# Operating Systems Basics

Modern operating systems provide two primary functions: hardware abstraction and resource management. Hardware abstraction gives software developers a common interface between their application programs and the computer's hardware so each individual programmer doesn't need to deal with the hardware's intricacies. Instead, the hardware-specific programming is developed once, in the operating system, and everyone shares.

The second function that operating systems provide is managing the computer's resources (CPU cycles, memory, and disk drive space, for example) so they can be shared efficiently among multiple applications. Like hardware abstraction, building resource management into the operating system keeps each application programmer from writing resource management code for every program.

## CPU Resource Management and Multitasking

Although some operating systems only allow one program to run at a time (most versions of MS-DOS operate this way, for example), it's more common to find operating systems managing multiple programs concurrently. Running multiple programs at once is called *multitasking* and operating systems that support it are typically called *multitasking operating systems*.

Computer programs written for multitasking operating systems themselves often contain multiple independent tasks that run concurrently. These little subprograms are called *threads* because they form a single thread of instruction execution within the program. Threads each have their own set of CPU register values, called a *context*, but can share the same memory address space with other threads in the same program. A group of threads that share a common memory space, share a common purpose, and collectively control a set of operating system resources is called a *process*. On operating systems and CPUs that support virtual memory, each process might run in a separate address space that is protected from other processes.

Because a processor can execute instructions for only one program at a time, the operating system must manage which set of program instructions (which thread) is allowed to run. Deciding which process should run is called *scheduling* and is usually performed by a core piece of the operating system called the *kernel*. An operating system can use one of several methods to schedule threads, depending on the type of applications the operating system has been optimized to support. Different types of applications (batch, interactive, transactional, real-time, and others) have different CPU utilization characteristics, and their overall performance is affected by the scheduling method used.

The simplest scheduling method is to assign each thread to the processor in the order its run request is received and let each thread run to completion. This method is called *FIFO* (first-in, first-out) *run-to-completion* scheduling. FIFO's advantages are: It is easy to implement, it has very low overhead, and it's "fair"—all threads are treated equally, first come, first served.

FIFO with run-to-completion scheduling is good for batch applications and some transactional applications that perform serial processing and then exit, but it doesn't work well for interactive or real-time applications. Interactive applications need relatively fast, short duration access to the CPU so they can return a quick response to a user or service some external device.

One possible solution for these applications is to assign priorities to each thread. Threads with a critical need for fast access to the CPU, such as real-time threads, can be assigned a higher priority than other less critical threads, such as batch. The high priority threads can jump to the head of the line and quickly run on the CPU. If multiple threads are waiting with the same priority, they are processed in the order in which they're received (just like basic FIFO). This method is called *run-to-completion priority scheduling*.

Run-to-completion priority scheduling, though an improvement over FIFO, still has one drawback that makes it unsuitable for interactive and real-time applications—it's easy for one thread to monopolize the processor. High priority threads can get stuck behind a long-running, low-priority thread already running on the

processor. To solve this problem, a method is needed to temporarily suspend or *preempt* a running thread so other threads can access the CPU.

### Thread Preemption

Involuntarily suspending one thread to schedule another is called *preemption*. Scheduling methods that utilize preemption instead of run to completion are said to be *preemptive*, and operating systems that employ these methods are called *preemptive multitasking* operating systems. Preemption relies on the kernel to periodically change the currently running thread via a *context switch*. The trigger for a context switch can be either a system timer (each thread is assigned a time slice) or a function call to the kernel itself. When a context switch is triggered, the kernel selects the next thread to run and the preempted thread is put back in line to run again at its next opportunity based on the scheduling method being used.

> **NOTE**
>
> A *context switch* occurs when an operating system's kernel removes one thread from the CPU and places another thread on the CPU. In other words, context switches occur when the computer changes the task on which it is currently working. Context switches can be quite expensive in terms of CPU time because all of the processor's registers must be saved for the thread being taken off the CPU and restored for the thread being put on the CPU. The context is essential for the preempted thread to know where it left off, and for the thread being run to know where it was the last time it ran.

There are several advantages to preemptive multitasking, including the following:

- **It's predictable—**

  A thread can, within limits, know when it will likely run again. For instance, given the limits of kernel implementations, a thread can be set up to run once a second and the programmer can be reasonably certain that the thread will be scheduled to run at that interval.

- **It's difficult to break—**

  No single thread can monopolize the CPU for long periods of time. A single thread falling into an endless loop cannot stop other threads from running.

Of course, there are also disadvantages to preemptive multitasking, such as:

- **It's less efficient than run-to-completion methods—**

  In general, preemptive multitasking systems tend to switch contexts more often, which means the CPU spends more time scheduling threads and switching between them than it does with run to completion.

- **It adds complexity to application software—**

  A thread running on a preemptive system can be interrupted anywhere. Programmers must design and write their applications to protect critical data structures from being changed by other threads when preempted.

## Memory Resource Management

Operating systems also manage the computer's memory, typically dividing it into various parts for storing actual computer instructions (code), data variables, and the *heap*. The heap is a section of memory from which processes can allocate and free memory dynamically.

Some operating systems provide a means for processes to address more memory than is physically present

as RAM, a concept called *virtual memory*. With virtual memory, the computer's memory can be expanded to include secondary storage, such as a disk drive, in a way that's transparent to the processes. Operating systems create virtual memory using a hardware feature, available on some processors, called a *memory map unit* (MMU). MMU automatically remaps memory address requests to either physical memory (RAM) or secondary storage (disk) depending on where the contents actually reside. The MMU also allows some address ranges to be protected (marked read-only) or to be left totally unmapped.

Virtual memory also has another benefit: In operating systems that support it, an MMU can be programmed to create a separate address space for each process. Each process can have a memory space all to itself and can be prevented from accessing memory in the address space of other processes.

Although it has many benefits, virtual memory does not come for free. There are resource requirements and performance penalties—some of them significant—associated with its use. For this reason, as you will see, IOS does not employ a full virtual memory scheme.

## Interrupts

Operating systems usually provide support for CPU interrupts. Interrupts are a hardware feature that cause the CPU to temporarily suspend its current instruction sequence and to transfer control to a special program. The special program, called an *interrupt handler*, performs operations to respond to the event that caused the interrupt, and then returns the CPU to the original instruction sequence. Interrupts often are generated by external hardware, such as a media controller requesting attention, but they also can be generated by the CPU itself. Operating systems support the interrupts by providing a set of interrupt handlers for all possible interrupt types.



< BACK  Make Note | Bookmark  CONTINUE >

## Index terms contained in this section

addresses
    MMU (memory map unit)
Cisco IOS
    virtual memory
context switches
    preemptive multitasking
contexts (threads)
CPUs
    context switches
    interrupts
    mutlitasking operating systems 2nd
    preemptive multitasking operating systems 2nd
FIFO (first-in, first-out)
    run-to-completion scheduling 2nd
hardware abstraction
heap (memory)
interrupt handler
IOS
    virtual memory
kernel
    preemptive multitasking

scheduling threads 2nd
management, resource
operating systems
CPU interrupts
memory 2nd
memory
operating systems 2nd
memory map unit (MMU)
MMU (memory map unit)
multitasking
multitasking operating systems 2nd
preempting threads 2nd
scheduling threads
operating systems
CPUs
interrupts
hardware abstraction
memory management 2nd
multitasking 2nd
scheduling threads
preemptive multitasking 2nd
resource management
preempting threads
preemptive multitasking operating systems 2nd
preemting
priority scheduling (threads)
processes 2nd
processors
context switches
interrupts
mutlitasking operating systems 2nd
preemptive multitasking operating systems 2nd
real-time applications
thread scheduling
resource management
operating systems
CPU interrupts
memory 2nd
run-to-completion priority scheduling (threads)
run-to-completion scheduling (threads) 2nd
scheduling
scheduling threads 2nd
switches
context switches
threads 2nd 3rd
scheduling
timers
preemptive multitasking
virtual memory 2nd