ECE 350: Real-Time Operating Systems Spring 2023

Lecture 16 — Real Time Scheduling Algorithms

Jeff Zarnett *2023-03-28*

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. For now we'll go back to the uniprocessor view of the world, because it's simpler – but eventually we must indeed talk about the multiprocessor scenario again.

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.

Rate-Monotonic scheduling is based around the idea of basing priority on the period – tasks that execute more often are given higher priority. Tasks with higher frequency require more frequent attention, and therefore should be given higher priority. So each task needs to be assigned a priority number based on that period and this priority does not change dynamically at runtime. Higher-priority tasks will preempt lower-priority ones as needed.

This algorithm is, however, not perfect in the sense that it's possible to fail to schedule things in such a way that all tasks meet their deadlines even if utilization is less than 1. An example from [HZMG20]: Suppose we want to dynamically schedule n periodic tasks with the form (C_k, τ_k): (1, 4), (3, 7), and (3, 10). If we try to schedule these, we'll find that the third task fails to meet its deadline: task 1 runs, then 2, then 1 again, then 3. So far so good. Then task 2 runs, but it gets interrupted by task 1, so task 1 runs to completion, then 2 resumes, and 3 misses its deadline. But utilization is less than 1 so it should be possible to solve this one: $(1/4 + 3/7 + 3/10 = 137/140)$. Earliest Deadline First would have been able to schedule this such that it met all deadlines. Oops.

To figure out if it's possible to schedule things, there's a test. It turns out that it's very difficult to do this dynamically in real-time, so we have to use a less-precise formula for calculating whether a group of tasks may be scheduled or not. Given that real-time systems are pessimistic and would prefer to say no when it is schedulable rather than say yes when it is not. There are some formulae and proofs linked in the source [HZMG20] but it is easy enough for the computer to calculate: $\Pi_{k=1}^{n}(1 + \frac{c_k}{\tau_k}) \leq 2$.

Remember, though, if it's not possible to guarantee that all deadlines can be fulfilled, it doesn't mean that the tasks are certain to fail to meet their deadlines. Task times are always worst-case, and so things might actually be fine in reality.

Deadline-Monotonic. A variant of this is deadline-monotonic scheduling, in which case priority is assigned based on deadlines. The task with the shortest deadline is assigned the highest priority. It's not that different or interesting, but it is possible to come up with some scenarios that Rate-Monotonic would not find a good solution and Deadline-Monotonic would.

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 (throw away the extra time) 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.

It's also possible to have multiple servers to handle different types of aperiodic tasks. Higher priority tasks could go into one server and lower priority tasks in another one. In the lunch break analogy, you might try to go to the supermarket at lunch time (larger, higher priority task) and do some smaller task (Duolingo?) in a coffee break.

This approach does not affect the ability of the system to complete the hard-deadline tasks, because the background server task runs following a schedule that's sufficiently known and predictable that the other hard-deadline tasks can also be scheduled with certainty that they will complete. Excellent!

The disadvantage of the polling server is around the response time for the aperiodic tasks; it's half the server period plus the average execution time and the only way to improve that is to make the server period shorter [GB95]. That might not be desirable, though, because it increases the overhead significantly.

Variant: Deadline-Deferrable Server. An improved version of this that gives better performance for aperiodic tasks and it's called the *deferrable server*: it's like the polling server, but instead of throwing away the time budget if it's unused at the end of the period and there's nothing to do, save that leftover time in case something arrives [GB95]. Eventually the period expires and at that point the remaining execution time is lost, or the time is exhausted by actually doing tasks.

To be clear, in the polling server situation, if the polling server task runs out of things to do with 40% of its time budget available, the polling server task ends and the remaining time is thrown away. In the deferrable approach, that budget is retained until the end of the period for the deferrable server, so if something comes in during that remaining window, it can start immediately. The advantage here is effectively that an aperiodic task will likely be served faster than they would have been if they arrive a bit later.

Continuing the lunch analogy let's say you have an hour and you use it for lunch and finish eating in 45 minutes. In the polling server approach you go back to work immediately even if your lunch break time isn't used up and your remaining 15 minutes of break time is discarded. In the deferrable approach, your extra 15 minutes of lunch time isn't used up and you could use those 15 minutes later on in the day to take a personal call. Still, when the day is over, if you didn't use it for something else, the remaining 15 minutes is lost. The next day you get another full hour for lunch, but the extra time from the day before didn't carry over. Note that none of this is a substitute for legal advice from an employment lawyer about the rules and regulations around legally required breaks during the workday.

Variant 2: Deadline Sporadic Server. The next idea for a high-priority task that handles the aperiodic is called the sporadic server, and it's like deadline-deferrable, but for two things: (1) instead of losing the extra unused period it can be saved for the future, indefinitely; and (2) the replenishment of the budget forces execution to be spread out more evenly [GB95].

Why does spreading it out matter? The deadline server could theoretically schedule two runs of the server back-to-back, which is convenient for some aperiodic tasks (the ones that are there and ready), but inconvenient for others. When I was doing my first co-op term in Toronto, I would take the Finch West bus to and from work. The schedule said “frequent service” which was TTC-speak for “every ten minutes, or more often”. It was subject to a phenomenon that’s called “bunching” in the literature, by which I mean you could stand at the bus stop for 30 minutes and then three busses would arrive and no more for the next 30. This is convenient if you happen to be standing at the bus stop at the time when the three busses arrive because you can get aboard the least-busy one and ideally have a more pleasant ride. But if you just missed it, it’s going to be a long wait.

As for spreading it out, the basic plan is that the execution time for the server is broken down into chunks. When there’s something to do and the server runs that aperiodic task, the amount of time that it runs is deducted from the available budget of chunks. The decrease can be fractions of a chunk, so, for example, if there are 10 chunks total, the budget is full, and an aperiodic task runs for 1.5, the budget is now at 8.5 chunks. When a chunk is depleted, it’s scheduled for replenishment at a fixed time in the future. That spreads out the replenishment. If chunks are exhausted, the server is suspended until the next period.

This sort of replenishment model is also used in other contexts, like, say, database credits for Amazon AWS database instances. You start out with some number of credits, they are depleted by using the database CPU time above a certain threshold, and replenished at a steady rate over time. There is a cap so you can’t stockpile too many credits.

I would explain this as being a little bit like the hearts system in Duolingo. If you’re not familiar with it, it’s a language learning app with a green owl mascot. You start with five hearts, which is also the maximum number of hearts. If you make a mistake when doing an exercise, you lose a heart. If you run out of hearts, you cannot continue and have to wait for hearts to refill and hearts refill at a rate of about one every four hours. You can pay money to avoid this but that’s a different subject. The key difference here is that in Duolingo the hearts only decrease if you get an exercise wrong; in the deadline sporadic server approach the budget always decreases from running aperiodic tasks.

Variant 3: Deadline Exchange Server. This is an approach builds on the Deadline-Deferrable server, and simplifies the idea of tracking the chunks. Chunks require individual tracking so there’s a lot of overhead of keeping a handle on them. The strategy is that any remaining execution time is discarded when the request queue is empty and the wait for replenishment is based on the last chunk of execution time used [GB95]. This is simpler to implement since there’s only ever one replenishment to track and the replenishment is always the full quota.

But does it work? The simulation studies show that different forms of the period server approach are effective for different workloads. The paper introducing this suggests that you get good average response time by making the period of the server be the same as the shortest period of the regular period tasks [GB95]. But it’s possible to have different servers configured for different aperiodic tasks to give the best balance of prioritization and performance.

Multiprocessor Math

So far when we’ve talked about real-time scheduling, we have not considered the possible complexities introduced by making it multiprocessor. In this section we talk about tasks as the things that need doing, an a job as a specific instance of the task to run.

If there are multiple processors in the system, we need to make a decision about whether tasks have dedicated CPUs or whether they can move between CPUs as needed. There are three interesting decisions that are relevant [MA05]: whether preemption is permitted, whether job migration is permitted, and whether job parallelism is forbidden. The last one is just that a job may execute on at most one processor at any given point in time.

If tasks have dedicated CPUs that means the second assumption is answered with a definite no. In a real-time situation it’s possible to just do manual assignment that says Tasks A and B are on CPU 0, Task B and C are on CPU 1... all the way to saying that aperiodic tasks are assigned CPU 4 and that’s it. Manual assignment makes the multiprocessor situation almost indistinguishable from a uniprocessor situation. It’s possible to just look at each CPU individually and the tasks that are scheduled to run on it and verify that given the requirements of the system, it’s possible to meet all hard deadlines.

If you actually want the computer to schedule the tasks by assigning tasks to processors, what we actually have is a variant of the bin-packing problem which is known to be NP-complete [MA05]: that is, to find the globally optimal solution, it's necessary to try *all* possibilities, but it's possible to get an approximation in polynomial time. Doing so in advance for an all-periodic system is doable, but for systems with sporadic and aperiodic tasks, the cost of doing this math might be prohibitive at runtime.

No migration does mean that there is a possibility that tasks are failing to meet their deadlines on CPU X for lack of CPU time even when CPUs Y and Z have plenty of idle capacity. That feels frustrating. Things are more interesting if there is migration allowed. Migration can alleviate the problem of computing resources being unused but increases the complexity and makes it much harder to predict whether we can guarantee all deadlines will be met.

A simple analysis might say that there's no cost to migrating a task from one CPU to another. We know that this is not true, based on our understanding of the hardware: when a task moves CPU, that new CPU may have none of the relevant pages in its cache, resulting in more cache misses and slower execution – not only of that task, but maybe other tasks as well if this increases contention for the bus. So, do we allow migration or not?

Let's step back a bit and consider our options as enumerated in [GRL$^+$12]. The first is global scheduling where all tasks go in one queue and the scheduler assigns tasks to available CPUs. There is potentially some overhead of managing this single queue, and migrations are allowed. The second option is referred to as *partitioned* scheduling and that's what happens when tasks are statically assigned to a CPU and each CPU manages its own queue. Some research shows here that this approach may only allow 50% utilization of the system. The reasoning is that The paper introduces third option is called semi-partitioned scheduling, where most tasks are fixed to specific processors but others are allowed to move. It's probably somewhat intuitive that this mixed approach works well because it allows for some flexibility, but not too much.

None of these approaches are actually solutions that ensure we have an optimal scheduling for multicore system. Remember that optimal in this sense means that if it's possible to schedule it such that all tasks meet their deadlines, the system finds a way. What we need is a different approach...

P-Fairness. The P in P-Fair scheduling stands for "proportional" and the goal is to allocate CPU time in such a way that tasks make progress at steady rates, or in a less-nice way of phrasing it, tasks are forced to proceed at proportionate rates. More formally, an application can request time x_i time units every y_i time quanta (time slices) and the system guarantees that over any T quanta (greater than zero) then a continuously-running application receives between $\lfloor x_i/y_i \times T \rfloor$ time and $\lceil x_i/y_i \times T \rceil$ quanta of service [CAS01]. In other words, time is divided up into little pieces and each process gets a proportional share of the CPU time and is never more than one timeslice away from the amount of time that it "should" receive.

The scheduling algorithm as defined in [BCPV96] is simple enough to explain once appropriate terms are defined. The term *lag* is used to reflect the difference between the actual allocated time and the time a task should have. If lag is greater than zero, the task is behind on the CPU time it should have had; if lag is below zero, the task has had more CPU time than it should normally have. A task is considered *urgent* if lag is greater than 0 and if the lag would exceed 1 full quantum if that task doesn't run in the next quantum. A task is *tnegru*[1] its lag is negative and would remain negative even if the task does not run during the next quantum. If a task is neither urgent nor tnegru, we'll call it "other". Then the scheduling algorithm consists of three parts:

1. Schedule all urgent tasks.
2. Do not schedule tnegru tasks.
3. Schedule other tasks in order of highest lag to lowest until capacity is filled.

In short, to make sure that the lag for any task is always between -1 and $+1$, the scheduler prioritizes the tasks with the highest lag to make sure they do not exceed 1. It also deliberately ignores tasks with negative lag so

[1] I hate this word, but this is "urgent" backwards. I wish it were something else but that's what's in the literature and we're stuck with it now.

that their lag increases and brings them back closer to the zero point where it's trying to keep every task. If it's possible to find an appropriate schedule where all tasks meet their deadlines, the algorithm not only will find it, but execution is fair in the sense that all tasks make progress at around the same rate.

Scheduling in a P-Fair approach is much more tractable because each task is broken down into small slices and each subtask executes on the CPU when it can (and must be before its deadline). This avoids some of the complexity of the bin-packing problem because subtasks just go where they can go and there's limited wasted space. In fact, the original paper [BCPV96] shows that a P-fair schedule is always periodic, which means it's quite suitable for real-time applications.

There exist a number of papers building on the P-Fair approach to address some of its shortcomings and adapt it to some more scenarios. One possible disadvantage of the system is that it requires a lot of computation whenever one of the timeslices ends – updating the lag of every process and reclassifying which processes fall into which category.

But is it worth it? To wrap it up, is P-Fairness a worthwhile algorithm for real-time scheduling? I'll argue yes for a multicore system. Other approaches have worst-case behaviour meaning that utilization is limited (in some cases to as little as 50%) to ensure that even in the very worst-case scenario that all deadlines will be met. Whatever the overhead of the calculations in P-Fairness, surely it's not 50% of the CPU capacity...

References

[BCPV96] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996. `doi:10.1007/BF01940883`.

[CAS01] A. Chandra, M. Adler, and P. Shenoy. Deadline fair scheduling: bridging the theory and practice of proportionate pair scheduling in multiprocessor systems, 2001. `doi:10.1109/RTTAS.2001.929861`.

[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`.

[GRL$^+$12] Joël Goossens, Pascal Richard, Markus Lindström, Irina Iulia Lupu, and Frédéric Ridouard. Job partitioning strategies for multiprocessor scheduling of real-time periodic tasks with restricted migrations. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, RTNS '12, page 141–150, New York, NY, USA, 2012. Association for Computing Machinery. `doi:10.1145/2392987.2393005`.

[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.

[MA05] Arezou Mohammadi and Selim G. Akl. Scheduling algorithms for real-time systems. Technical report, Queen's University, 2005.