**시스템 소프트웨어와 실습**

**연습과제 #2**

**보고서**

**이름: 정여준**

**학번: 2017112138  
학과: 컴퓨터공학과  
교수님: 문봉교 교수님**

**강의명: 시스템소프트웨어와 실습**

**1. Chapter 3의 Homework problem 3.60, 3.67, 3.72를 해결하고 보고서에 포함한다.**

**<3.60>**

Consider the following assembly code:

long loop(long x, int n)

x in %rdi, n in %esi 1 x는 %rdi이고 초기값 0, n은 %esi이고 초기값 1

loop:

movl %esi, %ecx

movl $1, %edx

movl $0, %eax

jmp .L2

.L3:

movq %rdi, %r8

andq %rdx, %r8

orq %r8, %rax

salq %cl, %rdx

.L2:

testq %rdx, %rdx

jne .L3

rep; ret

The preceding code was generated by compiling C code that had the following overall form:

long loop(long x, long n)

{

long result =\_\_\_\_\_\_\_\_\_\_ ;

long mask;

for (mask =\_\_\_\_\_\_\_\_ ; mask \_\_\_\_\_\_\_\_; mask =\_\_\_\_\_\_\_\_\_ ) {

result |= \_\_\_\_\_\_;

}

return result;

}

Your task is to fill in the missing parts of the C code to get a program equivalent to the generated assembly code. Recall that the result of the function is returned in register %rax. You will find it helpful to examine the assembly code before, during, and after the loop to form a consistent mapping between the registers and the program variables.

A. Which registers hold program values x, n, result, and mask?

| **val** | **reg** |
| --- | --- |
| x | %rdi |
| n | %esi |
| result | %rax |
| mask | %rdx |

B. What are the initial values of result and mask?

result = 0 , mask = 1

C. What is the test condition for mask?

mask != 0

D. How does mask get updated?

mask는 매번 왼쪽으로 n자리씩 이동

mask = mask << n

E. How does result get updated?

x에 의해서 결과가 수정된다.

F. Fill in all the missing parts of the C code

long loop2(long x, int n)

long result = 0;

long mask;

for(mask = 1; mask != 0; mask<<=n){

result |= (x & mask);

}

return result;

}

**<3.67>**

For this exercise, we will examine the code generated by gcc for functions that have structures as arguments and return values, and from this see how these language features are typically implemented. The following C code has a function process having structures as argument and return values, and a function eval that calls process:

typedef struct {

long a[2];

long \*p;

} strA;

typedef struct {

long u[2];

long q;

} strB;

strB process(strA s) {

strB r;

r.u[0] = s.a[1];

r.u[1] = s.a[0];

r.q = \*s.p;

return r; 1

}

long eval(long x, long y, long z) {

strA s;

s.a[0] = x;

s.a[1] = y;

s.p = &z;

strB r = process(s);

return r.u[0] + r.u[1] + r.q;

}

Gcc generates the following code for these two functions: strB process(strA s)

process:

movq %rdi, %rax

movq 24(%rsp), %rdx

movq (%rdx), %rdx

movq 16(%rsp), %rcx

movq %rcx, (%rdi)

movq 8(%rsp), %rcx

movq %rcx, 8(%rdi)

movq %rdx, 16(%rdi)

ret

long eval(long x, long y, long z) x in %rdi, y in %rsi, z in %rdx 1

eval:

subq $104, %rsp

movq %rdx, 24(%rsp)

leaq 24(%rsp), %rax

movq %rdi, (%rsp)

movq %rsi, 8(%rsp)

movq %rax, 16(%rsp)

leaq 64(%rsp), %rdi

call process

movq 72(%rsp), %rax

addq 64(%rsp), %rax

addq 80(%rsp), %rax

addq $104, %rsp

ret

A. We can see on line 2 of function eval that it allocates 104 bytes on the stack. Diagram the stack frame for eval, showing the values that it stores on the stack prior to calling process.

104 +-------------------+

| |

| |

| |

| |

| |

| |

| |

| |

64 +-------------------+ <-- %rdi

| |

| |

| |

| |

| |

| |

32 +-------------------+

| z |

24 +-------------------+

| &z |

16 +-------------------+

| y |

8 +-------------------+

| x |

0 +-------------------+ <-- %rsp

B. What value does eval pass in its call to process?

새로운 주소 %rsp+64를 전달

C. How does the code for process access the elements of structure argument s?

%rdi말고 %rsp+offset으로 접근

D. How does the code for process set the fields of result structure r?

%rsp+64 주소 프로세스에게 pass. 이 주소부터 시작으로 하여 저장 데이터 처리하고 최종적으로 이 주소를 반환한다.

E. Complete your diagram of the stack frame for eval, showing how eval accesses the elements of structure r following the return from process.

104 +-------------------+

| |

| |

| |

| |

| |

| |

88 +-------------------+

| z |

80 +-------------------+

| x |

72 +-------------------+

| y |

64 +-------------------+ <-- %rdi(eval pass in)

| | |

| | -- %rax(process pass out)

| |

| |

| |

| |

32 +-------------------+

| z |

24 +-------------------+

| &z |

16 +-------------------+

| y |

8 +-------------------+

| x |

0 +-------------------+ <-- %rsp in eval

| |

-8 +-------------------+ <-- %rsp in process

F. What general principles can you discern about how structure values are passed as function arguments and how they are returned as function results?

Caller가 공간을 찾고 그 공간의 주소를 Callee에게 전달한다. Callee는 데이터를 이 공간에 저장하고 이 주소를 반환한다.

**<3.72>**

Figure 3.54(a) shows the code for a function that is similar to function vfunct (Figure 3.43(a)). We used vfunct to illustrate the use of a frame pointer in managing variable-size stack frames. The new function aframe allocates space for local

(a) C code

#include

long aframe(long n, long idx, long \*q) {

long i;

long \*\*p = alloca(n \* sizeof(long \*));

p[0] = &i;

for (i = 1; i < n; i++)

p[i] = q;

return \*p[idx];

}

(b) Portions of generated assembly code

long aframe(long n, long idx, long \*q)

n in %rdi, idx in %rsi, q in %rdx

aframe:

pushq %rbp

movq %rsp, %rbp

subq $16, %rsp Allocate space for i (%rsp = s1)

leaq 30(,%rdi,8), %rax

andq $-16, %rax

subq %rax, %rsp Allocate space for array p (%rsp = s2)

leaq 15(%rsp), %r8

andq $-16, %r8 Set %r8 to &p[0]

array p by calling library function alloca. This function is similar to the more commonly used function malloc, except that it allocates space on the run-time stack. The space is automatically deallocated when the executing procedure returns. Figure 3.54(b) shows the part of the assembly code that sets up the frame pointer and allocates space for local variables i and p. It is very similar to the corresponding code for vframe. Let us use the same notation as in Problem 3.49: The stack pointer is set to values s1 at line 4 and s2 at line 7. The start address of array p is set to value p at line 9. Extra space e2 may arise between s2 and p, and extra space e1 may arise between the end of array p and s1.

A. Explain, in mathematical terms, the logic in the computation of s2.

s2 = s1 – [(n\*8+30)&0XFFFFFFF0]

n이 홀수일 때, s2 = s1 – (n\*8+24)

n이 짝수일 때, s2 = s1 – (n\*8+16)

B. Explain, in mathematical terms, the logic in the computation of p.

p = ( s2 + 15 )&0XFFFFFFF0

s2보다 큰 16의 최소 배수

C. Find values of n and s1 that lead to minimum and maximum values of e1.

|  | **e1** | **n** | **s1** |
| --- | --- | --- | --- |
| **최소** | **1** | **even** | **n%16==1** |
| **최대** | **24** | **odd** | **n%16==0** |

최소:

e1이 0이면 p과 s2는 동일해야 하므로 e1은 0이 될 수 없다.

n이 짝수면 e1 + e2는 16이고 만약 e2가 15라면 e1은 최소가 되어야 한다. 그래서 n%16==1이고 s1 == n이다.

최대:

n이 홀수일 때 e1 + e2는 24이다. 만약 p가 s2와 동일하다면 e2는 0이고 e1의 최댓값은 24가 된다. 따라서, n%16==0이고 s1==n이다.

D. What alignment properties does this code guarantee for the values of s2 and p?

p에는 16이 할당된다.

s2는 8\*n 크기의 공간을 보존하는 16의 최소 배수이다.

**2. Chapter 7의 Homework problem 7.6, 7.9, 7.12를 해결하고 보고서에 포함한다.**

**<7.6>**

This problem concerns the m.o module from Figure 7.5 and the following version of the swap.c function that counts the number of times it has been called:

extern int buf[];

int \*bufp0 = &buf[0];

static int \*bufp1;

static void incr()

{

static int count=0;

count++;

}

void swap()

{

int temp;

incr();

bufp1 = &buf[1];

temp = \*bufp0;

\*bufp0 = \*bufp1;

\*bufp1 = temp;

}

For each symbol that is defined and referenced in swap.o, indicate if it will have a symbol table entry in the .symtab section in module swap.o. If so, indicate the module that defines the symbol (swap.o or m.o), the symbol type (local, global, or extern), and the section (.text, .data, or .bss) it occupies in that module.

| **Symbol** | **.Symtab entry?** | **Symbol type** | **module** | **Section** |
| --- | --- | --- | --- | --- |
| buf | Yes | external | m | .data |
| bufp0 | Yes | global | swap | .data |
| bufp1 | Yes | local | swap | .bss |
| swap | Yes | global | swap | .text |
| temp | No | ------ | ----- | ------ |
| incr | Yes | local | swap | .text |
| count | Yes | local | swap | .bss |



**<7.9>**

Consider the following program, which consists of two object modules:

/\* foo6.c \*/

void p2(void);

int main()

{

p2();

return 0;

}

/\* bar6.c \*/

#include<stdio.h>

char main;

void p2()

{

printf("0x%x\n", main);

}

When this program is compiled and executed on an x86-64 Linux system, it prints the string 0x48\n and terminates normally, even though function p2 never initializes variable main. Can you explain this?

char main을 unsigned int main으로 수정한다.

그럼 코드는 다음과 같이 수정된다.

/\* bar6.c \*/

#include<stdio.h>

unsigned int main;

void p2()

{

printf("0x%x\n", main);

}

/\* foo6.c \*/

void p2(void);

void offset(void)

{

return;

}

int main(int argc, char\* argv[])

{

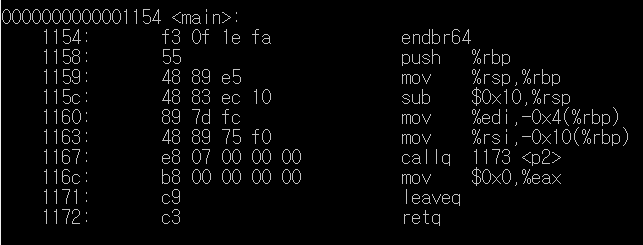
p2();

return 0;

}



main함수를 찾으면 다음과 같다.



다음과 같이 작동한다.

/\* 2-bar6.c \*/

#include<stdio.h>

int main(int argc, char\* argv[]);

void p2()

{

printf("0x%x\n", \*(unsigned int \*)main);

}

**<7.12>**

Consider the call to function swap in object file m.o (Problem 7.6).

9: e8 00 00 00 00 callq e <main+0xe> swap()

with the following relocation entry:

r.offset = 0xa

r.symbol = swap

r.type = R\_X86\_64\_PC32

r.addend = -4

A. Suppose that the linker relocates .text in m.o to address 0x4004e0 and swap to address 0x4004f8. Then what is the value of the relocated reference to swap in the callq instruction?

ADDR(s) = ADDR(.text) = 0x4004e0

ADDR(r.symbol) = ADDR(swap) = 0x4004f8

refaddr = ADDR(s) + r.offset = 0x4004ea

\*refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr) = 0x

B. Suppose that the linker relocates .text in m.o to address 0x4004d0 and swap to address 0x400500. Then what is the value of the relocated reference to swap in the callq instruction?

ADDR(s) = ADDR(.text) = 0x4004d0

ADDR(r.symbol) = ADDR(swap) = 0x400500

refaddr = ADDR(s) + r.offset = 0x4004da

\*refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr) = 0x22

**3. Chapter 8의 Homework problem 8.9, 8.18, 8.23을 해결하고 보고서에 포함한다.**

**<8.9>**

Consider four processes with the following starting and ending times:

| **Process** | **Start time** | **End time** |
| --- | --- | --- |
| **A** | **6** | **8** |
| **B** | **3** | **5** |
| **C** | **4** | **7** |
| **D** | **2** | **9** |

For each pair of processes, indicate whether they run concurrently (Y) or not (N):

| **Process pair** | **Concurrent?** |
| --- | --- |
| **AB** | **N** |
| **AC** | **Y** |
| **AD** | **Y** |
| **BC** | **Y** |
| **BD** | **Y** |
| **CD** | **Y** |

**<8.18>**

Consider the following program:

#include "csapp.h"

void end(void)

{

printf("2"); fflush(stdout);

}

int main()

{

if (Fork() == 0)

atexit(end);

if (Fork() == 0) {

printf("0"); fflush(stdout);

}

else {

printf("1");

fflush(stdout);

}

exit(0);

}

Determine which of the following outputs are possible. Note: The atexit function takes a pointer to a function and adds it to a list of functions (initially empty) that will be called when the exit function is called.

A. 112002

B. 211020

C. 102120

D. 122001

E. 100212

**프로그램을 나타내면 다음과 같다.**

**c**

**+-------+---------+**

**| "0" exit "2"**

**|**

**c | p**

**+-------+-------+---------+**

**| fork "1" exit "2"**

**| (atexit)**

**| c**

**| +-------+---------+**

**| | "0" exit**

**| |**

**| p | p**

**+------+-------+---------+---------+**

**main fork fork "1" exit**

2는 0 또는 1 뒤에 와야 한다. 따라서 B와 D가 불가능하다.

**<8.23>**

One of your colleagues is thinking of using signals to allow a parent process to count events that occur in a child process. The idea is to notify the parent each time an event occurs by sending it a signal and letting the parent’s signal handler increment a global counter variable, which the parent can then inspect after the child has terminated. However, when he runs the test program in Figure 8.45 on his system, he discovers that when the parent calls printf, counter always has a value of 2, even though the child has sent five signals to the parent. Perplexed, he comes to you for help. Can you explain the bug?

#include "csapp.h"

int counter = 0;

void handler(int sig)

{

counter++;

sleep(1); /\* Do some work in the handler \*/

return;

}

int main()

{

int i;

Signal(SIGUSR2, handler);

if (Fork() == 0) { /\* Child \*/

for (i = 0; i < 5; i++) {

Kill(getppid(), SIGUSR2);

printf("sent SIGUSR2 to parent\n");

}

exit(0);

}

Wait(NULL);

printf("counter=%d\n", counter);

exit(0);

}

<figure 8.45>

SIGUSR2 SIGUSR2 SIGUSR2 SIGUSR2 SIGUSR2

| | | | |

being handled Pending Canceld Canceld Canceld

1초

위의 그림과 같이 한번에 한 개의 signal만을 처리할 수 있다. 다른 signal은 취소된다

그래서 5개의 signal을 보내도 카운터는 항상 2의 값을 갖게 된다.

두가지 방법으로 수정여 이러한 버그를 처리할 수 있다.

1. sleep을 없애는 방법

sleep(1); 코드를 없애는 것이다.

2. 더 많은 signal을 보내는 방법

for(i=0; i<500000; i++)