callstack_2024350224_안철민

일단 주어진 base.c 에서 주어진 것이 무엇인지 살펴 보았습니다.

```
/*  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"
   ========================================================================
*/
```

```
#include <stdio.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;

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)
    print_stack();
    func2(11, 13);
    // func2의 스택 프레임 제거 (함수 에필로그 + pop)
    print_stack();
}


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

    // func2의 스택 프레임 형성 (함수 프롤로그 + push)
    print_stack();
    func3(77);
    // func3의 스택 프레임 제거 (함수 에필로그 + pop)
```

```
    print_stack();
}


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

    // func3의 스택 프레임 형성 (함수 프롤로그 + push)
    print_stack();
}


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

일단 주어진 base.c 파일을 살펴보면, 콜 스택의 구현을 위한 기본적인 뼈대가 제공되어 있습니다. 함수 호출 시 생성되는 스택 프레임을 시뮬레이션하기 위한 구조가 잡혀 있는데, 실제 메모리 대신 배열을 활용한 방식으로 구현하게 되어 있습니다.

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

데이터 저장은 두 개의 배열을 통해 이루어지는데, call_stack 배열은 정수 값과 메타데이터(-1)를 함께 저장하도록 구성되어 있으며, stack_info 배열은 각 스택 요소의 의미를 설명하는 문자열을 저장하는 구조로 되어 있습니다.

```
int SP = -1;
int FP = -1;
```

스택 동작을 제어하기 위해 SP와 FP라는 두 개의 포인터 변수가 사용됩니다. SP는 스택의 최상단 인덱스를 가리키며, 데이터의 삽입과 제거 시 이 값을 기준으로 움직이도록 되어 있습니다. FP는 현재 실행 중인 함수의 기준점을 나타내는 포인터로, 함수 호출 시 스택 프레임을 식별하는 데 사용되도록 구현되어 있습니다.

```
//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)
    print_stack();
    func2(11, 13);
    // func2의 스택 프레임 제거 (함수 에필로그 + pop)
    print_stack();
}


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

    // func2의 스택 프레임 형성 (함수 프롤로그 + push)
    print_stack();
    func3(77);
    // func3의 스택 프레임 제거 (함수 에필로그 + pop)
    print_stack();
}


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

```
    int var_4 = 400;

    // func3의 스택 프레임 형성 (함수 프롤로그 + push)
    print_stack();
}
```

함수 구조는 func1, func2, func3 등이 서로를 호출하는 방식으로 구성되어 있으며, 프로그램의 시작점인 main() 함수에서는 func1을 호출하면서 흐름이 시작되도록 되어 있습니다. 현재 스택의 상태를 출력하는 기능은 print_stack() 함수를 통해 제공되며, 이를 통해 스택에 저장된 값들과 그 설명을 확인할 수 있습니다.

값 저장 방식에도 일정한 규칙이 적용되어 있습니다. 매개변수와 지역 변수는 별도의 변환 없이 그대로 정수 값으로 스택에 저장되도록 구현되어 있으며, 이전 함수의 FP 값은 스택 상의 인덱스로 저장되어 SFP(Saved Frame Pointer) 역할을 하도록 구성되어 있습니다. 또한 반환 주소는 실제 메모리 주소 대신 -1이라는 특수 값을 사용하여 표현되도록 되어 있습니다.

---

base.c를 보면 함수들의 기본 구조만 있고, 아직 스택 프레임이 실제로 형성되고 해제되는 코드가 없습니다. 함수의 프롤로그(함수 시작 시 스택 프레임 설정)와 에필로그(함수 반환 시 스택 프레임 해제)를 직접 구현해야 합니다.

특히 함수의 주석에 있는

```
// func1의 스택 프레임 형성 (함수 프롤로그 + push)
// func2의 스택 프레임 제거 (함수 에필로그 + pop)
```

이 부분들을 구현해야 합니다.

또한 기본적인 스택 조작을 위한 push와 pop 함수도 만들어야 할 것 같습니다. 실제 스택과 유사하게 동작하려면 SP를 먼저 증가시키고 그 위치에 값을 저장하는 방식으로 구현해야겠네요.

print_stack() 함수는 이미 구현되어 있어서, 제가 구현한 스택 상태를 확인하는 데 유용할 것 같습니다. 함수들이 예상대로 스택 프레임을 생성하고 제거하는지 이 함수를 통해 검증할 수 있을 것 같습니다.

스택 프레임을 직접 구현하기 위해, 먼저 스택의 기본 동작을 담당하는 push와 pop 함수를 작성했습니다.

```
// 스택에 값을 추가하는 함수
void push(int value, const char* info) {
    if (SP >= STACK_SIZE - 1) {
        printf("Stack overflow!\n");
        return;
    }
    SP++;
    call_stack[SP] = value;
    snprintf(stack_info[SP], 20, "%s", info);
}
```

push 함수는 스택에 새로운 값을 넣는 역할을 합니다. 먼저 스택이 가득 찼는지 확인해서, 만약 SP가 STACK_SIZE - 1보다 크거나 같으면 "Stack overflow!" 메시지를 출력하고 함수를 바로 종료합니다.

오버플로우가 아니라면, SP를 먼저 증가시키고 그 위치에 값을 저장하는 순서로 구현했습니다.

마지막으로, 값에 대한 설명도 stack_info 배열에 같이 저장해서, 나중에 print_stack 함수로 스택 상태를 확인할 때 어떤 값이 어떤 용도였는지 쉽게 알 수 있도록 했습니다.

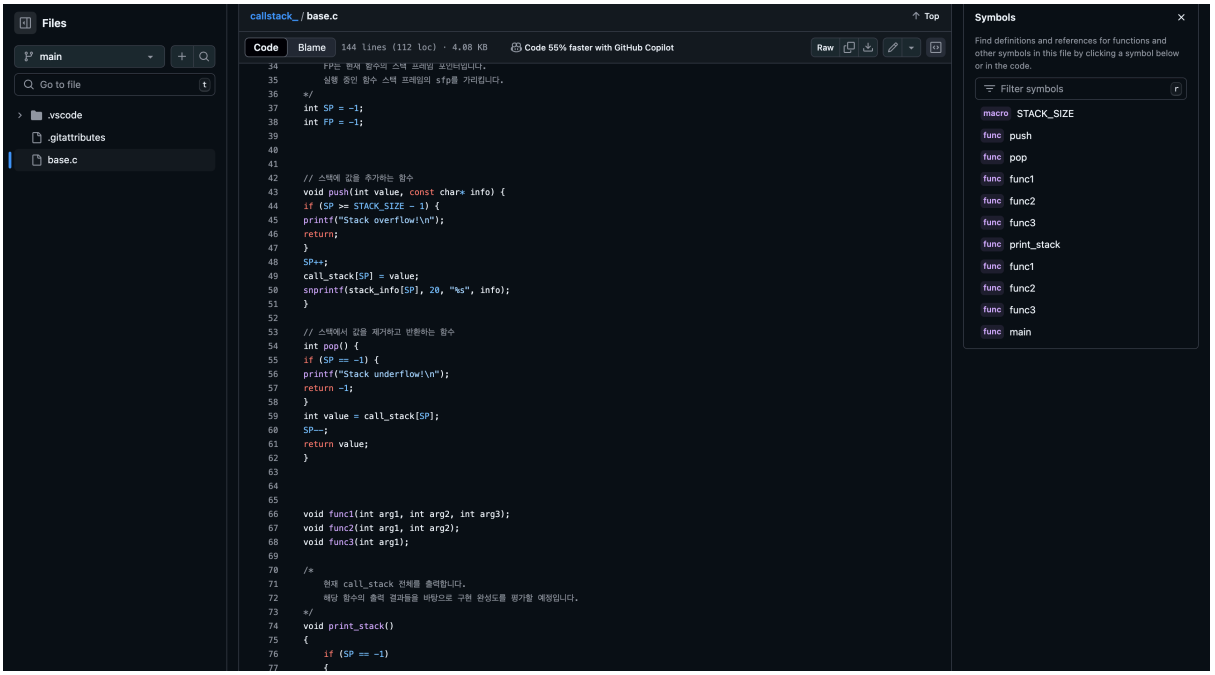
```
// 스택에서 값을 제거하고 반환하는 함수
int pop() {
    if (SP == -1) {
        printf("Stack underflow!\n");
        return -1;
    }
    int value = call_stack[SP];
    SP--;
```

```
    return value;
}
```

pop 함수는 스택에서 값을 꺼내는 역할을 합니다. 먼저 SP가 -1이면 스택이 비어있다는 뜻이므로, "Stack underflow!"를 출력하고 -1을 반환합니다. 스택이 비어있지 않으면, 현재 SP가 가리키는 위치의 값을 value에 저장한 뒤 SP를 1 감소시켜 스택의 크기를 줄입니다.

실제 시스템의 pop 명령처럼, 값을 먼저 꺼내고 포인터를 조작하는 순서로 구현했습니다. 이렇게 만든 push와 pop 함수는 이후 함수 프롤로그와 에필로그에서 스택 프레임을 만들고 해제할 때 기본적으로 계속 사용하게 했습니다.

이후 깃허브에 커밋을 해주었습니다.



함수 프롤로그, 에필로그 구현 과정

스택 프레임을 실제로 쌓고 관리하기 위해, 각 함수의 프롤로그와 에필로그를 구현했습니다.

func1

```
cvoid func1(int arg1, int arg2, int arg3)
{
   // 프롤로그: 스택 프레임 설정
   push(arg3, "arg3");
   push(arg2, "arg2");
   push(arg1, "arg1");
   push(-1, "Return Address");
   push(FP, "func1 SFP");
   FP = SP;

   int var_1 = 100;
   push(var_1, "var_1");

   // func1의 스택 프레임 형성 (함수 프롤로그 + push)
   print_stack();
   func2(11, 13);
   // func2의 스택 프레임 제거 (함수 에필로그 + pop)
   print_stack();
}
```

func1에서는 함수가 호출될 때,

먼저 전달받은 인자 arg3, arg2, arg1 를 차례대로 push해서 스택에 쌓습니다.

그 다음, 반환 주소(-1로 대체)와 이전 FP값(FP, 즉 func1 SFP)을 push합니다.

이후 FP를 현재 SP로 갱신해서, 이 시점이 func1의 스택 프레임 기준점이 되도록 했습니다.

지역 변수(var_1)도 push해서 스택에 올려주었습니다.

이렇게 프롤로그가 끝나면 print_stack()으로 스택 상태를 확인하고,

func2를 호출한 뒤 다시 print_stack()을 호출해 프레임 변화를 확인하도록 했습니다.

func2

```
cvoid func2(int arg1, int arg2)
{
   // 프롤로그: 스택 프레임 설정
   push(arg2, "arg2");
   push(arg1, "arg1");
```

```
    push(-1, "Return Address");
    push(FP, "func2 SFP");
    FP = SP;

    int var_2 = 200;
    push(var_2, "var_2");

    // func2의 스택 프레임 형성 (함수 프롤로그 + push)
    print_stack();
    func3(77);
    // func3의 스택 프레임 제거 (함수 에필로그 + pop)
    print_stack();
}
```

func2도 마찬가지로,

전달받은 인자(arg2, arg1), 반환 주소, 이전 FP(func2 SFP)를 차례대로 push해서 스택에 쌓고,

FP를 갱신합니다.

지역 변수 var_2도 push해서 스택에 올렸습니다.

프롤로그가 끝난 뒤 print_stack()으로 상태를 확인하고,

func3를 호출한 후 다시 print_stack()을 호출해 변화를 확인했습니다.

func3

```
cvoid func3(int arg1)
{
    // 프롤로그: 스택 프레임 설정
    push(arg1, "arg1");
    push(-1, "Return Address");
    push(FP, "func3 SFP");
    FP = SP;

    int var_3 = 300;
    int var_4 = 400;
    push(var_3, "var_3");
    push(var_4, "var_4");
```

```
    // func3의 스택 프레임 형성 (함수 프롤로그 + push)
    print_stack();
}
```

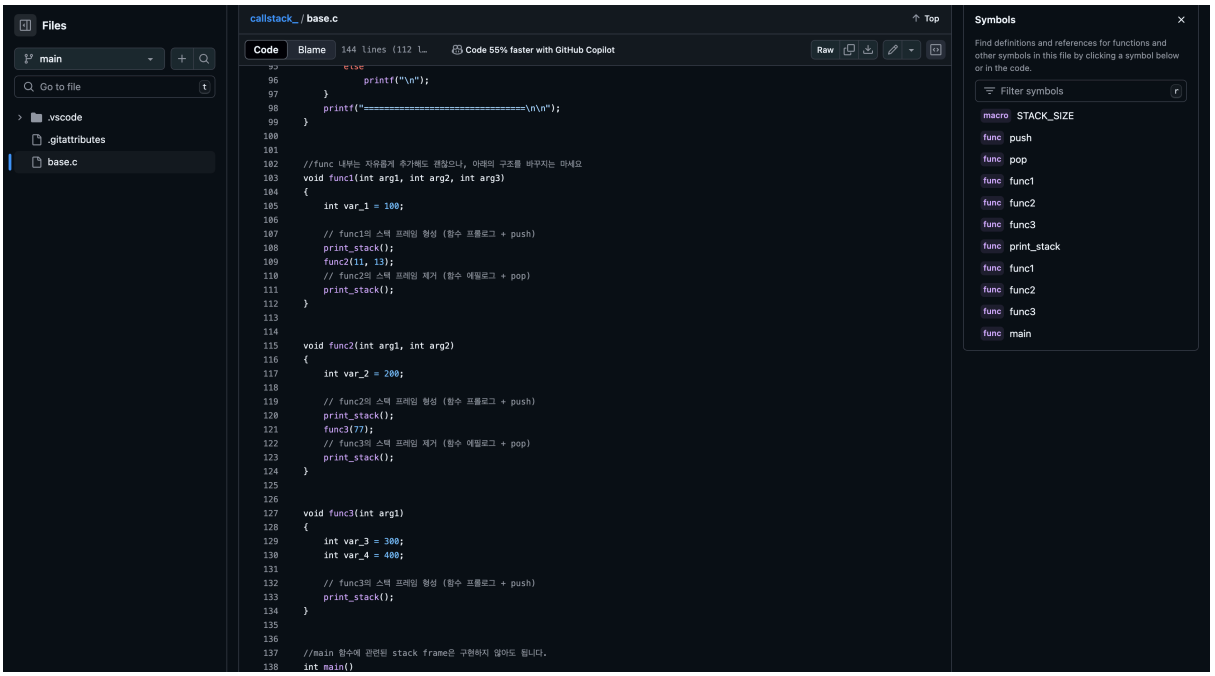
func3에서는 인자(arg1), 반환 주소, 이전 FP(func3 SFP)를 차례대로 push해서 쌓고,

FP를 갱신한 뒤, 지역 변수(var_3, var_4)도 push해서 스택에 올렸습니다.

마지막에 print_stack()을 호출해 스택 상태를 확인할 수 있도록 했습니다.

각 함수의 프롤로그에서 push **함수**를 사용해 인자, 반환 주소, SFP, 지역 변수까지 모두 스택에 올렸습니다. FP는 항상 SFP가 push된 직후의 SP로 갱신해서, 스택 프레임의 기준점이 정확하게 잡히도록 했습니다.

print_stack()을 적절한 위치에 호출해서, 함수 호출/반환 시점마다 스택 상태가 어떻게 변하는지 바로 확인할 수 있도록 했습니다. 실제 x86 함수 호출 규약과 최대한 유사하게, 스택 프레임이 쌓이고 해제되는 구조로 구현했습니다.



프롤로그에 이어서, 각 함수의 에필로그도 구현했습니다. 에필로그는 함수가 반환되기 전에 스택 프레임을 정리하고 호출자 함수로 제어를 돌려주는 역할을 합니다.

```
void func1(int arg1, int arg2, int arg3)
{
    // 프롤로그: 스택 프레임 설정
    push(arg3, "arg3");
    push(arg2, "arg2");
    push(arg1, "arg1");
    push(-1, "Return Address");
    push(FP, "func1 SFP");
    FP = SP;

    int var_1 = 100;
    push(var_1, "var_1");

    // func1의 스택 프레임 형성 (함수 프롤로그 + push)
    print_stack();
    func2(11, 13);
    // func2의 스택 프레임 제거 (함수 에필로그 + pop)
    print_stack();

    // 에필로그: 스택 프레임 정리
    SP = FP;
    FP = call_stack[SP];
    pop();  // SFP 제거
    pop();  // 반환 주소 제거
    pop();  // arg1 제거
    pop();  // arg2 제거
    pop();  // arg3 제거
}
```

func1 에필로그

func1 함수의 에필로그 부분에서는 다음 작업을 수행합니다

먼저 SP = FP;로 SP를 현재 함수의 프레임 포인터로 설정했습니다. 이렇게 하면 스택에서 지역 변수 영역(var_1)이 자연스럽게 제거됩니다. SP를 FP 위치로 옮기는 것은 x86 아키텍처에서 mov esp, ebp 명령과 동일한 역할을 합니다.

그 다음 FP = call_stack[SP];를 통해 이전 함수의 FP값을 복원했습니다. 현재 SP(=FP)가 가리키는 위치에는 func1을 호출한 함수의 FP값(SFP)이 저장되어 있기 때문에, 이 값을 다시 FP에 넣어주면 호출자의 스택 프레임으로 돌아가는 것입니다.

마지막으로 스택에 쌓았던 요소들을 pop 함수로 순서대로 제거했습니다. 프롤로그에서 푸시했던 순서와 반대로 제거해야 하므로, SFP, 반환 주소, 그리고 인자들(arg1, arg2, arg3)을 차례로 pop 했습니다.

인자들을 역순으로 푸시했기 때문에(arg3, arg2, arg1), 제거할 때는 순서대로(arg1, arg2, arg3) 제거해서 LIFO 원칙을 지켰습니다.

func2 에필로그

```
void func2(int arg1, int arg2)
{
    // 프롤로그: 스택 프레임 설정
    push(arg2, "arg2");
    push(arg1, "arg1");
    push(-1, "Return Address");
    push(FP, "func2 SFP");
    FP = SP;

    int var_2 = 200;
    push(var_2, "var_2");

    // func2의 스택 프레임 형성 (함수 프롤로그 + push)
    print_stack();
    func3(77);
    // func3의 스택 프레임 제거 (함수 에필로그 + pop)
    print_stack();

    // 에필로그: 스택 프레임 정리
    SP = FP;
    FP = call_stack[SP];
    pop();  // SFP 제거
```

```
    pop();  // 반환 주소 제거
    pop();  // arg1 제거
    pop();  // arg2 제거
}
```

func2의 에필로그도 동일한 원리로 작동합니다. 차이점은 func2는 인자가 2개뿐이므로 pop() 호출도 5번이 아닌 4번만 수행합니다.

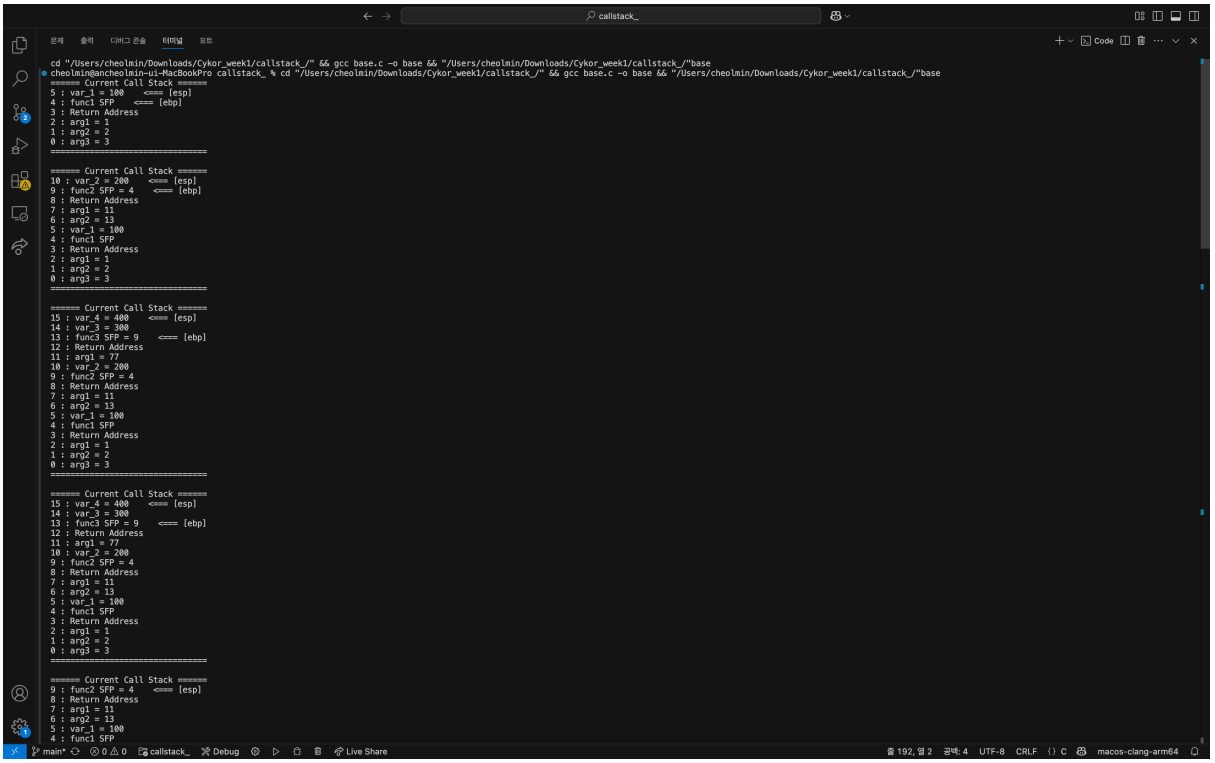
에필로그를 구현할 때 가장 중요하게 생각한 부분은 스택 정리 순서입니다. 스택은 LIFO 구조이므로, 프롤로그에서 push했던 순서의 정반대 순서로 pop해야 합니다. 따라서 인자들도 push했던 순서(arg3, arg2, arg1)와 반대로 pop(arg1, arg2, arg3)해야 합니다.

또한 SP와 FP의 조작 순서도 중요합니다. 먼저 SP를 FP로 설정하여 지역 변수들을 스택에서 제거하고, 그 다음에 FP를 복원해야 정확한 스택 프레임 관리가 가능합니다.

이러한 에필로그 구현을 통해 함수 호출이 완료된 후 스택이 정확하게 이전 상태로 복원되도록 했고, 결과적으로 print_stack() 함수로 확인했을 때 예상했던 스택 상태가 정확히 출력되는 것을 확인할 수 있었습니다.

결과

실행 결과를 보면, 의도한 대로 함수가 호출될 때마다 새로운 스택 프레임이 쌓이고, 함수가 반환될 때마다 프레임이 정확하게 해제되는 것을 확인할 수 있습니다.



func1 호출 직후

```
====== Current Call Stack ======
5 : var_1 = 100    ⇐= [esp]
4 : func1 SFP    ⇐= [ebp]
3 : Return Address
2 : arg1 = 1
1 : arg2 = 2
0 : arg3 = 3
==================================
```

func1이 호출되면서 인자(arg3, arg2, arg1), 반환 주소, SFP, 지역 변수(var_1)가 차례대로 쌓인 모습입니다. [esp], [ebp] 표시로 각각 SP, FP 위치가 명확히 보입니다.

func2, func3 호출 시

```
10 : var_2 = 200    ⇐= [esp]
9 : func2 SFP = 4    ⇐= [ebp]
```

```
...
15 : var_4 = 400    ⇐= [esp]
14 : var_3 = 300
13 : func3 SFP = 9    ⇐= [ebp]
...
```

func2, func3가 호출될 때마다 새로운 프레임이 위에 쌓이고, 각 함수의 SFP가 이전 함수의 FP 위치를 잘 가리키고 있습니다. 스택 프레임 체인이 정확하게 연결된 모습입니다.

함수 반환 시

```
text9 : func2 SFP = 4    ⇐= [esp]
...
4 : func1 SFP    ⇐= [esp]
...
Stack is empty.
```

func3, func2, func1이 반환될 때마다 스택 프레임이 순서대로 해제되고, 마지막에는 스택이 완전히 비워지면서 "Stack is empty."가 출력됩니다.

```
cheolmin@ancheolmin-ui-MacBookPro callstack_ % cd "/Users/cheolmin/Downloads/Cykor_week1/callstack_/" && gcc base.c -o base && "/Users/cheolmin/Downloads/Cykor_week1/callstack_/"base
====== Current Call Stack ======
5 : var_1 = 100    ⇐= [esp]
4 : func1 SFP    ⇐= [ebp]
3 : Return Address
2 : arg1 = 1
1 : arg2 = 2
0 : arg3 = 3
================================
```

```
====== 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
================================

====== Current Call Stack ======
15 : var_4 = 400    ⇐= [esp]
14 : var_3 = 300
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
================================

====== Current Call Stack ======
15 : var_4 = 400    ⇐= [esp]
14 : var_3 = 300
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
================================

====== Current Call Stack ======
9 : func2 SFP = 4    <== [esp]
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
================================

====== Current Call Stack ======
4 : func1 SFP    <== [esp]
3 : Return Address
2 : arg1 = 1
1 : arg2 = 2
0 : arg3 = 3
================================
```

과제에서 요구한 대로 함수 프롤로그와 에필로그, push/pop, SP/FP 조작을 모두 직접 구현했고,
실행 결과가 예시와 거의 동일하게 나와서 콜스택 동작 원리를 제대로 시뮬레이션할 수 있었습니다.

또한 GitHub 커밋도 기능별로 나눠서 남겼고, 이 과정을 통해 함수 호출과 스택 프레임 관리가 실제로 어떻게 이뤄지는지 체감할 수 있었던 유익한 과제였습니다.