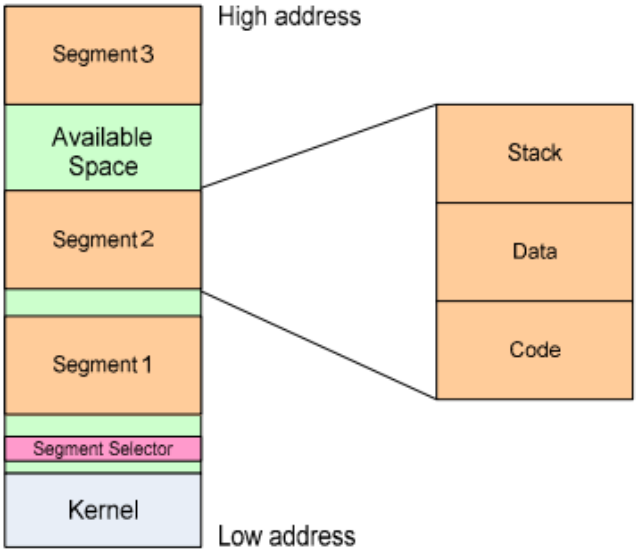
달고나 문서 요약

12191604 박세웅

1.8086 Memory architercture

시스템이 초기화 되기 시작하면 시스템은 커널을 메모리에 적재 그리고 가용 메모리 화인. 커널에서 시스템에 필요한 기본적인 명령어 집합을 찾음. 기본적 커널 = 64kbye 오늘날 더 크게 바낌. 32bit cpu가 한번에 처리가능 그래서 메모리 영역에 주소를 할당할 수 있는 범위가 0 ~ 2에32승 - 1까지 존재 최근은 64bit까지 존재. 프로세스 = 하나의 프로그램이 실행되기 위한 메모리 구조



<그림 2. segmented memory model>

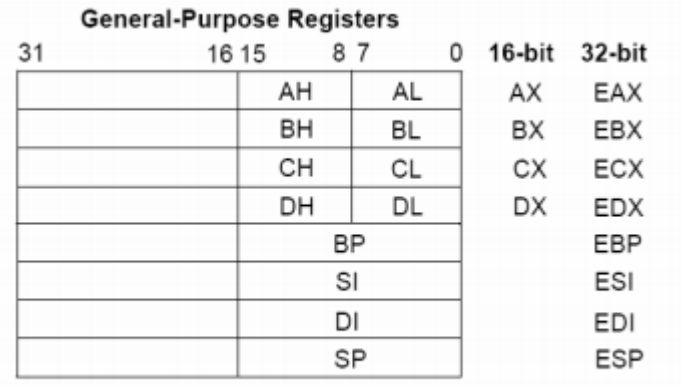
프로세스를 segment라는 단위로 묶어서 가용메모리에 저장. 오늘날은 멀티테스킹 가능 그러므로 여러 개의 segment저장 가능 하나의 segment는 stack segment, data segment, stack segment로 저장. 최대 16383개의 segment생성 그리고 최대 2의 32승 segment크기 가능 code segment에는 명령어 instruction 위치 instruction은 분기과정과 점프,시스템 호출 등을 수행. Segment는 위치 지정 안됨 그래서 logical address사용. Logical address는 physical address와 매핑 segment는 자신의 시작 위치 찾기 가능, 자신의 시작위치로부터 logical address에 있는 명령을 수행할지 결정 physical address = offset + logical address.

Data segment는 프로그램 실행시에 사용되는 데이터(전역 변수) 존재. Data segment는 현재 모듈의 data structure, 상위 레벨로부터 받아들이는 데이터 모듈, 동적 생성 데이터, 다른 프로그램과 공유하는 공유 데이터 부분.

Stack segment 현재 수행되고있는 handler, task, program이 저장됨 우리가 사용하는 버퍼. Multiple stack 사용 가능 switch가능 지역변수가 자리 잡는 공간. 스택은 필요 크기만큼 만들어지고 stack pointer는 프로세스의 명령에 의해 데이터를 저장해 나간다. stack pointer는 레지스터로 스택의 맨 꼭대기를 가리킨다 push와 pop작용 push = 데이터를 저장하면 맨 위로 올리고. Pop할 시에는 맨 위에 거부터 읽는다.

2. 8086 cpu레지스터 구조

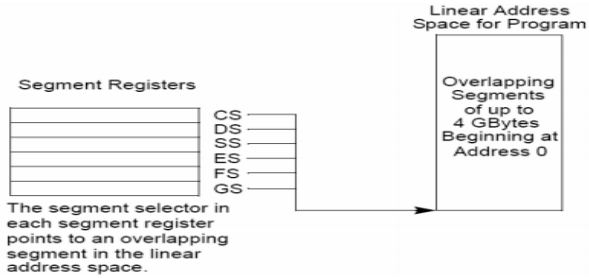
레지스터는 cpu 내부에 존재하는 메모리이다. 레지스터는 4종류 범용 레지스터, 세그먼트 레지스터, 플래그 레지스터, 인스트럭션 포인터로 구성.



<그림 5. 범용 레지스터>

범용 레지스터는 논리연선,수리연산의 피연산자, 주소를 계산하는데 사용하는 피연산자, 메모리포인트가 저장되는 레지스터가 존재한다 임의로 프로그래머가 조작가능. 상위부분을 ah 하위부분을 al이라 한다. 목적이 존재. EAX-피연산자와 연산 결과의 저장소 EBX-DS segment 안의 데이터를 가리키는 포인터

ECX – 문자열 처리나 루프(조건이 만족할 때 까지 반복해서 실행)를 위한 카운터 EDX – I/O 포인터(입출력 장치라는 뜻) ESI – DS 레지스터가 가리키는 data segment 내의 어느 데이터를 가리키고 있는 포인터. EDI- ES 레지스터가 가리키고 있는 data segment내의 어느 데이터를 가리키고 있는 포인터 ESP-ss 레지스터가 가리키는 stack segment의 맨 꼭대기를 가리키는 포인터 EBP-SS 레지스터가 가리키는 스텍상의 한 데이터를 가리키는 포인터



<그림 6. 세그먼트 레지스터>

세그먼트 레지스터는 code segment, data segment, stack segment.를 가리키는 주소가 존재. cs레지스터는 code segment ds, es, fs, gs 레지스터는 data segment, ss 레지스터는 stack segment

플래그 레지스터는 프로그램의 현재 상태나 조건 등을 검사하는데 사용되는 플래그가 존재. 상태 플래그, 컨트롤 플래그, 시스템 플래그들의 집합. 1,3,5,15, 22~31번 비트는 예약 되어있다.

status flags

CF-carry flag 연산을 수행하면서 carry 혹은 borrow가 발생하면 1이 된다.carry와 borrow는 덧셈 연산시 bit bound를 넘어가거나 뺄샘을 하는데 빌려오는 경우. **PF – Parity flag** 연산 결과 최하위 바이트의 값이 1이 짝수 일 경우에 1이 된다. 패리티 체크(이진 부호에서 1을 나타내는 비트의 개수가 짝수 또는 홀수가 되도록 하나의 비트를 부가하여 오류를 찾아내는 방법)를 하는데 사

용된다**.** **AF – Adjust flag**. 연산 결과 carry나 borrow가 3bit 이상 발생할 경우 1이 된다 **ZF – Zero flag.** 결과가 zero임을 가리킨다. If문 같은 조건문이 만족될 경우 set된다. **SF – Sign flag**. 이것은 연산 결과 최상위 비트의 값과 같다. Signed 변수의 경우 양수이면 0 음수이면 1이 된다. **OF – Overflow** flag. 정수형 결과값이 너무 큰 양수이거나 너무 작은 음수여서 피연산자의 데이터 타입에 모두 들어가지 않을 경우 1이 된다. **DF – Direction flag**. 문자열 처리에 있어서 1일 경우 문자열 처리 instruction이 자동으로 감소(문자열 처리가 high address에서 low address로 이루어진다), 0일 경우 자동으로 증가 한다.

System flag

IF – Interrupt enable flag. 프로세서에게 mask한 interrupt에 응답할 수 있게 하려면 1을 준다. **TF – Trap flag**. 디버깅을 할 때 single-step을 가능하게 하려면 1을 준다**. IOPL – I/O privilege level field.** 현재 수행 중인 프로세스 혹은 task의 권한 레벨을 가리킨다. 현재 수행 중인 프로세스의 권한을 가리키는 CPL이 I/O address 영역에 접근하기 위해서는 i/O privilege level보다 작거나 같아야 한다. **NT – Nested task flag**. Interrupt의 chain을 제어한다. 1이 되면 이전 실행 task와 현재 task가 연결되어 있음을 나타낸다**. RF – Resume flag.** Exception debug 하기 위해 프로세서의 응답을 제어한다**. VM – Virtual-8086 mode flag**. Virtual-8086 모드를 사용하려면 1을 준다. **AC – Alignment check flag**. 이 비트와 CR0 레지스터의 AM 비트가 set되어 있으면 메모리 레퍼런스의 alignment checking이 가능하다. **VIF – Virtual interrupt flag**. IF flag의 가상 이미지이다. VIP flag와 결합시켜 사용한다**. VIP – Virtual interrupt pending flag.** 인터럽트가 pending(경쟁 상태) 되었음을 가리킨다**. ID – Identification flag**. CPUID instruction을 지원하는 CPU인지를 나타낸다.

인스트럭션 포인터 다음 수행해야 하는 명령이 있는 메모리 상의 주소가 들어있는 레지스터 인스트럭션 포인트는 code segment와 offset값을 가진다. 하나의 명령어 범위에서 선형 명령 집합의 다음위치를 가리 킬 수 있다. EIP레지스터는 소프트웨어에 의해 바로 엑세스 할 수 없고 control-transfer instruction (JMP, Jcc, CALL,RET)이나 interrupt와 exception에 의해서 제어된다. 읽을 수 있는 방법은 call instruction을 수행하고 프로시저 스택으로부터 리턴하는 인스트럭션의 주소를 읽는 것이다.

3.프로그램 구동 시 segment에서는 어떤 일이?

```
void function(int a, int b, int c){
        char buffer1[15];
        char buffer2[10];
}

void main(){
        function(1, 2, 3);
}
```

프로그램이 실행되어 프로세스가 메모리에 적재되고 메모리와 레지스터가 어떻게 동작하는지 알아보기 위해 프로그램 제작

옆 상황처럼 프로그램을 만들어 c프로그램을 어셈블리 코드로 변환 시키면 $gcc -S -o simple.asm simple.c와 같은 옵션으로 컴파일(컴퓨터를 사용하기 위해 하드웨어를 동작시키는 데 필요한 언어처리를 하는 것) **시킨다.**

```
[dalgona@redhat8 bof]$ cat simple.asm
        .file   "simple.c"
        .text
.globl function
        .type   function,@function
function:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        leave
        ret
.Lfe1:
        .size   function,.Lfe1-function
.globl main
        .type   main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        subl    $4, %esp
        pushl   $3
        pushl   $2
        pushl   $1
        call    function
        addl    $16, %esp
        leave
        ret
.Lfe2:
        .size   main,.Lfe2-main
        .ident  "GCC: (GNU) 3.2.3 20030422 (Hancom Linux 3.2.3)"
[dalgona@redhat8 bof]$
```

컴파일러의 버전에 따라 어셈블리 코드는 다르게 나올 수 있다. 옆에 만들어진 프로그램은 simple.asm이라는 파일 이름으로 생성되었다.

옆의 결과는 한컴 리눅스 3.0 gcc 버전 3.2.3의 결과이다.

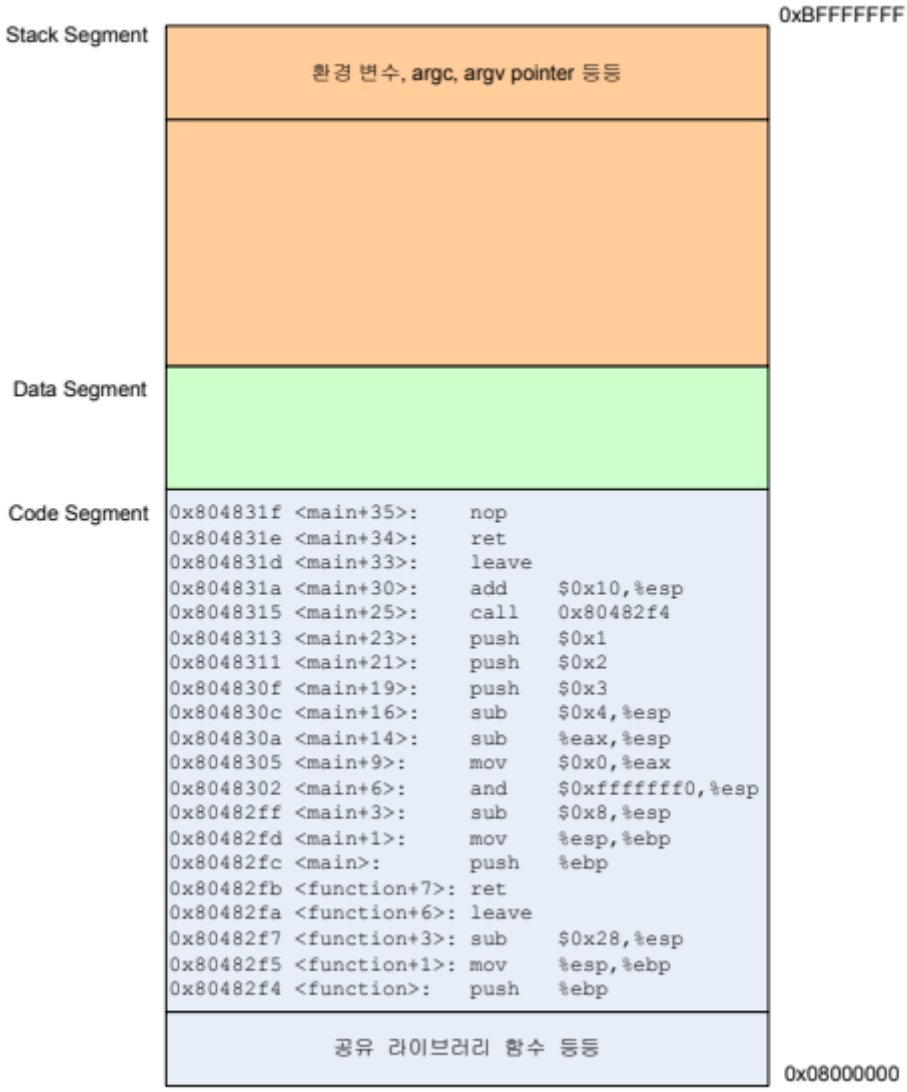
오른쪽은 버전 3.4.2 추가된 코드는 군더기이다.

```
[dalgona@testbed bof]$ cat simple.asm
        .file   "simple.c"
        .text
.globl function
        .type   tion, @function:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        leave
        ret
        .size   tion, .-tion
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        addl    $15, %eax
        addl    $15, %eax
        shrl    $4, %eax
        sall    $4, %eax
        subl    %eax, %esp
        subl    $4, %esp
        pushl   $3
        pushl   $2
        pushl   $1
        call    function
        addl    $16, %esp
        leave
        ret
        .size   main, .-main
        .section        .note.GNU-stack,"",@progbits
        .ident  "GCC: (GNU) 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)"
[dalgona@testbed bof]$
```

```
[dalgona@redhat8 bof]$ gcc -o simple simple.c
simple.c: In function `main':
simple.c:6: warning: return type of `main' is not `int'
[dalgona@redhat8 bof]$ gdb simple
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) disas main
Dump of assembler code for function main:
0x80482fc <main>:        push   %ebp
0x80482fd <main+1>:      mov    %esp,%ebp
0x80482ff <main+3>:      sub    $0x8,%esp
0x8048302 <main+6>:      and    $0xfffffff0,%esp
0x8048305 <main+9>:      mov    $0x0,%eax
0x804830a <main+14>:     sub    %eax,%esp
0x804830c <main+16>:     sub    $0x4,%esp
0x804830f <main+19>:     push   $0x3
0x8048311 <main+21>:     push   $0x2
0x8048313 <main+23>:     push   $0x1
0x8048315 <main+25>:     call   0x80482f4 <function>
0x804831a <main+30>:     add    $0x10,%esp
0x804831d <main+33>:     leave
0x804831e <main+34>:     ret
0x804831f <main+35>:     nop
End of assembler dump.
(gdb) disas function
Dump of assembler code for function function:
0x80482f4 <function>:    push   %ebp
0x80482f5 <function+1>:  mov    %esp,%ebp
0x80482f7 <function+3>:  sub    $0x28,%esp
0x80482fa <function+6>:  leave
0x80482fb <function+7>:  ret
End of assembler dump.
(gdb)
```

위의 내용은 simple.c프로그램이 컴파일 되어 실제 메모리 상에 어느 위치에 존대하기 될지 알아보기위해서 컴파일을 한 것이다. 한컴 리눅스 3.0 gcc 버전 3.2.3을 컴파일 한 것이다.

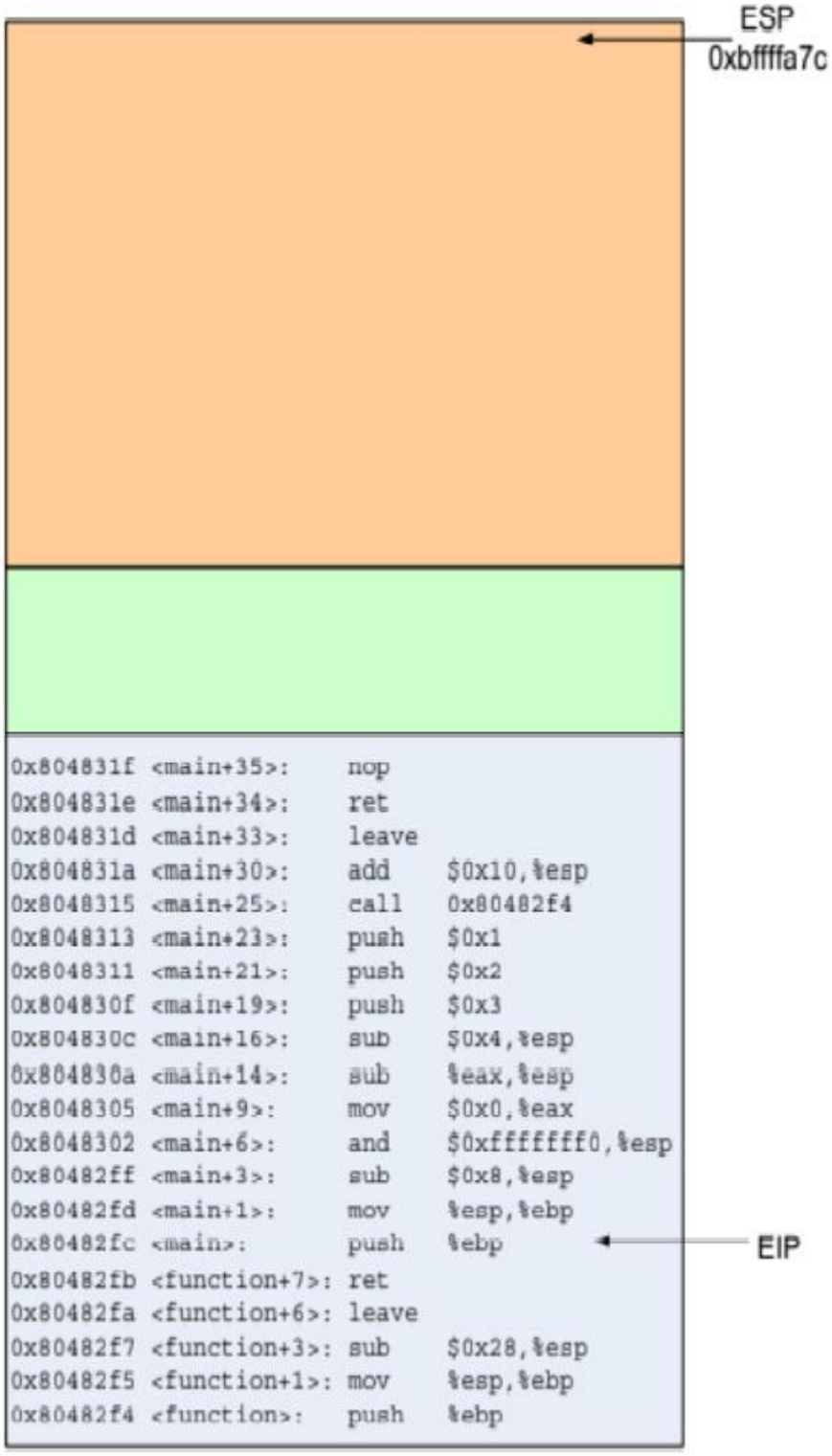
앞에 붙어 있는 주소는 logical address이다. Function()함수는 아래 잡아 있고 main()함수는 위에 자리 잡아 있다. 이 메모리 주소를 바탕으로 생성될 이 프로그램의 segment모양은 아래그림과 같다.



<그림 13. simple.c 프로그램이 실행 될 때의 segment 모습>

Segment의 크기는 프로그램마다 다르기 때문에 최상위 메모리의 주소는 그림과 같이 구성되지 않을 수 있다. 이 segment의 logical address는 0*08000000부터 시작 하지만 실제 프로그램이 컴파일 링크되는 과정에서 다른 라이브러리들을 필요하게 된다. 그래서 코딩한 코드가 시작되는 지점은 시작점과 일치하지 않는다. Simple.c는 전역변수를 지정하지 않아서 data segment에는 링크된 라이브러리의 전역변수 값만 들어 있다. 또한 stack segment 또한 실행옵션으로 주어진 변수 등에 의해서 가용한 영역보다 좀 더 아래에 있다. 위에서 main함수가 시작되는 부분은 0*80482fc

<step1>



EIP는 main()함수의 시작점, ESP가 스텍의 맨 꼭대기를 가리키는 이유는 수많은 push와 pop명령이 있을 것이기 때문이다.

System architecture에 따라 ESP인지점에 PUSH를 넣을지 아니면 그 아래지점에다 넣을지는 다르다. 마찬가지로 POP또한 명령이 ESP가 가리키는 지점의 데이터를 가져갈 것인지 ESP가 가리키는 지점 위를 가져 갈 것인지는 다르다.

하지만 EBP(SS레지스터가 가리키는 스텍상의 한 데이터를 가리키는 포인터)를 저장해 이전에 수행하던 함수의 데이터를 보존한다 이것을 base pointer라고 하고 함수가 시작할 때는 stack pointer와 base pointer를 새로 지정 이것을 함수 프롤로그 과정이라함.

<step2>

명령어를 하나씩 읽어보기

push %ebp : 이전 함수의 base pointer를 저장하면 stack pointer는 4바이트 아래인 0xbffffa78을 가리키게 된다.

mov %esp, %ebp : ESP 값을 EBP에 복사하였다. 이렇게 함으로써 함수의 base pointer와 stack pointer가 같은 지점을 가리키게 된다.

sub $0x8, %esp : ESP에서 8을 빼는 명령이다. 따라서 ESP는 8바이트 아래 지점을 가리키게 되고 스텍에 8바이트의 공간이 생기게 된다. 이것을 스텍이 8바이트 확장되었다고 말한다. 이 명령이 수행되고 나면 ESP에는0xbffffa70 이 들어가게 된다.

and $0xfffffff0, %esp : ESP와 11111111 11111111 11111111 11110000 과 AND 연산을 한다. 이것은 ESP의 주소값의 맨 뒤 4bit를 0으로 만들기 위함이다

mov $0x0, %eax : EAX 레지스터에 0을 넣는다

sub %eax, %esp : esp에 들어 있는 값에서 eax에 들어 있는 값만큼 뺀다. Stack pointer를 eax만큼 확장시키려 하는 것이다

sub $0x4, %esp : 스택을 4바이트 확장. 따라서 값은 0xbffffa6c가 된다.

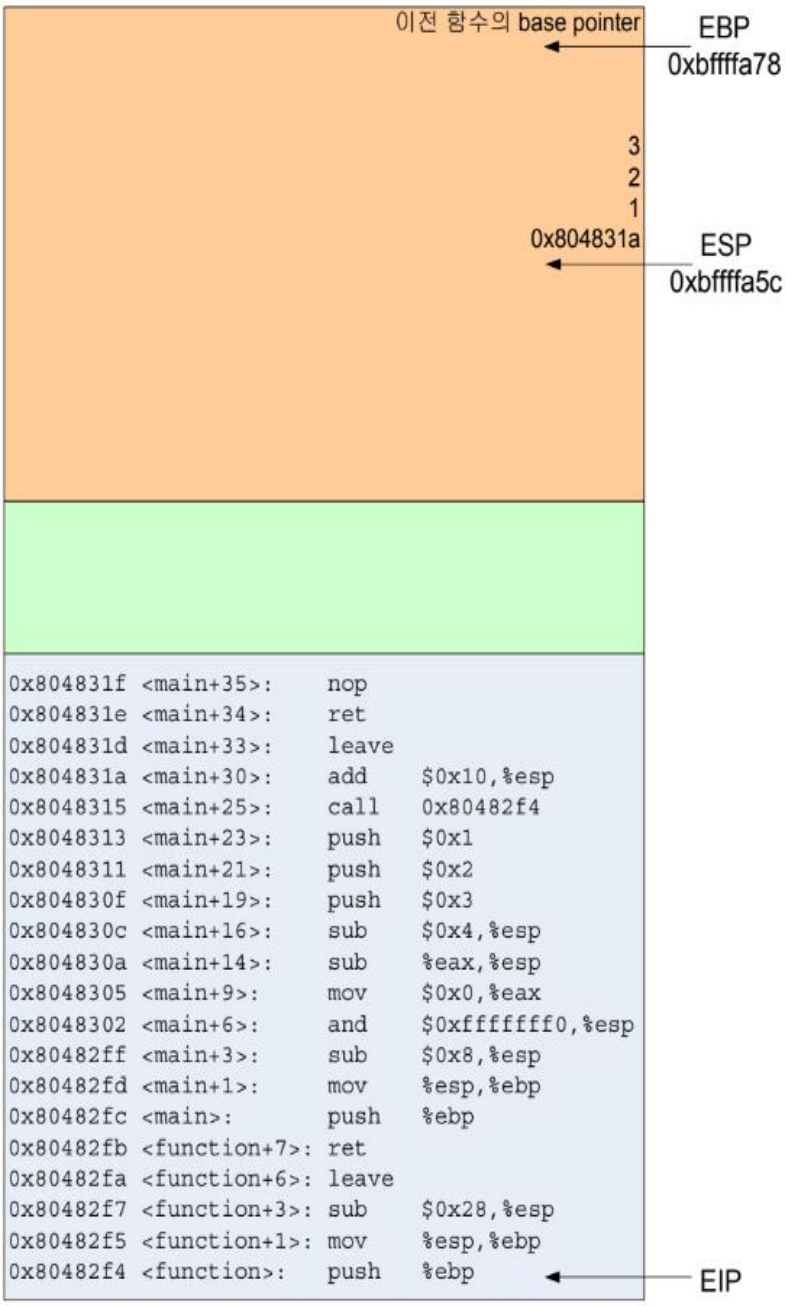
<step3>

Push $0x03, push $0x02, push $0x01 : 아까 제일 처음 int한 숫자들을 1 2 3을 넣기 위한 것. 순서가 3, 2, 1인 이유는 스텍에서 끄집어 낼 때에는 거꾸로 나오기 때문이다

Call 0x80482f4 : 0x80482f4에 있는 명령을 수행하라는 것이다. 여기에는 function 함수가 자리잡는 곳이고 call 명령은 함수를 호출할 때 사용되는 명령으로 함수 실행이 끝난 다음 다시 이 후 명령을 계속 수행할 수 있도록 이 후 명령이 있는 주소를 스텍에 넣은 다음 EIP에 함수의 시작 지점의 주소를 넣는다. add $0x10, %esp 명령이 있는 주소이다. 따라서 함수 수행이 끝나고 나면 이제 어디에 있는 명령을 수행해야 하는가 하는 것을 스텍에서 POP하여 알 수 있게 되는 것이다. 이것이 바로 buffer overflow에서 가장 중요한return address 이다. 이제 EIP에는 function함수가 있는 0x80482f4 주소값이 들어가게 된다.

이렇게 되면 add $0x10, %esp로 돌아가 call 0x80482f4 를 해서 처음 시작하는 부분이므로 int 값을 넣어주는 것이다

<step 4>



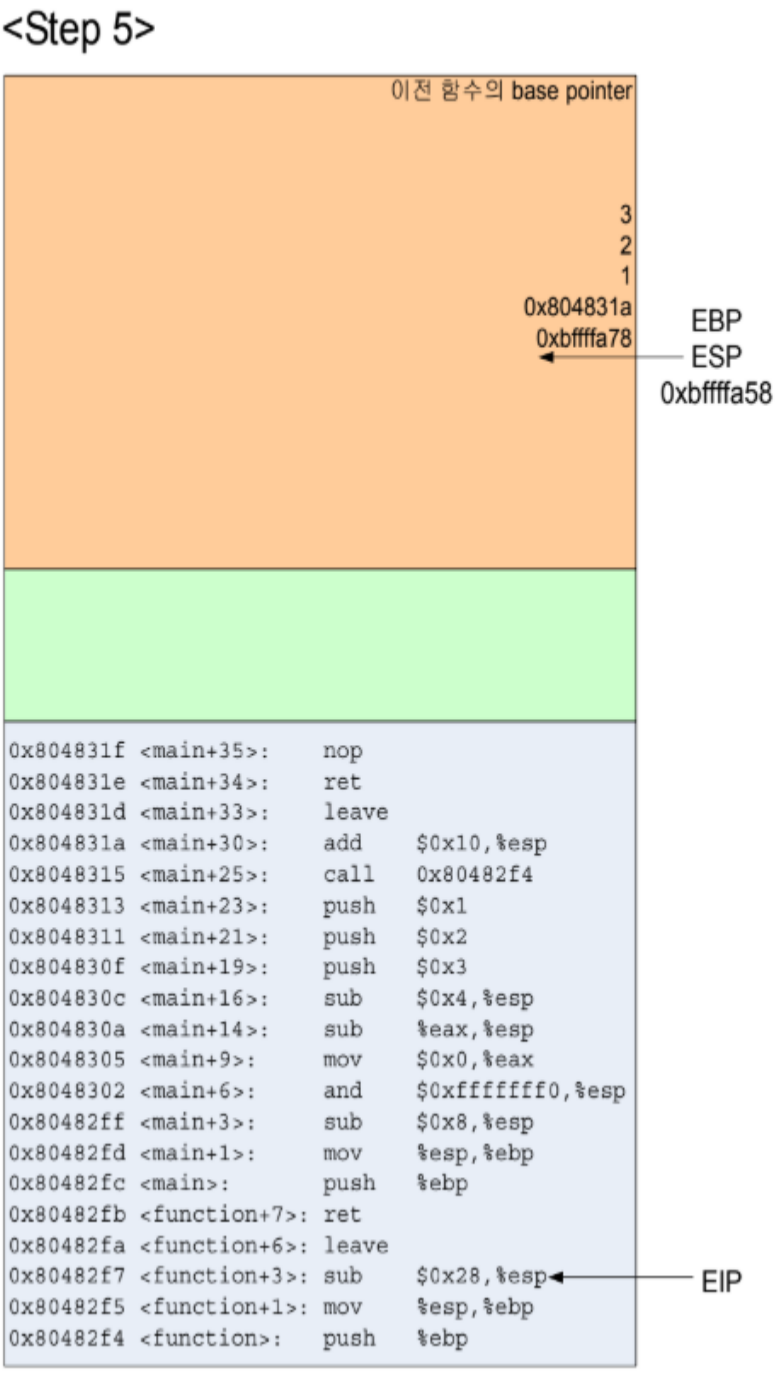
왼쪽의 그림은 step3까지 이루어 졌을 때 나타나는 stack segment이다. 이제 EIP는 function()함수가 시작되는 지점을 가리키고 있고 스텍에는 main()함수에서 넣었던 값들이 쌓여있다.

push %ebp

mov %esp, %ebp

main()함수에서 사용하던 base pointer가 저장되고 stack pointer를 function()함수의 base pointer로 삼는다.

<step 5>



Step4에서 function함수가 끝나고 그 다음 명령어는 이다.

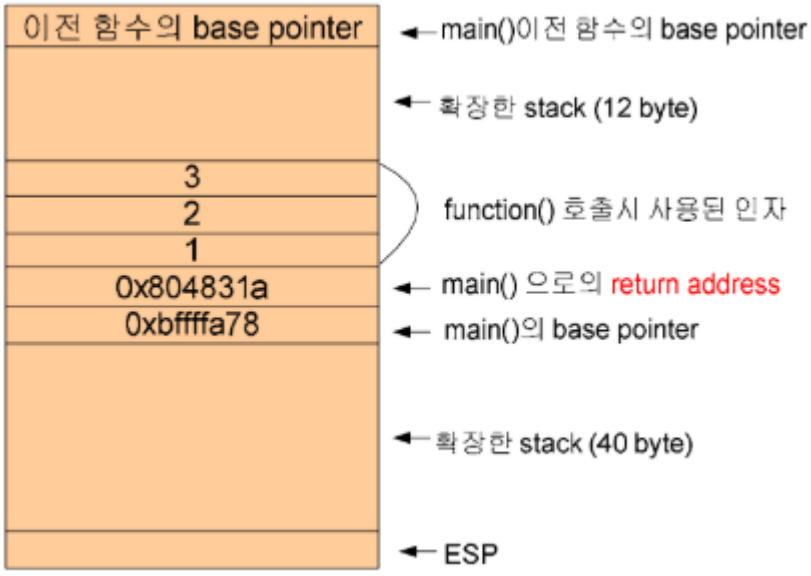
그 후**sub $0x28, %esp**

이걸 만나면 40바이트가 되는데 그 이유는 buffer1[15]를 선언 했을 때 스텍은 4byte단위이므로 16byte가 할당되고 buffer2[10]을 할때도 12byte가 할당 된다. 그러므로 28byte가 확장된다. 하지만 이것도 gcc버전에 따라 달라지는데 gcc 2.96미만은 위에 처럼되지만 gcc 2.96이상은 스텍이 16베수로 할당된다. 정확히 8byte 이하의 버퍼는 1word 단위로 할당되지만 9byte이상의 버퍼는 4word단위로 할당된다. 또한 8byte dummy값이 들어간다.

그러므로 buffer1[15]에 16byte할당그리고 buffer2[10]에 또 16byte할당된뒤 8byte의 dummy가 있어 총40byte **가 할당된다.**

<step6>

Step5에서 버퍼가 만들어졌고 이렇게 만들어진 버퍼에서는 우리가 필요한 데이터를 쓸 수 있게 된다. **mov $0x41, [$esp -4], mov $0x42, [$esp-8]** 과 같은 형식으로 esp를 기준으로 스텍의 특정 지점에 데이터를 복사해 넣는 방식으로 동작한다. 그리고 스텍을 살펴보면



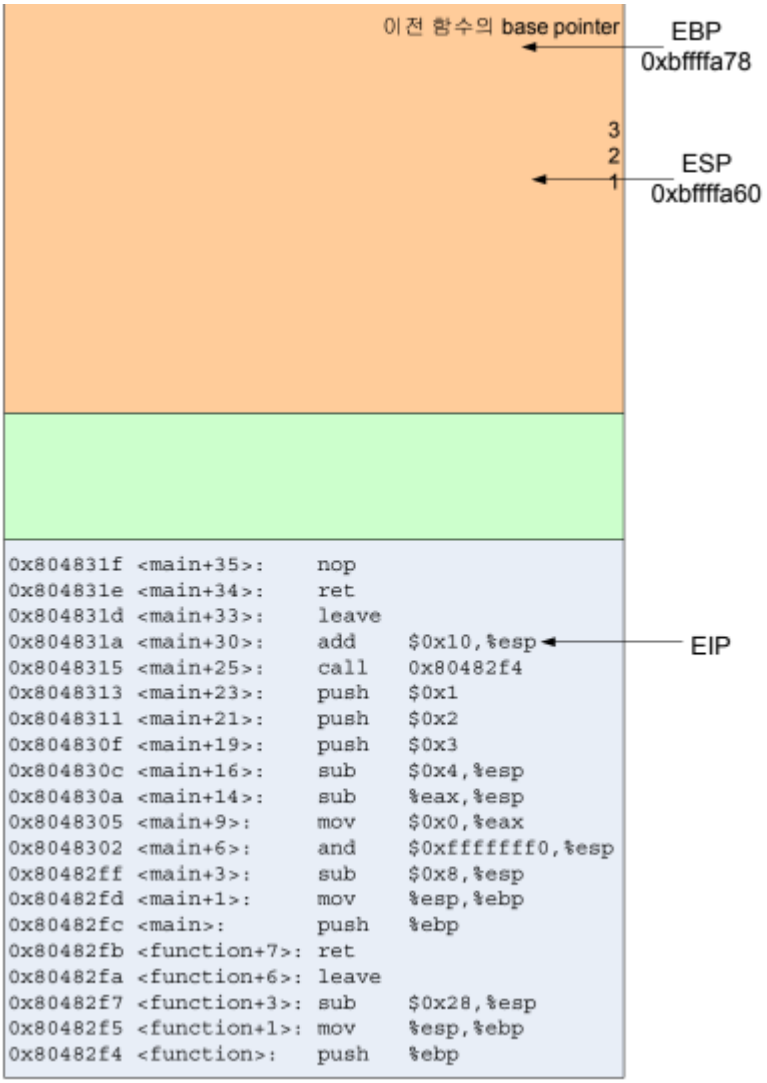
왼쪽의 그림처럼 stack 이 할당되어 만들어진다.

<step 7>

Leave instruction을 수행했고 이제 leave instruction은 함수 프롤로그 작업을 되돌리는 일을 한다.

push %ebp / mov %esp, %ebp 이게 프롤로그 였고 이것을 되돌리는 작업은 **mov %ebp, %esp / pop %ebp** 이 것을 실행함으로써 되돌리는데 이것은 **leave instruction**하나가 위의 두 가지일을 한꺼번에 하는 것이다. Stack pointer를 이전의 base pointer로 잡아서 function()함수에서 확장했던 스텍 공간을 없애 버리고 PUSH해서저장해 두었던 이전 함수 즉, main()함수의 base pointer를 복원 시킨다. Pop를 했으므로 1 word위로 올라간다. 이렇게 하면 stack pointer는 return address로 가게 된다. **Ret instruction** 은 이전의 함수로 return 하라는 소리이다. Eip 레지스터에 return address를 pop하여 집어 넣는 역할을 한다.

<step 8>



add $0x10, %esp

는 스텍을 16byte로 줄인다 따라서 stack pointer는 0x804830c에 있는 명령을 수행하기 이전의 위치로 돌아간다.

Leave ret

를 수행하게 되면 각 레지스터들의 값은 main()함수 프롤로그 작업을 되돌리고 main()함수 이전으로 돌아 가게 된다. 이것은 아마 int_process()함수로 되돌아가게 될 것이다.