**CS 471-001: Operating Systems**
**Spring 2017**

**Homework 2**
**Due date: Friday, March 10, 11:59 pm**

**Grading, submission and late policy:**

- You are expected to complete this homework **on your own** (not with a partner or in a group)
- The homework accounts for **4 %** of your final grade
- Standard late policy applies - Late penalty for this homework will be 15% for each day. Submissions that are late by 3 days or more will not be accepted - You will submit your assignment via Blackboard

**Part I: Multiple choice - circle the correct answer [20 points]**

1. If a process P executes the signal operation on semaphore S
   a. It moves to the waiting state if the integer value associated with S is 0
   b. It terminates without execution
   c. A process Q which was blocked on S will be moved to the ready state
   d. It decrements the value associated with S in atomic fashion
2. The following is a part of a monitor construct
   a. Condition variable
   b. Shared data variables
   c. Shared functions
   d. All of the above
3. Busy waiting is used in the following solution to critical section problem
   a. TestAndSet (TAS) instruction
   b. Semaphore
   c. Monitor with condition variables
   d. All of the above
4. The different between monitor condition variable and semaphore is that
   a. Semaphores are always initialized to 1 but not the condition variables
   b. A semaphore has a queue of waiting/blocked processes but a condition variable doesn't
   c. Semaphore has a value associated with it but condition variable doesn't
   d. No difference, they are the same
5. Which of the following is not true for spinlocks?
   a. They are useful when critical section is small

b. Because of busy waiting, they are not useful in multi-processor systems
c. They involve busy waiting which can result in performance issues
d. They can be implemented through TestAndSet (TAS) type instructions

**6.** Round-Robin algorithm is particularly useful for time-sharing (interactive) systems, because it provides a good:
a. Response time
b. Waiting time
c. Turnaround time
d. CPU utilization

**7.** In each of the following, you will be given a hypothetical sequence of states that a process enters during its lifetime. Which one is possible in a system with preemptive scheduling but not possible with non-preemptive scheduling?
a. New -> Ready -> Running -> Waiting -> Ready-> Running ->Terminated
b. New -> Ready -> Running -> Terminated
c. New -> Ready -> Running -> Ready -> Running -> Terminated
d. New -> Ready -> Running -> Waiting -> Running -> Terminated

**8.** Which of the following scheduling policy results in minimum average waiting time when processes arrive at different times
a. First-Come-First-Served (FCFS)
b. Shortest-Job-First (SJF)
c. Round-Robin (RR)
d. Shortest-Remaining-Time-First (SRTF)

**9.** Designing multilevel feedback queue scheduling policy requires
a. Determining when to promote or demote a queue
b. Determining how many queues to use
c. Deciding which scheduling policy to use for each queue
d. All of the above

**10.** If all processes in a system are less than 5 time units, a Round Robin scheduler with time quantum of size 10 is similar to?
a. Shortest-Job-First (non-preemptive)
b. Shortest-Remaining-Time-First
c. First-Come-First-Served
d. Priority scheduling

**Part II: True or False - mark the following as True or False [20 points]**

**11.** When a process executes a signal operation on a condition variable C within a monitor, the integer value associated with C is incremented by 1

True False

**12.** The semaphore-based solution to the Dining-Philosophers problem that we discussed in class (slides 3.36) prevents deadlock and starvation, while providing maximum concurrency

True False

**13.** Lock variables are only useful for critical section problems but semaphores are useful for a wide range of synchronization problems

True False

**14.** The race conditions pose problems for multi-processor systems; they cannot occur in single-processor systems

True False

**15.** Monitors can allow only one process to wait on its condition variable at any point of time

True False

**16.** Round Robin scheduler can result in large turn around times but provide good response time

True False

**17.** For any process set, increasing the time quantum value (for single-level Round Robin) will always increase the average turnaround time

True False

**18.** Rate monotonic scheduler uses inverse of a process's period time as its priority in a real-time system

True False

**19.** Preemptive kernels only allow preemption in user mode execution

True False

**20.** In thread scheduling with user level threads, the kernel also picks which user-level thread to run

True False

**Part III: Answer the following questions**

**21.** [5 points] What is the primary difference between spinlock and mutex (implemented as a semaphore)? Explain your answer.

A spinlock allows a process/thread to continually checks to see if the lock is released. A semaphore mutex puts a lock on the process/thread and when the mutex is signaled then the process/thread is released.

**22.** [5 points] In the H20 problem we solved in the class, the Oxygen thread has an additional mutex.signal() at the end (after makeWater(), Slide 3.48) which does not appear in the Hydrogen thread. What is the use of this signal call? Explain your answer.

The second mutex.signal() call in the Oxygen thread is to signal the Hydrogen thread to create a hydrogen molecule because you will always have at least 1 left over oxygen molecule that will need 2 hydrogen so it continues the process.

**23.** [5 points] Explain how processor affinity and load balancing can be at odds with each other in an SMP environment?

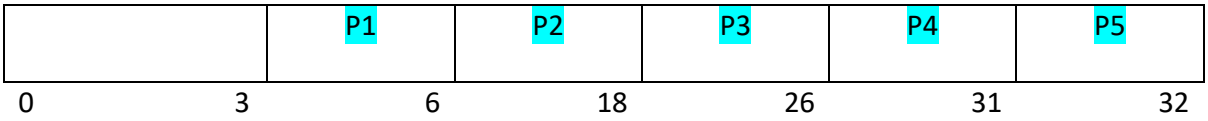
Processor affinity enables tasks to be bound to a processor so that task will only be executed on that processor; this utilizes private cache. Load balancing focuses on the optimization of resources by distributing tasks evenly across multiple processors; this happens by migrating waiting tasks on busy processors to idle processors.

**24.** [10 points] Consider the following set of processes on a single processor system

| Process name | Arrival time | CPU burst time |
|---|---|---|
| P1 | 3 | 3 |
| P2 | 4 | 12 |
| P3 | 6 | 8 |
| P4 | 8 | 5 |
| P5 | 9 | 1 |

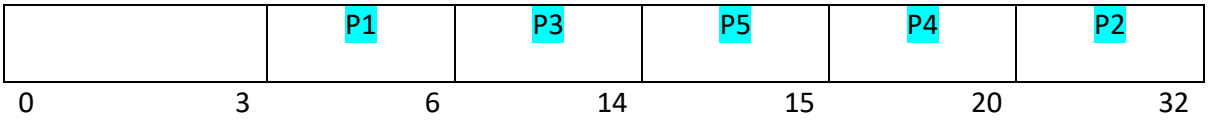
Answer the following questions

A. Show the Gantt chart for First Come First Served (FCFS) scheduling. Show the calculation of average waiting time.



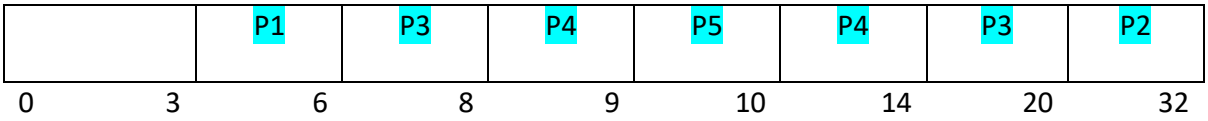

Waiting time: P1 = 0, P2 = 2, P3 = 12, P4 = 18, P5 = 22
Average waiting time: (0+2+12+18+22)/5 = 10.8

B. Show the Gantt chart for Shortest Job First - SJF (non-preemptive) scheduling. Show the calculation of average waiting time.



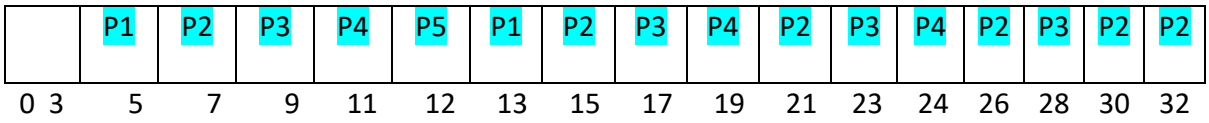

Waiting time: P1 = 0, P2 = 16, P3 = 0, P4 = 7, P5 = 5
Average waiting time: (0+16+0+7+5)/5 = 5.6

C. Show the Gantt chart for Shortest Remaining Time First - SRTF (preemptive). Show the calculation of average waiting time.




Waiting time: P1 = 0, P2 = 16, P3 = 6, P4 = 1, P5 = 0
Average waiting time: (0+16+6+1+0)/5 = 4.6

D. Show the Gantt chart for Round Robin (RR) scheduling with time quantum of 2 time units. Show the calculation of average waiting time.




Waiting time: P1 = 7, P2 = 20, P3 = 16, P4 = 12, P5 = 2
Average waiting time= (7+20+16+12+2)/5 = 12

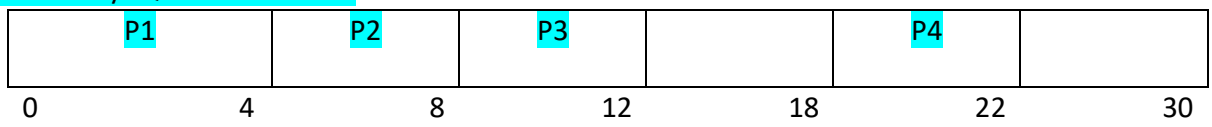
**25.** [10 points] Consider the following set of processes on a single-processor system.

| Process Name | Arrival Time | CPU Burst Length |
|---|---|---|
| P1 | 0 | 11 |
| P2 | 1 | 11 |
| P3 | 7 | 4 |
| P4 | 18 | 4 |

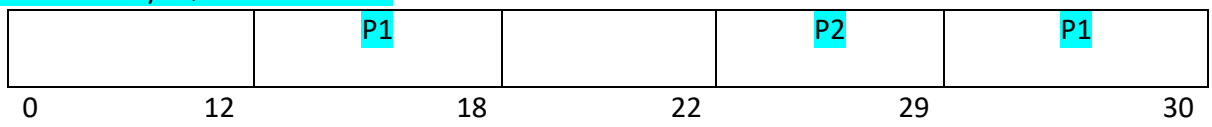
Show the Gantt chart for a system with multi-level feedback queue with the following rules (make sure to show process preemption and completion times):

- A newly arriving process joins the first queue, where Round Robin scheduling policy with Quantum Size = 4 is used.
- If a process from the first queue does not complete within its first time quantum, it is demoted to the second queue where Round Robin scheduling policy with Quantum Size = 8 is used.
- If a process from the first queue completes its execution, it exits.
- The processes in the first queue have higher priority than processes in the second queue. You will assume preemptive scheduling when solving this problem.

First-Priority: Quantum Size = 4




Second-Priority: Quantum Size = 8



**26.** [15 points] Our TA helps the students with OS/161 programming assignments during his office hours. The TA's office is rather small and has room for only 1 chair. You are expected to develop pseudo-code to meet the following requirements.

- There are 4 chairs in the hallway outside the office where students can sit and wait if the TA is currently helping another student.
- When there are no students how need help during office hours, the TA sits at the desk and takes a nap.
- If a student arrives and finds the TA sleeping, the student must wake him up.
- If a student arrives and finds that the TA is currently helping another student, the student sits on one of the chairs in the hallway and waits. If no chairs are available, the student will come back at a later time.

To get started, you are provided with the following hints.

1. If a student process arrives when TA process is helping another student process and all waiting chairs are also occupied by other student processes, it should call a function ComeBackLater() which does not return.
2. When the TA is helping a student, the TA process invokes HelpStudent() and (EXACTLY ONE) student process should invoke a function called GetHelpFromTA()

You are provided with the following variables and semaphores

Total number of chairs
    var TotalChairs
Number of students (with TA and waiting)
    var TotalStudents
Protecting TotalStudents var which can change as students come and go
    semaphore mutex

TA waits on the "student" semaphore
    semaphore student

Student waits on "TA" semaphore
    semaphore TA

After done getting help, student signals studentDone and waits on TADone
    semaphore studentDone
    semaphore TADone

Answer the following questions -

a. What is initial value you will choose for each of the variables and semaphores defined above? Briefly explain your answer.

```
int TotalChairs = 5; //number of chairs available inside and outside of room
int TotalStudents = 0; //no current students waiting or getting help from TA
semaphore mutex = 1; //initialized to 1 to block all other processes at .wait()
semaphore student = 0; //initialized to 0 to block current process at .wait()
semaphore TA = 0; //initialized to 0 to block current process at .wait()
semaphore studentDone = 0; //initialized to 0 to block current process at .wait()
semaphore TADone = 0; //initialized to 0 to block current process at .wait()
```

b. Write the TA process pseudo code below. Explain your answers.

```
student.wait()                  // wait for student process
TA.signal()                     // wake up the TA

HelpStudent()

studentDone.wait()              // wait till student is done
TADone.signal()                 // indicate TA is done
```

c. Write the Student process pseudo code below. Briefly explain your answers.

```
if(TotalChairs == 0)            // if no chairs (TA or waiting) available,
        ComeBackLater();        // ComeBackLater()

mutex.wait();                   // wait for a signal that it is okay for another student
TotalStudents++;                // indicate a new student
TotalChairs--;                  // decrement number of chairs
if(TotalStudents < 5)           // if students less than 5
        mutex.signal();         // signal that another student can come
student.signal();               // signal student ready
TA.wait();                      // wait if TA is busy

GetHelpFromTA()

studentDone.signal()            // indicate student is done
TADone.wait()                   // wait for TA to be done
TotalStudents--;                // decrease number of students by 1
TotalChairs++;                  // indicate chair is open
```

d. We have defined a “mutex” semaphore above. Explain why you need “mutex” semaphore? What will be the problem if we do not use the “mutex” semaphore.

   The mutex semaphore is there to protect the TotalStudents from being incremented past 5 because you have a maximum of 5 available slots. If we don’t have any mutex semaphore lock then we could have an overflow of students waiting and a negative amount of chairs available. The TA would never get his sleep!

**27.** [10 points] A certain bank account is shared by 2 persons. That is, any of the 2 account owners should be able to deposit money to or withdraw money from the account. To prevent race conditions while accessing the shared account, you will write a monitor construct.

The monitor will have two procedures: withdraw(X) and deposit(X). withdraw(X) will be executed by a transaction that deducts X dollars from the balance. Similarly, deposit(X) will be executed by a transaction that adds X dollars to the balance.

Your monitor construct should suspend the transaction (process) invoking the withdraw function if and only if there are no sufficient funds. The balance should never be negative during the execution. Following a deposit, suspended withdraw operation must be completed subject to the availability of funds. (For example, if balance = 100 and one process is suspended during the withdraw(500) transaction, a deposit of 200 dollars shouldn’t enable the completion of the pending withdraw operation.)

Write the monitor construct including withdraw(X) and deposit(X) procedures to solve this problem. You will assume that standard monitor wait and monitor signal operations are available for condition variables (as discussed during the lectures and in the textbook). If multiple processes are suspended on a condition variable C, the execution of C.signal() by Process A will resume the process waiting the longest on C (First-Come-First-Served ordering in conditionvariable waiting queues). However, this newly awakened process will have to wait until A leaves the monitor (or waits for another condition). There is no signal_all() or similar primitive that awakens all the processes waiting on a condition variable at once. Your solution should avoid deadlocks and busy waiting.

```
1  monitor atm_machine
2  {
3      condition notEnough, holdProcesses;
4      double balance, withdraw_amount = 0;
5
6
7      void deposit(double x)
8      {
9          balance = balance + x;
10         if(balance >= withdraw_amount) notEnough.signal();
11     }
12
13     void withdraw(double x)
14     {
15         if(withdraw_amount != 0) holdProcesses.wait();
16         withdraw_amount = x;
17         if(balance < withdraw_amount) notEnough.wait();
18         balance = balance - x;
19         withdraw_amount = 0;
20         holdProcesses.signal();
21     }
22 }
```