Project 1 - A Process Scheduler

CO004 Projects on Operating Systems

March 13, 2017

Due Date for Grouping: **Mar. 31, 2017**
Due Date for Phase 1: **Apr. 14, 2017**
Due Date for Phase 2: **May 5, 2017**

1 Objective

- Understanding how a process scheduler works.
- Increasing experience with system calls.
- Learning UNIX signals and controlling methods of signals

2 Introduction

Processes are the basic execution entities in every modern operating system. Process control and management is therefore an important topic. In this project, we are going to implement a workable process scheduler in Linux systems. This process scheduler should read a number of jobs with different parameters (including arrival time, CPU requirements, etc.) and involves with process creation, suspension, as well as termination.

This project will help you understand UNIX system working principles and system calls. After completing the project, you should be familiar with programming with LINUX system calls and be able to write a medium-sized program (more than 1,000 lines of codes).

This is a group project. You can form a group with **at most 3 members**. You shall learn how to solve a middle-level problem with collaboration with your partners and learn how to manage your time under a tight schedule.

3 Programming environment

Your system shall be able to be executable under the following environments:

- *Operating Systems.* You must write and test you programs under the Linux operating systems (**MS Windows are not acceptable**). The preferred Linux OSes include CentOS, Ubuntu, Debian, etc. The final testing Linux platform will be released later.
- *Language.* You must implement it in either C or C++. Using languages other than the above two will is also not acceptable.

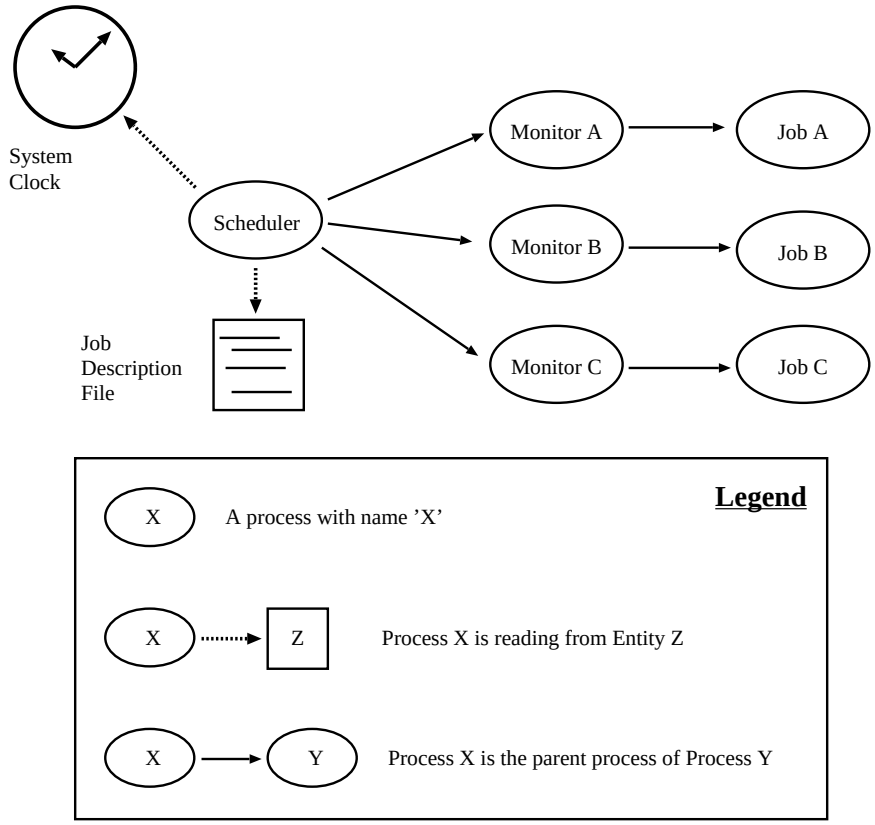

Figure 1: System architecture

- You are not allowed to invoke the `system(3)` library call. Otherwise, you would score 0 marks for this assignment.

Note that your development can be done in MacOS or other Linux/Unix platforms though the final program shall be tested in our given Linux.

4 System Design

Your design involves several entities that are depicted in Figure 1. In the following subsections, we introduce them one by one.

4.1 Scheduler process

The scheduler have the following major roles:

- Giving every process chances to be executed according to the defined scheduling policies;
- Maintaining internal status of each process so as to meet the requirements posed by the scheduling policy, such as fairness.

In addition, it has more things to do and we list the properties and the functionalities of the scheduler process as follows.

Job #1	`Arrival Time`	\t	`Command`	\t	`Duration`	\n
Job #2	`Arrival Time`	\t	`Command`	\t	`Duration`	\n
Job #3	.....					

Figure 2: The format of the job description file.

- The scheduler is an ordinary process, i.e., no special privilege should be given to the scheduler.
- The scheduler creates a set of jobs that are described in the job description file. The job description file contains a list of jobs that the scheduler should execute. The jobs may not be scheduled in a particular order, but according to the scheduling policy selected when the program starts.
- While scheduling, the scheduler reads from the real-time clock of the system to determine the time in scheduling.
- The scheduler should not be terminated until all the jobs are scheduled and are terminated.

4.2 System clock

Note hat the current time value is maintained by hardware. For simplicity of system implementation, we restrict the granularity of the time to be in terms of seconds. Although it is also possible for us to read the clock in the scale of microseconds (10^{-6}), this may make the design complicated. In practice, this simplified consideration may scarify the precision of the scheduling algorithms.

4.3 Job description file

This is a plain-text file that stores a list of job in the format shown in Figure 2. We describe the format shown as follows.

- There are **at most nine jobs** stated in a job description file.
- One line represents one job, i.e., every job description is separated by one newline character.
- The lines of job descriptions are sorted by the "`Arrival Time`" field, in ascending order.
- A line of job description must contain three non-empty fields: "`Arrival Time`", "`Command`" and "`Duration`". Every two fields are separated by exactly one tab character.
- The "`Arrival Time`" of a job is measured relative to the time that the scheduler starts executing and is measured in terms of seconds. The smallest value is 0 and the largest is the maximum value defined as "`INT MAX`" in the header file "`limits.h`".
  - When the scheduler has just been started, the time that is relative to the time that the scheduler starts executing is zero.

- If "`Arrival Time`" of a job is i, it means the job should start executing i seconds after the scheduler starts.

- The "`Command`" is a command string with the following constraints:
  - A command has a maximum length of 255 characters.
  - There will be no leading or trailing space characters in any commands.
  - A command is composed of a series of **tokens** (or words). The first token is the program name and the successive tokens are the program arguments. Every two tokens are separated by **exactly one space character**.

  Note that the first token may be the name of the program or the path to the program.

- The "`Duration`" is the number of seconds that the command is required to run. The possible values of the "`Duration`" can be either -1 or all non-zero positive integers. The value -1 means that the command is allowed to run indefinitely until it terminates. It is worth mentioning that the "`Duration`" is not the **accumulated CPU time of the process**, but the time elapsed after the job has been started. Although some of you may feel that it is a lousy implementation, this consideration can greatly reduce your workload.

The following shows an example job description file:

```
1  ls -lR / 10
2  ./while1 1
3  ls -R /home -1
```

According to the above example, when the first job is running, the second job arrives at the system. The time that the second job starts depends on the scheduling policy taken by the scheduler. Note that the jobs stated in the file are sorted according to the "`Arrival Time`" field in ascending order.

4.4 Monitor process

Referring back to Figure 1, the monitor process stands between the job process and the scheduler process. It is created by the scheduler process using the "`fork()`" system call and the "`exec()`" system call family. Its existence is only to serve one goal: to measure the real (or elapsed) time, the user time, and the system time taken by the job process.

A monitor process is a stand-alone program with the following required list of arguments.

```
$ ./monitor [command string]
```

where '$' means the prompt of the shell.

There are the following restrictions of a monitor process.

1. The monitor program should only accept one argument, i.e, the "`command string`". Otherwise, the monitor program terminates.

2. The name "`monitor`" is the required name of the monitor program.It must reside in the same directory as the scheduler program. Hence, in the above definition of invocation command line, we set it to be "`./monitor`".

3. The "`command string`" is the command to be invoked by the "*Job Process*". Note that the string can be a command with arguments.

4.4.1 Job process

The job process executes the command stated in the job description file. It is created by the monitor process using the "`fork()`" system call and the "`exec()`" system call family. A job has the following properties:

- A job process may print output to the console through both the standard output and the standard error streams.
- A job process may end before or after the time stated the "`Duration`" field in the job description file.
- A job process should not read from console or the standard input stream. Note that this is also a simplified assumption.
- A job process should not receive signals from other processes in the entire system, except its **parent monitor process**.
- A job process should not have any signal handling routine installed, meaning that every signal received should work in its default setting.

  Hint. The "`SIG_DFL`" argument value in the "`signal(2)`" system call may help you.

5 Scheduling Policy and Scheduler Internals

Your process scheduler has to support three basic scheduling policies:

- First-In First-Out (FIFO for short);
- Round Robin (RR for short);
- Shortest Job First (SJF for short).

Note that the SJF policy has two versions: **non-preemptive** and **preemptive**, you are required to implement the non-preemptive version as the basic requirement of this project.

5.1 Assumptions

To simplify the design of the scheduler, we the following assumptions:

- The scheduler is assumed to execute one job at a time. In other words, we assume that there is only one CPU with one core only.
- The scheduler itself will not be interrupted, meaning that it will not be suspended nor be killed while it is running.
- Every job will not create **pthreads** (e.g., using the `pthreadcreate()` call) or other processes.
- Every job is assumed to be executed successfully, meaning that the scheduler process will not meet the cases that the permission is denied nor the program cannot be found.

5.2 Scheduling policy and scheduler invocation

For one execution of the scheduler, it is bounded to one scheduling policy only. Every process under the supervision of the scheduler will be treated fairly and will follow the same policy without any priority over one another.

To invoke the scheduler, the following arguments are required:

```
$./scheduler [inputfilename] [policy]
```

where '$'is the shell prompt, "`./scheduler`" is the path to the scheduler program, "`[inputfilename]`" is the path to the job description file and "`[policy]`" is the scheduling policy which should take the following values:

- "`FIFO`" (which means the first-in first-out policy);
- "`RR`" (which means the round robin policy);
- "`SJF`" (which means the shortest job first policy).

  Note: do not type the quotes.

5.2.1 Reading the job description file

To read the job description file, the following library function calls may be useful to you: "`fgets()`" and "`strtok()`". **Assumption.** You can assume that the input file always has the correct format.

Hint: Queue data structure

While the file is read, data structures must be built in order to store the data read from the job description file. You are recommended to built a queue data structure in order that you can easily manage the jobs. E.g., under the SJF policy, the "*Duration field*" plays one of the major roles in ordering the queue.

5.2.2 Scheduler executes a job

The scheduler "forks and executes" the monitor process with the command string supplied in the job description file as the only program argument of the monitor process.

As mentioned in Section 4.4, the scheduler invokes the monitor program by supplying "`command string`" as the only argument. In turns, the monitor process creates the job process according the argument string supplied by the scheduler.

5.2.3 Scheduler changes the status of the job process

Since the scheduler process is not the parent of the job processes that are created by monitor process, the scheduler has no knowledge on the process information (e.g., PID) of the job processes. In order to change the status of the job processes, the scheduler process shall pass the control signals to the monitor process that will help to *relay* the signals to the corresponding job process. Figure 3 depicts this procedure. In particular, the monitor process shall only relay the following signals: `SIGTERM`, `SIGTSTP`, and `SIGCONT`. In other words, for each signal to be relayed, the monitor process should set up the corresponding signal handler. Inside that handler, the monitor process will send the same signal to the job process.

Hint. You may consider the following system calls: `signal()` and `kill()`.

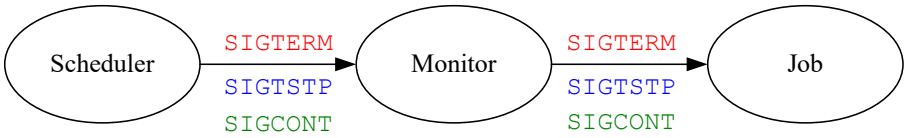

Figure 3: Signal relaying procedure

5.2.4 Scheduler checks if the job process is terminated

Since the scheduler does not has parent-child relationship with the job process, it is hard for the scheduler to detect the termination of the child process. The trick is: **When the job process terminates, the monitor process also terminates.**

To synchronize the scheduler, the monitor process and the job process, you need to consider the signal `SIGCHLD` and the system call `wait()` or `waitpid()`.

5.3 Finish scheduling

When the scheduler has finished scheduling all the jobs, it should print a **scheduling report**. The following is the example of the scheduling report:

```
Gantt Chart
===========
  Time    | 0                   1
          | 0 1 2 3 4 5 6 7 8 9 0 1 2
----------+---------------------------
  Job 1   |   # # # #
----------+---------------------------
  Job 2   |     . . . # # # #
----------+---------------------------
  Job 3   |       . . . . . . # # # #
----------+---------------------------
  Mixed   |   1 1 1 1 2 2 2 2 3 3 3 3
```

- The job name shown on each row, i.e, “Job1”, “Job2”, etc, is printed according to the order stated in the job description file. In other words, “Job$i$”is the ith line in the job description file.
- The symbol ‘#’ means that in the specified time period, a particular job is scheduled.
- The symbol ‘.’ means that in the specified time period, a particular job has arrived, but is not scheduled.

5.4 Monitor Process Internals

The monitor process works similar to the program “`/usr/bin/time`”.

- The monitor process “forks and executes” the command specified. That command becomes the job process.
- Then, when the job process terminates, no matter it is killed or self-terminating, the monitor process should know about it.

- The monitor process will print the timing report of the execution of the job process in the following format:

```
Process 1234 : time elapsed: 0.1900
               user time   : 0.1000
               system time : 0.0100
```

  In this example, "1234" is the PID of the job process. The "time elapsed" is the time period counting from the start of the job process to the termination of the job process and it is expressed in terms of seconds. The "usertime" is the user CPU time received by the job process. The "system time" is the system CPU time received by the job process.

  Hint. All the above time clicks can all be obtained by the correct use of the "`times(2)`" system call.

- After the timing report has been printed, the monitor process terminates.

5.5 Reports

Submit a report including:

- A design report including the system architecture, the flowcharts and descriptions of each module.
- A short manual about your program including how to use your program.
- All the source files (e.g., *.c or *.cpp and *.h) and Makefile (if any).
- The executable files are **NOT necessary** to be submitted.

6 Milestones

Make sure that you can understand all the assumptions and regulations of the system in Section 4.

6.1 Phase 1: Completion of the monitor program

The following is the list of tasks that you have to complete.

- Correctly invoke the job process with the correct program argument provided.
- Signal relaying mechanism works with `SIGTERM`, `SIGTSTP` and `SIGCONT`.
- Correc tmeasurement of the execution time of the job process using the `times()` system call. Note that the precision of the measured execution time may vary according to: (i) the background loading of the machine and (ii) the hardware configuration of the machine. Hence, correct measurement is not about the precision of the measured time; it is the way how you report the execution time.
- No zombie process is left in the system.

6.2 Phase 2: Completion of the entire project

The following is the list of tasks that you have to complete.

- The FIFO scheduler;
- The SJF scheduler;
- The RR scheduler with quantum set to 2 seconds;
- No zombie process is left in the system.
- Written Reports on your system according to Section 5.5.

7 Grading Policy

- Phase 1 (30%): Demonstration
- Phase 2 (50%): Demonstration
- Report (20%)

Late demonstration of each phase will lead to the score penalty.

8 Submission

Please submit your reports and program codes through http://moodle.must.edu.mo/.