Operating Systems (COMP2006)

CURTIN UNIVERSITY
Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences

Assignment

Multiple-Processor Scheduling Simulation

Due Date: **4pm, Monday 6 May, 2019**

The goal of this simulation is to give you some experiences using POSIX Pthreads library functions for thread creations and synchronizations. You will learn how to solve the critical section problems using pthread_mutex_lock(), pthread_mutex_unlock(), pthread_cond_wait(), and pthread_cond_signal().

Write a program **scheduler** in C under Linux environment to simulate the operations of three Processor Scheduling Simulation. The programs for this scheduler should include the following features.

1) There is one **Ready-Queue** that contains a list of tasks to be serviced by three **CPU**s: **CPU-1**, **CPU-2**, and **CPU-3**. The **Ready-Queue** has a capacity for *m* tasks, and is initially empty.

2) All **CPU**s are waiting for the arrival of the tasks in **Ready-Queue**. The arrival of a task should interrupt any of the waiting **CPU**s, which will grab the task, and execute it.

3) Your scheduler includes a list of tasks, stored in file *task_file*. A task in the file is represented by

   task# cpu_burst

   The *task#* is a positive integer, and the *cpu_burst* is another positive integer (in second). Note that, each *task* may have different *cpu_burst*. Create yourself a *task_file* that contains 100 tasks with random *cpu_burst* (1 to 50).

4) Write a function **task()** that gets two tasks at a time from the *task_file* and puts it into the **Ready_Queue**. For each task placed in the queue, the **task()** function should write this activity into a file, **simulation_log**.

   task#: *cpu_burst*
   Arrival time: 13:42:51

   The Arrival time is the time the task is placed into **Ready-Queue** (actual time).

5) Write a function **cpu()** that simulates the operations of each of the three **CPU**s. When there is at least one task in **Ready-Queue**, one of the **CPU**s takes the task from **Ready-Queue**, and executes it for its entire *cpu_burst*. In other words, it is a non pre-emptive scheduler. Simulate this event, for example, using a **sleep** call, proportional to the length of *cpu_burst*.

6) Create three variables *num_tasks*, *total_waiting_time* and *total_turnaround_time* to be shared by the three **CPU**s. The three variables are initialized to 0.

7) When **CPU-1**, for example, takes one task from the queue, **CPU-1** should write the following information in **simulation_log**:

    Statistics for CPU-1:
    Task *#n*
    Arrival time: 13:42:55
    Service time: 13:42:57

    The Service time is the time the **CPU** picked up the task from the queue. Notice that the task's waiting time is its Service time minus Arrival time. **CPU-1** then increases the value of *num_tasks* by one, and *total_waiting_time* by the computed waiting time.

8) When **CPU-1**, for example, finishes with one task, **CPU-1** should write the following information in **simulation_log**:

    Statistics for CPU-1:
    Task *#n*
    Arrival time: 13:42:55
    Completion time: 13:42:59

    The Completion time is the time when **CPU-1** finished servicing the task *#n*. The Completion time is computed from the Service time + *cpu_burst*. Notice that the task's Turnaround time is its Completion time minus its Arrival time. **CPU-1** then increases the value of *total_turnaround_time* by the computed Turnaround time.

9) The **task**() function terminates when all tasks in *task_file* have been placed in **Ready-Queue**. The following information should be written in **simulation_log**:

    Number of tasks put into Ready-Queue: #of tasks
    Terminate at time: *current time*

    The *current time* is the time it terminates, e.g., 13:52:55.

10) Each **CPU** terminates when there is no more task (NOT when **Ready-Queue** is empty). Each terminating **CPU**, e.g., **CPU-1** should write the following information into **simulation_log**:

    CPU-1 terminates after servicing *x* tasks

where *x* is the total number of tasks the CPU has served.

11) Finally, after all **CPU**s and **task** have terminated, the **scheduler** should write the following information to **simulation_log**:

    Number of tasks: #of tasks
    Average waiting time: *w* seconds
    Average turn around time: *t* seconds

    The Number of tasks is the total tasks serviced by the three **CPU**s, i.e, the value of *num_tasks*. The Average waiting time, and the Average turn around time are computed by dividing *total_waiting_time* and *total_turnaround_time* with *num_tasks*, respectively.

    Note, the assignment **does not require high degree of precision for the time**.

Implementation (80%)

1. Your **scheduler** creates a thread **task** that runs the **task()** function, and three threads **CPU-1**, **CPU-2**, and **CPU-3** each runs the **cpu()** function. Each **CPU** thread blocks when **Ready-Queue** is empty, and **task** thread blocks when the queue is full.

2. Create a First-In-First-Out **Ready-Queue** to be shared by threads **task, CPU-1**, **CPU-2**, and **CPU-3.** You have to synchronize the four threads when accessing the **Ready-Queue**. In essence, this is the bounded buffer producer-consumer problem.

3. Use pthread mutual exclusion functions, pthread_mutex_lock(), pthread_mutex_unlock(), pthread_cond_wait(), and pthread_cond_signal() to solve the critical section and synchronization problems in the **scheduler**. Make sure you consider all possible race conditions in the **scheduler**.

4. You have to describe / discuss in detail each of the variables, including its data structure, the threads that access them, and how mutual exclusion is achieved.

5. Remember to clean up all resources created in your program.

6. To test for the correctness of your program, you should run the program as follows:

    scheduler task_file *m*

    Set ***m*** to a value between 1 and 10.

Instruction for submission

1. Assignment submission is **compulsory**. Students will be penalized by a deduction of ten percent per calendar day for a late submission. **An assessment more than seven calendar days overdue will not be marked and will receive a mark of 0**.

2. You must (i) submit a hard copy of your assignment report, (ii) submit the soft copy of the report to the unit Blackboard (**in one zip file**), and (iii) put your program files e.g., scheduler.c, makefile, and other files, such as test input, in your home directory, under a directory named **OS/assignment**.

3. Your assignment report should include:

    - A signed cover page that includes the words "Operating Systems Assignment", your name in the form: family, other names, and a declaration stating the originality of the submitted work, that it is your own work, etc. Your name should be as recorded in the student database.
    - Software solution of your assignment that includes (i) all source code for the programs with proper in-line and header documentation. Use proper indentation so that your code can be easily read. Make sure that you use meaningful variable names, and delete all unnecessary comments that you created while debugging your program; and (ii) readme file that, among others, explains how to compile your program and how to run the program.
    - Detailed discussion on how any mutual exclusion is achieved and what threads access the shared resources.
    - Description of any cases for which your program is not working correctly or how you test your program that make you believe it works perfectly.
    - Sample inputs and outputs from your running programs.

    Your report will be assessed (worth 20% of the overall assignment mark).

4. Due dates and other arrangements may only be altered with the consent of the majority of the students enrolled in the unit and with the consent of the lecturer.

5. Demo requirements:

    - You may be required to demonstrate your program and/or sit a quiz during workshop sessions (to be announced).
    - For demo, you MUST keep the source code of your programs in your home directory, and the source code MUST be that submitted. The programs should run on any machine in the department labs.

    Failure to meet these requirements may result in the assignment not being marked