CYKOR 1주차 과제 보고서

2025350003 최동주

목차:

1. 전체 코드 및 코드 설명(일부 주석 생략)

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

// 매개변수들을 stack에 push
void push( char*stk1, int stk2) {
    SP =SP +1;
    call_stack[SP] =stk2;
    sprintf_s(stack_info[SP], sizeof(stack_info[SP]), "%s", stk1);
    if(strstr(stk1, "SFP") !=NULL) FP =SP;
}
// stack에 쌓인것들을 pop
void pop() {
    if(strstr(stack_info[SP], "SFP") !=NULL) FP =call_stack[SP];
    call_stack[SP] =0;
```

```
    strcpy_s(stack_info[SP],sizeof(STACK_SIZE), "");
    SP =SP -1;
}

/*
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)
    push("arg1", 1);
    push("arg2", 2);
```

```
    push("arg3", 3);
    push("Return Address", -1);
    push("func1 SFP", -1);
    push("var_1", 100);
    print_stack();
    func2(11, 13);
    // func2의 스택 프레임 제거 (함수 에필로그 + pop)
    pop();
    pop();
    pop();
    pop();
    pop();
    print_stack();
}

void func2(int arg1, int arg2)
{
    int var_2 =200;
    // func2의 스택 프레임 형성 (함수 프롤로그 + push)
    push("arg1", 11);
    push("arg2", 13);
    push("Return Address", -1);
    push("func2 SFP", 4);
    push("var_2", 200);
    print_stack();
    func3(77);
    // func3의 스택 프레임 제거 (함수 에필로그 + pop)
    pop();
    pop();
    pop();
    pop();
    pop();
    print_stack();
}

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

```
    // func3의 스택 프레임 형성 (함수 프롤로그 + push)
    push("arg1", 77);
    push("Return Address", -1);
    push("func3 SFP", 9);
    push("var_3", 300);
    push("var_4", 400);
    print_stack();
}

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

```
void push( char* stk1, int stk2) {
    SP = SP + 1;
    call_stack[SP] = stk2;
    sprintf_s(stack_info[SP], sizeof(stack_info[SP]), "%s", stk1);
    if (strstr(stk1, "SFP") != NULL) FP = SP;
}

// stack에 쌓인것들을 pop
void pop() {
    if (strstr(stack_info[SP], "SFP") != NULL) FP = call_stack[SP];
    call_stack[SP] = 0;
    strcpy_s(stack_info[SP],sizeof(STACK_SIZE), "");
    SP = SP - 1;
}
```

push와 pop 구현 함수. 두 함수 모두 특정 값을 반환하는 것은 아니기에 void 형식이다.

push: push는 문자열 형식의 stk1와 정수 형식의 stk2를 매개변수로 가지고 있다. 선언된 문자열 배열의 값을 수정해야 하기에 포인터 문법인 char*를 사용했으며, 이전에 오류 발생을 생각하면 const char*를 사용하는것이 좋다고 배워서 그리 했는데, const를 빼도 잘 작동하길래 보고서를 쓰는 과정에서 뺐다(사실 매개변수의 문자열 값을 수정할 일이 없으니 애초부터 안써도 되는 것 이었지만). 또한 매개변수 문자열에서 “SFP”라는 패턴을 발견할 경우, FP의 값을 해당 SP값으로 초기화하기 위해 조건문과 strstr 함수를 사용하였다.

pop: 별다른 매개변수를 받지 않는다. push와 마찬가지로 조건문과 strstr을 사용해 “SFP”라는 패턴을 발견하면 해당 인덱스의 call_stack값, 즉 저장된 이전 SFP값을 저장하도록 하였다. 이후 저장된 스택을 비우기 위해 call_stack 및 stack_info값을 0 또는 “”로 수정했으며, 과정이 마무리 된 후 SP의 값을 내렸다.

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

    // func1의 스택 프레임 형성 (함수 프롤로그 + push)
    push("arg1", 1);
    push("arg2", 2);
    push("arg3", 3);
    push("Return Address", -1);
    push("func1 SFP", -1);
    push("var_1", 100);
    print_stack();

    func2(11, 13);

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

main 함수를 통해 불려온 func1이다. push를 통해 매개변수 3개와 그 값을

저장하며, 이후 func1의 주소를 반환하고, func1의 SFP와 지역변수의 값을 push했다. 아래의 pop은 func2의 스택 프레임을 제거하기 위한 에필로그 과정이다.

```
void func2(int arg1, int arg2)
{
    int var_2 = 200;
    // func2의 스택 프레임 형성 (함수 프롤로그 + push)
    push("arg1", 11);
    push("arg2", 13);
    push("Return Address", -1);
    push("func2 SFP", 4);
    push("var_2", 200);
    print_stack();

    func3(77);

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

func1 내부와 비슷하다. 매개변수 -> 주소 반환 -> SFP값 저장 -> 지역변수 저장 순서이며, 아래의 pop들 역시 에필로그 과정이다.

```
void func3(int arg1)
{
    int var_3 = 300;
    int var_4 = 400;
    // func3의 스택 프레임 형성 (함수 프롤로그 + push)
    push("arg1", 77);
    push("Return Address", -1);
    push("func3 SFP", 9);
    push("var_3", 300);
    push("var_4", 400);
    print_stack();
}
```

func1,2 내부와 비슷하다. 매개변수 -> 주소 반환 -> SFP값 저장 ->

지역변수 저장 순서이며, 아래의 pop들 역시 에필로그 과정이다.

```
int main()
{

    func1(1, 2, 3);

    // func1의 스택 프레임 제거 (함수 에필로그 + pop)
    pop();
    pop();
    pop();
    pop();
    pop();
    pop();
    print_stack();

    return 0;
}
```

메인함수이다. func1의 스택프레임을 제거하기 위한 pop밖에 없다(에필로그).

2. 코드 실행 결과

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

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

====== 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 : arg2 = 13
6 : arg1 = 11
5 : var_1 = 100
4 : func1 SFP
3 : Return Address
2 : arg3 = 3
1 : arg2 = 2
0 : arg1 = 1
================================

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

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

Stack is empty.
```

3. 과제 수행 일지

3/20일: git test 및 초안 commit

아직 git을 이용하는것이 익숙치 않아 base.c를 commit해보는 것을 목표로 git을 공부하고 base.c를 commit 함. 이후 github에 접속하여 제대로 commit이 된 것을 확인했음.

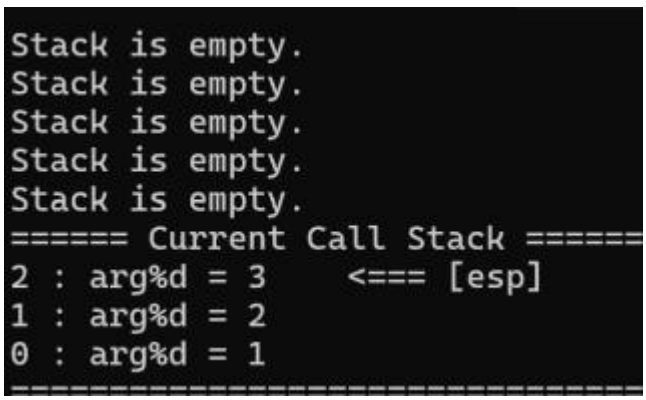
3/25일(1): main 함수 수정

우선, main 함수에 있는 func1의 매개변수와 그 값을 스택에 쌓아야 하므로 이 부분부터 구현하기로 했다. for문을 사용해서 i가 0~2일때 SP의 값을 늘리는 동시에 arg1~arg3을 stack_info에 저장하고 call_stack에 그 값을 저장하기로 했다.

다만 stack_info에 arg1,2,3을 저장하는 방식에 문제가 생겼다.

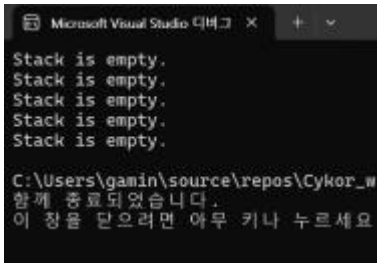
`strcat(stack_info[i], "arg%d", i + 1);`

를 사용하면 출력이 arg%d처럼 나오지만 stack이 정상적으로 출력이 되긴한다.



그래서 어떻게 대입을 해야하나 찾아보니, sprintf라는 함수를 발견했다. 따라서 코드를 수정해

`sprintf_s(stack_info[i], "arg%d", i + 1);` 로 작성하면



처럼 뜬다. 무엇이 문제인지 고민하고 리서치 해봐야겠다. 일단은 git

commit!

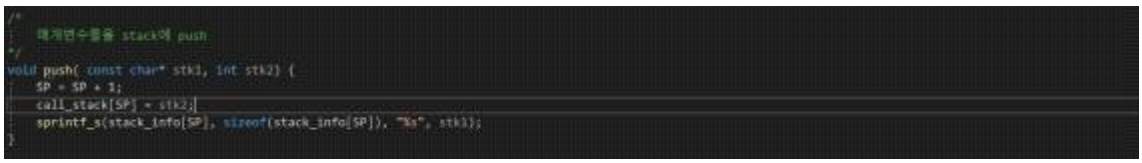
3월 25일(2) : 메인함수 내 func1 구현 완료!
오류가 난 이유는 단순했다. sprintf를 사용했을 때 버퍼오버플로우 방지를 위해 sizeof를 통해 버퍼의 크기를 전달했어야 했는데, 이를 깜빡했었다. 이것을 추가하고 나니 이제는 arg1,2,3으로 잘 뜬다.

또한 for문이 끝났을 때 주소값을 반환할 수 있도록

```
SP = SP + 1;
strcat(stack_info[SP], "Return Address");
call_stack[SP] = -1;
```

을 추가하니 return adress까지 출력이 정상적으로 된다. Commit!

3월/27일: 개편
메인함수 내에 구현을 기능을 했지만..생각을 해보니 이것이 과제의 취지가 아닐것이라는 생각이 들었다. Push 와 Pop 기능을 통해 stack을 구현하는 것이 목적이지 단순히 배열 수정 및 출력이 목적이 아니기 때문이다. 따라서 구현을 했던 기능들을 전부 갈아엎고 처음부터 다시 만들기로 결심했다. 우선의 목표는 함수의 형태로 push, pop을 구현하는 것이다.
우선은 push를 구현하였다.



const char* stk1을 통해 문자열을, int stk2를 이용해 정수값을 받은 후 각각 stack_info, call_stack에 저장한다. 이때 stack_info에 저장은 위에서 썻던 방식을 참고해 적용해주었다.
이후 메인함수에서 func1 위에

```
push("arg1", 1);
push("arg2", 2);
push("arg3", 3);
```

을 해주면..

정상적으로 작동한다! commit!

3/27일(2): 모든 함수에 push 적용 및 순서 배치
우선, 앞서 구현에 성공항 push 함수를 이용해 func1~func3의 매개변수들과 변수들을 push했고, 변수들이 구현이 된 후 push에 “Return adress”와 -1을 통해 주소값 반환 역시 구현하였다.

이제 sfp와 ebp, efp를 구현하면 된다. 그러나 일단은 다른 기능들은 정상 작동하기에 commit을 하겠다.

3월 28: pop 구현 및 sfp, ebp, esp 구현
pop을 어떻게 구현해야 할지 고민을 해보았는데, 일단 기본적으로 push 혹은 stack 출력등의 기능이 SP의 값을 따라가기에 우선 pop을 하면 SP의 값을 하나씩 줄여가는 것으로 구현해보았다.

```
void pop() {
    SP = SP - 1;
}
```

위와 같이 작성한 후, 각 함수의 주석의 위치를 참고하여 pop을 적절히 배치하니

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

====== Current Call Stack ======
4 : var_1 = 100    <=== [esp]
3 : Return Address
2 : arg3 = 3
1 : arg2 = 2
0 : arg1 = 1
================================

Stack is empty.
```

위와 같이 잘 작동했다. 물론, 어떻게 보면 SP의 값만을 조정한 것이기에 실제 스택을 보여준다기보다는 일종의 눈속임이라고 보는게 맞을 것 같다.
이것이 눈속임이라고 생각하니, 무언가 마음에 들지 않았다. 따라서 SP의 값이 변함에 따라 실제 stack_info와 call_stack의 값을 변화시키기로 하였다.

우선, 다음의 코드를 통해 주요 함수 작동 이후에도 배열들의 값이 어떻게 남아있는지 알아보기로 했다.

```
    printf("\n");
    for (int i = 0; i < 13; i++) {
        printf("%d ", call_stack[i]);
    }
    printf("\n");
    printf("\n");
    for (int i = 0; i < 13; i++) {
        printf("%s ", stack_info[i]);
    }
```

그 결과

```
1 2 3 -1 100 11 13 -1 200 77 -1 300 400

arg1 arg2 arg3 Return Address var_1 arg1 arg2 Return Address var_2 arg1 Return Address var_3 var_4
C:\Users\gamin\source\repos\Cykor_week1\x64\Debug\Cykor_week1.exe(프로세스 14324)이(가) 0 코드(0x0)와 함
```

예상대로 배열들의 값은 그대로인 것을 확인 할 수 있다.
pop을 통해 stack_info와 call_stack 각각의 값을 모두 “”과 0 로 처리하기로 했다.

```
void pop() {
    call_stack[SP] = 0;
    strcpy(stack_info[SP], "");
    SP = SP - 1;
}
```

pop을 이렇게 처리하고 실행을 해보니

```
0 0 0 0 0 0 0 0 0 0 0 0 0
```

처리가 잘 되었다. 이로써 pop 기능까지 구현을 완료하였다. commit!

이제 frame pointer, ebp 및 esp를 구현할 시간인 것 같다.
우선 frame pointer는 각 함수의 return adress 뒤에

```
push("func1 SFP", -1);
push("func2 SFP", 4);
push("func3 SFP", 9);
```

을 적었다. SFP가 이름 그대로 FP를 저장하는 개념이니, func1은 구현된 stack에서 첫번째 실행된 함수 이므로 func1 SFP에서 따로 저장할 값이 없으니 -1과 함께 push했고, 뒤에 두개의 SFP는 각각 이전의 SFP 위치인 4, 9와 함께 push했다.

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

정상적으로 잘 작동한다.
이제 esp, ebp를 구현해야 하는데, esp는 SF의 값에 따라 이미 최상단에 표기되지만, ebp는 SFP의 값에 따라 구현되기에 이 부분을 개발해야 한다. 어떻게 구현하는게 좋을까 생각해보았지만, 아주 간단하더라도 함수의 형태로 ebp를 지정하는 것이 좋을 것 같아 함수의 형태로 ebp를 저장하기로 했었다.

```
void set_ebp() {
    FP = SP;
}
```

그러나 이렇게 구현하면, 함수의 에필로그 과정에서의 ebp를 구현할 수가 없었다. 이를 해결하려면 어떻게 해야 할지 고민을 해보는 과정에서, push를 통해 구현한 SFP를 pop하지 않은것, 그리고 함수 에필로그를 잘못 이해한 탓에 에필로그 과정이 엉망인 것을 확인하고 수정했다. 일단은 ebp를 구현하기에 앞서 commit을 하였다.

반나절 정도 고민을 해본 결과, 잘 하면 push와 pop 함수 내부에 그것을 구현할 수 있을것 같다는 생각이 들었다. 조건문을 통해 SP 혹은 FP 같은 특정

값이 조건을 만족할 때 esp와 ebp를 설정하는 방식으로 말이다.

3월 29: esp, esb 구현

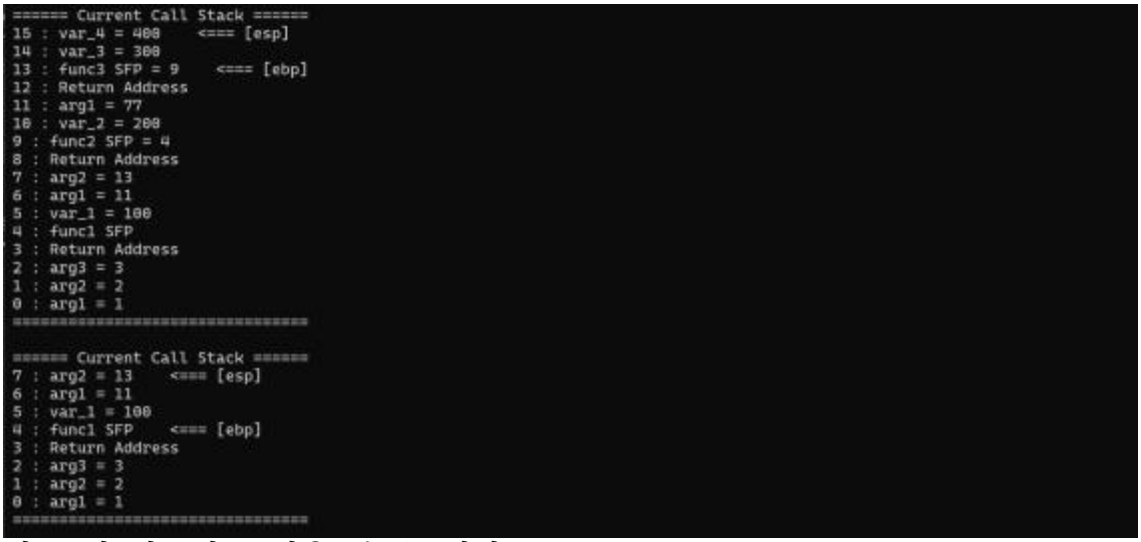
밤 사이 생각을 해본 결과, push의 문자열 매개변수에 “SFP”라는 패턴이 있는 것을 감지하면 조건문을 통해 FP의 값을 조작할 수 있지 않을까? 라는 생각이 들었다. 다만, 공부를 해본 내용 중 특정 패턴을 감지하는 함수에 대해 공부를 해본 기억이 없기에 인터넷에 찾아보았다.

찾아보니, strstr 함수를 통해 이를 구현할 수 있다는 것을 알았다. 알아본 내용을 통해 구현한 코드는 다음과 같다.

```
// 매개변수들을 stack에 push
void push( const char* stk1, int stk2) {
    SP = SP + 1;
    call_stack[SP] = stk2;
    sprintf_s(stack_info[SP], sizeof(stack_info[SP]), "%s", stk1);
    if (strstr(stk1, "SFP") != NULL) FP = SP;
}

// stack에 쌓인것들을 pop
void pop() {
    if (strstr(stack_info[SP], "SFP") != NULL) FP = call_stack[SP];
    call_stack[SP] = 0;
    strcpy_s(stack_info[SP],sizeof(STACK_SIZE), "");
    SP = SP - 1;
}
```

strstr 함수는 문자열에서 패턴을 발견하면 해당 패턴의 시작위치의 포인터를 반환하며, 패턴이 없을 경우 NULL pointer를 반환한다고 한다. 따라서 SFP를 발견하지 못해 NULL을 반환한 경우가 아니라면 (즉 패턴을 발견하여 포인터를 반환한 경우라면) push에서는 FP를 SF와 같은 값을, pop에서는 call_stack[FP]의 값을 가지게 하였다. 이렇게 한다면 쌓이는 과정에서는 SFP의 위치를 저장하게 되고, 빠지는 과정에서는 call_stack에 저장된 이전의 SFP 값을 저장하게 되기에 정상적으로 작동할 수 있을것이라 생각했다. 그 결과는



아주 잘 작동하는 것을 볼 수 있다. commit!
여담으로, strstr을 사용하기 위해 <string.h>라이브러리를 끌고 왔는데, 이때 문인지 vs에서 _s를 달으라고 하기에 수정하였다. 또한 push에 strstr을 먼저 구현했고, 아무 생각없이 이를 복붙했다가 에필로그 과정에서 하나의 ebp도 확인하지 못하는 참사를 겪었다.

이로써 주어진 과제에 대한 구현은 모두 끝났다.

4/7: 함수 프롤로그 수정(pop 재배치)
수업시간에 함수의 stack과정을 공부하던 중, 과제로 제출한 결과물이 잘못되었다는 것을 알게 되었다. 크게 수정할것은 없었고, pop을 재배치해 올바른 출력이 나올 수 있도록 하였다.