EDDI
Electronic Design
Development Institute

에디로봇아카데미

임베디드 마스터 Lv# 과정

제 1 기

2022. 08. 27

조승

디버깅 명령어

EDDI
Electronic Design Development Institute

1) gcc - o : 실행파일 이름을 결정
   실행파일을 만들면 a.out 이름을 생성됨 .
   그러면 어떤 소스코드의 실행파일인지 목적을 알 수 없기 때문에 이름을 명시하는 것이 좋다

2) gcc - g : 디버깅 활성화

3) gdb : 디버깅 sw 의 종류
   gdb 디버깅 과정을 통해 기계어 동작 과정을 파악한다 .

4) disas : 디스 어셈블리어 명령어
   기계어 동작 과정을 어셈블리어를 통해 파악한다 .
   si (step instruction) : 단위 어셈블리어
   s (step) : 단위 C 언어

5) info registers : 실제 cpu 레지스터 정보
   ax: 리턴값 저장하는 레지스터
   sp: 현재 스택의 최상위
   bp: 스택의 base 기준점
   ip: 다음에 실행할 명령어의 주소 포인팅

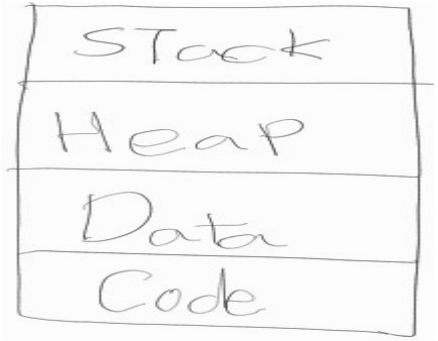
스택 자료구조

스택 자료구조는 베이스 포인터 (Base Pointer, BP) 를 기준으로 데이터가 추가될 때마다 쌓아 올리는 구조이며
새로운 데이터가 추가될 위치를 스택 포인터 (Stack Pointer, SP) 가 가리킨다 .

스택 자료구조는
SP 가 가리키는 주소 메모리에 데이터가 대입되고 SP 주소는 감소한다 (PUSH) *(스택은 거꾸로 자란다)
데이터를 꺼낼 때는 Sp 레지스터가 가지고 있는 메모리 주소에 있는 데이터를 피연산자에 복사한다 .
SP 주소는 증가한다 (POP)

그 외에 mov, call, sub, add 등의 어셈블리어 명령어를 통해 스택의 데이터를 추가 혹은 제거하면서 메모리를 관리한다

메모리 할당
코드 세그먼트 : 실행파일이 실행되어 프로세스가 만들어지면 기계어 명령들이 복사되는 곳으로 프로그램 실행에 사용
데이터 세그먼트 : 프로그램이 끝날 때까지 계속 사용되는 데이터가 저장되는 메모리 공간 (전역변수 , 문자열 상수 등)
스택 세그먼트 : 프로그램 실행 중 임시 데이터를 저장하는 메모리 공간 .
지역 변수가 놓이는 스택 (STACK) 과
동적으로 할당되는 메모리 공간 (HEAP) 으로 나뉨



정적 메모리 할당
컴파일러가 소스 코드를 기계어로 번역하는 시점에 변수를 저장할
메모리 크기나 위치를 배정하는 것을 정적 메모리 할당이라고 함
즉 , 프로그램이 실행될 때 실행 파일에 맞게 사용할 메모리 크기가 결정 됨 .

어셈블리어 명령어

EDDI
Electronic Design Development Institute

1)Push
메모리에 정보를 배치하는 명령어
현재 스택의 최상위 (SP) 에 데이터를 저장하는데 쓰인다 .
자동으로 SP 레지스터가 가리키는 주소 감소
ex)
push eax : 스택 (sp) 에 eax 의 값을 저장한다 .

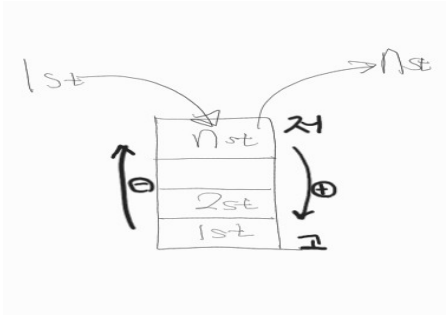
2)Move
메모리나 레지스터의 데이터값을 복사 (이동) 하는 명령어
ex)
mov rsp, rbp : rbp 레지스터 값을 rsp 로 옮긴다 (복사한다)
mov byte ptr [var], eax : eax 레지스터 값을 var 주소가 가리키는 곳으로 복사한다 .

movl : 4byte 처리하여 복사
mov8 : 8byte 처리하여 복사
ex)
movl 0x03 ,-0x8(%rbp) : rbp 기준 8byte 뺀 위치에 숫자 3 배치

어셈블리어 명령어

1)Sub
데이터를 뺄셈하는 명령어 , 데이터 공간을 확보하는 명령어
ex)
sub 0x10, rsp : rsp 에 데이터 값을 0x10 뺀다 . = (sp 가 가리키는 주소값을 0x10 만큼 뺀다) = 데이터 공간 확보

2)Pop
Sp 레지스터가 가지고 있는 데이터 값을 꺼내서 원하는 곳에 데이터를 저장 (스택 조작)
ex)
pop eax : sp 에 있는 값을 꺼내서 eax 에 저장



- Pop 과 Push 의 차이

우선 Stack 구조란 먼저 들어온 것이 나중에 나가는 구조이다 .

Push 가 될때마다 스택 포인터가 감소되고 Pop 이 될 때
스택 포인터가 증가한다 . (* 스택은 거꾸로 자라는 구조 *)

Push (Push on Stack) : 스택에 값을 넣는다 . SP 값이 4 만큼 줄어들고 (스택 포인터가 늘어난다 = SP 값이 줄어든다 * 스택구조) 이 위치에 새로운 값이 채워진다 .

Pop (Pop from Stack) : Sp 레지스터가 가지고 있는 메모리 주소에 있는 데이터를 피연산자에 복사한다 .
Sp 레지스터 값이 4 만큼 더해진다 . (4Byte 증가)

어셈블리어 명령어

EDDI
Electronic Design Development Institute

1) Call
함수를 호출하기 전에 인스트럭션 레지스터 (instruction pointer register , IP) 에 현재 실행 위치를 기억하는 데이터를 스택에 저장하는 명령어
ex)
call plus (*plus 함수 호출) : 스택에 현재 실행 위치 주소 데이터를 PUSH 함 .

2) 함수 호출 과정
C 언어로 작성한 프로그램은 main 함수가 호출되면서 프로그램이 시작되고 main 함수가 다른 함수를 호출하고 호출된 함수가 또 다른 함수를 호출하면서 프로그램이 진행된다 .

기본적으로 함수 호출 과정은
함수 내 작동 과정에서 명령어를 실행하며 필요한 데이터 공간을 확보하고 (BP 와 SP 사이 메모리 공간)
또 다른 함수를 호출할 때에는
실행 위치를 기억하는 IP 데이터와
이전 함수의 BP 를 기억하는 데이터 (이 데이터를 기준으로 새로운 함수의 BP , SP 가 계산됨)
를 스택에 추가하여 또 다른 함수를 작동시킨다 .

함수가 끝나면 실행이 종료된 함수의 메모리 공간을 없애고
이전 함수로 돌아가기 위한 IP 데이터와 BP 데이터를 사용하여
이전 함수의 실행 위치부터 다시 작동하도록 한다 .

어셈블리어 명령어

EDDI
Electronic Design Development Institute

1) Call
함수를 호출하기 전에 인스트럭션 레지스터 (instruction pointer register , IP) 에 현재 실행 위치를 기억하는 데이터를 스택에 저장하는 명령어
ex)
call plus (*plus 함수 호출) : 스택에 현재 실행 위치 주소 데이터를 PUSH 함 .

2) 함수 호출 과정
C 언어로 작성한 프로그램은 main 함수가 호출되면서 프로그램이 시작되고 main 함수가 다른 함수를 호출하고 호출된 함수가 또 다른 함수를 호출하면서 프로그램이 진행된다 .

기본적으로 함수 호출 과정은
함수 내 작동 과정에서 명령어를 실행하며 필요한 데이터 공간을 확보하고 (BP 와 SP 사이 메모리 공간)
또 다른 함수를 호출할 때에는
실행 위치를 기억하는 IP 데이터와
이전 함수의 BP 를 기억하는 데이터 (이 데이터를 기준으로 새로운 함수의 BP , SP 가 계산됨)
를 스택에 추가하여 또 다른 함수를 작동시킨다 .

함수가 끝나면 실행이 종료된 함수의 메모리 공간을 없애고
이전 함수로 돌아가기 위한 IP 데이터와 BP 데이터를 사용하여
이전 함수의 실행 위치부터 다시 작동하도록 한다 .

실습 예제

EDDI
Electronic Design Development Institute

```c
#include <stdio.h>

int mult2(int num)
{
        return num << 1;
}

int main(void)
{
        int data = 3;
        int result = mult2(data);
        printf("result = %d\n", result);

        return 0;
}
```

```
(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>:    callq  0x555555555149 <mult2>
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
   0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rdi        # 0x555555556004
   0x0000555555555187 <+44>:    mov    $0x0,%eax
   0x000055555555518c <+49>:    callq  0x555555555050 <printf@plt>
   0x0000555555555191 <+54>:    mov    $0x0,%eax
   0x0000555555555196 <+59>:    leaveq
   0x0000555555555197 <+60>:    retq
```

실습 예제

```
(gdb) info registers
rax            0x55555555515b     93824992235867
rbx            0x5555555551a0     93824992235936
rcx            0x5555555551a0     93824992235936
rdx            0x7fffffffdc88     140737488346248
rsi            0x7fffffffdc78     140737488346232
rdi            0x1                1
rbp            0x0                0x0
rsp            0x7fffffffdb88     0x7fffffffdb88
r8             0x0                0
r9             0x7ffff7fe0d60     140737354009952
r10            0x7                7
r11            0x2                2
r12            0x555555555060     93824992235616
r13            0x7fffffffdc70     140737488346224
r14            0x0                0
r15            0x0                0
rip            0x55555555515b     0x55555555515b <main>
eflags         0x246              [ PF ZF IF ]
cs             0x33               51
ss             0x2b               43
ds             0x0                0
es             0x0                0
```

명령어 : Info registers
cpu 내 실제 레지스터의 정보를 보여준다

Rbp: base_pointer

Rsp: stack_pointer

Rip : instructer_pointer

0x_____: pointer 주소

그 외
레지스터 : rax, rbx, rdi 등 등

실습 예제

```
(gdb) p/x $rsp
$2 = 0x7fffffffdb88
```

```
(gdb) si
0x0000555555555160
(gdb) p/x $rsp
$4 = 0x7fffffffdb80
```

Push %rbp

Push 는 Stack 최상위에 값을 넣는다

스택에 데이터를 추가하면 Sp 가
가리키는 주소의 메모리에 대입되고

Sp 의 주소는 8Byte 만큼 증가한다 .
(실제로는 수치상 감소함) * 스택 구조

88 → 80

실습 예제

```
(gdb) p/x $rsp
$5 = 0x7fffffffdb80
(gdb) p/x $rbp
$6 = 0x7fffffffdb80
```

Mov %rsp, %rbp

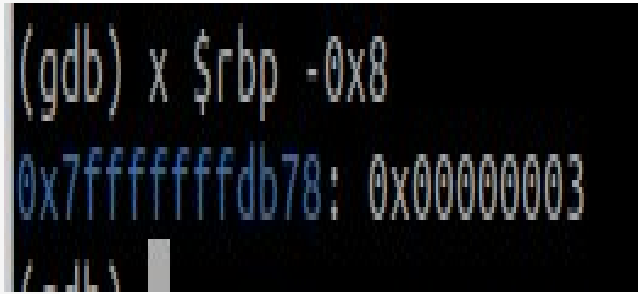
rsp 의 값을 rbp 에 넣어라
좌측 레지스터 정보를 우측 레지스터로
복사하는 명령어이다 .

```
(gdb) p/x $rsp
$7 = 0x7fffffffdb70
(gdb) p/x $rbp
$8 = 0x7fffffffdb80
```

Sub $0x10, %rsp

rsp 에서 0x10 을 빼서 rsp 에 대입
즉 rsp 에서 0x10 을 뺄샘하는 의미이다

실습 예제

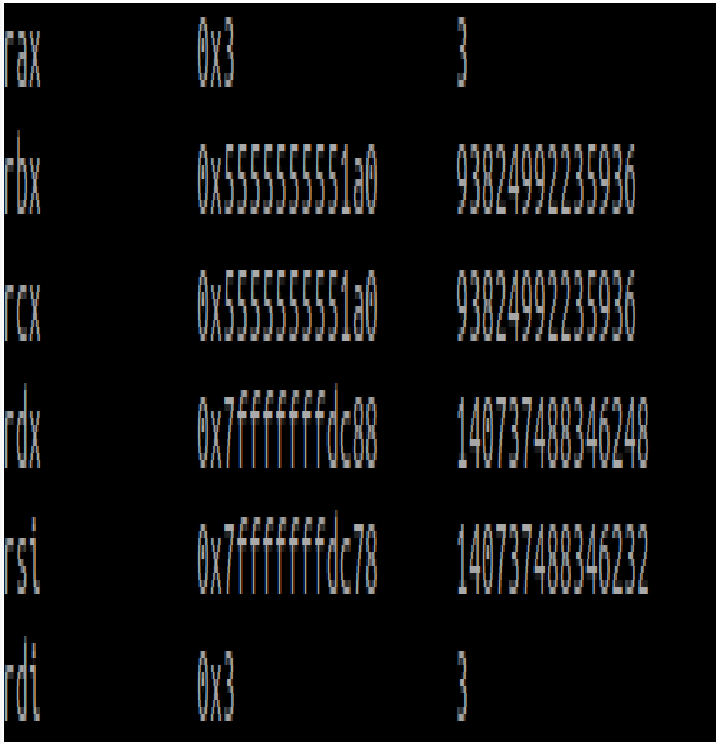


Movl $0x3, -0x8(%rbp)

숫자 3 을 rbp 로 부터 8byte 공간을 할당한 후 그 자리에 넣을 것

변수 num 의 주소는 db78
db80 → db78

실습 예제



Mov -0x8(%rbp) , %eax

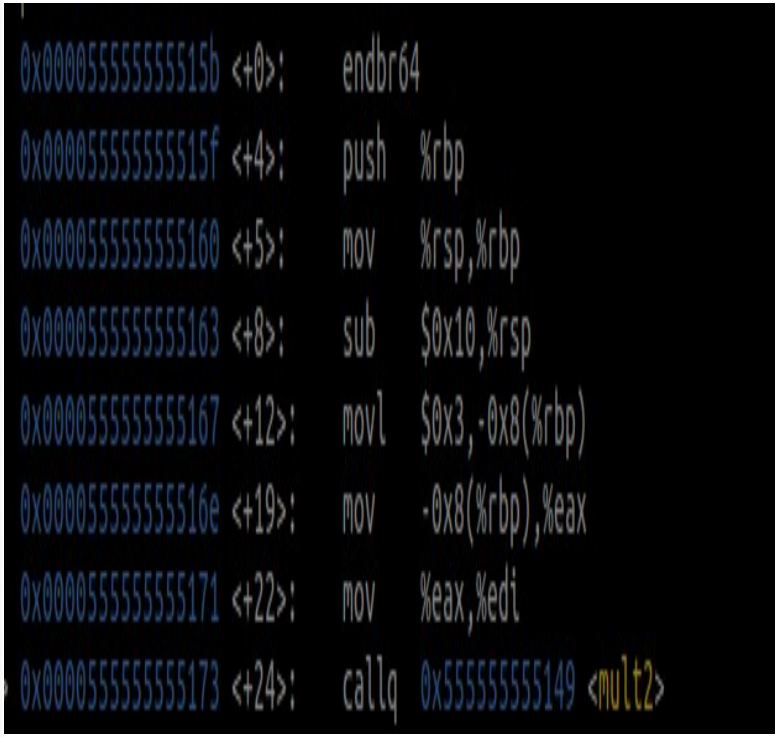
Num 변수가 있는 주소 (-0x8(%rbp) 의 메모리 값을 eax 레지스터에 복사

Mov %eax, %edi

Eax 레지스터에 있는 값을 edi 레지스터에 복사하기

Ax 레지스터는 리턴값을 저장함

실습 예제



Callq 0x~~~~<mult2>

Mult2 함수를 호출한다 .

Main 함수에서 mult2 함수를 호출할 때에는

Main 함수의 실행 위치를 기억하는

정보를 저장한다 .

자동으로 sp 포인터는 증가한다 .

실습 예제

```
mult2 (num=21845) at fun.c:4
4       {
(gdb) disas
Dump of assembler code for function mult2:
=> 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>:    retq
End of assembler dump.
```

Mult2 함수

1. Push %rbp : 이전 main 함수의 rbp 데이터값 (주소) 스택 추가

2. Mov %rsp, %rbp : rsp 데이터 값 rbp 값에 복사 (sp 와 bp 동일)

3. Mov %edi, -0x4(%rbp) : edi 레지스터의 값 (3 이라는 숫자) 을 rbp 데이터값 (주소) 로 부터 0x4 만큼 뺀 위치에 복사

4. Mov -0x4(%rbp), %eax : 이전 3 숫자 데이터가 있는 값을 eax 레지스터에 복사

5. add %eax , %eax : eax 레지스터 (3) + eax 레지스터 (3) = 6 eax 레지스터에 저장
   - num >>2 [0x3 => 0x6]

6. Pop %rbp : 현재 위치 rbp 데이터 값을 이전 main 함수 rbp 데이터로 이동하게 함

7. Retq : rsp 데이터 값이 이전 main 함수의 실행위치를 기억하는 ip 데이터를 받아 이전 main 함수내 call 함수가 실행됐을때의 sp 포인터 주소로 이동함

⇒결론적으로 mult2 함수가 종료되고 이전 main 함수 rbp 값과 실행위치 rsp 값이 복원 됨

실습 예제

스택 프레임 이동과정