

Lecture 27 — Scheduling Algorithms, Continued

Jeff Zarnett

Scheduling Algorithms, Continued

Carrying on from last time, we will examine some more scheduling algorithms.

Highest Response Ratio Next

We will introduce a new measure: normalized turnaround time. This is the ratio of the turnaround time (the time waiting plus the amount of time taken to execute) to the service time (the time it takes to execute). The relative amount of time waiting is somewhat more important; we can tolerate longer processes waiting a comparatively longer period of time. The goal, then, of the HRRN strategy, is to minimize not only the normalized turnaround time for each process, but to minimize the average over all processes [Sta14].

The way to calculate the response ratio, R , is by the following formula: $\frac{w+s}{s}$ where w is the waiting time and s is the service time. The service time is, as always, a guess. When it is time to select the next process to run, choose the process with the highest R value. A new process will have a value of 1.0 at the beginning, because it has spent no time waiting (yet). Thus it is not that likely to get selected.

Jobs with a small s , i.e., short jobs, are likely to get scheduled quickly, which is still a positive. In spite of this, the HRRN approach introduces something important we have not yet had: factoring in the age of the process. The term w indicates how much time a process has spent waiting. Thus, a process that has spent a long time waiting will rise in priority, over time, until it gets a turn. So processes will not starve, because even a process that is expected to have a very long s will eventually have a high enough R due to the growth of w .

We still need to have some way of estimating s , which may or may not be simple guessing.

Multilevel Queue

For the most part, until now, we have treated processes more or less equally (except when we have taken the highest priority process). While it might seem very fair, it may not be ideal for a situation where processes behave differently. A desktop or laptop has many processes running, some of which are visible to the user (foreground, or interactive processes) and some of which are not (background and batch processes). We could then apply different scheduling algorithms to different types of process.

The multilevel queue takes the ready queue and breaks it up into several queues. A process can be in one (but only one) of the queues and it is assigned to the queue based on some attribute of the process (priority, memory needs, foreground/background, et cetera). The foreground queue, for example, could be scheduled by Round Robin, and the background by First-Come, First-Served [SGG13].

When there are multiple queues, we also need a way of choosing which of the queues to take from next. The policy we choose depends on goals. We might say some queues have absolute priority over others (e.g., as in the earlier highest priority, period option), or we might have time slicing amongst the queues. This could be balanced evenly (rotate through each) or give more time slices to some queues at the expense of others.

An example of this was the CTSS (Compatible TimeSharing System) that ran on the IBM 7094. The CTSS designers decided that it was ideal to give CPU-Bound processes longer blocks of time to execute so they would not have to spend so much time swapping. They set up multiple queues. In the highest priority class, a process got 1 time slice; in the next one down, a process got 2 time slices; the third class meant 4 time slices, and so on. If a process ran

up against the limit of a time slice (e.g., used the full 2 time slices), it was moved down a class. So it got a lower priority, but when it did get selected to run, it was able to run with a lower chance of being interrupted [Tan08].

Like a few schemes, we have seen so far, this is a ratchet: a process can move down in the priority list, but there does not appear to be a way for it to move up. So a process that needed a lot of CPU early on was going to be punished “forever”. The designers of the system assumed that if the user pressed the Enter key, it might be a sign the process was likely to become interactive (and therefore should move up in priority). Some genius user (there’s always one), figured out that by pressing the enter button repeatedly, his long running processes would finish faster. This was a bit unfair; his processes got priority over the others. But things really broke down when this individual decided to be nice: he told all his friends. And suddenly everyone was doing it and the benefit of the system was lost [Tan08].

Guaranteed Scheduling

And now for something completely different. The idea behind guaranteed scheduling is to promise the users something and then fulfill that promise. We could promise that if there are n users, each gets an equal share ($1/n$) of the CPU time. Or with m processes, each process gets $1/m$ of the CPU time.

To make this happen, the system must keep track of how much CPU time each process has received since its creation. It then considers the how this value compares to the ideal (time since creation divided by n). If a process has a value of 0.5, it means it has had only half the CPU it “should” have received. If it has a value of 2.0, it has had double. So the scheduling algorithm is then to run the process with the lowest score, trying to keep all values as close to 1.0 as we can [Tan08].

Lottery

Feedback

Real Time Scheduling

The discussion above tends to be suitable for regular systems, but unsuitable for real-time systems; those in which there are deadlines and consequences (whether severe or lesser) for failing to complete tasks before those deadlines.

Earliest-Deadline First

[HZM14]

The Idle Task

Sometimes our scheduling algorithm cannot produce a new process to run next because there is, quite frankly, nothing to do.

Bumping the Priority

References

[HZM14] Douglas Wilhelm Harder, Jeff Zarnett, and Vajih Montaghani. *A Practical Introduction to Real-Time Systems for Undergraduate Engineering*. 2014. Online; version 0.14.12.22.

- [SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.
- [Sta14] William Stallings. *Operating Systems Internals and Design Principles (8th Edition)*. Prentice Hall, 2014.
- [Tan08] Andrew S. Tanenbaum. *Modern Operating Systems, 3rd Edition*. Prentice Hall, 2008.