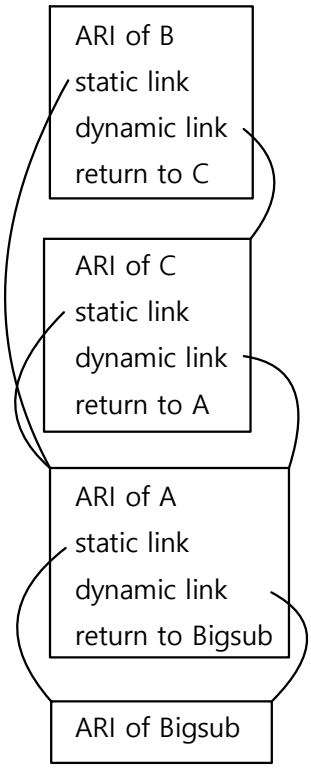
<Homework 6> 6 Problems & Analysis of Assembly & Subprogram

20201635 컴퓨터공학과 전찬

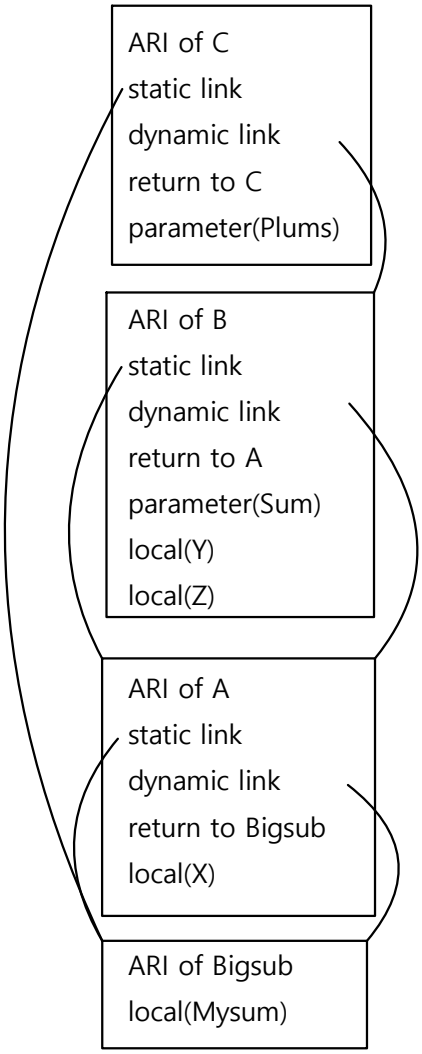
#1-1. Show the stack with all activation record instances, when execution reaches position (1) in the following program

Bigsub 가 실행된 이후, Bigsub() 는 A 를 실행하며, A 는 C 를 실행한다. 또 C 는 B 를 실행하며 (1)의 영역에 도달하게 된다. 따라서 ARI(activation record instace) 또한 위와 같은 순서로 쌓일 것이며, 이에 해당하는 표현은 아래와 같다.



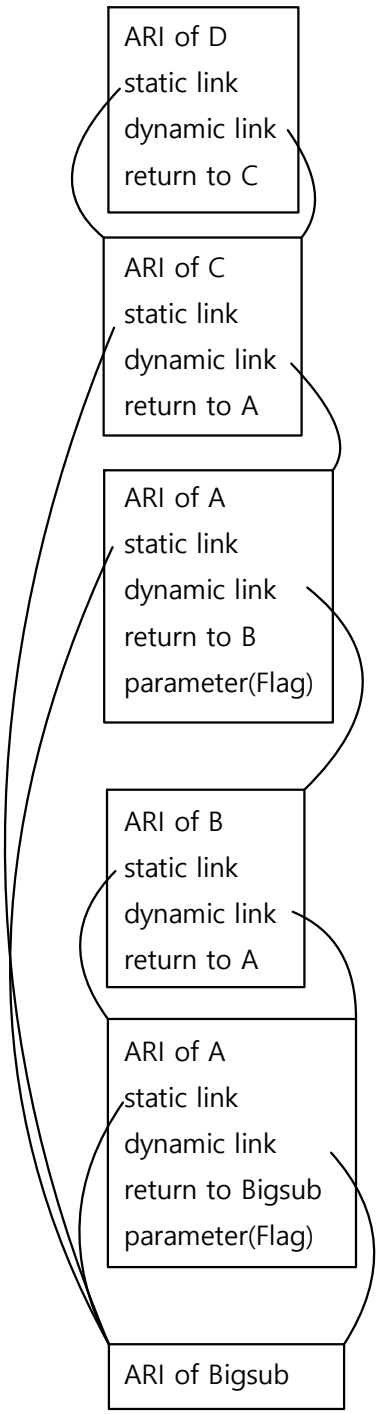
#1-2. Show the stack with all activation record instances, when execution reaches position (1) in the following program

Bigsub -> A -> B -> C 순서로 실행되며, 위에서 추가로 local 변수, parameter 를 갖고 있는 형태이다. 따라서 아래와 같다.

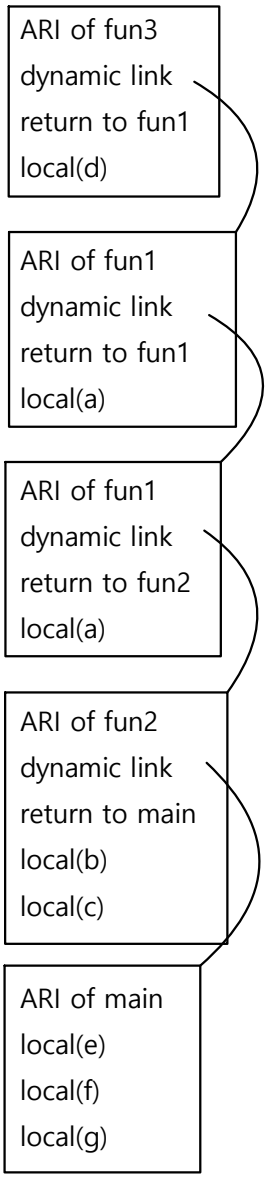


#1-3. Show the stack with all activation record instances, when execution reaches position (1) in the following program

Bigsub -> A ->B -> A -> C ->D 순서로 호출이 이루어지며, A 가 Boolean parameter 을 갖는 형식이다. 따라서 아래와 같다.



#1-4. Show the stack with all activation record instances, when execution reaches position (1) in the following program. This program uses the deep-access method



#1-5. Show the stack with all activation record instances, when execution reaches position (1) in the following program. This program uses the shallow-access method

shallow-access 는 local variable 이 ARI 에 저장되지 않는 형태이다. 따라서 별도의 변수마다 stack 을 보유하며, 이는 아래와 같다.

e -> main
f -> main
g -> main
b -> fun2
c -> fun2
a -> fun1 / fun1
d -> fun3

위에서 앞이 stack 의 bottom, 맨 마지막이 stack 의 top 을 의미한다.

#1-6. what circumstances could the value of local variable in a particular activation retain the value of previous activation in Java?

해당하는 method 의 local variable 이 static 키워드와 함께 static variable 로 정의되어 있다면, 이전 method 실행의 activation 을 보유할 수 있게 된다. 추가로 ARI local offset 이 동일하다면, 초기화하지 않은 변수는 동일한 값을 가질 수 있다. ARI 가 삭제되어도, stack top 이 바뀔 뿐 저장된 정보는 변경되지 않기 때문이다.

#2. Linux 에서 C 언어 프로그램을 작성하고, 이 프로그램에 대한 컴파일러가 생성한 Assembly 프로그램을 분석하여 Linux 운영 체제 상에서 C 언어의 subprogram 수행 방법을 4 가지 관점에서 분석하라.

작성한 코드는 아래와 같다. 3 가지 type (int, int array, struct)에 대해 add 를 수행할 수 있는 함수들을 정의했으며, 추가로 main() 내에 지역 변수가 존재하는 형태이다.

```c
typedef struct Data {
    int a;
    int b;
} data;


int add(int x, int y);
int add_array(int* arr);
int add_struct(data input);

int main(){
    int result;
    int arr[2];
    data int_str;
    int_str.a = 100;
    int_str.b = 200;

    arr[0] = 1;
    arr[1] = 2;
    result = add(arr[0], arr[1]);
    result = add_array(arr);
    result = add_struct(int_str);
}

int add(int x, int y){
    return x + y;
}

int add_array(int* arr){
    return arr[0] + arr[1];
}

int add_struct(data input){
    return add(input.a, input.b);
}
```

위와 같은 코드를 linux server 에서 gcc -S 명령어로 컴파일하면 .s file 을 만들 수 있다. 또한 이를 분석하여 ARI 의 형태와 함께, parameter passing 등을 파악해낼 수 있다. 이를 분석하면 아래와 같다.

1) activation record 구조 : parameter, local 변수, return address, return value 등 저장 순서/방법

저장되어 있는 add function 의 assembly code 를 통해 대체적인 activation record 구조를 파악할 수 있다. 이는 아래와 같다.

```
add:
.LFB1:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -4(%rbp), %edx
    movl    -8(%rbp), %eax
    addl    %edx, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

위 code 를 토대로, activation record 는 movq %rsp, %rbp 를 바탕으로 해당 영역을 설정하며, rsp 에서 sub 한 크기만큼 ARI size 가 할당됨을 파악할 수 있다.(이는 main 등에서 확인 가능하다.) – byte 형식으로 여러 data 가 저장되는 것을 알 수 있다.

2) Subprogram 상에서 parameter 및 local 변수, global 변수 참조 방법 분석
또한 위 코드를 통해, subprogram 상에서 rbp offset 을 바탕으로 -4, -8 와 같이 parameter 에 접근하는 것을 파악할 수 있다. main 내의 local variable 에 접근할 때도, 동일한 형식으로 rbp 를 바탕으로 접근함을 아래 코드를 통해 파악할 수 있다.

```
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $48, %rsp
    movq    %fs:40, %rax
    movq    %rax, -8(%rbp)
    xorl    %eax, %eax
    movl    $100, -32(%rbp)
    movl    $200, -28(%rbp)
```

3) 다양한 데이터 타입에 대한 Parameter Passing 방법

3 개의 함수의 비교를 통해 int, pointer, structure 에서 parameter passing 방법을 비교해볼 수 있다. int 를 전달할 때는, -4(%rbp) 와 같은 형식으로 데이터를 저장하며, pointer(array)를 전달할 때는 -8(%rbp) 형식으로 전달한다. 따라서 pointer size = 8byte 임을 파악할 수도 있다. 또한 전달한 array 의 원소 접근에는 movl 와 함께 addq $4 형식으로 주소값을 4 씩 늘려가면서 접근을 수행함을 파악할 수 있다. 마지막으로 struct 를 전달하는 경우에, 해당 struct 의 element 값을 분해해서 하나하나씩 저장함을 코드를 통해 파악할 수 있다.

4) Return value 전달 방식 분석

코드를 살펴보면, return value 를 eax 에 저장하고 main 에서 저장한 eax 를 바탕으로 return 값을 받는 형식으로 수행됨을 파악할 수 있다. 이번 코드에서는 return 값이 모두 4byte integer 이라 eax 를 사용했는데, 만약 Pointer return 등인 경우에는, rax 에 저장할 것이라고 예측해볼 수도 있다.

++ 전체 코드는 아래와 같다.

```
        .file   "ass.c"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $48, %rsp
        movq    %fs:40, %rax
        movq    %rax, -8(%rbp)
        xorl    %eax, %eax
        movl    $100, -32(%rbp)
        movl    $200, -28(%rbp)
        movl    $1, -16(%rbp)
        movl    $2, -12(%rbp)
        movl    -12(%rbp), %edx
        movl    -16(%rbp), %eax
        movl    %edx, %esi
        movl    %eax, %edi
        call    add
        movl    %eax, -36(%rbp)
        leaq    -16(%rbp), %rax
        movq    %rax, %rdi
        call    add_array
        movl    %eax, -36(%rbp)
```

```
        movq    -32(%rbp), %rax
        movq    %rax, %rdi
        call    add_struct
        movl    %eax, -36(%rbp)
        movl    $0, %eax
        movq    -8(%rbp), %rcx
        xorq    %fs:40, %rcx
        je      .L3
        call    __stack_chk_fail
.L3:
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .globl  add
        .type   add, @function
add:
.LFB1:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    %edi, -4(%rbp)
        movl    %esi, -8(%rbp)
        movl    -4(%rbp), %edx
        movl    -8(%rbp), %eax
        addl    %edx, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE1:
        .size   add, .-add
        .globl  add_array
        .type   add_array, @function
add_array:
.LFB2:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
```

```
        .cfi_def_cfa_register 6
        movq    %rdi, -8(%rbp)
        movq    -8(%rbp), %rax
        movl    (%rax), %edx
        movq    -8(%rbp), %rax
        addq    $4, %rax
        movl    (%rax), %eax
        addl    %edx, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE2:
        .size   add_array, .-add_array
        .globl  add_struct
        .type   add_struct, @function
add_struct:
.LFB3:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movq    %rdi, -16(%rbp)
        movl    -12(%rbp), %edx
        movl    -16(%rbp), %eax
        movl    %edx, %esi
        movl    %eax, %edi
        call    add
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE3:
        .size   add_struct, .-add_struct
        .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0 20160609"
        .section .note.GNU-stack,"",@progbits
```