So far in constructing our timesharing system, we've worked hard to build an execution environment that gives each process the illusion of running on its own independent virtual machine.

The processes appear to run concurrently although we're really quickly switching between running processes on a single hardware system. This often leads to better overall utilization since if a particular process is waiting for an I/O event, we can devote the unneeded cycles to running other processes. The downside of timesharing is that it can be hard to predict exactly how long a process will take to complete since the CPU time it will receive depends on how much time the other processes are using.

So we'd need to know how many other processes there are, whether they're waiting for I/O events, etc. In a timesharing system we can't make any guarantees on completion times. And we chose to have the OS play the intermediary between interrupt events triggered by the outside world and the user-mode programs where the event processing occurs. In other words, we've separated event handling (where the data is stored by the OS) and event processing (where the data is passed to user-mode programs via SVCs). This means that using a conventional timesharing system, it's hard to ensure that event processing will be complete by a specified event deadline, i.e., before the end of a specified time period after the event was triggered.

Since modern CPU chips provide inexpensive, high-performance, general-purpose computing, they are often used as the "brains" of control systems where deadlines are a fact of life.

For example, consider the electronic stability control (ESC) system on modern cars.

This system helps drivers maintain control of their vehicle during steering and braking maneuvers by keeping the car headed in the driver's intended direction.

The computer at the heart of the system measures the forces on the car, the direction of steering, and the rotation of the wheels to determine if there's been a loss of control due to a loss of traction, i.e., is the car "spinning out"?

If so, the ESC uses rapid automatic braking of individual wheels to prevent the car's heading from veering from the driver's intended heading.

With ESC you can slam on your brakes or swerve to avoid an obstacle and not worry that the car will suddenly fishtail out of control. You can feel the system working as a chatter in the brakes. To be effective, the ESC system has to guarantee the correct braking action at each wheel within a certain time of receiving dangerous sensor settings. This means that it has to be able to guarantee that certain subroutines will run to completion within some predetermined time of a sensor event. To be able to make these guarantees we'll have to come up with a better way to schedule process execution - round-robin scheduling won't get the job done! Systems that can make such guarantees are called "real-time systems". One measure of performance in a real-time system is the interrupt latency L, the amount of time that elapses between a request to run some code and when that code actually starts executing.

If there's a deadline D associated with servicing the request, we can compute the maximum allowable latency that still permits the service routine to complete by the deadline.

In other words, what's the largest L such that L_max+S = D?

Bad things can happen if we miss certain deadlines. Maybe that's why we call them "dead"-lines :) In those cases we want our real time system to guarantee that the actual latency is always less than the maximum allowable latency.

These critical deadlines give rise to what we call "hard real-time constraints".

What factors contribute to interrupt latency? Well, while handling an interrupt it takes times to save the process state, switch to the kernel context, and dispatch to the correct interrupt handler. When writing our OS, we can work to minimize the amount of code involved in the setup phase of an interrupt handler.

We also have to avoid long periods of time when the processor cannot be interrupted.

Some ISAs have complex multi-cycle instructions, e.g., block move instructions where a single instruction makes many memory accesses as it moves a block of data from one location to another. In designing the ISA, we need to avoid such instructions or design them so that they can be interrupted and restarted.

The biggest problem comes when we're executing another interrupt handler in kernel mode.

In kernel mode, interrupts are disabled, so the actual latency will be determined by the time it takes to complete the current interrupt handler in addition to the other costs mentioned above. This latency is not under the control of the CPU designer and will depend on the particular application.

Writing programs with hard real-time constraints can get complicated!

Our goal is to bound and minimize interrupt latency.

We'll do this by optimizing the cost of taking an interrupt and dispatching to the correct handler code. We'll avoid instructions whose execution time is data dependent. And we'll work to minimize the time spent in kernel mode. But even with all these measures, we'll see that in some cases we'll have to modify our system to allow interrupts even in kernel mode. Next we'll look at some concrete examples and see what mechanisms are required to make guarantees about hard real-time constraints.