파이썬 – HW4

임베디드스쿨1기

Lv1과정

2020. 08. 17

강경수

1. GDB분석

```
 1 int sum(int c,int d)
 2 {
 3     return c + d;
 4 }
 5
 6 int main(void)
 7 {
 8     int a = 1;
 9     int b = 2;
10     int d = 0;
11     d = sum(a,b);
12     return 0;
13 }
14
15
```

```
> 0x0000555555555141 <+0>:     endbr64
  0x0000555555555145 <+4>:     push   %rbp
  0x0000555555555146 <+5>:     mov    %rsp,%rbp
  0x0000555555555149 <+8>:     sub    $0x10,%rsp
  0x000055555555514d <+12>:    movl   $0x1,-0xc(%rbp)
  0x0000555555555154 <+19>:    movl   $0x2,-0x8(%rbp)
  0x000055555555515b <+26>:    movl   $0x0,-0x4(%rbp)
  0x0000555555555162 <+33>:    mov    -0x8(%rbp),%edx
  0x0000555555555165 <+36>:    mov    -0xc(%rbp),%eax
  0x0000555555555168 <+39>:    mov    %edx,%esi
  0x000055555555516a <+41>:    mov    %eax,%edi
  0x000055555555516c <+43>:    callq  0x555555555129 <sum>
  0x0000555555555171 <+48>:    mov    %eax,-0x4(%rbp)
  0x0000555555555174 <+51>:    mov    $0x0,%eax
  0x0000555555555179 <+56>:    leaveq
  0x000055555555517a <+57>:    retq
```

```
=> 0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
   0x0000555555555131 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555555134 <+11>:    mov    %esi,-0x8(%rbp)
   0x0000555555555137 <+14>:    mov    -0x4(%rbp),%edx
   0x000055555555513a <+17>:    mov    -0x8(%rbp),%eax
   0x000055555555513d <+20>:    add    %edx,%eax
   0x000055555555513f <+22>:    pop    %rbp
   0x0000555555555140 <+23>:    retq
```

1. GDB분석

GDB실행 매뉴얼

1. gcc -g (파일명)
2. gdb a.out
3. b main -main 함수에서 break
4. b (함수명) 해당 함수에서 break
5. disas #disassembly로 어셈블리어 확인
6. list (l) c언어 레벨 코드 확인
7. p 변수이름 예시: p $rsp 메모리 값 확인
8. p/x 변수이름 16진수로 데이터 확인
9. x 메로리 주소 메모리 안에 들어있는 값 확인
10. b 라인수 해당라인에서 break
11. r 프로그램 실행
12. si 어셈블리어 레벨에서 한줄씩 실행

1. GDB분석

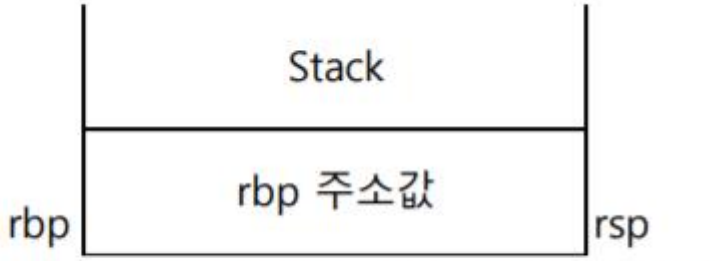
push %rbp

rbp의 값을 stack의 최상위에 밀어넣는다.
stack을 가리키는 rsp레지스터는 증가한다.
(포인터의 크기만큼. 즉 64bit기준 8byte)
rsp는 스택의 최상단을 가르킴.
즉 원래 rsp가 가리키던 값이 rbp크기 만큼 증가함.

mov %rsp,%rbp
복사 명령어

rsp의 값을 rbp에 넣어라
즉 rsp의 값과 rbp가 같아짐
이를 p $rbp , p $rsp명령어로 확인 가능함.

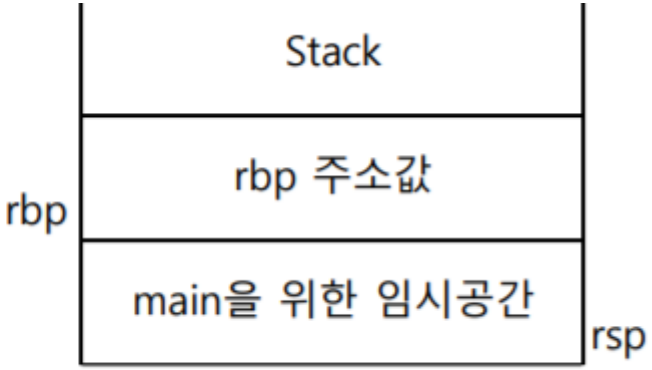
```
$3 = (void *) 0x7fffffffdf00
(gdb) p $rsp
$4 = (void *) 0x7fffffffdf00
```



rbp의 위치와 rsp의 위치가 같아짐

sub $0x10, %rsp

rsp에서 0x10을 빼서 rsp에 넣는다.
main함수를 사용하기 위한 공간을 할당
하기 위함.
0x10 = 16byte



rsp 이동함

1. GDB분석

mov -0x8(%rbp),%edx rbp로부터 8byte아래 쓰인 값을 edx로 복사
mov -0xc(%rbp),%eax rbp로부터 12byte아래 쓰인값은 eax로 복사

함수연산 에서 사용되는 레지스터 인듯함..
추가적인 공부 필요

mov %edx,%esi edx와 eax에 쓰인 값을 각각 esi edi에 복사
mov %eax,%edi

callq sum sum함수 호출

```
=> 0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
   0x0000555555555131 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555555134 <+11>:    mov    %esi,-0x8(%rbp)
   0x0000555555555137 <+14>:    mov    -0x4(%rbp),%edx
   0x000055555555513a <+17>:    mov    -0x8(%rbp),%eax
   0x000055555555513d <+20>:    add    %edx,%eax
   0x000055555555513f <+22>:    pop    %rbp
   0x0000555555555140 <+23>:    retq
```

sum함수 내에서도 마찬가지로 push %rbp로 시작한다
이때 메모리 위치를 확인하면 main함수보다 낮은 값이다.
즉 함수가 main함수보다 먼저 자리를 차지하고 있다.

함수가 불러 질때마다 push하여 stack에 공간을 차지한다.
완료되면 Pop하여 stack공간에서 소멸시키는 것을 확인 할 수 있다.
즉 재귀함수라고 하여 메모리 공간을 쪼개어 나가는 것이 아니라
낮은 메모리 방향으로 stack이 쌓여 나가게 됨을 확인 할 수 있다..

2. Review goto

1) Goto를 반드시 사용해야 하는 이유.

(1) 어떤 ERROR FLAG를 검출한다 가정할때, if문 혹은 for문과 같은 branch문 내에서 일단 파이프라인은 1차적으로 깨진상태이다
(2) 이런 상태에서 분기문을 만나면 ex: if errorflag==1 캐시미스가 발생할 확률은 더욱더 커지게 된다.
(3) 지속적으로 비교하여 검사하여야 하는 점이 성능 저하를 일으킨다.
(4) 코드가 지저분해진다. (break문을 두 번 써야 함.)

2) Python에서 Goto를 사용하는 방법

```
from goto import with_goto

@with_goto
def goto_test():
    for i in range(10):
        for j in range(10):
            #에러 발생 시점
            if (i == 3 and j == 2):
                goto .err_handler

            print("Item: {0}-{1}".format(i, j))

    label .err_handler
    print("Success to Error Handling!")
    print("Now check the reason!")

goto_test()
```

이때 반드시 유의해야 할 부분!

(1)Goto도 branch문이기 때문에 파이프라인을 깨지게 하며

(2)Python goto는 함수내에서만 사용이 가능하다.

3. Review Pipeline

1) 파이프라인

파이프라인 단계는 CPU마다 각각 다르다.

가장 기본적인 1.FETCH 2.DECODE 3.EXECUTE 3단계로 나누어질 수 있고
1. FETCH 2.DECODE 3.EXECUTE 4. STORED 4단계로 나누어 질 수 있다.
INTEL CPU의 경우 81단계로 쪼개어지기도 한다.

핵심은 각 단계를 쪼개어 한 클럭에 분업하여 일을 처리한다는 것이다.

가장 간단한 예시의 3단계를 설명하면

1. Instruction Cache로 실행시킬 Instruction들(어셈블리어)를 가져온다.
2. Instruction을 보고 이것이 레지스터 연산인지 메모리 연산인지 기타 다른 연산인지 명령어 자체를 해석한다.
3. Execute를 통해서 준비된 명령어(어셈블리어)를 실행한다.

3. Review pipeline

2) 슈퍼스칼라 프로세서에서 정말 병렬로 FETCH가 진행된다고 보면 된다.
이때에 Data Dependency이 없어야 가능하다. 즉

(1)c = a + b (2) c = a * b (3) c = d * e 이런 연산이 있을때
(1)과 (2)는 동시 수행이 불가능 하지만 (1)과 (3)은 동시수행이 가능하다.
(compiler 에서 c1 c2로 나누어 처리한다.)

FETCH	DECODE	EXCUTE	STORED	-	-	-
FETCH	DECODE	EXCUTE	STORED	-	-	-
-	FETCH	DECODE	EXCUTE	STORED	-	-
-	FETCH	DECODE	EXCUTE	STORED	-	-
-	-	FETCH	DECODE	EXCUTE	STORED	-
-	-	FETCH	DECODE	EXCUTE	STORED	-

(슈퍼스칼라)

4. Python Exceptions

4-1) Zero DivisionError
0으로 나눌 시

4-2) TypeError
Data Type이 맞지 않을 시

4-3) except
어떤 오류간에 무조건 발생 할 시

4-4) ArithmeticError
모든 산술 관련된 에러들 발생 시

```
def divide(a,b):
    return a/b

try:
    e = divide(5,2)

except ZeroDivisionError:
    print('Exception Occured')

except TypeError:
    print('숫자로 하세요')

except:
    print('Error발생')

else:
    print('Error가 없어요..')
```

```
Error가 없어요..
>>>
```

4-5) else
except TypeError or except Zero DivisionError에 검출되지 않을 시

4-6) finally
무조건 실행

4-7) as (변수) 발생하는 Error를 변수에 담음

```
def divide(a,b):
    return a/b

try:
    e = divide(5,0)

except ZeroDivisionError as e:
    print(e)
```

```
division by zero
>>>
```

4. Python Exceptions

4-8) Python,에서 file을 열고 읽은 후에는 예외에 상관없이
finally로 무조건 close()해준다.

4-9) raise (에러명)을 사용하여 사용자가 직접 에러를 일으킬 수 있다.

```
def user_error():
    raise TypeError

try:
    user_error()
except :
    print('error occur!')
```

```
error occur!
```

4-10) NegativeDivisionError
self.value 에 대해서 질문하기

4-11) assert 가정 설정문
assert 뒤의 조건이 TRUE가 아니면 ERROR를 발생시킨다.

```
def assert_f(a):
    assert a >=10, '10보다 작은값은 안돼요'

assert_f(5)
```

```
AssertionError: 10보다 작은값은 안돼요
```