Department Computer Science and Software Engineering
Concordia University

COMP 346 – Fall 2020
Assignment 3

Due date and time: Friday November 27, 2020 by midnight

Written Questions (50 marks):

Note:

1) You must submit the answers to all the questions below. However, only one or more questions, possibly chosen at random, will be corrected and will be evaluated to the full 50 marks.

2) TAs will not be able to answer **Theory Assignment** related queries. Please get in touch with Professor for any queries related to **Theory Assignment**.

3) Theory assignment to be **completed individually**.

Question # 1

Answer the following questions:

i. (a) What are relocatable programs? (b) What makes a program relocatable? (c) From the OS memory management context, why programs (processes) need to be relocatable?
ii. What is (are) the advantage(s) and/or disadvantage(s) of small versus big page sizes?
iii. What is (are) the advantage(s) of paging over segmentation?
iv. What is (are) the advantage(s) of segmentation over paging?

Explain your answers.

Question # 2

Consider the below implementations of a semaphore's *wait* and *signal* operations:

```
wait () {
     disable interrupts;
     sem.value--;
     if (sem.value < 0) {
          save_state (current) ; // current process
          State[current] = Blocked; //A gets blocked
          Enqueue(current, sem.queue);
          current = select_from_ready_queue();
          State[current] = Running;
          restore_state (current); //B starts running
     }
    Enable interrupts;
 }
```

```
signal(){
   disable interrupts;
    sem.value++;
    if (sem.value <= 0){
      k = Dequeue(sem.queue);
      State[k] = Ready;
      Enqueue (k, ReadyQueue);
     }
   Enable interrupts;
}
```

a) What are the critical sections inside the *wait* and *signal* operations which are protected by disabling and enabling of interrupts?

b) Give example of a specific execution scenario for the above code leading to inconsistency if the critical sections inside implementation of *wait()* and *signal()* are not protected (by disabling of interrupts

c) Suppose that process A calling semaphore *wait()* gets blocked and another process B is selected to run (refer to the above code). Since interrupts are enabled only at the completion of the *wait* operation, will B start executing with the interrupts disabled? Explain your answer.

<u>Question # 3</u>

Consider a demand-paged system where the page table for each process resides in main memory. In addition, there is a fast associative memory (also known as TLB which stands for Translation Look-aside Buffer) to speed up the translation process. Each single memory access takes 1 microsecond while each TLB access takes 0.2 microseconds. Assume that 2% of the page requests lead to page faults, while 98% are hits. On the average, page fault time is 20 milliseconds (includes everything: TLB/memory/disc access time and transfer, and any context switch overhead). Out of the 98% page hits, 80 % of the accesses are found in the TLB and the rest, 20%, are TLB misses. Calculate the effective memory access time for the system.

<u>Question # 4</u>

Consider the page reference string R={0, 1, 2, 0, 1, 2, 0, 1, 2, 3, 6, 7, 6, 7, 0, 1, 2, 3, 4} for a given process.

(a) Show the memory representation of the pages using the LRU algorithm and an allocation of 3 frames. How many page faults are there?

(b) Show the memory representation of the pages using the Belady Optimal algorithm and an allocation of 3 frames. How many page faults are there?

(C) Show the memory representation of the pages using the working set model with a window size $\Delta=3$ (Δ indicates the maximum number allowed for a page to be in memory before being replaced; i.e. if a page is not used for 3 consecutive times, then it must either be used/demanded next, or it has to be removed). How many page faults are there?

<u>Question # 5</u>

Consider a system that would implement the page table on the CPU if feasible.

(a) Give an advantage of this strategy.

(b) Give a disadvantage of this strategy.

<u>Question # 6</u>

Explain (i) an advantage and (ii) a disadvantage that a global page replacement algorithm has over a local page replacement algorithm.

<u>Question # 7</u>

Consider a system that adjusts the degree of multiprogramming by monitoring the mean time between page faults (i.e. T_{pf}) and the mean time to service a page fault (i.e. T_{fs}). Describe the performance of the paging system in terms of the degree of multiprogramming when (i) T_{pf} is greater than T_{fs}, (ii) T_{pf} is less than T_{fs} and (iii) T_{pf} is equal to T_{fs}.

Question # 8

Some systems automatically open a file when it is referenced for the first time and close the file when the job terminates. Discuss the advantages and disadvantages of this scheme as compared to the more traditional one, where the user has to open and close the file explicitly.

Question # 9

a) What is the difference between preemptive and non-preemptive scheduling? Why is strict non-preemptive scheduling unlikely to be used in a computer system that provides both batch and timesharing service?

b) What is the trade-off used to select the quantum size, say, in pure Round-Robin scheduling?

Question # 10

What advantage is there in having different values of the scheduling quantum on different levels of a multilevel feedback queuing system? Your answer should consider all aspects such as fairness, starvation, efficiency, etc.

Question # 11

Consider the following set of prioritized processes, where a smaller priority value represents a higher priority.

Process	Service Time	Priority
0	20	3
1	15	1
2	21	3
3	7	5
4	12	2

Assume that all processes arrived at the same time, however they are inserted in the ready list in the order indicated in the above table.

a) Draw Gantt charts for the execution scenarios assuming:
- FCFS scheduling
- Non-preemptive SJF scheduling
- Non-preemptive priority scheduling
- Pure Round-Robin scheduling with the quantum = 3

b) What is the waiting time of each process in each case?
c) What is the response time of each process in each case?
d) What is the turn-around time of each process in each case?

Submission: Create a .zip file by name t*a3_studentID.zip* containing all the solutions, where ta3 is the number of the assignment and *studentID* is your student ID number. Upload the .zip file on moodle under *TA3.*

Programming assignment 3 (50 Marks)

Late Submission:	No late submission
Teams:	The assignment can be done individually or in teams of 2. Submit only one assignment per team.
Purpose:	The purpose of this assignment is to Implement the dining philosopher's problem using a monitor for synchronization.

1 Source Code

There are five files that come with the assignment. A soft copy of the code is available to download from the course web site. This time the source code is barely implemented (though compiles and runs). You are to complete its implementation.

1.1 File Checklist

Files distributed with the assignment requirements:

common/BaseThread.java
unchanged DiningPhilosophers.java
the main() Philosopher.java - extends from BaseThread
Monitor.java - the monitor for the system
Makefile - take a look

2 Background

This assignment is a slight extension of the classical problem of synchronization – the Dining Philosophers problem. You are going to solve it using the Monitor synchronization construct built on top of Java's synchronization primitives. The extension refers to the fact that sometimes philosophers would like to talk, but only one (any) philosopher can be talking at a time while they are not eating. If you need help, consult the references at the bottom.

3 Tasks

Make sure you put comments for every task that involves coding to the changes that you've made. This will be considered in the grading process.

Task 1 (20%): The Philosopher Class

Complete the implementation of the **Philosopher** class, that is all its methods according to the comments in the code. Specifically, ***eat()***, ***think()***, **talk()**, and **run()** methods have to be implemented entirely. Non-mandatory hints are provided within the code.

Task 2 (60%): The Monitor

Implement the **Monitor** class for the problem. Make sure it is correct, deadlock- and starvation-free implementation that uses Java's synchronization primitives, such as ***wait()*** and ***notifyAll()***; no use of Semaphore objects is allowed. Implement the four methods of the Monitor class; specifically, ***pickUp()***, ***putDown()***, ***requestTalk()***, and ***endTalk()***. Add as many member variables and methods to monitor the conditions outlined below as needed:

1. A philosopher is allowed to pickup the chopsticks if they are both available. That implies having states of each philosopher as presented in your book. You might want to consider the order in which to pick the chopsticks up.

2. If a given philosopher has decided to make a statement, they can only do so if no one else is talking at the moment. The philosopher wishing to make the statement has to wait in that case.

Task 3 (20%): Variable Number of Philosophers

Make the application to accept a positive integer number from the command line, and spawn exactly that number of philosophers instead of the default one. If there are no command line arguments, the given default should be used. If the argument is not a positive integer, report this fact to the user, print the usage information as in the example below:

```
% java DiningPhilosophers -7.a
"-7.a" is not a positive decimal integer

Usage: java DiningPhilosophers [NUMBER_OF_PHILOSOPHERS]
%
```

Use Integer.parseInt() method to extract an int value from a character string.
Test your implementation with varied number of philosophers. Submit your output.

- **Submission.**
  - Create one zip file, containing the necessary files (.java, .txt and test cases). If the assignment is done individually, your file should be called p*a3_studentID*, where pa3 is the number of the assignment and *studentID* is your student ID number. If the work is done in a team of 2 people, the zip file should be called p*a3_studentID1_studentID2* where *studentID1* and *studentID2* are the student ID numbers of each student.
  - Upload your zip file on moodle under the *PA3*.

4 References

1. Java API: http://java.sun.com/j2se/1.3/docs/api/
2. http://java.sun.com/docs/books/tutorial/essential/TOC.html#threads