ECE 350: Real-Time Operating Systems Spring 2023

Lecture 15 — Real Time Scheduling

Jeff Zarnett *2023-07-12*

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.

The term "task" is used to refer to something that needs doing. Yes, the scheduler operates on threads rather than specific things to do, but we will say for the sake of the discussion that each task corresponds to one thread that is trying to carry out some work. When we say that the system schedules a task, please understand it means it schedules the thread associated with that task.

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the position of a satellite to transmit the data and the answer arrives too late, the location we calculate is too far out of date to be useful, but we can try again, so we might call this firm instead.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.

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

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 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, where users and administrators get control of some things but not everything.

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. That's pretty reasonable in certain contexts – you should not be able to play solitaire on the fire suppression system, should you?

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. The obvious downside is that if this is put in the hands of the user or administrator, they could always get it wrong.

The typical desktop or mobile OS goes somewhere in between; a regular user can typically start processes and even, to some extent, choose their priorities. Administrators can set certain elements of scheduling policy, but maybe not everything, so perhaps they cannot change the scheduling algorithm.

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 to complete as many tasks as it can, with priority given to the hard real-time tasks.

Scheduling Is Central

Having examined a little of why real-time systems are different than a typical system, we recognize that scheduling is critical to making it work. Choosing a bad scheduling routine where important things are ignored guarantees failure, obviously. However, the kind of scheduling algorithms we have examined so far are not suitable to the task of real-time systems.

In fact, for a real-time system, fairness and minimizing average response times are not important; the real objective is to ensure that all of the hard real-time tasks complete before their deadline and as many as possible of the soft real-time tasks [Sta18]. If the right conditions are present, then success is possible but it is not guaranteed: even an optimal scheduling algorithm cannot make a task that takes 120 seconds complete in 90 seconds, nor can it find a way to complete 400 tasks per day when the system has a maximum throughput of 350.

Any of the non-preemptive scheduling algorithms will not be suitable for a real-time system, because the whole idea is based around prioritization of hard real-time tasks. If one of those arrives while an unimportant task is in progress, it's not sensible for the high priority task to wait until the currently-executing one gets blocked, voluntarily yields the CPU, or exits. Similarly, some routine based solely around time slices probably does not work either, since it would not be optimal to force the higher priority task to wait for up to a full timeslice before it gets a turn. A more realistic approach is immediate preemption, which is just a fancy way of saying that as soon as something more important comes along, the system drops what it is doing to deal with it.

You might wonder why that's so important to drop everything, if we can make timeslices nearly arbitrarily small? Yes, but sometimes the speed of the response we need is in the order of microseconds or a few milliseconds, and there are big consequences for being late, so we need a strategy that recognizes that.

What Kind of Task is It?

There are four kinds of task that are relevant to our discussion: fixed instance, periodic, aperiodic, and sporadic. Each task can be categorized into those four kinds based on both how often it runs and how much time it takes to execute when it does run.

Fixed-Instance Tasks. A fixed-instance task is something that executes a fixed number of times (hence the name) and that is frequently just once, for initialization or cleanup purposes [HZMG20]. One-off tasks like initializing the database are important, but the analysis that we want to consider is what happens at runtime steady-state (after initialization but before a shutdown is called).

Periodic Tasks. If a 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 [HZMG20]:

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

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 8:30-9:50, course ECE 350 (a "task", if you will) takes place in classroom E7 5353 (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. But we'll come back to the subject of this sort of scheduling soon enough.

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.

Aperiodic Tasks. 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) [HZMG20].

Sporadic Tasks. 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?) [HZMG20].

It Takes How Long?

Up until this point we have talked about execution time of a task as if it's a known, fixed quantity, but where do those come from? When the goal is to ensure that hard deadlines are met, we need to know how long tasks take, assuming the worst-case scenario. And no matter how unlikely the worst-case scenario is, that's the value we choose. We'll consider two ways to estimate the worst-case execution time: source code analysis and empirical testing.

Whatever strategy we choose, we want to over-estimate the expected execution time or tasks. Are there tradeoffs to assuming the worst will happen? Certainly! It's possible that we would calculate that we'd need much more capacity than is actually needed during normal operation. That might mean spending more money or buying a more powerful device and reducing the battery life [HZMG20]. Is it bad to have more capacity than you need?

If you've ever been unfortunate enough to have a bad experience with an airline where one cancelled flight results in all sorts of chaos for everyone trying to travel, it's partly caused by the fact that airlines plan their operations such that there's no (or minimal) additional capacity. Thus, even a minor disruption can easily result in a cascade of problems that affect passengers whose trip is unrelated. That's how you end up with your Calgary to Vancouver flight cancelled or delayed because it's snowing in Nova Scotia; the flight crew that was supposed to travel from Halifax to Calgary didn't leave on time and there's no extra flight crew sitting in Calgary to take you to Vancouver. Thus, it doesn't matter that the weather in both your departure and destination locations is sunny and clear. Air travel is important, to be sure, but imagine that we're talking about a life or safety-critical system. Building in some extra capacity is vastly preferable to the alternative.

Code Analysis. One approach for estimating the runtime is through code analysis. This looks at the algorithm and the possible inputs and execution paths and trying to figure out how long the longest path could take since we could work out execution times of each step. That will be an overestimate since it won't account for things like CPU pipelining or a compiler optimization that improves the actual runtime [HZMG20].

Some things that we might do fairly frequently in a non-real time system make it difficult or impossible to accurately estimate the runtime. If you take a look at, say, the NASA/JPL coding guidelines[2], you'll find that they have important rules like: (1) no recursion and no `goto` statements; (2) loops must have a fixed-bound; (3) no dynamic memory allocation after initialization. In addition to each of these things being things that are easy to get wrong, they also qualify as things that make it hard to estimate how long a function is going to take.

The restriction around recursion as well as the fixed-iterations loops make sense, since it's hard to estimate how long a function will take if we don't know how many iterations of the loop are required. The memory one looks a little bit strange though, what's wrong with allocation of memory? As with numerous other system calls, it's hard to know exactly how long it will take for `malloc` or `free` to carry out your request. Is it usually pretty fast? Yes, but this is worst-case analysis, and because `malloc` promises nothing and may have to scour all of memory to find something, it could take arbitrarily long. Oh dear. Remember how we talked about the binary buddy routine?

Empirical Observation. The other approach is to measure instead of estimate. If you already have a version n of the system and want to build version $n + 1$ it might be easy to look at existing data and extrapolate, but if you want to build a new system then you'll need to do a simulation. How to effectively do a simulation could be a whole course on its own, but suffice it to say that if your simulation makes incorrect assumptions or is otherwise constructed poorly, the data you get will be useless or misleading.

If you do prepare a simulation and run it you'll presumably get some distribution of task runtimes. I don't want to get into the statistical math of it, but based on this, it's possible to estimate using known mathematical techniques the worst case runtime with the *confidence interval*. When we talk about confidence interval, we might say that the maximum time is t with 99% confidence, based on the standard deviation. You might want to push that to 99.99% confidence if required. That will give you the worst-case estimate that you're willing to use.

[2] `https://nasa.github.io/fprime/UsersGuide/dev/code-style.html`

Hurry Up and Do It

Now that we have some understanding of real-time systems, how to classify tasks in them, and even how to estimate how long a task takes, we can finally get to actually scheduling some!

References

[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.

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