OS THEORY ASSIGNMENT - 1

19CS30048

SUHAS JAIN

Ans 1) User mode to kernel mode transition can happen in following ways :-

1) System Call

2) Timer Interrupt

3) Exception.

Ans 2) Advantages :-

1) User level threads package can be implemented on an operating system that does not support threads.

2) Creating a thread, switching between threads, and synchronisation between threads can be done without interruption of the kernel hence reducing content switching overhead.

Disadvantages :-

1) User level threads require non-blocking system calls (multithreaded kernel). Otherwise entire process will be blocked in the kernel, even if there are runnable threads left. Eg - If one thread causes a page fault, the process blocks.

Ans 3) We assume that a response is produced at the end of a CPU burst.

(a) <u>For first request</u> :-

Response time for i^{th} process

$= 53 \times i$ msec.

$i = 1 \ldots\ldots 10$

$$\text{Average response time} = \frac{53 \times \sum_{i=1}^{10} i}{10}$$

$= 53 \times 5.5$

$= 291.5$ msec.

<u>For subsequent requests</u>:

Response Time $= 10 \times 53 - 200$

$= 330$ msec.

(b) For $\delta = 20$ msec a subrequest will be preempted after 20 msec. Then it will be scheduled for ~~second~~ second and third time.

<u>For first request</u> :-

Response time $= 2 \times 10 \times 23 + 13 \times n$ msec

Average response time $= 531.5$ ms

For subsequent requests:-

Response time = $2 \times 10 \times 23 + 10 \times 13 - 200$

$= 390$ msec.

Ans 4)

(1) Ready state to running state.
Yes it is possible when a process is selected by the scheduler from ready queue.

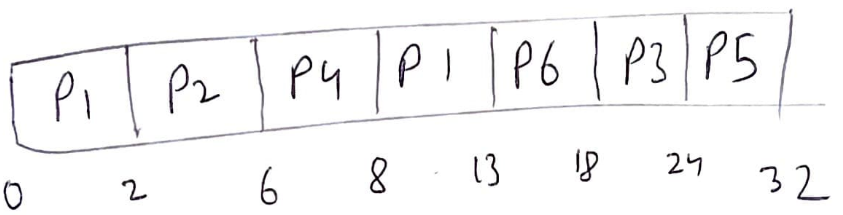
(2) Running state to ready state.- Yes
When timer interrupts a process the process is again sent to the ready queue. Eg. in round robin.

(3) Running state to waiting state. - Yes
Process moves from running state to waiting if it needs to wait for a resource such as user input.

(4) Ready state to waiting state.
No, not possible
Process cannot wait for some resource without starting the execution. So it will move to ~~ready~~ running state first.

5) Waiting state to ready state
Yes, it is possible.
When I/O event a process is waiting for is completed it moves back into the ready state.

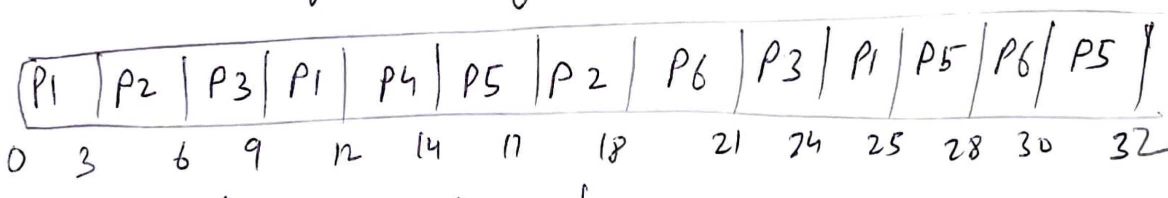
Ans 5) (i) Preemptive SJF



$$\text{Average waiting time} = \frac{6+0+15+1+18+5}{6}$$

$= 7.5$ msec.

(2) Round Robin Sheduling with time quantum of 3 msec.



$$\text{Average waiting time} = \frac{18+12+15+7+18+17}{6}$$

$= 14.5$ msec.

Ans6) There are 2 modes of operations of an operating system:-

1) User mode
2) Kernel Mode

User mode is normal mode where the process has limited access. Kernel mode is the privileged mode when process as unrestricted access to system resources.

The operating system enters "Kernel Mode" for ~~a~~ executing system calls.

Need for different modes:-

1) ~~[illegible]~~ Helps in protecting the OS from errant users and errant users from one another. This is done as ~~[illegible]~~ ~~[illegible]~~ priviledged instructions cannot be executed in user mode and thus errant user cannot run harmful machine instructions.

2) The kernel performs tasks like process management, memory management etc which should be hidden from the user. If any of this is altered by an user then it might lead to system wide failures and ~~[illegible]~~ deadlocks.

Ans 7)

1) True
Systems calls include process creation, management, file access, memory management etc ~~[illegible]~~ which are privileged instructions and hence need to be executed in ~~[illegible]~~ kernel mode.

2) True
This can happen because of a timer interrupt. The scheduler might want to switch to a different program ~~[illegible]~~ before current program finishes execution. This is possible in ~~[illegible]~~ preemptive ~~[illegible]~~ scheduling algorithms.

3) False.
There ~~[illegible]~~ are multiple threads of the same process which share code and data segments. If the heap is not shared, resource sharing will become slow. ~~[illegible]~~ Sharing heap ensures better efficiency than using pipes or shared memory.

4) True

If the stack is shared then there would be many concurrent modifications and there won't be any distinction between threads of the process. Usually stacks can be used in much faster way than heaps and using seperate stacks ensures speed.

Ans 8)

a) If P is the parent process it will create C_1 [illegible] after first fork(). Then C_1 [illegible] and P each will create ~~2~~ 1 more child process and so on.

So processes double every iteration

Total processes in the end = 2^6

= $\boxed{64}$

1 parent process + 63 new child processes.

(b) Parent process P will create C_1

C_1 will be forked 2 times

P will be forked only once.

$$2^2 + 2 = 6$$

Ans 9)

```
int shmid = shmget (IPC_PRIVATE, 100, 0666 | IPC_CREAT);
sh_mem = shmat ( shmid, (void *) 0 , 0);
```