Department of Computer Science and Software Engineering

CITS2002 Systems Programming

Project 1 2020, see also: Project 1 clarifications

The goal of this project is to implement a program to simulate a pre-emptive process scheduler supporting inter-process communication using pipes.

Successful completion of the project will develop your understanding of some advanced features of the C99 programming language, and your understanding of how an operating system can manage processes and inter-process communication.

Project description

Download, understand, and then extend this C99 starting file: pipesim.c

Run a web-based sample solution: pipesim-sample

UNIX-based operating systems employ pre-emptive process scheduling policies to execute hundreds of processes 'simultaneously' on a single CPU. Process take turns to execute on the CPU until they explictly exit, perform an operation that would stall their execution, or until their rationed time on the CPU has elapsed.

Only a single process may be 'on' the CPU at one time, and this process is uniquely marked as being in the **Running** state. All other processes waiting for their turn on the CPU are in the **Ready** state, recorded in a first-in-first-out queue, unless they are marked as being in one of a number of blocked states because they recently requested a blocking operation.

All processes are uniquely identified by a positive integer value termed a *process-identifier*, or PID. When the system has finished booting (at TIME=0), only a single process exists (with PID=1) and is executing on the CPU. Like all (future) processes, this initial process may use the CPU to perform some computation or request actions of the operating system kernel by making *system-calls*.

System-calls

The system simulated in this project supports a limited number of *system-calls*. Processes can only make system-calls when executing on the CPU. These are described here, along with reference to the possible *execution states* of the processes.

compute()
A process may execute on the CPU for a requested number of microseconds by calling the *compute()* system-call. The process does not occupy (does not own) the CPU for the requested time, uninterrupted. Instead, its use of the CPU is interleaved with other processes in a *pre-emptive* manner.

For example, consider a computer system with a *scheduling-quantum* of 1000 microseconds. If a process requests to compute (on the CPU) for 3200 microseconds, it will first occupy the CPU for 1000 microseconds, leave the CPU, be marked as **Ready** (**Running**→**Ready**) while other process(es) execute, occupy the CPU for another 1000 microseconds (**Ready**→**Running**), and so on..., until its final turn on the CPU (for the remaining 200 microseconds). It will then leave the CPU, be marked as **Ready** (**Running**→**Ready**) ready to make its next system-call when it next runs on the CPU.

sleep()
A process may relinquish its use of the CPU by calling the *sleep()* system-call, indicating for how many microseconds it wishes to sleep. The process will leave the CPU, and be marked as **Sleeping** (**Running**→**Sleeping**). When (at least) the requested time has elapsed the process will be marked as **Ready** (**Sleeping**→**Ready**).

exit()
A process may request its own termination by calling the *exit()* system-call. When a process terminates, its resources (if any?) are deallocated, any pipes that it had opened for reading or writing are closed (see later), though its PID is not re-used (re-assigned) for future processes. The parent of the terminating process may be waiting for the child to terminate. See also: Project 1 clarifications of Sept 7th.

fork()
A process may create a new process by calling the *fork()* system-call. The process calling *fork()* is termed the *parent* process, and the newly created process is termed the *child* process. The new child process receives the next available PID, and is connected to any pipes created by its parent that don't yet have a 'reading end' (see later). Both the child and then the parent (in that order) are marked as **Ready** to run.

wait()
A parent process may request to be informed of the termination of one of its child processes by calling the *wait()* system-call. Until the specified child process terminates the parent process will be marked as **Waiting** (**Running**→**Waiting**). When the child process eventually terminates, the parent process will be marked as **Ready** (**Waiting**→**Ready**).

pipe()
A process may request a new inter-process communication buffer by calling the *pipe()* system-call. A *pipe* is a unidirectional in-memory array of bytes (of finite size). Pipes connect two processes - exactly one process may write data to a pipe, and exactly one other process may read data from it. Each process has a limited number of *pipe-descriptors* - non-negative integers which refer to either the 'writing end' or the 'reading end' of pipes to which they are connected. Processes use their value of the pipe-descriptor to identify which pipe they are writing to or reading from.

When a process creates a new pipe, it immediately becomes the 'writing end' of the pipe. The pipe-descriptor on which it should write its data is the value returned by calling *pipe()*. When a parent process forks a new child process, the new child process immediately becomes the 'reading end' of all of its parent's pipes that don't yet have a reader. When a pipe has both a writer and a reader, the two processes may communicate by writing to and reading from that pipe. The process calling *pipe()* will be marked as **Ready** (**Running**→**Ready**).

writepipe()
A process may write data to a pipe using the *writepipe()* system-call. Pipes may hold upto a fixed amount of data (typically 4KB).

A process attempting to write more data than will fit in the pipe will write some (possibly none) of its data to the pipe, and then block until some space becomes available in the pipe. A writing process will be marked as **Writing** (**Running**→**Writing**) and remain blocked until all bytes of its *writepipe()* request has been written to the pipe, after which it will be marked as **Ready** (**Writing**→**Ready**).

readpipe()
A process may read data from a pipe using the *readpipe()* system-call.

A reading process will be marked as **Reading** (**Running**→**Reading**) and remain blocked until all bytes of its *readpipe()* request has been read from the pipe, after which it will be marked as **Ready** (**Reading**→**Ready**).

Note: the implementation and operation of pipes required for this project is slightly different, and much simpler, than 'true' pipes in a UNIX-based operating system.

Eventfiles

An *eventfile* is a simple text file containing the historic record of the system-calls requested by the processes of a simple computer system. After the computer system has booted, only a single process (with PID=1) will be running. Thus the first line of every *eventfile* is a system-call request by PID=1, and no other processes will appear until that first process performs a *fork()*. The very last line of an *eventfile* will record the last *exit()* call, after which no processes will be running (and the system halts).

Each line of the file consists of a number of white-space separated words. The first word is always a positive PID, indicating which process is performing the action described on the remainder of the line. The second word on each line is always the name of a system-call requested by the process. Some lines will also have one or two additional words which further describe input paramete(s) for the system-call, or the value returned by the system-call. The supported system-calls are:

Meaning	Example from eventfile
	... previous lines ...
compute *input-parameter: microseconds-required-on-CPU*	12 compute 2400
sleep *input-parameter: microseconds-to-sleep*	12 sleep 5000
pipe *output-result: new-writing-pipe-descriptor*	12 pipe 4
fork *output-result: PID-of-new-child-process*	12 fork 18
readpipe *two-input-parameters: reading-pipe-descriptor number-of-bytes*	18 readpipe 4 400
writepipe *two-input-parameters: writing-pipe-descriptor number-of-bytes*	12 writepipe 4 1200
	18 readpipe 4 400
wait *input-parameter: PID-to-wait-for*	12 wait 18
	18 readpipe 4 400
exit	18 exit
	12 exit

The provided starting code parses the information in the *eventfiles*. You will need to store this information in your own data-structures and variables before you can commence the simulation. You may assume that the format of each *eventfile* is correct, and its data consistent, so you do not need to check for any errors in the *eventfile*.

Some sample eventfiles (and, of course, you should write and test with your own eventfiles):
ev0 ev1 ev2 ev3 ev4 ev5 ev6 ev7 ev8 ev9 ev10 ev11

Project requirements

- You are required to develop and test a program that determines (through simulated execution, not by attempting to find a formula) the total *time taken* (in microseconds) for all events in the *eventfile* to complete. Each eventfile commences (at TIME=0) with the first system-call made by the initial process (PID=1) and finishes with the last call to *exit()* by the last process. (at TIME=timetaken).

  The total *time taken* will incorporate the accumulated time taken by processes while executing (on the CPU), time taken switching processes from one state to another, time taken by running processes to write and read data to and from pipes, and any 'dead' time if all processes are sleeping.

  Your program can print out any additional information (debugging?) that you want - **just ensure that the *last line* is of the form** "timetaken 160000".

- Your project must be written in C99 in a single source-code file named **pipesim.c**
  This is the only file you should submit for marking.
  Your submission will be compiled with the C compiler arguments **-std=c99 -Wall -Werror**, and marks will not be awarded if your submission does not compile.

- Your program, named **pipesim**, should accept command-line arguments providing: the name of the *eventfile*; a positive integer representing the *scheduling time-quantum* (in microseconds); and a positive integer representing the *pipe buffer size* (in bytes).

```
prompt> ./pipesim eventfile timequantum-usecs pipesize-bytes
```

- You should start by reading, understanding, and then extending **the starting code**. You should not modify any of the provided constants in the file (they will be used during marking), but may add your own additional constants.

- You may use any functions from the standard C99 library but must not employ any 3rd-party code or libraries to complete your project. If in doubt, please ask.

Assessment

This project is INDIVIDUAL work.
The project deadline is **5PM Fri 11th September (end of week 7)**, and is worth **30% of your final mark** for CITS2002.

The project will be marked out of 40. 20 of the possible 40 marks will come from the correctness of your solution. The remaining 20 marks will come from your programming style, including your use of meaningful comments, well chosen identifier names, appropriate choice of basic data-structures and data-types, and appropriate choice of control-flow constructs.

During the marking, attention will obviously be given to the correctness of your solution. However, a correct and efficient solution should not be considered as the perfect, nor necessarily desirable, form of solution. Preference will be given to well presented, well documented solutions that use the appropriate features of the language to complete tasks in an easy to understand and easy to follow manner. That is, do not expect to receive full marks for your project simply because it works correctly. Remember, a computer program should not only convey a message to the computer, but also to other human programmers.

Your project will be marked on a computer running a standard version of Linux. No allowance will be made for a program that *"works at home"* but not on a Linux computer, so be sure that your code compiles and executes correctly on a Linux system before you submit it.

Submission requirements

1. The deadline for the project is **5PM Friday 11th September (end of week 7)**.

2. Your submission's C99 source file should each begin with the C99 block comment:

```
/*  CITS2002 Project 1 2020
    Name:                student-name
    Student number(s):   student-number
 */
```

3. **You must submit your project electronically using *cssubmit*.** You should submit a single C99 source-code file, named **pipesim.c**. You do not need to submit any additional testing scripts or files that you used while developing your project. The *cssubmit* facility will give you a receipt of your submission. You should print and retain this receipt in case of any dispute. Note also that the *cssubmit* facility does not archive submissions and will simply overwrite any previous submission with your latest submission.

4. This project is subject to UWA's Policy on Assessment - particularly §10.2 *Principles of submission and penalty for late submission*. In accordance with this policy, you may *discuss* with other students the general principles required to understand this project, but the work you submit must be the result of your own efforts. All projects will be compared using software that detects significant similarities between source code files. Students suspected of plagiarism will be interviewed and will be required to demonstrate their full understanding of their project submission.

Clarifications

Please post requests for clarification about any aspect of the project to *help2002* so that all students may remain equally informed.
Significant clarifications will be also added to the **project clarifications** webpage.

Good luck!

Chris McDonald.