SAINT MARY'S UNIVERSITY SINCE 1802 | FACULTY OF SCIENCE

CSCI 3431.1 Fall, 2017 – Assignment 3

Operating Systems

Due Date: Friday, November 10, 2017
Closing Date: Sunday, November 12, 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.
- Fill out your Name and A# as indicated in the last page and attach this page to your report.
- 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-Assignment3-*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 Unix 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]** A distributed system using mailboxes has two IPC primitives, send and receive. The receive primitive specifies a process to receive from and blocks if no message from that process is available, even though messages may be waiting from other processes. There are no shared resources, but processes need to communicate frequently about other matters. Is deadlock possible? Discuss you answer.

2. **[3 points]** Consider the following state of a system with five processes, P0, P1, P2, P3, and P4, and four types of resources A-D.

	Allocation	*Max*	*Available*
	ABCD	ABCD	ABCD
P0	0012	0012	1520
P1	1000	1750	
P2	1354	2356	
P3	0632	0652	
P4	0014	0656	

   a. What is the content of the matrix *Need*?

   b. Is the process in a safe state?

   c. If a request from process P1 arrives for (0,4,2,0), can the request be granted immediately?

3. **[3 points]** Explain the difference between deadlock, livelock, and starvation.

4. **[4 points]** The four conditions (mutual exclusion, hold and wait, no preemption and circular wait) are necessary for a resource deadlock to occur.

   a. [3 points] Give an example to show that these conditions are not sufficient for a resource deadlock to occur.

   b. [1 points] When are these conditions sufficient for a resource deadlock to occur?

5. **[3 points]** In order to control network traffic, a router A, periodically sends a message to its neighbor, B, telling it to adjust the number of packets that it can handle. At some point, Router A is flooded with traffic and sends B a message asking to stop sending packets. It does this by specifying that the number of bytes B may sent to A is 0 (A's window size = 0). As traffic surges decrease, A sends a new message, telling B to restart transmission. It does this by increasing the window size from 0 to a positive number. That message is lost. As described, neither side will ever transmit.

   a. [1 point] Can you consider the situation as a deadlock?

   b. [2 points] If yes, what type of deadlock is this? If no, please justify how network traffic resumes between A and B.

6. **[8 points]** A spooling system consists of an input process ***I***, a user process ***P***, and an output process ***O*** connected by two buffers. The processes exchange data in blocks of equal size. These blocks are buffered on a disk using floating boundary between the input and the output buffers, depending on the speed of the processes. Let ***max*** denote the maximum number of blocks on disk***, i*** denotes the number of input blocks on disk, and ***o*** denotes the number of output blocks on disk. The communication primitives used ensure that the following resource constraint is satisfied***: i + o <= max.***

   The following is known about the processes:

   - As long as the environment supplies data, process ***I*** will eventually input it to the disk (provided disk space is available)

   - As long as input is available on the disk, process ***P*** will eventually consume it and output a finite amount of data on the disk for each block input (provided disk space becomes available)

   - As long as output is available on the disk, process ***O*** will eventually consume it.

a) [4 points] Show that this system can become deadlocked.

b) [4 points] Suggest additional resource constraint that will prevent the deadlock, but still permit the boundary between input and output buffers to vary in accordance with the present needs of the processes.

7. **[7 points]** Assume the following code snippets for three processes P_1, P_2 and P_3. All processes are competing for 6 resources labeled S1…S6.

```
void P1(){
   while (1){
        get(S1);
        get(S2);
        get(S3);
/* Critical Section:
Use S1, S2, S3 */
        release(S1);
        release(S2);
        release(S3);
  }
}
```

```
void P2(){
   while (1){
        get(S4);
        get(S5);
        get(S2);
/* Critical Section:
Use S4, S5, S2 */
        release(S4);
        release(S5);
        release(S2);
 }
}
```

```
void P3(){
  while (1){
        get(S3);
        get(S6);
        get(S4);
/* Critical Section:
Use S3, S6, S4 */
        release(S3);
        release(S6);
        release(S4);
  }
}
```

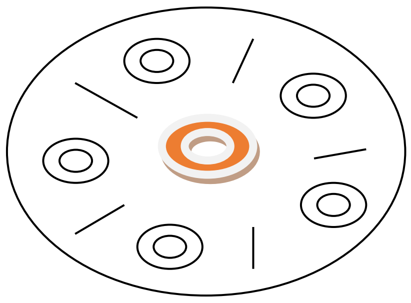
a. [3 points] Using a resource allocation graph (see Fig. 7.1-7.3), show the possibility of a deadlock in this implementation.

b. [4 points] Modify the order of some of the get requests to prevent the possibility of any deadlock. You cannot move requests across procedures, only change the order inside each procedure. Use a resource allocation graph to justify your answer.

0.2 Programming Exercise

The objective of this exercise is to practice implementing a concurrent program while ensuring mutual exclusion requirements using Monitors. You will implement a solution to a classic synchronization problem.

Dining Philosopher Problem



Five philosophers live in a house where a table is set for them. Philosophers live to think and eat and their favorite food is spaghetti! They enjoy eating their spaghetti using **two forks**. Philosophers gather around a shared table where they have 5 plates (one for each philosopher), 5 forks (one on each side of the plate), and a large bowl of spaghetti as shown in the figure. When a philosopher wishes to eat, he/she grabs the two forks (from either side of the plate), takes and eats some spaghetti.

Your challenge:

Design an algorithm that will allow philosophers to eat and enjoy their spaghetti. The algorithm must satisfy:

- **Mutual Exclusion**: No two philosophers can use the same fork at the same time.
- **Allow Progress**: No philosopher must starve to death.

You solution:

You are provided with a Dining Server interface (`DiningServer.java`). Implement your DiningServer using a **Monitor** and use the `DiningPhilosophersDriver.java` to test your implementation.

- ✓ You will need to record and display each philosopher's state. A philosopher is either Thinking, Eating, or is Hungry.
- ✓ To handle synchronization problems:
    - o You can use a reentrant lock (available through

`Java.util.concurrent.locks` api) which allows a thread to reacquire the same lock multiple times without any issue).

- You will need a condition variable for each fork. These condition variables are used to let the philosopher wait for the availability of a fork.

✓ The monitor should implement the two procedures:

- `takeforks(int pnum)` : used by a philosopher to get left and right forks. If either forks is not available, the philosopher's process is queued on the appropriate condition variable.
- `returnforks(int pnum)`**:** used to release two forks and make them available.

What to hand in:

- DiningServerImpl.java : Monitor implementation of the dining philosophers' problem.
- Philosopher.java : class representing each philosopher's thread. A philosopher thread alternates between eating and thinking.
- Readme: including a few screenshots showing sample runs of your program. Sample run shown below:

```
…
philosopher 2 is eating.
philosopher 0 is eating.
philosopher 1 is hungry.
philosopher 4 is hungry.
philosopher 0 is done eating.
philosopher 0 is thinking.
philosopher 4 is eating.
philosopher 2 is done eating.
philosopher 1 is eating.
philosopher 2 is thinking.
philosopher 3 is hungry.
philosopher 1 is done eating.
philosopher 1 is thinking.
philosopher 0 is hungry.
philosopher 4 is done eating.
philosopher 3 is eating.
philosopher 4 is thinking.
philosopher 0 is eating.
….
```

Self-evaluation

Please answer the following questions:

1. [**3 points**] Is your solution to the dining philosophers' problem deadlock free? Justify.
2. **[1 points]** Were you able to complete this assignment? What grade are you expecting? Justify.
3. **[2 points]** Describe 2-3 challenges you faced while completing this assignment. How did you tackle those challenges?
4. **[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!

References

https://docs.oracle.com/javase/tutorial/essential/concurrency/newlocks.html

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. Links: http://www.smu.ca/academics/academic-calendar.html http://www.smu.ca/academics/calendar/academic-integrity.html

Evaluation Scheme

Student Name:	
Student ID:	
Total Points	**/130**
Practice Exercises	**/30**

Question	1	2	3	4	5	6	7
Points	2	4	3	4	3	8	7
Score							

Programming Exercise	**/92**
Program runs properly • Program runs successfully • Program keeps displaying information regarding the state of the philosophers • No exceptions or blocked threads • Once started, program keeps running until terminated explicitly	/12
Mutual Exclusion • No two philosophers can grab the same fork	/20
Progress • The algorithm is designed in such a way to enable progress for all threads • Deadlock free • Starvation free	/20
Monitor • Lock: proper use of lock • Condition Variable: appropriate condition variables are used	/20
Handling Errors and Exceptions Code deals with exceptions in an appropriate manner.	/5
Documentation Code shows good indentation, meaningful variable names, modularity, and helpful comments.	/5
Testing Readme file showing test cases performed along with screenshots.	/10
Self-evaluation questions	**/8**