**What is a process and a thread?**

Process is a term that we use to describe an instance of a computer program that is being executed. Process consists of the program code and its current activity. Process varies in OSes. Process may be made up of multiple threads of execution that execute instructions concurrently. It also hides the messy hardware details from the user. A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history

**A *microkernel* is an operating system technique that places minimal functionality in the kernel, but instead provides that functionality through user-state servers. What is a major performance obstacle that microkernel designers must face?**

Drivers can touch only authorized I/O ports, but access to kernel calls is also controlled on a per-process basis, as is the ability to send messages to other processes. Processes can also grant limited permission for other processes to have the kernel access their address spaces. As an example, a file system can grant permission for the disk driver to let the kernel put a newly read-in disk block at a specific address within the file system’s address space.

**A *context-switch* is when the processor (CPU) stops executing one process and begins executing another. List the necessary steps for the operating system to perform a context-switch, including what information must (typically) be saved.**

The first process state must be saved, after that, when the scheduler gets back to the execution of the first process, it can restore this state and continue. The state of the process includes all the registers that the process may be using, especially the program counter, plus any other operating system specific data that may be necessary. This data is usually stored in a data structure called a process control block (PCB), or switch frame. To switch processes, we can have the PCB for the first process so that the process must be created and saved. Sometimes, the PCBs are stored upon a per-process stack in kernel memory, which is different from the user-mode stack, or there may be some specific operating system defined data structure for this information. Because we know that the operating system has effectively suspended the execution of the first process, we can say that the OS can now load the PCB and the context of the second process. In this procedure, the program counter from the PCB is loaded, and then it allows execution to continue in the new process. New processes are selected from a queue or many queues. The priority of process and thread can influence which process continues execution, with processes of the highest priority checked first for read threads to execute.

**Briefly describe why *round robin* scheduling is appropriate for an interactive environment, while *non-preemptive priority* scheduling is not.**

With Round robin scheduling, interactive performance depends on the length of the quantum and the number of processes in the run queue. A very long quantum makes the algorithm behave very much like first come, first served scheduling since it’s very likely that a process with block or complete before the time slice is up. A small quantum lets the system cycle through processes quickly. This is wonderful for interactive processes. Unfortunately, there is an overhead to context switching and having to do so frequently increases the percentage of system time that is used on context switching rather than real work.

A big advantage of Round robin scheduling over non-preemptive schedulers is that round robin dramatically improves average response times. Round robin scheduling eliminates each task to a certain amount of time, and the operating system can ensure that it can cycle through all ready tasks. It gives each one a chance to run.

Another advantage of the Round robin scheduling is that it is fair in that every process gets an equal share of the CPU. Moreover, it is easy to implement and, if the users know the number of processes on the run queue, then the users can know the worst-case response time for a process.

**Disabling interrupts is a common method for the operating system to ensure mutual exclusion on a uniprocessor. Is it appropriate for a shared memory (MIMD) multiprocessor? Why or why not?**

Because the operating system can choose not to preempt itself, we could choose not to preempt system processes (if the OS is a client server) or we can have processes running in system mode (if the OS is self service). Forbidding preemption for system processes would prevent the problem above where x<--x+1 not being atomic crashed the printer spooler if the spooler is part of the OS. The way to prevent preemption of kernel-mode code is to disable interrupts. Indeed, disabling interrupts is often done for exactly this reason.

However, that is not sufficient all the time.

* It does not work for user-mode programs. So the Unix print spooler, which is a user-mode program would need another solution.
* We do not want to block interrupts for too long or the system will seem unresponsive.
* Disabling interrupts is insufficient if the system has several processors.
  + The main line can be executing on both processors simultaneously so interrupts are not involved.
  + One processor cannot block interrupts on the other.
* In multi-processor system, each CPU/core execute code simultaneously, so whether the current CPU has disabled interrupts has nothing to do to avoid other CPU from entering the same region.

**Disabling interrupts is inappropriate for *user mode* and is only acceptable for brief periods in kernel mode on uniprocessors. Suppose you have a processor (CPU) like the MIPS R3000 that has no *test- and-set* instruction. How would you go about providing for mutual exclusion?**

***Hint*: You do not need to detail an algorithm, only mention it by name.**

It is called Lamport’s Bakery Algorithm

**Deadlock can be prevented by negating any one (or more) of four necessary and sufficient conditions (Coffman 1971). List the four conditions, and briefly describe the consequences of negating each of them.**

There are four conditions that must hold simultaneously for there to be a deadlock. According to Coffman (1971):

**1. Mutual Exclusion Condition**

The resources involved are non-shareable.

**Explanation:** At least one resource (thread) must be held in a non-sharable mode, that is, only one process at a time claims exclusive control of the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

**2. Hold and Wait Condition**

Requesting process hold already, resources while waiting for requested resources.

**Explanation:** There must exist a process that is holding a resource already allocated to it while waiting for additional resource that are currently being held by other processes.

**3. No-Preemptive Condition**

Resources already allocated to a process cannot be preempted.

**Explanation:** Resources cannot be removed from the processes are used to completion or released voluntarily by the process holding it.

**4. Circular Wait Condition**

The processes in the system form a circular list or chain where each process in the list is waiting for a resource held by the next process in the list.

**In terms of deadlock avoidance, there are three states: *safe, unsafe* and *deadlocked.* Briefly define each of these states, and say why it is unwise to enter the *unsafe* state. Which is possible, safe->unsafe, unsafe->deadlocked, safe->deadlocked**

**Safe State** in this state, the key is to a state being safe is that there is at least one way for all users to finish.

An **unsafe state** may not necessarily lead to deadlock, it just means that we cannot guarantee that deadlock will not occur. Thus, it is possible that a system in an unsafe state may still allow all processes to complete without deadlock occurring. This is not a good state to be in because we cannot be sure that the process will be able to finish.

**Deadlock** occurs when there are multiple processes (usually two or more) are dependent on the result of another waiting process. No processes can run, release resources, or be awakened in this state. Safe->unsafe possible, unsafe->deadlocked possible, safe->deadlocked not possible

**There are many policies for variable-sized partition memory management. In a sentence or two each, describe and rank (in terms of wasted memory): *first-fit, worst-fit* and *best-fit.***

**First-fit:** In the [first fit](http://www.memorymanagement.org/glossary/f.html#term-first-fit) algorithm, the allocator keeps a list of free blocks (known as the [free list](http://www.memorymanagement.org/glossary/f.html#term-free-list)) and, on receiving a request for memory, scans along the list for the first block that is large enough to satisfy the request. If the chosen block is significantly larger than that requested, then it is usually split, and the remainder added to the list as another free block.

**Worst-fit:** The first block on the free list will always be large enough, if a large enough block is available. This approach encourages [external fragmentation](http://www.memorymanagement.org/glossary/e.html#term-external-fragmentation), but allocation is very fast. Highest amount of memory wasted (used to avoid fragmentation within memory -- where there are small chunks of memory that cannot be used for anything)

**Best-fit:** the free block with the “tightest fit” is always chosen. The fit is usually sufficiently tight that the remainder of the block is unusably small. Least amount of memory wasted.

**Explain the purpose of system calls in an operating system and give an example, explaining how your explanation applies to it.**

System calls allow user processes to request service from the kernel. For instance, the system calls allow the user to ask the kernel to do something for the user process that is not allowed to do for itself because of protection or sharing concerns. For example, user processes are allowed to access some files but not others, and have read access to some file and read/write access to others. Thus the write() system call allows a user process to ask the OS to write some data to a file. Before performing the operation, the OS verifies that the user has write access for that file. In addition, the system calls abstract away the details of the hardware and the lower level software, providing a clean, uniform, and portable application programming interface to the user processes.

**The five (basic) states that a process can be in are: *new, ready, blocked, running* and *terminated.* Describe the transitions among these states, and state when they occur.**

***Hint*: The best way to do this is to draw a diagram.**

The OS needs to know the following state of a process is in to control which utilizes the processor.

1. A *running* process has control of CPU.
2. A *ready* process is in a queue (called the *ready queue*) and will eventually get access to CPU.
3. A *blocked* process is waiting for some event. When it happens successfully the process will be ready and will join the ready queue.
4. A *new* process has just been born and will be admitted to the ready queue as soon as it is fully set up.
5. An *exit* process is finished and will be removed from memory.

A *suspend* process is swapped out and when there is room in memory it will be swapped back in.

**The shortest job first (SJF) scheduling CPU algorithm and the shortest seek-time first (SSF) disk arm scheduling algorithm have a common benefit, and a common problem. The common benefit is that they are provably optimal. In what way are they considered optimal? What is their common problem?**

Common benefit: optimal because in both cases, jobs that take the shortest time (or shortest time to get) are completed first - leaving longer, bigger processes for later. Common problem: long jobs (or jobs that take long to seek) will delay every job after them.

The difference between preemptive and non-preemptive scheduling is that whether a running process is involuntarily removed from the running state.

Mutual exclusion is when a set of processes are prevented from simultaneously accessing a shared data structure.

The difference between preemptive and non- preemptive scheduling is whether a running process is involuntarily removed from the running state.

The four conditions that must hold for deadlock to occur are mutual exclusion, hold-and-wait, circular wait, and no preemption.