gdb와 어셈블리어 분석

3주차

5기 장 수 원

x86 vs. ARM

- 이번 주는 x86 머신의 어셈블리어에 대해 살펴보고 다음 수업 때는 ARM 머신 어셈블리어에 대해 살펴본다

gcc로 디버깅 하는 방법

- 컴파일 시 –g 옵션을 붙인다
  - ex. gcc –o function function.c –g
- gdb “실행파일” (맥은 lldb 사용)
  - ex. gdb function

Intel Machine의 특징 (x86)

- ax레지스터는 함수의 리턴값을 저장
- sp는 스택의 최상위
- bp는 스택의 베이스
- ip는 다음에 실행할 주소
- stack은 거꾸로 자란다!

실습 : function.c 작성 및 디버깅 옵션 사용

```c
#include <stdio.h>

int multiply_two (int num)
{
        return num * 2;
}

int main (void)
{
        int num = 3;
        int result = multiply_two(num);
        printf("result = %d\n", result);

        return 0;
}
```

▲ function.c 작성

```
owen@DESKTOP-OT42HI3:~/EDDI/W3$ ls
function.c
owen@DESKTOP-OT42HI3:~/EDDI/W3$ gcc -o function function.c -g
owen@DESKTOP-OT42HI3:~/EDDI/W3$ ls
function  function.c
owen@DESKTOP-OT42HI3:~/EDDI/W3$ gdb function
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 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:
<https://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 function...
(gdb)
```

▲ 디버깅 옵션 사용하여 컴파일

실습 : gdb 사용

```
(gdb) b main
Breakpoint 1 at 0x1167: file function.c, line 10.
(gdb) r
Starting program: /home/owen/EDDI/W3/function
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at function.c:10
10              int num = 3;
(gdb) disas
Dump of assembler code for function main:
   0x000055555555515b <+0>:     endbr64
   0x000055555555515f <+4>:     push   %rbp
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
=> 0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
   0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    call   0x555555555149 <multiply_two>
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
   0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rax        # 0x555555556004
   0x0000555555555187 <+44>:    mov    %rax,%rdi
   0x000055555555518a <+47>:    mov    $0x0,%eax
   0x000055555555518f <+52>:    call   0x555555555050 <printf@plt>
   0x0000555555555194 <+57>:    mov    $0x0,%eax
   0x0000555555555199 <+62>:    leave
   0x000055555555519a <+63>:    ret
End of assembler dump.
```

- b main : main 문에 break point 설정
- r : 실행(run)
- disas : 어셈블리어로 출력

```
(gdb) b *0x000055555555515f
Breakpoint 2 at 0x55555555515f: file function.c, line 9.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/owen/EDDI/W3/function
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 2, 0x000055555555515f in main () at function.c:9
9       {
(gdb) disas
Dump of assembler code for function main:
   0x000055555555515b <+0>:     endbr64
=> 0x000055555555515f <+4>:     push   %rbp
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
   0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    call   0x555555555149 <multiply_two>
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
   0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rax        # 0x555555556004
   0x0000555555555187 <+44>:    mov    %rax,%rdi
   0x000055555555518a <+47>:    mov    $0x0,%eax
   0x000055555555518f <+52>:    call   0x555555555050 <printf@plt>
   0x0000555555555194 <+57>:    mov    $0x0,%eax
   0x0000555555555199 <+62>:    leave
   0x000055555555519a <+63>:    ret
End of assembler dump.
(gdb)
```

- b *"해당주소" : 디버깅 시작 위치 바꾸기 위해 push 명령어에 해당하는 주소로 break point 변경

실습 : gdb 사용

```
(gdb) info reg
rax            0x55555555515b      93824992235867
rbx            0x0                 0
rcx            0x555555557dc0      93824992247232
rdx            0x7fffffffe398      140737488348056
rsi            0x7fffffffe388      140737488348040
rdi            0x1                 1
rbp            0x1                 0x1
rsp            0x7fffffffe278      0x7fffffffe278
r8             0x7ffff7fa6f10      140737353772816
r9             0x7ffff7fc9040      140737353912384
r10            0x7ffff7fc3908      140737353890056
r11            0x7ffff7fde680      140737354000000
r12            0x7fffffffe388      140737488348040
r13            0x55555555515b      93824992235867
r14            0x555555557dc0      93824992247232
r15            0x7ffff7ffd040      140737354125376
rip            0x55555555515f      0x55555555515f <main+4>
eflags         0x246               [ PF ZF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb) p/x $rsp
$1 = 0x7fffffffe278
(gdb) x $rsp
0x7fffffffe278: 0xf7db5d90
(gdb)
```

- **info reg** : 레지스터에 대한 정보를 보기 위한 명령어
- **p/x $"레지스터"** : 해당 레지스터가 가리키고 있는 주소를 16진수로 출력
- **x $"레지스터"** : 해당 레지스터가 가리키고 있는 주소와 그 주소에 있는 값을 16진수 출력

실습 : stack에 데이터가 어떻게 저장되는가?

- 어셈블리어 명령어를 하나씩 따라가보며 아래 3가지를 이해해 본다
  1. 각 명령어가 어떤 동작을 수행하는가?
  2. 명령어를 수행함에 따라 stack에 어떻게 데이터가 쌓이는가?
  3. 그 때마다 SP와 BP가 어떻게 바뀌는가?

- 총 13개+@의 명령어 흐름을 따라가본다

실습 : stack에 데이터가 어떻게 저장되는가?

1. push %rbp

```
(gdb) disas
Dump of assembler code for function main:
   0x000055555555515b <+0>:     endbr64
=> 0x000055555555515f <+4>:     push   %rbp
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
   0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    call   0x555555555149 <multiply_two>
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
   0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rax        # 0x555555556004
   0x0000555555555187 <+44>:    mov    %rax,%rdi
   0x000055555555518a <+47>:    mov    $0x0,%eax
   0x000055555555518f <+52>:    call   0x555555555050 <printf@plt>
   0x0000555555555194 <+57>:    mov    $0x0,%eax
   0x0000555555555199 <+62>:    leave
   0x000055555555519a <+63>:    ret
End of assembler dump.
```

```
(gdb) p/x $rbp
$3 = 0x1
(gdb) x $rsp
0x7fffffffe270: 0x00000001
```

- push %rbp : stack에 rbp가 가리키고 있는 주소값을 밀어넣어라(push)

0xf7db5d90	E278
0x1	E270
	E268
	E260
	E258
	E250
	E248
	E240

실습 : stack에 데이터가 어떻게 저장되는가?

2. mov %rsp, %rbp

```
(gdb) disas
Dump of assembler code for function main:
   0x000055555555515b <+0>:     endbr64
=> 0x000055555555515f <+4>:     push   %rbp
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
   0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    call   0x555555555149 <multiply_two>
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
   0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rax        # 0x555555556004
   0x0000555555555187 <+44>:    mov    %rax,%rdi
   0x000055555555518a <+47>:    mov    $0x0,%eax
   0x000055555555518f <+52>:    call   0x555555555050 <printf@plt>
   0x0000555555555194 <+57>:    mov    $0x0,%eax
   0x0000555555555199 <+62>:    leave
   0x000055555555519a <+63>:    ret
End of assembler dump.
```

```
(gdb) x $rbp
0x7fffffffe270: 0x00000001
(gdb) x $rsp
0x7fffffffe270: 0x00000001
```

- mov %rsp, %rbp : rsp에 있는 값을 rbp에 대입

BP ➡

0xf7db5d90	E278
0x1	E270
	E268
	E260
	E258
	E250
	E248
	E240

⬅ SP

실습 : stack에 데이터가 어떻게 저장되는가?

3. sub $0x10, %rsp

```
(gdb) disas
Dump of assembler code for function main:
   0x000055555555515b <+0>:     endbr64
=> 0x000055555555515f <+4>:     push   %rbp
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
   0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    call   0x555555555149 <multiply_two>
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
   0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rax        # 0x555555556004
   0x0000555555555187 <+44>:    mov    %rax,%rdi
   0x000055555555518a <+47>:    mov    $0x0,%eax
   0x000055555555518f <+52>:    call   0x555555555050 <printf@plt>
   0x0000555555555194 <+57>:    mov    $0x0,%eax
   0x0000555555555199 <+62>:    leave
   0x000055555555519a <+63>:    ret
End of assembler dump.
```

```
(gdb) x $rsp
0x7fffffffe260: 0x00001000
(gdb) x $rbp
0x7fffffffe270: 0x00000001
```

- sub $0x10, %rsp : rsp = rsp – 16
  - 지역변수 공간 확보
  - 변수가 1개라 1칸만 확보하는 듯

BP ➡

0xf7db5d90	E278
0x1	E270
	E268
0x1000	E260
	E258
	E250
	E248
	E240

⬅ SP

실습 : stack에 데이터가 어떻게 저장되는가?

4. movl $0x3, -0x8(%rbp)

```
(gdb) disas
Dump of assembler code for function main:
   0x000055555555515b <+0>:     endbr64
=> 0x000055555555515f <+4>:     push   %rbp
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
   0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    call   0x555555555149 <multiply_two>
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
   0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rax        # 0x555555556004
   0x0000555555555187 <+44>:    mov    %rax,%rdi
   0x000055555555518a <+47>:    mov    $0x0,%eax
   0x000055555555518f <+52>:    call   0x555555555050 <printf@plt>
   0x0000555555555194 <+57>:    mov    $0x0,%eax
   0x0000555555555199 <+62>:    leave
   0x000055555555519a <+63>:    ret
End of assembler dump.
```

```
(gdb) x $rbp-0x8
0x7fffffffe268: 0x00000003
(gdb) x $rsp
0x7fffffffe260: 0x00001000
```

- movl $0x3, -0x8(%rbp) : rbp-0x8 위치에 0x3을 대입

BP ➡

0xf7db5d90	E278
0x1	E270
0x3	E268
0x1000	E260
	E258
	E250
	E248
	E240

⬅ SP

실습 : stack에 데이터가 어떻게 저장되는가?

5. mov -0x8(%rbp), %eax

```
(gdb) disas
Dump of assembler code for function main:
   0x000055555555515b <+0>:     endbr64
=> 0x000055555555515f <+4>:     push   %rbp
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
   0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    call   0x555555555149 <multiply_two>
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
   0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rax        # 0x555555556004
   0x0000555555555187 <+44>:    mov    %rax,%rdi
   0x000055555555518a <+47>:    mov    $0x0,%eax
   0x000055555555518f <+52>:    call   0x555555555050 <printf@plt>
   0x0000555555555194 <+57>:    mov    $0x0,%eax
   0x0000555555555199 <+62>:    leave
   0x000055555555519a <+63>:    ret
End of assembler dump.
```

```
(gdb) p/x $rax
$4 = 0x3
(gdb) x $rbp
0x7fffffffe270: 0x00000001
(gdb) x $rbp-0x8
0x7fffffffe268: 0x00000003
(gdb) x $rsp
0x7fffffffe260: 0x00001000
(gdb)
```

- mov -0x8(%rbp), %eax : rbp-0x8에 있는 값을 eax 레지스터에 대입

BP ➡

0xf7db5d90	E278
0x1	E270
0x3	E268
0x1000	E260
	E258
	E250
	E248
	E240

⬅ SP

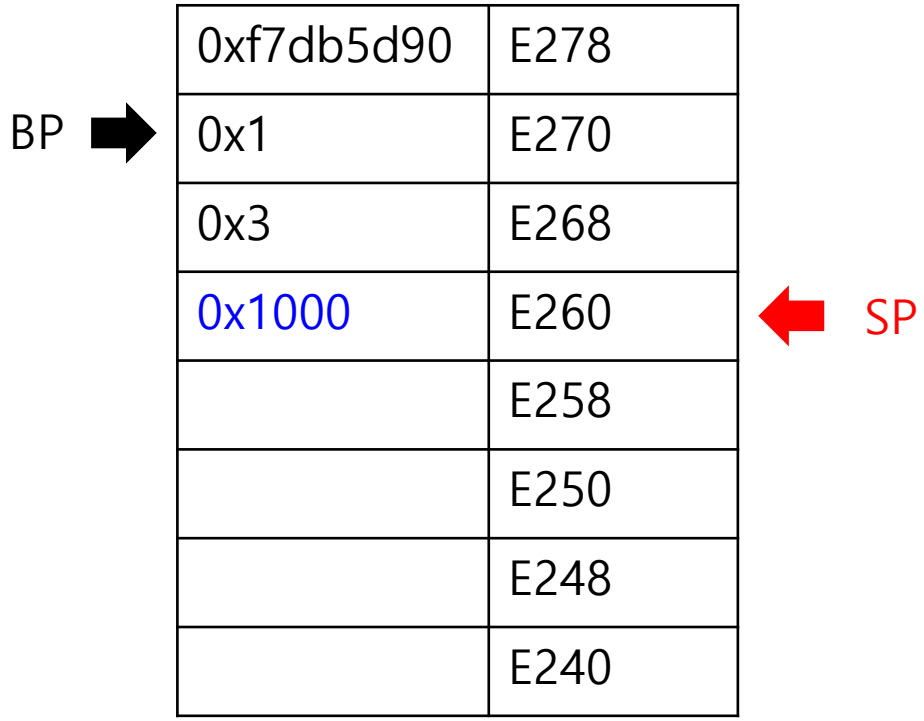
실습 : stack에 데이터가 어떻게 저장되는가?

6. mov %eax, %edi

```
(gdb) disas
Dump of assembler code for function main:
   0x000055555555515b <+0>:     endbr64
=> 0x000055555555515f <+4>:     push   %rbp
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
   0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    call   0x555555555149 <multiply_two>
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
   0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rax        # 0x555555556004
   0x0000555555555187 <+44>:    mov    %rax,%rdi
   0x000055555555518a <+47>:    mov    $0x0,%eax
   0x000055555555518f <+52>:    call   0x555555555050 <printf@plt>
   0x0000555555555194 <+57>:    mov    $0x0,%eax
   0x0000555555555199 <+62>:    leave
   0x000055555555519a <+63>:    ret
End of assembler dump.
```

```
(gdb) info reg
rax            0x3
rbx            0x0
rcx            0x555555557dc0
rdx            0x7fffffffe398
rsi            0x7fffffffe388
rdi            0x3
```

- mov %eax, %edi : eax값을 edi 레지스터에 대입
  - 왜 하는 지 모르겠음



실습 : stack에 데이터가 어떻게 저장되는가?

7. call “addr. " <multiply_two>

```
(gdb) disas
Dump of assembler code for function main:
   0x000055555555515b <+0>:     endbr64
=> 0x000055555555515f <+4>:     push   %rbp
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
   0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    call   0x555555555149 <multiply_two>
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
   0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rax        # 0x555555556004
   0x0000555555555187 <+44>:    mov    %rax,%rdi
   0x000055555555518a <+47>:    mov    $0x0,%eax
   0x000055555555518f <+52>:    call   0x555555555050 <printf@plt>
   0x0000555555555194 <+57>:    mov    $0x0,%eax
   0x0000555555555199 <+62>:    leave
   0x000055555555519a <+63>:    ret
End of assembler dump.
```

```
(gdb) x $rsp
0x7fffffffe258: 0x55555178
```

- call = push + jmp : 복귀할 주소 즉, 다음 명령어의 주소를 stack에 push 하고 뒤에 나온 주소로 점프

BP ➡

0xf7db5d90	E278
0x1	E270
0x3	E268
0x1000	E260
0x55555178	E258
	E250
	E248
	E240

⬅ SP

실습 : stack에 데이터가 어떻게 저장되는가?

7-1. multiply_two 함수로 점프한 이후

- rbp 값이 함수 내에서 바뀔 수 있기 때문에 미리 저장

- push %rbp

```
Dump of assembler code for function multiply_two:
   0x0000555555555149 <+0>:     endbr64
=> 0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555555154 <+11>:    mov    -0x4(%rbp),%eax
   0x0000555555555157 <+14>:    add    %eax,%eax
   0x0000555555555159 <+16>:    pop    %rbp
   0x000055555555515a <+17>:    ret
End of assembler dump.
(gdb)
```

```
(gdb) x $rbp
0x7fffffffe270: 0x00000001
(gdb) x $rsp
0x7fffffffe250: 0xffffe270
```

BP ➡

0xf7db5d90	E278
0x1	E270
0x3	E268
0x1000	E260
0x55555178	E258
0xffffe270	E250
	E248
	E240

⬅ SP

실습 : stack에 데이터가 어떻게 저장되는가?

7-2. multiply_two 함수로 점프한 이후

- 기준이 되는 BP 값을 현재 SP값으로 변경

- mov %rsp, %rbp

```
Dump of assembler code for function multiply_two:
   0x0000555555555149 <+0>:     endbr64
=> 0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555555154 <+11>:    mov    -0x4(%rbp),%eax
   0x0000555555555157 <+14>:    add    %eax,%eax
   0x0000555555555159 <+16>:    pop    %rbp
   0x000055555555515a <+17>:    ret
End of assembler dump.
(gdb)
```

```
(gdb) x $rbp
0x7fffffffe250: 0xffffe270
```

BP ➡

0xf7db5d90	E278
0x1	E270
0x3	E268
0x1000	E260
0x55555178	E258
0xffffe270	E250
	E248
	E240

⬅ SP

실습 : stack에 데이터가 어떻게 저장되는가?

7-3. multiply_two 함수로 점프한 이후

- mov %edi, -0x4(%rbp)

```
Dump of assembler code for function multiply_two:
   0x0000555555555149 <+0>:     endbr64
=> 0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555555154 <+11>:    mov    -0x4(%rbp),%eax
   0x0000555555555157 <+14>:    add    %eax,%eax
   0x0000555555555159 <+16>:    pop    %rbp
   0x000055555555515a <+17>:    ret
End of assembler dump.
(gdb)
```

```
(gdb) x $rbp-0x4
0x7fffffffe24c: 0x00000003
```

- edi 레지스터에 있는 값을 rbp-4 주소에 대입
  - 왜 하는지 모르겠음

	0xf7db5d90	E278
	0x1	E270
	0x3	E268
	0x1000	E260
	0x55555178	E258
BP →	0xffffe270	E250 ← SP
	0x3	E24C
		E240

실습 : stack에 데이터가 어떻게 저장되는가?

7-4. multiply_two 함수로 점프한 이후

- mov -0x4(%rbp), %eax

```
Dump of assembler code for function multiply_two:
   0x0000555555555149 <+0>:     endbr64
=> 0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555555154 <+11>:    mov    -0x4(%rbp),%eax
   0x0000555555555157 <+14>:    add    %eax,%eax
   0x0000555555555159 <+16>:    pop    %rbp
   0x000055555555515a <+17>:    ret
End of assembler dump.
(gdb)
```

```
(gdb) p/x $rax
$5 = 0x3
```

- rbp-0x4에 있는 값을 eax 레지스터에 저장

	0xf7db5d90	E278
	0x1	E270
	0x3	E268
	0x1000	E260
	0x55555178	E258
BP →	0xffffe270	E250 ← SP
	0x3	E24C
		E240

실습 : stack에 데이터가 어떻게 저장되는가?

7-5. multiply_two 함수로 점프한 이후

- add %eax, %eax

```
Dump of assembler code for function multiply_two:
   0x0000555555555149 <+0>:     endbr64
=> 0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555555154 <+11>:    mov    -0x4(%rbp),%eax
   0x0000555555555157 <+14>:    add    %eax,%eax
   0x0000555555555159 <+16>:    pop    %rbp
   0x000055555555515a <+17>:    ret
End of assembler dump.
(gdb)
```

```
(gdb) p/x $rax
$6 = 0x6
```

- multiply_two 함수에서 *return num*2* 에 해당하는 명령

	0xf7db5d90	E278
	0x1	E270
	0x3	E268
	0x1000	E260
	0x55555178	E258
BP →	0xffffe270	E250 ← SP
	0x3	E24C
		E240

실습 : stack에 데이터가 어떻게 저장되는가?

7-6. multiply_two 함수로 점프한 이후

- pop %rbp

- pop $rbp : SP를 한 칸 위로 올리고 그 위치에 있는 값을 rbp로 전달

```
Dump of assembler code for function multiply_two:
   0x0000555555555149 <+0>:     endbr64
=> 0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555555154 <+11>:    mov    -0x4(%rbp),%eax
   0x0000555555555157 <+14>:    add    %eax,%eax
   0x0000555555555159 <+16>:    pop    %rbp
   0x000055555555515a <+17>:    ret
End of assembler dump.
(gdb)
```

```
(gdb) x $rsp
0x7fffffffe258: 0x55555178
(gdb) x $rbp
0x7fffffffe270: 0x00000001
```

BP →

0xf7db5d90	E278
0x1	E270
0x3	E268
0x1000	E260
0x55555178	E258
0xffffe270	E250
0x3	E24C
	E240

← SP

실습 : stack에 데이터가 어떻게 저장되는가?

7-7. multiply_two 함수로 점프한 이후

- ret

```
Dump of assembler code for function multiply_two:
   0x0000555555555149 <+0>:     endbr64
=> 0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555555154 <+11>:    mov    -0x4(%rbp),%eax
   0x0000555555555157 <+14>:    add    %eax,%eax
   0x0000555555555159 <+16>:    pop    %rbp
   0x000055555555515a <+17>:    ret
End of assembler dump.
(gdb)
```

```
(gdb) x $rsp
0x7fffffffe260: 0x00001000
(gdb) x $rbp
0x7fffffffe270: 0x00000001
(gdb) x $rip
0x555555555178 <main+29>:       0x8bfc4589
```

- ret : pop $rip와 같은 것. 즉, SP를 한 칸 위로 올리고 다음 실행할 명령어 주소로 가는 것
  - 여기서는 multiply_two 함수가 끝났음으로 호출 종료 후 그 다음 실행할 명령어로 돌아가는 것

BP ➡

0xf7db5d90	E278
0x1	E270
0x3	E268
0x1000	E260
0x55555178	E258
0xffffe270	E250
0x3	E24C
	E240

⬅ SP

실습 : stack에 데이터가 어떻게 저장되는가?

8. mov %eax, -0x4(%rbp)

- eax 레지스터의 값을 rbp-0x4 위치에 대입

```
(gdb) disas
Dump of assembler code for function main:
   0x000055555555515b <+0>:     endbr64
=> 0x000055555555515f <+4>:     push   %rbp
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
   0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    call   0x555555555149 <mu
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%r
   0x000055555555517b <+32>:    mov    -0x4(%
   0x000055555555517e <+35>:    mov
   0x0000555555555180 <+37>:                                  # 0x555555556004
   0x0000555555555187 <+44>:
   0x000055555555518a <+47>:                  ax
   0x000055555555518f <+52>:                  0x555555555050 <printf@plt>
   0x0000555555555194 <+57>:    mov    $0x0,%eax
   0x0000555555555199 <+62>:    leave
   0x000055555555519a <+63>:    ret
End of assembler dump.
```

여기부터는 잘 모르겠음!

```
(gdb) x $rbp-0x4
0x7fffffffe26c: 0x00000006
```

xf7db5d90	E278
x1	E270
0x6	E26C
0x3	E268
0x1000	E260
0x55555178	E258
0xffffe270	E250
0x3	E24C

SP

실습 : stack에 데이터가 어떻게 저장되는가?

9. mov -0x4(%rbp), %eax

- rbp-0x4 위치의 값을 다시 eax 레지스터에 저장

```
(gdb) disas
Dump of assembler code for function main:
   0x000055555555515b <+0>:     endbr64
=> 0x000055555555515f <+4>:     push   %rbp
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
   0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    call   0x555555555149 <mu
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%r
   0x000055555555517b <+32>:    mov    -0x4(%
   0x000055555555517e <+35>:    mov
   0x0000555555555180 <+37>:                                  # 0x555555556004
   0x0000555555555187 <+44>:
   0x000055555555518a <+47>:                  ax
   0x000055555555518f <+52>:                  x555555555050 <printf@plt>
   0x0000555555555194 <+57>:    mov    $0x0,%eax
   0x0000555555555199 <+62>:    leave
   0x000055555555519a <+63>:    ret
End of assembler dump.
```

여기부터는 잘 모르겠음!

```
(gdb) p/x $rax
$7 = 0x6
```

xf7db5d90	E278
x1	E270
0x6	E26C
0x3	E268
0x1000	E260
0x55555178	E258
0xffffe270	E250
0x3	E24C

SP

실습 : stack에 데이터가 어떻게 저장되는가?

10. mov %eax, %esi

- eax 레지스터 값을 esi 레지스터에 대입

```
(gdb) disas
Dump of assembler code for function main:
   0x000055555555515b <+0>:     endbr64
=> 0x000055555555515f <+4>:     push   %rbp
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
   0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    call   0x555555555149 <mu
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%r
   0x000055555555517b <+32>:    mov    -0x4(%
   0x000055555555517e <+35>:    mov
   0x0000555555555180 <+37>:                                  # 0x555555556004
   0x0000555555555187 <+44>:
   0x000055555555518a <+47>:                  ax
   0x000055555555518f <+52>:                  x555555555050 <printf@plt>
   0x0000555555555194 <+57>:    mov    $0x0,%eax
   0x0000555555555199 <+62>:    leave
   0x000055555555519a <+63>:    ret
End of assembler dump.
```

여기부터는 잘 모르겠음!

```
(gdb) info reg
rax            0x6                 6
rbx            0x0                 0
rcx            0x555555557dc0      93824992247232
rdx            0x7fffffffe398      140737488348056
rsi            0x6                 6
rdi            0x3                 3
```

xf7db5d90	E278
x1	E270
0x6	E26C
0x3	E268
0x1000	E260
0x55555178	E258
0xffffe270	E250
0x3	E24C

SP

실습 : stack에 데이터가 어떻게 저장되는가?

11. lea 0xe7d(%rip), %rax

- ?

```
(gdb) disas
Dump of assembler code for function main:
   0x000055555555515b <+0>:     endbr64
=> 0x000055555555515f <+4>:     push   %rbp
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
   0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    call   0x555555555149 <mu
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%r
   0x000055555555517b <+32>:    mov    -0x4(%
   0x000055555555517e <+35>:    mov
   0x0000555555555180 <+37>:                                  # 0x555555556004
   0x0000555555555187 <+44>:
   0x000055555555518a <+47>:                  ax
   0x000055555555518f <+52>:                  x555555555050 <printf@plt>
   0x0000555555555194 <+57>:    mov    $0x0,%eax
   0x0000555555555199 <+62>:    leave
   0x000055555555519a <+63>:    ret
End of assembler dump.
```

여기부터는 잘 모르겠음!

실습 : stack에 데이터가 어떻게 저장되는가?

12. mov %rax, %rdi

실습 : stack에 데이터가 어떻게 저장되는가?

13. mov $0x0, %eax

```
(gdb) disas
Dump of assembler code for function main:
   0x000055555555515b <+0>:     endbr64
=> 0x000055555555515f <+4>:     push   %rbp
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
   0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    call   0x555555555149 <mu
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%r
   0x000055555555517b <+32>:    mov    -0x4(%
   0x000055555555517e <+35>:    mov
   0x0000555555555180 <+37>:                                  # 0x555555556004
   0x0000555555555187 <+44>:
   0x000055555555518a <+47>:                  ax
   0x000055555555518f <+52>:                  x555555555050 <printf@plt>
   0x0000555555555194 <+57>:    mov    $0x0,%eax
   0x0000555555555199 <+62>:    leave
   0x000055555555519a <+63>:    ret
End of assembler dump.
```

여기부터는 잘 모르겠음!