

callstack_2021350024_서기열

Call stack 개략적으로 구현해보기.

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"
=====

*/
#include <stdio.h>
#include <stdarg.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);

void stack_frame_creation(int args, ...);
void stack_frame_delection(int num_of_arg);

/*
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;

    stack_frame_creation(5, 3, arg3, arg2, arg1, var_1);
    print_stack();
    func2(11, 13);
    stack_frame_delection(2);
    print_stack();
}

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

    stack_frame_creation(4, 2, arg2, arg1, var_2);
    print_stack();
    func3(77);
    stack_frame_delection(1);
    print_stack();
}

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

    stack_frame_creation(4, 1, arg1, var_3, var_4);
    print_stack();
}

//main 함수에 관련된 stack frame은 구현하지 않아도 됩니다.
int main()
{

```

```

func1(1, 2, 3);
stack_frame_delection(3);
print_stack();
return 0;
}

void stack_frame_creation(int args, ...)
{
    va_list ap;

    va_start(ap, args);
    int num_of_arg = va_arg(ap, int);
    int num_of_local_var = args - 1 - num_of_arg;

    //매개변수 PUSH
    for (int i = 0; i < num_of_arg; i++)
    {
        SP++;
        call_stack[SP] = va_arg(ap, int);
        strcpy(stack_info[SP], "arg");
        stack_info[SP][3] = '0' + (num_of_arg - i);
        stack_info[SP][4] = '\0';
    }

    //RET PUSH
    SP++;
    call_stack[SP] = -1;
    strcpy(stack_info[SP], "Return Address");

    //SFP PUSH
    SP++;
    call_stack[SP] = FP;

    int var_idx;
    //함수마다 인자의 개수가 다르므로, 이를 이용해 어떤 함수의 SFP인지 확인
    if (num_of_arg == 3)
    {
        strcpy(stack_info[SP], "func1 SFP");
        var_idx = 1;
    }
    else if (num_of_arg == 2)
    {
        strcpy(stack_info[SP], "func2 SFP");
        var_idx = 2;
    }
    else
    {
        strcpy(stack_info[SP], "func3 SFP");
        var_idx = 3;
    }

    FP = SP;

    //지역변수 PUSH
    // 실제 메모리 작동처럼 지역변수 전체 크기에 맞게 SP를 새로 설정.
    SP += num_of_local_var;
    for (int i = FP + 1; i <= SP; i++)
    {
        call_stack[i] = va_arg(ap, int);
        strcpy(stack_info[i], "var_");
        stack_info[i][4] = '0' + (var_idx + i - FP - 1);
        stack_info[i][5] = '\0';
    }
}

```

```

    }

    va_end(ap);
}

void stack_frame_delection(int num_of_arg)
{
    // SFP 복원
    SP = FP;
    FP = call_stack[FP];
    SP--;

    //RET pop
    SP--;

    //매개변수 정리
    SP -= num_of_arg;
}

```

새로운 함수가 호출될 때 스택에는 함수의 정보를 저장하는 하나의 독립적인 블록이 생기며, 이를 스택 프레임이라고 부른다.

SP와 FP가 전역변수이므로 스택 프레임을 만들 때 전부 따로 만들어서 함수마다 구현해도 되지만, 한번에 처리하고 싶어서 함수 2개를 만들었다.

스택 프레임을 형성하는 함수 : `stack_frame_creation`

스택 프레임을 제거하는 함수 : `stack_frame_delection`

1. Stack_frame_creation

```

void stack_frame_creation(int args, ...)
{
    va_list ap;

    va_start(ap, args);
    int num_of_arg = va_arg(ap, int);
    int num_of_local_var = args - 1 - num_of_arg;

    //매개변수 PUSH
    for (int i = 0; i < num_of_arg; i++)
    {
        SP++;
        call_stack[SP] = va_arg(ap, int);
        strcpy(stack_info[SP], "arg");
        stack_info[SP][3] = '0' + (num_of_arg - i);
        stack_info[SP][4] = '\0';
    }

    //RET PUSH
    SP++;
    call_stack[SP] = -1;
    strcpy(stack_info[SP], "Return Address");

    //SFP PUSH
    SP++;
    call_stack[SP] = FP;

    int var_idx;
    //함수마다 인자의 개수가 다르므로, 이를 이용해 어떤 함수의 SFP인지 확인
    if (num_of_arg == 3)
    {

```

```

    strcpy(stack_info[SP], "func1 SFP");
    var_idx = 1;
}
else if (num_of_arg == 2)
{
    strcpy(stack_info[SP], "func2 SFP");
    var_idx = 2;
}
else
{
    strcpy(stack_info[SP], "func3 SFP");
    var_idx = 3;
}

FP = SP;

//지역변수 PUSH
// 실제 메모리 작동처럼 지역변수 전체 크기에 맞게 SP를 새로 설정하도록 수정해야함.
SP += num_of_local_var;
for (int i = FP + 1; i <= SP; i++)
{
    call_stack[i] = va_arg(ap, int);
    strcpy(stack_info[i], "var_");
    stack_info[i][4] = '0' + (var_idx + i - FP - 1);
    stack_info[i][5] = '\0';
}

va_end(ap);
}

```

- 가변인자 처리

```

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

    stack_frame_creation(5, 3, arg3, arg2, arg1, var_1);
    print_stack();
    func2(11, 13);
    stack_frame_delection(2);
    print_stack();
}

```

func1의 경우 매개변수로 (5, 3, arg3, arg2, arg1, var_1) 가 들어온다.

함수마다 매개변수의 개수, 지역함수의 개수가 다르기 때문에 이를 가변인자로 받아와야겠다고 생각했다.

함수 원형의 args에는 받아오는 가변인자의 개수가 저장된다.

3, arg3, arg2, arg1, var_1 총 5개가 들어오므로 args의 값이 5가 된다.

`#stdarg.h` 에는 가변인자를 처리하는 매크로가 있다.

`va_list ap` : 가변인자의 목록을 메모리 주소를 저장하는 포인터 ap 선언

`va_start(ap, args);` 가변인자를 가져올 수 있도록 목록 설정.

`call_stack[SP] = va_arg(ap, int);` : 가변인자 목록에서 int형 자료를 call_stack[SP]에 저장. 이후 ap 포인터가 int 크기만큼 이동함.

처음에 arg3값을 가져왔다면, ap 포인터는 arg2를 가르키게 됨.

- 매개변수 PUSH

```

int num_of_arg = va_arg(ap, int);

for (int i = 0; i < num_of_arg; i++)
{
    SP++;
    call_stack[SP] = va_arg(ap, int);
    strcpy(stack_info[SP], "arg");
    stack_info[SP][3] = '0' + (num_of_arg - i);
    stack_info[SP][4] = '\0';
}

```

num_of_arg 변수에 매개변수의 개수를 저장한다.

`stack_frame_creation` 에서 2번째 인자에 해당하는 값이다.

먼저 스택에 들어가는 매개변수가 앞선 인자가 되도록 값을 넣어줬기 때문에, for문에서 순서대로 넣어주면 된다.

매개변수를 push할경우 stack_info에는 변수 이름을 저장해야 하므로,

num_of_arg - i 값을 따로 입력한다.

- RET PUSH

```

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

```

실제 메모리에서는 Return Address, 즉 다음 EIP의 값을 저장하는데, 여기서는 -1이라는 값을 RET로 설정한다.

- SFP PUSH

```

SP++;
call_stack[SP] = FP;

int var_idx;
//함수마다 인자의 개수가 다르므로, 이를 이용해 어떤 함수의 SFP인지 확인
if (num_of_arg == 3)
{
    strcpy(stack_info[SP], "func1 SFP");
    var_idx = 1;
}
else if (num_of_arg == 2)
{
    strcpy(stack_info[SP], "func2 SFP");
    var_idx = 2;
}
else
{
    strcpy(stack_info[SP], "func3 SFP");
    var_idx = 3;
}

```

스택 프레임을 생성하면서 이전 함수의 base Pointer값을 SFP 위치로 이동시킨다.

그다음 Stack_info의 값에 어떤 함수의 SFP값인지 입력해야하는데, `stack_frame_creation` 함수의 매개변수로 값을 하나 더 보내기에는 비효율적이라고 생각했다.

```

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

```

함수마다 num_of_arg가 다른 것에서 착안하여, num_of_arg 값에 따라서 SFP위치에 있는 값이 어떤 함수의 SFP인지 확인할 수 있도록 코드를 짰다.

var_idx또한 마찬가지로 다음에 나올 지역변수의 index의 값이 무엇인지 표시해두기 위해서 설정했다.

- 지역변수 PUSH

```
int num_of_local_var = args - 1 - num_of_arg;

FP = SP;

// 실제 메모리 작동처럼 지역변수 전체 크기에 맞게 SP를 새로 설정하도록 수정해야함.
SP += num_of_local_var;
for (int i = FP + 1; i <= SP; i++)
{
    call_stack[i] = va_arg(ap, int);
    strcpy(stack_info[i], "var_");
    stack_info[i][4] = '0' + (var_idx + i - FP - 1);
    stack_info[i][5] = '\0';
}

va_end(ap);
```

지역변수 PUSH전에 FP 값을 SP 값으로 갱신해준다.

어셈블리 과정에서 지역변수를 스택 프레임에 저장할 때 하나씩 공간을 확보하지 않고, 함수에서 필요한 총 공간을 미리 정리하고, 함수가 실행될 때 SP를 한 번에 조정한다.

그렇기 때문에 num_of_local_var이라는 변수에 지역변수의 개수를 저장하려고 한다. 함수에 받은 가변인자 개수에서 매개변수의 개수를 담은 값과 매개변수의 개수를 빼면 지역변수의 개수가 된다.

이 값을 미리 SP에 더해서 공간을 확보한다.

이후 지역변수가 선언된 순서대로 값을 call_stack과 stack_info에 저장한다. SFP PUSH할때 설정해두었던 var_idx를 이용해 몇번째 지역변수인지를 표시한다.

2. Stack_frame_delection

```
void stack_frame_delection(int num_of_arg)
{
    // SFP 복원
    SP = FP;
    FP = call_stack[FP];
    SP--;

    //RET pop
    SP--;

    //매개변수 정리
    SP -= num_of_arg;
}
```

스택을 정리할때, 값을 초기화하지 않는다. SP의 위치를 조절함으로써, 스택이 정리되었음을 확인할 수 있다.

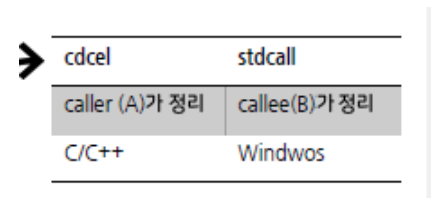
- SFP 복원

SP위치를 FP위치로 갱신한다. 이후 FP에 있는 SFP 값을 이용해 이전 함수의 SFP 위치로 FP를 갱신한다.

- RET pop

RET에 들어있는 다음 명령어의 주소를 EIP에 넣고, RET 사이즈 만큼 SP 를 조정한다. 여기까지가 함수 에필로그 과정으로, 함수가 종료될 때 호출 이전 상태로 스택을 복원한 과정이다.

- 매개변수 정리



cdecl	stdcall
caller (A)가 정리	callee(B)가 정리
C/C++	Windows

Calling convention에 따라서 매개변수는 함수를 호출하는 함수가 정리한다. main 함수에서 func1을 호출하고, func1 함수가 종료되면, main 함수는 매개변수의 개수 * 사이즈의 값만큼 SP를 조절한다.