SAINT MARY'S UNIVERSITY SINCE 1802 | FACULTY OF SCIENCE

CSCI 3431.1 Fall, 2017 – Assignment 2

Operating Systems

Due Date: Friday, October 20, 2017
Closing Date: Sunday, October 22, 2017

Deadline Policy

There will be a due date and a closing date for each assignment

- If you submit your work after the due date (but before the closing date), a *late submission* is applied to your overall assignments submission.
- You are allowed to use a *late submission* for up to 2 assignments throughout the semester without being penalized. After that, your assignment grade will be penalized by 5%.
- You will not be able to submit or receive credit for your work after the closing date.

Submission

Once you are done with your assignment, make sure you do the following before the assignment due date:

- Prepare a report containing the answers to the assignment questions.
- Make sure to include your Name and A# as indicated in the last page.
- Create one PDF file for your report.
- Combine your source code and PDF report into one zip file.
- Name your file following this convention: CSCI3431-Assignment2- <*StudentID*>, where *StudentID* is your Banner ID number starting with A.
- Submit your PDF file via Brightspace at https://smu.brightspace.com before the deadline.
- Your programs will be compiled and tested on the CS machine.
- Do not submit a program that either won't compile or won't run. Instead of debugging your code, the marker/I will assign **Zero** credit for your submission.

Assignment Description

The assignment consists of two parts: practice exercises and a programming exercise.

0.1 Practice Exercises

1. **[2 points]** why is it important for a scheduler to distinguish I/O-bound processes from CPU-bound processes?

2. **[8 points]** Consider the following set of processes, with the length of the CPU burst time given in milliseconds:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

   Assume the processes arrive in the following order: P1, P2, P3, P4, P5.

   a) [1.5 points] Draw three Gantt charts to illustrate the execution of these processes using FCFS, SJF, and RR (with quantum= 1)

   b) [3 points] What is the turnaround time of each process for each of the scheduling algorithms?

   c) [3 points] What is the waiting time of each process for each of the scheduling algorithms?

   d) [0.5 points] Which of the algorithms results in the minimum average waiting time (over all processes)?

3. **[2 points]** Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system?

4. **[6 points]** In addition to general-purpose server software, a number of classes of applications benefit directly from the ability to scale throughput with the number of threads and/cores. Give at least 3 classes of applications with examples.

5. **[4 points]** Consider you have the following program:

```
1.  boolean blocked [];
2.          int turn;
3.          void P(int id)
4.          {
5.                  while (true){
6.                          blocked[id] = true;
7.                          while (turn!=id){
8.                                  while (blocked[1-id]) ;//do nothing
9.                                  turn = id;
10.                         }
11.                         //critical section
12.                         blocked[id] = false;
13.                         //remainder section
14.                 }
15.         }
16.
17.         void main(){
18.                 blocked[0] = false;
19.                 blocked[1] = false;
20.                 turn = 0;
            beginProcess(P(0), P(1));
21.
22.         }
```

Assume beginProcess(P(0), P(1)) does the following: suspend the execution of the main program; initiate concurrent execution of P(0), P(1); when P(0) and P(1) terminate, resume the main program.

This program represents a software solution to the mutual exclusion problem for two processes proposed in 1966 by Hymann, H. in a paper submitted to the *Communications of the ACM*. Find a counterexample that demonstrates that this solution is incorrect. Notice how it is easy to get fooled on this one!

6. **[8 points]** Consider a sharable resource with the following characteristics:
   1. As long as there are fewer than 3 processes using the resource, ne processes can start using it right away.
   2. Once there are 3 processes using the resource, all three must leave before any new process can begin using it.

Counters are needed to keep track of how many processes are waiting and active, and these counters are themselves shared resources that must be protected with mutual exclusion.

```
//share variables: semaphores, counters, and state information
Semaphore mutex = new Semaphore (1);
Semaphore block = new Semaphore (0);
int active = 0, waiting = 0;
boolean mustWait = false;
```

The following program offers a solution that appears to do everything right.

```
/* entry section */
mutex.acquire();          // enter the critical section
if (mustWait) {           // if there are (or were) 3, then
    ++waiting;            // we must wait, but also
    mutex.release();      // we must leave the critical section first
    block.acquire();      // wait for all current users to depart
}else{
    ++active;             // update active count
}

    mustWait = active == 3;      // record if the count reached 3
    mutex.release();             // leave mutual exclusion


/* critical section */

mutex.acquire();          //enter the critical section
--active;                 //update the active count
if (active == 0){         //check if this is the last one to leave
    int n;
    if (waiting <3) n = waiting;
    else n = 3;                          //see how many processes to unblock
    waiting -= n;                        //deduct this number from waiting count
    active = n;                          //set active to this number
    while (n>0){                         //now unblock the processes
           block.release();              //one at a time
           --n;
    }
    mustWait= active ==3;                //record if the count is 3
}
mutex.release();                         //Leave the critical section

/* remainder section */
```

a. [4 points] Explain how this program works and why it is correct.
b. [2 points] This solution does not completely prevent newly arriving processes from cutting in line but it does make it less likely. Give an example of cutting in line.
c. [2 points] This program is an example of a general design pattern that is a uniform way to implement solutions to many concurrency problems using semaphores. It has been referred to as the ***I"ll Do It For You*** pattern. Describe the pattern.

0.2 Programming Exercise

The objective of this exercise is to help you practice using threads.

A **Sudoku** puzzle uses a 9 × 9 grid in which each column and row, as well as each of the nine 3 × 3 subgrids, must contain all of the digits 1 · · · 9. The grid below presents an example of a valid Sudoku puzzle.

Your task consists of designing a multithreaded JAVA application that determines whether the solution to a Sudoku puzzle is valid. There are several different ways of multithreading this application. One suggested strategy is to create threads that check the following criteria:

- ✓ A thread to check that each column contains the digits 1 through 9
- ✓ A thread to check that each row contains the digits 1 through 9
- ✓ Nine threads to check that each of the 3 × 3 subgrids contains the digits 1 through 9

This would result in a total of eleven separate threads for validating a Sudoku puzzle. However, you are welcome to create even more threads for this application.

6	2	4	5	3	9	1	8	7
5	1	9	7	2	8	6	3	4
8	3	7	6	1	4	2	9	5
1	4	3	8	6	5	7	2	9
9	5	8	2	4	7	3	6	1
7	6	2	3	9	1	4	5	8
3	7	1	9	5	6	8	4	2
4	9	6	1	8	2	5	7	3
2	8	5	4	7	3	9	1	6

Passing Parameters to Each Thread

The parent thread will create the worker threads, passing each worker the location that it must check in the Sudoku grid. This step will require passing several parameters to each thread. For example, you will need to pass the row and column where a thread must begin validating.

Returning Results to the Parent Thread

Each worker thread is assigned the task of determining the validity of a particular region of the Sudoku puzzle. Once a worker has performed this check, it must pass its results back to the parent. One good way to handle this is to create an array of integer values that is visible to each thread. The ith index in this array corresponds to the ith worker thread. If a worker sets its corresponding value to 1, it is indicating that its region of the Sudoku puzzle is valid. A value of 0 would indicate otherwise. When all worker threads have completed, the parent thread checks each entry in the result array to determine if the Sudoku puzzle is valid.

Testing Your Application

You can test your application by loading your Sudoku puzzle from a .txt file assuming the values are tab separated. You can use Sudoku1.txt and Sudoku2.txt available on brightspace.

Self-evaluation Please answer the following questions:

1. **[1 points]** Were you able to complete this assignment? What grade are you expecting? Please justify.

2. **[2 points]** Describe 2-3 challenges you faced while completing this assignment. How did you tackle those challenges?

3. **[2 points]** Provide a break down for the activities/milestones for this assignment. Give an estimate of hours spent on each activity. Try to be honest!

Date	Activities	Hours	Outcome

Hints and Suggestions

- ✓ Start early!
- ✓ Document your work properly.
- ✓ Backup your work frequently. It's possible (and most likely) you go try a new feature and your program crashes!

Academic Integrity

You are required to demonstrate academic integrity in all of the work that you do. The University provides policies and procedures that every member of the university community is required to follow to ensure academic integrity. Unless stated otherwise, it is expected that all the work you submit for this course, is your OWN work.

Lack of knowledge of the academic integrity policy is not a reasonable explanation for a violation. You are encouraged to consult the Academic Integrity and Student Code of Conduct sections of the Academic Regulations in the Academic Calendar, in order to be well informed on the consequences of dishonest behavior. Please visit the links below for more information.

Evaluation Scheme

<table>
<tr><td colspan="2">Student Name:</td></tr>
<tr><td colspan="2">Student ID:</td></tr>
<tr><td>Total Points</td><td>/130</td></tr>
<tr><td>Practice Exercises
<table>
<tr><td>Question</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr>
<tr><td>Points</td><td>2</td><td>8</td><td>2</td><td>6</td><td>4</td><td>8</td></tr>
<tr><td>Score</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table></td><td>/30</td></tr>
<tr><td>Programming Exercise</td><td>/95</td></tr>
<tr><td>Running the Sudoku
✓ Your sukoku program runs successfully</td><td>/15</td></tr>
<tr><td>Creating Threads
✓ The tasks are divided properly among worker threads</td><td>/20</td></tr>
<tr><td>Passing Parameters to Threads
✓ The parameters needed for threads are passed properly</td><td>/20</td></tr>
<tr><td>Returning Results to the Parent Thread
✓ The child threads correctly pass the results to the parent thread
✓ The verification of the puzzle is sound</td><td>/20</td></tr>
<tr><td>Handling Errors and Exceptions
Code deals with exceptions in an appropriate manner.</td><td>/5</td></tr>
<tr><td>Documentation
Code shows good indentation, meaningful variable names, modularity, and helpful comments.</td><td>/5</td></tr>
<tr><td>Testing
Readme file contains a script showing all the test cases performed along with screenshots.
Tests include sudoku1.txt and sudoku2.txt and self-prepared sudokus</td><td>/10</td></tr>
<tr><td>Self-evaluation questions</td><td>/5</td></tr>
</table>