COMP 304 - Operating Systems: Assignment 2

Dilara Deveci

0068182

Problem 1

Part A

A1. First Come First Served

P0	P1	P2	P3	P4

0 14 15 23 24 44 45 49 50 56

A2. Shortest Job First (SJF)

P1	P3	P4	P0	P2

0 8 9 13 14 20 21 35 36 56

A3. Non-preemptive with Priority

P1	P0	P4	P3	P2

0 8 9 23 24 30 31 35 36 56

A4. Round-Robin

P0	P1	P2	P3	P4	P0	P1	P2	P0	P2	P2

0 6 7 13 14 20 21 25 26 32 33 39 40 42 43 49 50 52 53 59 60 62

Part B

B1.

P0 = 0

P1= 15

P2 = 22

P3 = 41

P4 = 44

Avg. Waiting Time = 24.4

B2.

P0 = 21

P1= 0

P2 = 34

P3 = 5

P4 = 8

Avg. Waiting Time = 13.6

B3.

P0 = 9

P1= 0

P2 = 34

P3 = 27

P4 = 18

Avg. Waiting Time = 17.6

B4.

P0 = 0+27+11=38

P1= 7+27=34

P2 = 12+23+4+1=40

P3 = 17

P4 = 20

Avg. Waiting Time = 29.8

Hence, Shortest Job First (SJF) results in the minimal average waiting time with waiting time of 13.6 ms

Part C

C1.

P0 = 14

P1= 23

P2 = 42

P3 = 45

P4 = 50

Avg. Turnaround Time = 34.8

C2.

P0 = 35

P1= 8

P2 = 54

P3 = 9

P4 = 14

Avg. Turnaround Time = 24

C3.

P0 = 23

P1= 8

P2 = 54

P3 = 31

P4 = 24

Avg. Turnaround Time = 28

C4.

P0 = 52

P1= 42

P2 = 60

P3 = 21

P4 = 26

Avg. Turnaround Time = 40.2

Hence, Shortest Job First (SJF) has the lowest average turnaround time with turnaround time of 24 ms.

Part D

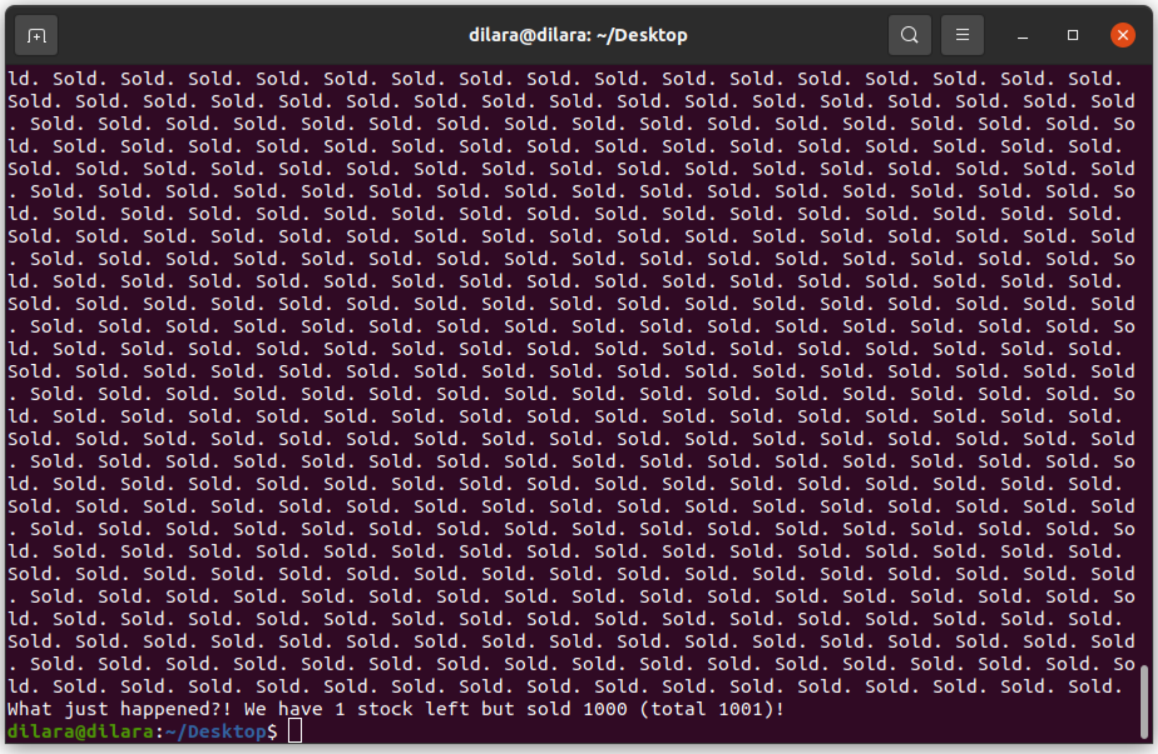
Round Robin is best in terms of response time. After time quantum amount of time has elapsed, the process is preempted and added to the end of the ready queue therefore each process is scheduled at worst about 24 ms later than the start time. Besides, in other three algorithms Response Time = Waiting Time but for RR Response Time < Waiting Time.

Problem 2

Part A

I have created total_stock number of pthreads in a for loop using the create_new_thread() function. I give the sell function as its parameter so that each thread will execute sell().
Since they can concurrently execute the sell function, they enter to the critical section concurently; modify the global variable *sold* hence a race condition is observed.

Here is the screenshot of such a case:



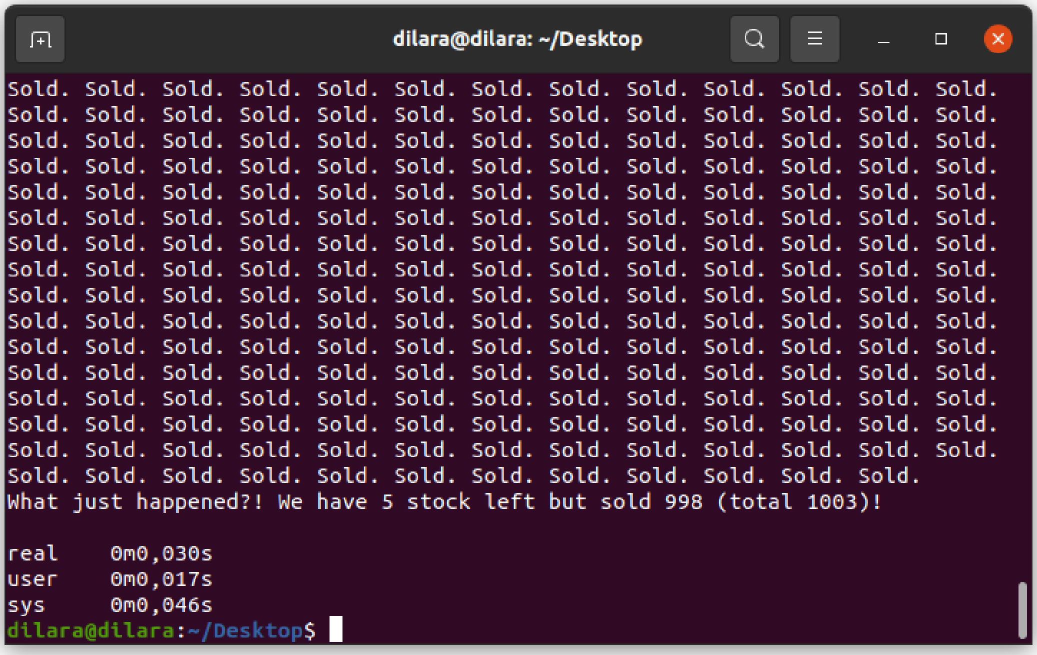
Part B

I have faced problems when using the given API on Ubuntu hence I used original functions. I have created a semaphore as a global variable; sem_t sem.
Then, I filled out the init() function by sem_init(), the created semaphore sem and value 1 are passed as parameters.
Then, I filled out the lock() function with sem_wait() and passed the semaphore sem as the parameter. Then, I filled out the unlock() function with sem_post() and passed the semaphore sem as the parameter. Finally, I have added lock() and unlock() to the beginning and to the end of the sell function as sell function modifies the shared variable; so it is a critical section.

Part C

When the time ./a.out is used for program execution, it can be seen that the code for Part A is run much faster than Part B. The screenshots are provided for Part A and Part B respectively.

Part A:



```
dilara@dilara: ~/Desktop
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. 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 5 stock left but sold 998 (total 1003)!

real    0m0,030s
user    0m0,017s
sys     0m0,046s
dilara@dilara:~/Desktop$
```

Part B:

```
dilara@dilara: ~/Desktop
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold. Sold.
Sold. 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. 0 stock left and 1000 sold (total 1000).

real    0m2,449s
user    0m0,053s
sys     0m0,205s
dilara@dilara:~/Desktop$
```

Part B's code takes longer to run as each time sell function is called, we first lock meaning execute some other code (sem_wait) and when we are done executing we unlock; again some other code (sem_post) which all cause a latency.

Problem 3

```
Monitor dentist_office {
        patient waiting_room[N];
        condition patient, dentist;

        void get_dental_treatment(patient x) {
                if(array "waiting_room" is full)
                        //leave
                else
                        waiting_room.add(x);
                        patient.signal();
                        dentist.wait();
        }

        void get_next_patient() {
                if(array "waiting_room" is empty)
                        wait(patient);
                dentist.signal();
        }

        void finish_treatment(patient x) {
                waiting_room.remove(x);
        }
}
```

The condition variables represents whether the process is the dentist or a patient. There is an array representing the waiting room with N seats.

If a patient wants to get dental treatment, he walks in to the waiting room if there are no seats; if the array is full, he leaves. If not, he is added to the array as he is now waiting in the waiting room. Then patient.signal() is called indicating a patient is ready to enter the treatment room. Also, dentist.wait() is called indicating we are now waiting the dentist to be done with another patient in the room.

If the dentist is ready to get the next patient, he checks if there is anyone in the waiting room, if there is no one he waits for a patient to come; so patient.wait() is called. (It is only signaled when a patient wants to get treatment) If the waiting room is not empty, dentist.signal is called indicating dentist is ready to take a patient.

Finally, when the treatment is done and patient is leaving, the patient is removed from the waiting room.

Problem 4

The system is deadlock free. There are 3 processes and 4 resources and since the resources are of the same type in the worst scenario one process must be able to obtain two resources. Each process needs at most 2 resources so this process gets all the resources it needs therefore it will execute and release the resources afterwards. After that other three resources can execute without a deadlock.