ECE 350: Real-Time Operating Systems Spring 2023

Lecture 15 — Real Time Scheduling

Jeff Zarnett *2023-03-12*

Real-Time Scheduling

It's been a while since we introduced it, but we started earlier by saying a real-time system one that is supposed to respond to events within a certain amount of real (wall-clock) time. There are deadlines, and there are consequences for missing deadlines. Furthermore, fast is not as important as predictable. If real time systems are of interest to you, you may wish to take some of the later embedded systems courses, or the famous CS trains course[1]. There's also a 4th year ECE technical elective on real-time systems.

We say that a task is *hard real-time* if it has a deadline that must be met to prevent an error, prevent some damage to the system, or for the answer to make sense. If a task is attempting to calculate the position of an incoming missile, a late answer is no good. A *soft real-time* task has a deadline that is not, strictly speaking, mandatory; missing the deadline degrades the quality of the response, but it is not useless [Sta18].

Sometimes there's a distinction between *firm* and *hard* real-time in the literature. In that situation, it's really just about the severity of the consequences. A firm deadline is one in which the response is useless if it arrives a little too late; a hard deadline is one in which the system itself fails if it doesn't meet the deadline. If the Mars rover misses a deadline and drives off a cliff to its theatrical-style doom, that qualifies as a truly hard deadline. If we are calculating the position of a satellite to transmit the data and the answer arrives too late, the location we calculate is too far out of date to be useful, but we can try again, so we might call this firm instead.

Real-Time vs Non-Real-Time Operating System

As a general note, most of the operating systems you are familiar with (standard Desktop/Server Linux, Mac OS, Windows) are not very suitable to real time systems. They make few guarantees, if any, about service. When there are consequences for missing deadlines, this kind of thing matters. In Windows you can set a high priority (e.g. level 31) to a process and it calls the priority "Realtime", but please don't be fooled: this does not mean it guarantees it will make a particular deadline.

This was, as we already discussed, a reason why Java is not a very good choice for a real-time system: the garbage collector runs whenever it pleases and can "stop the world" (halt all execution) until that is finished. And how long is that going to take? Nobody knows.

That's a user-space thing, though. As an example of why the OS isn't suitable for real-time operations, the system provides you no real guarantee as to the length of time it will take to execute a system call. If you want to read from disk, the system call takes an arbitrary amount of time to execute on its own and will also have to wait an arbitrary amount of time to get the data from the disk depending on the speed of the device and how busy it is. We can estimate how long it will be, of course, and we can measure things, but even though it might be okay on average, what about the worst case scenario? Are we sure that will be within some bounded time limit? The answer is that you probably cannot be sure, and if this is a motor vehicle or industrial machinery or similar, certainty about time limits can be a question of life or death.

If we think of other possible reasons why a task may take an arbitrary amount of time, you would likely think of interrupts and concurrency control functionality. Interrupts are supposed to be quick, and we talked about things you can and cannot do in an interrupt handler (e.g., signal) but there's no enforcement of the rule and no time limit on the length of the interrupt handling routine; there's also no limit on the number of interrupts that can

[1] Abandon sanity, all ye who enter here.

be generated in a given period of time. Similarly, when a thread is blocked on a mutex, there could be arbitrarily many threads ahead of it in the queue – even if you used a queueing system that did not have any risk of starvation – so the wait to enter the critical section could accordingly be a very long time indeed.

So it's fair to say that a real-time operating system is just different from a general purpose one. A lot of the difference comes down to how things are scheduled, which is why the topic of the real-time operating system comes up now. To illustrate, let's consider two scenarios in which a hard real-time task could fail to meet its deadline.

The first is that it is scheduled too late; like an assignment that will take two hours to complete being started one hour before the deadline. If that is the case, the system will likely reject the request to start the task, or perhaps never schedule the task to run at all. Why waste computation time on a task that will not finish in time? We could certainly argue that a more intelligent scheduling decision – start this task earlier – may have made it possible to complete it in time.

The second scenario is that at the time of starting, completion was possible, but for whatever reason (e.g., other tasks with higher priority have occurred) it is no longer possible to meet the deadline. In that case, execution of the task may be terminated partway through so that no additional effort is wasted on a task that cannot be completed. Again, was scheduling here the culprit? Maybe; if the interrupts were treated differently perhaps it would have been possible to complete the original task.

Properties of Real-Time Systems

Real-time systems are considered to be unique in five key areas [Sta18]:

1. Determinism
2. Responsiveness
3. User (administrator) control
4. Reliability
5. Fail-Soft operation

Determinism. Determinism means that operations are predictable, either performing them at fixed times or within certain time limits. When there are multiple things happening concurrently, perfect determinism is not going to happen. And even for a system that does only one thing, if it is triggered by some external factors, determinism is not possible because of the randomness of life. For most RTOS scenarios, it's sufficient to know that things will happen within some fixed time period, that is, that no matter how unlucky the timing or sequence of events, we can still start the task on time to successfully complete it.

Nondeterminism is not necessarily bad. We learned already about caching and this can mean that some requests take less time to service than others, just because the data happens by chance to be in the cache of the CPU doing the work. If you wanted to minimize nondeterminism you could disable the CPU cache (or use one without cache). That wouldn't make the system faster; on the contrary, it would make it slower, but it would make it more likely that every request takes about the same amount of time. I can't say I really endorse this strategy, because the presence of caching makes the worst case no worse and the best case much better – this sounds like free performance to me! – but there may be a system I haven't thought of where less determinism is actually preferable.

Responsiveness. Responsiveness sounds like it might be the same as determinism, but the key distinction is that determinism is how long it takes before the operating system takes before acknowledging the request/interrupt and responsiveness is how long it takes after acknowledgement to handle it [Sta18]. Responsiveness includes not only the time to execute the interrupt handler, but also the time it takes to start the interrupt handler and what happens if the interrupt is itself interrupted by another higher-priority interrupt.

Administrator Control. Administrator control is often quite different in a real-time operating system and there are two ways that it can go: more control or no control. A non-real-time system usually lands somewhere in between.

In the no-control scenario, the system takes no instructions from users or administrators and simply runs as it has been programmed and configured. Users have no way to start new processes or tell the system to change priorities.

In the more control scenario, administrators get to choose a lot. The OS itself has no way of knowing which tasks are real-time and which (if any) are not, nor can it know which of the real-time tasks are soft/firm/hard. Accordingly, an administrator must be able to specify what is what, and maybe even be able to make other choices like what scheduling algorithm should be used.

Reliability and Fail-Soft Operation. Actually, these topics aren't as relevant to scheduling for the moment, so let's defer that discussion until we get a bit further in the course. At this point, all we should consider is that fail-soft means that if it's somehow not possible to succeed in completing all tasks, the system will try its best.

Timeline Scheduling

If a process or task recurs at regular intervals, it is *periodic* – repeats at a fixed period. Periodic tasks are very common: check a sensor, decode and display a frame of video to a screen, keep the wifi connection alive, etc.

Consider a periodic task to have two attributes: τ_k, the period (how often the task occurs) and c_k, the computation time (how long, in the worst case, the task might execute). In real-time systems we are usually pessimists and care almost exclusively about the worst case scenario. We can calculate the processor utilization of periodic tasks according to the following formula, to get the long-term average of processor utilization U [HZMG15]:

$$U = \sum_{k=1}^{n} \frac{c_k}{\tau_k}$$

If $U > 1$, it means the system is overloaded: there are too many periodic tasks and we cannot guarantee that the system can execute all tasks and meet the deadlines. Otherwise, we will be able to devise a schedule of some sort that makes it all work.

If the only tasks in the system are periodic ones, then we can create a fixed schedule for them. This is what the university administrators do when they create the schedule of classes for a given term. Every Monday, for example, from 13:30-14:50, course ECE 254 (a "process", if you will) takes place in classroom EIT 1015 (a resource). There is no way to have two classes in the same lecture hall at the same time, so if there are more requests for room reservations than rooms and time slots available, it means some requests cannot be accommodated.

A world in which all the tasks are periodic and behave nicely is, well, a very orderly world (and that has its appeal). Unfortunately, the real world is not so accommodating most of the time. So we will need to deal with tasks that are not periodic, which we can categorize as *aperiodic* or *sporadic*.

Aperiodic tasks are ones that respond to events that occur irregularly. There is no minimum time interval between two events; therefore it would be very difficult to make a guarantee that we will finish them before the next one occurs, so they are rarely hard real-time (hard deadlines). Tasks like this should be scheduled in such a way that that they do not prevent another task with a hard deadline from missing its deadline. As an example, if we expect an average arrival rate of 3 requests per second, there is still a 1.2% chance that eight or more requests appear within 1 second. If so, there is not much we can do to accommodate them all (most likely) [HZMG15].

Sporadic tasks are aperiodic, but they require meeting deadlines. To make such a promise we need a guarantee that there is a minimum time τ_k between occurrences of the event. Sporadic tasks can overload the system if there are too many of them, and if that is the case, we must make decisions about what tasks to schedule and which ones

will miss their deadlines. If we know a task will not make its deadline, we will likely not even bother to schedule it (why waste the time on a task where the answer will be irrelevant?) [HZMG15].

So, these aperiodic and sporadic tasks really mess with timeline scheduling. This unpredictability makes it hard to create a simple timetable and follow it, just as we would expect when it comes to assigning classes and rooms.

If we have pre-emptive scheduling, then we can examine two optimal alternatives. They are called optimal because they will ensure a schedule where all tasks meet their deadlines, not because they are the ideal algorithms.

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. If a task is executing because its deadline is the earliest and another task arrives with a sooner deadline, then preemption means the currently executing task should be suspended and the new task scheduled. This might mean a periodic task being preempted by an aperiodic or sporadic task [HZMG15].

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.

References

[HZMG15] Douglas Wilhelm Harder, Jeff Zarnett, Vajih Montaghami, and Allyson Giannikouris. *A Practical Introduction to Real-Time Systems for Undergraduate Engineering*. 2015. Online; version 0.15.08.17.

[Sta18] William Stallings. *Operating Systems Internals and Design Principles (9th Edition)*. Pearson, 2018.