Chapter 5.1

Process Management

General Presentation and Linux System Calls

Print Version of Lectures Notes of *Operating Systems*

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

Adrian Coleșa

April 3rd, 2019

5.1.1

Purpose and Contents

The purpose of today's lecture

- Presents general aspects related to process management
- Give examples and details about Linux system calls for processes

5.1.2

Bibliography

- A. Tanenbaum, *Modern Operating Systems*, 2nd Edition, 2001, Chapter 2, Processes, pg. 71 – 100, pg. 132 – 151

5.1.3

Contents

5.1.4

1 General Aspects

Process Definition

- Longman dictionary's definition of **process**
  - *a series of actions that are done in order to achieve a particular result*
- **a program in execution** $\Leftrightarrow$ an user application
  - a **sequential** stream of **execution** in its own **memory address space**
  - including the current **values of CPU's registers** (e.g. IP)
- **OS abstraction for using the computer**
  - composed by all that is needed to run a program: CPU, memory, I/O devices etc.
  - it is a **virtualization concept** $\rightarrow$ **virtualizes an entire system** (computer)
  - $\Rightarrow$ **isolation mechanism**, i.e. isolates one execution (process) by another

5.1.5

Process vs Program

- program's source code $\xrightarrow{compilation}$ program's executable $\xrightarrow{launching}$ process
- **program = static (inactive) entity**
- **process = active entity**
- a process
  - is an activity of some kind
  - is created from a program loaded in memory
  - is allocated system resources (memory, file descriptors, CPU etc.)
  - has input, output, and a state
- the two **parts of a process**
  - **sequential execution**: no concurrency inside a process; everything happens sequentially
  - **process state**: everything that process interacts with (registers, memory, files etc)

5.1.6

The Considered Context

- single-processor systems
- multiprogramming and time (processor) sharing
  - pseudo-parallelism
  - switching among processes
  - scheduling algorithm

5.1.7

Process Creation

- automatically by the OS (uncommon case)
  - at system initialization
  - reacting to different events
  - usually run as background processes (vs. foreground processes)
- **by another process** (common case)
  - there is a **system call** provided for process creation
  - situations
    - when a process needs help doing some computation
    - when a user action occurs, e.g. interaction with the shell
  - leads to a **process hierarchy** based on the **parent-child** relationship

5.1.8

Process termination

- **voluntarily**, using a special **system call**
  - **normal exit**, i.e. end of program's execution
  - **error detection exit**, like: inexistent files, insufficient or incorrect input etc.
- **involuntarily**, being forcefully terminated
  - initiated by the system due to a "fatal error", like: illegal instructions, division by zero, segmentation fault etc.
  - initiated by another process (i.e. "killed")

5.1.9

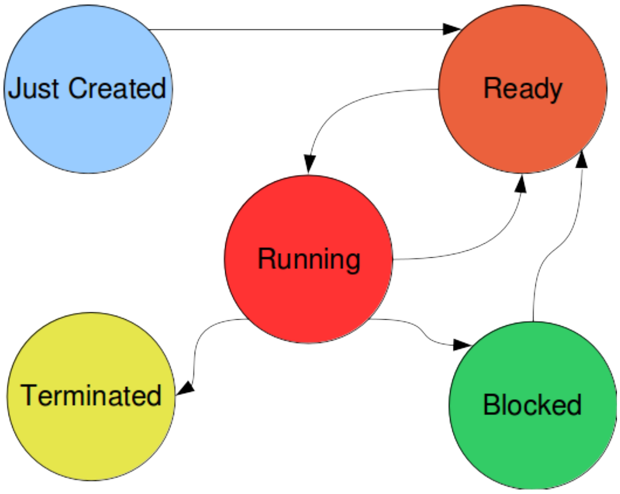


Figure 1: Process States Transition

Process states

- **running**
  - **executed by the CPU**, i.e. using the CPU, at that moment
  - only one process in that state / CPU, actually as many as the number of system's CPUs
- **ready**
  - ready to be executed, but no CPU available
  - so **wait for a CPU to become available**
  - **transparent to the program**
- **blocked**
  - **wait for an event to occur**, a resource to become available
  - **triggered by the application explicitly** through **blocking system calls**
- just created (optional)
  - waiting for some resources to be allocated
- **terminated** (optional)
  - keeping information about the exit state

5.1.10

Process States Transitions

5.1.11

2 Linux Processes

2.1 System Calls

Process Creation: `fork()`

- system call used to **create a new process**
- child process' contents is identical with that of its parent
- *still, two distinct and independent processes*
- the two processes are scheduled independently on the CPU
- parent processes continue its execution returning from `fork`
- child starts its execution returning from `fork`
- `fork` returns
  - a positive value (child's PID) in parent
  - zero in child

5.1.12

fork Usage Example

```
int x;
static int y;
int *px;

int main(int argc, char **argv)
{
    int pid;

    x = 0;
    px = &x;
    y = 0;

    // up to this point only the parent exists
    // now parent callds fork() to create a new process
    pid = fork();
    if (pid < 0) {
        // error case: no child process created
        perror("Cannot create a new process");
        exit(1);
    }
    // from now on there are two processes: parent and child

    // executed by both processes
    printf("x=%d, px =%p, *px=%d, y=%d\n", x, px, *px, y);
       // parent: x=0, px =0x601050, *px=0, y=0
       // child:  x=0, px =0x601050, *px=0, y=0

    if (pid == 0) { // executed only by the child
        y = 20;
        *px = 200;
    } else {        // executed only by the parent
        y = 10;
        *px = 100;
    }

    // executed by both processes
    printf("x=%d, px=%p, *px=%d, y=%d\n", x, px, *px, y);
       // parent: x=100, px=0x601050, *px=100, y=10
       // child:  x=200, px=0x601050, *px=200, y=20
}
```

5.1.13

fork's Effect Illustration

5.1.14

fork() syscall creates an *independent* child process, which starts as a *copy of its parent!*

5.1.15

Let's practice!

Have you really understood how fork() works?

If you have, try solving the following problems:

1. You are given the following code:

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

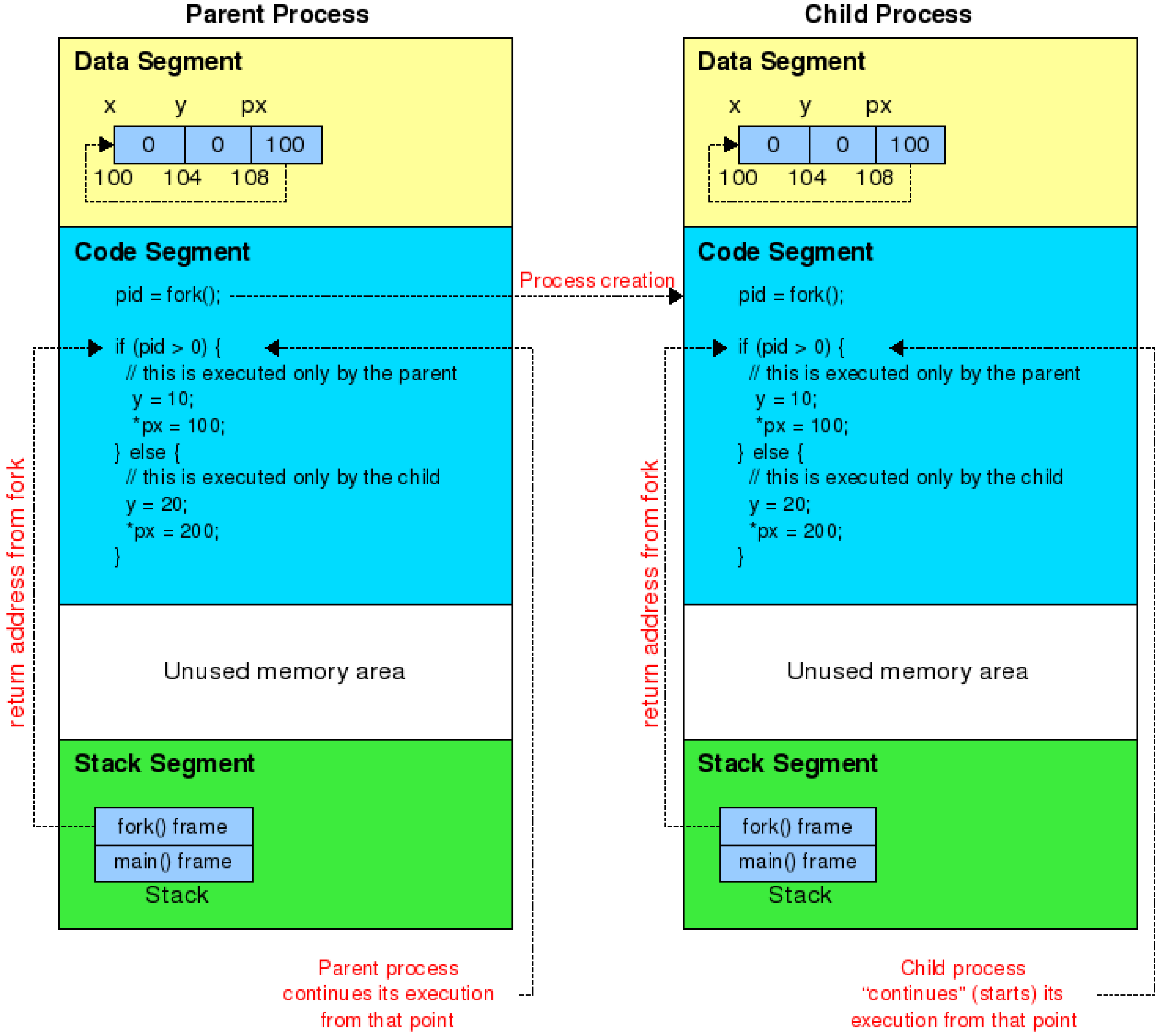
   - How many processes does the following code creates?
   - Draw the resulted process hierarchy.

2. You are given the following code:

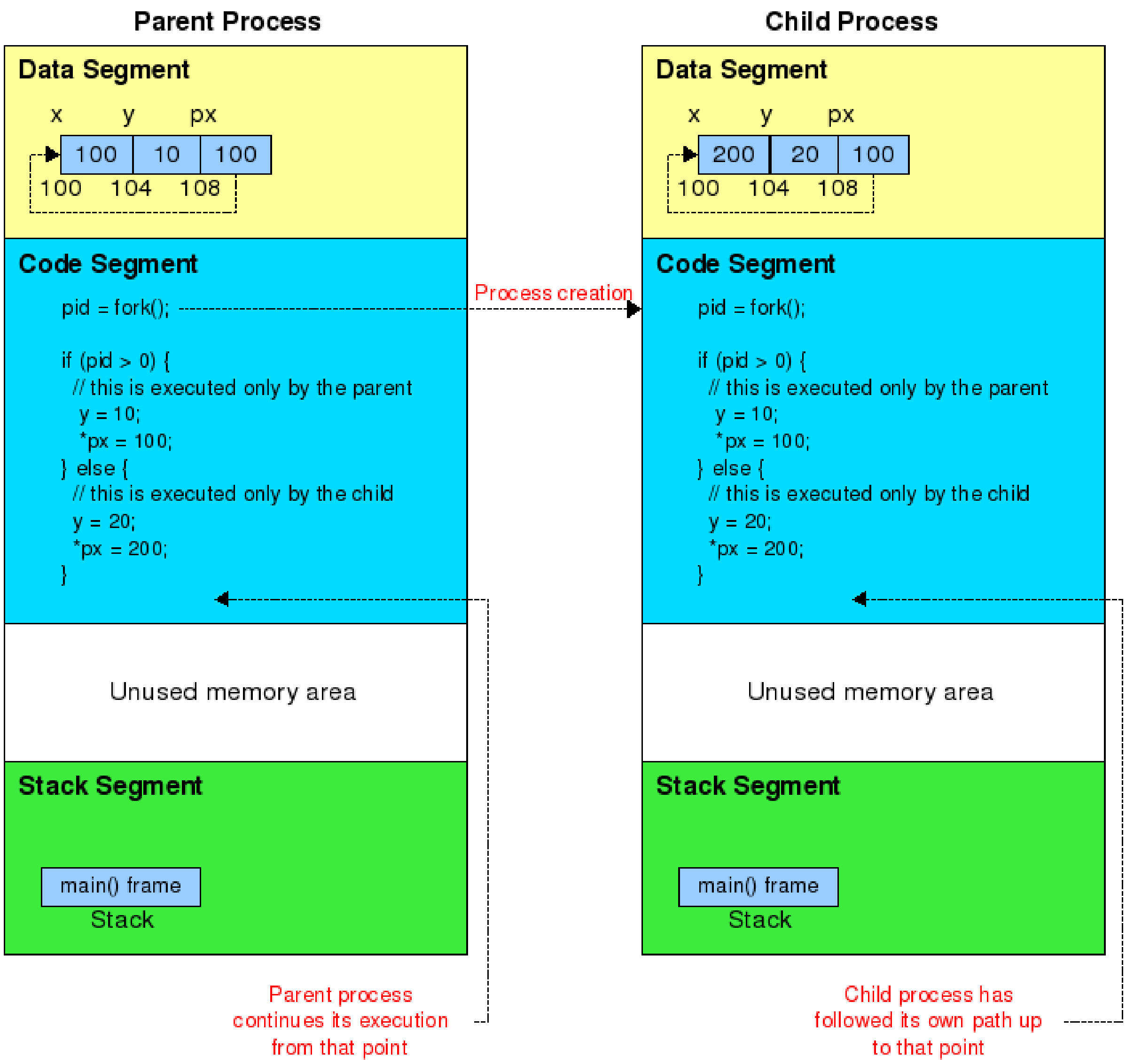
```
for(i=1; i<=100; i++)
    fork();
```

   - How many processes does the following code creates?
   - Draw the resulted process hierarchy.

5.1.16



Processes Status Immediately After Fork.
The Child Contents Is Identical With That of Its Parent



Processes Status After Some Time.
The Two Processes Evolve Independently

Code Execution: exec Family

- system call used to **load a new code into the calling process**
  - replace the calling process' contents, but not its identity
- there are more exec system calls
  - `execl`, `execlp`: with variable number of arguments
  - `execv`, `execvp`: with a fixed number of arguments
- the `exec`'s parameters **similar to a command line**
  - the first argument is **always the path** to the executable file
  - the next argument(s) describe the command line, *starting with command name*

5.1.17

execl and execlp Usage Example

```
// first parameter is the EXPLICIT path to the executable file
execl("/bin/ls", "ls", "-l", NULL);
execl("./myprg.exe", "myprg.exe", "param1", "param2", NULL);

// first parameter is the IMPLICIT path to the executable file
// the path is searched in the directories stored
// in the PATH environment variable
execlp("ls", "ls", "-l", 0);
```

5.1.18

execv and execvp Usage Example

```
char cmdline[10][100]; // equiv. to char *cmdline[];
                        // equiv. to char **cmdline;

// build the command line
strncpy(cmdline[0], "ls", 99);
strncpy(cmdline[1], "-l", 99);
cmdline[2] = NULL;

// call the exec

// first parameter is the EXPLICIT path to the executable file
execv("/bin/ls", cmdline);

// first parameter is the IMPLICIT path to the executable file
// the path is searched in the directories stored
// in the PATH environment variable
execvp("ls", cmdline);
```

5.1.19

exec() syscalls loads a new code in the calling process! There is no return from exec() if successfully executed!

5.1.20

Relationship Between fork and exec

- used to
  - create a child process
  - executing something else than its parent
- **Why there are two separated steps instead of just one?**
  - between them the parent "has control" over its child (see standard input and output redirection below)
  - the **parent is released** by the burden of (i.e. time spent) **loading a new code in child**
  - ⇒ **better performance** for the parent

5.1.21

fork and exec Usage Example

- parent code

```
int main()
{
    pid = fork();

    if (pid > 0) {
        // parent doing something
    } else {
        // child loading and executing a new code
        execl("./child.exe", "child.exe", "p1", "10", 0);
        perror("execl has not succeded");
    }
}
```

- child code

```
int main(int argc, char **argv)
{
    int p;
    for (p=0; p<argc; p++)
        printf("argv[%d]=%s\n", argv[p]);
}
```

5.1.22

Process Termination: exit

- system call used to **terminate voluntarily a process**
- terminate the calling process
- specify an exit code
  - 0 (zero) exit code considered successfully termination
  - anything else considered erroneously termination
- exit code is kept until the parent process asks for it
- example
  - `exit(0);`
  - `exit(1);`

5.1.23

Wait For Termination: wait and waitpid

- system calls used by a process to **wait for the termination of its children**
- return the exit code of the terminated child
- example

```
int child_status;

wait(&child_status);

printf("Child process terminated with code %d\n",
        WEXITSTATUS(status));
```

5.1.24

Relationship Between wait and exit

- a way to **synchronize two processes**' execution (parent and child)
- a simple way to **communicate between processes** (parent and child)
- when a (parent) process terminates
  - all its children get as their new parent a system process
    - on older systems: the *init* process, having pid = 1
    - on newer systems: a per user *init* process
- when a (child) process terminates before its parent
  - its state is said to be **zombie** and
  - its exit state is maintained by OS until its parent process asks for it or terminates

5.1.25

Let's practice!

You are given the following C code and are required to:

1. Draw the process hierarchy corresponding to the processes created by the code below.
2. Specify the number of times each *printf* is executed, supposing every instruction is executed successfully.

```
printf ("[1] Hello world!\n");

pid = fork();

printf ("[2] Hello world!\n");

pid = fork();

printf ("[3] Hello world!\n");

if (pid == 0) {
    execlp("ps", "ps", 0);
    printf ("[4] Hello world!\n");
}

fork();

printf ("[5] Hello world!\n");
```

5.1.26

2.2 Examples

Shell Basic Code (Functionality)

```
char **cmdline; // it must be build like argv param of main
while (TRUE) {
    display_prompt_on_screen();
    cmdline = read_cmd_line();

    pid = fork();      // creates a new process
    if (pid < 0) {
        perror("canot creat a new process");
        continue;
    }

    if (pid == 0)
        execvp(cmdline[0], cmdline);
    else
        waitpid(pid, NULL, 0);
}
```

5.1.27

Standard Input Redirection

- command line

  ```
  cat < file.txt
  ```

- STDIN redirection in C program

  ```
  pid = fork();

  if (pid > 0) {
      // parent
  } else {
      // child
      fd = open("file.txt", O_RDONLY);
      dup2(fd, 0);
      close(fd);

      execlp("cat", "cat", 0);
      perror("execl has not succeded");
  }
  ```

5.1.28

Standard Output Redirection

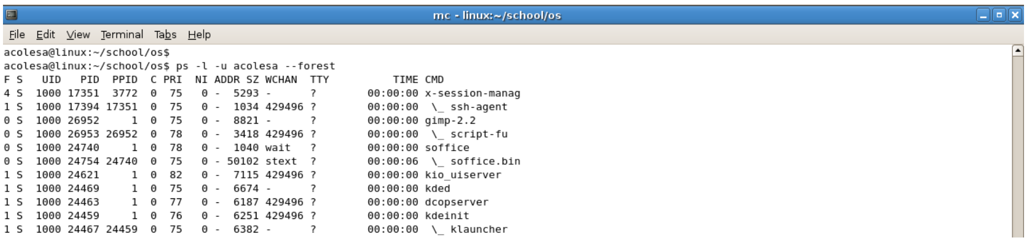
- command line

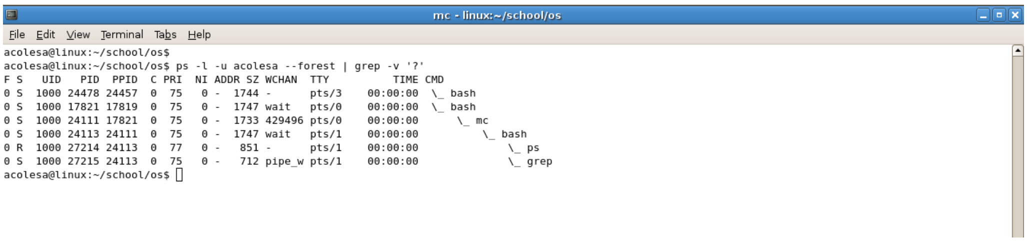
  ```
  ls > file.txt
  ```

- STDIN redirection in C program

  ```
  pid = fork();

  if (pid > 0) {
      // parent
  } else {
      // child
      fd = creat("file.txt", 0600);
  ```






```
        dup2(fd, 1);
        close(fd);

        execlp("ls", "ls", 0);
        perror("execl has not succeded");
    }
```

5.1.29

2.3 Relates Issues

ps Command

- displays a snapshot of the active processes in the system
- `ps -l -u acolesa --forest`
- `ps -l -u acolesa --forest | grep -v '?'`
- `ps -l -e --forest | head -n 50`

5.1.30

top and htop Commands

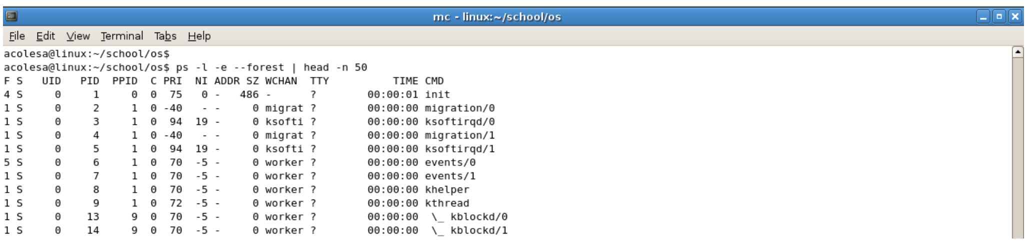
- display a continuously updated list of processes and their on-line scheduling

5.1.31

proc File System

- It is a pseudo file system
- It is mounted in `/proc`
- It is used by the OS to display information about processes
  - each process has a directory named with the process id
  - reading this information is similar to reading any other files and dirs

5.1.32



3 Conclusions

What we talked about

- process definition
- process states and state transitions
  - running, ready, blocked, terminated
- system calls to create and terminate a process
- Linux system calls related to processes
  - fork
  - exec
  - exit
  - wait

5.1.33

Lessons Learned

- process is a virtualization and isolation concept
  - virtualize an entire compute for a program's execution
  - isolate one execution by another
- process states
  - running: the desired one
  - ready: exists due to limited no of CPUs; is transparent to processes
  - blocked: triggered explicitly by a process due to a blocking syscall
- create a new process
  - `fork()` called by the parent to create an identical child process
  - `exec()` called by the child to load new code
- every process terminates with `exit()`

5.1.34