SOLUTION

Assignment 2 (Chapters 5, 6, 7)
Operating Systems (ENGR 3950U / CSCI 3020U), Fall 2012
Due: November 30, 2012 (11:59pm) via Blackboard
Instructor: Dr. Kamran Sartipi

1st part (Chapter5 – CPU Scheduling)

Question 1 [15 marks]. Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

The processes are assumed to have arrived in the order P1, P2, P3, P4, P5, all at time 0.

a. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, nonpreemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 1).

b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

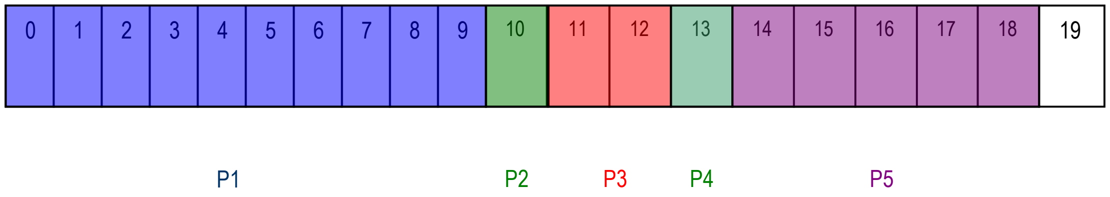
c. What is the waiting time of each process for each of the scheduling algorithms?

d. Which of the algorithms results in the minimal average waiting time (over all processes)?

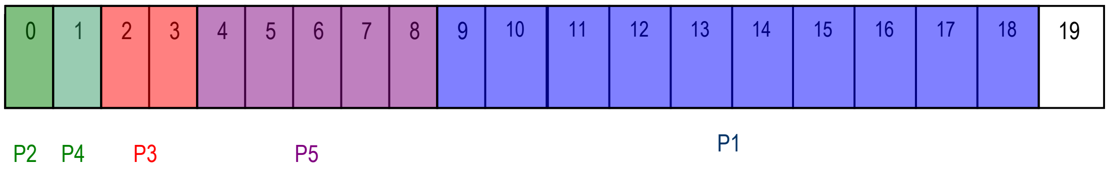
Answer:

a. four Gantt charts are as follow

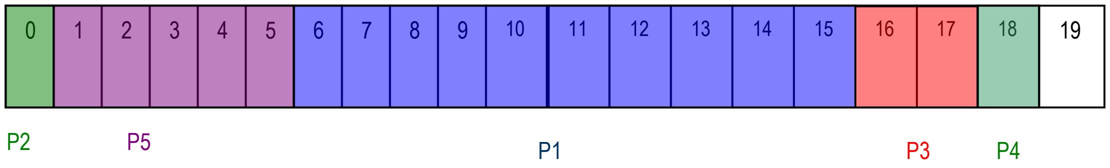
FCFS



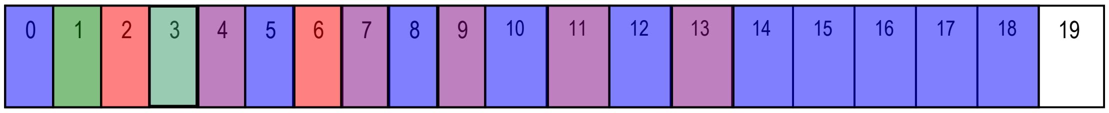
SJF



Non-preemptive
Priority



RR Quantum



b. Turnaround time

	FCFS	RR	SJF	Priority
P1	10	19	19	16
P2	11	2	1	1
P3	13	7	4	18
P4	14	4	2	19
P5	19	14	9	6

c. Waiting time (turnaround time minus burst time)

	FCFS	RR	SJF	Priority
*P*1	0	9	9	6
*P*2	10	1	0	0
*P*3	11	5	2	16
*P*4	13	3	1	18
*P*5	14	9	4	1

d. Shortest Job First (3.2 ms).

The detailed procedure:

(1) FCFS

P1	10
P2	10+1 = 11
P3	11+2 = 13
P4	13+1 = 14
P5	14+5= 19

Finish Time = (Start-time + Burst-time)
Turnaround Time = Finish Time (since Arrival Time = 0)
Waiting time = turnaround time - burst time

(2) SJF:

P2	1
P4	1+1 = 2
P3	2+2 = 4
P5	4+5 = 9
P1	9+10 = 19

(3) Nonpreemptive priority:

P2	1
P5	1+5 = 6
P1	6+10 = 16
P3	16+2 = 18
P4	18+1 = 19

(4) RR(q=1)

Question 2 [7 marks]. Which of the following scheduling algorithms could result in starvation?

a. First-come, first-served
b. Shortest job first
c. Round robin
d. Priority5

Answer:
Priority-based scheduling algorithms and the shortest job first algorithm could result in starvation, since low priority processes may never execute. Shortest job first algorithm is a kind of priority scheduling algorithm where priority is the predicted next CPU burst time (larger CPU burst, lower priority).

Question 3 [10 marks]. Consider a variant of the RR scheduling algorithm in which the entries in the ready queue are pointers to the PCBs.

a. What would be the effect of putting two pointers to the same process in the ready queue?
b. What would be two major advantages and two disadvantages of this scheme?
c. How would you modify the basic RR algorithm to achieve the same effect without the duplicate pointers?

Answer:
a. In effect, that process's priority will be increased. During one round, the process will appear twice in the ready queue and hence will be scheduled twice as often as other processes.

b. The advantage is that 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. Another disadvantage is that the scheduling requires more overheads in maintaining pointers.

c. Allot a longer amount of time to processes deserving higher priority, in other words, have two or more quantums possible in the round-robin scheme. (Adaptive quantums for each process. A higher priority process can use up to 2/3/4/etc.)

Question 4 [8 marks]. Explain the differences in how much the following scheduling algorithms discriminate in favor of short processes:

a. FCFS
b. RR
c. Multilevel feedback queues

Answer:

a. FCFS – discriminates against short jobs since any short jobs arriving after long jobs will have a longer waiting time (i.e., convoy effect).

b. RR – treats all jobs equally (giving them equal bursts of CPU time) so short jobs will be able to leave the system faster since they will finish first.
c. Multilevel feedback queues – work similar to the RR algorithm — they discriminate favorably toward short jobs.

Question 5 [10 marks]. The traditional UNIX scheduler enforces an inverse relationship between priority numbers and priorities: The higher the number, the lower the priority. The scheduler recalculates process priorities once per second using the following function:

Priority = (Recent CPU usage / 2) + Base

where Base = 60 and recent CPU usage refers to a value indicating how often a process has used the CPU since priorities were last recalculated.

Assume that recent CPU usage for process P1 is 40, for process P2 is 18 and for process P3 is 10. What will be the new priorities for these three processes when priorities are recalculated? Based on this information, does the traditional UNIX scheduler raise or lower the relative priority of a CPU-bound process?

Answer:
The priorities assigned to the processes are 80, 69, and 65 respectively. The scheduler lowers the relative priority of CPU-bound processes.

2nd part (Chapter6 – Process Synchronization)

Question 6 [15 marks]. The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P_0 and P_1, share the following variables:

```
 boolean flag[2] ;    /* initially false */
int turn;
do {
   flag[i] = TRUE;

   while (flag[j]) {
       if (turn == j) {
          flag[i] = FALSE;
          while (turn == j)
               ;  // do nothing
           flag[i] = TRUE;
       }
   }
      // critical section
   turn = j;
   flag[i] = FALSE;

      // reminder section
} while (TRUE)
```

The structure of process P_i (i == 0 or 1) is shown above; the other process is P_j (j == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

Answer:
To prove that this algorithm provides a valid solution to the problem of critical section problem, we need to show that the following three conditions are held: (1) **mutual exclusion is preserved**, (2) **progress requirement is satisfied**, and (3) **bounded-waiting requirement is met**.

To investigate that condition (1) holds: If both Pi and Pj intend to enter their critical sections, both set their flags to true and pass the "while (flag[x]{" statement (assume x=i or j). Then both check the variable "turn" which can be either "i" or "j". If "turn = j" then Pi gives up (changes its flag to false and will wait in the inner while loop) and Pj enters its critical section. In case "turn = i" then Pj gives up (similar to above for Pi) and Pi enters its critcal section. Hence mutual-exclustion condition holds.

To investigate that condition (2) holds: If Pj does not intend to enter its critical section (i.e., flag[j] = false) Pi checks flag[j] in its entry-guard for critical section and can enter its critical section several times one after the other. Therefore, Pj which is outside its critical section does not prevent Pi to enter its critical section. Hence the progress condition holds.

To investigate that condition (3) holds: As we mentioned above, if Pj does not want to enter its critical section, Pi can enter its critical section several times one after the other (condition 2 holds). However, if Pj also wants to enter its critical section, then at most Pi can enter its critical section once more before Pj will be able to enter its critical section. In this case, initially Pj is prevented to enter its critical section and is stuck in the inner while loop with the conditions flag[i] == true and turn == i. However, when Pi exits its critical section it changes the turn (turn = j) and resets its flag (flag[i] = false). Pj is now released from the inner while-loop and enters its critical section immediately after Pi leaves its critical section. Therefore, condition 3 holds.

Question 7 [7 marks]. What is the meaning of the term busy waiting? What other kinds of waiting are there? Can busy waiting be avoided altogether? Explain your answer.

Answer:
(1) Busy waiting: the situation that a process is waiting for an event while it is executing instructions in a loop in the running state. In software implementation of the critical section solutions, while a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry guard, essentially doing nothing useful, until whatever is being waited on occurs (for example, until a semaphore variable changes value). This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively.

(2) Rather than busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then, control is transferred to the CPU scheduler, which selects another process to execute.

(3) Since disabling interrupts is not a good solution in a multiprocessor environment, some busy waiting is required in most implementations of semaphores. However, this is a small amount of busy waiting in comparison to waiting for a prolonged period in the critical section of a program.

Question 8 [8 marks]. Show that, if the wait() and signal() semaphore operations are not executed atomically, then mutual exclusion may be violated.

Answer:
A wait(S) operation atomically decrements the value associated with a semaphore "S". Suppose two processes P1 and P2 simultaneously perform wait(S) operations (S=1) to enter their critical sections. If wait(S) operations are not performed atomically, then it is possible that the code for wait(S) that are executed by two processess get interleaved which causes that both processes proceed and enter their critical sections at the same time, violateing mutual exclusion condition.

For example, suppose the value of semaphore S = 1 and processes P1 and P2 execute wait(S) concurrently.

a.
Time t0: P1 determines that value of S =1
b.
Time t1: P2 determines that value of S =1
c.
Time t2: P1 decrements S by 1 and enters critical section
d.
Time t3: P2 decrements S by 1 and enters critical section

3rd part (Chapter7 – Deadlocks)

Question 9 [5 marks]. Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock-free.

Answer:
Suppose the system is in deadlock situation. This implies that each of the three processes is holding one resource and is waiting for another resource which is held by one of the other two processes. Since there are three processes and four resources, one process must be able to obtain two resources. This process requires no more resources; therefore it will eventually terminate and returns its two resources back which can be used by other two processes to execute and terminate.

Question 10 [15 marks]. Consider the following snapshot of a system:

	Allocation	Max	Available
	ABCD	ABCD	ABCD
P0	0012	0012	1520
P1	1000	1750	
P2	1354	2356	
P3	0632	0652	
P4	0014	0656	

Answer the following questions using the banker's algorithm:

a. What is the content of the matrix Need?

b. Is the system in a safe state?

c. If a request from process P1 arrives for (0, 4, 2, 0), can the request be granted immediately?

Answer:

a. Since NEED = MAX - ALLOCATION, the content of NEED is as follows:

A B C D
0 0 0 0
0 7 5 0
1 0 0 2
0 0 2 0
0 6 4 2

b. With *Available* being equal to (1, 5, 2, 0), both process P0 (needs no resources) and P3 will terminate and release their allocated resources. The new available resources (1,5,2,0 + 0,0,1,2 + 0,6,3,2 = 1, 11, 6, 4) are sufficient for other processes to terminate. For more detailed solutions, pleae refer to the course slides 25 to 29 of week 10 (Mar 9).

c. Yes it can. This results in the value of *Available* being (1, 1, 0, 0).

	Allocation	**Max**	**Need**	**Available**
	ABCD	**ABCD**	**ABCD**	**ABCD**
P0	**0012**	**0012**	**0 0 0 0**	**1 1 0 0**
P1	**1420**	**1750**	**0 3 3 0**	
P2	**1354**	**2356**	**1 0 0 2**	
P3	**0632**	**0652**	**0 0 2 0**	
P4	**0014**	**0656**	**0 6 4 2**	

Work := Available = 1 1 0 0
Work := Work + Allocation
P0 => Work = 1 1 0 0 + 0 0 1 2 = 1 1 1 2
P2 => Work = 1 1 1 2 + 1 3 5 4 = 2 4 6 6
P3 => Work = 2 4 6 6 + 0 6 3 2 = 2 10 9 8
P4 => Work = 1 10 9 8 + 0 0 1 4 = 1 10 10 12
P1 => Work = 1 10 10 12 + 1 4 2 0 = 2 14 12 12

Safe sequence <P0, P2, P3, P4, P1>