SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

[CS205] Operating Systems Concepts with Android

Assignment #2

Purpose of this exercise

Upon successful completion of this exercise students should be able to do the following:

1. Perform system level (OS) programming with the C programming language in Linux.
2. Perform scheduling and management of a running state for processes in the system.

Overview

In this assignment you are required to design a **process manager** program for Linux that performs management of processes and scheduling of their execution that follows a policy described below, and then implement its functionality in the C programming language (versions `gnu11` or `C11`, or newer) .

This exercise consists of two parts distinct parts:

- Firstly, you need to author a **design document** describing your approach with pseudo-code.
- Secondly, you must provide a **source code** of a program implementing this approach.

The process management logic and the execution policy for the program are described below in the Requirements section. The details of the two parts of the assignment and their artefacts required for submission are described in detail in the Instructions section.

Requirements

Overview

Your task is to write a **process manager** program (`manager`) that allows a user to admit up to **64 processes**. The program must manage a **process state** of each admitted process. At any one time only a single process is allowed to execute, and multiple processes are scheduled according to **the shortest job first** policy.

Process states

A user can *run* a program, thus creating a process. When a process is first admitted, it is in the "**ready**" state. The process would only transition to the "**running**" state by the process manager according to the shortest job first policy. A user can *stop* a process, putting it in the "**stopped**" state. A stopped process can be resumed and hence transitioned to the "**ready"** state. A user can also *kill* a process, putting it in the "**terminated**" state, but a terminated process cannot be resumed.

You should design appropriate data structures to manage the processes of different states.

Scheduling policy

The scheduling policy is based on the shortest job first. Given several ready processes, the scheduler should run the one with the shortest remaining time first. The process should run until completion, unless preempted by another ready process with shorter remaining time. If the process gets preempted, it gets transitioned to ready state. If the process runs to completion, the next process to run is the ready process with the shortest remaining time.

Therefore, the scheduling policy works as follows:

- When the first process is admitted, it will be run immediately.
- A process runs to completion unless it gets preempted by a newly admitted process, i.e. when a new process is admitted, the scheduler checks for the remaining time of the current running process, as well as that of ready processes and the newly admitted process (which is just the expected runtime). The process with the shortest remaining time will be run next.
- Whenever a user stops a running process, resumes a stopped process, or kills a process, the scheduler selects the ready process with the shortest remaining time to run.

Note that in this assignment you only create process instances of an executable called `prog`. You will need to specify how long each process is run. This information will help you deduce the remaining runtime of processes (see examples below).

Commands

Your program should display a prompt:

```
cs205>
```

It should then interactively accept commands to list, run, stop, resume or terminate a process. The commands your implementation must support are as follows:

- `run [program] [arguments]`
  Run an executable program with given arguments.
- `stop [PID]`
  Put a process with the specified `PID` in the “stopped” state. If a “running” process is stopped, then dispatch another “ready” process with shortest remaining time.
- `kill [PID]`
  Terminate a process with the specified `PID`. If a "running" process is killed, then dispatch another “ready” process with shortest remaining time.
- `resume [PID]`
  Switch a “stopped” process with the specified `PID` to a “ready” state. Determine among the “ready” processes which one should be run next, possibly preempting the current running process.
- `list`
  List all the processes showing their `PID`, then `state`, ordering them by `PID`. Use the following mapping to represent the states:
  - `0`: running,
  - `1`: ready,
  - `2`: stopped,
  - `3`: terminated.
- `exit`
  Terminate all child processes if they have not yet been terminated, and exit from the parent process.

The executable file we will use for testing is `prog` that takes two arguments: a *file name*, and a number *n*. The executable `prog` overwrites the contents of the file with given *file name* every second for *n* seconds with the following text:

> *“Process ran `x` out of `n` secs“*

The source code of this program (`C++17`) is shared with you. A sample run of the `prog` after its complete execution is shown below:

```
$ ./prog abc.txt 10
$ cat abc.txt
Process ran 10 out of 10 secs
```

Examples

Example 1

- First we admit a process that runs 100s, and list the processes and observe that the first admitted process is “running” (`state=0`).
- One second later, we admit another process that runs for 200s, and list the processes and observe that the second process is “ready” (`state=1`) while the first process continues to run, because the first process has a remaining time of 99s, which is shorter than the second process with a remaining time of 200s.
- Another second later, we admit a new process that runs for 10s. We list the processes and observe that this third process becomes the running process, because it has the shortest remaining time (10s) compared to the first process (98s) and the second process (200s).

The expected output is shown below:

```
$ ./manager
cs205>run ./prog x10 100
cs205>list
7200,0
cs205>run ./prog x11 200
cs205>list
7200,0
7201,1
cs205>run ./prog x12 10
cs205>list
7200,1
7201,1
7202,0
```

Example 2

- First we admit three processes that run 100s, 200s, 300s, and list the processes. Since the second and third processes have a longer runtime, the first process is expected to run to completion first, unless preempted.
- 5 seconds later we stop the first process with a command `stop 7200`. We list the processes and observe that the second admitted process is now running. This is because among the two ready processes - the second and third only, since the first process is now stopped - the second one has a shorter remaining time: 200s, as compared to third process’ 300s.
- Another 5 seconds later, we kill the second process and observe that the third process begins to run, because it is the only "ready" process after the first process is stopped and the second is killed.
- Finally, 5 seconds later, we resume the first process. Now we have one "ready" process (the first process) and one "running" process (the third process). Among these two, the first process (remaining time of 95s) has a shorter job as compared to the third process (with remaining time of 295s).

The expected output is shown below:

```
$ ./manager
cs205>run ./prog x20 100
cs205>run ./prog x21 200
cs205>run ./prog x22 300
cs205>list
7200,0
7201,1
7202,1
cs205>stop 7200
cs205>list
7200,2
7201,0
7202,1
cs205>kill 7201
cs205>list
7200,2
7201,3
7202,0
cs205>resume 7200
cs205>list
7200,0
7201,3
7202,1
```

Instructions

Submission

To complete the assignment, complete the following 2 steps.

1. **Prepare the design document**

   Firstly, prepare and submit a design document describing your approach with pseudo-code in a *PDF format*, using no more than a single A4 page. This exercise is meant to ensure that you are on the right path before you start implementing your solution. While there are no marks allocated for the design document itself, the instructors will provide feedback based on your submission.

   The design document must be submitted by **4th Feb 2024, 23:59**.

2. **Implement the code**

   Secondly, write and test the code implementing your design. Your code must not contain any unnecessary external references, or third-party libraries (standard headers and libraries are permitted), and compile in a typical Linux environment (Ubuntu or a similar distribution will be used for grading) with the following compilation flags:

   ```
   gcc -Wall -Werror -Wextra -Wpedantic -Wstrict-prototypes -std=gnu11 -o manager manager.c
   ```

   For your reference, you can review the `build.sh` scripts provided with the examples from the in-class laboratory exercise.

Submit the code as a single *ZIP archive*. It must contain only the source code file named `manager.c`, your additional code needed to compile it (`*.c`, `*.h`), and a script named `build.sh` with your compilation commands. The code must be compliable without warnings, and stable in execution to get full marks for the assignment. Do not submit any executables, source code or libraries that are not created by you, nor additional files unnecessary during compilation.

The source code must be submitted by **13th Feb 2024, 23:59**.

Grading

This assignment will contribute **10%** to the overall course grade. All students are required to submit their solutions individually before the due date indicated in eLearn.

Late submissions will be graded one full letter grade down (i.e. from A- to B-) for each started period of 12 hours of delay.

Students may request for extensions should they provide valid formal reasons with documented evidence to justify their case. Such cases will be handled on a case-by-case basis. Request for extension after the deadline will be accepted only in extraordinary cases.

Your code will be tested against instances covering `run`, `stop`, `kill`, `resume`, `list`, and `exit` functionalities. Furthermore, you need to ensure that "ready" processes are scheduled to transition to "running" states in a way that follows the designated policy. These test instances make up **85%** of the total marks. Further **5%** is for meaningful error messages. Final **10%** is for programming style and documentation, e.g. in-line comments. Additional penalties for memory leaks, unstable execution, or compilation warnings may apply.

This is an individual assignment. While you are encouraged to discuss with your classmates, submitted codes must be your own work. No direct assistance from generative AI tools are allowed.

Honor code

SMU expects students to abide by the honor code and The SMU Student Code of Conduct. Specifically, students are expected to submit only their own work and not submit answers or code that have not been created by students themselves.