Lecture Note 5.
Process Programming

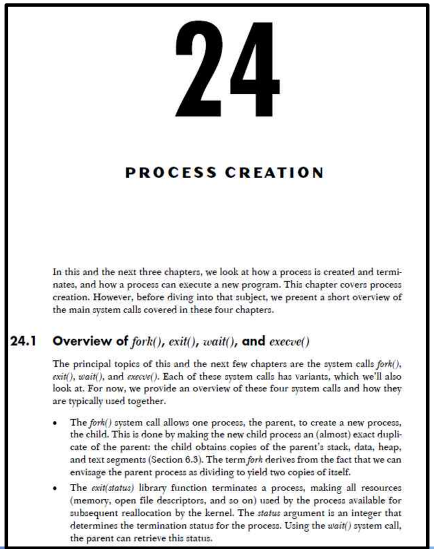
October 23, 2023

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

Objectives

- Understand process-related system calls
- Learn how to create a new process
- Learn how to execute a new program
- Discuss about shell (command interpreter)
- Understand issues on multitask
  - ✓ Synchronization, virtual address, thread, …

- Refer to Chapter 24, 27, 29 in the LPI and Chapter 8 in the CSAPP



24

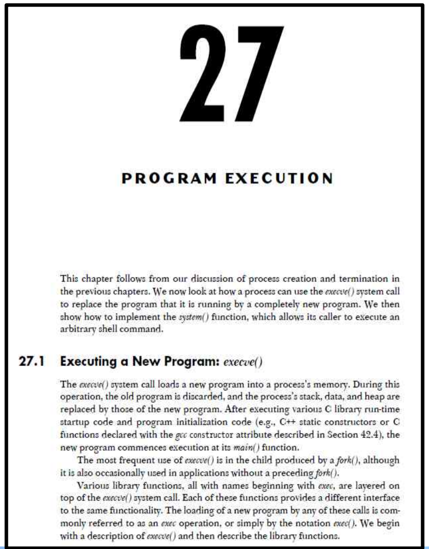
PROCESS CREATION

In this and the next three chapters, we look at how a process is created and terminates, and how a process can execute a new program. This chapter covers process creation. However, before diving into that subject, we present a short overview of the main system calls covered in these four chapters.

24.1 Overview of *fork()*, *exit()*, *wait()*, and *execve()*

The principal topics of this and the next few chapters are the system calls *fork()*, *exit()*, *wait()*, and *execve()*. Each of these system calls has variants, which we'll also look at. For now, we provide an overview of these four system calls and how they are typically used together.

- The *fork()* system call allows one process, the parent, to create a new process, the child. This is done by making the new child process an (almost) exact duplicate of the parent: the child obtains copies of the parent's stack, data, heap, and text segments (Section 6.3). The term *fork* derives from the fact that we can envisage the parent process as dividing to yield two copies of itself.
- The *exit(status)* library function terminates a process, making all resources (memory, open file descriptors, and so on) used by the process available for subsequent reallocation by the kernel. The *status* argument is an integer that determines the termination status for the process. Using the *wait()* system call, the parent can retrieve this status.



27

PROGRAM EXECUTION

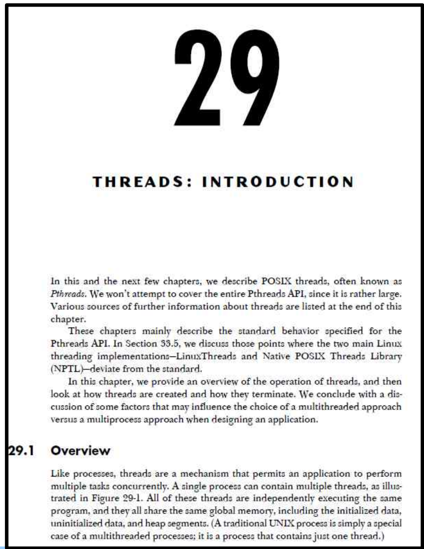
This chapter follows from our discussion of process creation and termination in the previous chapters. We now look at how a process can use the *execve()* system call to replace the program that it is running by a completely new program. We then show how to implement the *system()* function, which allows its caller to execute an arbitrary shell command.

27.1 Executing a New Program: *execve()*

The *execve()* system call loads a new program into a process's memory. During this operation, the old program is discarded, and the process's stack, data, and heap are replaced by those of the new program. After executing various C library run-time startup code and program initialization code (e.g., C++ static constructors or C functions declared with the *gcc* constructor attribute described in Section 42.4), the new program commences execution at its *main()* function.

The most frequent use of *execve()* is in the child produced by a *fork()*, although it is also occasionally used in applications without a preceding *fork()*.

Various library functions, all with names beginning with *exec*, are layered on top of the *execve()* system call. Each of these functions provides a different interface to the same functionality. The loading of a new program by any of these calls is commonly referred to as an *exec* operation, or simply by the notation *exec()*. We begin with a description of *execve()* and then describe the library functions.



29

THREADS: INTRODUCTION

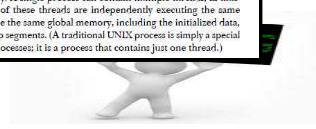
In this and the next few chapters, we describe POSIX threads, often known as *Pthreads*. We won't attempt to cover the entire Pthreads API, since it is rather large. Various sources of further information about threads are listed at the end of this chapter.

These chapters mainly describe the standard behavior specified for the Pthreads API. In Section 33.5, we discuss those points where the two main Linux threading implementations—LinuxThreads and Native POSIX Threads Library (NPTL)—deviate from the standard.

In this chapter, we provide an overview of the operation of threads, and then look at how threads are created and how they terminate. We conclude with a discussion of some factors that may influence the choice of a multithreaded approach versus a multiprocess approach when designing an application.

29.1 Overview

Like processes, threads are a mechanism that permits an application to perform multiple tasks concurrently. A single process can contain multiple threads, as illustrated in Figure 29-1. All of these threads are independently executing the same program, and they all share the same global memory, including the initialized data, uninitialized data, and heap segments. (A traditional UNIX process is simply a special case of a multithreaded processes; it is a process that contains just one thread.)



Introduction

- Process-related system calls
  - ✓ Basic
    - fork(), clone() : create a process, make a task_struct (like inode)
    - execve() : execute a new program (loading)
    - exit() : terminate a process, inform child status to parent
    - wait(), waitpid() : wait for a process's termination (child or designated)
    - getpid() : get a process ID

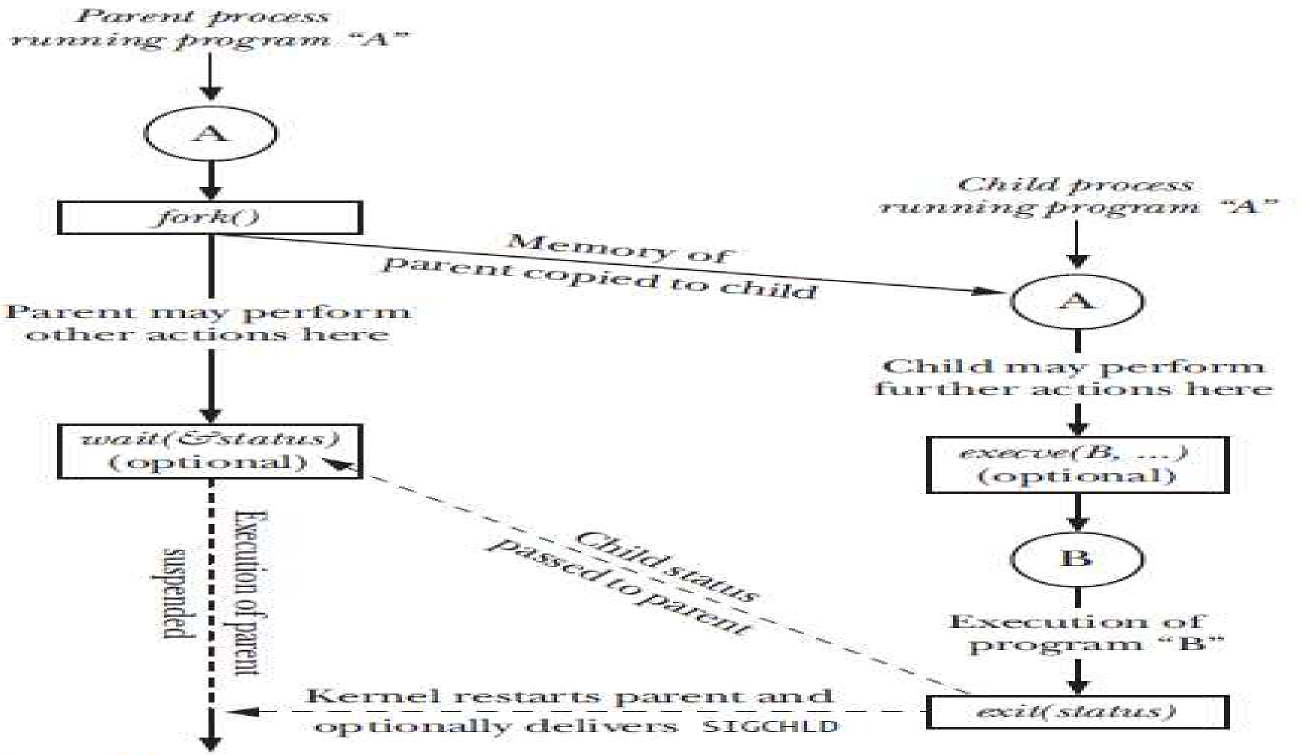
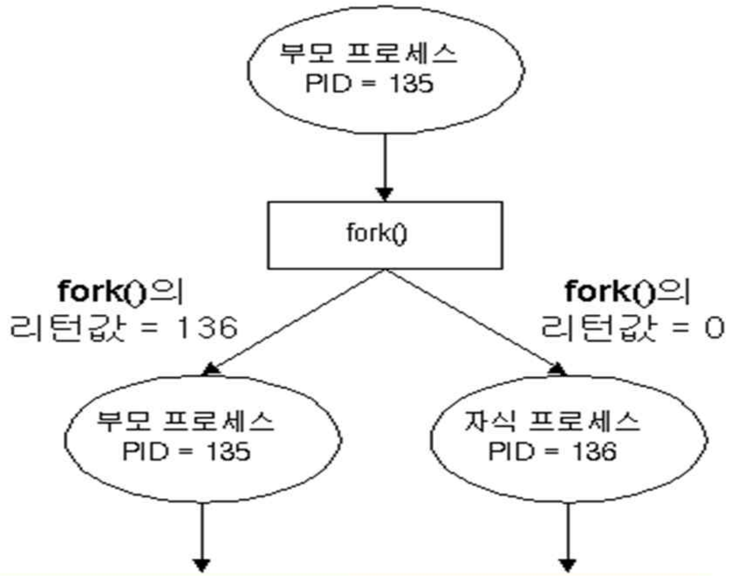


Figure 24-1: Overview of the use of *fork()*, *exit()*, *wait()*, and *execve()*

(Source: LPI)

Process creation (1/6)

- fork() system call
  - ✓ Make a new process whose memory image (text, data, …) is the same as the existing process
    - Existing process: parent process
    - New process: child process
  - ✓ Split the flow control into two (system's viewpoint)
    - One for parent and the other for child process
  - ✓ Two return values (program's viewpoint)
    - Parent process: child's pid (always larger than 0)
    - Child process: 0



Process creation (2/6)

- Practice 1: making two control flows

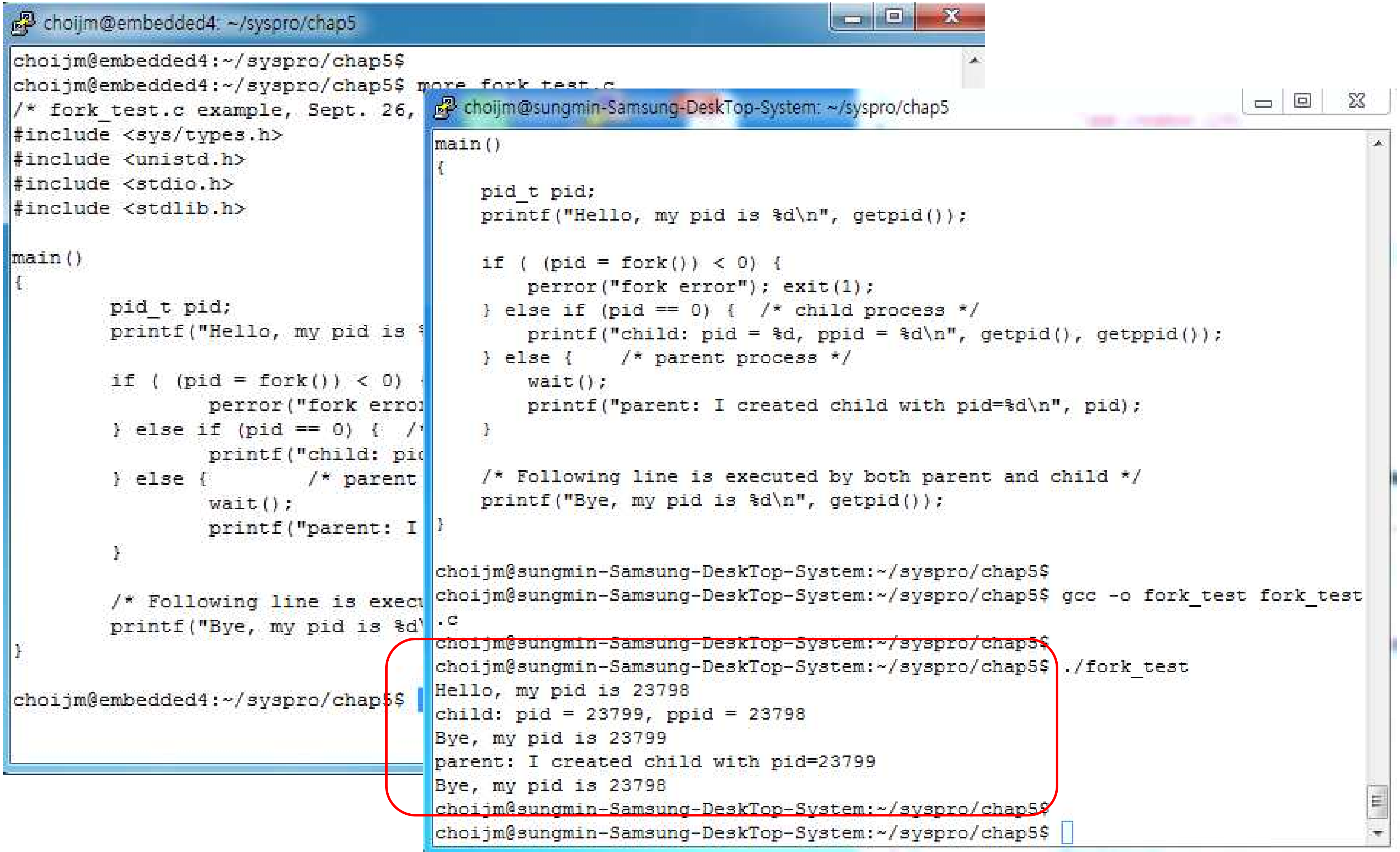
```
/* fork_test.c example, Sept. 26, choijm@dku.edu */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
        pid_t fork_return;
        printf("Hello, my pid is %d\n", getpid());

        if ( (fork_return = fork()) < 0) {
                perror("fork error"); exit(1);
        } else if (fork_return == 0) {          /* child process */
                printf("child: pid = %d, ppid = %d\n", getpid(), getppid());
        } else {        /* parent process */
                wait();
                printf("parent: I created child with pid=%d\n", fork_return);
        }

        /* Following line is executed by both parent and child */
        printf("Bye, my pid is %d\n", getpid());
}
```

The flow of control is divided here.

This message is printed out twice.

Process creation (3/6)

- Practice 1: execution results



Process creation (4/6)

- Practice 2: variable (local and global) management

```
/* fork_test2.c: accessing variables, Sept. 26, choijm@dku.edu */
/* Note: This code is borrowed from "Advanced Programming in the UNIX Env." */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int         glob = 6;  char        buf[] = "a write to stdout\n";

int main(void)
{
        int         var = 88;  pid_t       fork_return;

        if (write(STDOUT_FILENO, buf, sizeof(buf)) != sizeof(buf)) {
                perror("write error"); exit(1);
        }
        printf("before fork\n");                    /* we don't flush stdout */

        if ( (fork_return = fork()) < 0) {
                perror("fork error"); exit(1);
        } else if (fork_return == 0) {              /* child */
                glob++; var++;                      /* modify variables */
        } else
                sleep(2);                           /* parent */
        printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
        exit(0);
}
```

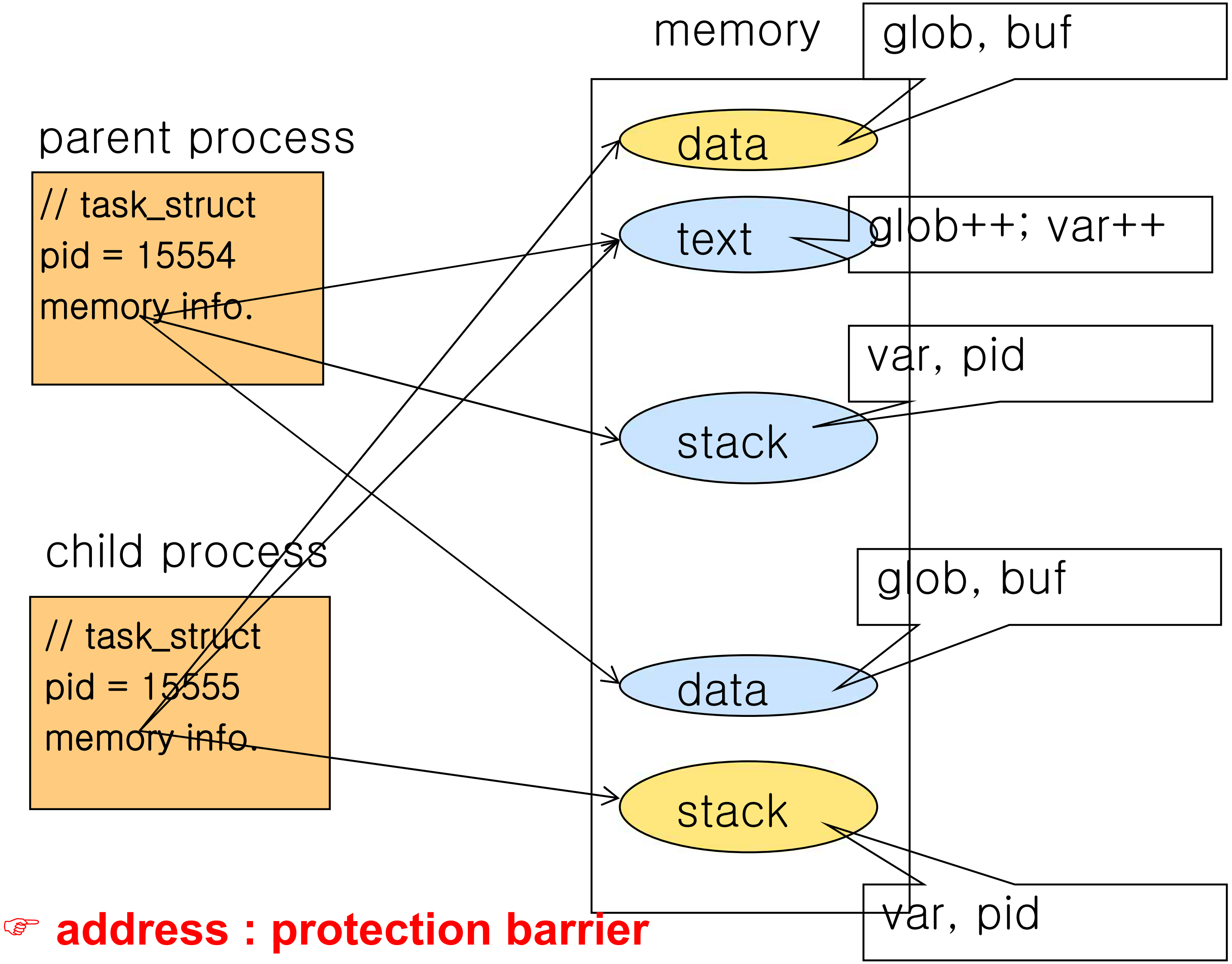
Process creation (5/6)

- Practice 2: execution results

```
choijm@embedded4: ~/syspro/chap5
choijm@embedded4:~/syspro/chap5$
choijm@embedded4:~/syspro/chap5$ vi fork_test2.c
choijm@embedded4:~/syspro/chap5$
choijm@embedded4:~/syspro/chap5$ gcc -o fork_test2 fork_test2.c
choijm@embedded4:~/syspro/chap5$
choijm@embedded4:~/syspro/chap5$ ./fork_test2
a write to stdout
before fork
pid = 15555, glob = 7, var = 89
pid = 15554, glob = 6, var = 88
choijm@embedded4:~/syspro/chap5$
choijm@embedded4:~/syspro/chap5$
choijm@embedded4:~/syspro/chap5$ ./fork_test2 &
[1] 15557
choijm@embedded4:~/syspro/chap5$ a write to stdout
before fork
pid = 15558, glob = 7, var = 89

choijm@embedded4:~/syspro/chap5$ ps
  PID TTY          TIME CMD
15085 pts/1    00:00:00 bash
15557 pts/1    00:00:00 fork_test2
15558 pts/1    00:00:00 fork_test2 <defunct>
15559 pts/1    00:00:00 ps
choijm@embedded4:~/syspro/chap5$ pid = 15557, glob = 6, var = 88

[1]+  완료                    ./fork_test2
choijm@embedded4:~/syspro/chap5$
choijm@embedded4:~/syspro/chap5$ ps
  PID TTY          TIME CMD
15085 pts/1    00:00:00 bash
15560 pts/1    00:00:00 ps
choijm@embedded4:~/syspro/chap5$
```

Process creation (6/6)

- System's viewpoint of fork()



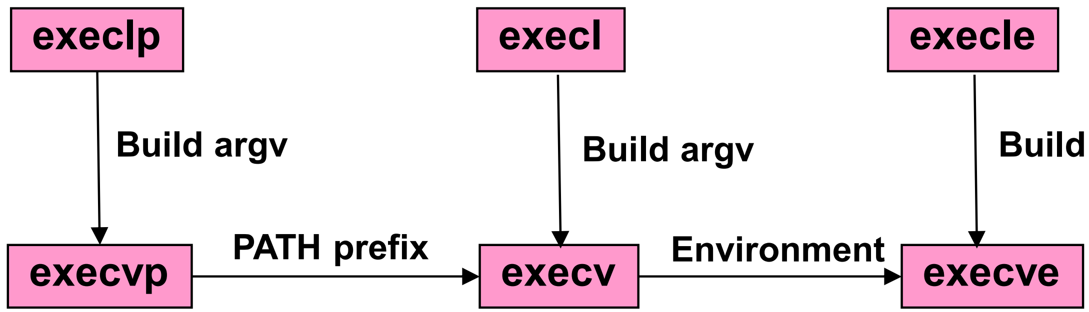
☞ **address : protection barrier**

☞ **We can exploit "COW(Copy_on_Write)" for enhancing performance**

☞ **We do not consider "Paging" in this slide.**

Process execution (1/7)

- execve() system call
  - ✓ Execute a new program
    - Replace the current process's memory image (text, data, stack) with new binary
  - ✓ Six interfaces



<u>Syntax</u>

int execlp(const char *filename, const char *arg0, …, const char argn, (char *) 0);
int execvp(const char *filename, char *const argv[ ]);
int execl(const char *pathname, const char *arg0, …, const char *argn, (char *) 0);
int execv(const char *pathname, char *const argv[ ]);
int execle(const char *pathname, const char *arg0, …, const char *argn, (char *) 0, char *const envp[ ]);
int execve(const char *pathname, char *const argv[ ], char *const envp[ ]);

Process execution (2/7)

- Practice 3: executing a new program (binary)

```c
/* execl_test.c:  execute a hello program, Sept. 27, choijm@dku.edu */
#include      <unistd.h>
#include      <stdio.h>
#include      <stdlib.h>

int main(int argc, char *argv[])
{
        pid_t fork_return, d_pid; int exit_status = -1;

        if ((fork_return = fork()) == -1) {
                // fork error handling
        } else if (fork_return == 0) {          // child
                execl("./hello", "./hello", (char *)0);
                printf("Child.. I'm here\n");
                // if execl() succeeds, the above printf() is not executed!!
                exit(1);
        } else {                        // parent
                d_pid = wait(&exit_status);
                printf("Parent.. I'm here\n");
                printf("exit status of process %d is %d\n", d_pid, exit_status);
        }
}
```

What does this comment mean?

Process execution (3/7)

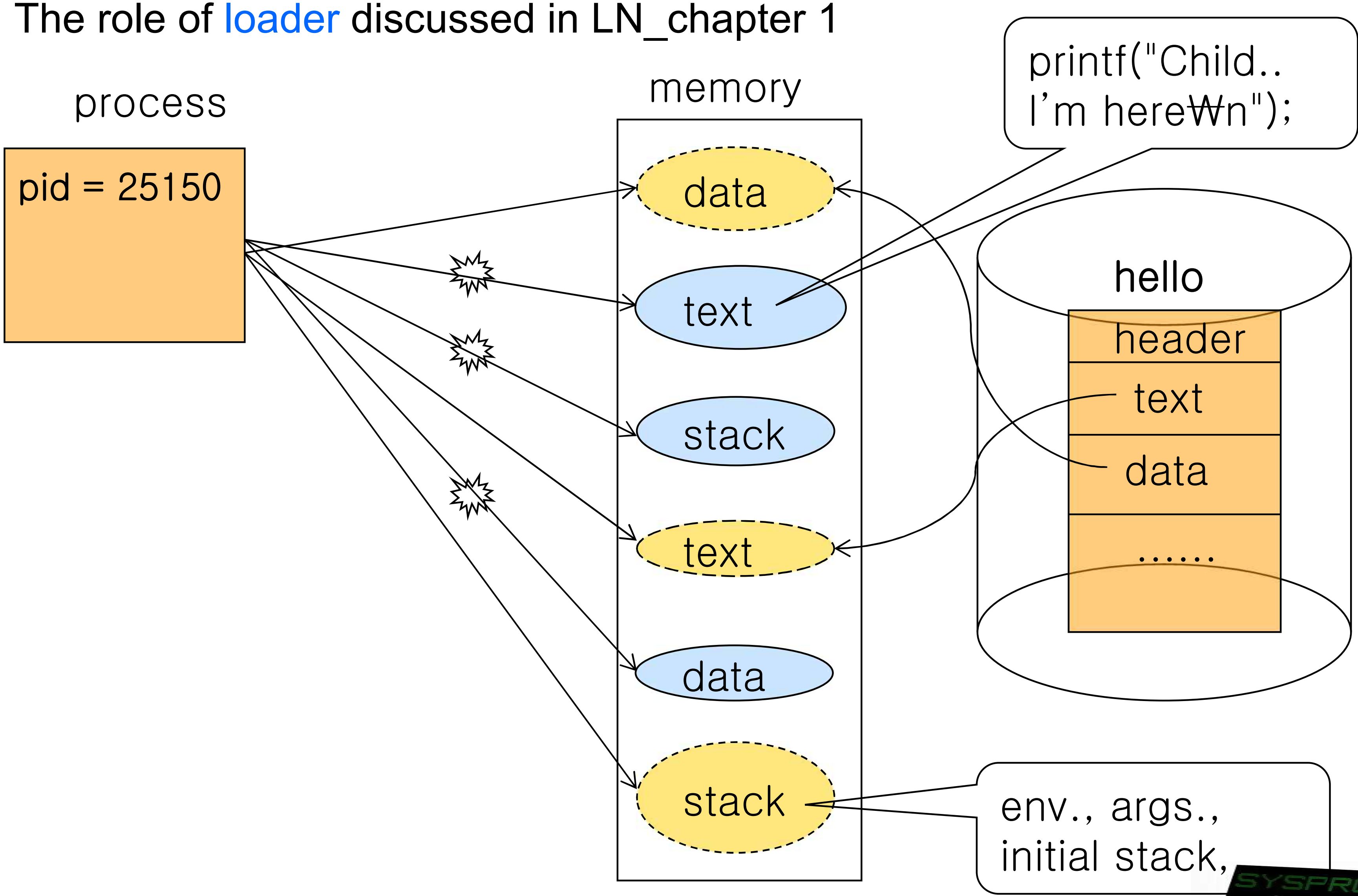
- Practice 3: execution results

```
[choijm@localhost chap5]$
[choijm@localhost chap5]$ more hello.c
#include <stdio.h>
#include <stdlib.h>

main()
{
        printf("Hello World\n");
        exit(0);
}
[choijm@localhost chap5]$
[choijm@localhost chap5]$
[choijm@localhost chap5]$ more execl_test.c
/* execl_test.c:  hello 수행 . 9월 27일. choijm@dku.edu */
#include        <unistd.h>
#include        <stdio.h>
#include        <stdlib.h>

int main(int argc, char *argv[])
{
        pid_t pid, d_pid; int exit_status = -1;

        if ((pid = fork()) == -1) {
                // fork error 처리
        } else if (pid == 0) {          // child
                execl("./hello", "./hello", (char *)0);
                printf("Child.. I'm here\n");
                // execl 성공일 경우 여기는 수행될 수 없음
                exit(1);
        } else {                        // parent
                d_pid = wait(&exit_status);
                printf("Parent.. I'm here\n");
                printf("exit status of task %d is %d\n", d_pid, exit_status);
        }
}
[choijm@localhost chap5]$
```

```
[choijm@localhost chap5]$
[choijm@localhost chap5]$ gcc -o hello hello.c
[choijm@localhost chap5]$
[choijm@localhost chap5]$ ./hello
Hello World
[choijm@localhost chap5]$
[choijm@localhost chap5]$
[choijm@localhost chap5]$ gcc -o execl_test execl_test.c
[choijm@localhost chap5]$
[choijm@localhost chap5]$
[choijm@localhost chap5]$ ./execl_test
Hello World
Parent.. I'm here
exit status of task 25150 is 0
[choijm@localhost chap5]$
[choijm@localhost chap5]$ ps
  PID TTY          TIME CMD
24693 pts/0    00:00:00 bash
25152 pts/0    00:00:00 ps
[choijm@localhost chap5]$
[choijm@localhost chap5]$
```

Process execution (4/7)

- System's viewpoint of execve()
  - ✓ Replace memory image (text, data, stack) with new one
  - ✓ The role of loader discussed in LN_chapter 1



Process execution (5/7)

- Practice 4: parameter passing to main() via shell

```
/* execl_test2.c:  printing argv[] and env[], Sept. 27, choijm@dku.edu */
#include <stdio.h>
int main(int argc, char *argv[], char *envp[])
{
        int i;
        for (i=0; argv[i]; i++)
                printf("arg %d = %s\n", i, argv[i]);
        for (i=0; envp[i]; i++)
                printf("env %d = %s\n", i, envp[i]);
}
```

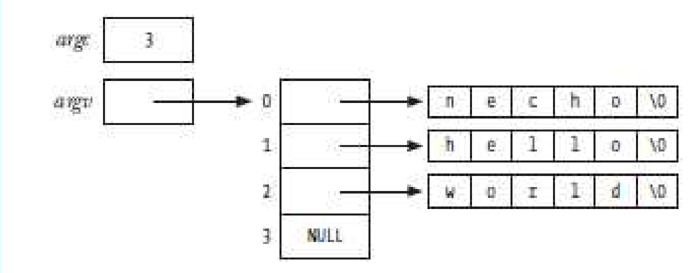
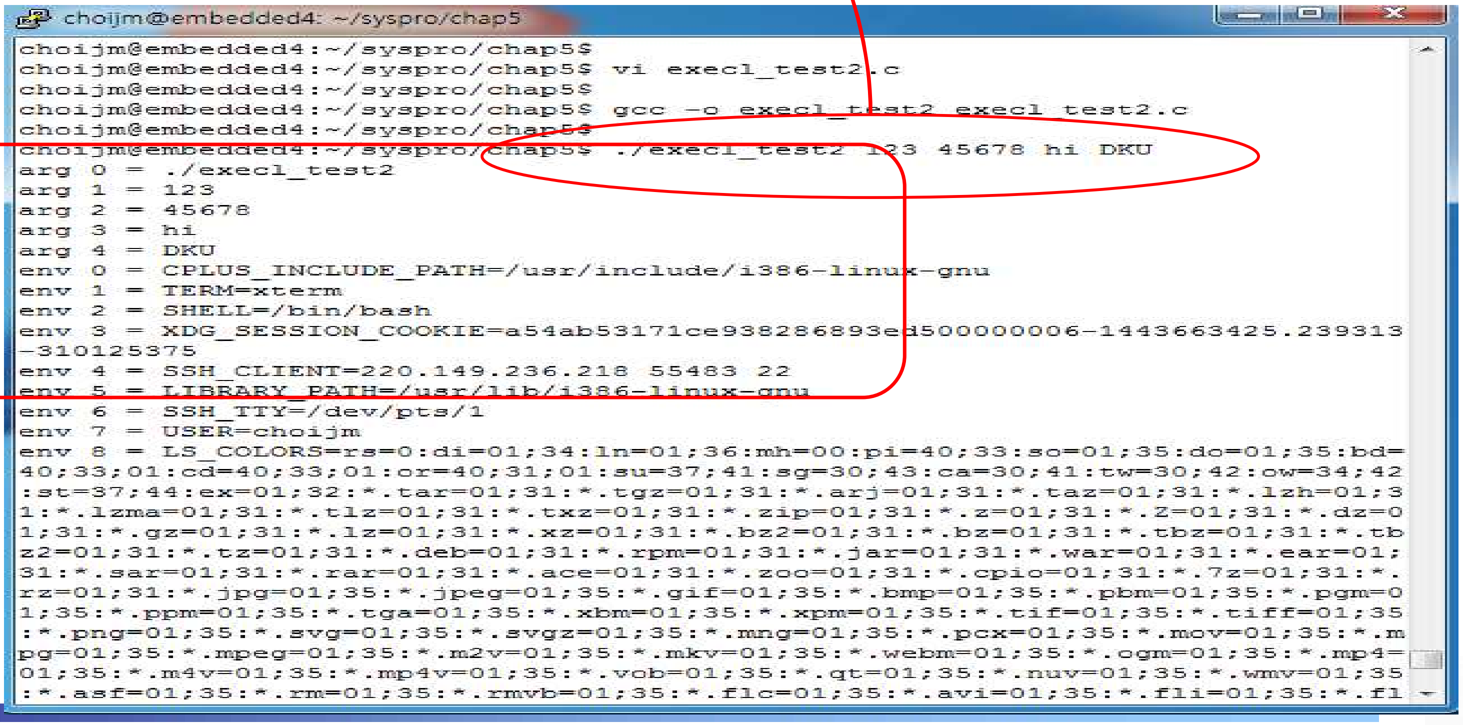


Figure 6-4: Values of *argc* and *argv* for the command *necho hello world*



```
choijm@embedded4: ~/syspro/chap5
choijm@embedded4:~/syspro/chap5$
choijm@embedded4:~/syspro/chap5$ vi execl_test2.c
choijm@embedded4:~/syspro/chap5$
choijm@embedded4:~/syspro/chap5$ gcc -o execl_test2 execl_test2.c
choijm@embedded4:~/syspro/chap5$
choijm@embedded4:~/syspro/chap5$ ./execl_test2 123 45678 hi DKU
arg 0 = ./execl_test2
arg 1 = 123
arg 2 = 45678
arg 3 = hi
arg 4 = DKU
env 0 = CPLUS_INCLUDE_PATH=/usr/include/i386-linux-gnu
env 1 = TERM=xterm
env 2 = SHELL=/bin/bash
env 3 = XDG_SESSION_COOKIE=a54ab53171ce938286893ed500000006-1443663425.239313
-310125375
env 4 = SSH_CLIENT=220.149.236.218 55483 22
env 5 = LIBRARY_PATH=/usr/lib/i386-linux-gnu
env 6 = SSH_TTY=/dev/pts/1
env 7 = USER=choijm
env 8 = LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=
40;33;01:cd=40;33;01:or=40;31;01:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42
:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arj=01;31:*.taz=01;31:*.lzh=01;3
1:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.dz=0
1;31:*.gz=01;31:*.lz=01;31:*.xz=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tb
z2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;
31:*.sar=01;31:*.rar=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.
rz=01;31:*.jpg=01;35:*.jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=0
1;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35
:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.m
pg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=
01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35
:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.fl
```

Process execution (6/7)

- Practice 5: parameter passing to main() via execle()

```
/* execle_test3.c:  parameter passing, Sept. 27, choijm@dku.edu */
#include    <sys/types.h>
#include    <unistd.h>
#include    <stdio.h>
#include    <wait.h>

int main(int argc, char *argv[])
{
   pid_t fork_return, d_pid; int exit_status;
   char *const myenv[] = {"sys programming", "is", "fun", (char *)0};

   if ((fork_return = fork()) == -1) {
      // fork error handling
   } else if (fork_return == 0) {      // child
      execle("./execl_test2", "./execl_test2", “Hi", “DKU", (char *)0, myenv);
      printf("Child.. I’m here\n");
      // if execl succeeds, this printf() is not carried out!!
   } else {           // parent
      d_pid = wait(&exit_status);
      printf(“exit pid = %d with status = %d\n", d_pid, WEXITSTATUS(exit_status));
   }
}
```

Process execution (7/7)

- Practice 5: execution results

```
choijm@embedded:~/syspro18/chap5$ gcc -o execle_test3 execle_test3.c
choijm@embedded:~/syspro18/chap5$
choijm@embedded:~/syspro18/chap5$ ./execle_test3
arg 0 = ./execl_test2
arg 1 = Hi
arg 2 = DKU
env 0 = sys programming
env 1 = is
env 2 = fun
exit pid = 31727 with status = 0
choijm@embedded:~/syspro18/chap5$
choijm@embedded:~/syspro18/chap5$ more execle_test3.c
/* execle_test3.c:  parameter passing, Sept. 27, choijm@dku.edu */
#include    <sys/types.h>
#include    <unistd.h>
#include    <stdio.h>
#include    <wait.h>

int main(int argc, char *argv[])
{
    pid_t pid, d_pid; int exit_status;
    char *const myenv[] = {"sys programming", "is", "fun", (char *)0};

    if ((pid = fork()) == -1) {
        // fork error handling
    } else if (pid == 0) {       // child
        execle("./execl_test2", "./execl_test2", "Hi", "DKU", (char *)0, myenv);
        printf("Child.. I'm here\n");
        // if execl succeeds, this printf() is not carried out!!
    } else {             // parent
        d_pid = wait(&exit_status);
        printf("exit pid = %d with status = %d\n", d_pid, WEXITSTATUS(exit_status));
    }
```

Binary format (1/2)

- ELF (Executable Linking Format)

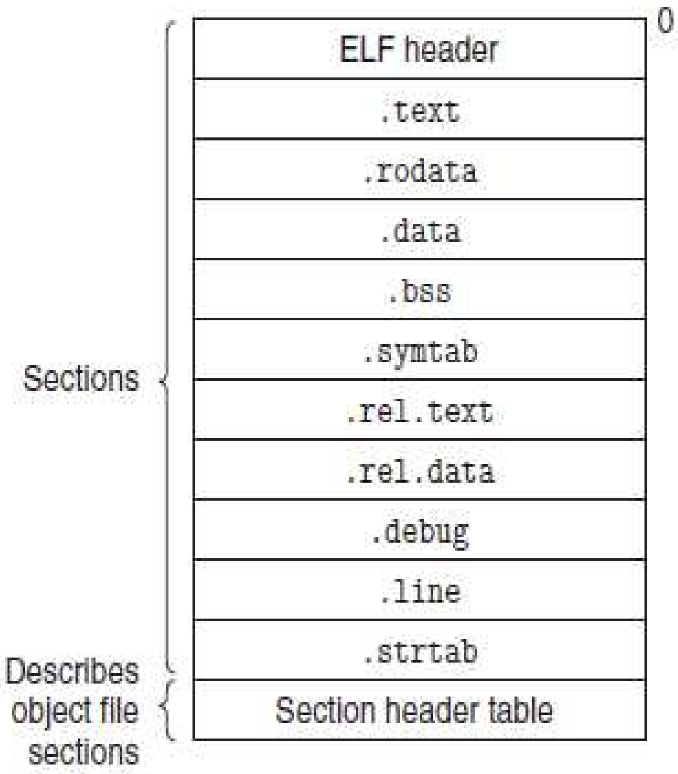


Fig. 7.3 Typical ELF relocatable object file

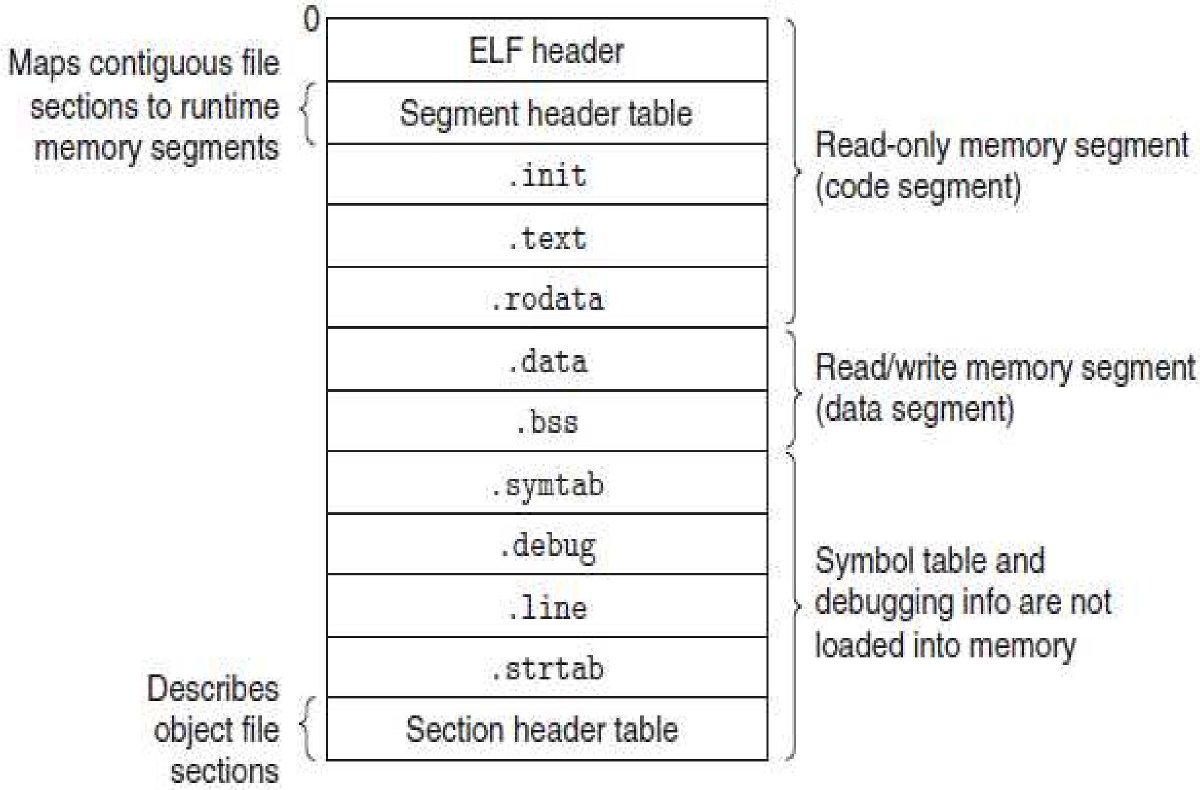


Fig. 7.11 Typical ELF executable object file

(Source: CSAPP)

☞ **Why we separate data into two sub-regions (initialized data and bss)?**

Binary format (2/2)

- Real view in Linux

choijm@embedded: ~/syspro18/chap1

```
choijm@embedded:~/syspro18/chap1$ vi test.c
choijm@embedded:~/syspro18/chap1$ more test.c
#include <stdio.h>
int a = 10;
int b = 20;
int c;

main()
{
        c = a + b;
        printf("c = %d\n", c);
}
choijm@embedded:~/syspro18/chap1$ gcc -S test.c
choijm@embedded:~/syspro18/chap1$ gcc -c test.c
choijm@embedded:~/syspro18/chap1$ objdump -h test.o

test.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000043  00000000  00000000  00000034  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000008  00000000  00000000  00000078  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000080  2**0
                  ALLOC
  3 .rodata       00000008  00000000  00000000  00000080  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .note.GNU-stack 00000000  00000000  00000000  00000088  2**0
                  CONTENTS, READONLY
  5 .comment      00000023  00000000  00000000  00000088  2**0
                  CONTENTS, READONLY
choijm@embedded:~/syspro18/chap1$ more test.s
        .file   "test.c"
.globl a
        .data
        .align 4
        .type   a, @object
        .size   a, 4
a:
        .long   10
.globl b
        .align 4
        .type   b, @object
        .size   b, 4
```

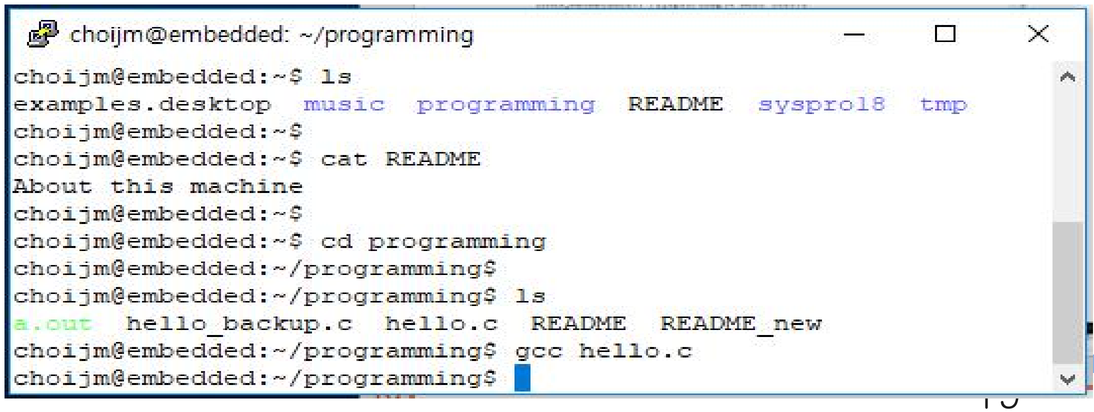
choijm@embedded: ~/syspro18/chap1

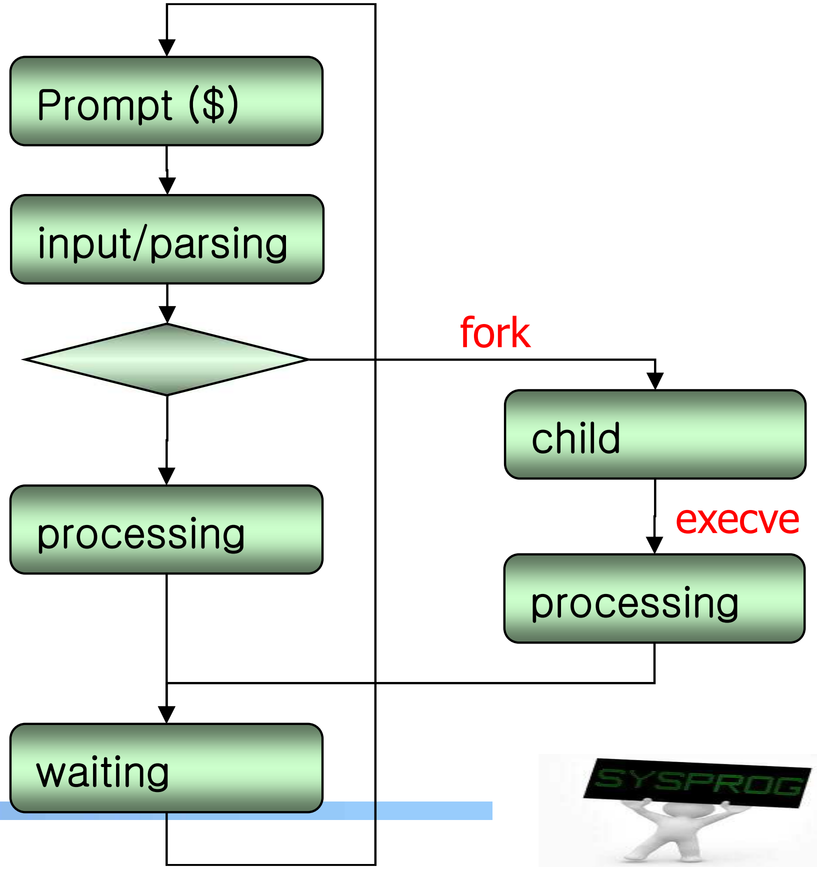
```
choijm@embedded:~/syspro18/chap1$ more test.s
        .file   "test.c"
.globl a
        .data
        .align 4
        .type   a, @object
        .size   a, 4
a:
        .long   10
.globl b
        .align 4
        .type   b, @object
        .size   b, 4
b:
        .long   20
        .section        .rodata
.LC0:
        .string "c = %d\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        addl    $15, %eax
        addl    $15, %eax
        shrl    $4, %eax
        sall    $4, %eax
        subl    %eax, %esp
        movl    b, %eax
        addl    a, %eax
        movl    %eax, c
        movl    c, %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
        leave
        ret
        .size   main, .-main
        .comm   c,4,4
        .section        .note.GNU-stack,"",@progbits
        .ident  "GCC: (GNU) 3.4.6 (Debian 3.4.6-5)"
```

☞ **Refer to other commands such as readelf and size**

Shell (1/5)

- Command interpreter
  - ✓ Execute commands requested by users
- Basic logic
  - ✓ display prompt, input parsing
  - ✓ for external commands: do fork() and execve() at child process
  - ✓ for internal commands: perform in shell without fork() and execve()
- Advanced functions
  - ✓ Background processing
  - ✓ Redirection
  - ✓ Pipe (fork twice)
  - ✓ Shell script





Shell (2/5)

- Sample example

```
/* Simple shell, Kyoungmoon Sun(msg2me@msn.com), */
/* Dankook Univ. Embedded System Lab.  2008/7/2  */
#include <unistd.h>
...

bool cmd_help( int argc, char* argv[]) {
                ...
}
int tokenize( char* buf, char* delims, char* tokens[], int maxTokens ) {
                ...
                token = strtok( buf, delims );
                while( token != NULL && token_count < maxTokens ) {
                                ...
                }
}
bool run( char* line ) {
                ...
                token_count = tokenize( line, delims, tokens, sizeof( tokens ) / sizeof( char* ) );
                // handling internal command such as cd, stty and exit
                // handling redirection, pipe and background processing
                if( (child = fork()) ==  0 ) {
                                execvp( tokens[0], tokens );
                }
                wait (); ...
}
int main() {
                char line[1024];
                while(1) {
                                printf( "%s $ ", get_current_dir_name() );
                                fgets( line, sizeof( line ) - 1, stdin );
                                if( run( line ) == false ) break;
                }
                ...
}
```

tokens[0] = “cat”
tokens[1] = “alphabet.txt”
or
tokens[0] = “gcc”
tokens[1] = “-o”
tokens[2] = “hello”
tokens[3] = “hello.c”

same as execlp(“cat”, “cat”, “alphabet.txt”, (char *)0);

$ cat alphabet.txt
or
$ gcc –o hello hello.c

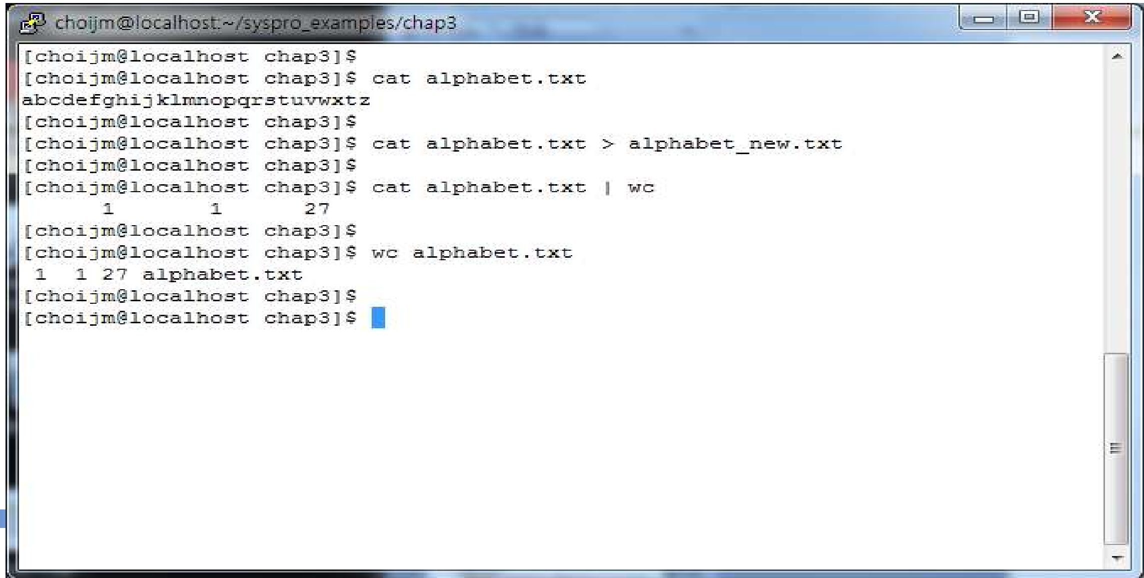
Shell (3/5)

- Execution example

```
choijm@localhost:~/syspro_examples/chap5
[choijm@localhost chap5]$ ls
exam1.c      execl_test.c    execl_test3    fork_test.c    hello
exam1.o      execl_test2     execl_test3.c  fork_test2     hello.c
execl_test   execl_test2.c   fork_test      fork_test2.c   mysh.c
[choijm@localhost chap5]$
[choijm@localhost chap5]$ gcc -o mysh mysh.c
[choijm@localhost chap5]$
[choijm@localhost chap5]$ ./mysh
/home/choijm/syspro_examples/chap5 $
/home/choijm/syspro_examples/chap5 $ ls
exam1.c      execl_test.c    execl_test3    fork_test.c    hello     mysh.c
exam1.o      execl_test2     execl_test3.c  fork_test2     hello.c
execl_test   execl_test2.c   fork_test      fork_test2.c   mysh
/home/choijm/syspro_examples/chap5 $
/home/choijm/syspro_examples/chap5 $ gcc -o hello hello.c
/home/choijm/syspro_examples/chap5 $
/home/choijm/syspro_examples/chap5 $ ./hello
Hello World
/home/choijm/syspro_examples/chap5 $
/home/choijm/syspro_examples/chap5 $ help
/***************Simple Shell****************/
You can use it just as the conventional shell

Some examples of the built-in commands
cd          : change directory
exit        : exit this shell
quit        : quit this shell
help        : show this help
?           : show this help
/*******************************************/
/home/choijm/syspro_examples/chap5 $
/home/choijm/syspro_examples/chap5 $ ps
  PID TTY          TIME CMD
  307 pts/1    00:00:00 ps
32568 pts/1    00:00:00 bash
32765 pts/1    00:00:00 mysh
/home/choijm/syspro_examples/chap5 $
/home/choijm/syspro_examples/chap5 $ exit
[choijm@localhost chap5]$
[choijm@localhost chap5]$ ps
  PID TTY          TIME CMD
  308 pts/1    00:00:00 ps
32568 pts/1    00:00:00 bash
[choijm@localhost chap5]$
```

Shell (4/5)

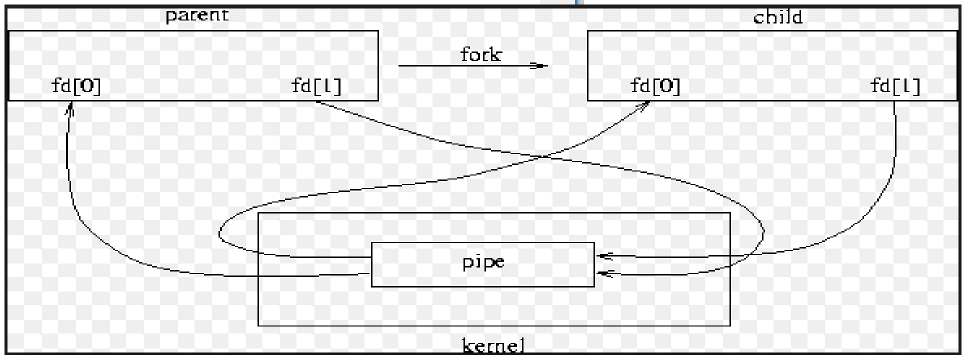
- Background processing
  - ✓ both shell and command run concurrently
  - ✓ how to: do not use wait()
- Redirection
  - ✓ read/write data from/to file instead of STDIN/STDOUT
  - ✓ how to: replace STDIN/STDOUT with file's fd using dup2() before execve() (➔ refer to LN3)
- pipe
  - ✓ create two processes and make them communicate via pipe
  - ✓ how to: replace STDIN/STDOUT with fd[0]/fd[1] using pipe() and dup2() before execve()



Shell (5/5)

- pipe() example
  - ✓ One of IPC (Inter Process Communication) mechanisms

```
choijm@embedded: ~/syspro18/chap5
choijm@embedded:~/syspro18/chap5$ vi pipe_exam.c
choijm@embedded:~/syspro18/chap5$
choijm@embedded:~/syspro18/chap5$ cat pipe_exam.c
/* Pipe example by J. Choi, choijm@dankook.ac.kr */
#include <unistd.h>
#include <stdio.h>

int main()
{
     int fd[2];
     char bufc[16], bufp[16];
     int read_size = 0;

     pipe(fd);     // need to handle if exceptions occur
     if (fork() == 0) {
          write(fd[1], "Thank you", 10);
          sleep(1);
          read_size = read(fd[0], bufc, 16);
          bufc[read_size] = '\0';
          printf("%s by pid %d\n", bufc, getpid());
          exit(0);
     }
     else {
          read_size = read(fd[0], bufp, 16);
          bufp[read_size] = '\0';
          printf("%s by pid %d\n", bufp, getpid());
          write(fd[1], "My pleasure", 12);
          wait();
          close(fd[0]); close(fd[1]);
     }
}
choijm@embedded:~/syspro18/chap5$ gcc -o pipe_exam pipe_exam.c
choijm@embedded:~/syspro18/chap5$ ./pipe_exam
Thank you by pid 838
My pleasure by pid 839
choijm@embedded:~/syspro18/chap5$
```



Advanced Process Programming (1/10)

- Until now
  - ✓ We have learned about the fork() and execve()
  - ✓ We can create multiple processes and run multiple programs

- From now on
  - ✓ Advanced process related system calls
    - signal, nice, gettimeofday, ptrace
  - ✓ Multiple processes raise several issues
    - Scheduling and Context switch
    - Memory management (memory sharing/protection)
    - IPC (Inter Process Communication)
    - Race condition and Synchronization
    - thread
    - …

Advanced Process Programming (2/10)

- Process-related system calls
  - ✓ Advanced
    - signal(), kill(), alarm() : signal handling such as register a signal handler (signal catch function) and signal delivery
    - sleep(), pause() : block for a certain period or until receiving a signal
    - nice(), getpriority(), setpriority() : control process priority
    - sched_setscheduler(), sched_getscheduler(), sched_setparam(), sched_getparam() : control process scheduling policy and parameters
    - times(), gettimeofday() : get timing information of a process and get the current time
    - ptrace() : allow a process to control the execution of other processes

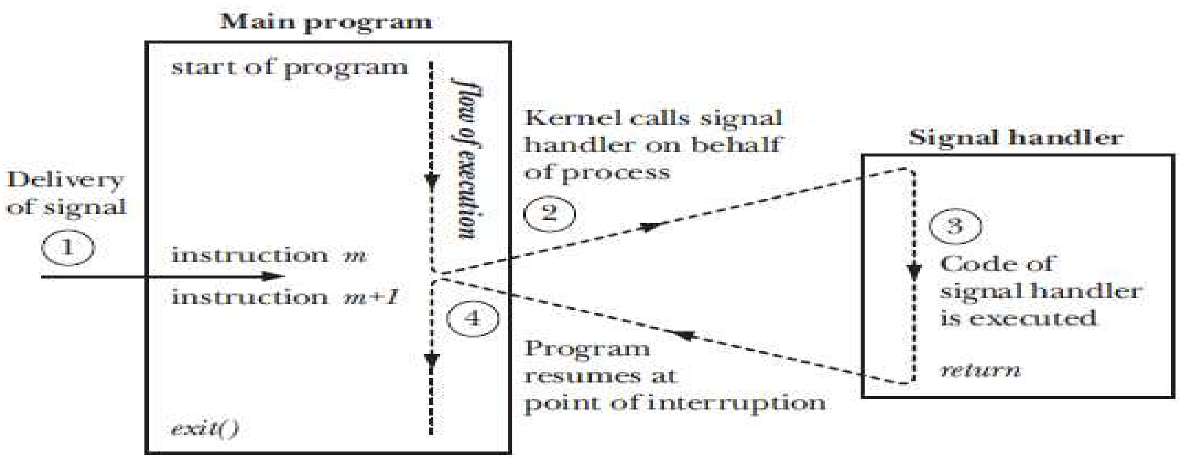


Figure 20-1: Signal delivery and handler execution **(Source: LPI)**

- Process-related system calls
  - ✓ File descriptor after fork()

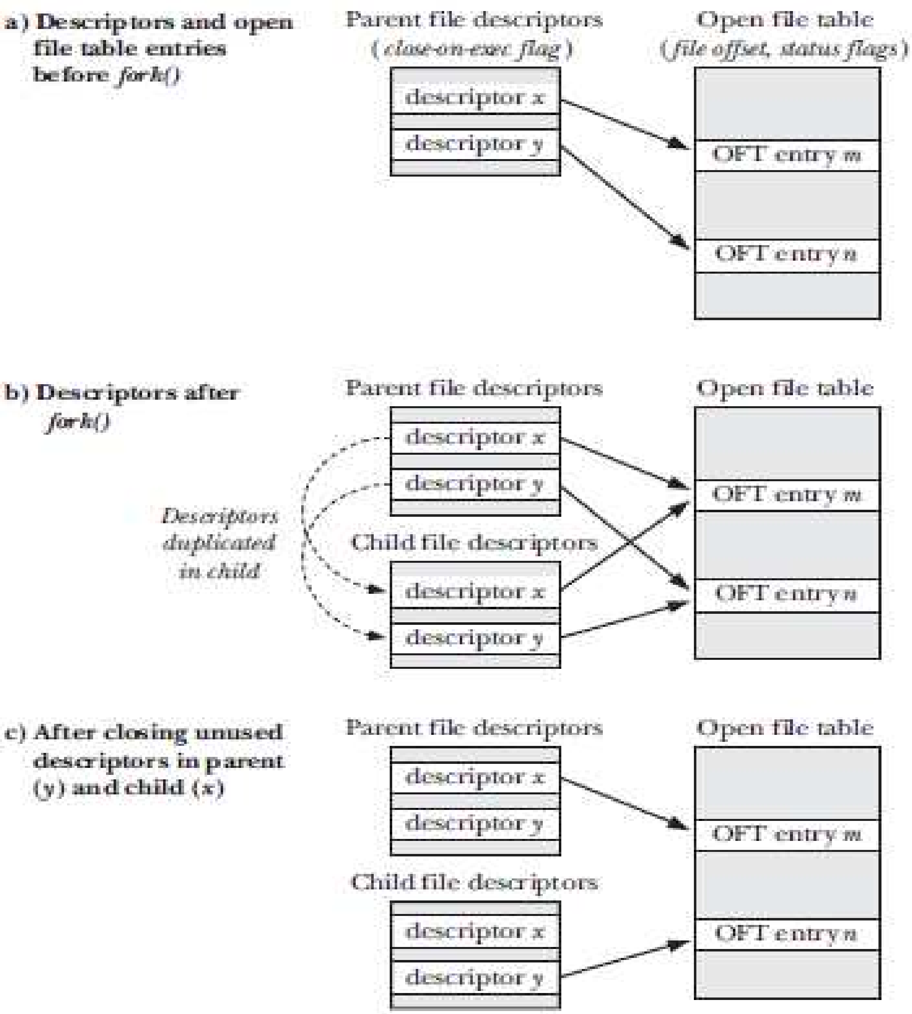


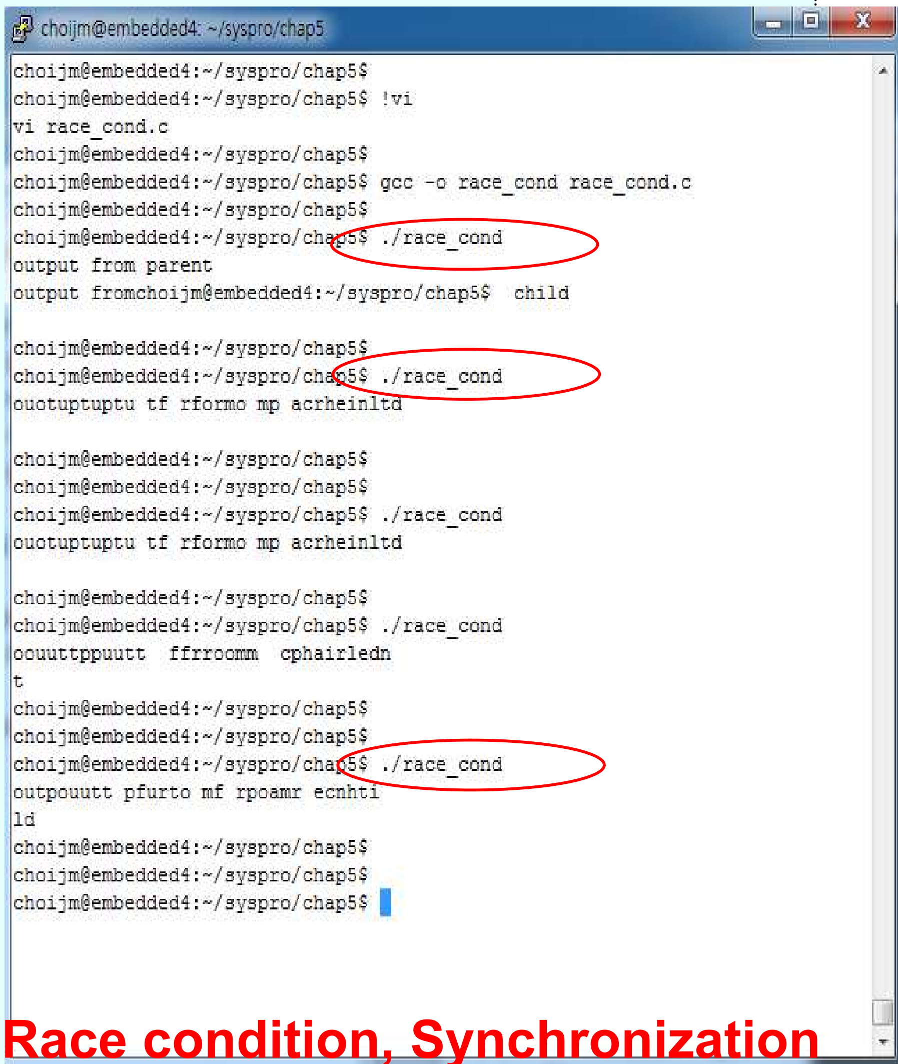
Figure 24-2: Duplication of file descriptors during *fork()*, and closing of unused descriptors

(Source: LPI)

Advanced Process Programming (4/10)

- Race condition

```
/* Race condition example by choijm. From Advanced Programming in UNIX Env.*/
#include   <sys/types.h>
#include   <unistd.h>

static void charatatime(char *str) {
   int i;

   for (; *str; str++) {
      for (i=0; i<1000; i++);
      write(STDOUT_FILENO, str, 1);
   }
}

int main(void) {
   pid_t   pid;

   if ( (pid = fork()) < 0) {
      perror("fork");
      exit(1);
   } else if (pid == 0) {
      charatatime("output from child\n");
   } else {
      charatatime("output from parent\n");
   }
}
```



☞ **Concurrency, Shared resource, Race condition, Synchronization**

Advanced Process Programming (5/10)

- When two processes run concurrently

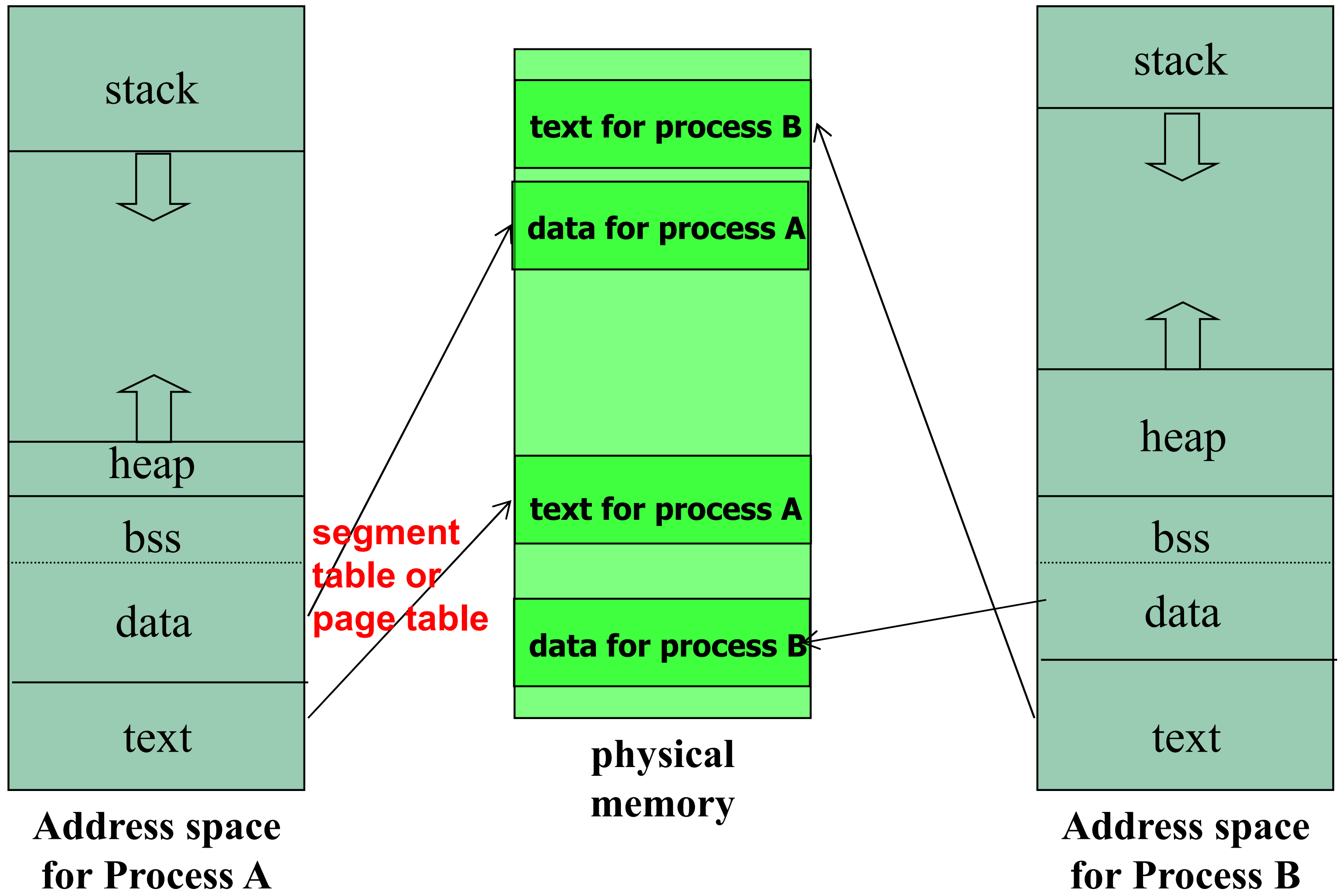
```c
/* virtual_address.c: printing memory address , Oct. 9, choijm@dku.edu  */

int glob1, glob2;
main()
{
   int m_local1, m_local2;

   printf("process id = %d\n", getpid());
   printf("main local: \n\t%p, \n\t%p\n", &m_local1, &m_local2);
   printf("global: \n\t%p, \n\t%p\n", &glob2, &glob1);

   while (1);
}
```

☞ **Virtual address**

```
choijm@embedded4: ~/syspro/chap5
choijm@embedded4:~/syspro/chap5$ vi virtual_address.c
choijm@embedded4:~/syspro/chap5$
choijm@embedded4:~/syspro/chap5$ gcc -o virtual_address virtual_address.c
choijm@embedded4:~/syspro/chap5$
choijm@embedded4:~/syspro/chap5$ ./virtual_address &
[1] 16579
choijm@embedded4:~/syspro/chap5$ process id = 16579
main local:
        0xffca1294,
        0xffca1290
global:
        0x80496f4,
        0x80496f0

choijm@embedded4:~/syspro/chap5$ ./virtual_address &
[2] 16580
choijm@embedded4:~/syspro/chap5$ process id = 16580
main local:
        0xffdcaba4,
        0xffdcaba0
global:
        0x80496f4,
        0x80496f0

choijm@embedded4:~/syspro/chap5$
choijm@embedded4:~/syspro/chap5$ ps
  PID TTY          TIME CMD
15085 pts/1    00:00:01 bash
16579 pts/1    00:00:09 virtual_address
16580 pts/1    00:00:07 virtual_address
16581 pts/1    00:00:00 ps
choijm@embedded4:~/syspro/chap5$
```

Advanced Process Programming (6/10)

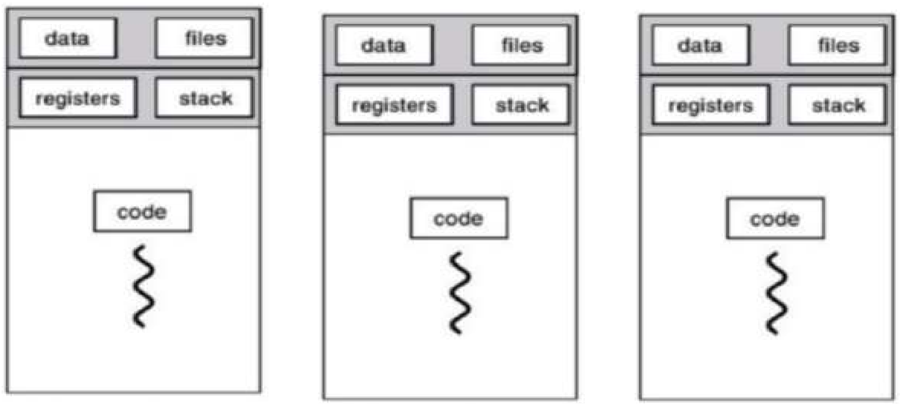
- When two processes run concurrently (cont')



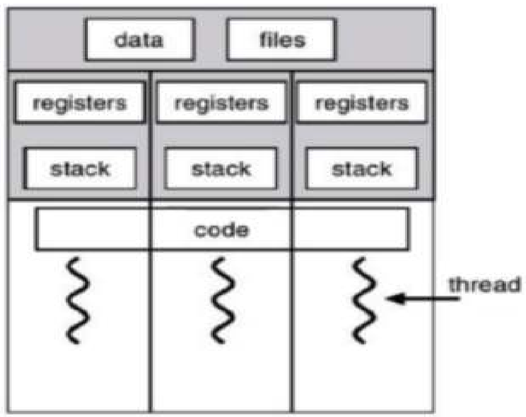
☞ **Please do not forget "killing the background process" after prev. experiment**

Advanced Process Programming (7/10)

- Thread introduction
  - ✓ Process
    - Data: independent model
    - Pros: 1) isolation, 2) easy to debug
    - Cons: 1) slow, 2) need explicit IPC (Inter-process communication)
  - ✓ Thread
    - Data: shared model
    - Pros: 1) fast and use less memory, 2) sharing
    - Cons: 1) all threads are killed if a thread has a problem, 2) hard to debug



VS



(Source: https://www.toptal.com/ruby/ruby-concurrency-and-parallelism-a-practical-primer)

Advanced Process Programming (8/10)

- Thread: programming example

```
// fork example
// by J. Choi  (choijm@dku.edu)
#include <stdio.h>
#include <stdlib.h>

int a = 10;

void *func()
{
        a++;
        printf("pid = %d\n", getpid());
}

int main()
{
        int pid;

        if ((pid = fork()) == 0) { //need exception handle
                func();
                exit(0);
        }
        wait();
        printf("a = %d by pid = %d\n", a, getpid());
}
```

```
// thread example
// by J. Choi (choijm@dku.edu)
#include <stdio.h>
#include <stdlib.h>

int a = 10;

void *func()
{
        a++;
        printf("pid = %d\n", getpid());
}

int main()
{
        int p_thread;

        if ((pthread_create(&p_thread, NULL, func, (void
*)NULL)) < 0) {
                exit(0);
        }
        pthread_join(p_thread, (void *)NULL);
        printf("a = %d by pid = %d\n", a, getpid());
}
```

Advanced Process Programming (9/10)

- Thread: compile and execution

```
choijm@sungmin-Samsung-DeskTop-System: ~/syspro/chap5
choijm@sungmin-Samsung-DeskTop-System:~/syspro/chap5$ vi fork_sharing_test.c
choijm@sungmin-Samsung-DeskTop-System:~/syspro/chap5$
choijm@sungmin-Samsung-DeskTop-System:~/syspro/chap5$ vi thread_sharing_test.c
choijm@sungmin-Samsung-DeskTop-System:~/syspro/chap5$
choijm@sungmin-Samsung-DeskTop-System:~/syspro/chap5$ gcc -o fork_sharing_test fork_sharing_test.c
choijm@sungmin-Samsung-DeskTop-System:~/syspro/chap5$
choijm@sungmin-Samsung-DeskTop-System:~/syspro/chap5$ gcc -o thread_sharing_test thread_sharing_test.c -lpthread
choijm@sungmin-Samsung-DeskTop-System:~/syspro/chap5$
choijm@sungmin-Samsung-DeskTop-System:~/syspro/chap5$ ./fork_sharing_test
pid = 16134
a = 10 by pid = 16133
choijm@sungmin-Samsung-DeskTop-System:~/syspro/chap5$
choijm@sungmin-Samsung-DeskTop-System:~/syspro/chap5$ ./thread_sharing_test
pid = 16135
a = 11 by pid = 16135
choijm@sungmin-Samsung-DeskTop-System:~/syspro/chap5$
choijm@sungmin-Samsung-DeskTop-System:~/syspro/chap5$ more thread_sharing_test.c
// thread example
// by J. Choi (choijm@dku.edu)
#include <stdio.h>
#include <stdlib.h>

int a = 10;

void *func()
{
    a++;
    printf("pid = %d\n", getpid());
}

int main()
{
    int p_thread;

    if ((pthread_create(&p_thread, NULL, func, (void *)NULL)) < 0) {
        exit(0);
    }
    pthread_join(p_thread, (void *)NULL);
```

Advanced Process Programming (10/10)

- Process structure with multiple threads
  - ✓ Text, Data, Heap and multiple Stacks

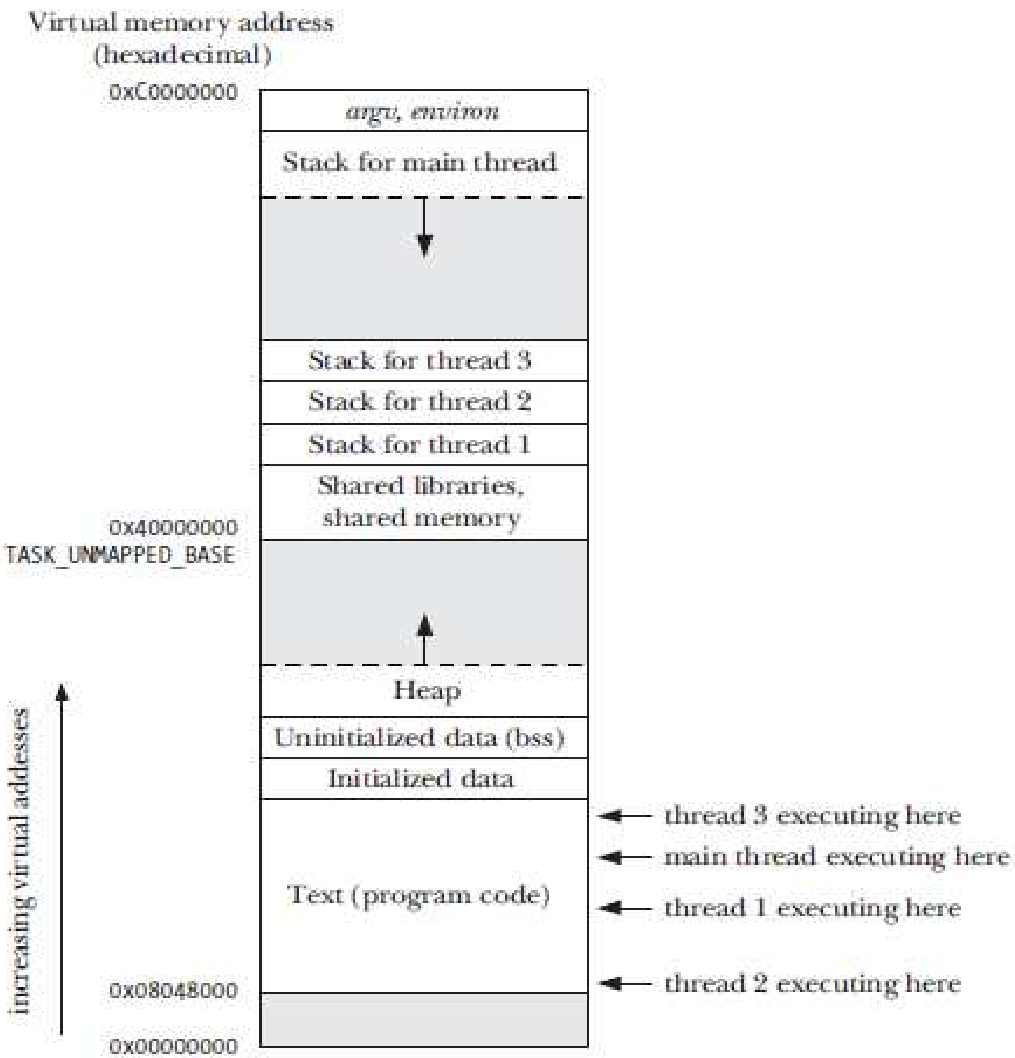


Figure 29-1: Four threads executing in a process (Linux/x86-32)

(Source: LPI)

Summary

- Understand how to create a process & execute a program
- Grasp the role and internals of shell
- Discuss issues on multitask
  - ✓ IPC (Inter Process Communication)
  - ✓ Race condition
  - ✓ Virtual memory
  - ✓ Differences between process and thread

☞ **Homework 5: Make a shell (mysh)**

1.1 Requirements

- implement basic logic (parsing, fork(), execve())

- using Makefile

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

1.2 Bonus: implement redirection

1.3 Write a report

- 1) Introduction: What to do, How, …

- 2) Design and Source code description

- 3) Execution Snapshots

- 4) Discussion: what you learn, issues, …

1.4 How to submit? Send 1) report and 2) source code to mgchoi@dankook.ac.kr

1.5 Deadline: two weeks later (same time)

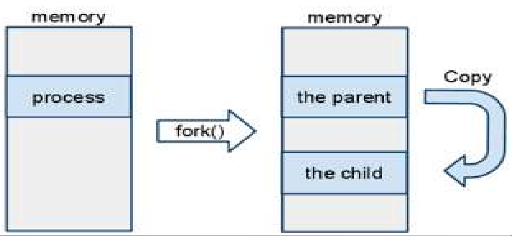
Quiz for this Lecture

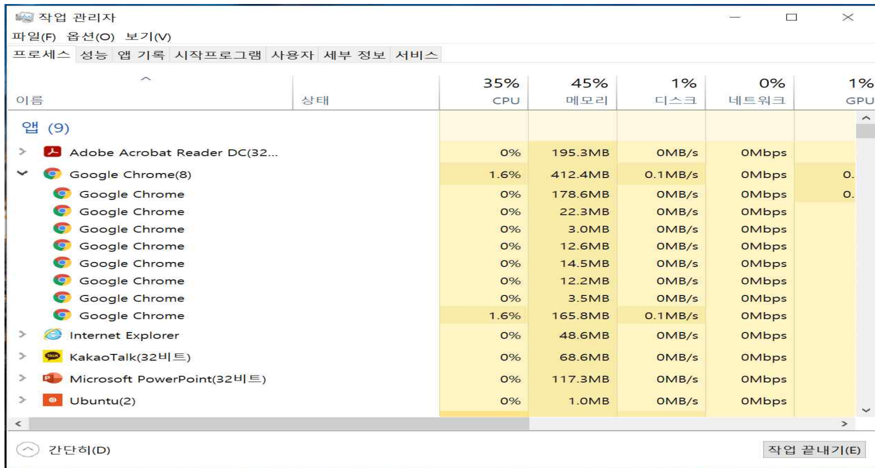
- Quiz
  - ✓ 1. After a program invokes three fork() calls, how many new processes will result? (from LPI 24-1). Discuss what is the “fork bomb attack”?
  - ✓ 2. Why we separate the data segment into two parts, “data” and “bss”?
  - ✓ 3. Discuss the role the wait() system call using the terms of shared resource, race condition and synchronization.
  - ✓ 4. Each tab of a browser can be implemented either process or thread. Which is better? Explain your own opinion.

24.7 Exercises

24-1. After a program executes the following series of *fork()* calls, how many new processes will result (assuming that none of the calls fails)?

```
fork();
fork();
fork();
```





작업 관리자

파일(F) 옵션(O) 보기(V)

프로세스 성능 앱 기록 시작프로그램 사용자 세부 정보 서비스

이름	상태	35% CPU	45% 메모리	1% 디스크	0% 네트워크	1% GPU
앱 (9)						
Adobe Acrobat Reader DC(32...		0%	195.3MB	0MB/s	0Mbps	
Google Chrome(8)		1.6%	412.4MB	0.1MB/s	0Mbps	0.
Google Chrome		0%	178.6MB	0MB/s	0Mbps	0.
Google Chrome		0%	22.3MB	0MB/s	0Mbps	
Google Chrome		0%	3.0MB	0MB/s	0Mbps	
Google Chrome		0%	12.6MB	0MB/s	0Mbps	
Google Chrome		0%	14.5MB	0MB/s	0Mbps	
Google Chrome		0%	12.2MB	0MB/s	0Mbps	
Google Chrome		0%	3.5MB	0MB/s	0Mbps	
Google Chrome		1.6%	165.8MB	0.1MB/s	0Mbps	
Internet Explorer		0%	48.6MB	0MB/s	0Mbps	
KakaoTalk(32비트)		0%	68.6MB	0MB/s	0Mbps	
Microsoft PowerPoint(32비트)		0%	117.3MB	0MB/s	0Mbps	
Ubuntu(2)		0%	1.0MB	0MB/s	0Mbps	

간단히(D) 작업 끝내기(E)

Appendix 1

- Revisit "gdb"

```
choijm@embedded: ~/syspro

int tokenize(char *line , char *tokens[], int maxToken) {
    int t_cnt = 0;
    char *token, *delimiter = " \n";

    token = strtok(line, delimiter);

    while(token && t_cnt < maxToken) {
        tokens[t_cnt++] = token;
        token = strtok(NULL, delimiter);
    }
    tokens[t_cnt] = '\0';
    return t_cnt;
}

bool run(char *line) {
    pid_t pid;  int i, j, fd, t_cnt;
    bool t_bg = false;
    char *tokens[10];

    t_cnt = tokenize(line, tokens, sizeof(tokens) /sizeof(char *));
    if(t_cnt == 0) return true;
    if(strcmp(tokens[0], "exit") == 0) return false;
"mysh.c" 72 lines --30%--                                   27,10-13
```

```
choijm@embedded: ~/syspro
choijm@embedded:~/syspro$ vi mysh.c
choijm@embedded:~/syspro$ gcc -g -o mysh mysh.c
choijm@embedded:~/syspro$ gdb mysh
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu
This is free software: you are free to change and redi
There is NO WARRANTY, to the extent permitted by law.
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to
Reading symbols from mysh...done.
(gdb) br 27
Breakpoint 1 at 0x80486ec: file mysh.c, line 27.
(gdb) run
Starting program: /home/choijm/syspro/mysh
/home/choijm/syspro$ ls -l

Breakpoint 1, run (line=0xffffd4b0 "ls -l\n") at mysh.
27              t_cnt = tokenize(line, tokens, sizeof(
(gdb) p line
$1 = 0xffffd4b0 "ls -l\n"
(gdb) p tokens[0]
$2 = 0x41fd <error: Cannot access memory at address 0x
(gdb) n
28              if(t_cnt == 0) return true;
(gdb) p tokens[0]
$3 = 0xffffd4b0 "ls"
(gdb) p tokens[1]
$4 = 0xffffd4b3 "-l"
(gdb) c
```

Appendix 1

- Typical gdb commands

Command	Effect
Starting and stopping	
`quit`	Exit GDB
`run`	Run your program (give command line arguments here)
`kill`	Stop your program
Breakpoints	
`break sum`	Set breakpoint at entry to function `sum`
`break *0x8048394`	Set breakpoint at address `0x8048394`
`delete 1`	Delete breakpoint 1
`delete`	Delete all breakpoints
Execution	
`stepi`	Execute one instruction
`stepi 4`	Execute four instructions
`nexti`	Like `stepi`, but proceed through function calls
`continue`	Resume execution
`finish`	Run until current function returns
Examining code	
`disas`	Disassemble current function
`disas sum`	Disassemble function `sum`
`disas 0x8048397`	Disassemble function around address `0x8048397`
`disas 0x8048394 0x80483a4`	Disassemble code within specified address range
`print /x $eip`	Print program counter in hex
Examining data	
`print $eax`	Print contents of `%eax` in decimal
`print /x $eax`	Print contents of `%eax` in hex
`print /t $eax`	Print contents of `%eax` in binary
`print 0x100`	Print decimal representation of `0x100`
`print /x 555`	Print hex representation of 555
`print /x ($ebp+8)`	Print contents of `%ebp` plus 8 in hex
`print *(int *) 0xfff076b0`	Print integer at address `0xfff076b0`
`print *(int *) ($ebp+8)`	Print integer at address `%ebp` + 8
`x/2w 0xfff076b0`	Examine two (4-byte) words starting at address `0xfff076b0`
`x/20b sum`	Examine first 20 bytes of function `sum`
Useful information	
`info frame`	Information about current stack frame
`info registers`	Values of all the registers
`help`	Get information about GDB

Figure 3.30 **Example GDB commands.** These examples illustrate some of the ways GDB supports debugging of machine-level programs.

(Source: CSAPP)

Appendix 2

- Shell example in CSAPP

code/ecf/shellex.c

```c
#include "csapp.h"
#define MAXARGS   128

/* Function prototypes */
void eval(char *cmdline);
int parseline(char *buf, char **argv);
int builtin_command(char **argv);

int main()
{
    char cmdline[MAXLINE]; /* Command line */

    while (1) {
        /* Read */
        printf("> ");
        Fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);

        /* Evaluate */
        eval(cmdline);
    }
}
```

code/ecf/shellex.c

Figure 8.22 The main routine for a simple shell program.

code/ecf/shellex.c

```c
/* eval - Evaluate a command line */
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];   /* Holds modified command line */
    int bg;              /* Should the job run in bg or fg? */
    pid_t pid;           /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return;   /* Ignore empty lines */

    if (!builtin_command(argv)) {
        if ((pid = Fork()) == 0) {   /* Child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }

        /* Parent waits for foreground job to terminate */
        if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else
            printf("%d %s", pid, cmdline);
    }
    return;
}

/* If first arg is a builtin command, run it and return true */
int builtin_command(char **argv)
{
    if (!strcmp(argv[0], "quit")) /* quit command */
        exit(0);
    if (!strcmp(argv[0], "&"))    /* Ignore singleton & */
        return 1;
    return 0;                     /* Not a builtin command */
}
```

code/ecf/shellex.c

Figure 8.23 eval: Evaluates the shell command line.

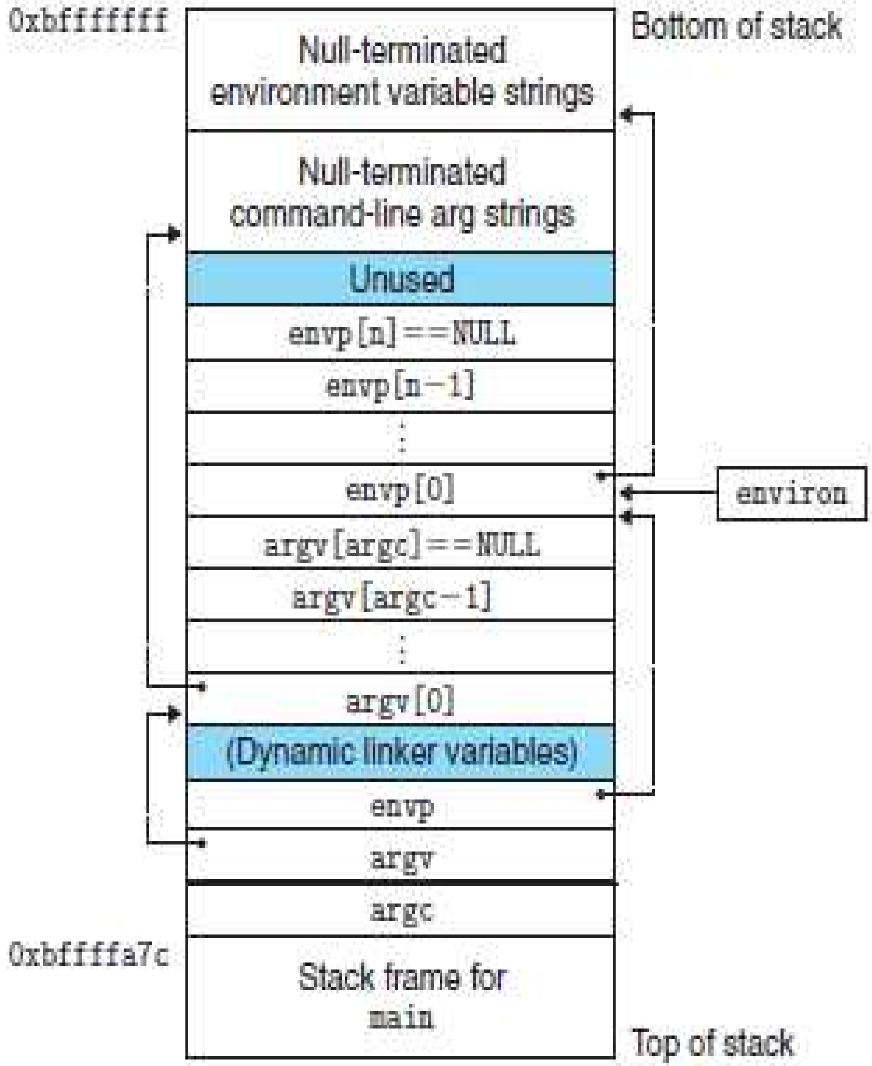
Appendix 3

- Relation between execve() and main()

Figure 8.21
Typical organization of the user stack when a new program starts.

```
#include <unistd.h>

int execve(const char *filename, const char *argv[],
           const char *envp[]);
                        Does not return if OK, returns −1 on error
```

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



(Source: CSAPP)

Appendix 4

- Binary format
  - ✓ Real view in Linux with commands size and readelf

```
choijm@LAPTOP-LR5HOQBH: ~/Syspro/LN4
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ more test.c
#include <stdio.h>

int a = 10;
int b = 20;
int c;

int main()
{
        c = a + b;
        printf("C = %d₩n", c);
}
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ gcc -c test.c
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ size test.o
   text    data     bss     dec     hex filename
    156       8       0     164      a4 test.o
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ gcc test.c
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ size a.out
   text    data     bss     dec     hex filename
   1595     608       8    2211     8a3 a.out
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ objdump -h a.out

a.out:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .interp       0000001c  0000000000000318  0000000000000318  00000318  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.gnu.property 00000020  0000000000000338  0000000000000338  00000338  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .note.gnu.build-id 00000024  0000000000000358  0000000000000358  00000358  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .note.ABI-tag 00000020  000000000000037c  000000000000037c  0000037c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .gnu.hash     00000024  00000000000003a0  00000000000003a0  000003a0  2**3
```

```
choijm@LAPTOP-LR5HOQBH: ~/Syspro/LN4
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ readelf -a a.out
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1060
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14784 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000000318  00000318
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.propert NOTE             0000000000000338  00000338
       0000000000000020  0000000000000000   A       0     0     8
  [ 3] .note.gnu.build-i NOTE             0000000000000358  00000358
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .note.ABI-tag     NOTE             000000000000037c  0000037c
       0000000000000020  0000000000000000   A       0     0     4
  [ 5] .gnu.hash         GNU_HASH         00000000000003a0  000003a0
       0000000000000024  0000000000000000   A       6     0     8
  [ 6] .dynsym           DYNSYM           00000000000003c8  000003c8
       00000000000000a8  0000000000000018   A       7     1     8
  [ 7] .dynstr           STRTAB           0000000000000470  00000470
```