Lecture 27 — Scheduling Algorithms

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

October 16, 2022

Scheduling Algorithms

Earlier we saw one example of a simple scheduling algorithm.

Always choose the highest priority (non-blocked) task, and execute it.

There are many options:

1. Highest Priority, Period
2. First-Come, First-Served
3. Round Robin
4. Shortest Process Next
5. Shortest Job First
6. Smallest Remaining Time
7. Highest Response Ratio Next
8. Multilevel Queue (Feedback)
9. Guaranteed Scheduling
10. Lottery

Scheduling Criteria

The OS may maintain data about each process to be used in making decisions about scheduling:

1. The time spent waiting to run.
2. The time spent executing.
3. The total time of execution.

The first two can be measured; the third must be estimated or supplied.

Desktop OSes do not ask users for estimates, but supercomputers might.

Highest Priority, Period

Implementing this is not difficult; priority queues (one for each priority level).

If a task is not blocked, put it in its appropriate priority queue.

If the process's priority is changed, move it to its new home.

We might have a priority heap, or just one big linked list or array that we keep sorted by priority.

Highest Priority, Period

The flaw in this has already been identified: it is vulnerable to starvation.

A process of relatively low priority may never get the chance to run, because there is always something better to do right now.

Software projects may have bug reports open for years on end because they are never important enough to be addressed.

Highest Priority, Period

In some systems this is a desirable property.

It does not fulfill all short term scheduling criteria (response time, fairness).

It may be suitable for life-and-safety-critical systems.

Example: control a robot arm, to prevent a situation where the robot arm goes through the wall and the building falls down and you're dead.

First-Come, First-Served

This is an obvious algorithm that is simple to implement.

Whichever process requests the CPU first gets the CPU first.

Just imagine a queue of processes in which all processes are equal.

A process enters the queue at the back and whichever process is at the front will be dequeued and get to run.

If the current process finishes or is blocked for some reason, the next ready process is selected.

First-Come, First-Served

This is actually a simplification of the highest priority, period scheme.

It ignores priority altogether.

All processes get a chance to run eventually; low priority processes don’t starve.

FCFS: Wait Times

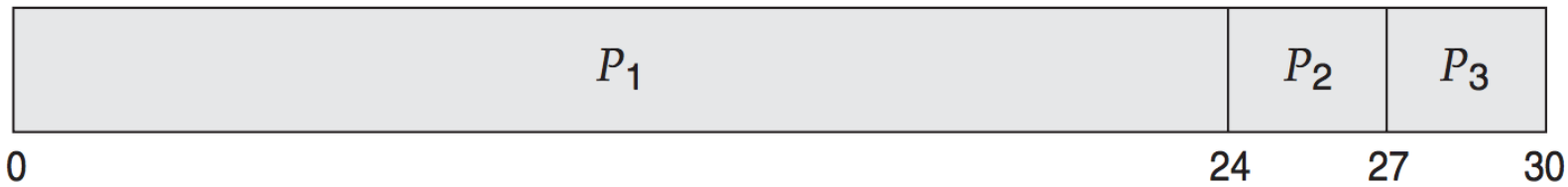
FCFS can result in some undesirable outcomes.

The average waiting time for processes might vary wildly.

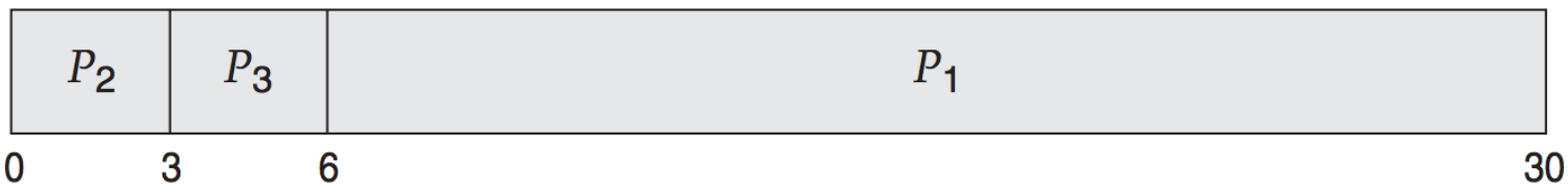
Consider if we have three processes, P_1, P_2, and P_3.

P_1 needs 24 units of CPU time; P_2 and P_3 require 3 units each.

FCFS: Wait Times



Total time: 30 units. Average completion time: 27 units.



Total time: 30 units. Average completion time: 13 units.

FCFS: Favouritism

FCFS also tends to favour CPU-Bound processes.

When a CPU-Bound process is running, the I/O bound processes must wait in the queue like everybody else.

This might lead to inefficient use of the resources; disk is slow so we would like to keep it busy at all times.

With FCFS, however, the I/O devices are likely to suffer idle periods.

FCFS: Preemption

The FCFS algorithm as it is generally described, is non-preemptive.

A process that gets selected from the front of the queue runs until there is a reason to make the swap.

So in theory, one process could monopolize the CPU (remember that some people are jerks).

If we modify FCFS with periodic preemption, then we get Round Robin.

Round Robin

The idea of time slicing has already been introduced.

Every t units of time, a timer generates an interrupt that is the prompt to run the short-term scheduler.

Time slicing itself can be combined with many of the strategies for choosing the next process, but when it is combined with FCFS we get Round Robin.

How Big of a Slice?

The principal issue is: how big should *t* be?

If *t* is too long, then processes may seem to be unresponsive while some other process has the CPU.

Short processes may have to wait quite a while for their turn.

If *t* is too short, the system spends a lot of time handling the clock interrupt and running the scheduling algorithm.

How Big of a Slice?

We could decide about the size of *t* based on the patterns of the system.

Suppose a typical process tends to run for *r* time units before getting blocked.

It would be logical to choose *t* such that it is slightly larger than *r*.

How Big of a Slice?

If *t* is smaller than *r*, processes will frequently be interrupted by the time slice.

Processes that are going to use a lot of CPU will be split up over multiple time slices anyways, but it's frustrating if the process would take 1.1 time slices.

If *t* is larger than *r*, many processes will not run up against that time slice limit.

They will hopefully accomplish a useful chunk of work before getting blocked for I/O or some other reason.

Round Robin

Round Robin tends to favour CPU-Bound processes.

An I/O-Bound process spends a lot less time using the CPU.

It runs for a short time, gets blocked on I/O, then when the I/O is finished, it goes back in the ready queue.

So CPU-Bound processes are getting more of the CPU time.

Virtual Round Robin

Round Robin can be improved to Virtual Round Robin to address this.

It works like Round Robin, but a process that gets unblocked after I/O gets higher priority.

Instead of just rejoining the general queue, there is an auxiliary queue for processes that were previously blocked on I/O.

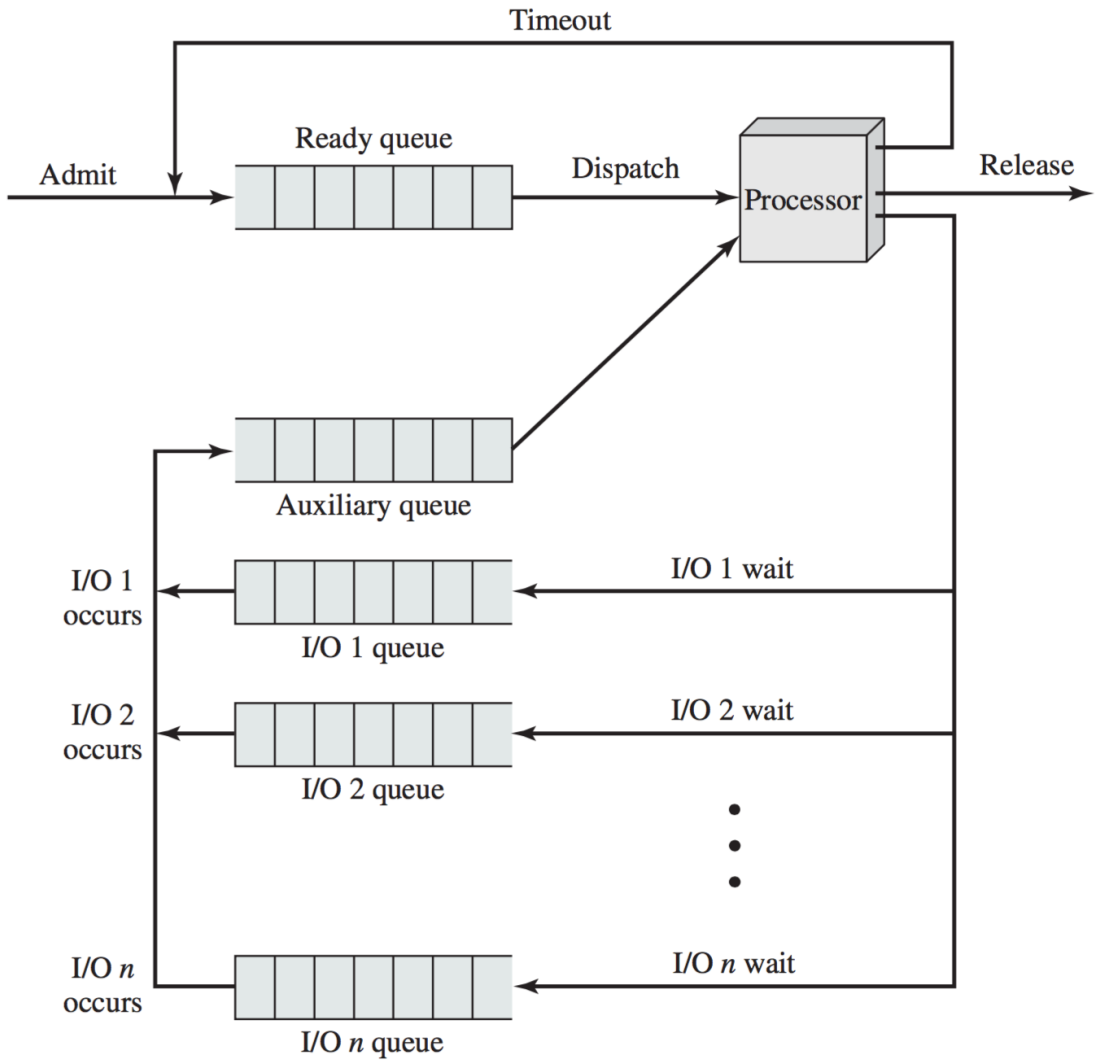
Virtual Round Robin

When the scheduler is choosing a process to run, it takes them from the auxiliary queue if possible.

If a process simply ran up against the time limit, it goes into the regular ready queue instead.

If dispatched from the auxiliary queue, it runs for up to the as-yet-unused fraction of a slice.

Virtual Round Robin



Shortest Process Next

If some information is available about the total length of execution then we may wish to give priority to short processes.

Average time to completion of a task was a lot lower when the shorter processes of P_2 and P_3 ran before P_1.

This means we will get faster turnaround times and better responsiveness, but longer processes may be waiting an unpredictable amount of time.

Shortest Process Next

This is the sort of thing that used to happen in batch job processing on mainframes.

The programmer was asked to give an estimate of the amount of time the program (e.g., compile) would run.

If the programmer's estimate was too low, the program execution would be terminated early.

If the programmer's estimate was too high, the job may never be scheduled to run, or at least, have to wait a very long time.

Shortest Process Next

This might work, but you have noticed that the OS does not ask you, when you start a text editor, roughly how long you expect to be.

Your boss on co-op may not be so accommodating.

This is also a global assessment: how long the whole process takes.

It might be better to worry about the length of CPU bursts and make decisions somewhat more locally…

Shortest Job First

This strategy, unfortunately, is not given quite the right name, but it's the common name for the scheme.

We should call it "shortest next CPU burst".

Then again, the other sciences are not so good at naming things either… the Red Panda is not a Panda at all.

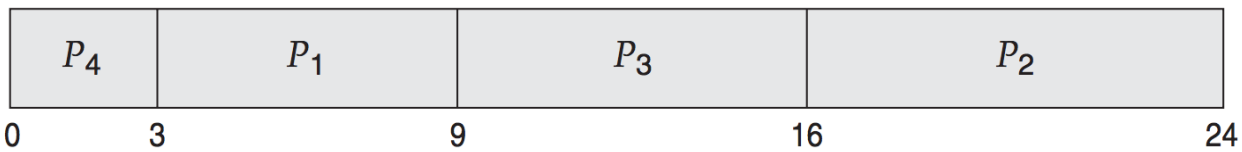
Choose the process that is likely to have the smallest CPU burst.

If two are the same, then FCFS can break the tie (or just choose randomly).

Smallest CPU Burst

Imagine P_1 through P_4, whose predicted burst times are 6, 8, 7, and 3 respectively.

We should schedule then such that the order would be P_4, P_1, P_3, P_2.



The average waiting time is $(3 + 16 + 9 + 0)/4 = 7$ units of time.

Smallest CPU Burst

This is significantly better than the FCFS scheduling.

The algorithm is provably optimal in giving the minimum average waiting time for processes.

Moving a short process up means it finishes faster, and that decreases its waiting time, while moving longer processes back increases their waiting time.

Overall, this scheme is a net positive; the decreases outweigh the increases.

Smallest CPU Burst

The problem is predicting the CPU burst times.

The best thing we may be able to do is gather information about the past and use that to guess about the future.

$$S_{n+1} = \frac{1}{n} \sum_{i=1}^{n} T_i$$

where:

T_i is the burst time for the ith instance of this process;
S_i is the predicted value for the ith instance;
and S_1 is a guess at the first value.

Estimating CPU Burst

Instead of a sum each time, modify the equation to just update the value:

$$S_{n+1} = \frac{1}{n} T_n + \frac{n-1}{n} S_n$$

This routine gives each term in the summation equal weight of $\frac{1}{n}$.

Estimating CPU Burst

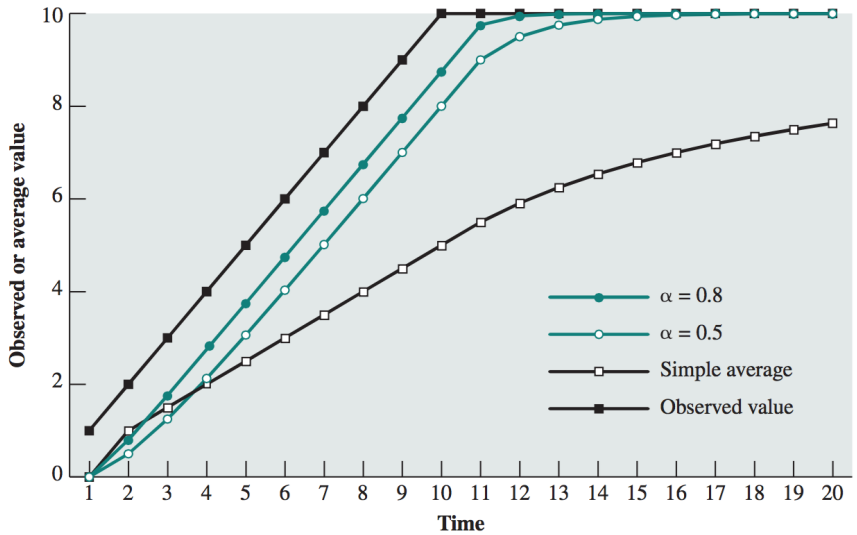
Give greater weight to the more recent values: exponential averaging.

Define a weighting factor α, somewhere between 0 and 1, that determines how much weight the observations are given:

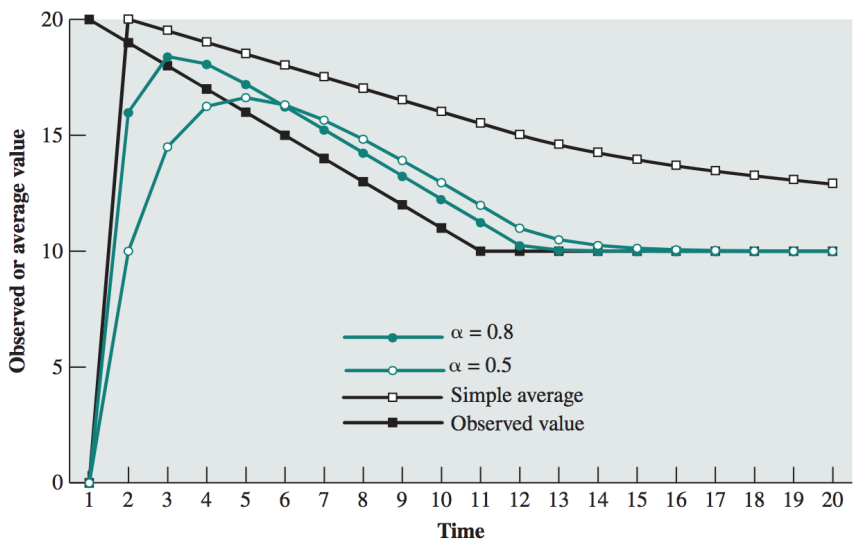
$$S_{n+1} = \alpha T_n + (1 - \alpha) S_n$$

The larger the value of α, the more the recent observations matter.

Exponential Averaging



(a) Increasing function



(b) Decreasing function

Choosing a Starting Value

We might give an estimate of $S_1 = 0$ to start with, which gives new processes priority and a chance to get started.

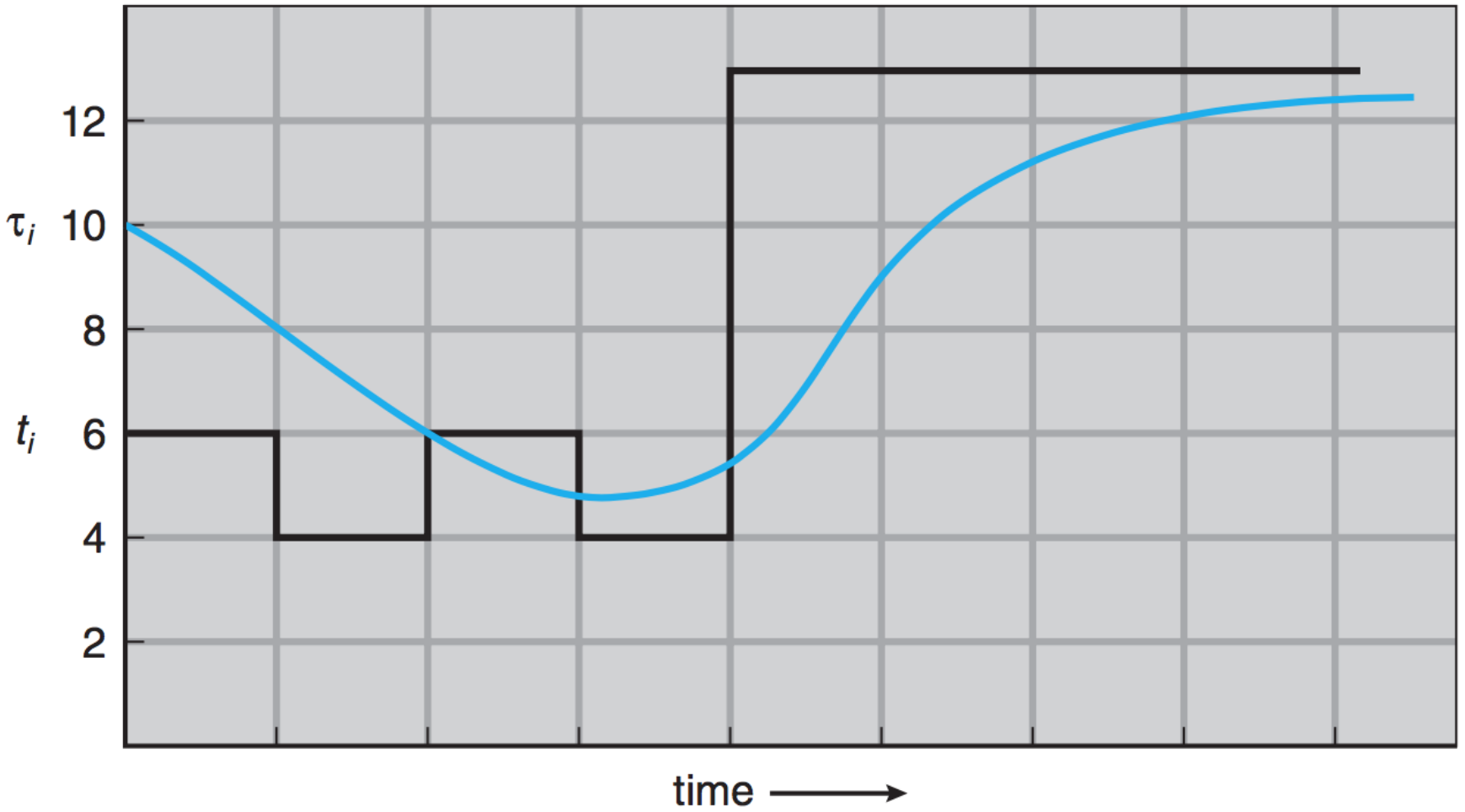
After they have started and run a bit, we will have some data.

But first we have to give them a chance.

There is still a chance that longer processes will starve.

If there is a constant stream of shorter processes, they will continue to get scheduled ahead of a long one…

Predictions and CPU Burst



Shortest Remaining Time

Shortest remaining time is a modification of the preceding strategy, which allows for some additional preemption.

When a new process is scheduled or an old one becomes unblocked, the scheduler will run.

It evaluates if the candidate has a shorter predicted running time than the currently-executing process.

If so, then the candidate will displace the currently-executing one and start running right away.

Shortest Remaining Time

As with Shortest Job First, there is a chance that long processes will starve because of a steady stream of shorter processes.

If we choose S_1 to be zero for new processes, it means they will always preempt the running process.

This may or may not be desirable.

Shortest Remaining Time Slicing

One advantage: we no longer need to have time slicing.

Instead of interrupting the running process every *t* units of time, the other interrupts will be the prompts to run the scheduler.

Thus, the system does not spend any time handling the clock interrupts, which will be a performance increase.

Handling the clock interrupt is not expensive, but even an inexpensive operation, done a million times, will eventually add up…

Batch and Interactive

So far the scheduling routines we have looked at are more suitable to batch processing systems than to interactive desktop systems.

The remaining scheduling algorithms in the list will look a lot more like what we expect to see on our laptops and phones…