2018. 02. 22. - 2일차 수업

---

* github에서 파일 불러오기

1. 불러올 디렉토리를 만든다

mkdir my_proj

cd my_proj

2. git에서 해당 링크를 복사하여 명령어를 입력한다.

git clone https://github.com/SHL-Education/Homework.git

3. 해당 디렉토리로 이동한 뒤 명령어를 입력하여 갱신(업데이트)한다.

cd Homework

git pull origin master

4. pwd 를 입력하여 현재 위치를 확인하고 원하는 위치로 이동한다.

cd ~/my_proj/Homework/sanghoonlee

---

* 메모리 계층구조

- 속도 순서

1. 레지스터

2. 캐시

3. 메모리

4. 디스크

- 용량 순서

1. 디스크

2. 메모리

3. 캐시

4. 레지스터

---

* 메모리 공간의 스택(Stack)

스택은 임시 데이터를 보관하는 메모리 공간이다

스택에서는 특이하게 데이터 값이 쌓이면 - 의 주소를 가지게 된다. 반대로 데이터 값이 빠지면 +하게 된다

(스택은 아래 방향으로 쌓인다)

스택(Stack)을 제외하고는 나머지 전부 일반적인 절차로 쌓이게된다 (쌓이면 + 빠지면 -)

* 스택(Stack)의 용도

레지스터는 메모리 계층구조를 봤을 때 용량은 작고 속도는 빠르다

레지스터 수천만개를 쓰면 좋겠지만

단가가 너무 높아진다는 문제가 있다.

단가도 낮추고 용량 문제도 해결하기 위해 '메모리'를 이용한다

레지스터가 부족하면 메모리에 값을 잠깐 저장했다가 필요하면 다시 레지스터로 불러쓴다

(엄밀하게는 함수 호출을 위한 용도)

x86의 경우 함수의 입력을 스택으로 전달한다

ARM의 경우엔 함수 입력을 4개까진 레지스터로 처리하고 4개가 넘어갈 경우엔 스택에 집어넣는다

(그러므로 성능을 높이고자 한다면 ARM에서는 함수 입력을 4개 이하로 만드는것이 좋음)

* 추가 정보

우선 모든 프로세서는 레지서터에서 레지스터로 연산이 가능하다

x86은 메모리에서 메모리로 연산이 가능하다

하지만 ARM은 로드/스토어 아키텍처라고 해서

메모리에서 메모리로 연산이 불가능하다.

반도체 다이 사이즈가 작아서 Functional Unit 갯수가 적기 때문에 이런 연산을 지원해줄 장치가 부족해서 연산이 안된다.

그래서 ARM은 먼저 메모리에서 레지스터로 값을 옮기고 다시 이 레지스터 값을 메모리로 옮기는 작업을 한다.

이런 방식을 로드하고 스토어하는 방식이라고 해서

로드/스토어 아키텍처라고 한다.

---

* Instruction Scheduling(명령어 스케쥴링)

컴퓨터에서 어떤 명령어를 처리하는데 10 clock이 필요한 것이 있고 어떤 것은 5 clock이 걸리고 또 어떤 것은 1clock이 걸린다

이러한 명령어 배치를 효율적으로 만들어주는 작업을 명령어 스케줄링이라한다

요즘 gcc(컴파일러)가 최적화 옵션을 주지 않아도 스스로 최적화해버리는데 강제로 최적화를 못하게 할 수 있다.

-O0 옵션을 추가하면 된다: 영문자 O와 숫자 0이다.

즉 컴파일 옵션을 아래와 같이 바꾼다.

gcc -g -O0 -o debug func1.c

(모든 최적화를 방지하고 디버깅 옵션을 집어넣었음을 의미함)

* 왜 이 작업이 필요한가

디버깅 작업시에 C언어 소스 코드와 실제 CPU의 동작 흐름이 일치하지 않으면 분석하기 굉장히 어렵다

그런데 컴파일러의 최적화 옵션이 활성화되면

프로그램이 여기 저기 왔다 갔다 하는 문제가 생겨서 분석하기가 매우 힘들다

이러한 이유 때문에 디버깅 시에는 반드시 -O0 옵션을 주도록 한다.

---

*디버거 작동

gdb debug

이와 같이 입력하면 디버거가 켜지면서 (gdb)창이 보일 것이다. 이제 main함수에 breakpoint를 걸어야 한다.

b main

위와 같이 입력하면 main 함수에서 멈춰달라는 의미인데 이제 r을 눌러서 프로그램을 구동하면 main 함수에 걸릴 것이다.

(단 C언어 기반으로 움직여서 선두의 어셈블리어를 지나치게 됨)

우선 현재 어느 위치에 있는지 파악하기 위해 디스어셈블리를 수행하도록 한다.

disas

이 명령어를 입력하면 현재 프로그램에 대한 디스어셈블리된 어셈블리어 코드와 화살표가 보일 것이다.

여기서 p/x $rip 라고 입력해보도록 한다

흥미롭게도 '=>' 가 나타내는 주소값과 p/x $rip가 출력하는 값이 같을 것이다.

앞서서 레지스터를 설명할 때 ip값은

다음에 실행할 명령어의 주소를 가르킨다고 했었다.

실행을 맨 처음으로 돌려야 하므로 disas해서 맨 첫 라인의 주소를 복사한다.

그리고 아래와 같이 breakpoint를 걸어준다

b *[복사한 주소]

그리고 다시 r을 눌러서 실행하면 '=>'가 맨 처음에 배치되어 있음을 확인할수 있다.

==========

* 16 진수

숫자	16 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

* 진수 시스템

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 - 역시 컴퓨터가 씀

? 2 진수랑 16 진수의 목적이 일치하는 것 아니냐 ?

라는 질문이 나올 수 있음

0101010101 2 진수

0x155 16 진수

10101010101010101010 2 진수

0x2aaaaa 16 진수

뭐가 보기 편한가요 ?

10101010101010010101010101010101001010101010

보기만 해도 토할 것 같아요.

즉 컴퓨터가 기계어를 사용하면서도

인간이 상대적으로 쉽게 볼 수 있는

16 진수 시스템을 채택한 용도로서 16 진수가 사용됨

(컴퓨터는 2 진수만을 쓰는 것임)

즉, 16 진수의 목적은

컴퓨터를 배운 사람과 기계의 혼용어라고 보면됨

* 2 진수, 16 진수를 어떻게 변환하는가 ?

144 이녀석을 2 진수로 쉽게 만드는 방법

일단 자릿수를 생각해야 한다.

먼저 144 를 10 진수 개념에서 분해해보자!

1 x 10^2 + 4 x 10^1 + 4 x 10^0

가만 보면 10 진수 시스템에서는

곱하는 부분에 10 이 계속 곱해지고 있음을 볼 수 있다.

그리고 승수로 붙는 곳에 자릿수 - 1 이

배치되고 있음을 볼 수 있다.

그렇다면 2 진수도 비슷하게 생각해보면 되지 않을까 ?

먼저 2 진수의 자릿수를 쭉 적는다.

7	6	5	4	3	2	1
	0					
128	64	32	16	8	4	2
	1					
2^7	2^6	2^5	2^4	2^3	2^2	
	2^1	2^0				

위와 같은 형식을 가제 한다면 아래와 같이 적으면 된다.

144 = 128 + 16 이다.

위 색인에서 7 번째에 1 을 셋팅하고

4 번째에 1 을 셋팅하면 아래와 같이 된다.

1001 0000

이것이 144 의 2 진수 변환에 해당한다.

그렇다면 정말로 이게 10 진수 144 가

맞는지 확인할 필요가 있다.

10 진수에 적용한대로 동일한 계산을 적용한다.

1 x 2^7 + 1 x 2^4 = 128 + 16 = 144

즉 10 진수 144 가 2 진수 1001 0000 과 같음이 입증되었다.

여기서 16 진수로 바꾸는 작업은 훨씬 쉽다.

16 진수는 한 자리에 16 개가 온다.

먼저 2 진수 자리 1 개를 생각해보자

0, 1 2 개

다음으로 2 진수 2 자리를 생각해본다.

00, 01, 10, 11 4 개

2 진수 3 자리

000, 001, 010, 011, 100, 101, 110, 111 8 개

2 진수 4 자리면 ? 16 개

그래서 16 진수 변환을 수행할 때 4 자리씩 끊어치면 빠르다.

1001 0000 은 결국 0x90 이 된다.

확인은 10 진수 분해와 동일하게 하면 된다.

16^1	16^0
9	0

16^1 x 9 = 144

ex) 10 진수 33 을 2 진수 및 16 진수로 표기해보자.

33 = 32 + 1

32 16 8 4 2 1

1 0 0 0 0 1

10 0001

8421 8421

0010 0001

0x2 1

0x21 => 2 x 16^1 + 1 x 16^0 = 33

ex) 10 진수 2568 을 2 진수 및 16 진수로 표기해보자.

2^10 = 1024

2^11 = 2048

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

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

0xA08 => A x 16^2 + 8 x 16^0 = 256 x 10 + 1 x 8 = 2568

ex) 0x48932110 을 2 진수로 변환해보자.

16 진수 1 자리가 2 진수 4 자리라는 것을 기억하고 풀면됨

8421 8421 8421 8421 8421 8421
8421 8421

0100 1000 1001 0011 0010 0001
0001 0000

0100 1000 1001 0011 0010 0001 0001 0000(2)

---

* 단위: 비트

2 진수(0, 1)로 표기할 수 있는 하나의 단위

2 진수 1 자리가 결국 1 bit(비트)를 의미함

1 bit 가 8 개 모이면 8 bit 이것을 1 byte 라고함

2^10 byte = 1 KB

2^10 KB = 1 MB

2^10 MB = 1 GB

32 bit = 2^32 = 4GB = 2^10 x 2^10 x 2^10 x 2^2

= 2^(10 + 10 + 10 + 2)

0 ~ 0xffff ffff 16 진수 8 자리

16 진수는 16 개를 표현하므로 1 비트가 4 개 있으면 됨

16 진수 1 자리는 결국 4 비트로 표현됨

32 비트는 몇 개의 16 진수 자리로 구성되는가 ?

32 / 4 = 8 개

0x~~~~~~~~~~ 64 비트는 16 개

64 비트에서 16 진수 16 개에 모두 f 를 채우면 표현할 수 있는 최대값

8 비트 시스템을 생각해서 문제를 좀 더 단순화시켜보자!

0 ~ 0xff

표현할 수 있는 갯수는 2^8 = 256

256 128 64 32 16 8 4 2 1

1 0 0 0 0 0 0 0 0

0x100 - 1 = 0xff

* 포인터의 크기는 ?

8 비트 시스템의 경우 1 byte

16 비트는 2 byte

32 비트는 4 byte

64 비트는 8 byte

- 왜 그럴까 ?

컴퓨터의 산술 연산이 ALU 에 의존적이기 때문이다.

ALU 의 연산은 범용 레지스터에 종속적이고

컴퓨터가 64 비트라는 의미는 이들이 64 비트로 구성되었음을 의미한다.

변수의 정의는 메모리에 정보를 저장하는 공간이였다.

포인터의 정의는 메모리에 주소를 저장하는 공간이다.

그렇다면 64 비트로 표현할 수 있는 최대값 또한 저장할 수 있어야한다.

포인터의 크기가 작다면 이 주소를 표현할 방법이 없기 때문에

최대치인 64 비트(8 byte) 가 포인터의 크기가 된 것이다.

실제로 확인을 해보자!

지금 띄워놓은 터미널창을 우클릭한다.

New Terminal 이 보일텐데 클릭하면 새로운 터미널이 나타난다.

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;
}
```

결과가 전부 8 이 나오는 것을 볼 수 있을 것이다.

포인터의 크기

16bit -> 2byte

32bit -> 4byte

64bit -> 8byte

위 내용을 배우는 이유는

스택의 동작 과정이 포인터 베이스이기 때문이다

또한 모든 컴퓨터의 동작과정이 이 포인터 베이스로 동작하게된다.

==========

* 어셈블리어 코드 분석 준비

먼저 디버거를 작동 시켜서 앞서 수행했던 push rbp 쪽에 화살표를 오게하는 일련의 과정을 진행하여 어셈블리어 코드 분석을 준비한다.

1. vi [파일이름.c] >>> insert 키를 누르고 >>> 다음 코드 작성하기

```
#include <stdio.h>

int myfunc(int num)

{
	return num + 3;
}

int main(void)

{
	int num = 3, res;
	res = myfunc(num);
	printf("res = %d₩n", res);
	return 0;
}
```

2. 작성을 완료하였다면, gdb 를 사용할 수 있도록 컴파일하기

gcc -g -o [파일의별칭아무거나] [파일이름.c]

3. 컴파일이 성공하였다면, gdb 소환

gdb [파일의별칭설정한것]

4. gdb 소환에 성공하였다면, breakpoint 을 main 으로 잡기

b main

5. 실행하기

r

6. 디버깅 전체보기

disas

그러면 현재 화살표가 가리키고 있는 곳이 breakpoint 로 잡혀있는 것을 확인할 수 있다.

7. 디버깅을 첫줄부터 시도해보기 위해 맨 윗줄의 0x0000000000400534 주소 복사

8. breakpoint 추가하기 (주소 앞에 *붙음)

b *0x0000000000400534

9. breakpoint 가 잘 추가됐는지 확인해보기

info b

(breakpoint가 2개 나옴)

10. breakpoint가 추가 된 것을 확인했다면, 다시 실행하기(재시작)

r

11. 다시 실행 후 현재 화살표가 가리키고 있는 곳이 첫 줄임을 확인하자

disas

12. 여기까지 완료했다면 디버깅을 한 줄씩 해보면서,

실제 각 레지스터의 역할과 데이터가 메모리상에서 어떻게 처리되는지를 확인해 볼 수 있다.

(어셈블리어 코드 분석 준비 완료)

---

* 어셈블리어 코드 분석시 참고

ax: 함수의 return 값을 저장함

cx: 무언가를 반복하고자 할 때 사용

bp: 스택의 기준점

sp: 스택의 최상위점

bp(기준점) sp(최상위점) 사이가 Stack

ip: 다음에 실행할 명령어의 주소

ALU 비트 (ax cx dx bx ...)

ALU비트는 운영체제 bit에 따라 맨 앞에 e 또는 r이 붙기도 한다.

Ex) 16bit -> ax / 32bit -> eax / 64bit -> rax

16bit -> sp / 32bit -> esp / 64bit -> rsp

어셈블리어 단위로 1줄 진행:	si
C언어 단위로 1줄 진행하기:	s
메모리 상태 검사:	x
변수를 16진수로 출력	p/x [변수]
함수 호출(push + jump)	call

리눅스 gdb 명령어 정리 링크

http://egloos.zum.com/psyoblade/v/2653919

PUSH A : A를 Stack에 넣는다.

POP [레지스터] : Stack에서 값을 꺼내 [레지스터]에 넣는다.

자주쓰는 어셈블리 명령어 정리 링크

http://securityfactory.tistory.com/153

----스택 프레임 생성 프롤로그----

push %rbp

mov %rsp, %rbp

----스택 프레임 해제 에필로그----

pop %rbp

retq

---

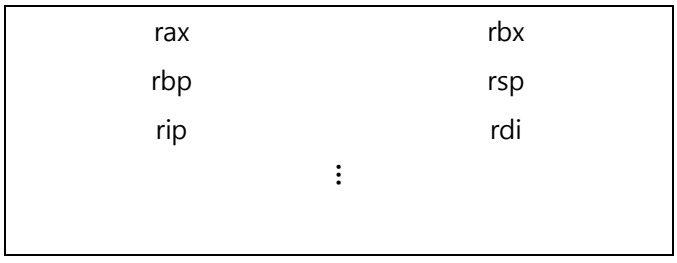
* 어셈블리어 코드 분석의 예

분석을 위한 코드 작성

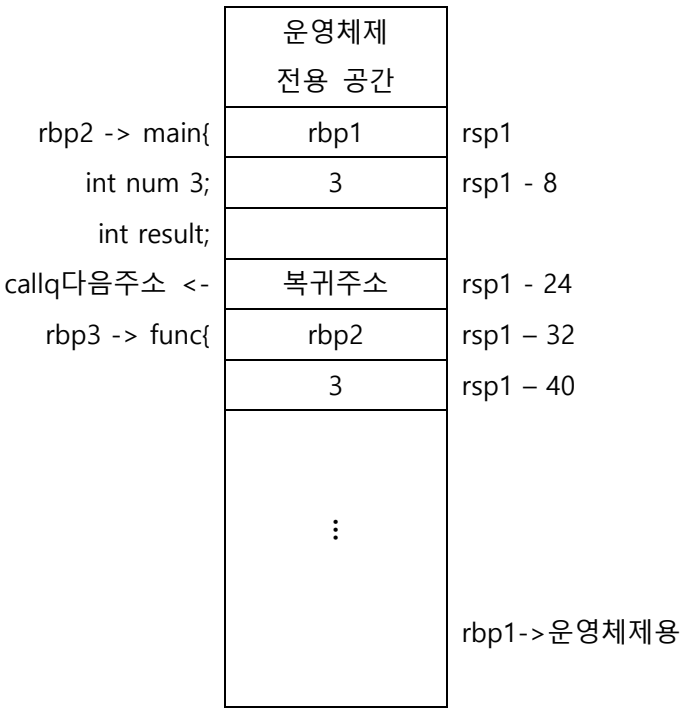
```
#include <stdio.h>
int myfunc(int num)
{
        return num + 3;
}
int main(void)
{
        int num = 3, res;
        res = myfunc(num);
        printf("res = %d₩n", res);
        return 0;
}
```

어셈블리어로 변환

```
Dump of assembler code for function main:
0x0000000000400535 <+0>: push   %rbp
0x0000000000400536 <+1>: mov    %rsp,%rbp
0x0000000000400539 <+4>: sub    $0x10,%rsp
0x000000000040053d <+8>: movl   $0x3,-0x8(%rbp)
0x0000000000400544<+15>: mov    -0x8(%rbp),%eax
0x0000000000400547<+18>: mov    %eax,%edi
0x0000000000400549<+20>:callq 0x400526 <myfunc>
0x000000000040054e<+25>: mov    %eax,-0x4(%rbp)
0x0000000000400551<+28>: mov    -0x4(%rbp),%eax
0x0000000000400554<+31>: mov    %eax,%esi
0x0000000000400556<+33>: mov    $0x4005f4,%edi
0x000000000040055b<+38>: mov    $0x0,%eax
0x0000000000400560<+43>:    callq   0x400400
<printf@plt>
   0x0000000000400565 <+48>:   mov    $0x0,%eax
   0x000000000040056a <+53>:   leaveq
   0x000000000040056b <+54>:   retq
End of assembler dump.
```



64bit CPU



64bit 메모리

어셈블리어 분석 참고 링크

http://yunreka.tistory.com/6?category=601357

어셈블리어 분석

(gdb) x $rsp

0x7fffffffdcb8: 0xf7a2d830

-> (운영체제에서 설정한 rsp의 주소 값): (운영체제에서 설정한 rsp의 주소값에 있는 데이터 값)

// main() 코드랑 상관없는 데이터

(gdb) x $rbp (=rbp1)

0x400570 <__libc_csu_init>: 0x56415741

-> (운영체제에서 설정한 rbp1의 주소 값): (운영체제에서 설정한 rbp1의 주소값에 있는 데이터 값)

// main() 코드랑 상관없는 데이터

(gdb) si

0x0000000000400536 11 {

(바로 아래 기계어 코드 한줄 실행)

->0x0000000000400535 <+0>: push %rbp

-> %rbp 주소에 있는 값을 Stack의 최상위(rsp주소의 데이터 값)에 밀어넣는다(push)

(gdb) x $rsp

0x7fffffffdcb0: 0x00400570

-> (운영체제에서 설정한 rsp의 주소 값): (운영체제에서 설정한 rbp1의 주소 값)

//코드가 끝났을 때 rbp 주소 값을 코드 시작 전 초기 rbp1 주소 값으로 돌아가기 위해 데이터 저장

(gdb) x $rbp (=rbp1)

0x400570 <__libc_csu_init>: 0x56415741

-> (운영체제에서 설정한 rbp1의 주소 값): (운영체제에서 설정한 rbp1의 주소값에 있는 데이터 값)

//코드랑 상관없는 데이터

(gdb) si

0x0000000000400539 11 {

(바로 아래 기계어 코드 한줄 실행)

->0x0000000000400536 <+1>: mov %rsp,%rbp

->rsp의 값을 rbp에 넣는다(mov)

//좌측의 레지스터 정보를 우측 레지스터로 복사함

(gdb) x $rsp

0x7fffffffdcb0: 0x00400570

-> (운영체제에서 설정한 rsp의 주소 값 = 코드 Main 함수의 Stack을 쌓기 시작할 메모리 주소의 위치): (운영체제에서 설정한 rbp1의 주소 값)

(gdb) x $rbp (=rbp2)

0x7fffffffdcb0: 0x00400570

-> (운영체제에서 설정한 rsp의 주소 값 = 코드 Main 함수의 Stack을 쌓기 시작할 메모리 주소의 위치): (rbp1의 주소 값)

-> 코드의 Stack을 쌓을 준비작업으로 rbp의 주소 값을 rsp의 주소 값으로 복사하여 같은 값으로 만든다. (rbp1 -> rbp2 == rsp)

(gdb) si

Breakpoint 1, main () at func1.c:12

12 int num = 3, res;

(바로 아래 기계어 코드 한줄 실행)

-> 0x0000000000400539 <+4>: sub $0x10,%rsp

->rsp의 주소 값에서 0x10을 마이너스(sub)하고 다시 rsp값에 넣는다. (0x10byte = 16byte의 공간을 임시로 확보한다)

(gdb) x $rsp

0x7fffffffdca0: 0xffffdd90

-> (Main함수의 Stack을 쌓기 시작할 메모리 주소의 위치 -0x10 == rbp2 - 0x10): (쓰레기 값)

(gdb) x $rbp (=rbp2)

0x7fffffffdcb0: 0x00400570

-> (Main함수의 Stack을 쌓기 시작할 메모리 주소의 위치 = rbp2): (rbp1의 주소 값)

(gdb) si

13 res = myfunc(num);

(바로 아래 기계어 코드 한줄 실행)

->0x000000000040053d <+8>:movl $0x3,-0x8(%rbp)

->rbp로 부터 0x8 byte 공간을 할당하고(-음수) 데이터 0x3 값을 그 위치에 넣는다

(gdb) x $rsp

0x7fffffffdca0: 0xffffdd90

-> (rbp2 - 0x10): (쓰레기 값)

(gdb) x $rbp

0x7fffffffdcb0: 0x00400570

-> (rbp2): (rbp1의 주소 값)

(gdb) x $rbp-0x8

0x7fffffffdca8: 0x00000003

-> (rbp2 - 0x10): (데이터 0x3 값)

(gdb) si

0x0000000000400547 13 res = myfunc(num);

(바로 아래 기계어 코드 한줄 실행)

->0x0000000000400544 <+15>:mov -0x8(%rbp),%eax

->rbp로 부터 0x8 byte 아래의 주소에 저장된 데이터 값을 CPU에 있는 eax레지스터에 넣는다

->(?ax: 주로 산술 연산에 활용하며 모든 return 값은 ax레지스터에 저장됨)