Real Time Scheduling

Introduction

Priority based scheduling enables us to give better service to certain processes. In our discussion of multi-queue scheduling, priority was adjusted based on whether a task was more interactive or compute intensive. But most schedulers enable us to give any process any desired priority. Isn't that good enough?

Priority scheduling is inherently a *best effort* approach. If our task is competing with other high priority tasks, it may not get as much time as it requires. Sometimes best effort isn't good enough:

- During reentry, the space shuttle is aerodynamically unstable. It is not actually being kept under control by the quick reflexes of the well-trained pilots, but rather by guidance computers that are collecting attitude and acceleration input and adjusting numerous spoilers hundreds of times per second.
- Scientific and military satellites may receive precious and irreplaceable sensor data at extremely high speeds. If it takes us too long to receive, process, and store one data frame, the next data frame may be lost.
- More mundanely, but also important, many manufacturing processes are run by computers nowadays. An assembly line needs to move at a particular speed, with each step being performed at a particular time. Performing the action too late results in a flawed or useless product.
- Even more commonly, playing media, like video or audio, has real time requirements. Sound must be produced at a certain rate and frames must be displayed frequently enough or the media becomes uncomfortable to deal with.

There are many computer controlled applications where delays in critical processing can have undesirable, or even disastrous consequences.

What are Real-Time Systems

A real-time system is one whose correctness depends on timing as well as functionality.

When we discussed more traditional scheduling algorithms, the metrics we looked at were *turn-around time* (or throughput), *fairness*, and mean *response time*. But real-time systems have very different requirements, characterized by different metrics:

- *timeliness* ... how closely does it meet its timing requirements (e.g. ms/day of accumulated tardiness)
- *predictability* ... how much deviation is there in delivered timeliness

And we introduce a few new concepts:

- *feasibility* ... whether or not it is possible to meet the requirements for a particular task set
- *hard real-time* ... there are strong requirements that specified tasks be run a specified intervals (or within a specified response time). Failure to meet this requirement (perhaps by as little as a fraction of a micro-second) may result in system failure.
- *soft real-time* ... we may want to provide very good (e.g. microseconds) response time, the only consequences of missing a deadline are degraded performance or recoverable failures.

It sounds like real-time scheduling is more critical and difficult than traditional time-sharing, and in many ways it is. But real-time systems may have a few characteristics that make scheduling easier:

- We may actually know how long each task will take to run. This enables much more intelligent scheduling.

- *Starvation* (of low priority tasks) may be acceptable. The space shuttle absolutely must sense attitude and acceleration and adjust spolier positions once per millisecond. But it probably doesn't matter if we update the navigational display once per millisecond or once every ten seconds. Telemetry transmission is probably somewhere in-between. Understanding the relative criticality of each task gives us the freedom to intelligently shed less critical work in times of high demand.
- The work-load may be relatively fixed. Normally high utilization implies long queuing delays, as bursty traffic creates long lines. But if the incoming traffic rate is relatively constant, it is possible to simultaneously achieve high utilization and good response time.

Real-Time Scheduling Algorithms

In the simplest real-time systems, where the tasks and their execution times are all known, there might not even be a scheduler. One task might simply call (or yield to) the next. This model makes a great deal of sense in a system where the tasks form a producer/consumer pipeline (e.g. MPEG frame receipt, protocol decoding, image decompression, display).

In more complex real-time system, with a larger (but still fixed) number of tasks that do not function in a strictly pipeline fashion, it may be possible to do *static* scheduling. Based on the list of tasks to be run, and the expected completion time for each, we can define (at design or build time) a fixed schedule that will ensure timely execution of all tasks.

But for many real-time systems, the work-load changes from moment to moment, based on external events. These require *dynamic* scheduling. For *dynamic* scheduling algorithms, there are two key questions:

1. how they choose the next (ready) task to run
    - shortest job first
    - static priority ... highest priority ready task
    - soonest start-time deadline first (ASAP)
    - soonest completion-time deadline first (slack time)
2. how they handle overload (infeasible requirements)
    - best effort
    - periodicity adjustments ... run lower priority tasks less often.
    - work shedding ... stop running lower priority tasks entirely.

Preemption may also be a different issue in real-time systems. In ordinary time-sharing, preemption is a means of improving mean response time by breaking up the execution of long-running, compute-intensive tasks. A second advantage of preemptive scheduling, particularly important in a general purpose timesharing system, is that it prevents a buggy (infinite loop) program from taking over the CPU. The trade-off, between improved response time and increased overhead (for the added context switches), almost always favors preemptive scheduling. This may not be true for real-time systems:

- preempting a running task will almost surely cause it to miss its completion deadline.
- since we so often know what the expected execution time for a task will be, we can schedule accordingly and should have little need for preemption.
- embedded and real-time systems run fewer and simpler tasks than general purpose time systems, and the code is often much better tested ... so infinite loop bugs are extremely rare.

For the least demanding real time tasks, a sufficiently lightly loaded system might be reasonably successful in meeting its deadlines. However, this is achieved simply because the frequency at which the task is run happens to be high enough to meet its real time requirements, not because the scheduler is aware of such requirements. A lightly loaded machine running a traditional scheduler can often display a video to a user's satisfaction, not because the scheduler "knows" that a frame must be rendered by a certain deadline, but simply because the machine has enough cycles and a low enough work load to render the frame before the deadline has arrived.

Real-Time and Linux

Linux was not designed to be an embedded or real-time operating system, but many tasks that were once-considered embedded applications now require the capabilities (e.g. file systems, network protocols) of a general purpose operating system. As these requirements have increased and processors have gotten faster, increasingly many embedded and real-time applications have moved to Linux.

To support these applications Linux now supports a real-time scheduler, which can be enabled with sched_setscheduler(2), and is described in this Linux Journal Article. This real-time scheduler does not provide quite the same level of response-time guarantees that more traditional Real-Time-OSs do, but they are adequate for many soft real-time applications.

What about Windows? Conventional wisdom states that Windows is not well suited for real time needs, offering no native real time scheduler and too many ways in which desired real time deadlines might be missed. Windows favors general purpose throughput over meeting deadlines, as a rule. With sufficiently low load, a Windows system may, nonetheless, provide fast enough service for some soft real time requirements, such as playing music or video. One should be careful in relying on Windows for critical real time operations, however, as it is not designed for that purpose.