COMP 304: Assignment 2

Berkay Barlas

April 14, 2019

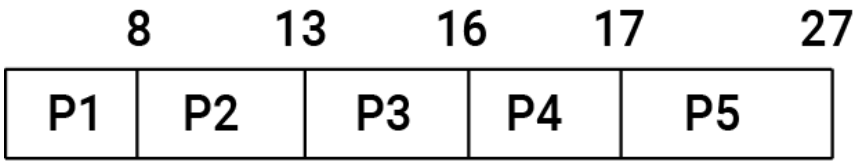
Date Performed: April 15, 2019
Instructor: Didem Unat

1 Problem 1

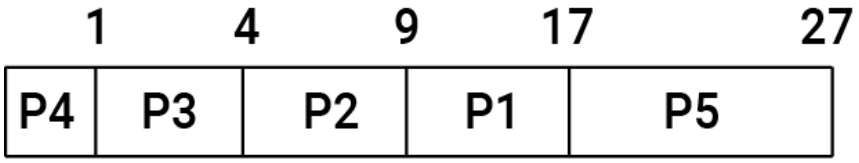
1.a

Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF, a non- preemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 4 ms) scheduling

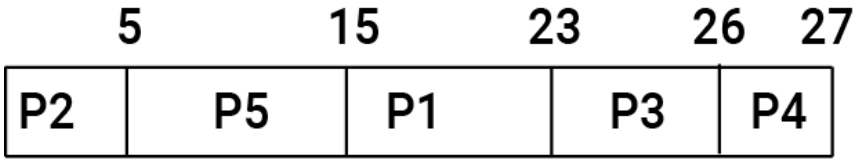
Process Execution Gantt Chart of FCFS



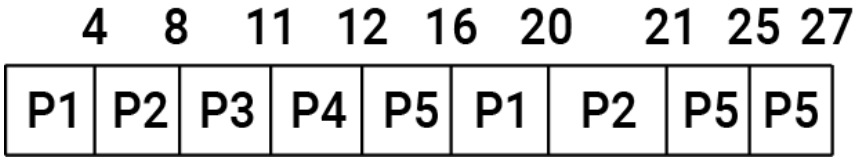
Process Execution Gantt Chart of SJF



Process Execution Gantt Chart of non-preemptive Priority



Process Execution Gantt Chart of RR



1.b

If the context-switch overhead is 1 ms, calculate the waiting time of each process for each of the scheduling algorithms in part (a). Which of the schedules results in the minimal average waiting time?
SJF(Shortest Job First) results in the minimal average waiting time.

Process Execution Gantt Chart of FCFS

Waiting Time of P1: 0
Waiting Time of P2: 9
Waiting Time of P3: 15
Waiting Time of P4: 19
Waiting Time of P5: 21
Average waiting time: (9 + 15 + 19 + 21) / 5 = 12,8

Process Execution Gantt Chart of SJF

Waiting Time of P1: 12
Waiting Time of P2: 6
Waiting Time of P3: 2
Waiting Time of P4: 0
Waiting Time of P5: 20
Average waiting time: (12 + 6 + 2 + 0 + 20) / 5 = 8

Process Execution Gantt Chart of non-preemptive Priority

Waiting Time of P1: 17
Waiting Time of P2: 0
Waiting Time of P3: 26
Waiting Time of P4: 30
Waiting Time of P5: 6
Average waiting time: (17 + 0 + 26 + 30 + 6) / 5 = 15,8

Process Execution Gantt Chart of RR

Waiting Time of P1: 16 + 5 - 8 = 13

Waiting Time of P2: 21 + 6 - 5 = 22

Waiting Time of P3: 10

Waiting Time of P4: 14

Waiting Time of P5: 27 + 8 -10 = 25

Average waiting time: (13 + 22 + 10 + 14 + 25) / 5 = 16,8

1.c Calculate average turnaround time for each of the scheduling algorithms in part (a).

Average turnaround time for FCFS
(8 + 13 + 16 + 17 + 27) / 5 = 16.2

Average turnaround time for SJF
(1 + 4 + 9 + 17 + 27) / 5 = 11.6

Average turnaround time for non-preemptive Priority
(5 + 15 + 23 + 26 + 27) / 5 = 19.2

Average turnaround time for RR
(20 + 21 + 11 + 12 + 27) / 5 = 18.2

2 Problem 2

Now assume that the context-switching overhead is equivalent to 0.5 ms. Calculate the CPU utilisation for all four scheduling algorithms in Problem 1.

$$CPUutilisation = 100 * \frac{TotalTimeSpendonProcesses}{\text{TotalTimeSpendonExecution}}$$

CPU utilisation of FCFS
(27 / 29 * 100) = 93,10

CPU utilisation of SJF
(27 / 29 * 100) = 93,10

CPU utilisation of non-preemptive Priority
(27 / 29 * 100) = 93,10

CPU utilisation of RR
(27 / 31 * 100) = 87,09

3 Problem 3

3.a

There is race condition in code, thus, output is non-deterministic and depends on order of process execution.
The available_connections is shared variable and it might be written by multiple threads simultaneously inside of the connect() and disconnect() methods.
Since write is actually consist of multiple Instructions

3.b

We could use locks to prevent race condition.

```
//PROBLEM 3
#define MAX_CONNECTIONS 5000
int available_connections = MAX_CONNECTIONS;
mutex lock;

/* When a thread wishes to establish a connection with the rver,
it invokes the connect() function:*/

int connect() {
    mutex_lock(&lock);
    if (available_connections < 1) {
        mutex_unlock(&lock);
        return -1;
    } else {
        available_connections--;
    }
    mutex_unlock(&lock);
    return 0;
}

 /* When a thread wishes to drop a connection with the server,
 it invokes disconnect() */
 int disconnect() {
    mutex_lock(&lock);
    available_connections++;
    mutex_unlock(&lock);
    return 0;
 }
```

3.c

To prevent race condition(s), could we replace the integer variable available connections with atomic integer, which allows atomic update of a variable of this type? Explain your answer.
No, replacing the integer with atomic integer is not enough for preventing race condition. While one of the threads inside of if statement of connect (before returning) another thread might increase the available_connections variable which might effect of the result of if statement.

4 Problem 4

We could use semaphore with initial value M ,however, if a party with more than 4 numcustomers arrive we need to be sure we have enough rooms. Thus, we can use a semaphore with initial value 1 and use it to check available number of rooms.

```
1
2     semaphore room = 1;
3     int roomNum = M;
4
5     int makeReservation(int numCustomers) {
6         wait(room);
7         int roomNeeded = numCustomers / 4 + 1;
8         // int a = 2 / 4 + 1 = 1; a = 1
9         if(roomNum >=roomNeeded ) {
10            //accept
11            roomNum = roomNum - roomNeeded;
12
13        } else {
14            //decline
15        }
16        signal(room);
17        return 0;
18    }
19
20    int checkout(int roomNumber) {
21        wait(room);
22        roomNum = roomNum + roomNumber;
23        signal(room);
24    }
```

5 Problem 5

```
1
2       data d1;
3       data d2;
4       data d3;
5       semaphore d1;
6       semaphore d2;
7       semaphore d3;
8   /* process P runs in this function */
9   void *P() {
10      wait(d1);
11
12      //makeCalculation
13
14      signal(d1);
15  }
16
17  /* process Q runs in this function */
18  void *Q() {
19      wait(d2);
20      wait(d1);
21
22      //makeCalculation
23
24      signal(d1);
25      signal(d2);
26  }
27
28  /* process R runs in this function */
29  void *R() {
30      wait(d2);
31      wait(d3);
32
33      //makeCalculation
34
35      signal(d3);
36      signal(d2);
37  }
38
39  /* process S runs in this function */
40  void *S() {
41      wait(d3);
42
43      //makeCalculation
44
45      signal(d3);
46  }
```