In a weak priority system the currently-running task will always run to completion before considering what to run next.

This means the worst-case latency for a device always includes the worst-case service time across all the other devices, i.e., the maximum time we have to wait for the currently-running task to complete.

If there's a long-running task that usually means it will be impossible to meet tight deadlines for other tasks.

For example, suppose disk requests have a 800 us deadline in order to guarantee the best throughput from the disk subsystem.

Since the disk handler service time is 500 us, the maximum allowable latency between a disk request and starting to execute the disk service routine is 300 us.

Oops!

The weak priority scheme can only guarantee a maximum latency of 800 us, not nearly fast enough to meet the disk deadline.

We can't meet the disk deadline using weak priorities.

We need to introduce a preemptive priority system that allows lower-priority handlers to be interrupted by higher-priority requests.

We'll refer to this as a "strong" priority system.

Suppose we gave the disk the highest priority, the printer second priority, and keyboard the lowest priority, just like we did before.

Now when a disk request arrives, it will start executing immediately without having to wait for the completion of the lower-priority printer or keyboard handlers.

The worst-case latency for the disk has dropped to 0.

The printer can only be preempted by the disk, so it's worst-case latency is 500 us.

Since it has the lowest priority, the worst-case latency for the keyboard is unchanged at 900 us since it might still have to wait on the disk and printer.

The good news: with the proper assignment of priorities, the strong priority system can guarantee that disk requests will be serviced by the 800 us deadline.

We'll need to make a small tweak to our Beta hardware to implement a strong priority system.

We'll replace the single supervisor mode bit in PC[31] with, say, a three-bit field (PRI) in PC[31:29] that indicates which of the eight priority levels the processor is currently running at.

Next, we'll modify the interrupt mechanism as follows.

In addition to requesting an interrupt, the requesting device also specifies the 3-bit priority it was assigned by the system architect.

We'll add a priority encoder circuit to the interrupt hardware to select the highest-priority request and compare the priority of that request (PDEV) to the 3-bit PRI value in the PC.

The system will take the interrupt request only if PDEV > PRI, i.e., if the priority of the request is *higher* than the priority the system is running at.

When the interrupt is taken, the old PC and PRI information is saved in XP, and the new PC is determined by the type of interrupt and the new PRI field is set to PDEV.

So the processor will now be running at the higher priority specified by the device.

A strong priority system allows low-priority handlers to be interrupted by higher-priority requests, so the worst-case latency seen at high priorities is unaffected by the service times of lower-priority handlers.

Using strong priorities allows us to assign a high priority to devices with tight deadlines and thus guarantee their deadlines are met.

Now let's consider the impact of recurring interrupts, i.e., multiple interrupt requests from each device.

We've added a "maximum frequency" column to our table, which gives the maximum rate at which requests will be generated by each device.

The execution diagram for a strong priority system is shown below the table.

Here we see there are multiple requests from each device, in this case shown at their maximum possible rate of request.

Each tick on the timeline represent 100 us of real time.

Printer requests occur every 1 ms (10 ticks), disk requests every 2 ms (20 ticks), and keyboard requests every 10 ms (100 ticks).

In the diagram you can see that the high-priority disk requests are serviced as soon as they're received.

And that medium-priority printer requests preempt lower-priority execution of the keyboard handler.

Printer requests would be preempted by disk requests, but given their request patterns, there's never a printer request in progress when a disk request arrives, so we don't see that happening here.

The maximum latency before a keyboard requests starts is indeed 900 us.

But that doesn't tell the whole story!

As you can see, the poor keyboard handler is continually preempted by higher-priority disk and printer requests and so the keyboard handler doesn't complete until 3 ms after its request was received!

This illustrates why real-time constraints are best expressed in terms of deadlines and not latencies.

If the keyboard deadline had been less that 3 ms, even the strong priority system would have failed to meet the hard real-time constraints.

The reason would be that there simply aren't enough CPU cycles to meet the recurring demands of the devices in the face of tight deadlines.

Speaking of having enough CPU cycles, there are several calculations we need to do when thinking about recurring interrupts.

The first is to consider how much load each periodic request places on the system.

There's one keyboard request every 10 ms and servicing each request takes 800 us, which consumes 800us/10ms = 8% of the CPU.

A similar calculation shows that servicing the disk takes 25% of the CPU and servicing the printer takes 40% of the CPU.

Collectively servicing all the devices takes 73% of the CPU cycles, leaving 27% for running user-mode programs.

Obviously we'd be in trouble if takes more than 100% of the available cycles to service the devices.

Another way to get in trouble is to not have enough CPU cycles to meet each of the deadlines.

We need 500/800 = 67.5% of the cycles to service the disk in the time between the disk request and disk deadline.

If we assume we want to finish serving one printer request before receiving the next, the effective printer deadline is 1000 us.

In 1000 us we need to be able to service one higher-priority disk request (500 us) and, obviously, the printer request (400 us).

So we'll need to use 900 us of the CPU in that 1000 us interval.

Whew, just barely made it!

Suppose we tried setting the keyboard deadline to 2000 us.

In that time interval we'd also need to service 1 disk request and 2 printer requests.

So the total service time needed is 500 + 2*400 + 800 = 2100 us.

Oops, that exceeds the 2000 us window we were given, so we can't meet the 2000 us deadline with the available CPU resources.

But if the keyboard deadline is 3000 us, let's see what happens.

In a 3000 us interval we need to service 2 disk requests, 3 printer requests, and, of course, 1 keyboard request, for a total service time of 2*500 + 3*400 + 800 = 3000 us.

Whew!

Just made it!