THE UNIVERSITY OF WESTERN AUSTRALIA

Department of Computer Science and Software Engineering

CITS2002 Systems Programming

Project 1 2019. See also: Project 1 clarifications

The goal of this project is to implement a program to simulate a single-CPU, multi-device, pre-emptive process scheduler and to evaluate its effectiveness for varying job-mixes. 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.

Project Description

Modern operating systems maintain a lot of state information to support their operation. Some of this information can be periodically captured to provide a snapshot of a currently running system, or logged for later analysis. For example, a **tracefile** can contain a summary of the execution history of completed processes, and the information may be used to improve the execution of future processes.

In combination, the execution requirements of processes and the input/output (I/O) operations they perform, is termed the **job-mix**. This project will use the information in a *tracefile* to find the near-optimal scheduling of processes, in the belief that future *job-mixes* will be similar to that in a recent *tracefile*.

Our Operating System Model

Consider an operating system's 5-State Model of Process Execution, as introduced in Lecture 8.
New processes are admitted to the system and are immediately marked as Ready to run.
Each process executes, in turn, until it:

- completes its execution (at which time the process Exits),
- executes for a finite **time-quantum** (after which the process is marked as Ready and is queued until it can again Run), or
- requests some input or output (I/O) (at which time the process is marked as Blocked and queued until its I/O request is satisfied).

For this project we'll consider a simplified operating system in which only a single process occupies the single CPU at any one time. The CPU has a clock speed of 1GHz, enabling it to execute one-billion instructions per second (or 1000 instructions per microsecond, 1000000 instructions per millisecond). For this simplified project, we do not need to consider any RAM accesses.

It takes 5 microseconds to perform a *context-switch* - to move one process from **Running** → **Ready** (or → **Blocked**), and then to move another process from **Ready** → **Running**. No time is consumed deciding that the currently **Running** process can remaining **Running** (and start a new *time-quantum*).

The CPU is connected to a number of input/output (I/O) devices of differing speeds, using a single high-speed **data-bus**. Only a single process can use the *data-bus* at any one time, and it takes 5 microseconds for any process to first acquire the *data-bus*.

Only a single process can access each I/O device (and the *data-bus*) at any one time. If the *data-bus* is in use (data is still being transferred) and a second process also needs to access the *data-bus*, the second process must be queued until the current transfer is complete. When a data transfer completes, all waiting (queued) processes are consider to determine which process can next acquire the *data-bus*. If multiple processes are waiting to acquire the *data-bus*, the process that has been waiting the longest for the device with the highest **priority** will next acquire the *data-bus*. Thus, all processes waiting on higher priority devices are serviced before any processes that are waiting on lower priority devices.

The result is a need to support Multiple blocked queues, as introduced in Lecture 8.

Tracefiles

Consider the following lines from the beginning of text file termed a *tracefile*. We may assume that the format of each *tracefile* is correct, and its data consistent, so we do not need to check for errors in the *tracefile*.

```
device    usb2       60000000 bytes/sec
device    kb         10 bytes/sec
device    ssd        240000000 bytes/sec
device    hd         80000000 bytes/sec
device    wifi       6750000 bytes/sec
device    screen     200000 bytes/sec
reboot
```

Each line provides a device definition including its name and its data transfer rate (all transfer rates are measured in bytes/second). Devices with higher transfer rates have a higher *priority* and are serviced first. The final line indicates the end of all device definitions, and that the operating system commences execution setting the **system-time** to zero microseconds. All times in the *tracefile* are measured in microseconds.

The following lines of the same *tracefile* provide the execution summary of two processes:

```
process  1  200 {
  i/o      100     hd      1600
  i/o      110     usb2    1600
  i/o      180     hd      1000
  i/o      190     usb2    1000
  exit     400
}
process  2  480 {
  i/o      8000    screen  40
  exit     8005
}
```

- Process number 1 commenced execution at 200 microseconds (after the operating system rebooted). Then, after the process has executed for a total of 100 microseconds (occupying the CPU for a total of 100 microseconds), the process transfers 1600 bytes from the device named 'hd'. After a further 10 microseconds of execution time, the process transfers 1600 bytes of data to the device named 'usb2', and so on. We might assume that the process is copying a small file from 'hd' to 'usb2'. The process finally exits after it has executed for a total on 400 microseconds.
- Process 2 is computationally intensive - it commenced execution at 480 microseconds (after the operating system reboots), performs only a single I/O transfer (printing its answer), and then exits after 8005 microseconds of execution time.

As process 2 commenced execution before process 1 exits, they will need to share the CPU. When both processes exist, each process executes on the single CPU until it exits, until it performs some I/O, or until its finite *time-quantum* expires.

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

Evaluating Operating System Scheduling Effectiveness

There are a number of ways to measure the effectiveness of an operating system's process scheduling. There can be no single perfect measure of its effectiveness as it is very dependent on the *job-mix* to be scheduled, and the data transfer rates of the system's devices. For this project we will use the **total process completion time** to evaluate our process scheduling. With reference to our simple *tracefile* above, the *total process completion time* is the time from the commencement of the first process to the time that the final process exits.

Under our simple operating system model, the *time-quantum* may be varied to permit each process to occupy the CPU for different lengths of time. Varying the time-quantum will result in the same job-mix exhibiting different total process completion times. With a short time-quantum, a system running many processes will give each of those processes a frequent chance to 'make progress', and will appear very responsive. With a long time-quantum, computationally-intensive processes can perform a lot of their execution, but delay the execution of other processes waiting for the CPU, and may make the system appear sluggish.

Each job-mix (recorded in a *tracefile*) will have a *near optimal* time-quantum that minimises the total process completion time. If future job-mixes are similar to the one in a given *tracefile*, and we can determine a good time-quantum, then we can *tune* our process scheduling to match our anticipated job-mix.

Project requirements

You are required to develop and test a program that determines the *time-quantum* providing the best (shortest) *total process completion time* for a given *tracefile*.

Your program, named **besttq**, should accept command-line arguments providing the name of the *tracefile* and three integers representing times measured in microseconds. The program is only required to produce a single line of output (though you may wish to produce several more lines, as a form of debugging) reporting the best *time-quantum* found.

For example, the following command might produce the single line of output:

```
prompt> ./besttq  tracefile1  100  2000  100
best  1600  440800
```

The command reads and evaluates (simulates, 'executes') the contents of the *tracefile* several times, employing different *time-quanta*, to find which *time-quantum* results in the lowest *total process completion time*.

The command, above, considers the different *time-quantum* values of 100 microseconds, 200 microseconds, ... 2000 microseconds (just like a **for** loop in C99). The program has determined that the best *time-quantum* for the given *job-mix* is 1600 microseconds and which results in a *total process completion time* of 440800 microseconds.

Your program can print out any additional information (debugging?) that you want - just ensure that the *last line* is of the form "best 1600 440800".

You will be provided with a number of sample *tracefiles* and a sample solution, against which you can test your developing solution.

Assessment

The project is due **11:59PM Fri 13th September**, and is worth **20% of your final mark** for CITS2002.
It will be marked out of 40. The project may be completed **individually or in teams of two** (but not teams of three).

You are **strongly** encouraged to work with someone else - this will enable you to discuss your initial design, and to assist each other to develop and debug your joint solution. Work together - do not attempt to split the project into two equal parts, and then plan to meet near the deadline to join your parts together.

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 the computers in CSSE Lab 2.03, using the macOS environment. No allowance will be made for a program that *"works at home"* but not on CSSE Lab 2.03 computers, so be sure that your code compiles and executes correctly on these machines before you submit it.

Submission requirements

1. The deadline for the project is **11:59PM Friday 13th September (end of week 7)**.

2. Your submission will be compiled, run, and examined using the macOS platform on computers in CSSE Lab 2.03. Your submission must work as expected on this platform. While you may develop your project on other computers, excuses such as *"it worked at home, just not in the lab!"* will not be accepted.

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

   ```
   /* CITS2002 Project 1 2019
      Name(s):             student-name1 (, student-name2)
      Student number(s):   student-number-1 (, student-number-2)
    */
   ```

   If working as a team, only one team member should make the team's submission.

4. **You must submit your project electronically using *cssubmit*.** You should submit a single C99 source-code file, named **besttq.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.

5. 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.

Good luck!

Chris McDonald.