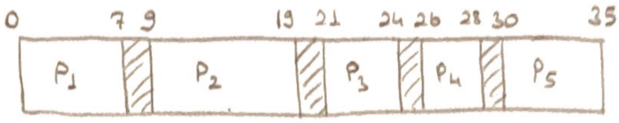
COMP304 Assignment 2

Report

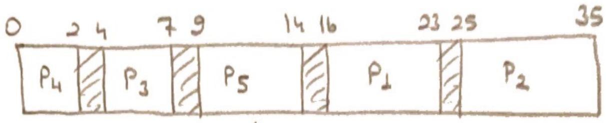
Doğa Ege İNHANLI
69033

Problem 1:

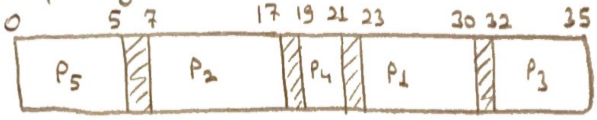
a) • FCFS:



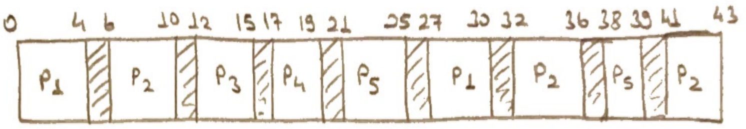
• SJF:



• Non-preemptive priority:



• RR



b) • FCFS:

R(P1) = 0 ms
R(P2) = 9 ms
R(P3) = 21 ms
R(P4) = 26 ms
R(P5) = 30 ms

$\Rightarrow R_{Avg} = \frac{0+9+21+26+30}{5} = 17{,}2\,ms$

• SJF

R(P1) = 16ms
R(P2) = 25 ms
R(P3) = 4 ms
R(P4) = 0 ms
R(P5) = 9ms

$\Rightarrow R_{Avg} = \frac{16+25+4+0+9}{5} = 10{,}8\,ms$

• Non-preemptive priority:

R(P1) = 23ms
R(P2) = 7ms
R(P3) = 32ms
R(P4) = 19ms
R(P5) = 0ms

$\Rightarrow R_{Avg} = \frac{23+7+32+19+0}{5} = 16.2\,ms$

• RR

R(P1) = 0 + (27 − 4) = 23ms
R(P2) = 6 + (32 − 10) + (41 − 36) = 33ms
R(P3) = 12 ms
R(P4) = 17 ms
R(P5) = 21 + (38 − 25) = 34ms

$\Rightarrow R_{Avg} = \frac{23+33+12+17+34}{5} = 23{,}8\,ms$

* SJF results with the minimal average waiting time.

c) • FCFS: $T_{TRend}(P_1) = 7$ ms

$T_{TRend}(P_2) = 19$ ms

$T_{TRend}(P_3) = 24$ ms $\Rightarrow T_{TRendAvg} = \frac{7+19+24+28+35}{5} = 22{,}6$ ms

$T_{TRend}(P_4) = 28$ ms

$T_{TRend}(P_5) = 35$ ms

• SJF: $T_{TRend}(P_1) = 23$ ms

$T_{TRend}(P_2) = 35$ ms

$T_{TRend}(P_3) = 7$ ms $\Rightarrow T_{TRendAvg} = \frac{23+35+7+2+14}{5} = 16{,}2$ ms

$T_{TRend}(P_4) = 2$ ms

$T_{TRend}(P_5) = 14$ ms

• Non-preemptive priority:

$T_{TRend}(P_1) = 30$ ms

$T_{TRend}(P_2) = 17$ ms

$T_{TRend}(P_3) = 35$ ms $\Rightarrow T_{TRendAvg} = \frac{30+17+35+21+5}{5} = 21{,}6$ ms

$T_{TRend}(P_4) = 21$ ms

$T_{TRend}(P_5) = 5$ ms

• RR: $T_{TRend}(P_1) = 30$ ms

$T_{TRend}(P_2) = 43$ ms

$T_{TRend}(P_3) = 15$ ms $\Rightarrow T_{TRendAvg} = \frac{30+43+15+19+39}{5} = 29{,}2$ ms

$T_{TRend}(P_4) = 19$ ms

$T_{TRend}(P_5) = 39$ ms

Comp 304 - Assignment 2

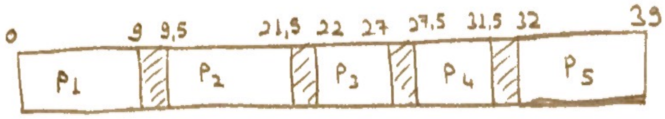
Doğa Ege İnhanlı
69033

Problem 2:

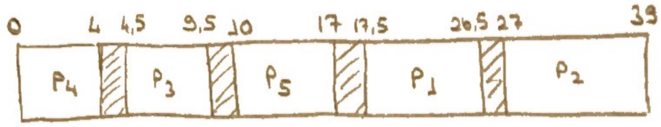
In this problem, the table becomes:

Process	Burst time	Priority
P_1	9	3
P_2	12	2
P_3	5	3
P_4	4	2
P_5	7	1

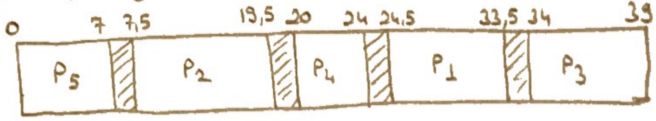
- FCFS:



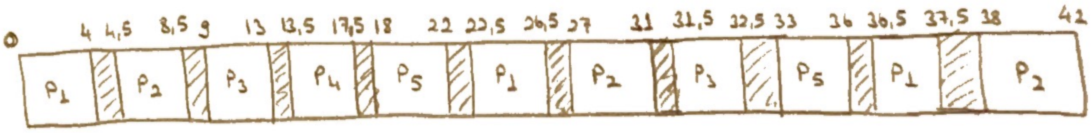
- SJF:



- Non-preemptive priority:



- RR



a)
- FCFS: 37 / 39 = 0,94871
- SJF: 37 / 39 = 0,94871
- Non-preemptive priority: 37 / 39 = 0,94871
- RR: 37 / 42 = 0,88095

b)
- FCFS: $R(P_1) = 0ms$
  $R(P_2) = 9,5ms$
  $R(P_3) = 22ms$
  $R(P_4) = 27,5ms$
  $R(P_5) = 32ms$
  $\Rightarrow R_{avg} = \frac{0 + 9,5 + 22 + 27,5 + 32}{5} = 18,2\ ms$ → Higher than the previous one

- SJF: $R(P_1) = 17,5ms$
  $R(P_2) = 27ms$
  $R(P_3) = 4,5ms$
  $R(P_4) = 0ms$
  $R(P_5) = 10ms$
  $\Rightarrow R_{avg} = \frac{17,5 + 27 + 4,5 + 0 + 10}{5} = 11,8ms$ → Higher than the previous one.

- Non-preemptive priority: $R(P_1) = 24{,}5\,ms$
  $R(P_2) = 7{,}5\,ms$
  $R(P_3) = 34\,ms$
  $R(P_4) = 20\,ms$
  $R(P_5) = 0\,ms$

$$\Rightarrow R_{avg} = \frac{24{,}5 + 7{,}5 + 34 + 20 + 0}{5} = 17{,}2\,ms$$
↓
Higher than the previous one.

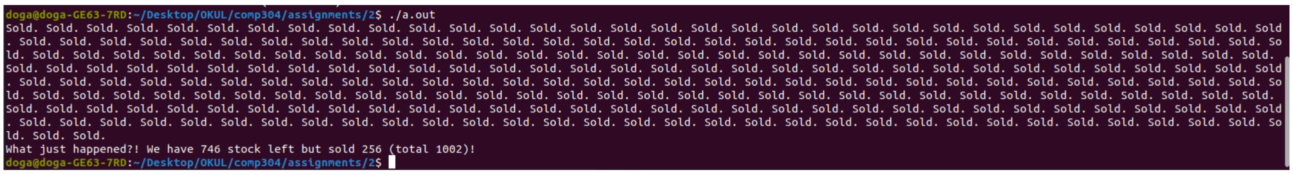
- RR
  $R(P_1) = 0 + (22{,}5 - 4) + (36{,}5 - 26{,}5) = 28{,}5\,ms$
  $R(P_2) = 4{,}5 + (27 - 8{,}5) + (38 - 31) = 30\,ms$
  $R(P_3) = 9 + (31{,}5 - 13) = 27{,}5\,ms$
  $R(P_4) = 13{,}5\,ms$
  $R(P_5) = 18 + (33 - 22) = 29\,ms$

$$\Rightarrow R_{avg} = \frac{28.5 + 30 + 27{,}5 + 13{,}5 + 29}{5}$$
$= 25{,}7\,ms$ → Higher than the previous one

Question 3:

Part A)

For this part, I basically created 256 new threads which executes the *sell* function. Since there are not any synchronization mechanisms, the created threads trigger race conditions. A sample output of a race condition situation is shown below.



```
doga@doga-GE63-7RD:~/Desktop/OKUL/comp304/assignments/2$ ./a.out
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold
. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. So
ld. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold
. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. So
ld. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold
. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. So
ld. Sold. Sold.
What just happened?! We have 746 stock left but sold 256 (total 1002)!
doga@doga-GE63-7RD:~/Desktop/OKUL/comp304/assignments/2$ 
```

Figure 1: Output of The Race Condition

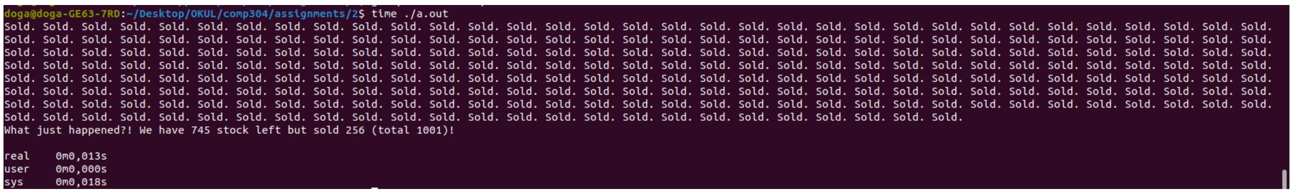
Part B)

For this part,

- I declared a global semaphore named *my_mutex*. (line 62)
- The function *init()* basically initializes the semaphore by calling *sem_init.* The second parameter is 0 since it will be used among the threads of the same process. (line 79)
- The function *lock()* acquires the lock by calling *sem_wait.* (line 85)
- The function *unlock()* releases the lock (signaling) by calling *sem_post* (line 92)
- I called the *lock()* function at line 37 in *sell()* because the critical section starts with the SLOWDOWN. After the critical section is executed, I called *unlock()* at line 51 in *sell()*. By this way, I prevent the race condition.

Part C)

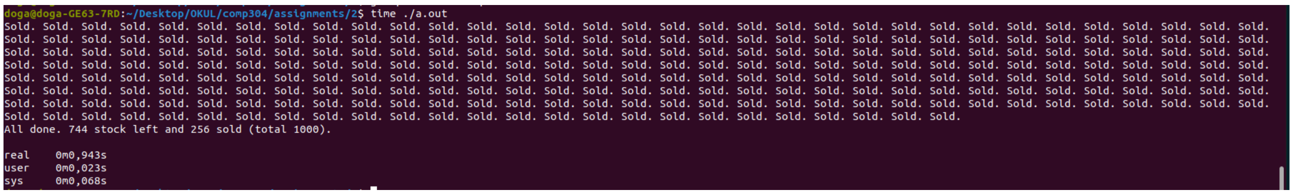
The time measures for part A and B are shown below. The fact that it takes more time with the semaphore is expected since it continuously acquires and releases the locks, which is an extra work by comparing with part A.



```
doga@doga-GE63-7RD:~/Desktop/OKUL/comp304/assignments/2$ time ./a.out
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
What just happened?! We have 745 stock left but sold 256 (total 1001)!

real	0m0,013s
user	0m0,000s
sys	0m0,018s
```

Figure 2: Time for Part A



```
doga@doga-GE63-7RD:~/Desktop/OKUL/comp304/assignments/2$ time ./a.out
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
All done. 744 stock left and 256 sold (total 1000).

real	0m0,943s
user	0m0,023s
sys	0m0,068s
```

Figure 3: Time for Part B

Question 4:

Part A)

```
monitor deliveryMonitor {
        struct order {
                int id;
                int dist;
                int size;
        };
        int available_cars;  //number of available delivery guys
        struct order waiting_orders[M];
        int num_waiting;  //number of orders currently waiting
        // !you can introduce shared variables
        semaphore order = 1;
        semaphore release = 1;
        cond thereIsAvailable;

        // !implement this function
        void request_delivery(int order_id, int distance, int size)
        {
                wait(order);
                waiting_orders[num_waiting].id = order_id;
                waiting_orders[num_waiting].dist = distance;
                waiting_orders[num_waiting].size = size;
                num_waiting++;
                signal(order);
                sort();
                while(waiting_orders[0].id != order_id || (waiting_orders[0].id == order_id &&
                                                        available_cars == 0)){
                        wait(thereIsAvailable);
                }
                available_cars--;
                num_waiting--;
                pop(waiting_orders); // I assume that there is a pop method which removes the 0th
                                element from the queue and slides the remaining ones one by one.
        }

        void release_car() {
                // !implement this function
                wait(release);
                available_cars++;
                signal(release);
                broadcast(thereIsAvailable);
        }

        void initialize()
        {
                available_cars = 5;
                num_waiting = 0;
        }
}
```

Part B)

It may cause a bankruptcy because if the restaurant gets receive orders from very far places continuously, they never deliver the order of the closer clients. So, the clients from the closer places will starve and the restaurant lose money continuously. This finally ends up with the starvation of the closest ones and the bankruptcy of the restaurant.

Part C)

```
void request_delivery(int order_id, int distance, int size){
        wait(order);
        waiting_orders[num_waiting].id = order_id;
        waiting_orders[num_waiting].dist = distance;
        waiting_orders[num_waiting].size = size;
        num_waiting++;
        signal(order);
        sort();
        while(waiting_orders[0].id != order_id || (waiting_orders[0].id == order_id &&
                                        available_cars < waiting_orders[0].size)){
                wait(thereIsAvailable);
        }
        available_cars -= waiting_orders[0].size;
        num_waiting--;
        pop(waiting_orders); // I assume that there is a pop method which removes the 0th
                        element from the queue and slides the remaining ones one by one.
}
```

Part D)

It increases the risk of bankruptcy since more cars will be busy in the scenario which causes the enlargement of the waiting queue, delays and failures of the deliveries.