CallStack

2023350031 김은채

기본 base.c에서 추가적으로 구현한 함수들에 대한 설명과 실행 결과에 대해서 설명해보겠습니다.

push

```
void push(int value, char *str)
{

    SP++;
    call_stack[SP] = value;
    strcpy(stack_info[SP], str);
}
```

push함수는 call_stack 배열에 저장될 int형 인자와, stack_info 배열에 저장될 char*형 인자를 받습니다.

push 함수가 실행될 때 새로운 값이 스택에 쌓여야 하므로 당연히 전역변수 SP가 1씩 증가됩니다.

call_stack에서 SP 위치에 인자로 받은 value를 저장해줍니다.

이후 string.h의 strcpy함수를 사용해서 stack_info 배열의 SP 위치에 str을 저장해주었습니다.

정리하자면, push 함수에서는 실제 스택에 저장되는 값과 그 값에 대한 설명을 같이 저장하는 과정을 수행하며, 이때 SP를 통해 둘이 같은 위치에 저장될 수 있도록 해주었습니다.

pop

```
int pop()
{
    if(SP==-1)
        return -1;
    int value=call_stack[SP];
    SP--;
    return value;
}
```

pop 함수는 스택포인터를 줄여줌과 동시에 맨위에 있는 값을 꺼내는 것이기 때문에 따로 인자가 필요 없습니다.

먼저 SP==-1임을 확인해줍니다. SP가 -1이라는 뜻은 스택이 비었다는 뜻이 됩니다.

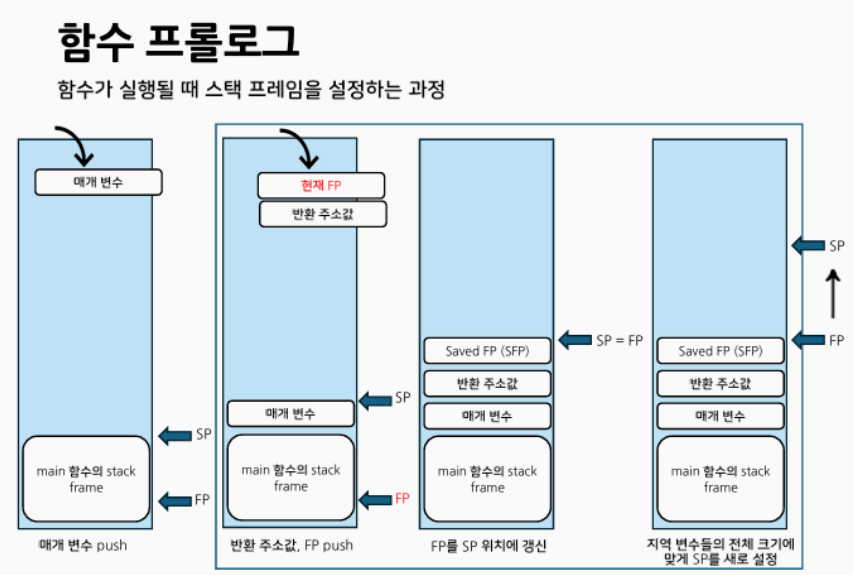
즉, pop을 하고 싶어도 스택 안에 값이 없다는 뜻이므로 -1을 리턴해줍니다.

만약 스택안에 값이 있다고 하면 가장 최상단에 있는 값을 value에 저장해주고 스택포인터를 하나 낮춰줍니다.

이후 아까 저장해놓은 값을 리턴합니다.

prologue

프롤로그 함수는 강의 자료에 있는 "함수 프롤로그" 페이지를 참고했습니다.



해당 그림을 참고해보면 함수 프롤로그에서는 우선 반환주소값을 스택에 넣고, 이후 현재 FP의 값을 젖아합니다. 마지막으로 FP의 위치를 SP의 위치로 만들어주는 과정이 필요합니다.

```
void prologue(char *str)
{
   push(-1, "Return Address");
   push(FP, str);
   FP = SP;
}
```

이해한 프롤로그 과정을 구현해보았습니다.

prologue 함수에는 str을 인자로 받습니다.

위에서 설명한 것처럼 Return Address라는 설명과 함께 -1을 push합니다.

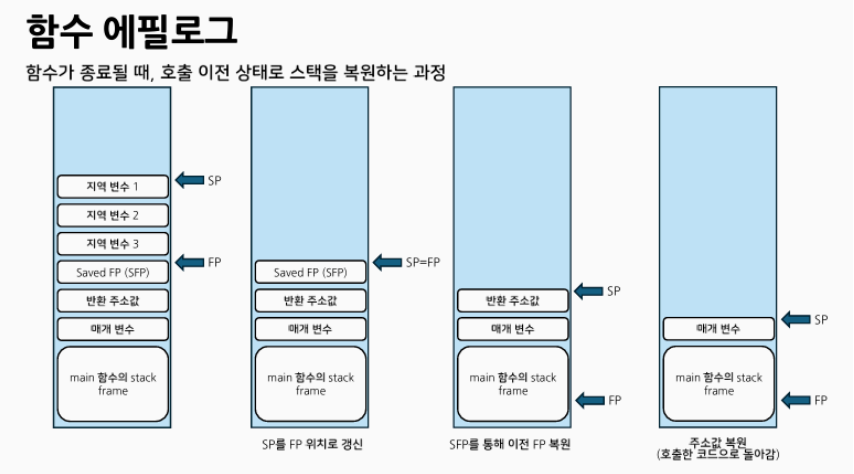
그러면 call_stack에는 -1이, stack_info에는 Return Address가 저장될 것입니다.

다음으로는 현재의 FP를 저장해야합니다.

이때 prologue에서 인자로 받은 str은 FP의 설명이되어 stack_info에 저장됩니다.

마지막으로 FP=SP라인을 통해 FP의 위치를 SP의 위치로 만들어주는 과정을 거쳤습니다.

epilogue



함수 에필로그는 함수 종료시 호출 이전 상태로 스택을 복원하는 과정을 말합니다. 그러면 우선 FP의 위치로 SP를 내려주는 과정이 필요합니다. SP로 내려준 이후 pop을 하게 되면 저장되었던 SFP가 리턴됩니다. 마지막으로 한번 더 pop을 해주게 되면 처음 프롤로그에서 저장해주었던 반환 주소값도 pop이 됩니다.

```
void epilogue(int num_args)
{
    SP = FP;
    FP = pop();
    pop();

    for (int i = 0; i < num_args; i++) {
        pop();
    }
}
```

에필로그 함수에 대해서 설명해보겠습니다.

우선 SP=FP라인은 FP의 위치로 SP를 내려주는 과정을 말합니다.

FP=pop()은 sfp를 pop하는 과정을 말합니다. 그리고 그 pop된 값을 FP에 저장해줌으로써 에필로그 호출 이전으로 FP를 되돌려 놓습니다.

다음 pop()라인은 반환주소값을 pop하는 과정을 말합니다.

마지막으로 에필로그 설명엔 없지만 매개변수로 받은 num_args만큼 반복문을 통해 pop하는 과정을 추가해주었습니다.

어떤 함수가 실행된다고 하면 프롤로그 선언 이전에 그 함수의 매개변수 값들이 스택에 저장되게 됩니다. (ex. arg1, arg2...) 즉, 매개변수까지 pop해주는 과정입니다.

최종 작동 과정

main

```
int main()
{
    func1(1, 2, 3);
```

```
    print_stack();
    return 0;
}
```

main 함수부터 살펴보도록 하겠습니다. main에서는 func1을 호출하고 있는 모습입니다. 이때 함수의 인자는 3개인 것을 확인할 수 있습니다. func1실행이 끝나고 나면 print_stack()을 통해 현 스택의 상태를 출력할 것입니다.

func1

```
void func1(int arg1, int arg2, int arg3)
{

    push(arg3,"arg3");
    push(arg2,"arg2");
    push(arg1,"arg1");
    prologue("func1");
    int var_1 = 100;
    push(var_1, "var_1");
    print_stack();
    func2(11, 13);
        print_stack();
    epilogue(3);
}
```

main에서 호출한 func1 함수로 가보겠습니다. call_stack을 구현하기 위해서는 우선 매개변수 3개를 오른쪽부터 스택에 넣어주어야 합니다. 따라서 설명과 함께 stack에 push해주었습니다.

다음으로는 prologue 함수를 실행해주었습니다. prologue함수를 통해 함수의 반환주소 값과 현재의 FP가 차례차례 저장될 것입니다.

이렇게 스택 프레임 설정 이후 func1의 변수 var_1을 stack에 push해주었습니다.

var_1 push이후 다음 동작이 func2 호출이기 때문에 그전에 print_stack()함수를 통해 현재의 스택 상태를 출력해주었습니다.

그다음 func2를 호출해주었고 이 과정이 끝나면 다시 print_stack()이 실행될 것입니다.

이제 최종적으로 func1의 역할이 끝났으므로 epilogue 함수를 실행시켜서 이전 상태로 스택을 복원하는 과정을 거칩니다.

이때 3은 func1의 매개변수 arg1, arg2, arg3 3개를 pop하기 위해 넣어준 값입니다.

func2

```
void func2(int arg1, int arg2)
{
    push(arg2,"arg2");
```

```
    push(arg1,"arg1");
    prologue("func2");
        int var_2 = 200;
    push(var_2,"var_2");
    print_stack();
    func3(77);
    print_stack();
    epilogue(2);
}
```

func1에서 호출한 func2로 가보겠습니다. 마찬가지로 매개변수를 오른쪽부터 stack에 push하는 과정을 거칩니다. 이후 prologue를 실행해서 스택 프레임을 설정해줍니다.

func2에는 var_2라는 지역변수가 있기 때문에 이를 stack에 push해줍니다.

이 다음 func2의 동작이 func3을 호출하는 것이기 때문에 그 전에 print_stack()함수를 실행해서 현 스택을 출력해줍니다.

이후 func3을 호출하고 실행이 끝나면 한번 더 print_stack()함수를 호출해 현재의 스택 상태를 출력해줍니다.

마지막으로 epliogue(2)함수를 실행해줍니다. func2에는 매개변수가 arg1, arg2 두개였으므로 이를 pop하기 위한 과정입니다.

func3

```
void func3(int arg1)
{
    push(arg1,"arg1");
        prologue("func3");
    int var_3 = 300;
    int var_4 = 400;
    push(var_3,"var_3");
    push(var_4,"var_4");
    print_stack();
    epilogue(1);
}
```

마지막으로 func3을 보겠습니다. func3은 매개변수가 arg1이므로 한번 push해줍니다. 이후 prologue 함수를 실행해서 스택프레임을 생성해줍니다.

func3에서는 지역변수가 2개이므로 이 2개의 지역변수를 각각 push해줍니다.

func3은 func1, func2와는 다르게 따로 함수를 호출하지 않으므로 현재의 스택 상태만 print_stack()을 이용해서 확인해줍니다.

마지막으로 epilogue(1)을 실행해서 스택 프레임을 제거해줍니다.

실행결과

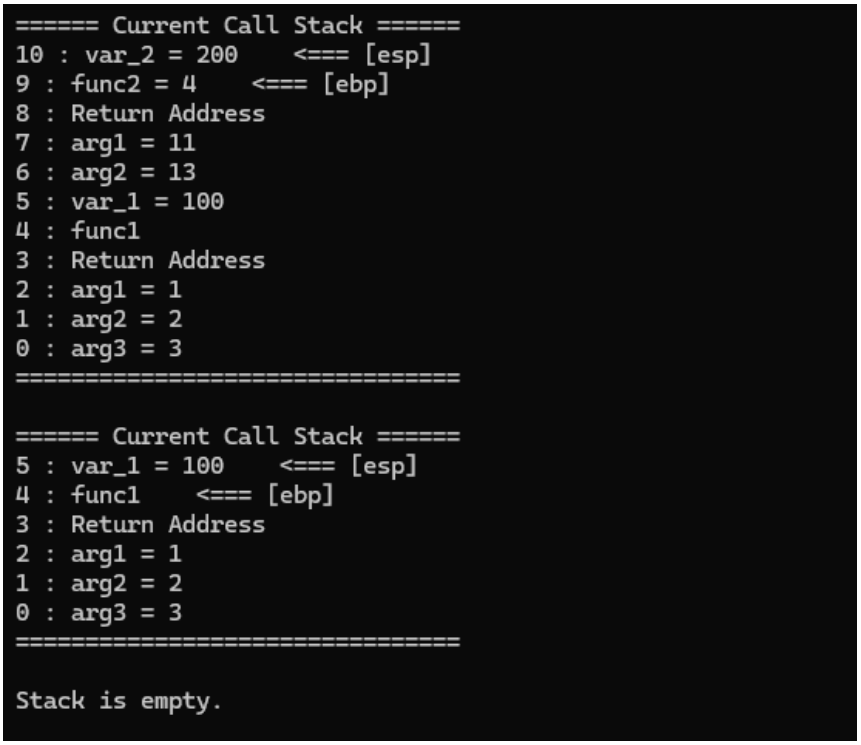
```
C:\Users\82108\Desktop\Cykor\cykor\callstack>gcc base.c -o result

C:\Users\82108\Desktop\Cykor\cykor\callstack>result
```

```
C:\Users\82108\Desktop\Cykor\cykor\callstack>result
====== Current Call Stack ======
5 : var_1 = 100    <=== [esp]
4 : func1    <=== [ebp]
3 : Return Address
2 : arg1 = 1
1 : arg2 = 2
0 : arg3 = 3
================================

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



스택 프레임이 올바르게 출력되는 것을 확인할 수 있습니다.

코드를 전체적으로 쭉 보면 print_stack()이 총 6번 호출되었던 것을 확인할 수 있는데, 실행 결과에서도 6번이 출력되는 것을 확인할 수 있습니다.