Comp304

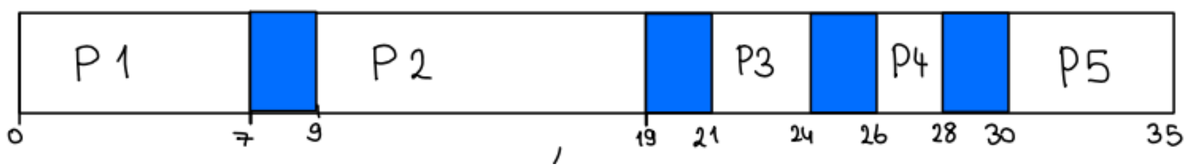
Assignment -2

Irem Demir, 69563

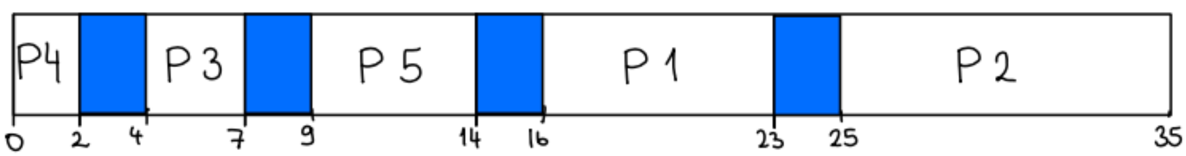
Problem-1:

A)

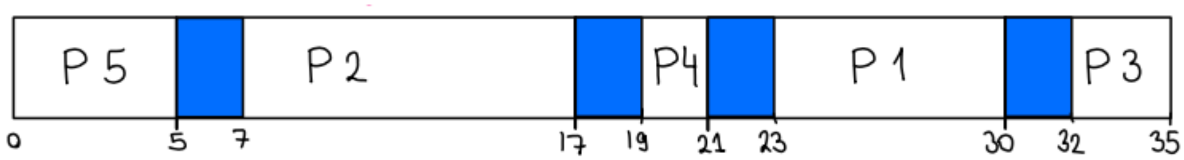
FCFS:



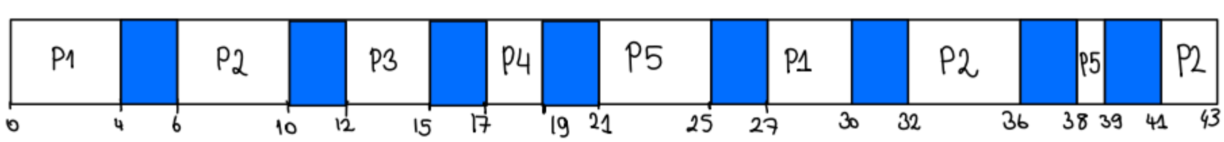
SJF:



Priority:



RR:



B)

FCFS:

Waiting Time : $P_1 = 0$, $P_2 = 9$, $P_3 = 21$, $P_4 = 26$, $P_5 = 30$

$\sum$ Waiting Time = 86

Avg Waiting Time = $86/5 = 17.2$

SJF:

Waiting Time : $P_4 = 0$, $P_3 = 4$, $P_5 = 9$, $P_1 = 16$, $P_2 = 25$

$\sum$ Waiting Time = 54

Avg Waiting Time = $54/5 = 10.8$ Minimum Average Waiting Time

Priority:

Waiting Time : $P_5 = 0$, $P_2 = 7$, $P_4 = 19$, $P_1 = 23$, $P_3 = 32$

$\sum$ Waiting Time = 81

Avg Waiting Time = $81/5 = 16.2$

RR:

Waiting Time : $P_1 = 23$, $P_2 = 6+22+5$, $P_3 = 12$, $P_4 = 17$, $P_5 = 21 + 13$

$\sum$ Waiting Time = 119

Avg Waiting Time = $119/5 = 23.8$

C)

FCFS:

Turnaround Time : $P_1 = 7$, $P_2 = 19$, $P_3 = 24$, $P_4 = 28$, $P_5 = 35$

Σ Turnaround Time = 113

Avg Turnaround Time = 113 / 5 = 22.6

SJF:

Turnaround Time : $P_4 = 2$, $P_3 = 7$, $P_5 = 14$, $P_1 = 23$, $P_2 = 35$

Σ Turnaround Time = 81

Avg Turnaround Time = 81 / 5 = 16.2 → Minimum Turnaround Avg

Priority:

Turnaround Time : $P_5 = 5$, $P_2 = 17$, $P_4 = 21$, $P_1 = 30$, $P_3 = 35$

Σ Turnaround Time = 108

Avg Turnaround Time = 108 / 5 = 21.6

RR:

Turnaround Time : $P_1 = 30$, $P_2 = 43$, $P_3 = 15$, $P_4 = 19$, $P_5 = 39$

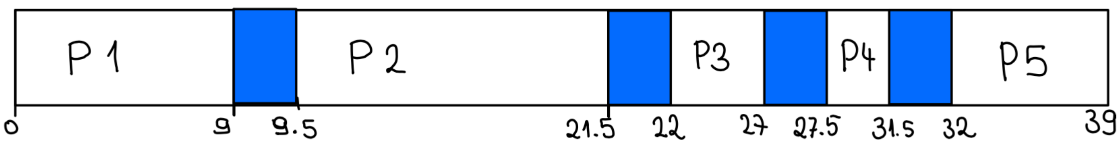
Σ Turnaround Time = 146

Avg Turnaround Time = 146 / 5 = 29,2

Problem-2:

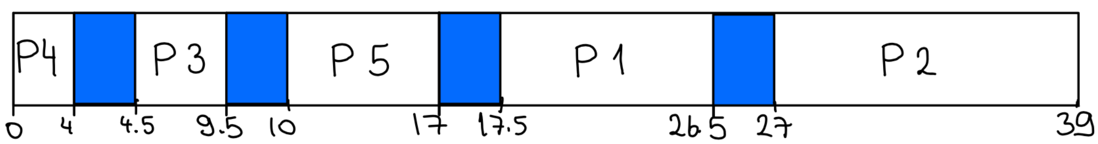
A)

FCFS:



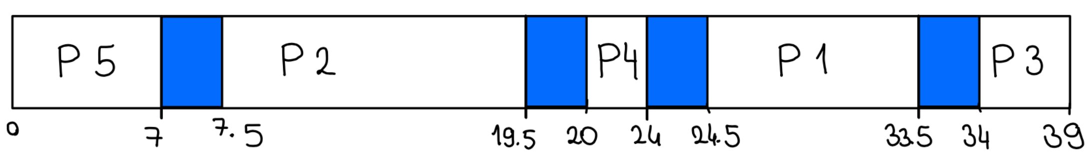
CPU Utilisation := 37/39 = 94.8%

SJF:



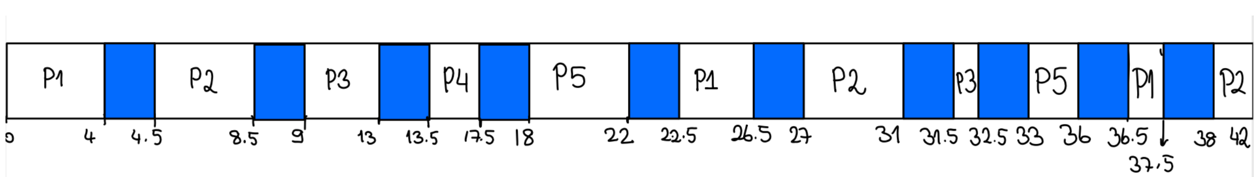
CPU Utilisation := 37/39 = 94.8%

Priority:



CPU Utilisation := 37/39 = 94.8%

RR:



CPU Utilisation := 37/42 = 88%

B)

FCFS:

Waiting Time : $P_1 = 0$, $P_2 = 9.5$, $P_3 = 22$, $P_4 = 27.5$, $P_5 = 32$

$\sum$ Waiting Time = 91

Avg Waiting Time = $91/5 = 18.2$

SJF:

Waiting Time : $P_4 = 0$, $P_3 = 4.5$, $P_5 = 10$, $P_1 = 17.5$, $P_2 = 27$

$\sum$ Waiting Time = 59

Avg Waiting Time = $59/5 = 11.8$ → Minimum Average Waiting Time

Priority:

Waiting Time : $P_5 = 0$, $P_2 = 7.5$, $P_4 = 20$, $P_1 = 24.5$, $P_3 = 34$

$\sum$ Waiting Time = 86

Avg Waiting Time = $86/5 = 17.2$

RR:

Waiting Time : $P_1 = 18.5 + 10$, $P_2 = 4.5 + 18.5 + 7$, $P_3 = 9 + 18.5$, $P_4 = 13.5$, $P_5 = 18 + 11$

$\sum$ Waiting Time = 128,5

Avg Waiting Time = $128{,}5/5 = 25.7$

	Problem-1	Problem-2
FCFS	17.2	18.2
SJF	10.8	11.8
Priority	16.2	17.2
RR	23.8	25.7

For all of the algorithms, problem-2 setup has higher average waiting time. This makes sense since although CPU has smaller CS time the increase in time that each process requires outnumbers it.

Problem-3:

A)

I added a for loop that iterates race_trigger times that creates a new thread using create_new_thread function given sell as a function to be executed by each tread. With different race_trigger=128, result that does not add up to total stock is obtained as below. The reason for that is when a thread sell before it updates the conditions other can enter the processes and we can end up with total that does not add up to initial or minus stocks (w/ higher race_trigger we obtain that).

```
irem@irem-Inspiron-3576:~/Downloads/comp304_a2$ ./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. Sol
. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. So
d. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. S
What just happened?! We have 874 stock left but sold 128 (total 1002)!
```

B)

I filled the init, lock, and unlock functions using sem_init, sem_wait, and sem_post functions respectively using a global variable I defined as mymutex. After that I added lock and unlock statements in the critical sections of sell function using mymutex variable. Right before checking the stock number I added a lock because if someone checks it when it is greater than zero and another just decreases number it will create race conditions. Compiling with race_trigger=128 can be seen as below:

```
rem@irem-Inspiron-3576:~/Downloads/comp304_a2$ ./a.out
old. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. So
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. S
 Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
ll done. 872 stock left and 128 sold (total 1000).
rem@irem-Inspiron-3576:~/Downloads/comp304_a2$ ./a.out
old. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. So
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. S
 Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
ll done. 872 stock left and 128 sold (total 1000).
```

C)

Part-a takes less time then part-b. This is reasonable since in part-b other threads wait for first one in the checking process/ when enters the critical section. This causes high waiting time for each process and as a result it take lots of time.

```
real    0m0,039s
user    0m0,020s
sys     0m0,061s
```

```
real    0m2,399s
user    0m0,026s
sys     0m0,214s
```

Problem-4:

A)

```
monitor deliveryMonitor { struct order {
int id; int dist; int size;
};
int available_cars;
struct order waiting_orders[M];
int num_waiting; //number of orders currently waiting
// !you can introduce shared variables
condition not_all_on_delivery;
// !implement this function
void request_delivery(int order_id, int distance) {
  waiting_orders[num_waiting].id = order_id;
  waiting_orders[num_waiting].dist = distance;
  num_waiting ++;
  //wait while order is not the priority or it is the priorty but available cars are 0
  waiting_orders.sort(dist)
  while(
  waiting_orders[0].order_id != order_id ||(waiting_orders[0] == order_id && available_cars == 0)){
    wait(not_all_on_delivery);
  }
  available_cars -=1;
  //assume this pop function shifts all sorted queue by one while poping the prior one
  waiting_orders.pop(prior_delivery);
}
void release_car() {
// !implement this function
  available_cars += 1;
  //signal all the deliveries that are waiting
  broadcast(not_all_on_delivery);

}
void initialize() {
  available_cars = 5;
  num_waiting = 0;
   /* initialize condition variable */
    }
}
```

I used a condition variable to indicate when a car comes from delivery it broadcast to it (signal all waiting deliveries) so that they can check the described while condition in the commented code and get delivered if they satisfy it. Although there can be multiple deliveries ordered at the same time we do not need to use mutex in critical section of request_delivery since monitor provides such exclusion by its implementation.

B)

Here, we are using a priority based on distance. Assume we get 6 orders 5 within a long and 1 within a close distance. Our first 5 delivery is made to long distance. While they are on service assume our close distance is failed and new 5 long and 10 small distance order is come. Now we will have again long distance deliveries and 10 small distance may fail again. Therefore, this delivery mechanism can put us in a risk of bankruptcy.

C)

```
monitor deliveryMonitor { struct order {
int id; int dist; int size;
};
int available_cars;
struct order waiting_orders[M];
int num_waiting; //number of orders currently waiting
// !you can introduce shared variables
condition not_all_on_delivery;
// !implement this function
void request_delivery(int order_id, int distance, int size) {
  waiting_orders[num_waiting].id = order_id;
  waiting_orders[num_waiting].dist = distance;
  waiting_orders[num_waiting].size = size;
  num_waiting ++;
  //wait while order is not the priority or it is the priorty but available cars are 0
  waiting_orders.sort(dist)
  while(
  waiting_orders[0].order_id != order_id ||(waiting_orders[0] == order_id && available_cars != size)){
    wait(not_all_on_delivery);
  }
  available_cars -=size;
  //assume this pop function shifts all sorted queue by one while poping the prior one
  waiting_orders.pop(prior_delivery);
}
void release_car() {
// !implement this function
  available_cars += 1;
  //signal all the deliveries that are waiting
  broadcast(not_all_on_delivery);

}
void initialize() {
  available_cars = 5;
  num_waiting = 0;
   /* initialize condition variable */
    }
}
```

There are two options in here. Either we can wait until we have enough cars for most distant order to be delivered, either we can change our priority. However, since question does not ask to change our priority condition when most distant cannot be delivered, I only updated waiting condition in the while loop.

D)

With this setup, we have higher risk of bankruptcy since while waiting for satisfying size condition we have chance to miss more deadlines for deliveries. Like assume we will wait for 5 cars to come to deliver an order. Comparing the first setup, in the first setup at least we do not have a condition when there is an order waiting also our cars are waiting. They are busy at all times. Comparing in a general way we can conclude the following: While waiting we may deliver all close distant ones however here we are putting risk of fail each of them. This shows that we need more complicated and broader approach to solve the priority issue rather than just giving priority to most distant one.