CS31202/CS39002 Operating Systems, Spring 2023–2024

Class Test 1

06–February–2024 06:30pm–07:30pm Maximum marks: 25

Roll no: _____ **Name:** _____

[*Write your answers in the question paper itself. Be brief and precise. Answer all questions.*]

1. Three processes P_0, P_1, and P_2 are running concurrently on a single-CPU system, where P_2 has the highest priority. Currently, Process P_1 is in the running state, P_0 is waiting in the ready queue, and P_2 is doing an I/O operation. After some time, P_1 terminates and P_2 completes I/O, simultaneously. Clearly list step by step **all** the events that will take place in the system, before the next process gets allocated to the CPU. Assume that the system implements a preemptive priority CPU-scheduler. **(6)**

Solution The steps are as follows.

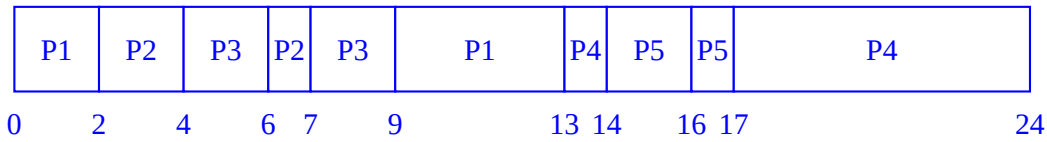
1. The resources allocated to P_1 are released, and the PCB for P_1 is removed from the process table.
2. The state in the PCB of P_2 is changed from *waiting* to *ready*.
3. Now, there are two processes P_0 and P_2 in the ready state. The scheduler finds that P_2 has higher priority than P_0.
4. The scheduler changes the state in the PCB of P_2 from *ready* to *running*.
5. The scheduler restores the context (registers) of P_2 from its PCB to the CPU.
6. The scheduler loads the PC (program counter) of the CPU by the saved PC value in the PCB of P_2.

2. Consider a multi-level queue scheduling setup with one CPU and two ready queues Q_0 and Q_1. Q_0 and Q_1 are assigned to store the foreground processes and the background processes, respectively. The Round Robin scheduling is implemented for queue Q_0 with quantum = 2, whereas FCFS scheduling is implemented for queue Q_1. Preemptive priority scheduling is followed across the two ready queues, with Q_0 having the higher priority. Consider the five processes given in the following table. All times are in milliseconds (ms).

Process	Arrival Time	CPU Burst time	Type
P1	0	6	Background
P2	2	3	Foreground
P3	3	4	Foreground
P4	4	8	Background
P5	14	3	Foreground

(a) Draw the Gantt chart in this case, and calculate the average waiting times of all the processes. There is no need to consider context-switch times in the Gantt chart. **(7)**

Solution The Gantt chart is given below.



From this chart, we get the following waiting times.

P1: $13 - 0 - 6 = 7$

P2: $7 - 2 - 3 = 2$

P3: $9 - 3 - 4 = 2$

P4: $24 - 4 - 8 = 12$

P5: $17 - 14 - 3 = 0$

(b) Now, assume that the scheduler takes 0.2ms of CPU time in context switching for each completed job, and 0.1ms of **additional** CPU time for saving the context of each incomplete job. Neglect the time for scheduling the same foreground process after its time quantum is over. Calculate the percentage of CPU time that gets wasted for the scheduling overhead, in this example. Ignore, in your calculations, the time for the first scheduling, and the time after the last process finishes. **(5)**

Solution The scheduling overheads at the context-switch instances are as follows.

	0		2		4		6	7		9		13	14		16	17		24
Scheduling Overhead	0		0.3		0.3		0.3	0.2		0.2		0.2	0.3		0	0.2		0

The total scheduling overhead is 2.0, and the total CPU time is $24+2=26$. Therefore the percentage of CPU time wasted for scheduling is $(2/26)\times 100 \approx 7.69\%$.

3. Two cooperative processes P_0 and P_1 are running concurrently, and share a buffer capable of storing only one item. P_0 keeps on producing items, but multiple items cannot be stored in the buffer. So the other process P_1 must read each item from the buffer before P_0 writes the next item to the buffer. Writing an item to the buffer by P_0 and reading an item from the buffer by P_1 should be mutually exclusive. In order to guard their critical sections and to alternate their turns, the processes use a variant of Peterson's algorithm, shown on the next page. The algorithm is given for P_i ($i=0$ or 1). The other process is called P_j, where $j=1-i$.

Assume that the compiler or the hardware makes no instruction swaps. Prove/Disprove: This algorithm enforces mutual exclusion of the critical sections of the two processes. If mutual exclusion is guaranteed, establish additionally that the turns of the two processes actually alternate. If mutual exclusion is not guaranteed, give an explicit situation where both the processes may be in their respective critical sections simultaneously (concurrently) at the same time. **(7)**

```
shared boolean flag[2];   /* Initialized to {false, false} */
shared int turn;          /* Initialized to 0 (the producer produces first) */

while (1) {
   /* Entry section */
   flag[i] = true;
   while ((flag[j] == true) && (turn == j)) { }   /* Busy wait */

   /* Critical section */
   . . .

   /* Exit section */
   flag[i] = false;
   turn = j;

   /* Remainder section */
   . . .
}
```

Solution This algorithm will not provide mutual exclusion. It may lead to a situation as in the original Peterson algorithm with the first two instructions swapped. In order to see that this may happen, consider the following sequence of events. Let us assume that we have only one CPU.

1. P_0 is running in its remainder section (that is, after producing an item). P_1 is preempted in its critical section (that is, while consuming that item). At this point, we have flag[0] = false, turn = 1, flag[1] = true.
2. P_0 is preempted in its remainder section, and P_1 is rescheduled.
3. P_1 completes its critical section, exit section, and remainder section, and wants to enter its critical section again. At this point, we have flag[0] = false, turn = 0, flag[1] = true. Since flag[0] = false, the while loop in the entry section of P_1 breaks. So P_1 enters its critical section, and is again preempted.
4. P_0 is rescheduled, completes its remainder section, and goes back to the top of the entry section. It sets flag[0] = true, and sees turn = 0, flag[1] = true. Therefore the while loop in the entry section of P_0 breaks, and P_0 too enters its critical section.