LINUX- SCHEDULING

- Scheduling is the job of allocating CPU time to different tasks within an operating system. Linux, like all UNIX systems, supports preemptive multitasking.
- While scheduling is normally thought of as the running and interrupting of processes, in Linux, scheduling also includes the running of the various kernel tasks
- Running kernel tasks encompasses both tasks that are requested by a running process and tasks that execute internally on behalf of a device driver

Process Scheduling

- Linux has two separate process-scheduling algorithms. One is a time-sharing algorithm for fair, preemptive scheduling among multiple processes. The other is designed for real-time tasks, where absolute priorities are more important than fairness.
- As of 2.5, new scheduling algorithm – preemptive, priority-based, known as *O*(1)
    - Real-time range 0 to 99
    - nice value 20 to 19
    - Had challenges with interactive performance
- Version 2.6 introduced **Completely Fair Scheduler** (**CFS**)

CFS

- Eliminates traditional, common idea of time slice
- Instead all tasks allocated portion of processor's time
- CFS calculates how long a process should run as a function of total number of tasks
- *N* runnable tasks means each gets 1/*N* of processor's time
- Then weights each task with its nice value
    - Smaller nice value -> higher weight (higher priority)
- CFS then runs each process for a "time slice" proportional to the process's weight divided by the total weight of all runnable processes.

The **time slice** is the length of time — the ***slice*** of the processor that a process is afforded. Traditional UNIX systems give processes a fixed time slice, perhaps with a boost or penalty for high- or low-priority processes, respectively.

A process may run for the length of its time slice, and higher-priority processes run before lower-priority processes. It is a simple algorithm that many non-UNIX systems employ.

CFS introduced a new scheduling algorithm called **fair scheduling** that eliminates time slices in the traditional sense. Instead of time slices, all processes are allotted a proportion of the processor's time. CFS calculates how long a process should run as a function of the total number of runnable processes.

To start, CFS says that if there are *N* runnable processes, then each should be afforded 1*/N* of the processor's time. CFS then adjusts this allotment by weighting each process's allotment by its nice value.

Processes with the default nice value have a weight of 1 their priority is unchanged. Processes with a smaller nice value (higher priority) receive a higher weight, while processes with a larger nice value (lower priority) receive a lower weight. CFS then runs each process for a "time slice" proportional to the process's weight divided by the total weight of all runnable processes.

To calculate the actual length of time a process runs, CFS relies on a configurable variable called **target latency**, which is the interval of time during which every runnable task should run at least once. For example, assume that the target latency is 10 milliseconds

With the switch to fair scheduling, CFS behaves differently from traditional UNIX process schedulers in several ways. Most notably, as we have seen, CFS eliminates the concept of a static time slice. Instead, each process receives a proportion of the processor's time. How long that allotment is depends on how many other processes are runnable. This approach solves several problems in mapping priorities to time slices inherent in preemptive, priority- based scheduling algorithms.

Real-Time Scheduling

Linux's real-time scheduling algorithm is significantly simpler than the fair scheduling employed for standard time-sharing processes. Linux implements the two real-time scheduling classes required by POSIX.1b: first-come, first- Served (FCFS) and round-robin.

In both cases, each process has a priority in addition to its scheduling class. The scheduler always runs the process with the highest priority. Among processes of equal priority, it runs the process that has been waiting longest. The only difference between FCFS and round-robin scheduling is that FCFS processes continue to run until they either exit or block, whereas a round-robin process will be preempted after a while and will be moved to the end of the scheduling queue, so round-robin processes of equal priority will automatically time-share among themselves.

Linux's real-time scheduling is soft rather than hard real time. The scheduler offers strict guarantees about the relative priorities of real-time processes, but the kernel does not offer any guarantees about how quickly a real-time process will be scheduled once that process becomes runnable. In contrast, a hard real-time system can guarantee a minimum latency between when a process becomes runnable and when it actually runs.

Kernel Synchronization

The way the kernel schedules its own operations is fundamentally different from the way it schedules processes. A request for kernel-mode execution can occur in two ways.

A running program may request an operating- system service, either explicitly via a system call or implicitly for example, when a fault occurs. Alternatively, a device controller may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt.

The problem for the kernel is that all these tasks may try to access the same internal data structures. If one kernel task is in the middle of accessing some data structure when an interrupt service routine executes, then that service routine cannot access or modify the same data without risking data corruption.

This fact relates to the idea of critical sections portions of code that access shared data and thus must not be allowed to execute concurrently. As a result, kernel

synchronization involves much more than just process scheduling. A framework is required that allows kernel tasks to run without violating the integrity of shared data.

Prior to version 2.6, Linux was a non preemptive kernel, meaning that a process running in kernel mode could not be preempted even if a higher-priority process became available to run. With version 2.6, the Linux kernel became fully preemptive. Now, a task can be preempted when it is running in the kernel

The Linux kernel provides spinlocks and semaphores (as well as reader – writer versions of these two locks) for locking in the kernel.

On SMP machines, the fundamental locking mechanism is a spinlock, and the kernel is designed so that spinlocks are held for only short durations. On single-processor machines, spinlocks are not appropriate for use and are replaced by enabling and disabling kernel preemption.

That is, rather than holding a spinlock, the task disables kernel preemption. When the task would otherwise release the spinlock, it enables kernel preemption. This pattern is summarized below:

Single processor	Multiple processors
Disable kernel preemption	Acquire spin lock
Enable kernel preemption	Release spin lock

- Linux uses an interesting approach to disable and enable kernel pre-emption.

It provides two simple kernel interfaces preempt disable() and preempt enable(). In addition, the kernel is not preemptible if a kernel-mode task is holding a spinlock.

To enforce this rule, each task in the system has a thread-info structure that includes the field preempt count, which is a counter indicating the number of locks being held by the task.

The counter is incremented when a lock is acquired and decremented when a lock is released.

If the value of preempt count for the task currently running is greater than zero, it is not safe to preempt the kernel, as this task currently holds a lock. If the count is zero, the kernel can safely be interrupted, assuming there are no outstanding calls to preempt disable().

Spinlocks along with the enabling and disabling of kernel preemption are used in the kernel only when the lock is held for short durations. When a lock must be held for longer periods, semaphores are used.

The second protection technique used by Linux applies to critical sections that occur in interrupt service routines. The basic tool is the processor's interrupt-control hardware. By disabling interrupts (or using spinlocks) during a critical section, the kernel guarantees that it can proceed without the risk of concurrent access to shared data structures.

However, there is a penalty for disabling interrupts. On most hardware architectures, interrupt enable and disable instructions are not cheap. More importantly, as long as interrupts remain disabled, all I/O is suspended, and any device waiting for servicing will have to wait until interrupts are reenabled; thus, performance degrades.

To address this problem, the Linux kernel uses a synchronization architecture that allows long critical sections to run for their entire duration without having interrupts disabled. This ability is especially useful in the networking code. An interrupt in a network device driver can signal the arrival of an entire network packet, which may result in a great deal of code being executed to disassemble, route, and forward that packet within the interrupt service routine.

Linux implements this architecture by separating interrupt service routines into two sections: the top half and the bottom half.

The ***top half*** is the standard interrupt service routine that runs with recursive interrupts disabled. Interrupts of the same number (or line) are disabled, but other interrupts may run.

The ***bottom half*** of a service routine is run, with all interrupts enabled, by a miniature scheduler that ensures that bottom halves never interrupt themselves. The bottom-half scheduler is invoked automatically whenever an interrupt service routine exits.

This separation means that the kernel can complete any complex processing that has to be done in response to an interrupt without worrying about being interrupted itself. If another interrupt occurs while a bottom half is executing, then that interrupt can request that the same bottom half execute, but the execution will be deferred

until the one currently running completes. Each execution of the bottom half can be interrupted by a top half but can never be interrupted by a similar bottom half.

The top-half/bottom-half architecture is completed by a mechanism for disabling selected bottom halves while executing normal, foreground kernel code. The kernel can code critical sections easily using this system.

Interrupt handlers can code their critical sections as bottom halves; and when the foreground kernel wants to enter a critical section, it can disable any relevant bottom halves to prevent any other critical sections from interrupting it.

At the end of the critical section, the kernel can reenable the bottom halves and run any bottom-half tasks that have been queued by top-half interrupt service routines during the critical section.

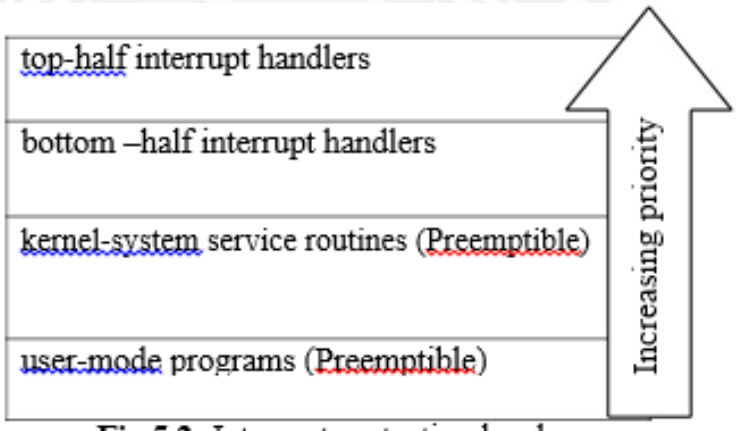

Fig 5.2: Interrupt protection levels

Figure 5.2 summarizes the various levels of interrupt protection within the kernel. Each level may be interrupted by code running at a higher level but will never be interrupted by code running at the same or a lower level. Except for user-mode code, user processes can always be preempted by another process when a time-sharing scheduling interrupt occurs.

Symmetric Multiprocessing

The Linux 2.0 kernel was the first stable Linux kernel to support **symmetric multiprocessor (SMP)** hardware, allowing separate processes to execute in parallel on separate processors. The original implementation of SMP imposed the restriction that only one processor at a time could be executing kernel code.

In version 2.2 of the kernel, a single kernel spinlock (sometimes termed **BKL** for "big kernel lock") was created to allow multiple processes (running on different processors) to be active in the kernel concurrently.

However, the BKL provided a very coarse level of locking granularity, resulting in poor scalability to machines with many processors and processes. Later releases of the kernel made the SMP implementation more scalable by splitting this single kernel spinlock into multiple locks, each of which protects only a small subset of the kernel's data structures.

Such spinlocks are described in Section 18.5.3. The 3.0 kernel provides additional SMP enhancements, including ever- finer locking, processor affinity, and load-balancing algorithms.