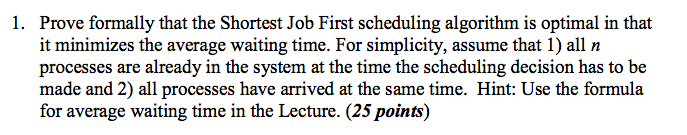
CS 520 Homework 2 | CWID 10430147 | Divyendra Patil | Username: dpatil3  
Date: 09/09/2017



Solution:

Shortest Job First is a scheduling algorithm that assigns to each process the length of its next CPU burst/execution time. CPU is then given to the process with the minimal CPU burst from the waiting queue. Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process.

Consequently, the average waiting time decreases.

To prove that SJF is optimal, i.e, it minimizes the average waiting time let us consider processes with different burst times that arrive in the system at the same time.

Process Burst Time

P1 7

P2 3

P3 6

P4 2

Running these processes in the given order, i.e., using FCFS Scheduling.

FCFS scheduling Gantt chart:

P1 P2 P3 P4

0 7 10 16 18

Average waiting time = (0 + 7 + 10 + 16) / 4 = 8.25

Now, running these processes using SJF Scheduling.

SJF scheduling Gantt chart:

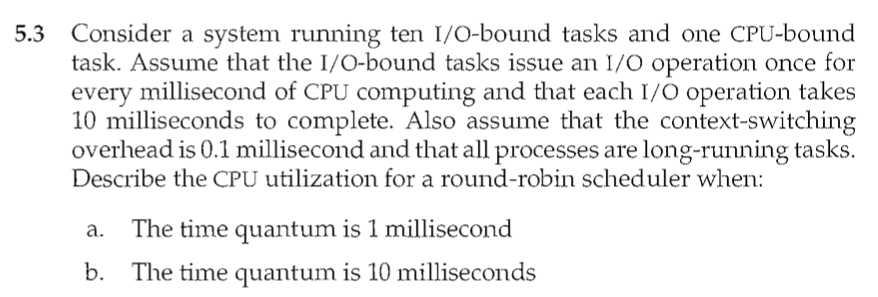
P4 P2 P3 P1

0 2 5 11 19

Average waiting time = (0 + 2 + 5 + 11) / 4 = 4.5

From the above calculations we can see that SJF gives the minimum average time, which satisfies our claim SJF scheduling algorithm is optimal.

If we continue this process, shorter jobs will be moved towards head of the queue and longest will be towards the tail of the queue and achieve minimum waiting.



(a) Since the time quantum is 1 millisecond, the scheduler incurs a 0.1 millisecond context-switching cost for every context-switch. This results in a CPU utilization of 1/1.1 \* 100 = 91%.

(b) The time quantum is 10 milliseconds: The I/O-bound tasks incur a context switch after using up only 1 millisecond of the time quantum. The time required to cycle through all the processes is therefore 10\*1.1 + 10.1 (as each I/O-bound task executes for 1 millisecond and then incur the context switch task, whereas the CPU-bound task executes for 10 milliseconds before incurring a context switch). The CPU utilization is therefore 20/21.1 \* 100 = 94%.

=====================================================

**Deep Explanation for similar problem**:

On every 1ms i/o operation done, it will be complete in 10 ms so 10 i/o operation have   
time = 10\*1+.1\*10 = 11ms ...(here 0.1 switching overhead )

CPU task, for which we can consider it runs for 10ms because input ouput operation will be completed in 10ms ... (given as all process are long running tasks) & hence if case 1 ... time quantum is 1ms then CPU task takes = 10\*1+10\*0.1   
(switching after every 1ms quantum time) = 11ms

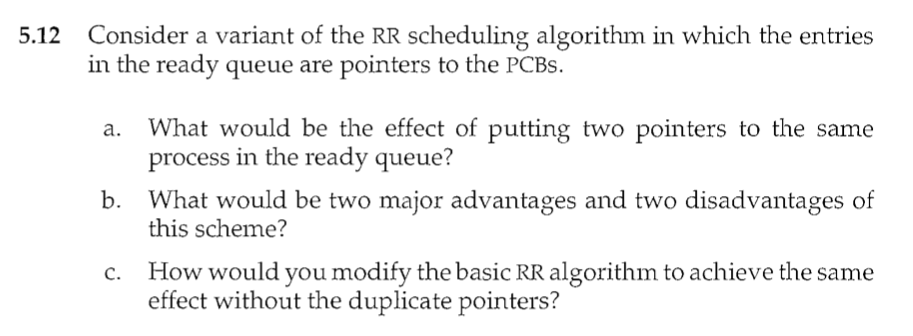
so CPU utilization = useful work/ total work

which is….

(10\*1+10\*1)/11+11= 20/22=90.90%

**For case 2** time quantum =10ms so cpu takes = 10\*1+1\*.1=10.1

cpu utilization = 20/ 11+10.1= 20/ 21.1 = 94.78%



a] The process would be run twice as many times.

b] **Advantages**

1. It would allow the user to prioritize more important processes. (more important jobs could be given more time, in other words, higher priority in treatment. The consequence, of course, is that shorter jobs will suffer.)

2. It prevents starvation of lower priority processes.

3. We do not have to change the scheduling algorithm.

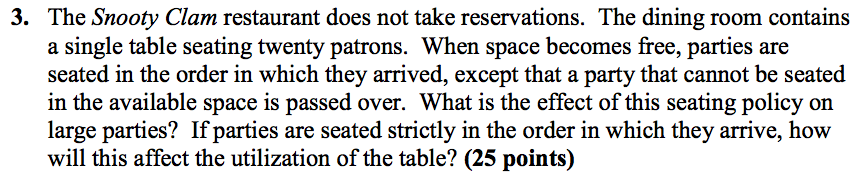
**Disadvantage**

1. Context switching will now have a larger effect than it did before.

2. Removing processes from the running queue is now significantly harder, as

you would have to search through the whole list.

c] Allow the quantum time each process gets to be changed on a per process basis. Allot a longer amount of time to processes deserving higher priority. (have two or more quantum’s possible in the algorithm)



**Problem Statement Decoded:**

In the problem statement, There is a single table with twenty patrons. If there is space on the table when “parties” (or a group of people as they term in the problem) enter, they can occupy the patrons and when they leave they another party can occupy the space. A party cannot occupy the space only if the party size is more than the patrons that are available in that case they will have to wait. And from the statement they strictly follow the First Come First Serve order (The party which enters first are served first).

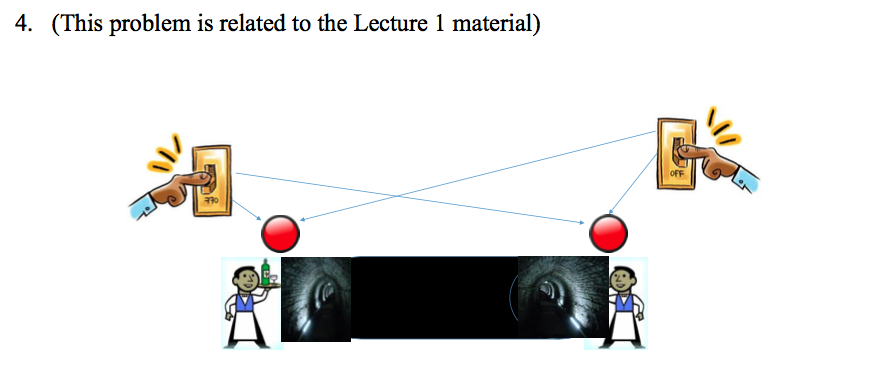
**Problem**:

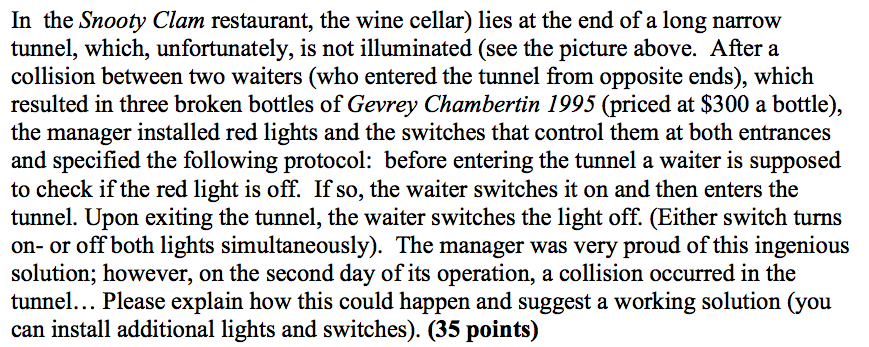
**What is the effect of this seating policy on large parties?**

The issue with this type of arrangement is parties have to wait if there are no patrons available. (For eg, if a party of 5 come and there are only 2 patrons empty, they have to wait, but if a group of party of 3 comes, they will have to wait too. Now in such case if one patron gets empty, the party of three can be seated but the party of 5 still have to wait) This kind of problem can keep on occurring and the party of 5 will have to wait forever if it does not find patrons available for it. This issue is basically termed as Starvation in Computing terms for which a process has to wait forever to utilize the resource.

**If parties are seated strictly in the order in which they arrive, how will this affect the utilization of the table?**

The question states, what might happen if the parties are seated strictly according to their order in which they arrive, that is FIRST COME FIRST SERVE (FCFS)basis. The issue with methodology is that if there is a very large party that arrives and there is no seating available, it will have to wait. If one more party arrives after it even though with a smaller quantity, in such a case, even if there is seating available on the patrons, they might not be able to sit since FCFS is followed here and it causes a poor utilization of the Table.





**Problems Occur when:**

#1: Both WAITERS have turned on the light at the same time

Explanation:

A collision between two waiters is possible in below scenario:

1. Waiter Number 1 checks if light is switched off.
2. Waiter Number 1 finds that the light is off. He then switches it on and enters the tunnel.
3. Waiter Number 2 at the other end of the tunnel checks if the lights off.
4. Waiter Number 2 finds that light is on and he then enters the tunnel.

Since, both the waiters are into the tunnel at the same time collision occurs between them.

**SOLUTION:**

This problem can be solved by adding another condition in the solution in which the waiter has to wait if the red light is turned ON

Implementation of the modified problem for resolving the collision after the installation of light

Void Restaurant (){

While(true){

Wait(waiter) //Check for the below condition

If (red\_light == 1)

/\* Enter wait state & wait for waiter to exit \*/

Else

Mutex(waiter) /\* Acquire the lock and enter \*/

/\* Enter the tunnel from one of the entrance and go to the wine cellar \*/

Retrieve wine and exit tunnel \*/

Signal(Mutex) /\* Release lock \*/

Signal(Waiter) /\* Signal waiter that tunnel is free to enter \*/

}

}