시스템해킹 1 주차 과제: C 언어를 이용한 STACK 작동 양상 구현

2025350015 박규민

1. Stack 의 작동 양상에 관한 이해 과정.

-Cykor 에서 제공한 강의를 꼼꼼히 시청하며 메모리 구조와 Stack 의 개념을 이해하였으나, 스택의 작동 과정은 강의만으로 확실하게 이해가 되지 않아 구글링을 시도.

-여러 자료를 찾아본 끝에 스택의 작동 과정을 어느 정도 정리하였고, 이것을 기반으로 과제를 진행하게 됨.

-정리한 내용 요약:

<전제: ESP 는 스택 맨 위, EBP 는 SFP, EIP 는 반환주소값을 가리키는 포인터를 저장한다. 이들은 모두 포인터 레지스터이다>

<전제: ESP 는 스택 크기의 변화에 따라 움직인다>

==========함수 프롤로그==========

1. 호출할 함수의 전달인자들을 스택에 push 한다. (오른쪽에 있는 것부터 스택의 아래쪽, 즉 높은 메모리 주소로 간다!!

2. (EIP 에 저장되어 있던) 이전 함수의 반환 주소값을 스택에 push 한다.(이후 EIP 는 현재 함수의 반환 주소값을 가리키게 된다)(반환 주소값=Return Address=함수 호출이 끝나고 현재 함수로 돌아왔을 때 처리해야 할 것들)

3. 이전 함수에서 EBP 에 저장되어 있던 FP 값을 push 해 준다. 이것이 SFP 가 된다.

4. ebp 를 esp 처럼 stack 맨 위를 가리키도록 옮긴다. 결국 FP 의 위치가 옮겨지고, 함수 실행 준비가 완료된다. (아까 이전 함수의 FP 가 저장되었으니 이렇게 탈출하더라도 이전 함수로 돌아갈 수 있는 방법도 있다.)

5. esp 를 움직여 변수 등이 들어갈 칸을 확보한다. (함수호출완료)

==========함수 에필로그==========

5. 함수 실행이 끝나면, esp 를 3 번에서 ebp 가 있던 자리로 옮긴다. (각종 변수 등을 효율적으로 지워낸다. ESP 범위 밖에 있는 것은 정의되지 않은 것으로 간주되기 때문이다.)

6. SFP 를 pop 시킨 후 ebp 레지스터에 넣는다. 이렇게 되면 ebp 가, SFP 에 미리 확보된 이전 함수의 FP 를 저장하게 된다.

7.반환 주소값을 pop 시킨 후 eip 레지스터에 돌려주니, 다시 eip 레지스터가 이전 함수의 반환 주소값 포인터를 저장하게 된다. (이전 함수로 복귀 완료)

8. 마지막으로 ESP 를 옮김으로써, 매개변수까지 모두 지워버린다.

-참고한 자료: https://textbook.cs161.org/memory-safety/x86.html ,
https://cseweb.ucsd.edu/classes/sp05/cse127/Smash.htm, https://www.geeksforgeeks.org/stack-frame-in-computer-organization

2. 제시된 print_stack 함수 분석

```
void print_stack()
{
    if (SP == -1)
    {
        printf("Stack is empty.\n");
        return;
    }

    printf("====== Current Call Stack ======\n");

    for (int i = SP; i >= 0; i--)
    {
        if (call_stack[i] != -1)
            printf("%d : %s = %d", i, stack_info[i],
call_stack[i]
        else
            printf("%d : %s", i, stack_info[i]);//

        if (i == SP && i == FP)
            printf("    <=== [esp],[ebp]\n");
        else if (i == SP)
            printf("    <=== [esp]\n");
        else if (i == FP)
            printf("    <=== [ebp]\n");
        else
            printf("\n");
    }

printf("================================\n\n"
);
```

SP=-1 은 존재하지 않는 곳을 가리키고 있다. 따라서 이 경우 스택이 비어있는 것이다.

이 함수가 스택 전체를 출력할 수 있는 이유는, 윗줄부터 한 칸씩 내려오며 각각 1 차원과 문자열 배열인 call_stack 과 stack_info 를 차례대로 출력하기 때문이다.

Call_stack 의 값이 -1 일 경우,그 값은 없는 것으로 간주하고 출력하지 않는다.(return address 출력 등에 유용하게 사용하자)

본인이 추가한 코드. 기존 코드에선 SP 와 FP 가 같을 경우 SP 가 FP 를 덮어쓰지만, 사실 FP 도 거기에 있다는 것을 보여준다.

3. 제시된 전역변수와 매크로 분석

```
#define STACK_SIZE 50
int     call_stack[STACK_SIZE];//여기부터 Line 1
char    stack_info[STACK_SIZE][20];
int SP = -1;
int FP = -1;
```

-Line 1: call_stack 이라는 이름의 1 차원 배열 선언.

-Line 2: stack_info 라는 이름의 2 차원 문자열 배열 선언. (문자열 길이는 20 자가 넘으면 안된다)

-call_stack 과 stack_info 모두 한 열의 길이가 50 을 넘어갈 수 없다.

-Line 4,5: SP 와 FP 의 기본값은 Stack 에 아무것도 없음을 나타내는 -1 이다.

*call_stack[1]이 맨 끝 값이면(스택에 원소 2 개)맨 위를 가리키는 포인터인 SP 는 2 가 된다.

call_stack[0]이 맨 끝 값이면(스택에 원소 1 개) 맨 위를 가리키는 포인터인 SP 도 0 이 된다. (C 언어는 0 부터 숫자를 센다)

즉 SP 는 원소개수-1 임을 추측할 수 있다.

4. push 와 pop 함수의 정의.

```
void push(int element, char *info)//이제부터 이와 같은 코드 박스에서 맨 위를 Line 1 이라 한다.
{
    call_stack[SP+1] = element;
    strcpy(stack_info[SP+1], info);
    SP++;
}
void pop(int array1[], char array2[][20])
{
    array1[SP] = array1[SP + 1];
    array2[-1][20] = array2[-1][20];
    SP--;
}
```

Line 1: push 와 pop 함수는, 각각 call_stack 에 넣어야할 정수와, stack_info 에 넣어야 하는 문자열을 다루는 함수이다. 반환 값은 없다. (굳이 포인터 변수 char *info 를 쓰는 이유는, 이를 이용해야만 문자열을 한 번에 추가할 수 있기 때문이다. 사용하지 않는다면 문자열의 각 글자를 하나하나 쳐야 한다)

Line 3~5: SP 의 작동 과정을 고려해보면, 항상 스택의 맨 위 구성 요소를 가리키고 있어야 한다. 그렇다면 새로운 요소를 추가할 때는 현재 SP 가 가리키는 위치에 추가해서는 안 되고, 그보다 한 칸 뒤의 위치에 추가한 후에 SP 의 값도 1 더 증가시키면 된다. (빨간 글씨)

Line 4(보충): char *info 는 문자열(배열)을 가리키는 포인터 변수이니, 이것이 가리키는 문자열을 stack_info 에 넣어야 한다. (아래는 이 문제를 해결하기 위해 사용해 본 방법들과 결과이다)

-stack_info[SP+1]=info : 그냥 stack_info [SP+1]자체가 info 가 가리키는 문자열을 가리키게 한다.즉, 배열 stack_info[] 에는 아무것도 들어가지 않아서 실패.

-이후 구글링을 통해 strcpy 함수에 관한 정보를 찾아내었다. strcpy(a,b) 는 b 가 문자열(배열)포인터라고 하더라도 b 가 가리키는 값 전체를 확보하여, a(a 에는 문자열 배열상에서의 특정 위치, 또는 문자열 포인터가 들어간다)에 붙여넣어 주는 함수라고 한다.

-strcpy(stack_info[SP+1],*info): 배열 포인터 변수 선언을 [타입] * [변수명]으로 하였을 때, 호출을 *[변수명]으로 하면 이는 포인터가 가리키는 값 대신 주소 자체를 가져오라는 것이며, '해당 배열의 첫 번째 요소를 주소로 판단하여 가져온다'는 것을 알게 되었다. 즉 이것 역시 passing argument makes pointer from integer without a cast 오류 발생하며 실패.

-strcpy(stack_info[SP+1], info): 성공. 아래는 이 문제를 해결하는데 참고하였던 자료들이다.

https://wikidocs.net/86260 , https://blog.naver.com/1stwook/30182859109

Line 7: pop 함수는 2 개의 배열을 매개변수로 받으며 반환값은 없다.

Line 9,10: pop 을 할 때는 배열 내의 맨 끝 값을 지워야 한다. 아래는 과제 당시 배열 내용을 완전히 학습하지 않았던 본인이 해당 코드를 수정한 과정이다. (약간 참고한 자료: https://c-for-dummies.com/blog/?p=6557)

-array2[SP]= "": 배열의 요소 수가 줄어드는 것이 아닌 요소 하나가 빈칸으로 바뀌는 것에 불과하므로, 해당 요소가 출력될 곳에 빈칸이 생겨 스택이 제대로 구현되지 않는다.

-array2[SP]=array2[-1]: 성공. -1 은 인덱싱에서 '해당 배열에 존재하지 않는 것' 취급하므로, 아예 요소 하나를 지워 버리는 효과가 있다.

Line 11: call_stack 에서는 하나의 값, stack_info 에서는 하나의 행을 지웠으니, 이제 SP 도 1 감소시켜도 된다.

5. 함수 프롤로그 구성.

```
void func1(int arg1, int arg2, int arg3)
{
    int var_1 = 100;
    char* first = "arg1", * second = "arg2", * third = "arg3", * var = "var_1", * SFP = "func1
SFP";
    push(arg3, third);
    push(arg2, second);
    push(arg1, first);
    push(-1, ret);
    push(-1, SFP);
    ebp = FP;
    SP++;
    push_sfp(var_1, var, 0);//(push_sfp 관련 설명은 마지막 슬라이드에)
    /*func1 의 스택 프레임 형성 (함수 프롤로그 + push)*/
    print_stack();
/*이후 func2 호출 후  함수 에필로그*/
```

Line 4: push 와 pop 함수의 매개변수는 포인터변수 이므로 arg1, arg2 를 비롯한 , 스택의 func1 부분에서 출력해야 할 문자열들을 가리키는 포인터변수들을 선언해 둘 필요가 있다.

Line 5~7: 스택에 데이터를 넣는 규칙에 따라 매개변수 중 가장 오른쪽에 있는 arg3 부터 차례대로 push 를 해준다.

-전역변수 ret 와 ebp 의 선언:

char *ret 는 문자열 Return Address 를 가리키는 포인터 변수이다. func2, func3 에서도 계속 사용해야 할 것이므로 미리 전역변수로 선언한다(위의 코드에는 선언 모습은 나타나지 않는다)

ebp 는, 구현된 스택에서 SFP 의 기능을 담당하는 정수 전역변수이다. FP 의 현재 위치를 저장하기도 하고, pop 시에 FP 를 원래 함수로 되돌리는 기능도 수행할 것이다.

Line 8~10: 먼저 문자열 “Return Address” 를 출력 후, SFP 는 스택에 나타나지 않는 main 함수에서의 FP 위치이므로 무시하기 위해 call_stack 에 -1 을 넣어주며, 현재 FP 의 위치를 ebp 변수에 저장한다.

Line 11: 상술한 스택의 작동 과정 에 의해, 한 개의 지역변수를 담기 위해 SP 를 움직여 한 칸을 확보.

(이후 지역변수 넣는 과정에서 버그 발생. 후술할 push_sfp 함수 통해 해결하였음)

```
void func2(int arg1, int arg2)
{ // func2 의 스택 프레임 형성 (함수 프롤로그 + push)
    int var_2 = 200;
```

-Line 7 까지의 과정은 func1 과 같지만, Line 8 에서 -1 이 아닌, 앞서 func1 에서 저장해 온 SFP 를 넣어야 한다. 이는 정수 전역변수 ebp 가 저장하고 있으니 call_stack 에 push 함수를 사용해 넣을 수 있다. 이후의 과정은 func1 과 동일하다.

-func3 은 func2 에서 매개변수와 지역변수의 개수와 이름만 바꿔서 하면 된다.

6. 함수 에필로그 구성.

```
/*func2 의 에필로그를 예시로 들자.*/
// func2 의 스택 프레임 제거 (함수 에필로그 + pop)
 SP--;
```

-Line 3: 일단 func2 의 지역변수 var_2 를 스택에서 지워낸다. 상술한 스택의 작동 과정을 참고하면, 이 단계는 굳이 POP 함수를 쓰지 않고도 SP 를 움직이는 것만으로 지워낼 수 있다.

-Line 4~5: 이전 함수의 FP 를 가진(함수 프롤로그에서 call_stack 안에 넣어줌) 행을 지워야 하므로, ebp 레지스터가 이전 함수로 돌아갈 수 있게, 이전 함수의 FP 를 저장한다. (그 후 pop 함수로 삭제한다)

-Line 6: 이전 함수의 FP 로 ebp 레지스터(←==[ebp])의 위치를 옮긴다. 스택의 작동 과정을 보면 이 과정이 Return Address 를 pop 시키기 전보다 먼저이다)

-Line 7~8: Return Address 를 pop 시키고 전달인자(매개변수)들 역시 SP 의 위치를 움직여 지운다.

func1, func3 역시 이와 같은 방법으로 하면 된다.

func1 에서 이전 함수의 FP 를 저장하고 ebp 레지스터 위치를 옮기는 부분은, 어차피 func1 의 SFP 에 앞서 push 한 -1 이 저장되어 있으니 ebp 레지스터(←=ebp)는 사라질 것이다.

7.push_sfp 함수를 통한 push 과정에서의 버그 해결

```
void push_sfp(int element, char *info1,int i)
   {
```

-5. 함수 프롤로그 구성 에서 push_sfp 대신 push 함수를 사용한 결과, 다음과 같이 출력되는 문제 발생

var_1=100

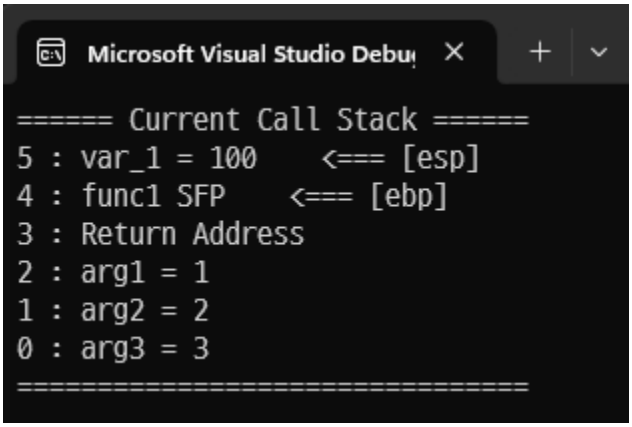
=0

func1 SFP

-이는 var_1 을 위한 행을 확보하고 SP 를 이동시켰음에도, 해당 칸에 var_1 을 넣는 대신 확보한 행보다 더 위에 새로 행을 만들어서 발생한 문제. =0 이 출력된 이유는 저것이 빈칸 발생 시 출력 기본값이기 때문인 것으로 추정.

-기본적으로는 push 함수와 유사하나, SP 를 이동시켜 이미 확보한 행에 넣는 것이므로 끝부분에 SP++가 필요없으며, Line 2~3 의 i 는 해당 함수(func1, func2, func3 의 지역변수의 개수를 나타낸다. 그만큼 SP 가 위로 이동해 행을 확보해 두었을 것이며, 그 아래에 push 를 해야하기 때문이다.

8. 실행 결과



func1 프롤로그가 끝난 후의 모습.

```
====== Current Call Stack ======
10 : var_2 = 200    <=== [esp]
9 : func2 SFP = 4    <=== [ebp]
8 : Return Address
7 : arg1 = 11
6 : arg2 = 13
5 : var_1 = 100
4 : func1 SFP
3 : Return Address
2 : arg1 = 1
1 : arg2 = 2
0 : arg3 = 3
================================
```

func2 프롤로그까지 끝난 후의 모습.

```
====== Current Call Stack ======
15 : var_3 = 300    <=== [esp]
14 : var_4 = 400
13 : func3 SFP = 9    <=== [ebp]
12 : Return Address
11 : arg1 = 77
10 : var_2 = 200
9 : func2 SFP = 4
8 : Return Address
7 : arg1 = 11
6 : arg2 = 13
5 : var_1 = 100
4 : func1 SFP
3 : Return Address
2 : arg1 = 1
1 : arg2 = 2
0 : arg3 = 3
================================
```

func3 프롤로그까지 끝난 후의 모습.

이제 LIFO 에 따라 func3 부터 pop 될 것이다.

```
    SP-=2;
    print_stack();//매개변수 제거 직후
    ebp = call_stack[FP];
    pop(call_stack, stack_info);
    FP = ebp;
    print_stack();//ebp 위치이동 직후
    pop(call_stack, stack_info);
    SP--;
print_stack();
```

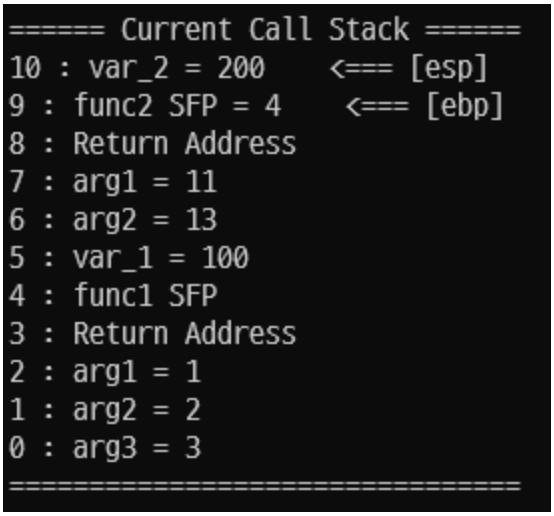
본인은 스택의 실행과정을 좀 더 세분화해서 보여주기 위해, 기존에 없던 print_stack 함수 호출을 2 번 더 추가하였다.

```
====== Current Call Stack ======
13 : func3 SFP = 9    <=== [esp],[ebp]
12 : Return Address
11 : arg1 = 77
10 : var_2 = 200
9 : func2 SFP = 4
8 : Return Address
7 : arg1 = 11
6 : arg2 = 13
5 : var_1 = 100
4 : func1 SFP
3 : Return Address
2 : arg1 = 1
1 : arg2 = 2
0 : arg3 = 3
================================
```

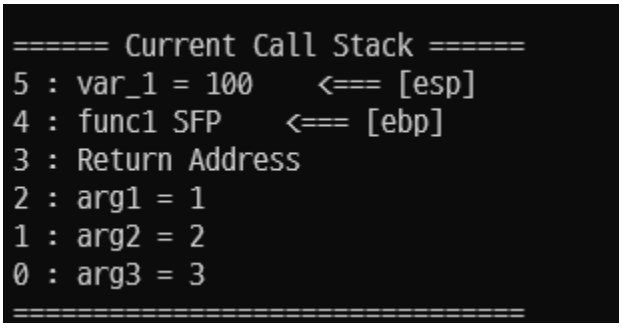
SP 를 이동시켜 func3 의 매개변수 2 개를 스택에서 제거한 후의 모습

```
====== Current Call Stack ======
12 : Return Address    <=== [esp]
11 : arg1 = 77
10 : var_2 = 200
9 : func2 SFP = 4    <=== [ebp]
8 : Return Address
7 : arg1 = 11
6 : arg2 = 13
5 : var_1 = 100
4 : func1 SFP
3 : Return Address
2 : arg1 = 1
1 : arg2 = 2
0 : arg3 = 3
================================
```

func3 의 Return Address 를 pop 하기 전 ebp 가 미리 저장해 둔 func2 의 FP 로 옮겨간 모습.



func3 전체를 pop 완료한 후의 모습.



func2 까지 pop 이 완료되고 func1 만 남은 모습.

SP 가 다시 -1 이 되어 Stack is empty 가 출력된