Lecture Note 4.
Process Structure

September 26, 2020

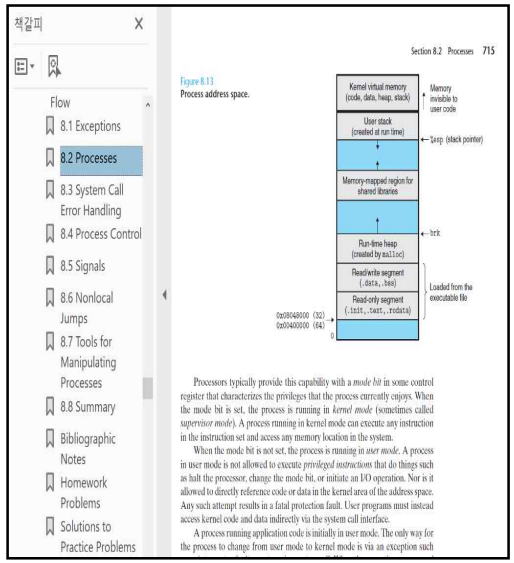
Jongmoo Choi
Dept. of Software
Dankook University
http://embedded.dankook.ac.kr/~choijm

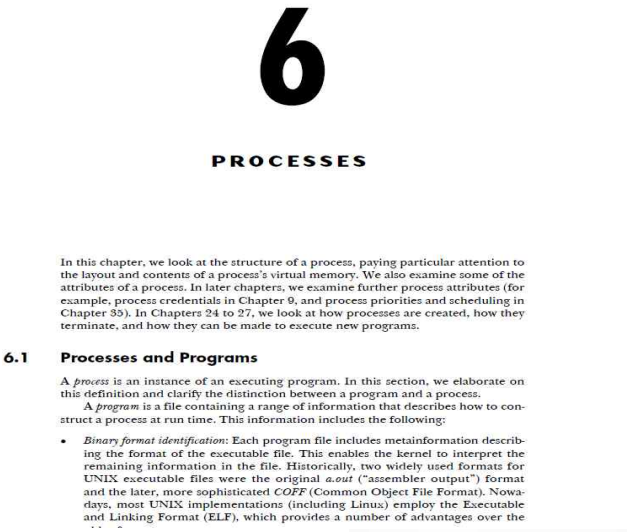
(Copyright © 2020 by Jongmoo Choi, All Rights Reserved. Distribution requires permission)

Objectives

- Understand the definition of a process
- Explore the process structure
- Discuss the relation between program and process structure
- Grasp the details of stack

- Refer to Chapter 8 in the CSAPP and Chapter 6 in the LPI





6

PROCESSES

In this chapter, we look at the structure of a process, paying particular attention to the layout and contents of a process's virtual memory. We also examine some of the attributes of a process. In later chapters, we examine further process attributes (for example, process credentials in Chapter 9, and process priorities and scheduling in Chapter 35). In Chapters 24 to 27, we look at how processes are created, how they terminate, and how they can be made to execute new programs.

6.1 Processes and Programs

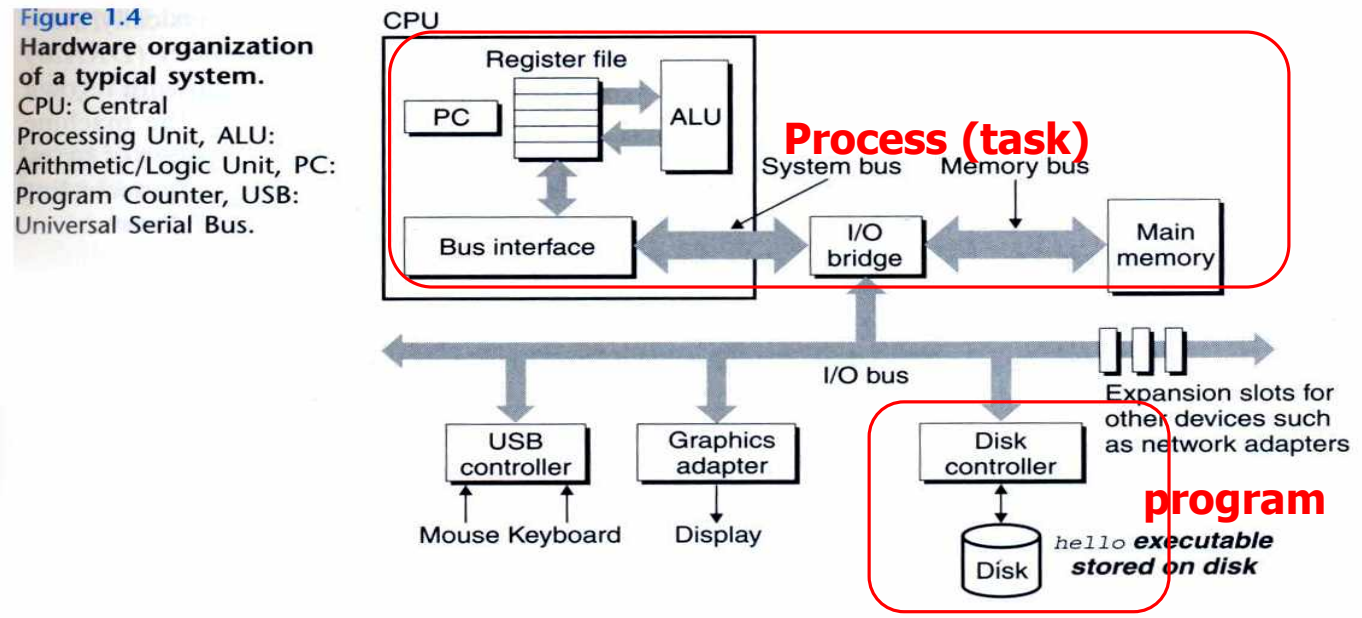
A *process* is an instance of an executing program. In this section, we elaborate on this definition and clarify the distinction between a program and a process.

A *program* is a file containing a range of information that describes how to construct a process at run time. This information includes the following:

- *Binary format identification*: Each program file includes metainformation describing the format of the executable file. This enables the kernel to interpret the remaining information in the file. Historically, two widely used formats for UNIX executable files were the original *a.out* ("assembler output") format and the later, more sophisticated *COFF* (Common Object File Format). Nowadays, most UNIX implementations (including Linux) employ the Executable and Linking Format (ELF), which provides a number of advantages over the

Process Definition (1/2)

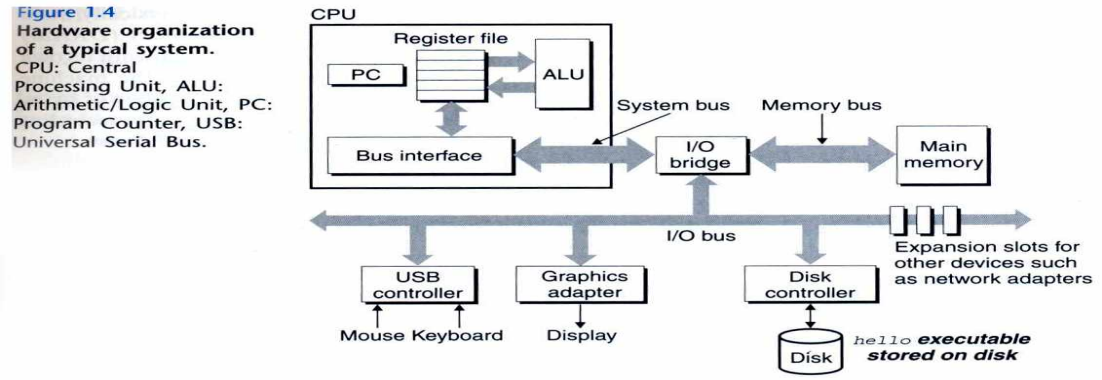
- What is a process (also called as task)?
  - ✓ Program in execution
  - ✓ Having its own memory space and CPU registers
  - ✓ Scheduling entity
  - ✓ Conflict each other for resource allocation
  - ✓ Parent-child relation (family)



(Source: CSAPP)

Process Definition (2/2)

- Related terminology
  - ✓ Load
    - from disk into main memory
    - disk: file system (LN 3)
    - main memory: virtual memory (CSAPP 9, OS Course)
    - carried out by OS (e.g. page fault mechanism)
  - ✓ Fetch
    - From memory into CPU
    - instruction fetch and data fetch (LN 7)
    - carried out by hardware



Process Structure (1/6)

- Conceptual structure
  - ✓ text, data, heap, stack

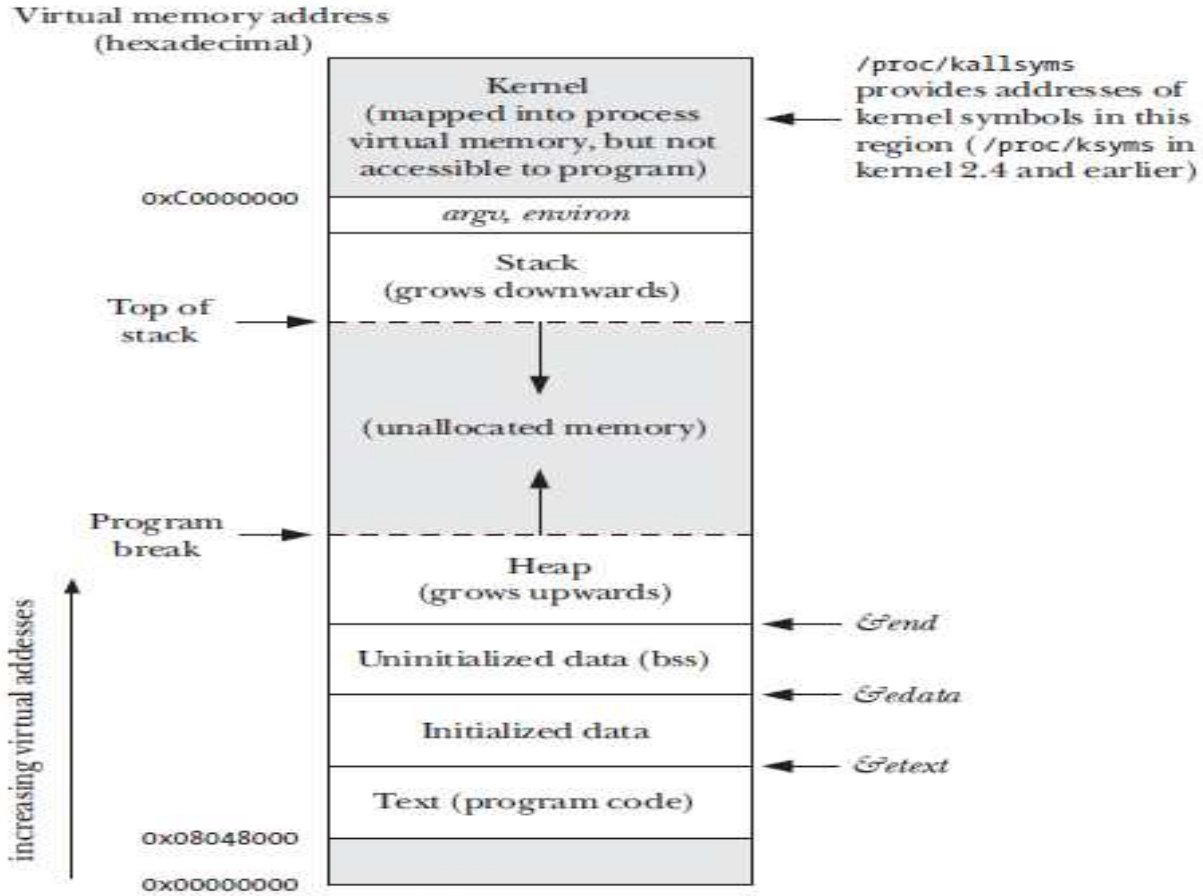


Figure 6-1: Typical memory layout of a process on Linux/x86-32

(Source: LPI)

Process Structure (2/6)

- Process structure in C program: function pointer

```
/* f_pointer.c: for function pointer exercise, by choijm, choijm@dku.edu */
#include <stdio.h>

int a = 10;

int func1(int arg1)
{
   printf("In func1: arg1 = %d\n", arg1);
}

main()
{
   int *pa;
   int (*func_ptr)(int);

   pa = &a;
   printf("pa = %p, *pa = %d\n", pa, *pa);
   func1(3);

   func_ptr = func1;
   func_ptr(5);

   printf("Bye..^^\n");
}
```

Process Structure (3/6)

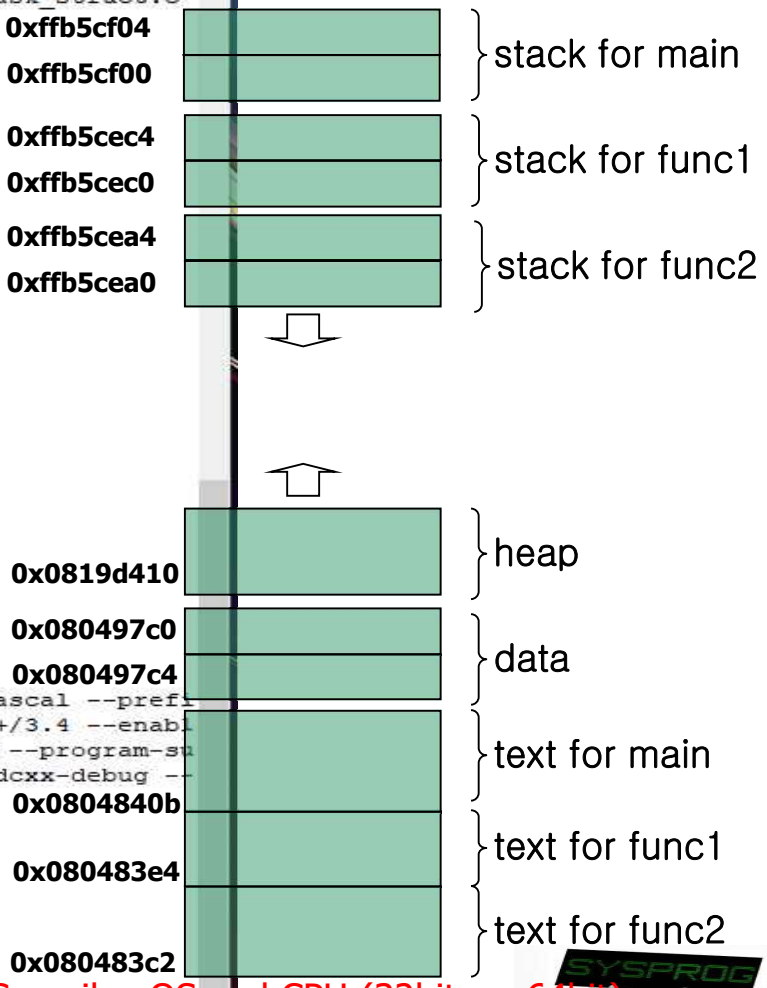
- Process structure in C program: address printing

```
/* task_struct.c: display addresses of variables and functions, choijm@dku.edu  */
#include <stdlib.h>
#include <stdio.h>

int glob1, glob2;

int func2() {
   int f2_local1, f2_local2;

   printf("func2 local: \n\t%p, \n\t%p\n", &f2_local1, &f2_local2);
}

int func1() {
   int f1_local1, f1_local2;

   printf("func1 local: \n\t%p, \n\t%p\n", &f1_local1, &f1_local2);
   func2();
}

main(){
   int m_local1, m_local2; int *dynamic_addr;

   printf("main local: \n\t%p, \n\t%p\n", &m_local1, &m_local2);
   func1();

   dynamic_addr = malloc(16);
   printf("dynamic: \n\t%p\n", dynamic_addr);
   printf("global: \n\t%p, \n\t%p\n", &glob1, &glob2);
   printf("functions: \n\t%p, \n\t%p, \n\t%p\n", main, func1, func2);
}
```

Process Structure (4/6)

- Process structure in C program: address printing

```
choijm@embedded: ~/Syspro/chap4
choijm@embedded:~/Syspro/chap4$ ls
a.out  f_pointer.c  stack_destroy.c  stack_struct.c  task_struct  task_struct.c
choijm@embedded:~/Syspro/chap4$
choijm@embedded:~/Syspro/chap4$ vi task_struct.c
choijm@embedded:~/Syspro/chap4$
choijm@embedded:~/Syspro/chap4$ gcc -o task_struct task_struct.c
choijm@embedded:~/Syspro/chap4$
choijm@embedded:~/Syspro/chap4$ ./task_struct
main local:
        0xffb5cf04,
        0xffb5cf00
funcl local:
        0xffb5cec4,
        0xffb5cec0
func2 local:
        0xffb5cea4,
        0xffb5cea0
dynamic:
        0x819d410
global:
        0x80497c0,
        0x80497c4
functions:
        0x804840b,
        0x80483e4,
        0x80483c2
choijm@embedded:~/Syspro/chap4$
choijm@embedded:~/Syspro/chap4$ gcc -v
Reading specs from /usr/lib/gcc/i486-linux-gnu/3.4.6/specs
Configured with: ../src/configure -v --enable-languages=c,c++,f77,pascal --prefi
x=/usr --libexecdir=/usr/lib --with-gxx-include-dir=/usr/include/c++/3.4 --enabl
e-shared --with-system-zlib --enable-nls --without-included-gettext --program-su
ffix=-3.4 --enable-__cxa_atexit --enable-clocale=gnu --enable-libstdcxx-debug --
with-tune=i686 i486-linux-gnu
Thread model: posix
gcc version 3.4.6 (Debian 3.4.6-5)
choijm@embedded:~/Syspro/chap4$
choijm@embedded:~/Syspro/chap4$ date
2020. 09. 28. (월 ) 16:51:06 KST
choijm@embedded:~/Syspro/chap4$
```



☞ Addresses can be different based on Compiler, OS and CPU (32bit vs. 64bit)

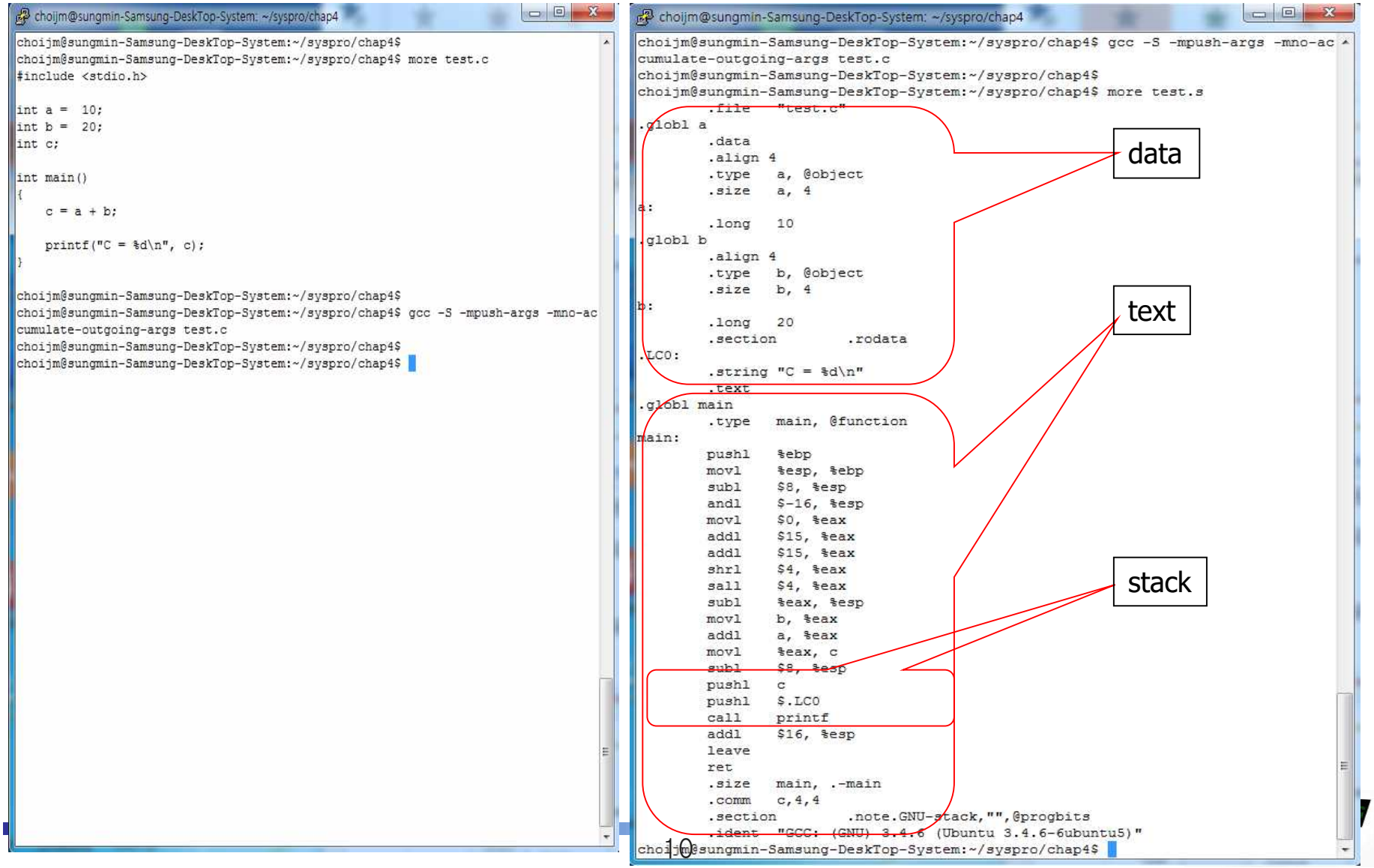
Process Structure (5/6)

- Summary
  - ✓ Process: consist of four regions, text, data, stack and heap

    Also called as segment or vm_object
  - ✓ Text
    - Program code (assembly language)
    - Go up to the higher address according to coding order
  - ✓ Data
    - Global variable
    - Initialized and uninitialized data are managed separately (for the performance reason)
  - ✓ Stack
    - Local variable, argument, return address
    - Go down to the lower address as functions invoked
  - ✓ Heap
    - Dynamic allocation area (malloc(), calloc(), …)
    - Go up to the higher address as allocated

Process Structure (6/6)

- Relation btw program and process



Process Structure in CSAPP

- Another viewpoint for process structure
  - ✓ text, data, heap, stack + shared region, kernel

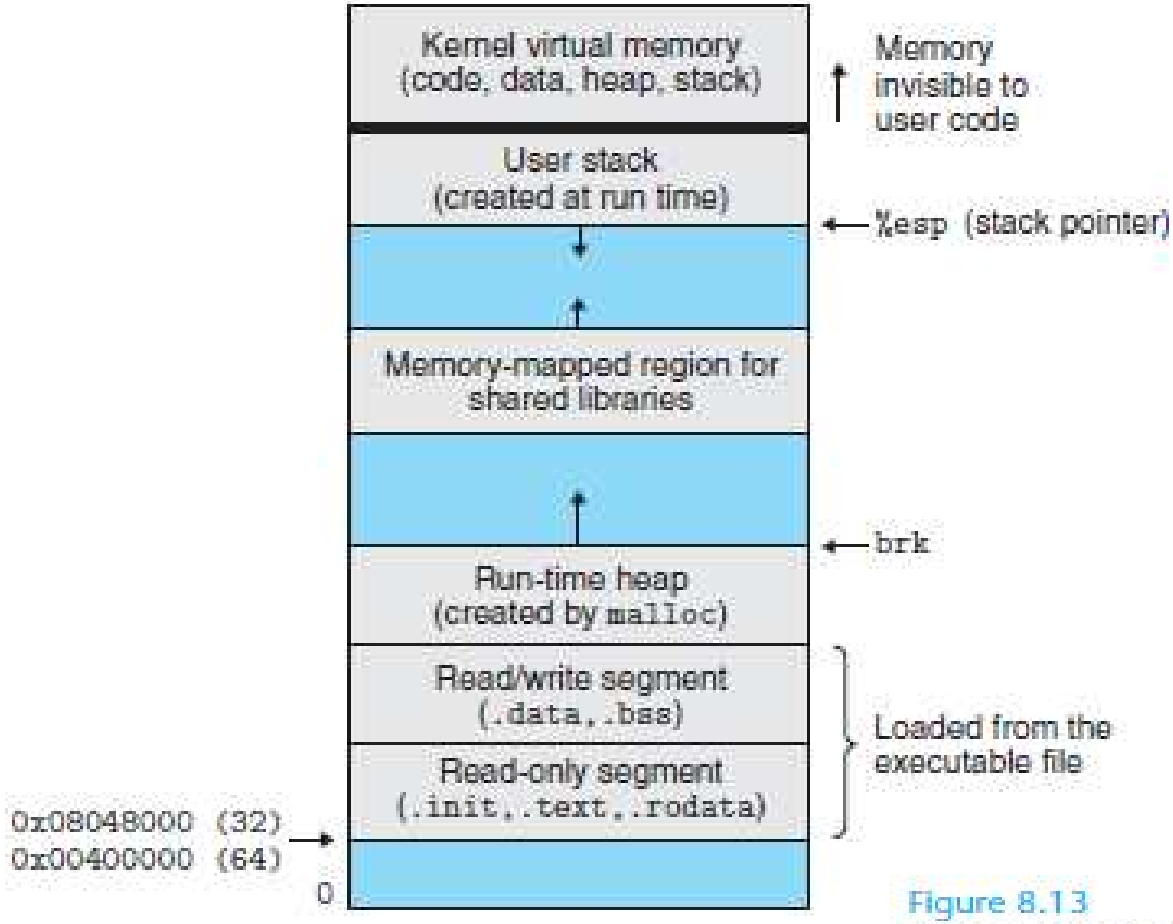


Figure 8.13
Process address space.

(Source: CSAPP)

Stack Details (1/6)

- What is Stack?
  - ✓ A contiguous array of memory locations with LIFO property
    - Stack operation: push and pop
    - Stack management: bottom and top (e.g. SS and ESP in intel)

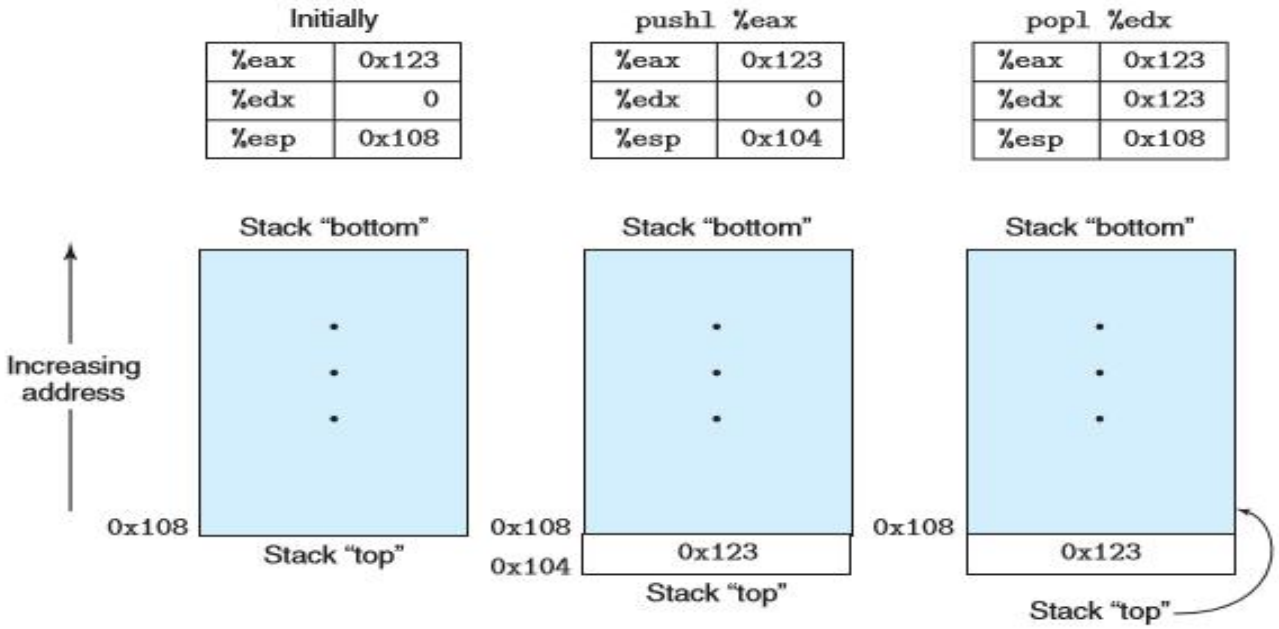
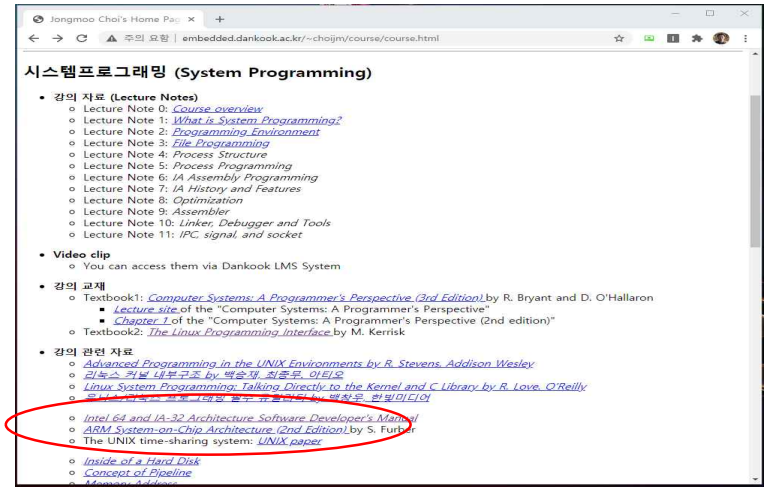


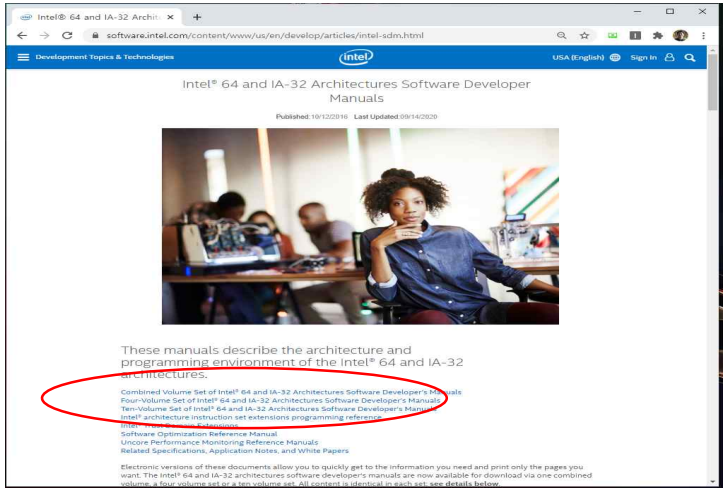
Figure 3.5 **Illustration of stack operation.** By convention, we draw stacks upside down, so that the "top" of the stack is shown at the bottom. IA32 stacks grow toward lower addresses, so pushing involves decrementing the stack pointer (register %esp) and storing to memory, while popping involves reading from memory and incrementing the stack pointer.

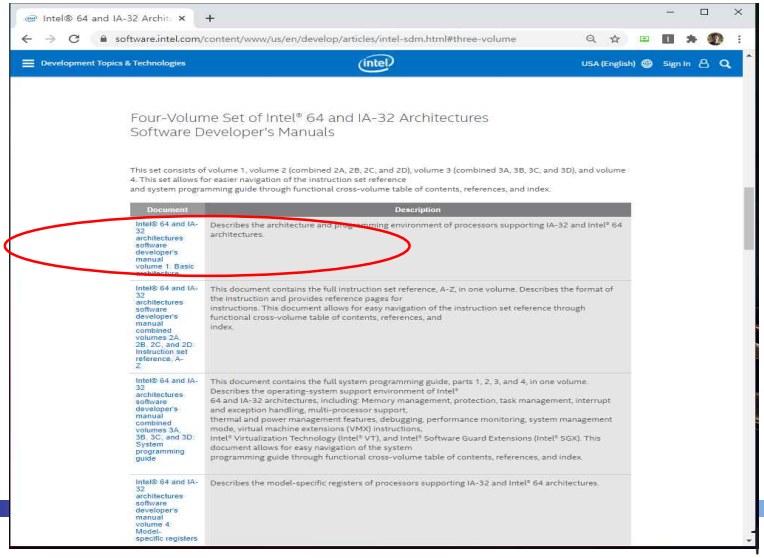
(Source: CSAPP)

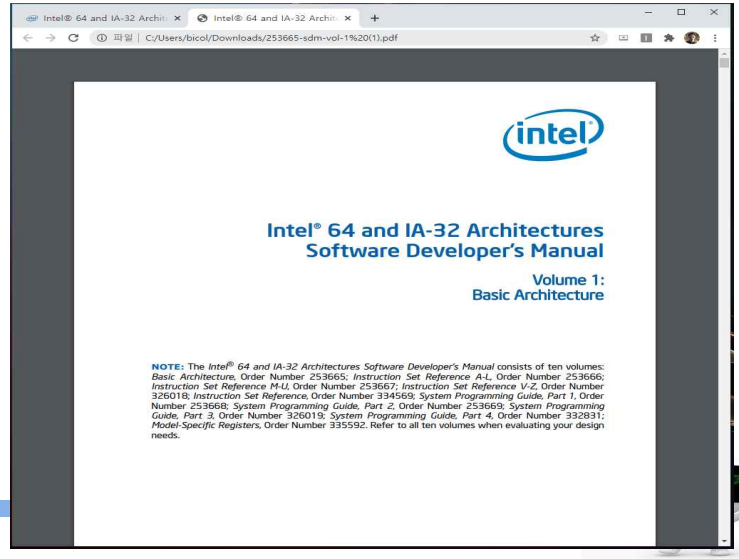
Stack Details (2/6)

- Stack in Intel architecture
  - ✓ How to access Intel manual?









Stack Details (3/6)

- Stack in Intel architecture
  - ✓ Real manipulation of push and pop
    - ESP (Extended Stack Pointer): pointing the top position (LN 6)
    - push: decrement the ESP and write data at the top of stack (down)
    - pop: read data from the top and increment the ESP (up)
  - ✓ What are in the stack?
    - 1) argument (parameters), 2) return address, 3) local variable, …
    - Return address: an address that returns after finishing a function (usually an address of an instruction after “call”)

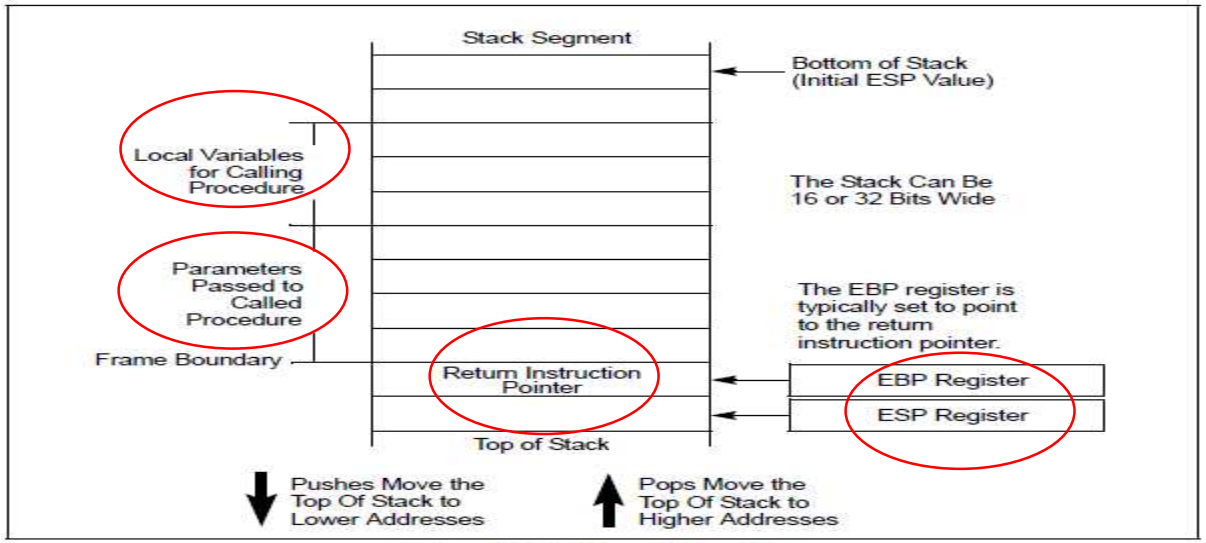


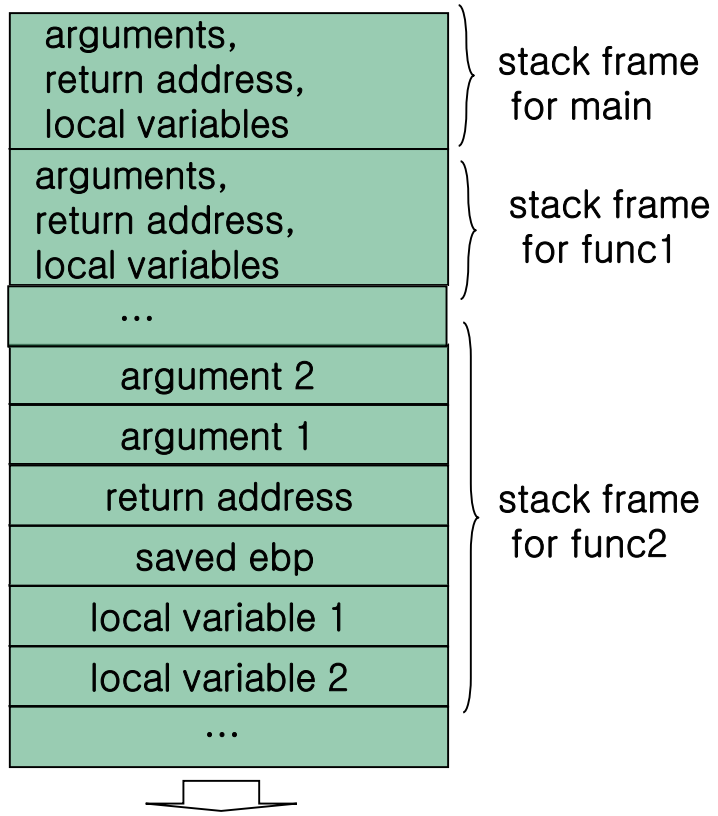
Figure 6-1. Stack Structure

(Source: Intel 64 and IA-32 Architectures Software Developer’s Manual)

Stack Details (4/6)

- Stack in Linux

```
int func2(int x, int y) {
   int f2_local1 = 21, f2_local2 = 22;
   int *pointer, i;

   ...
}

void func1()
{
   int ret_val;
   int f1_local1 = 11, f1_local2 = 12;

   ...
   ret_val = func2(111, 112);
   f1_local++;
   ...
}

int main()
{
   ...
   func1();
   ...
}
```



- Compiler (and version) dependent (see **Appendix 2**)
- Especially, recent compiler makes use of obfuscation, where the locations of local variables are changed according to program contents.
- But, gcc 3.* version comply with the Intel's suggestion (like this figure)
  For lecturing purpose, gcc 3.* is more effective (Use 3.4 in this lecture note)

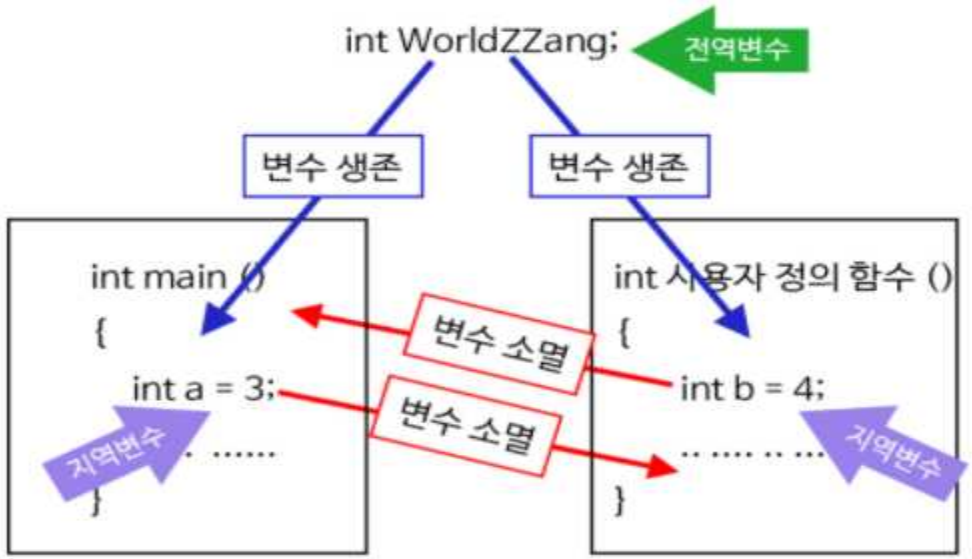
Stack Details (5/6)

- Stack example 1

```
/* stack_struct.c: stack structure analysis, by choijm. choijm@dku.edu  */
#include <stdio.h>

int func2(int x, int y) {
   int f2_local1 = 21, f2_local2 = 22;
   int *pointer;

   printf("func2 local: \t%p, \t%p, \t%p\n", &f2_local1, &f2_local2, &pointer);
   pointer = &f2_local1;

   printf("\t%p  \t%d\n", (pointer), *(pointer));
   printf("\t%p  \t%d\n", (pointer-1), *(pointer-1));
   printf("\t%p  \t%d\n", (pointer+3), *(pointer+3));

   *(pointer+4) = 333;
   printf("\ty = %d\n", y);
   return 222;
}

void func1() {
   int ret_val, f1_local1 = 11, f1_local2 = 12;

   ret_val = func2(111, 112);
}

main() {
   func1();
}
```

Quiz for 6th-Week 1st-Lesson

- Quiz
  - ✓ 1. Explain the differences among 1) file, 2) program (binary program), and 3) process.
  - ✓ 2. In C language, the scope of local variables and global variables are different. Discuss the reason of the differences using the process structure.
  - ✓ Due: until 6 PM Friday of this week (9th, October)



(Source: https://dasima.xyz/c-local-global-variables/)

Stack Details (6/6)

- Stack example 2

```
/* stack_destroy.c: 스택 구조 분석 2, 9월 19일, choijm@dku.edu  */
#include <stdio.h>

void f1() {
   int i;
   printf("In func1\n");
}

void f2() {
   int j, *ptr;
   printf("f2 local: \t%p, \t%p\n", &j, &ptr);
   printf("In func2 \n");

   ptr = &j;
   *(ptr+2) = f1;
}

void f3() {
   printf("Before invoke f2()\n");
   f2();
   printf("After invoke f2()\n");
}

main() {
   f3();
}
```

```
choijm@sungmin-Samsung-DeskTop-System: ~/syspro/chap4
void f2() {
    int j, *ptr;
    printf("f2 local: \t%p, \t%p\n", &j, &ptr);
    printf("In func2 \n");

    ptr = &j;
    *(ptr+2) = f1;
}

void f3() {
    printf("Before invoke f2()\n");
    f2();
    printf("After invoke f2()\n");
}

main() {
    f3();
}

choijm@sungmin-Samsung-DeskTop-System:~/syspro/chap4$
choijm@sungmin-Samsung-DeskTop-System:~/syspro/chap4$ gcc -o stack_destroy stack
_destroy.c
stack_destroy.c: In function `f2':
stack_destroy.c:15: warning: assignment makes integer from pointer without a cas
t
choijm@sungmin-Samsung-DeskTop-System:~/syspro/chap4$
choijm@sungmin-Samsung-DeskTop-System:~/syspro/chap4$ ./stack_destroy
Before invoke f2()
f2 local:       0xbfb829f4,     0xbfb829f0
In func2
In func1
세그멘테이션 오류 (core dumped)
choijm@sungmin-Samsung-DeskTop-System:~/syspro/chap4$
choijm@sungmin-Samsung-DeskTop-System:~/syspro/chap4$
```

Summary

- Understand the differences between process and program
- Discuss the differences among text, data, heap and stack
- Find out the details of stack structure
  - ✓ Argument passing, Return address, Local variables
  - ✓ Stack overflow

☞ **Exercise 1 (old homework 4): Make a program of the stack example 2 and examine its results.**

✓ **Requirements**

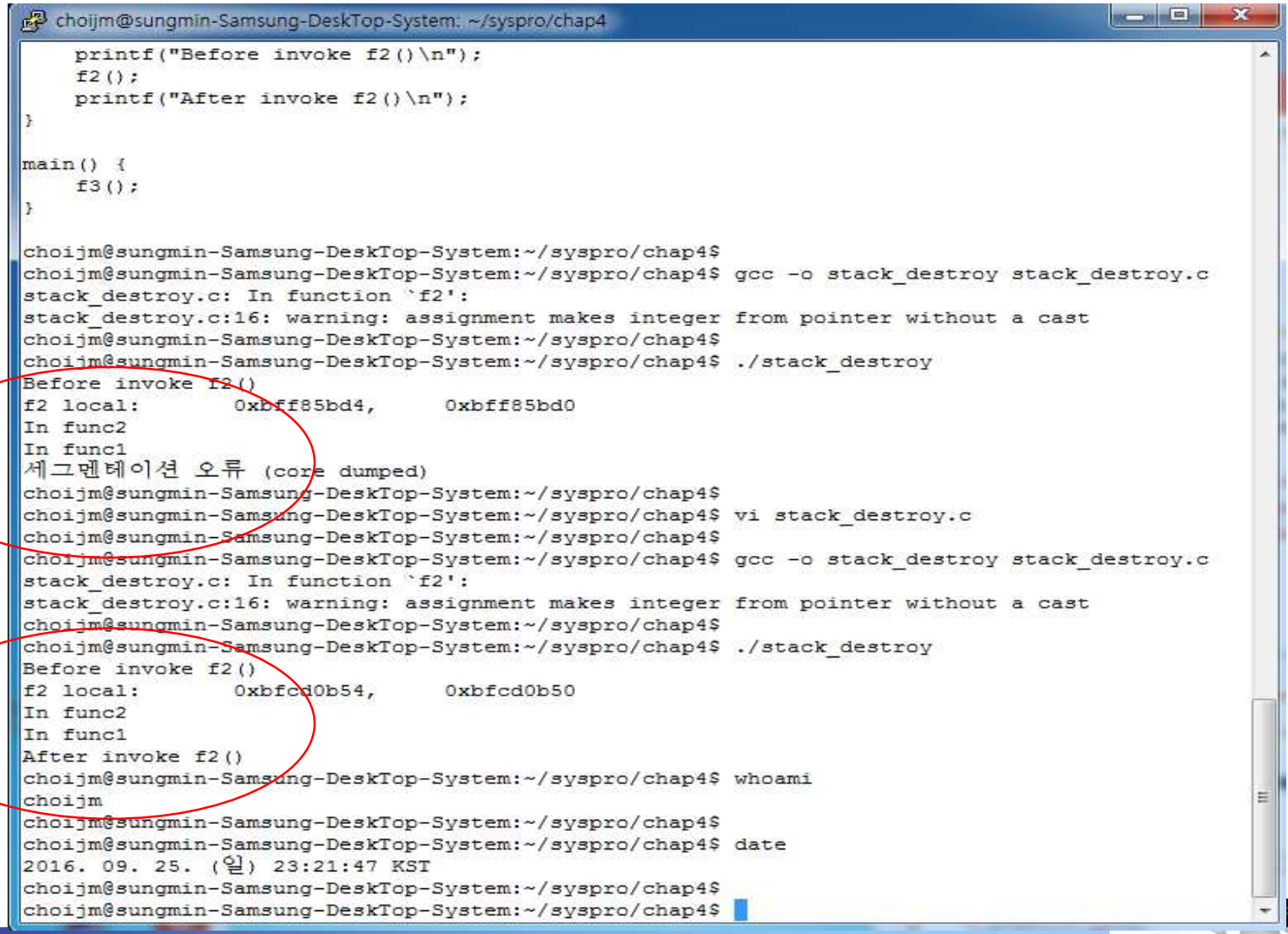
- shows student's ID and date (using whoami and date)

- overcome the segmentation fault problem

- hand out the report that includes a snapshot and discussion

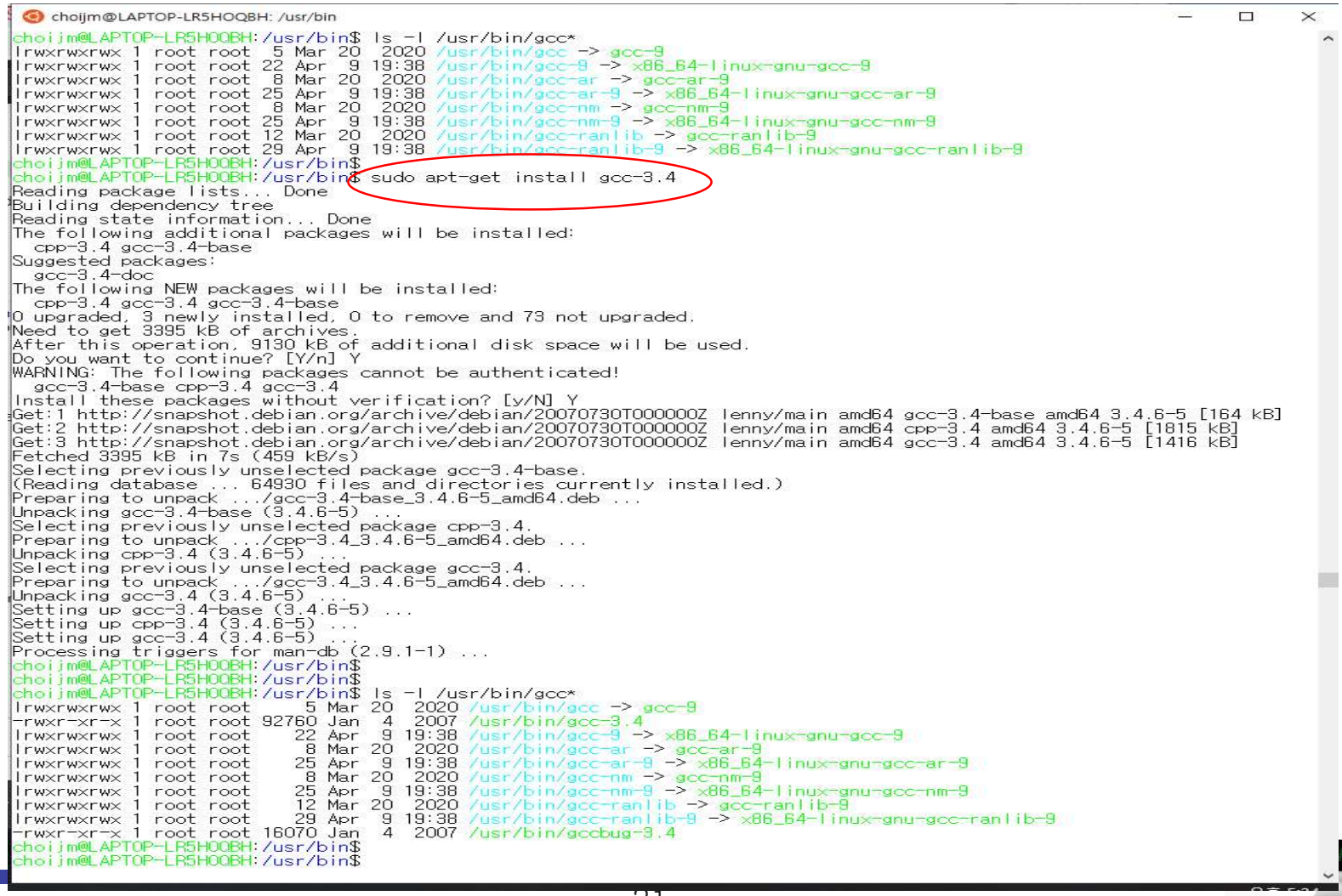
Appendix 1

- Snapshot for the Exercise 1



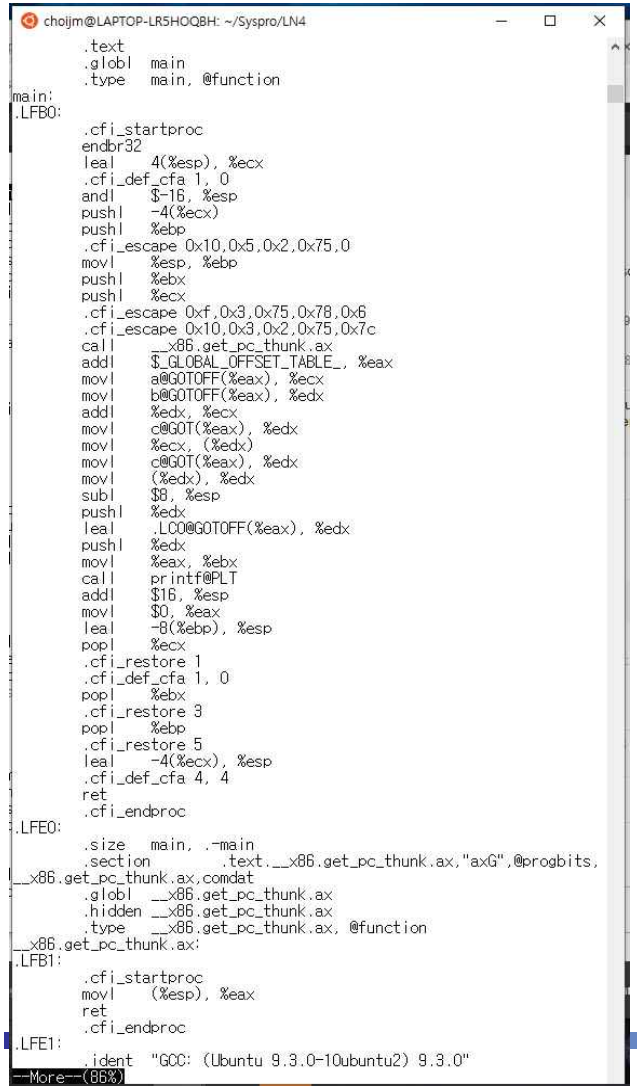
Appendix 2

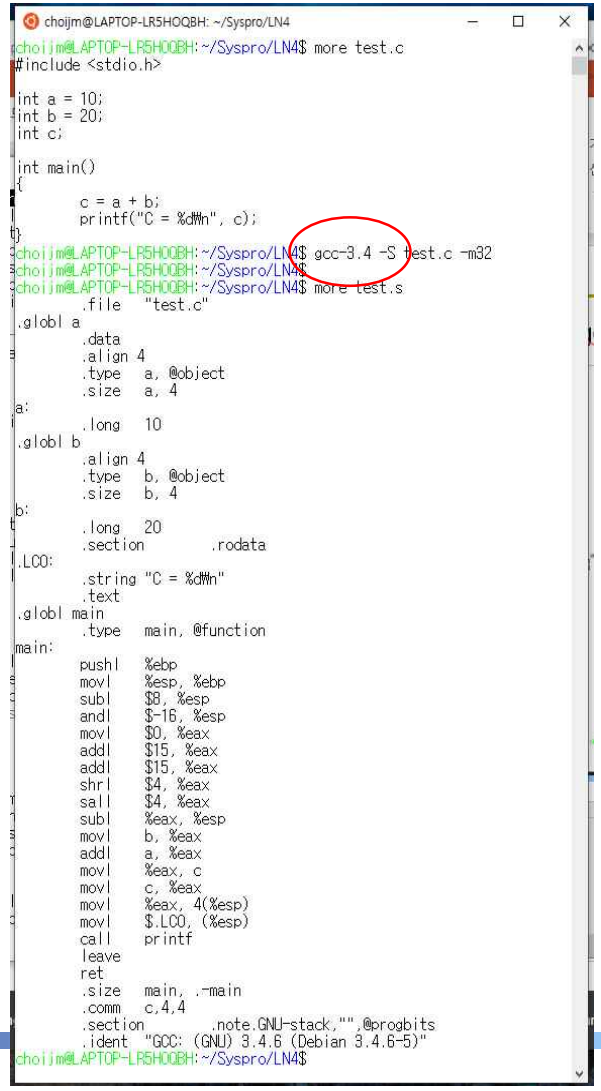
- Assembly differences between gcc 9.* and gcc 3.4.*
  - ✓ Using WSL (Windows subsystem for Linux) in my computer

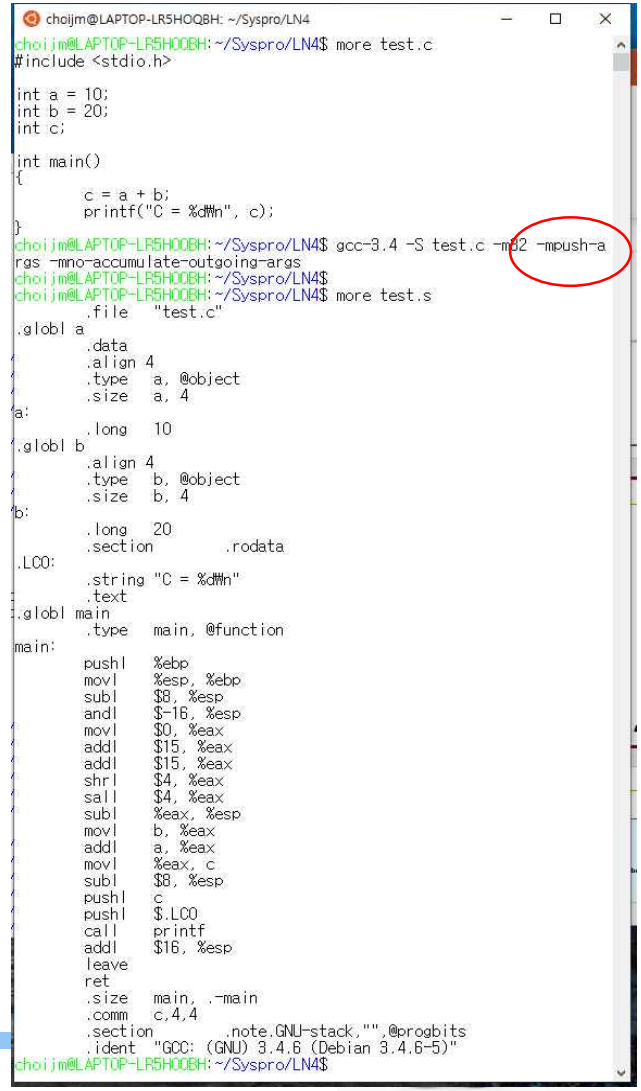


Appendix 2

- Assembly differences between gcc 9.* and gcc 3.4.*
  - ✓ 1) Obfuscation, 2) Optimization, 3) CFI, …

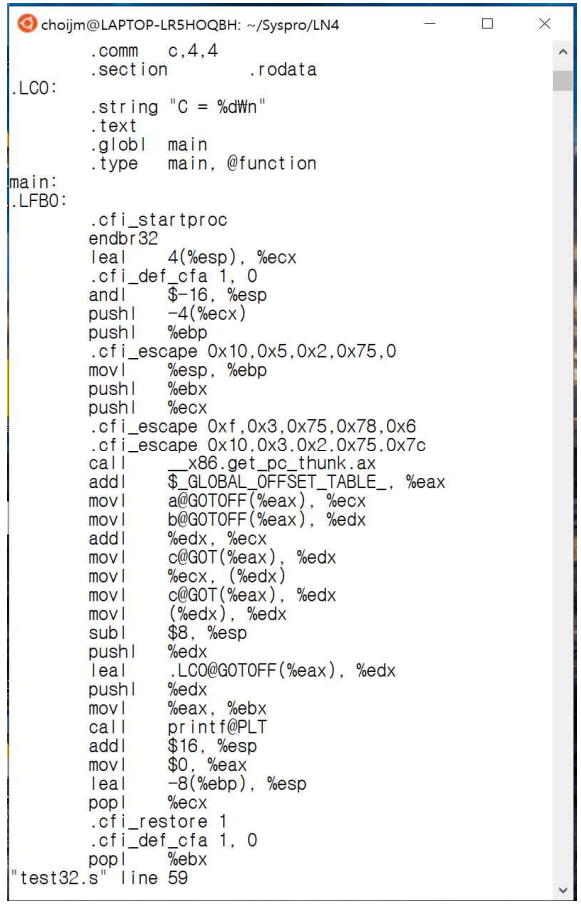


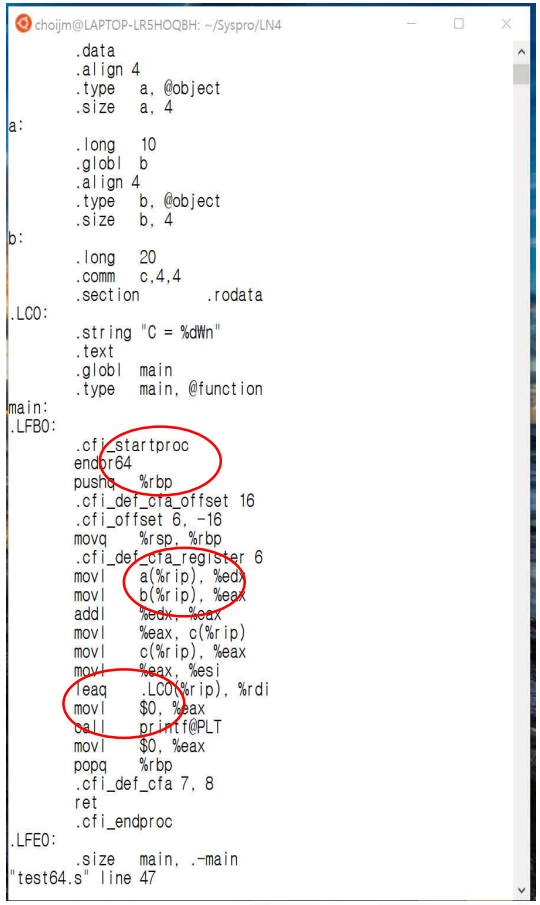




Appendix 2

- Assembly differences between 32-bit and 64-bit CPU
  - ✓ 1) Register (eax vs rax), 2) PIC, 3) Argument passing, 4) …
  - ✓ We will discuss further in LN6 and LN9





Appendix 3

- Function pointer practice

```
/*****************************/
/* f_ptr_exam.c : 함수 포인터 예, */
/* 10월 4일, choijm@dku.edu     */
/*****************************/
#include <stdio.h>

int add(int a, int b)
{
      return a+b;
}

in
{

}

in
{

}

in
{

}
```

```
main()
{
      int (*ALU)(int x, int y);
      int a, b;
      char c;

      printf("USAGE: number operator number ==>");
      scanf("%d %c %d", &a, &c, &b);

      switch (c)
      {
case '+':
            ALU = add;
            break;
case '-':
            ALU = substract;
            break;
case '*':
            ALU = multiply;
            break;
case '/':
            ALU = divide;
            break;


f("₩n%d %c %d = %d₩n", a, c, b, ALU(a,b));
}
```

```
choijm@localhost:~/syspro_examples/chap4
                        ALU = divide;
                        break;
        }

        printf("\n%d %c %d = %d\n", a, c, b, ALU(a,b));
}
[choijm@localhost chap4]$
[choijm@localhost chap4]$ gcc -o f_ptr_exam f_ptr_exam.c
[choijm@localhost chap4]$
[choijm@localhost chap4]$ ./f_ptr_exam
USAGE: number operator number ==>4 + 5

4 + 5 = 9
[choijm@localhost chap4]$
[choijm@localhost chap4]$ ./f_ptr_exam
USAGE: number operator number ==>10 - 23

10 - 23 = -13
[choijm@localhost chap4]$
[choijm@localhost chap4]$ ./f_ptr_exam
USAGE: number operator number ==>8 / 2

8 / 2 = 4
[choijm@localhost chap4]$
```