Let's finish up by looking at two extended examples.

The scenario for both examples is the control system for the International Space Station, which has to handle three recurring tasks: supply ship guidance (SSG), gyroscope control (G), and cabin pressure (CP).

For each device, the table shows us the time between successive requests (the period), the service time for each request, and the service deadline for each request.

We'll first analyze the system assuming that it's using a weak priority system.

First question: What is the maximum service time for the cabin pressure task that still allows all constraints to be met?

Well, the SSG task has a maximum allowable latency of 20 ms, i.e., it's service routine must start execution within 20 ms if it is to meet its 25 ms deadline.

The G task has a maximum allowable latency of 10 ms if it's to meet its deadline.

So no other handler can take longer than 10 ms to run or the G task will miss its deadline.

2.

Give a weak priority ordering that meets the constraints.

Using the earliest deadline strategy discussed earlier, the priority would be G with the highest priority, SSG with the middle priority, and CP with the lowest priority.

3.

What fraction of time will the processor spend idle?

We need to compute the fraction of CPU cycles needed to service the recurring requests for each task.

SSG takes 5/30 = 16.67% of the CPU cycles.

G takes 10/40 = 25% of the CPU cycles.

And CP takes 10/100 = 10% of the CPU cycles.

So servicing the task requests takes 51.67% of the cycles, leaving 48.33% of the cycles unused.

So the astronauts will be able to play Minecraft in their spare time :) 4.

What is the worst-case delay for each task until completion of its service routine?

Each task might have to wait for the longest-running lower-priority handler to complete plus the service times of any other higher-priority tasks plus, of course, its own service time.

SSG has the lowest priority, so it might have to wait for CP and G to complete (a total of 20 ms), then add its own service time (5 ms).

So it's worst-case completion time is 25 ms after the request.

G might to wait for CP to complete (10 ms), then add its own service time (10 ms) for a worst-case completion time of 20 ms.

CP might have to wait for SSG to finish (5 ms), then wait for G to run (10 ms), then add its own service time (10 ms) for a worst-case completion time of 25 ms.

Let's redo the problem, this timing assuming a strong priority system where, as before, G has the highest priority, SSG the middle priority, and CP the lowest priority.

What is the maximum service time for CP that still allows all constraints to be met?

This calculation is different in a strong priority system, since the service time of CP is no longer constrained by the maximum allowable latency of the higher-priority tasks - they'll simply preempt CP when they need to run!

Instead we need to think about how much CPU time will be used by the SSG and G tasks in the 100 ms interval between the CP request and its deadline.

In a 100 ms interval, there might be four SSG requests (at times 0, 30, 60, and 90) and three G requests (at times 0, 40, and 80).

Together these requests require a total of 50 ms to service.

So the service time for CP can be up 50 ms and still meet the 100 ms deadline.

2.

What fraction of the time will the processor spend idle?

Assuming a 50 ms service time for CP, it now consumes 50% of the CPU.

The other request loads are as before, so 91.67% of the CPU cycles will be spent servicing requests, leaving 8.33% of idle time.

3.

What is the worst-case completion time for each task?

The G task has the highest priority, so its service routine runs immediately after the request is received and its worst-case completion time is exactly its service time.

In the 25 ms interval between an SSG request and its deadline, there might be at most one G request that will preempt execution.

So the worst-case completion time is one G service time (10 ms) plus the SSG service time (5 ms).

Finally, from the calculation for problem 1, we chose the service time for the CP task so that it will complete just at its deadline of 100 ms, taking into account the service time for multiple higher-priority requests.

We covered a lot of ground in this lecture!

We saw that the computation needed for user-mode programs to interact with external devices was split into two parts.

On the device-side, the OS handles device interrupts and performs the task of moving data between kernel buffers and the device.

On the application side, user-mode programs access the information via SVC calls to the OS.

We worried about how to handle SVC requests that needed to wait for an I/O event before the request could be satisfied.

Ultimately we came up with a sleep/wakeup mechanism that suspends execution of the process until the some interrupt routine signals that the needed information has arrived, causing the sleeping process to marked as active.

Then the SVC is retried the next time the now active process is scheduled for execution.

We discussed hard real-time constraints with their latencies, service times and deadlines.

Then we explored the implementation of interrupt systems using both weak and strong priorities.

Real-life computer systems usually implement strong priorities and support a modest number of priority levels, using a weak priority system to deal with multiple devices assigned to the same strong priority level.

This seems to work quite well in practice, allowing the systems to meet the variety of real-time constraints imposed by their I/O devices.