2025_1주차_과제

스마트보안학부 2024350034 한주영

들어가기 전

터미널(명령어 입력창)

지금까지 파일을 다루거나 프로그램을 실행할 때는 Run 버튼을 누르는 식의 마우스를 사용하는 환경이 익숙했기에 터미널을 다루는 방법을 몰랐다. 이전에 'C언어및실습' 과목을 들을 때도 Visual Studio를 사용했을 뿐더러 Run 버튼들은 직접 클릭해서 사용했었다. 그렇기에 강의에서 gcc와 같은 명령어들을 사용할 때 강의 내용을 바로 이해할 수 없었다. 강의 내용을 이해하는 데에 있어서 터미널을 다루는 방법을 추가로 공부해야 했고 이에 관한 내용을 과제 작성에 앞서 정리하였다.

아래는 터미널 주요 명령어를 정리한 표이다.

명령어	설명
ls	현재 파일과 폴더 목록을 보여줌
cd 폴더명	다른 폴더로 이동
cd ..	상위 폴더로 이동
pwd	현재 폴더의 경로를 확인
mkdir 폴더명	새로운 폴더를 만듦
rm 파일명	파일 삭제
touch 파일명	새로운 빈 파일 생성
cat	파일 내용 보여주기
clear	터미널 화면을 깨끗하게 지움
gcc 파일명.c -o 실행파일명	C 컴파일러. 코드를 기계어로 바꿔줌.

과제 핵심 개념 정리

1주차 강의 내용은 시스템 해킹을 위해 알아야 할 메모리 구조, 레지스터, 그리고 스택 동작 원리에 관한 내용이었다. C언어로 작성된 프로그램이 실제로 메모리에서 어떻게 실행되고 함수 호출 시 어떤 일이 벌어지는지를 이해할 수 있었다. 그 중 이번 과제에 사용되는 중요 개념들을 아래에 정리하였다.

레지스터

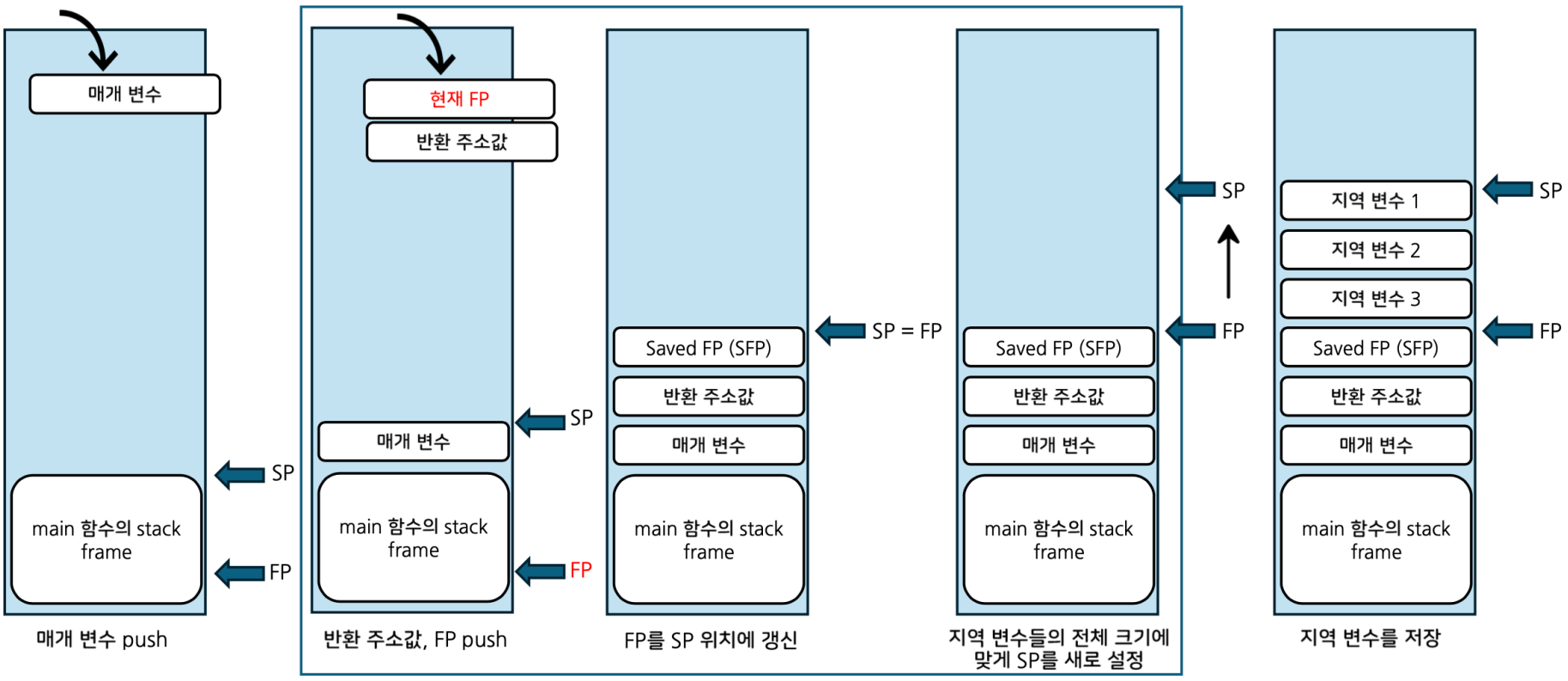
CPU 내부에 있는 초고속 데이터 저장 공간이다. 연산, 메모리 주소 저장, 함수 호출 시 스택 프레임 관리 등을 할 수 있다. ESP는 현재 stack의 최상단 주소를 가리키고 EBP는 함수의 기준 위치를 가리킨다.

SP와 FP

SP는 Stack Pointer로, 스택의 현재 최상단 주소를 가리키고 FP는 Frame Pointer로, 현재 함수의 시작 지점을 가리킨다.

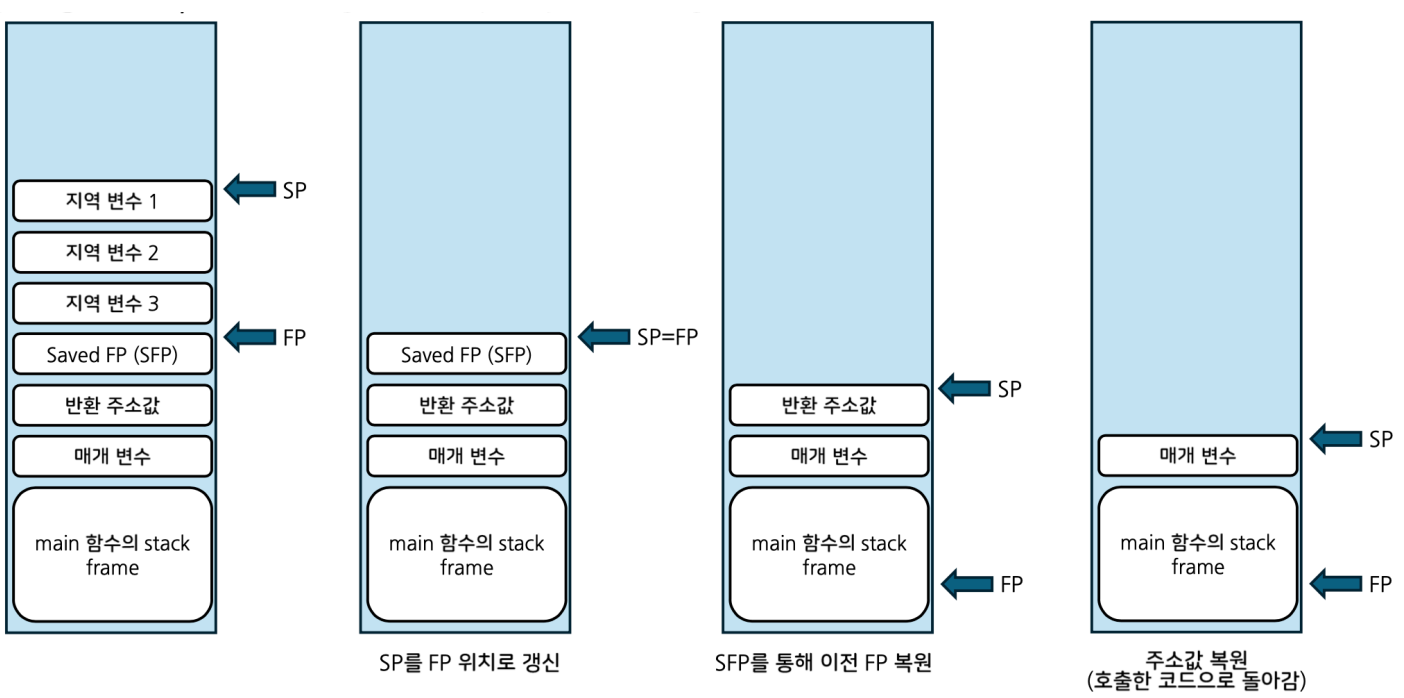
함수 프롤로그

함수 프롤로그는 함수가 실행될 때 스택 프레임을 설정하는 과정이다.



함수 에필로그

함수 에필로그는 함수가 종료될 때 호출 이전 상태로 스택을 복원하는 과정이다. 즉, 쉽게 말해 함수가 끝날 때 스택을 어떻게 정리하느냐를 말한다.



1단계	2단계	3단계	4단계
함수 실행 도중 상태	SP를 FP 위치로 이동	FP 복원	스택이 복원 후, 함수 호출 한 곳으로 되돌아감

과제 작성 내용

push(), pop() 작성

중복되는 코드를 줄이고 필요할 때마다 쉽게 사용하기 위하여 push와 pop에 대해 아래와 같이 코드를 작성하여 func1, func2, func3에 상황에 맞게 활용하였다. 이를 위해 `srtcpy`를 사용하였다.

`srtcpy(dest, src)`는 복사할 원본 문자열인 source와 복사한 내용을 저장할 목적지인 destination으로 구성된 함수이다. 쉽게 말하면 src에 있는 글자를 dest에 그대로 복사하라는 뜻이다. 이를 활용한 push와 pop 함수는 전역에 적용되어야 하므로 전역 영역에 선언하였다.

```
// push, pop 함수
void push(int value, const char* desc) {
    SP++;
    call_stack[SP] = value;
    strcpy(stack_info[SP], desc);
}
void pop(int pop_num) {
    SP -= pop_num;
}
```

push 함수는 스택 포인터를 증가시키고, 해당 위치에 값과 그에 대한 설명을 저장하여 스택 프레임을 구성하도록 하였다. pop 함수는 지정된 수만큼 스택 포인터를 감소시켜, 함수 종료 시 스택 프레임을 한 번에 제거할 수 있도록 하였다.

함수 프롤로그 작성

함수 프롤로그는 Return Address, SFP, 매개변수들, 지역변수들을 순서로 하여 그 다음에 새로운 FP를 설정하는 것으로 순서를 정해 구성했다. 프롤로그를 작성하는 데에도 `strcpy`를 활용하였다.

```
// 함수 프롤로그
void prologue(int args[], int num_args, int locals[], int num_locals) {
    // 반환 주소 저장
    push(-1, "Return Address");
    // 이전 FP 저장 & 새 FP 설정
    push(FP, "Saved FP");
    FP = SP;
    // 매개변수 push_args
    for (int i = 0; i < num_args; i++) {
        char label[7];
        strcpy(label, "arg_");
        label[4] = '1' + i;
        label[5] = '\0';
        push(args[i], label);
    }
    // 지역변수 push_local
    for (int i = 0; i < num_locals; i++) {
        char label[6];
        strcpy(label, "var_");
        label[4] = '1' + i;
        label[5] = '\0';
        push(locals[i], label);
    }
}
```

우선, 함수가 끝나고 돌아갈 곳을 기억하기 위해 반환 주소를 저장했다. 그 다음에는 이전 FP를 저장하고 앞으로 이 함수의 프레임이 어디서 시작되는지 표시하기 위해 `FP = SP`로 하여 새 FP를 설정했다.

매개변수의 경우, var_1처럼 최대 5문자 그리고 NULL을 고려하여 char label을 7바이트로 할당했다. 기본 문자열 복사 시 먼저 arg를 넣어 label이 arg가 되게 했고 i가 0이면 1, 1이면 2가 되게 하기 위해 숫자 문자를 추가 했다. 문자열 끝은 \0로 표시해야 안전하기에 문자열 종결에 label[4]='\0'로 했다. 최종적으로 `push(args[i], lable)`을 통해 실제 매개변수 값과 `arg ?` 설명을 스택에 저장하도록 하였다.

지역변수의 경우, 매개변수와 비슷하게 작성하였다. 지역변수의 경우 var_1처럼 최대 5문자 그리고 NULL이 필요하여 char label을 6으로 설정하고 기존 문자열 복사를 strcpy(label, "var_"로 하여 label을 var_로 초기화했다. 매개변수와 비슷한 방식으로 label[4]='1'+i로 하여 숫자 문차를 추가하고 label[5]='\0'로 했다. 마지막에는 마찬가지로 `push(locals[i], label)`을 통해 지역변수 값과 `var_ ?` 설명을 스택에 저장할 수 있게 했다.

쉽게 정리하자면 반환 위치를 먼저 쌓고 FP를 저장한 뒤 새 FP를 설정하여 매개변수를 차례로 쌓고 지역변수도 차례로 쌓았다. 처음 배우는만큼 강의 영상에서 배운 매커니즘을 거의 비슷하게 적용했다.

함수 에필로그 작성

지역 변수를 제거하고 FP 복원을 한 후 호출자 상태를 복원하는 순서로 구성하였다.

```
// 함수 에필로그
void epilogue(int pop_num) {
    pop(pop_num);
    FP = call_stack[SP + 1];
}
```

pop_num개의 데이터를 스택에서 없애면 내부적으로 `SP -= pop_num`이 되어 스택의 맨 위를 pop_num만큼 내려가게 하도록 코드를 작성했다. 이로 인해 RET, SFP, 매개변수, 지역변수 등 이 함수가 쌓아 올린 모든 항목이 사라진 것처럼 될 수 있다.

그 뒤는 `FP = call_stack[FP+1];`를 통해 pop()으로 SP가 내려간 뒤에도 바로 그 아래 위치인 SP+1에 이전 함수의 FP가 저장되게 했다. 이 값을 꺼내어 다시 FP에 넣어줌으로써 함수 호출 전의 스택 기준점으로 돌아가게 했다.

func1 작성

```
void func1(int arg1, int arg2, int arg3)
{
    int var_1 = 100;

    // func1의 스택 프레임 형성 (함수 프롤로그 + push)
    int args[] = {arg1, arg2, arg3};
    int locals[] = {var_1};
    prologue(args, 3, locals, 1);
    print_stack();

    func2(11,13);

    // func2의 스택 프레임 제거 (함수 에필로그 + pop)
    epilogue(5);
    print_stack();
}
```

먼저 프롤로그를 통해 반환 주소, 이전 FP, 세 개의 매개변수와 지역변수 var_1을 스택에 차례로 쌓아 스택 프레임을 구성했다. 그 후 `print_stack()`로 현재 상태를 확인하고 `func2(11, 13)`을 호출하는 기존 코드를 두었다. 함수 호출 후에는 `epilogue(5)`로 쌓은 5개의 스택 항목을 제거하고 FP를 복원한 뒤 다시 한 번 스택 상태를 출력하게 구성하였다.

func2 작성

```
void func2(int arg1, int arg2)
{
    int var_2 = 200;

    // func2의 스택 프레임 형성 (함수 프롤로그 + push)
    int args[] = {arg1, arg2};
    int locals[] = {var_2};
    prologue(args, 2, locals, 1);
    print_stack();

    func3(77);

    // func3의 스택 프레임 제거 (함수 에필로그 + pop)
    epilogue(5);
    print_stack();
}
```

func1와 유사하게 프롤로그로 반환 주소, 이전 FP, 대신 이번에는 두 개의 매개변수와 지역 변수 var_2를 스택에 push하여 프레임을 만든 후에 `print_stack()`로 이를 출력하게 하였다. 이후 `func3(77)`을 호출하고 func3의 스택 프레임이 쌓이도록 하는 기존 코드를 두었다. 마지막에는 역

시나 `epilogue(5)`로 5개의 항목을 pop하고 FP를 복원하면서 스택을 정리하고 최종 상태를 출력하게 했다.

func3 작성

```
void func3(int arg1)
{
    int var_3 = 300;
    int var_4 = 400;

    // func3의 스택 프레임 형성 (함수 프롤로그 + push)
    int args[] = {arg1};
    int locals[] = {var_3, var_4};
    prologue(args, 1, locals, 2);
    print_stack();
}
```

프롤로그를 통해 반환 주소, 이전 FP, 하나의 매개변수와 두 개의 지역변수인 var_3와 var_4를 스택에 쌓고 `print_stack()`로 프레임을 출력했다. func3는 더 이상의 함수를 호출하지 않기 때문에 epilogue 작업은 호출한 func2 내에서 처리된다.

main 함수 작성

```
//main 함수에 관련된 stack frame은 구현하지 않아도 됩니다.
int main()
{
    func1(1, 2, 3);
    // func1의 스택 프레임 제거 (함수 에필로그 + pop)
    epilogue(6);
    print_stack();
    return 0;
}
```

`func1(1,2,3)`을 호출하여 가장 상위 스택 프레임을 생성하고 내부에서 func2, func3까지 순차적으로 실행되도록 한 후 epilogue(6)을 호출해 func1이 쌓은 반환 주소, SFP, 변수 세 개, 지역 변수 한 개를 제거하여 스택을 완전히 비우도록 했다. 마지막으로 `print_stack()`으로 스택이 비어 있음을 확인한 뒤 프로그램을 종료한다.

결과

최종 코드

```
/*  call_stack

    실제 시스템에서는 스택이 메모리에 저장되지만, 본 과제에서는 `int` 배열을 이용하여 메모리를 구현합니다.
    원래는 SFP와 Return Address에 실제 가상 메모리 주소가 들어가겠지만, 이번 과제에서는 -1로 대체합니다.

    int call_stack[]      : 실제 데이터(`int 값`) 또는 `-1` (메타데이터 구분용)을 저장하는 int 배열
    char stack_info[][]   : call_stack[]과 같은 위치(index)에 대한 설명을 저장하는 문자열 배열

    ============================call_stack 저장 규칙============================
    매개 변수 / 지역 변수를 push할 경우   : int 값 그대로
    Saved Frame Pointer 를 push할 경우  : call_stack에서의 index
    반환 주소값을 push할 경우       : -1
    ============================================================================


    ============================stack_info 저장 규칙============================
```

```
   매개 변수 / 지역 변수를 push할 경우        : 변수에 대한 설명
   Saved Frame Pointer 를 push할 경우  : 어떤 함수의 SFP인지
   반환 주소값을 push할 경우                : "Return Address"
   ================================================================
*/

// 스마트보안학부 2024350034 한주영

#include <stdio.h>
#include <string.h>
#define STACK_SIZE 50 // 최대 스택 크기

int     call_stack[STACK_SIZE];         // Call Stack을 저장하는 배열
char    stack_info[STACK_SIZE][20];     // Call Stack 요소에 대한 설명을 저장하는 배열

/*  SP (Stack Pointer), FP (Frame Pointer)

   SP는 현재 스택의 최상단 인덱스를 가리킵니다.
   스택이 비어있을 때 SP = -1, 하나가 쌓이면 `call_stack[0]` → SP = 0, `call_stack[1]` → SP = 1, ...

   FP는 현재 함수의 스택 프레임 포인터입니다.
   실행 중인 함수 스택 프레임의 sfp를 가리킵니다.
*/
int SP = -1;
int FP = -1;

// push, pop 함수
void push(int value, const char* desc) {
   SP++;
   call_stack[SP] = value;
   strcpy(stack_info[SP], desc);
}
void pop(int pop_num) {
   SP -= pop_num;
}

// 함수 프롤로그
void prologue(int args[], int num_args, int locals[], int num_locals) {
   // 반환 주소 저장
   push(-1, "Return Address");
   // 이전 FP 저장 & 새 FP 설정
   push(FP, "Saved FP");
   FP = SP;
   // 매개변수 push_args
   for (int i = 0; i < num_args; i++) {
      char label[7];
      strcpy(label, "arg_");
      label[4] = '1' + i;
      label[5] = '\0';
      push(args[i], label);
   }
   // 지역변수 push_local
   for (int i = 0; i < num_locals; i++) {
      char label[6];
      strcpy(label, "var_");
      label[4] = '1' + i;
      label[5] = '\0';
      push(locals[i], label);
```

```
    }
}
// 함수 에필로그
void epilogue(int pop_num) {
    pop(pop_num);
    FP = call_stack[SP + 1];
}

void func1(int arg1, int arg2, int arg3);
void func2(int arg1, int arg2);
void func3(int arg1);

/*
    현재 call_stack 전체를 출력합니다.
    해당 함수의 출력 결과들을 바탕으로 구현 완성도를 평가할 예정입니다.
*/
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)
            printf("    ⇐= [esp]\n");
        else if (i == FP)
            printf("    ⇐= [ebp]\n");
        else
            printf("\n");
    }
    printf("================================\n\n");
}


//func 내부는 자유롭게 추가해도 괜찮으나, 아래의 구조를 바꾸지는 마세요
void func1(int arg1, int arg2, int arg3)
{
    int var_1 = 100;

    // func1의 스택 프레임 형성 (함수 프롤로그 + push)
    int args[] = {arg1, arg2, arg3};
    int locals[] = {var_1};
    prologue(args, 3, locals, 1);
    print_stack();

    func2(11,13);

    // func2의 스택 프레임 제거 (함수 에필로그 + pop)
```

```
    epilogue(5);
    print_stack();
}


void func2(int arg1, int arg2)
{
    int var_2 = 200;

    // func2의 스택 프레임 형성 (함수 프롤로그 + push)
    int args[] = {arg1, arg2};
    int locals[] = {var_2};
    prologue(args, 2, locals, 1);
    print_stack();

    func3(77);

    // func3의 스택 프레임 제거 (함수 에필로그 + pop)
    epilogue(5);
    print_stack();
}


void func3(int arg1)
{
    int var_3 = 300;
    int var_4 = 400;

    // func3의 스택 프레임 형성 (함수 프롤로그 + push)
    int args[] = {arg1};
    int locals[] = {var_3, var_4};
    prologue(args, 1, locals, 2);
    print_stack();
}


//main 함수에 관련된 stack frame은 구현하지 않아도 됩니다.
int main()
{
    func1(1, 2, 3);
    // func1의 스택 프레임 제거 (함수 에필로그 + pop)
    epilogue(6);
    print_stack();
    return 0;
}
```

출력 결과

```
====== Current Call Stack ======
5 : var_1 = 100    <=== [esp]
4 : arg_3 = 3
3 : arg_2 = 2
2 : arg_1 = 1
1 : Saved FP    <=== [ebp]
0 : Return Address
================================

====== Current Call Stack ======
10 : var_1 = 200    <=== [esp]
9 : arg_2 = 13
8 : arg_1 = 11
7 : Saved FP = 1    <=== [ebp]
6 : Return Address
5 : var_1 = 100
4 : arg_3 = 3
3 : arg_2 = 2
2 : arg_1 = 1
1 : Saved FP
0 : Return Address
================================

====== Current Call Stack ======
15 : var_2 = 400    <=== [esp]
14 : var_1 = 300
13 : arg_1 = 77
12 : Saved FP = 7    <=== [ebp]
11 : Return Address
10 : var_1 = 200
9 : arg_2 = 13
8 : arg_1 = 11
7 : Saved FP = 1
6 : Return Address
5 : var_1 = 100
4 : arg_3 = 3
3 : arg_2 = 2
2 : arg_1 = 1
1 : Saved FP
0 : Return Address
================================
```

```
====== Current Call Stack ======
10 : var_1 = 200    <=== [esp]
9 : arg_2 = 13
8 : arg_1 = 11
7 : Saved FP = 1
6 : Return Address
5 : var_1 = 100
4 : arg_3 = 3
3 : arg_2 = 2
2 : arg_1 = 1
1 : Saved FP
0 : Return Address
================================

====== Current Call Stack ======
5 : var_1 = 100    <=== [esp]
4 : arg_3 = 3
3 : arg_2 = 2
2 : arg_1 = 1
1 : Saved FP
0 : Return Address
================================

Stack is empty.
```

셀프 피드백

처음 하다보니 SP, FP 등 용어들 모두 생소하게 느껴져 어려웠지만 1주차 강의 영상을 반복적으로 들으며 용어를 익히며 스택 동작 과정을 이해할 수 있었다. 개인적으로 아쉬웠던 점은 효율적 코드 작성에 서툴러 많은 수정과 디버깅을 반복했다는 점과 터미널 용어들조차도 처음하다 보니 초반에 이해하는 데에 시간을 많이 투자해야만 했다는 점이었다. 하지만 모두 시간을 더 투자해서라도 하다보니 스택 프레임, 레지스터 원리, FP와 SP의 역할 등에 대해 이해할 수 있었다. 다음에는 이번 경험을 바탕으로 더 깔끔하고 효율적인 코드 작성을 위해 노력해야겠다는 목표가 생겼다.