Part – 1 Theory

Question 1: What are the important data items related to a process that are maintained by the kernel for process management?

It could contain process state, program counter, CPU registers, CPU-scheduling information, memory-management information, inter process communication information, process structuring information, IO status information, and accounting information, pending signals, signals masked, process context.

Question 2: For each of the following four cases, identify the conditions under which the scheduler will change the status of a process:

a. Running to Ready time quantum expired

b. Swapped to Running

Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers and transfers control to the selected process

c. Running to Waiting (Blocked) a service of the OS requires a wait/ I/O request must wait for the result.

d. Ready or Waiting to Swapped When there are no blocked processes and must free up memory for performance reasons

Question 3: When a UNIX process executes fork(), does the child process inherit

a. any pending signals of the parent?

b. the signal handlers of the parent process?

c. the signal mask of the parent?

a. No
b. Yes
c. Yes

Question 4: Why is a separate stack in the kernel memory space used for handling system call functions and interrupt handlers for a process, instead of using the process stack?

The reason for having a separate kernel stack is that the kernel needs a place to store information where user-mode code can't touch it. That prevents user-mode

code running in a different thread/process from accidentally affecting execution of the kernel. There has to be a separate place for each process to hold its set of saved registers each process also needs its own kernel stack, to work as its execution stack when it is executing in the kernel. For example, if a process is doing a read system call, it is executing the kernel code for read, and needs a stack to do this. It could block on user input, and give up the CPU, but that whole execution environment held on the stack (and in the saved CPU state in the process table entry) has to be saved for its later use. Another process could run meanwhile and do its own system call, and then it needs its own kernel stack, separate from that blocked reader's stack, to support its own kernel execution. The kernel stack is also used for interrupt handler execution, for the interrupts that occur while a particular thread is running. the interrupts are almost always doing something for another, blocked process/thread.