# COMP 304: Assignment 2

Berkay Barlas

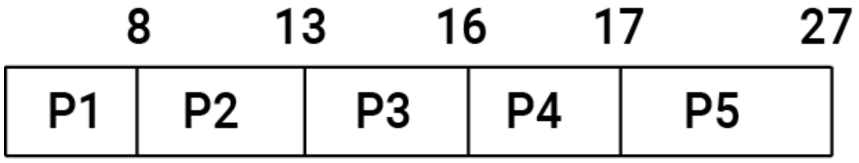
April 16, 2019

| | |
|---|---|
| Date Performed: | April 15, 2019 |
| Instructor: | Didem Unat |

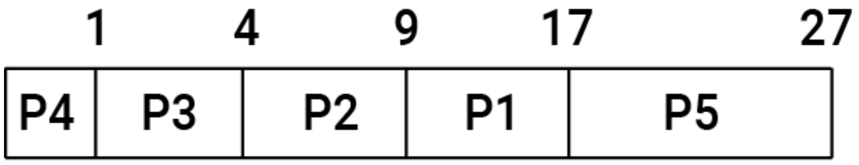
## 1 Problem 1

### 1.a

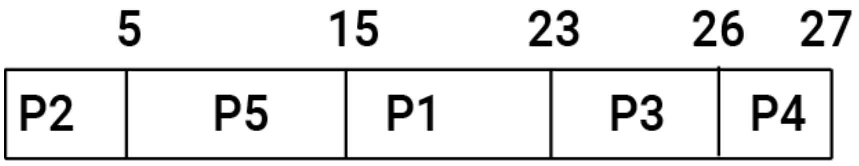
**Process Execution Gantt Chart of FCFS**



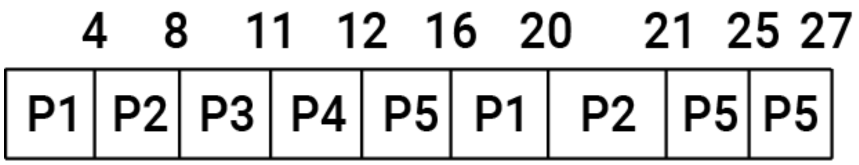
**Process Execution Gantt Chart of SJF**



**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:** 21

**Average waiting time:** ( 12 + 6 + 2 + 0 + 21 ) / 5 = 8.2

**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, a race condition may occur when two or more threads access the shared data and they try to change it at the same time.

## 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

Using atomic integer will prevent the race condition that caused by change of shared variable by multiple threads at the same time. However, replacing the integer with atomic integer is not enough. While one of the threads inside of the if statement of connect method (before returning) another thread might increase the available_connections variable which might effect of the result of if statement. For example, connect() method could return 1 even available_connections is 1 (It was 0 when the statement checked but then inreased to 1 before return by another thread inside of the disconnect() method. ).

## 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 a global variable to check available number of rooms.

```

    semaphore roomLock = 1;
    int roomNum = M;

    int makeReservation(int numCustomers) {
        int roomNeeded = (numCustomers + 3 ) / 4;
        // I added 3 to round upper bound.
        // When 1 || 2 || 3 || 4 customer come roomNeeded should be
     1
        // When 5 customer come roomNeeded should be 2
        // int a = (2 + 3) / 4 = 1; a = 1
        wait(roomLock);

        if(roomNum >=roomNeeded ) {
            //accept
            roomNum = roomNum - roomNeeded;

        } else {
            //decline
        }
        signal(roomLock);
        return 0;
    }

    int checkout(int roomNumber) {
        wait(roomLock);
        roomNum = roomNum + roomNumber;
        signal(roomLock);
    }
```

## 5 Problem 5

I we can use 3 different semaphores with initial value 1 for each data item. Process P and S requires only one semaphore since they only share single data item with other process. However, processes Q and R share two data item therefore they need 2 semaphores.

```

    data d1; // shared data item, could be any data
    data d2; // shared data item, could be any data
    data d3; // shared data item, could be any data
    semaphore lock1 = 1;
    semaphore lock2 = 1;
    semaphore lock3 = 1;
/* process P runs in this function */
void *P() {
    wait(lock1);

    //makeCalculation

    signal(lock1);
}

/* process Q runs in this function */
void *Q() {
    wait(lock2);
    wait(lock1);

    //makeCalculation

    signal(lock1);
    signal(lock2);
}

/* process R runs in this function */
void *R() {
    wait(lock2);
    wait(lock3);

    //makeCalculation

    signal(lock3);
    signal(lock2);
}

/* process S runs in this function */
void *S() {
    wait(lock3);

    //makeCalculation

    signal(lock3);
}
```