#include <stdio.h>

표준 입출력 라이브러리를 포함합니다. printf(), scanf() 등 함수 사용을 가능하게 합니다.

#define STACK\_SIZE 50

매크로 상수 선언입니다. STACK\_SIZE를 50으로 정의합니다. 즉, 스택에 최대 50개의 요소를 저장할 수 있습니다.

int call\_stack[STACK\_SIZE];

정수형 배열 call\_stack을 선언합니다. 이 배열은 함수 호출 시 해당 함수의 정보를 숫자로 저장하는 콜 스택 역할을 합니다. 예를 들어, 함수의 ID나 주소값 등을 저장할 수 있습니다.

char stack\_info[STACK\_SIZE][20];

각 스택 요소에 대한 설명을 문자열 형태로 저장합니다. [20]을 보면 설명 문자열을 최대 19글자까지 쓸 수 있습니다.(1칸은 문자열 종료를 나타내는 \0에 사용)

int SP = -1;

스택 포인터로, SP는 스택의 현재 꼭대기를 가리키는 변수입니다. 이때 스택은 후입선출(LIFO) 방식이기 때문에, 맨 위에 어떤 데이터가 들어왔는지를 추적해야 합니다. 초기값이 -1인 이유는 아직 스택에 아무것도 없다는 뜻입니다. 예시를 들어보겠습니다.

SP = -1 → 스택이 비어 있음

SP = 0 → 첫 번째 데이터가 들어감 (call\_stack[0])

SP = 1 → 두 번째 데이터가 들어감 (call\_stack[1])

데이터를 꺼낼 때는 LIFO 방식에 따라 위에서부터 꺼냅니다.

int FP = -1;

프레임 포인터로, FP는 함수 호출 시 현재 함수의 시작 지점을 기억하기 위한 포인터입니다. 함수가 호출되면 스택에 지역변수, 리턴 주소 등을 저장하는데, 그 함수의 프레임이 생깁니다. FP는 이 함수 프레임의 시작 위치를 저장합니다. 예를 들어보겠습니다.

함수 A에서 함수 B를 호출했을 때, 스택에 함수 A의 정보들이 쌓이고 함수 B가 실행되면서 새로운 프레임이 쌓입니다. 이때 FP는 함수 B의 프레임 시작 위치를 기억하게 됩니다. 나중에 B가 끝나면 FP를 통해 A의 프레임으로 돌아가는 기준점을 찾을 수 있습니다.

void push(int value, const char\* description)

void: 이 함수는 값을 반환하지 않습니다. 즉, 결과값을 반환하는 것이 아니라 단순히 스택에 값을 추가하는 작업만 합니다.

int value: 이 매개변수는 스택에 푸시할 정수형 값입니다.

const char\* description: 이 매개변수는 스택에 푸시하는 값에 대해 설명하는 문자열입니다. const char\*는 문자열을 가리키는 포인터인데, 문자열을 저장할 수 있는 메모리 주소를 가리키고 있습니다.

Push 함수

if (SP < STACK\_SIZE - 1)

아까 보셨던 대로 SP는 스택 포인터, STACK\_SIZE는 스택에 들어갈 수 있는 최대 크기입니다.

SP < STACK\_SIZE – 1: 이 조건은 현재 스택이 꽉 차지 않았는지 확인하는 부분입니다. 이 조건이 참일 때, 즉 스택이 꽉 차지 않았을 때에만 값을 푸시할 수 있습니다.

SP++;

SP의 값을 1 증가시켜 스택에 푸시할 준비를 합니다. 스택에 새로운 값을 넣으려면 먼저 SP를 하나 올려야 합니다.

call\_stack[SP] = value;

이제 SP 위치에 푸시할 값을 저장합니다. call\_stack은 스택 자체를 나타내는 배열이고, SP는 그 배열의 위치입니다. call\_stack[SP]에 value 값을 넣습니다.

snprintf(stack\_info[SP], 20, "%s", description);

stack\_info[SP]: 이 배열은 call\_stack에 저장된 각 값에 대한 설명을 저장하는 곳입니다. SP 위치에 대한 설명이 들어가게 됩니다.

snprintf: 이 함수는 문자열을 포맷에 맞춰 지정된 크기만큼 저장하는 함수입니다.

stack\_info[SP]: 설명을 저장할 곳입니다.

20: 설명이 최대 20글자까지만 저장되도록 지정합니다.

"%s": 포맷 문자열로, 전달된 description 값을 문자열로 받습니다.

description: 설명 문자열이 들어옵니다.

이 구문은 스택에 푸시한 값에 대한 설명을 stack\_info[SP]에 저장하는 역할을 합니다.

함수 작동 흐름 요약:

1. SP 값이 스택의 최대 크기보다 작으면
2. 스택 포인터 SP를 증가시킵니다.
3. 그 위치에 value 값을 저장합니다.
4. 그 위치에 해당 값에 대한 설명도 stack\_info 배열에 저장합니다.

Pop 함수

if (SP >= 0) {

이 조건은 스택이 비어있지 않을 때만 pop 동작을 수행하도록 합니다. SP가 -1이면 스택이 비어 있는 상태이기 때문에 꺼낼 게 없기 때문입니다.

printf("Popping: %d : %s = %d\n", SP, stack\_info[SP], call\_stack[SP]);

이 줄은 현재 꺼내는 값을 화면에 출력합니다.

출력 내용:

SP: 현재 스택 포인터 위치 (즉, 꺼내려는 값이 어디 있는지)

stack\_info[SP]: 해당 위치에 대한 설명

call\_stack[SP]: 실제 스택에 저장된 값

출력의 예시를 들어보겠습니다.

Popping: 2 : return address = 1024

의미: "스택의 2번 위치에 있는 return address라는 설명이 붙은 값 1024를 꺼냅니다."

SP--;

스택에서 값을 제거합니다. SP를 하나 줄이면, 그 위에 있던 값은 더 이상 접근할 수 없게 되므로 값을 꺼낸 효과를 냅니다.

함수 작동 흐름 요약:

1. 스택이 비어 있지 않은지 확인합니다. (SP >= 0)
2. 가장 위에 있는 값(SP 위치의 값)을 출력합니다.
3. SP를 줄여서, 값을 제거합니다.

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

스택 정보를 보여준다는 시작 표시로, 단순한 줄 출력입니다.

for (int i = SP; i >= 0; i--) {

i = SP: SP가 가리키는 맨 위 위치부터 시작해서,

i >= 0: 스택 바닥(인덱스 0)까지 하나씩 내려갑니다.

i--: 한 줄씩 내려가면서 출력합니다.

if (call\_stack[i] != -1)

printf("%d : %s = %d", i, stack\_info[i], call\_stack[i]);

else

printf("%d : %s", i, stack\_info[i]);

call\_stack[i]는 해당 위치에 저장된 값입니다.

stack\_info[i]는 그 값에 대한 설명 문자열입니다.

조건문의 내용은 이렇습니다.

만약 값이 -1이 아니라면 값이 유효하다고 보고 출력합니다.

만약 값이 -1이면 비어 있거나 특별한 값으로 보고 값은 생략합니다.

if (i == SP)

printf(" <=== [esp]\n");

else if (i == FP)

printf(" <=== [ebp]\n");

else

printf("\n");

현재 출력 중인 인덱스가 SP나 FP이면 해당 위치임을 옆에 표시해주는 코드입니다.

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

스택의 정보가 출력이 끝났음을 알려주는 구분선입니다.

여기서 esp, ebp는 CPU안에 진짜 있는 레지스터로, 컴퓨터가 실제 함수를 호출하거나 리턴할 때 자동으로 사용하는 하드웨어 수준의 포인터로, 각각 우리가 직접 구현한 SP,FP에 대응됩니다.

전체적인 함수의 흐름은 이렇습니다.

1. main() 함수가 호출됩니다. func1(1, 2, 3)이 호출되면서 func1의 스택 프레임이 생성됩니다.
2. func1()의 스택 프레임에서 arg1, arg2, arg3, var\_1이 푸시됩니다.
3. func1 안에서 func2(11, 13)가 호출되면서 func2의 스택 프레임이 생성됩니다.
4. func2()의 스택 프레임에서 arg1, arg2, var\_2가 푸시됩니다.
5. func2 안에서 func3(77)가 호출되면서 func3의 스택 프레임이 생성됩니다.
6. func3()의 스택 프레임에서 arg1, var\_3, var\_4가 푸시됩니다.
7. 각 함수의 print\_stack()은 함수 호출 후, 스택의 상태를 출력합니다.
8. 각 함수는 끝나면 팝을 하며 스택 프레임을 정리합니다.

func1(1, 2, 3);

main에서 func1을 호출하면서 스택에 func1의 프레임이 쌓입니다.

push(arg1, "arg1");

push(arg2, "arg2");

push(arg3, "arg3");

push(var\_1, "var\_1");

func1의 인자 arg1 = 1, arg2 = 2, arg3 = 3, 그리고 지역 변수 var\_1 = 100을 스택에 푸시합니다.

func1의 스택 프레임 출력:

==== Func1 Stack ====

4 : var\_1 = 100

3 : arg3 = 3

2 : arg2 = 2

1 : arg1 = 1

====================

func2(11, 13);

func1에서 func2를 호출하면서 func2의 스택 프레임이 쌓입니다.

push(arg1, "arg1");

push(arg2, "arg2");

push(var\_2, "var\_2");

func2의 인자 arg1 = 11, arg2 = 13, 그리고 지역 변수 var\_2 = 200을 스택에 푸시합니다.

func2의 스택 프레임 출력:

==== Func2 Stack ====

3 : var\_2 = 200

2 : arg2 = 13

1 : arg1 = 11

====================

func3(77);

func2에서 func3을 호출하면서 func3의 스택 프레임이 쌓입니다.

push(arg1, "arg1");

push(var\_3, "var\_3");

push(var\_4, "var\_4");

func3의 인자 arg1 = 77, 그리고 지역 변수 var\_3 = 300, var\_4 = 400을 스택에 푸시합니다.

func3의 스택 프레임 출력:

==== Func3 Stack ====

3 : var\_4 = 400

2 : var\_3 = 300

1 : arg1 = 77

====================

pop();

:var\_4

pop();

:var\_3

pop();

:arg1

func3의 스택 프레임을 정리하면서, 먼저 var\_4, var\_3, arg1을 팝하여 스택에서 제거합니다.

func3 종료 후 스택 상태:

==== Func2 Stack ====

3 : var\_2 = 200

2 : arg2 = 13

1 : arg1 = 11

====================

pop();

:var\_2

pop();

:arg2

pop();

:arg1

func2의 스택 프레임을 정리하면서, 먼저 var\_2, arg2, arg1을 팝하여 스택에서 제거합니다.

func2 종료 후 스택 상태:

==== Func1 Stack ====

4 : var\_1 = 100

3 : arg3 = 3

2 : arg2 = 2

1 : arg1 = 1

====================

pop();

:var\_1

:pop();

arg3

:pop();

arg2

:pop();

arg1

func1의 스택 프레임을 정리하면서, 먼저 var\_1, arg3, arg2, arg1을 팝하여 스택에서 제거합니다.

func1 종료 후 스택 상태:

Empty Stack, 즉 스택이 비어있는 상태가 됩니다.