA$ ./hw_jump
input number 0~4 : 2
switch
asm
example
sys00@localhost:~/lab05/TA$ ./hw_jump
input number 0~4 : 3
asm
example
```

참고 1. $의 사용 (1/2)

1. $는 상수를, %는 레지스터를 표시한다.
   - ✓ ex) movq $0, %rsi
2. 변수와 사용된다면?

: 우선 변수는 프로그래머가 이해하기 쉽게 라벨로 표시한 것으로, 컴퓨터 입장에서는 단순 주소에 지나지 않는다.

```
.section .data
message :
    .string "%d + %d = %d ₩n"
val1 :
    .int 100
val2 :
    .int 200
```

이 경우, data 영역에 val1 의 위치에 int 형태로 메모리를 잡아 100으로 초기화해 준 것으로 볼 수 있다.

```
movq    val1, %rsi
```

즉, 좌측 명령어는 (주소) val1 에 위치한 값 (100) 을 rsi 로 이동시키는 명령어이다.

```
movl    $val1, %esi
```

$는 상수를 의미한다. 따라서 $val1 은 val1(주소) 자체를 의미하게 된다.
즉, %esi 에는 val1 의 주소값이 이동된다.

참고 1. $의 사용 (2/2)

2. GDB를 이용해 값을 확인하면 보다 이해가 쉽다.

```
(gdb) l
15
16      .section .text
17      .globl  main
18      main:
19              movl    $val1, %esi
20              movl    $val2, %edx
21              movq    $scanf_str, %rdi
22
23              movq    $0, %rax
24              call    scanf
```

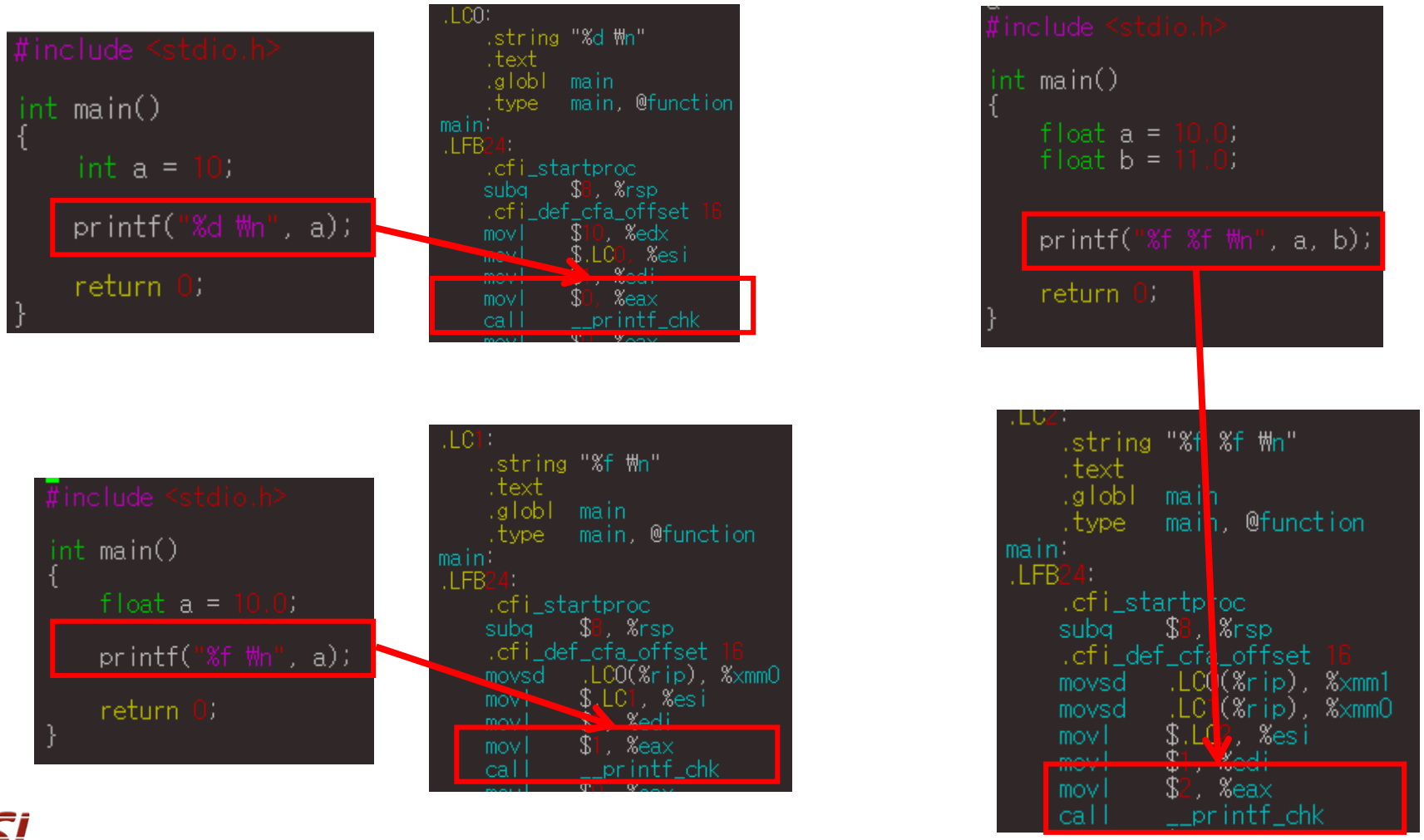
```
(gdb) disas
Dump of assembler code for function main:
=> 0x000000000040057d <+0>:     mov    $0x60107d,%esi
   0x0000000000400582 <+5>:     mov    $0x601081,%edx
   0x0000000000400587 <+10>:    mov    $0x601048,%rdi
   0x000000000040058e <+17>:    mov    $0x0,%rax
   0x0000000000400595 <+24>:    callq  0x400480 <scanf@plt>
   0x000000000040059a <+29>:    mov    0x60107d,%esi
   0x00000000004005a1 <+36>:    mov    0x601081,%edx
   0x00000000004005a8 <+43>:    cmp    %edx,%esi
   0x00000000004005aa <+45>:    je     0x4005c0 <equal>
   0x00000000004005ac <+47>:    mov    $0x601063,%rdi
   0x00000000004005b3 <+54>:    mov    $0x0,%rax
```

```
Breakpoint 1, main () at ex04.s:19
19              movl    $val1, %esi
4: /x $esi = 0xffffe668
3: &val1 = (<data variable, no debug info> *) 0x60107d
2: val1 = 0
(gdb) n

Breakpoint 2, main () at ex04.s:20
20              movl    $val2, %edx
4: /x $esi = 0x60107d
3: &val1 = (<data variable, no debug info> *) 0x60107d
2: val1 = 0
```

참고 2. rax, floating point?

1. C library 를 호출할 때, floating point 를 몇 개 사용할 것인지에 대해 결정해 줘야한다. 이때 rax 에 저장하여 전달한다.



과제

1. 따라하기와 과제를 모두 보고서로 작성하여 사이버캠퍼스와 서면으로 제출
   1) 파일 제목 : [sys00]HW06_학번_이름
   2) 반드시 파일 제목과 파일 양식을 지켜야 함. (위반 시 감점)
   3) 보고서는 제공된 양식 사용
   4) 결과, 코드, 설명이 포함되어야 함

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

2. 제출 일자
   1) 사이버캠퍼스 : 2016년 10월 25일 08시 59분 59초
   2) 서면 : 2016년 10월 25일 수업시간