

School of Computer Engineering
KIIT Deemed to be University, Bhubaneswar-751024
Operating Systems
[CS3009]

Full Mark: 20

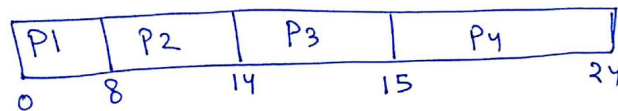
1.	Answer all questions.	
a)	<p>What resources are used when a thread is created? How do they differ from those used when a process is created?</p> <p>Ans: Because a thread is smaller than a process, thread creation typically uses fewer resources than process creation. Creating a process requires allocating a process control block (PCB), a rather large data structure. The PCB includes a memory map, list of open files, and environment variables. Allocating and managing the memory map is typically the most time-consuming activity. Creating either a user or kernel thread involves allocating a small data structure to hold program counter, register set, stack, and priority.</p>	[1]
b)	<p>What is the necessity of context switching? Why is it considered as an overhead for the system?</p> <p>Ans: A context switch is the mechanism by which CPU switches to a new process before finishing execution of the current process. The necessity arises because of i) <u>arrival of high priority process</u>, ii) <u>completion of time quantum in RR scheduling</u>.</p> <p>During context switching CPU stores and restores the state or context of the CPU, which is an overhead to the system, and does not contribute any useful work. Context switching is overhead because it is cycles (time) that the processor is being used but no user code is executing, so no directly productive computing is getting done.</p>	[1]
c)	<p>Consider following concurrent processes P0 and P1 using an integer shared variable i =10.</p> <div style="display: flex; justify-content: space-around; margin-top: 20px;"> <div style="text-align: center;">P0</div> <div style="text-align: center;">P1</div> </div>	

		<div><div>begin</div><div><div>A</div></div><div>i++;</div><div>printf("%d",i);</div><div>end</div></div> <div><div>begin</div><div>printf("%d",i);</div><div><div>B</div></div><div>end</div></div>																					
		<p>Write the code for A and B using wait() and signal() operations such that output will be 10 11. (i.e. 10 will be printed prior to 11)</p> <p>Ans: Semaphore S = 0; A = wait(S); B = signal(S);</p>	[1]																				
	d)	<p>Suppose a short term scheduler uses an algorithm that selects processes which have used least processor time in recent past. Justify that the algorithm favors I/O bound processes without starvation to CPU bound processes.</p> <p>Ans: It will favor the I/O-bound programs because of the relatively short CPU burst request by them; however, the CPU-bound programs will not starve because the I/O-bound programs will relinquish the CPU relatively often to do their I/O.</p>	[1]																				
	e)	<p>A system has 10 identical resources and N processes competing for them. Each process can request at most 2 resources. Find the minimum value of N for which the system may lead to deadlock.</p> <p>Ans: 10</p>	[1]																				
2.		<p>Consider the following set of processes arrive in a system.</p> <table><tr><td>Process #</td><td>CPU burst time</td><td>Priority</td><td>Arrival Time</td></tr><tr><td>P1</td><td>8</td><td>4</td><td>0</td></tr><tr><td>P2</td><td>6</td><td>1</td><td>1</td></tr><tr><td>P3</td><td>1</td><td>2</td><td>3</td></tr><tr><td>P4</td><td>9</td><td>2</td><td>6</td></tr></table> <p>Show the order of execution and calculate the Response time, Waiting time of the processes for the following scheduling algorithms.</p> <p>i. FCFS</p> <p>ii. RR (quantum = 3 ms)</p> <p>iii. Preemptive priority scheduling. (Note: consider lowest number as highest priority)</p>	Process #	CPU burst time	Priority	Arrival Time	P1	8	4	0	P2	6	1	1	P3	1	2	3	P4	9	2	6	
Process #	CPU burst time	Priority	Arrival Time																				
P1	8	4	0																				
P2	6	1	1																				
P3	1	2	3																				
P4	9	2	6																				

Ans:

(i) FCFS

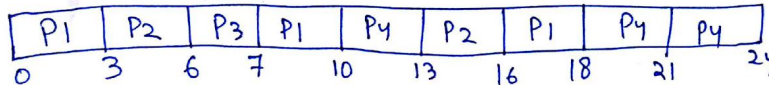
Process	BT	AT	WT	TAT	RT
P1	8	0	0	8	0
P2	6	1	7	13	7
P3	1	3	11	12	11
P4	9	6	9	18	9



— 1 mark

(ii) RR

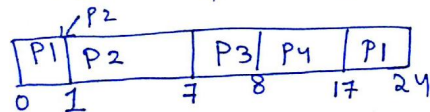
Process	BT	AT	WT	TAT	RT
P1	8 52	2 310	(0+4+6) 10	18	0
P2	6 2	1 6	(2+7) 9	15	2
P3	1 3	3	3	4	3
P4	9 62	6 13	(4+5) 9	18	4



— 2 marks

(iii) Preemptive Priority

<u>Process</u>	<u>BT</u>	<u>AT</u>	<u>Priority</u>	<u>WT</u>	<u>TAT</u>	<u>RT</u>
P1	8 7	0 1	4	0+16	24	0
P2	6	1	1	0	6	0
P3	1	3	2	4	5	4
P4	9	6	2	2	11	2



— 2 marks

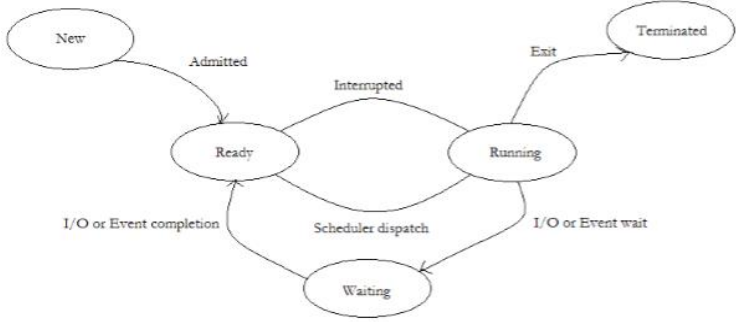
3. a) What are the various states of a process? Show and explain the transitions of states of a process by means of a diagram.

Ans:

Process can have one of the following five states at a time.

- New state: A process that just has been created but has not yet been admitted to the pool of execution processes by the operating system. Every new operation which is requested to the system is known as the new born process.

[1½]

	<p>ii. Ready state: When the process is ready to execute but he is waiting for the CPU to execute then this is called as the ready state. After completion of the input and output the process will be on ready state means the process will wait for the processor to execute.</p> <p>iii. Running state: The process that is currently being executed. When the process is running under the CPU, or when the program is executed by the CPU, then this is called as the running process and when a process is running then this will also provide us some outputs on the screen.</p> <p>iv. Waiting or blocked state: A process that cannot execute until some event occurs or an I/O completion. When a process is waiting for some input and output operations then this is called as the waiting state and in this process is not under the execution instead the process is stored out of memory and when the user will provide the input and then this will again be on ready state.</p> <p>v. Terminated state: After the completion of the process, the process will be automatically terminated by the CPU. So this is also called as the terminated state of the process. After executing the complete process the processor will also deallocate the memory which is allocated to the process. So this is called as the terminated process.</p>  <p style="text-align: center;">Fig: Process state transition diagram.</p>	[1½]												
b)	<p>Consider three concurrent processes as P0, P1 and P2 and three semaphore variables as S0, S1 and S2. The semaphores are initialized as S0=1, S1=0 and S2=0.</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="padding: 5px;">P0</th><th style="padding: 5px;">P1</th><th style="padding: 5px;">P2</th></tr> </thead> <tbody> <tr> <td style="padding: 5px;">do{ printf("0"); printf("0") }while(1);</td><td style="padding: 5px;">do{ printf("2"); printf("2") }while(1);</td><td style="padding: 5px;">do{ printf("1"); printf("1") }while(1);</td></tr> </tbody> </table> <p>Use wait() and signal() operations on the semaphore variables in the above processes such that the following string will be printed. Justify your solution. 002211002211.....</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="padding: 5px;">P0</th><th style="padding: 5px;">P1</th><th style="padding: 5px;">P2</th></tr> </thead> <tbody> <tr> <td style="padding: 5px;">do{ wait(S0); printf("0"); printf("0"); signal(S1); }while(1);</td><td style="padding: 5px;">do{ wait(S1); printf("2"); printf("2"); signal(S2); }while(1);</td><td style="padding: 5px;">do{ wait(S2); printf("1"); printf("1"); signal(S0); }while(1);</td></tr> </tbody> </table>	P0	P1	P2	do{ printf("0"); printf("0") }while(1);	do{ printf("2"); printf("2") }while(1);	do{ printf("1"); printf("1") }while(1);	P0	P1	P2	do{ wait(S0); printf("0"); printf("0"); signal(S1); }while(1);	do{ wait(S1); printf("2"); printf("2"); signal(S2); }while(1);	do{ wait(S2); printf("1"); printf("1"); signal(S0); }while(1);	[2]
P0	P1	P2												
do{ printf("0"); printf("0") }while(1);	do{ printf("2"); printf("2") }while(1);	do{ printf("1"); printf("1") }while(1);												
P0	P1	P2												
do{ wait(S0); printf("0"); printf("0"); signal(S1); }while(1);	do{ wait(S1); printf("2"); printf("2"); signal(S2); }while(1);	do{ wait(S2); printf("1"); printf("1"); signal(S0); }while(1);												

4.	a)	<p>Consider a producer/consumer problem with a bounded buffer of size 10. Write a semaphore based solution to the problem with a condition that the consumer always retrieves the oldest item.</p> <p>Ans: Semaphore Full=0, Empty=10, mutex=1; int in=0, out=1; // Assumption, array index starts from 1</p> <pre>void Producer() { do{ wait(Empty); wait(mutex); in=in%10+1; buffer[in]=item; signal(mutex); signal(Full); }while(1); }</pre> <pre>void Consumer() { do{ wait(Full); wait(mutex); item=buffer[out]; out=out%10+1; signal(mutex); signal(Empty); }while(1); }</pre>	<div>[1½]</div> <div>[1½]</div>																		
	b)	<p>Show that, if the wait() and signal() semaphore operations are not executed atomically, then mutual exclusion may be violated.</p> <p>Ans: A wait() operation atomically decrements the value associated with a semaphore. If two wait() operations are executed on a semaphore when its value is 1, if the two operations are not performed atomically, then it is possible that both operations might proceed to decrement the semaphore value, thereby violating mutual exclusion.</p> <table><tr><td>Pi</td><td>Pj</td></tr><tr><td>wait(semaphore S) { while(s<=0); s=s-1; }</td><td>wait(semaphore S) { while(s<=0); s=s-1; }</td></tr></table>	Pi	Pj	wait(semaphore S) { while(s<=0); s=s-1; }	wait(semaphore S) { while(s<=0); s=s-1; }	[2]														
Pi	Pj																				
wait(semaphore S) { while(s<=0); s=s-1; }	wait(semaphore S) { while(s<=0); s=s-1; }																				
5.	a)	<p>Consider the set of 5 processes whose arrival time and burst time are given below.</p> <table><tr><td>Process id</td><td>Arrival time</td><td>Burst time</td></tr><tr><td>P1</td><td>0</td><td>3</td></tr><tr><td>P2</td><td>1</td><td>5</td></tr><tr><td>P3</td><td>2</td><td>1</td></tr><tr><td>P4</td><td>2</td><td>4</td></tr><tr><td>P5</td><td>4</td><td>5</td></tr></table> <p>If the CPU scheduling policy is FCFS and context switching time is 1 time unit, then,</p>	Process id	Arrival time	Burst time	P1	0	3	P2	1	5	P3	2	1	P4	2	4	P5	4	5	
Process id	Arrival time	Burst time																			
P1	0	3																			
P2	1	5																			
P3	2	1																			
P4	2	4																			
P5	4	5																			

	<p>i. Find the CPU utilization.</p> <p>ii. What will be the CPU Utilization in case the context switching time is 2 time units? (Hint: CPU Utilization is the percentage of time the CPU is busy in execution)</p> <p>Ans:</p> <p>(i) Total burst time = 3+5+1+4+5=18 units Total number of context switch possible is 4. Each context switch takes 1 time unit. So, total context switch time = $4 \times 1 = 4$ time units Total time = Burst time + context switch time = 18 + 4 = 22 time units</p> $\text{CPU utilization} = \frac{18}{22} \times 100 = 81.81\%$ <p>(ii) If each context switch time is 2 time units, then total time unit required for 4 context switch is $4 \times 2 = 8$ time units. Total time = Burst time + context switch time = 18 + 8 = 26 time units</p> $\text{CPU utilization} = \frac{18}{26} \times 100 = 69.23\%$	<p>[1½]</p> <p>[1½]</p>
b)	<p>What are the conditions for deadlock? How can the operating system prevent deadlock?</p> <p>Ans: There are four conditions that are necessary to achieve deadlock:</p> <ol style="list-style-type: none"> Mutual Exclusion: At least one resource must be held in a non-sharable mode; If any other process requests this resource, then that process must wait for the resource to be released. Hold and Wait: A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process. No preemption: Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it. Circular Wait: A set of processes { P₀, P₁, P₂, . . . , P_N } must exist such that every P_i is waiting for P_[(i + 1) % (N + 1)]. (Note that this condition implies the hold-and-wait condition, but it is easier to deal with the conditions if the four are considered separately.) <p>Deadlocks can be prevented by preventing at least one of the four required conditions:</p> <ol style="list-style-type: none"> Mutual Exclusion: Shared resources such as read-only files do not lead to deadlocks. Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process. Hold and Wait: To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. There are several possibilities for this: Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later. Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it. Either of the methods described above can lead to starvation if a process requires one or more popular resources. No Preemption: Preemption of process resource allocations can prevent this condition of 	<p>[1]</p> <p>[1]</p>

	<p>deadlocks, when it is possible. One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, (preempted), forcing this process to re-acquire the old resources along with the new resources in a single request, similar to the previous discussion. Another approach is that when a resource is requested and not available, then the system looks to see what other processes currently have those resources and are themselves blocked waiting for some other resource. If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting. Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives.</p> <p>iv. Circular Wait: One way to avoid circular wait is to number all resources, and to require that processes request resources only in strictly increasing (or decreasing) order. In other words, in order to request resource R_j, a process must first release all R_i such that $i \geq j$. One big challenge in this scheme is determining the relative ordering of the different resources</p>	
--	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--