ECE 350: Real-Time Operating Systems Spring 2023

Lecture 16 — Real Time Scheduling Algorithms

Jeff Zarnett *2023-03-18*

Real-Time Scheduling Algorithms

Given our understanding of scheduling algorithms in general and an undemanding of what makes real-time system scheduling a little bit different from just regular scheduling, we can consider some other scheduling algorithms that work for real-time systems. We already covered the idea of timeline scheduling. If everything is predictable and orderly then making a schedule that looks like a schedule of classes is effective. But we also know that's not the case, so we'll talk about some other scheduling algorithms that are able to handle aperiodic and sporadic tasks.

Earliest Deadline First

The earliest deadline first algorithm is, presumably, very familiar to students. If there is an assignment due today, an assignment due next Tuesday, and an exam next month, then you may choose to schedule these things by their deadlines: do the assignment due today first. After completing an assignment, decide what to do next (probably the new assignment, but perhaps a new task has arrived in the meantime?) and get on with that.

The principle is the same for the computer. Choose the task with the soonest deadline; if there is a tie, then random selection between the two will be sufficient (or other criteria may be used, if desired). If there exists some way to schedule all the tasks such that all deadlines are met, this algorithm will find it. There's a proof of this in [HZMG20] if you'd like to see it.

Part of what makes this work is preemption, because a task could arise that is more urgent (i.e., has an earlier deadline) than the currently-executing one. From the point of view of the operating system, on completion of the system call to handle the request to schedule the new task, suspend the previously-executing task and start running the new one. This might mean a periodic task being preempted by an aperiodic or sporadic task [HZMG20].

To implement this, a priority queue is reasonable. A simple view says that the priority is determined by the deadline: keep it sorted in ascending order of the time of deadline. However, this doesn't account for the possibility that soft-real time tasks may have a sooner deadline than a firm- or hard-real time task. If the system is not overloaded then there is no issue, and ideally good system design means that overload isn't an issue. But if it is, then things get a little more interesting.

Do we skip the soft-real time task? Do we start it but cancel it if the situation gets dire for a more important task? Let's come back to that after discussing a couple of other algorithms.

Deadline Interchange. This deadline-based approach is subject to a problem that very much resembles priority inversion. Suppose a task A has locked a mutex and then a new task B arrives that also needs that mutex and has a sooner deadline than A. Task A would therefore be preempted in favour of B, at least until B gets blocked. If there are other tasks C, D, E... that also want this resource, A could be waiting a long time to proceed. In fact, it could be waiting so long that task B could miss its deadline!

To solve this, A needs to finish the critical section and release the mutex. The best way to make that happen would be to assign to A a new deadline, specifically the soonest deadline from all the tasks waiting for it. That looks a lot like priority inheritance, doesn't it? It's a very similar problem, so it is not surprising that the solution looks similar.

Least Slack First

A similar algorithm to earliest deadline first, is least slack first. The definition of *slack* is how long a task can wait before it must be scheduled to meet its deadline. If a task will take 10 ms to execute and its deadline is 50 ms away, then there are (50 - 10) = 40 ms of slack time remaining. We have to start the task before 40 ms are expired if we want to be sure that it will finish. This does not mean, however, that we necessarily want to wait 40 ms before starting the task (even though many students tend to operate on this basis). All things being equal, we prefer tasks to start and finish as soon as possible. It does, however, give us an indication of what tasks are in most danger of missing their deadlines and should therefore have priority.

Much like the earliest deadline first approach, a queue where priority is determined by the slack makes for a reasonable implementation. Some work is needed periodically to recalculate the slack for each task, though.

Rate-Monotonic Scheduling

Unlike the previous two scheduling algorithms, the name doesn't explain as much about how this one works. We consider the rate monotonic algorithm because something like the earliest deadline first or least slack first approach is that they focus solely on deadlines but do not consider priorities otherwise.

To figure out if it's possible to schedule things, there's a test.

Aperiodic Servers

Let's return to the idea of aperiodic tasks in the earliest-deadline-first approach. A task with a soft deadline is challenging to schedule here, because it's hard for the scheduler to know whether a task is soft- or hard-real time. One possibility is to say that aperiodic or soft-deadline tasks are always lower in priority than any firm- or hard-real time task, but that may not be optimal.

We'll examine an approach called a polling server as explained in the original paper introducing the idea. A polling server is, in its own way, a little container in which aperiodic tasks occur. The server is itself a hard-deadline periodic task with a fixed execution time budget and a deadline equal to its period [GB95]. Every time this task runs, it's really a trojan horse for the aperiodic tasks that want to run. During the execution time of the server task, the aperiodic tasks waiting will run sequentially. If there are too many tasks or they otherwise take too long and they do not finish, the aperiodic tasks just carry over to the next time the server runs. If there are not enough aperiodic tasks and there's time left over, just end the server task execution and let something else run.

An analogy that makes some sense here is a lunch break at work. You don't have to use this time period solely for eating, but you can use it to do various other tasks if you want or need: go to the bank, go shopping, etc. The lunch break task is "important" in the sense that you cannot skip it or reschedule it indefinitely. But when it is lunch break time, it's your time to do with as you wish to do tasks that otherwise might be difficult to schedule.

References

[GB95] T. M. Ghazalie and T. P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 9(1):31–67, 1995. doi:10.1007/BF01094172.

[HZMG20] Douglas Wilhelm Harder, Jeff Zarnett, Vajih Montaghami, and Allyson Giannikouris. *A Practical Introduction to Real-Time Systems for Undergraduate Engineering*. 2020. Online; version 0.2020.07.19.