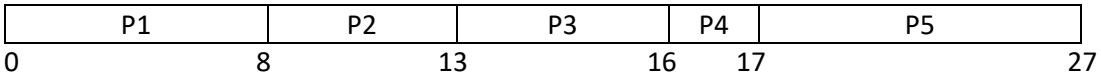
COMP 304 Operating Systems: Assignment 2

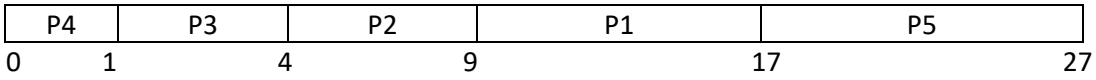
Due: April 12, midnight

Problem 1:

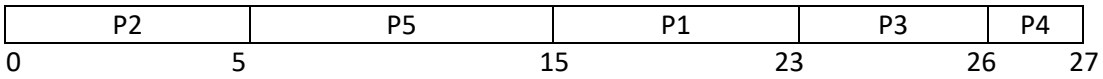
a) FCFS:



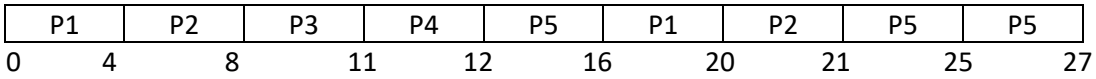
SJF:



Non-Preemptive Priority:



RR:



b) Waiting Time = Turnaround time – Burst time

Process	FCFS	SJF	Non-preemptive priority	RR
P1	8 – 8 = 0	17 – 8 = 9	23 – 8 = 15	20 – 8 = 12
P2	13 – 5 = 8	9 – 5 = 4	5 – 5 = 0	21 – 5 = 16
P3	16 – 3 = 13	4 – 3 = 1	26 – 3 = 23	11 – 3 = 8
P4	17 – 1 = 16	1 – 1 = 0	27 – 1 = 26	12 – 1 = 11
P5	27 – 10 = 17	27 – 10 = 17	15 – 10 = 5	27 – 10 = 17
Average	10.8	6.2	13.8	12.8

c) Turnaround Time = Completion time – Arrival time

Arrival time is 0 for all processes so -> turnaround time = completion time – 0

Process	FCFS	SJF	Non-preemptive priority	RR
P1	8	17	23	20
P2	13	9	5	21
P3	16	4	26	11
P4	17	1	27	12
P5	27	27	15	27
Average	16.2	11.6	19.2	18.2

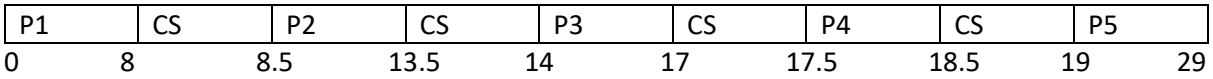
SJF has the minimal average turnaround time and the minimal average waiting time.

Sitare Arslantürk, 57677

Problem 2:

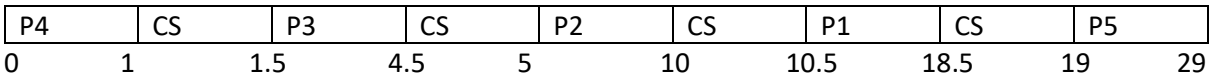
CPU Utilization = 100% * Total Process Time / Total Execution Time

FCFS:



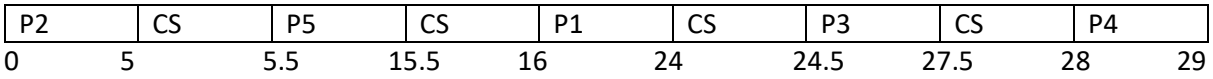
CPU Util = 100% * (29-0.5*4)/29 ~ 93%

SJF:



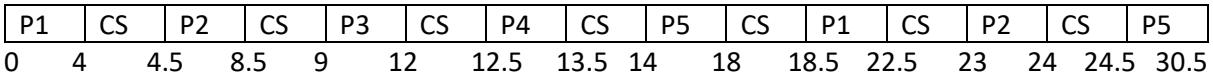
CPU Util = 100% * (29-0.5*4)/29 ~ 93%

Non-preemptive priority:



CPU Util = 100% * (29-0.5*4)/29 ~ 93%

RR:



CPU Util = 100% * (30.5 – 0.5*7)/30.5 ~ 89%

Problem 3

a) If there is multiple thread execution such as thread 1 performing connect() and thread 2 performing disconnect(), the order in which the threads execute matter causing a race condition and non-determinism in the output.

b)
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define MAX_CONNECTIONS 5000
int available_connections = MAX_CONNECTIONS;
pthread_mutex_t myMutex = PTHREAD_MUTEX_INITIALIZER;

int connect() {
        pthread_mutex_lock(&myMutex)
        if (available_connections < 1) {
                pthread_mutex_unlock(&myMutex)
                return -1;
        }
        else {
                available_connections --;
                pthread_mutex_unlock(&myMutex)
        }
        return 0;
 }

int disconnect() {
        pthread_mutex_lock(&myMutex)
        available_connections ++;
        pthread_mutex_unlock(&myMutex)
        return 0;
}
```

c) We can replace it, however the data race would not be completely prevented. The atomic integer would allow for atomic operations of incrementing and decrementing the number of available connections. This will result in safely updating or fetching data between shared threads/tasks or variables. The atomic operations will not allow other threads to observe a modification that is half-complete. So if the available connection value was increased from 0 to 1, the other thread will not be able to read the newly updated value of 1 and would go into the fi statement not being able to perform a connection although it could have.

Problem 4

totalRooms is initialized to M, the number of rooms available for online reservation. The semaphore is initialized as 1 to be able to allocate more than 1 room when necessary. When a reservation is accepted, the acquire() method is called. When a party leaves the hotel, the release() method is called. If the system reaches the number of allowable customers, subsequent calls to acquire() will do nothing to decline.

```
semaphore s =  1;
int totalRooms = M;

int acquire(int customers) {
        int rooms = 1
        if  (customers > 4) {
                rooms = customer / 4
                if (customer % 4 != 0) {
                        rooms++;
                }
        }
        wait(s);
        if ( totalRooms >= rooms) {
                totalRooms -= rooms;
        } else {
        }
        signal(s);
        return 0;
}


int release(int rooms) {
        wait(s);
        totalRooms += rooms;
        signal(s);
}
```

Problem 5

```
Monitor getService {

        int seniors[M];
        int availableDoctors = 4;
        int waitlist = 0;

        mutex myMutex;
        condition cond;

        void request_doctor(int priority) {
                mutex_lock(myMutex);
                waitlist++;
                seniors[waitlist] = priority;
                sort(seniors);

                while(availableDoctors == 0) {
                        if(seniors[waitlist] != priority) {
                                cond_wait(myMutex, cond);
                        }
                }

                waitlist--;
                availableDoctors--;
                seniors[waitlist] = 0;
                mutex_unlock(myMutex);
        }

        void release_doctor(){
                mutex_lock(myMutex);
                availableDoctors++;
                cond_signal(cond);
                mutex_unlock(myMutex);
        }
}
```

Problem 6

a) Need Matrix

	A	B	C	D
P1	0	6	4	2
P2	0	0	2	0
P3	1	0	0	2
P4	0	0	0	0
P5	0	7	5	0

b) Yes, it is in safe state. When the algorithm for checking whether the system is in safe state or not is followed, the final step Finish[i] == true for all i which means the system was able to execute all processes and ended up in a safe state. One of the possible safe sequences is P4, P3, P2, P5, P1.

c) No. We check three conditions to see if it can be granted.

1. Request_5 ? <= Need_5
   (0, 3, 3, 0) <= (0, 7, 5, 0)
   The first condition is satisfied.

2. Request_5 ? <= Available
   (0, 3, 3, 0) ?<= (1, 5, 2, 0)
   The second condition is not satisfied because the available resources of C does not meet the required 3. No need to check for the third condition. Request of P5 is not granted right away, it has to wait until the resources available meet its requests.