Cairo University

Computer Engineering Dept. Cairo University	Operating Systems Lab 3 Process Management	Fall 2020

Introduction

In this experiment, we will test the process management and scheduling in a UNIX-like operating system. In UNIX, any instance of a running program is called a **process**. All UNIX commands are *programs*, and thus contribute processes to the system when they are running. Every time you issue a foreground command, UNIX starts a new process, and suspends the current process (the terminal or shell) until the new process completes.

Each process on UNIX in its abstract form can be analyzed in five components:

- ***Process code***, which contains the executable portion of the process.
- ***Process data*****,** which contains the static data of the process.
- ***Process stack*****,** which contains temporary data of the process.
- ***User area*****,** which holds the information about signal handling, opened files, and another CPU info for the process.
- ***Page tables*****,** which are used for memory management.

Each process can run in one of two modes during its life:

- ***User Mode***, where the process may only execute certain instructions, and access its own data, and can't access any privileged instructions or data.
- ***Kernel Mode*****,** where the process can execute any instruction or access any piece of data. To enter the kernel mode, this must be done by executing a system call which one of many system calls that are used to talk to the kernel. (The kernel is the brain of the operating system in which all the major decisions are taken)

The kernel keeps track of all processes on the system by using a process table, which contains information about each process (this information will be soon clarified). Each process on the system has a unique ID number, which is called **process id** (or PID). It also has **a user id** (or UID) and **group id** (or GID), which are the process owner's UID and GID.

Another thing to be found for every process in the process table is the **process state** and **location of its code, data, stack, and user area.**

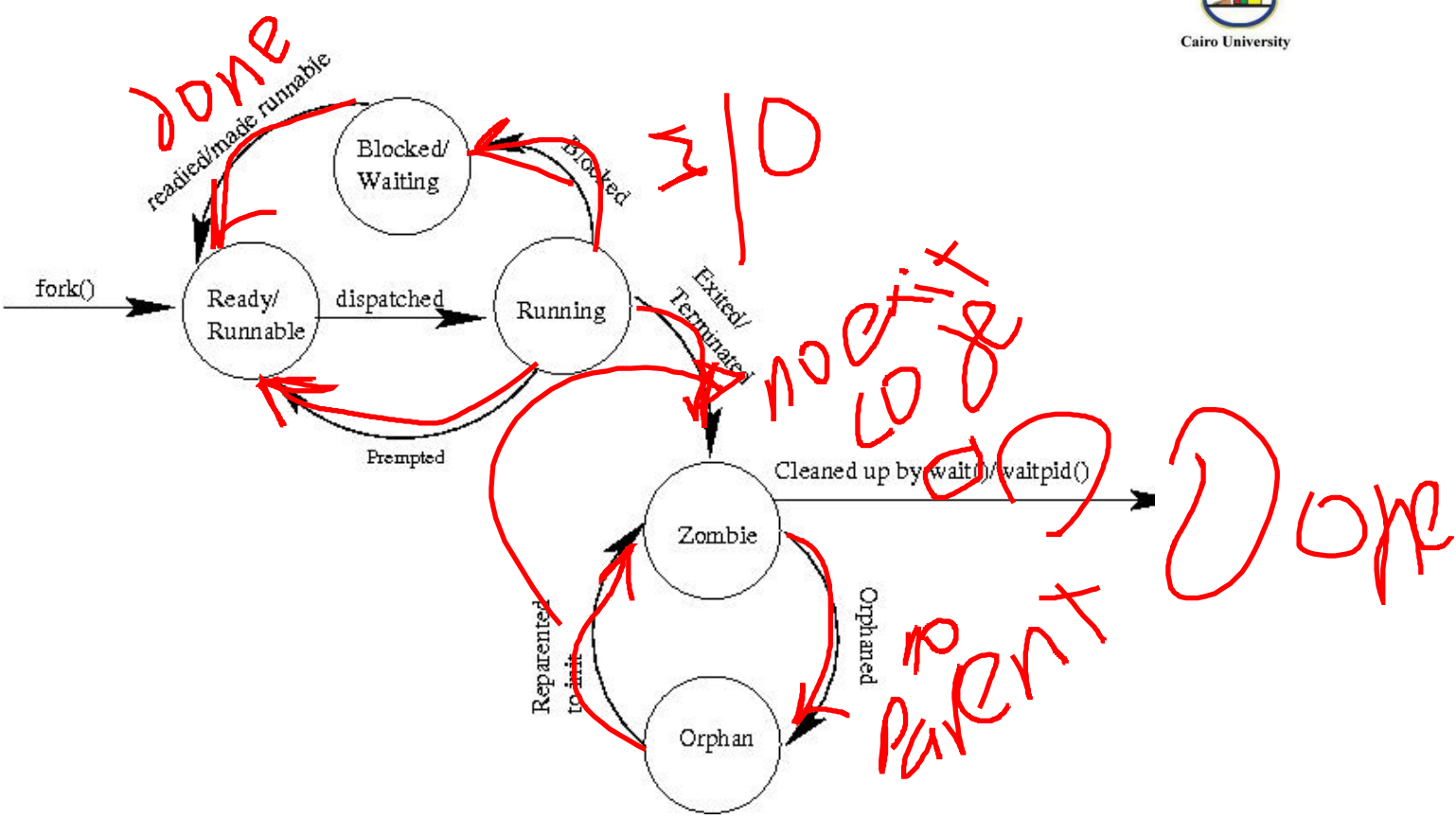
Also each process has a parent that **spawns** it, or **forks** it in UNIX language, except the first process created in the system that is created exclusively when the system is starting up without being spawned by another one and this process creates the "init", father of all processes in the system, process which has a PID = 1.

Process states in UNIX

During its life course, a process may have one of the following states:

- ***Running***: has control over the CPU now.
- ***Ready***: can be running at any time when the CPU is available.
- ***Sleeping***: waiting for an event to occur.
- ***Stopped***: it is frozen by a special signal.
- ***Zombified***: when it terminates but has not returned its exit code to its parent.

Cairo University



The Scheduler

The scheduler in UNIX works in a multi-level priority scheme. It means that processes are classified in several categories, and each category has a certain priority and processes are arranged in a queue on which Round-Robin scheduling technique is performed.

Reminder:

- To compile the source code files use: gcc file_name.c -o file_name
- To run the executable in your terminal type: ./file_name
- To kill a running process in the foreground use ctrl+c
- To kill a running process in the foreground/background use the unix command “ps”, find the process id that you want to kill and use the unix command “kill -9 <process_id>”

Experiment Steps:

1. The first step is to test the behavior of the process creation with fork() system call by compiling and running the code in process01.c

2. Test the case of an orphan process by adding the following to the child code:

```
sleep(3);
printf("\nI am now an orphan child, my pid = %d and my parent's pid = %d\n\n", getpid(), getppid());
```

3. Before terminating, the parent should wait to receive exit code from all its children. To test this, add the following code to the parent code
```
pid = wait(&stat_loc);
printf("\nChild terminated with status %d", WEXITSTATUS(stat_loc));
```

4. A child can send a specific exit code to the parent. To see that add the following statement at the end of the child code:
```
exit(42)
```

5. If a child could not return its exit code to its parent, it becomes zombified. Compile the code in process05.c and run it in background. Run ps several times and notice the state of the child process.

6. The execl function replaces the current process image with a new process image. The new image is constructed from a regular, executable file. Add the following line to the child code in process01.c to see how execl works:
```
execl("/bin/ps", "ps", "-f", NULL);
```

7. The code in process07.c has a parent that forks a child and then both of them enter an infinite loop. Compile and run this code in background. Try different forms of ps and top commands to check the processes status. Use kill command to kill the parent then the child and check the status of the processes.

8. The nice system call adjusts the process scheduling priority. Compile and run the code in process08.c to see how it works

Functions Wrap-up

Function: fork (void)
Return: pid_t
Desc: fork() creates a child process that differs from the parent process only in its PID and PPID. Penalty that it incurs is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child. On success, the PID of the child process is returned in the parent's thread of execution, and a 0 is returned in the child's thread of execution. On failure, a -1 will be returned in the parent's context, no child process will be created, and errno will be set appropriately.

Function: sleep (int seconds)
Return: int
Desc: The sleep() function shall cause the calling thread to be suspended from execution until either the number of real-time seconds specified by the argument seconds has elapsed or a signal is delivered to the calling thread and its action is to invoke a signal-catching function or to terminate the process. The suspension time may be longer than requested due to the scheduling of other activity by the system. The remaining seconds is returned.

Function: wait (int * status)
Return: pid_t (pid)
Desc: Waits for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then terminated the child remains in a "zombie" state. If successful, wait returns the process ID of the child process. If unsuccessful, a -1 is returned. wait (&status) is equivalent to waitpid(-1, &status, 0);

Function: waitpid(pid_t pid, int * status, int opts)
Return: pid_t (pid)
Desc: Same as wait but for a specific process

Function: getpid(void)
Return: pid_t (pid)
Desc: Gets the process id of the current process

Function: getppid(void)
Return: pid_t (pid)
Desc: Gets the parent process id of the current process

Function: getpgrp(void)
Return: pid_t (pid)
Desc: Gets the parent process id of the current process

Macro: WEXITSTATUS(int status)
Return: int
Desc: This macro queries the child termination status provided by the wait and waitpid functions. If the WIFEXITED macro indicates that the child process exited normally, the WEXITSTATUS macro returns the exit code specified by the child process. It's equivalent to (status >> 8).

Macro: WIFEXITED(int status)
Return: int
Desc: This macro queries the child termination status provided by the wait and waitpid functions, and determines whether the child process ended normally. 1 means normal, 0 means a failure. It's equivalent to (status & 0x00FF)

Function: execl(void)
Return: int
Desc: The exec family of functions shall replace the current process image with a new process image. The new image shall be constructed from a regular, executable file called the new process image file. There shall be no return from a successful exec, because the calling process image is overlaid by the new process image. The exec() functions return only if an error has occurred. The return value is -1, and errno is set to indicate the error.

Function: exit (int status)
Return: void
Desc: The function exit() terminates the calling process "immediately". Any open file descriptors belonging to the process are closed; any children of the process are inherited by process 1, init, and the process's parent is sent a SIGCHLD signal.

Function: nice (int inc)
Return: int
Desc: nice() adds inc to the nice value for the calling process. (A higher nice value means a low priority.) Only the superuser may specify negative increment, or priority increase.

Requirement:

Due to the large number of students enrolled in the course of Operating Systems this year, the process of evaluating whether each student should pass or not has become extremely difficult. As a result, our department decided to hire N teaching assistants such that each TA is responsible only for evaluating a single group of students. Each TA is required to decide whether a student should pass the course or not by calculating the grades of the student in the midterm and the final exam respectively. If the total grades **is equal to or exceeds** the minimum passing grade P, then the student passes the course. Otherwise, he fails the course. Finally, each TA should report the number of passing students from their assigned group. The department committee should receive the count of passing students from each TA

Write a C program that simulates this scenario using **forking concepts** with a single parent process (department committee) and N children processes (TAs).

The program accepts **three** arguments from the command line:

1. A file path indicating the grades of all students enrolled in the course. All grades are non-negative integer values. Grades Path
2. N: The number of TAs **(N>0)**
3. P: The minimum pass grade. **(P>0)**

Input File Format:

- The **first** line in the file indicates the number of students (S) in the list.
- Each line in the following **S** lines represents the grades of a single student. It consists of two integers separated by a space. The first integer is the grade of the student in the midterm exam and the second integer is the grade of the student in the final exam.

Output Format:

- The program should output **ONE** line **ONLY** as follows:

1. The output line consists of N integers separated by a space such that each integer represents the number of passing students in the TA's assigned group. **The first TA is assigned the first ⌊S/N⌋ students, the second is TA is assigned the second ⌊S/N⌋ students, and so on. The output should be ordered correctly. The number of TAs is not necessarily divisible by the number of students. In this case, the overflowing students are assigned to the N-th TA (or the last one).**

Example:

gcc passCounter.c -o passCounter.o
./ passCounter.o "test.txt" 3 40

Example for "test.txt":

```
9
12      25
18      34
10      66
20      10
13      20
7       45
8       52
0       41
1       5
```

Expected output:
2 1 2

Notes:

1. The output of your program should be written as the expected output displayed in the previous example. Do not do any of the following:
    a) The expected output is: 2 1 2
    b) Expected output is 2 1 2
    c) Expected: 2 1 2
    d) Result: 2 1 2
    e) Result is 2 1 2
    f) >> 2 1 2
    g) 2
       1
       2
    h) 2, 1, 2
    i) 2-1-2
    j) 2/1/2

2. Do necessary printing for debugging your code (or better use a debugger). Do not forget to comment these printf functions before submission.

3. Do not resort to any user inputs from the standard input (i.e. Do not use scanf/gets/etc.) Your code will report a time-out exception in the test cases. The inputs should be given as arguments from the terminal.

4. All the test cases are valid inputs. You do not need to check for any invalid inputs.

5. You are required to use forking in this lab. You have to decide the parent's responsibilities and each child responsibilities. You also have to find out how you will communicate between both processes.

6. IMPORTANT NOTE (1): Do not assume any assumptions about the process IDs of the children. For example, do not assume that the first forked process will have PID+1 where PID is the process ID of the parent process and so on.

7. IMPORTANT NOTE (2): Keep in mind that if a parent process creates a process, then the PID of the child process is not necessarily greater than that of the parent. Sometimes, the system wraps around to reusing the free lower PIDs. Therefore, do not make any assumptions about the sequence of children PIDs. Do not say the PID first forked children is surely less than that of the second forked one.

8. For forking, you may use **ONLY** the functions specified in this document.