

```
(gdb) disas
```

```

seo@seo-VirtualBox: ~/my_porj/Homework/jaeonseo/day_2
Start it from the beginning? (y or n) y
Starting program: /home/seo/my_porj/Homework/jaeonseo/day_2/debug2

Breakpoint 2, main () at function1.c:27
27      {
(gdb) disas
Dump of assembler code for function main:
=> 0x000000000400585 <+0>:      push    %rbp
0x000000000400586 <+1>:      mov     %rsp,%rbp
0x000000000400589 <+4>:      mov     $0xa,%r8d
0x00000000040058f <+10>:     mov     $0x8,%ecx
0x000000000400594 <+15>:     mov     $0x6,%edx
0x000000000400599 <+20>:     mov     $0x4,%esi
0x00000000040059e <+25>:     mov     $0x2,%edi
0x0000000004005a3 <+30>:     callq  0x400558 <sumFunc>
0x0000000004005a8 <+35>:     mov     %eax,%esi
0x0000000004005aa <+37>:     mov     $0x400648,%edi
0x0000000004005af <+42>:     mov     $0x0,%eax
0x0000000004005b4 <+47>:     callq  0x400400 <printf@plt>
0x0000000004005b9 <+52>:     mov     $0x0,%eax
0x0000000004005be <+57>:     pop     %rbp
0x0000000004005bf <+58>:     retq

End of assembler dump.
(gdb)

```

step	1	초기 stack 상태
		0xf7a2d830
		rsp
		0x7fffffffde58
		<__libc_csu_init> 0x56415741
		rbp
		0x4005c0

step. 2	push %rbp		0xf7a2d830		
			0x4005c0	rsp	0x7fffffffde50
			<__libc_csu_init> 0x56415741	rbp	0x4005c0

step. 3	mov %rsp, %rbp		0xf7a2d830 0x4005c0	%rsp	%rbp	0x7ffffffde50
			<__libc_csu_init> 0x56415741			

[illegible]

step, 9	callq 0x400558		0xf7a2d830		
			0x004005c0	rbp	0x7fffffffde50
			0x004005a8	rsp	0x7fffffffde48
			<__libc_csu_init>	0x56415741	

step, 9-1	push %rbp		0xf7a2d830		
			0x004005c0	rbp	0x7fffffffde50
			0x004005a8		
			0xffffde50	rsp	0x7fffffffde40
			<__libc_csu_init> 0x56415741		

step. 9-2	mov %rsp, %rbp					
		0xf7a2d830				
		0x004005c0				
		0x004005a8				
		0xffffde50	rbp	rsp		0x7fffffffde40
		<__libc_csu_init> 0x56415741				

step. 9-3	mov %edi, -0x4(%rbp)		0xf7a2d830			
			0x004005c0			
			0x004005a8			
			0xffffde50	rbp	rsp	0x7fffffffde40
			0x2(edi값)	0x7fffffffde4c		
			<__libc_csu_init> 0x56415741			
			rbp주소값 - 0x4 = edi값			

...(지속적으로 업데이트)

2. 포인터크기 내용정리

(1) 포인터의 크기

시스템	크기(Byte)
8 비트	1
16 비트	2
32 비트	4
64 비트	8

(2) 시스템에 따라 포인터의 크기가 변하는 이유

- 컴퓨터의 산술 연산은 ALU에 의존
- ALU의 연산은 범용 레지스터에 종속

(컴퓨터가 64 비트라는 의미는 이들이 64 비트로 구성되었음을 의미)

(3) 변수의 정의

메모리 정보를 저장하는 공간

(4) 포인터의 정의

포인터의 정의는 메모리에 주소를 저장하는 공간

(5) 우분투에서 시스템 포인터의 크기 확인

step.1 터미널 생성

step.2 소스코드 작성

```
vi pointer_size.c
```

```
#include <stdio.h>
```

```
int main(void) {  
    printf("sizeof(int *) = %lu\n", sizeof(int *));  
    printf("sizeof(double *) = %lu\n", sizeof(double *));  
    printf("sizeof(float *) = %lu\n", sizeof(float *));  
    return 0;  
}
```

step.3 결과확인

결과가 전부 8 출력

(6) 스택과 포인터의 관계

- 스택과 컴퓨터의 동작은 포인터 베이스로 이루어짐
- 어셈블리어 분석을 통한 스택 확인방법

step.1

gdb 실행을 통한 디버깅

step.2

```
(gdb) disas
Dump of assembler code for function main:
=> 0x0000000000400585 <+0>:      push    %rbp
    0x0000000000400586 <+1>:      mov     %rsp,%rbp
    0x0000000000400589 <+4>:      mov     $0xa,%r8d
    0x000000000040058f <+10>:     mov     $0x8,%ecx
    0x0000000000400594 <+15>:     mov     $0x6,%edx
    0x0000000000400599 <+20>:     mov     $0x4,%esi
    0x000000000040059e <+25>:     mov     $0x2,%edi
    0x00000000004005a3 <+30>:     callq   0x400558 <sumFunc>
    0x00000000004005a8 <+35>:     mov     %eax,%esi
    0x00000000004005aa <+37>:     mov     $0x400648,%edi
    0x00000000004005af <+42>:     mov     $0x0,%eax
    0x00000000004005b4 <+47>:     callq   0x400400 <printf@plt>
    0x00000000004005b9 <+52>:     mov     $0x0,%eax
    0x00000000004005be <+57>:     pop     %rbp
    0x00000000004005bf <+58>:     retq
```

- 앞서 수행했던 push rbp 쪽에 화살표를 오게하는 일련의 과정을 진행
(gdb) b *메모리주소

(push 명령어는 현재 sp뒤에 오는 값을 집어넣는 명령)

- stack의 증가와 rbp확인

① 화살표가 push %rbp 쪽에 위치시킴

② rbp, rsp 기록

(gdb) p/x \$rbp (rbp 기록)

(gdb) p/x \$rsp (rsp 기록)

③ si실행

(gdb) si

(명령어 한 줄 실행)

④ rsp기록

(gdb) p/x \$rsp

(이전 rsp 값에서 8 이 빠진 것 확인)

⑤ x를 통한 rsp값 확인

(gdb) x \$rsp

(rsp 의 주소 안에 값이 들어갔으므로 이를 확인하기 위해 x 명령어 사용)

∴ 결론

결국 위에서 설명한 컴퓨터의 동작이 포인터 베이스(8 byte) 기준으로 동작함을 증명

3. 2진수, 16진수 내용 정리

(1) 16 진수

숫자	진수
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	a
11	b
12	c
13	d
14	e
15	f

(2) 진수 시스템

진수	사용범위
16 진수 0 ~ f 까지 총 16 개	컴퓨터가 씀
10 진수는 0 ~ 9 까지 총 10 개	인간이 씀
8 진수는 0 ~ 7 까지 총 8 개	리눅스 권한에 사용
3 진수 0 ~ 2 까지 총 3 개	RNA 분석에 사용됨
2 진수 0, 1 로 총 2 개	컴퓨터가 씀

(3) 2진수와 16진수

- 표현방법

표현	진수
10101010101010101010	2 진수
0x2aaaaa	16 진수

- 16진수의 목적

컴퓨터를 배운 사람과 기계의 혼용어로서의 역할

2 진수 4 자리면 ? 16 개

- 16 진수 변환을 수행할 때 4 자리씩 끊어치면 빠름

(예-1) 1001 0000(2진수) \Rightarrow 0x90 (16진수) $\Rightarrow 16^1 \times 9 = 144(10진수)$

(예-2) 10 진수 33 을 2 진수 및 16 진수로 변환

$$33 = 32 + 1$$

10진수	32	16	8	4	2	1
2진수	1	0	0	0	0	1

\Rightarrow 0010 0001(2진수 결과)

\therefore 16진수 변환법

```

8421   8421
0010   0001
-----
0x2     1

```

$$0x21 \Rightarrow 2 \times 16^1 + 1 \times 16^0 = 33$$

(예-3) 10 진수 2568 을 2 진수 및 16 진수로 표기

\therefore 2진수 변환법

$$2^{10} = 1024$$

$$2^{11} = 2048$$

10진수	2048	1024	512	256	128	64	32	16	8	4	2	1
2진수	1	0	1	0	0	0	0	0	1	0	0	0

$$2568 - 2048 = 520$$

$$520 - 512 = 8$$

$$8 - 8 = 0$$

1010 0000 1000

	8421	8421	8421
	1010	0000	1000

0x	a	0	8

∴ 16진수 변환법

$$0xA08 \Rightarrow A \times 16^2 + 8 \times 16^0 = 256 \times 10 + 1 \times 8 = 2568$$

(예-4) 0x48932110 을 2 진수로 변환

∴ 16 진수 1 자리가 2 진수 4 자리라는 것을 기억하고 풀이

8421	8421	8421	8421	8421	8421	8421	8421
0100	1000	1001	0011	0010	0001	0001	0000

0100 1000 1001 0011 0010 0001 0001 0000(2진수)