Lecture 28 — Real-Time Scheduling, Windows & UNIX

Jeff Zarnett

`jzarnett@uwaterloo.ca`

Department of Electrical and Computer Engineering
University of Waterloo

April 17, 2018

Part I

Real-Time Scheduling

Real-Time scheduling is just scheduling for real-time systems.

But what is a real-time system?

Supposed to respond to events within some real (wall-clock) time.

There are deadlines, and there are consequences for missing deadlines.

Fast is not as important as predictable.

Hard real-time: 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.

If a task is hard real-time, there are two scenarios in which it might not complete before 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?

The second scenario is that at the time of starting, completion was possible.

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.

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.

Remember Java's "stop the world" garbage collector scenario.

If a process/task recurs at regular intervals, it is periodic.

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:

1. τ_k, the period (how often the task occurs); and

2. c_k, the computation time (how long, in the worst case, the task might run).

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:

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

If $U > 1$, it means the system is overloaded.

If 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 that makes it all work.

If the only tasks in the system are periodic ones, create a fixed schedule.

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).

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).

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.

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?)

So, these aperiodic and sporadic tasks really mess with timeline scheduling.

This unpredictability makes it hard to create a simple timetable and follow it.

If we have pre-emptive scheduling, then we can examine 2 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.

The earliest deadline first algorithm is, presumably, very familiar to students.

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 start.

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 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 and another task arrives with a sooner deadline, the currently executing task should be suspended and the new task scheduled.

This may mean a periodic task being preempted by an aperiodic/sporadic task.

A similar algorithm to earliest deadline first, is least slack first.

Slack: how long a task can wait before it must be scheduled to meet a 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.

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.

Part II

Windows & UNIX Scheduling

In this lecture we will examine how real commercial operating systems schedule their processes and threads.

We will examine UNIX, Linux, and finally Windows scheduling.

We will see what approaches are used and what is interesting/novel about them.

The traditional UNIX scheduling is really ancient; as in System V R3 and BSD 4.3. It was replaced in SVR4 (which had some real-time support).

Multilevel feedback system using Round Robin within each of the queues.

Time slicing is implemented and the default time slice is a (very long) 1 second.

So if a process does not block or complete within 1 s, then it will be preempted.

Priority is based on the process type as well as the execution history.

Processor utilization for a process j is calculated for an interval i as:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

And the priority is for process j at interval i is calculated by the formula:

$$P_j(i) = B_j + \frac{CPU_j}{2} + N_j$$

where B_j is the base priority of process j and N_j is the "nice" value of process j.

The "nice" value is a UNIX way to allow a user to voluntarily reduce the priority of a process to be "nice" to other users (but honestly, who uses this?).

Actually, the answer to that question is: system administrators.

An admin can "re-nice" a process and make it somewhat nicer than it would otherwise be.

The *CPU* and *N* components of the equation are restricted to prevent a process from migrating outside of its assigned category.

A process is assigned to a given category based on what kind of process it is.

To put it in simple terms, the OS puts its own needs first and tries to make the best use of resources it can.

From highest to lowest priority, the categories are:

1. Swapper (move processes to and from disk)
2. Block I/O device control (e.g., disk)
3. File manipulation
4. Character I/O device control (e.g., keyboard)
5. User processes

Yes, unfortunately, user processes get piled at the bottom of the list.

It should provide for efficient use of I/O devices and tends to penalize processor-bound processes at the expense of I/O bound processes.

CPU-bound processes should be able to carry on executing when an I/O-bound process waits for the I/O operation to complete.

When an I/O operation is finished, we would like to start the next I/O operation so the I/O device is not waiting.

This strategy is reasonably effective for a general-purpose, time-sharing operating system.

Windows schedules threads using a priority-based, preemptive scheduling algorithm, ensuring that the highest priority thread runs.

The official name for the selection routine is the dispatcher.

A thread runs until it is preempted, blocks, terminates, or its time slice expires.

If a higher priority thread is unblocked, it will preempt a lower priority thread.

Windows has 32 different priority levels, the regular (priority 1 to 15) and real-time classes (16 to 31).

A memory management task runs at priority 0.

The dispatcher maintains a queue for each of the scheduling priorities and goes through them from highest to lowest until it finds something to do.

If there is nothing else currently ready, the System Idle Process will "run".

There are six priority classes you can set a process to via Task Manager:

1. Realtime
2. High
3. Above Normal
4. Normal
5. Below Normal
6. Low

A process is usually in the Normal class.

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

If a process reaches the end of a time slice, the thread is interrupted.

Unless it is in the real-time category, its priority will be lowered, to a minimum of the base priority of each class.

When a process that was blocked on something (e.g., a wait or I/O), its priority is temporarily boosted (unless it is real-time, in which case it cannot be boosted).

The amount of the boost depends on what the event was.

A process that was waiting for keyboard input gets a bigger boost than one that was waiting for a disk operation, for example.

The operating system also gives a priority to whatever process is running in the selected foreground window.

This is different from foreground vs. background processes, because the definition of a foreground process was one that was user-interactive.

Here, the distinction is which of the user-interactive processes is currently "on top" in the UI.

Not only does this process get a priority boost, but it also gets longer time slices.

It highlights the different heritages of Windows and UNIX...