CS 3113 Intro to Operating Systems

Name and ID _Kyumin Lee 113591341_

Homework #4

Due date: 11/28/2023 at 11:59 pm

Instructions:
1) HW must be submitted on a typed pdf or word document.
<u>You must do your work on your own.</u>

Q1 (12 points). Many CPU-scheduling algorithms are parameterized. For example, the RR algorithm requires a parameter to indicate the time slice. Multilevel feedback queues require parameters to define the number of queues, the scheduling algorithms for each queue, the criteria used to move processes between queues, and so on.

These algorithms are thus really sets of algorithms (for example, the set of RR algorithms for all time slices, and so on). One set of algorithms may include another (for example, the FCFS algorithm is the RR algorithm with an infinite time quantum). What (if any) relation holds between the following pairs of algorithm sets?

a. RR and SJF
   RR is time-slice based, without consideration of the length of the job
      RR is more fair
   SJF is not time-slice based, but considers the length of the job
      SJF is more efficient for short jobs
b. Priority and SJF
   Priority doesn't care about the length of the job, it just considers the given priority.
   SJF does care about the length of the job, it can be seen as a special case of
   a Priority-Scheduling, if job length is considered as priority.
c. Multilevel feedback queues and FCFS
   Multilevel feedback queue is more complex and adoptable, it can even be configured to operate as FCFS. It's typically preemptive.
   FCFS Is Straightforward And Non-Preemptive.
d. Priority and FCFS
   Priority scheduling essentially becomes FCFS when all the processes have the same priority.
   Their main difference is that Priority scheduling is preemptive based on the priority.

Q2 (15 points). Show a **Gantt chart** for the processes in Table 1 using round robin scheduling with a quantum of 4. Also, what is the average wait time? What about the average turnaround time?

Table 1: Process table

Process id	Arrival time	Burst time
3	1	3
5	2	10
1	10	7
2	11	5
4	16	13

| none | P3 | P5 | P5 | P1 | P2 | P4 | P5 | P1 | P2 | P4 | P4 | P4 |
0 1 4 8 12 16 20 24 26 29 30 34 38 39

Waiting Time = Finish Time - Arrival Time - Burst Time
Turnaround Time = Finish Time - Arrival Time

Each waiting time and turn around time:
P1w = 29 - 10 - 7 = 12
T = = 19
P2w = 30 - 11 - 5 = 14
T = = 19
P3w = 4 - 1 - 3 = 0
T = = 3
P4w = 39 - 16 - 13 = 10
T = = 23
P5w = 26 - 2 - 10 = 14
T = = 24

Avg. waiting time = (12 + 14 + 0 + 10 + 14) / 5 = 10
Avg. turnaround time = (19 + 19 + 7 + 23 + 24) / 5 = 17.6

Q3 (15 points). Define the difference between preemptive and non-preemptive scheduling, and give at least two examples for each of them.

Preemptive Scheduling:

The OS can interrupt and suspend the currently running process to start or resume another process.

Examples: RR, Priority

Non-Preemptive Scheduling:

Once a process starts its execution on the CPU, it cannot be interrupted until it completes its CPU burst.

Examples: FCFS, SJF, unstoppable brother

Q4 (20 points). One GPEL (General Purpose Engineering Lab) system is having issues with its servers lagging heavily due to too many students being connected at a time. Below is the code that a student runs his/her code on a server"

```
void run_session (struct server s) {
        connect(s);
        execute_code();
        disconnect(s);
}
```

After testing, it turns out that the servers can run without lagging for a max of up to 100 students concurrently connected. How can you add semaphores to the above code to enforce a Strict limit of 100 players connected at a time? Assume that a GPEL server can create semaphores and share them amongst the student threads. You need to provide corresponding pseudocodes in order to get the credits.

```
// Global Semaphore
semaphore connectionLimiter = Semaphore(100); // Initialize semaphore with 100 permits

void run_session (struct server s) {
   wait(connectionLimiter);   // Decrement semaphore, wait if no permits are available
   connect(s);
   execute_code();
   disconnect(s);
   signal(connectionLimiter); // Increment semaphore, allowing another connection
}
```

Q5 (18 points). Suppose that processes P1, P2, and P3 shown below are running concurrently. S1 and S2 are among the statements that P1 will eventually execute, S3 and S4 are among the statements that P2 will eventually execute, and S5 and S6 are among the statements that P3 will eventually execute.

You need to use semaphores to guarantee that 1) Statement S3 will be executed AFTER statement S6 has been executed; 2) Statement S1 will be executed BEFORE statement S4; 3) Statement S5 will be executed BEFORE statement S2.

a. (10 points) Within the structure of the processes below, show how you would use semaphores to coordinate these three processes. Insert semaphore names and operations. Be sure to show initialize value of semaphores you are using. List semaphores you are using and their initial values here:

Semaphore SemS6 = 0;
Semaphore SemS1 = 0;
Semaphore SemS5 = 0;

… Process P1 … S1 Signal(SemS1); … Wait(SemS5); S2 ……	… Process P2 … Wait(SemS6); S3 … Wait(SemS1); S4 ……	… Process P3 … S5 Signal(SemS5); … S6 Signal(SemS6); ……

b. (8 points) After the semaphores have been appropriately placed, is it possible to determine which of the 6 statements above will be executed first, and which one will be executed last? If yes, give the statements, if no, give your reason.

Either S1 or S5 can be executed first, because they don't depend on any semaphore
S6 can execute independently at any point.
Either S2, S3 or S4 would be executed last, because they depend on a semaphore.

Q6 (20 points). True of False. A Semaphore is a useful synchronization primitive. Which of the following statements are true of semaphores?

a. Each semaphore has an integer value.
T

b. If a semaphore is initialized to 1, it can be used as a lock.
T

c. Semaphores can be initialized to values higher than 1.
T

d. Semaphore values can be negative in certain cases.
T (when there are more waiting processes than the number available)

e. We have discussed the classic wait() and signal() semaphore operations. POSIX declares these operations sem_wait() and sem_post(), respectively. Thus, calling sem_post() may block, depending on the current value of the semaphore.
F