6

Scheduling: Introduction

By now you should understand the basic machinery of running processes, including how to context-switch between processes and the details therein. Thus, the low-level **mechanisms** should be clear.

However, we have yet to understand the high-level **policies** that the OS scheduler employs. In this note, we will do just that, presenting a series of scheduling policies (sometimes called disciplines) that people have developed over the years.

We will now develop some scheduling policies that have been put forth through the years. The origins of scheduling, in fact, predate computer systems, as early approaches were taken from the field of operations management and applied to computer systems. This should be no surprise: assembly lines and many other human constructions also require scheduling.

6.1 Workload Assumptions

Before getting into the range of possible policies, let us first make a number of simplifying assumptions about the processes running in the system, sometimes collectively called the **workload**. These assumptions are unrealistic, but that is OK: we will relax them as we go and eventually develop a **fully-operational**

DESIGN TIP: SEPARATION OF POLICY/MECHANISM
In many operating systems, a common design paradigm is to separate high-level policies from their low-level mechanisms [1]. You can think of the mechanism as providing the answer to a *how* question about a system; for example, *how* does an operating system perform a context switch? The policy provides the answer to a *which* question; for example, *which* process should the operating system run right now? Separating the two allows one easily to change policies without having to rethink the mechanism and is thus a form of **modularity**, a general software design principle.

scheduling discipline[1].

We will make the following assumptions about the processes that are running in the system:

1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. All jobs only use the CPU (i.e., they perform no I/O)
4. The run-time of each job is known.

We said all of these assumptions were unrealistic, but, as in Orwell's *Animal Farm*, some assumptions are more unrealistic than others. In particular, it might bother you that the run-time of each job is known: this would make the scheduler omniscient, which, although it would be great (probably), is not likely to happen anytime soon.

6.2 Scheduling Metrics

Beyond making workload assumptions, we also need one more thing to enable us to compare different scheduling policies: a **scheduling metric**. A metric is just something that we

[1]Said in the same way you would say "A fully-operational Death Star."

use to *measure* something, and of course there a number of different metrics that make sense in scheduling.

For now, however, let us also simplify our life by simply having a single metric: **turnaround time**. The turnaround time of a job, is defined as the time at which the job finally completes minus the time at which the job arrived in the system. More formally, the turnaround time $T_{turnaround}$ is:

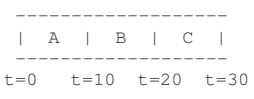
$$T_{turnaround} = T_{completion} - T_{arrival} \tag{6.1}$$

Because we have assumed that all jobs arrive at the same time, for now $T_{arrival} = 0$ and hence $T_{turnaround} = T_{completion}$. This fact will change as we relax the aforementioned assumptions.

6.3 First In, First Out (FIFO)

The most basic algorithm a scheduler can implement is known as **First In, First Out (FIFO)** scheduling (sometimes also called **First Come, First Served (FCFS)**). FIFO has a number of positive properties: it is clearly very simple and thus easy to implement. And given our assumptions, it works pretty well.

Let's do a quick example together. Imagine three jobs arrive in the system, A, B, and C, at roughly the same time ($T_{arrival} = 0$). Because FIFO has to put some job first, let's assume that while they all arrived simultaneously, A arrived just a hair before B which arrived just a hair before C. Assume also that each job runs for 10 seconds. What will the **average turnaround time** be for these jobs?



From the diagram above, you can see that A finished at 10, B at 20, and C at 30. Thus, the average turnaround time for

the three jobs is simply $\frac{10+20+30}{3} = 20$. Computing turnaround time is as easy as that.

But now let's relax one of our assumptions. In particular, let's relax assumption 1, and thus no longer assume that each job runs for the same amount of time. How does FIFO perform now? What kind of workload could you construct to make FIFO perform poorly? (think about this before reading on)

Presumably you've figured this out by now, but just in case, let's do an example to show how jobs of different lengths can lead to trouble for FIFO scheduling. In particular, let's again assume three jobs (A, B, and C), but this time A runs for 100 seconds which B and C run for 10 each.

As you can see from the diagram, A runs first and takes up the full 100 seconds before B or C even get a chance to run. Thus, the average turnaround time for the system is high: 110 seconds.

This problem is generally referred to as the **convoy effect**, where a number of relatively-short potential consumers of a resource get queued behind a heavyweight resource consumer [2]. This might remind you of a single line at a grocery store and what you feel like when you see the person in front of you with three carts full of provisions and their checkbook out; it's going to be a while.

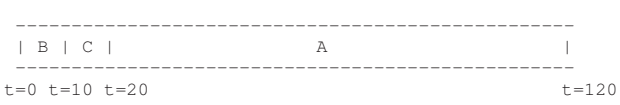
So what should we do? How can we develop a better algorithm to deal with our new reality of jobs that run for different amounts of time? Think about it first; then read on.

6.4 Shortest Job First (SJF)

It turns out that a very simple approach solves this problem; in fact it is an idea stolen from operations research [3] and applied to scheduling of jobs in computer systems. This new

scheduling discipline is known as **Shortest Job First (SJF)**, and the name should be easy to remember because it describes the policy quite completely: it runs the shortest job first, then the next shortest, and so on.

Let's take our example above but with SJF as our scheduling policy:

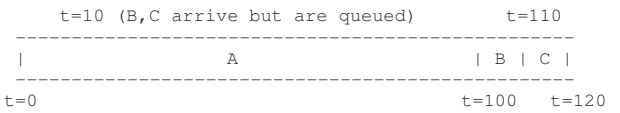


Hopefully the diagram makes it clear why SJF performs much better with regards to average turnaround time. Simply by running B and C before, A, SJF reduces average turnaround from 110 seconds to 50, more than a factor of two improvement.

In fact, given our assumptions about jobs all arriving at the same time, we could prove that SJF is indeed an **optimal** scheduling algorithm. A proof of this might use **contradiction**, i.e., assume that SJF does not lead to an optimal average turnaround time and then show that this assumption leads to a contradiction. However, you are in a systems class, not theory; no proofs are allowed.

Thus we arrive upon a good approach to scheduling with SJF, but our assumptions are still fairly unrealistic. Let's relax another. In particular, we can target assumption 2, and now assume that jobs can arrive at any time instead of all at once. What problems does this lead to? (think about it again)

Here we can illustrate the problem again with an example. This time, assume A arrives at $t = 0$ and needs to run for 100 seconds, whereas B and C arrive at $t = 10$ and each need to run for 10 seconds. With pure SJF, we'd get the following schedule:



PRINCIPLE OF SJF
Shortest Job First represents a general scheduling principle that can be applied to any system where the perceived turnaround time per customer (or, in our case, process/job) matters. Think of any line you have waited in: if the establishment in question cares about customer satisfaction, it is likely they have taken SJF into account. For example, grocery stores commonly have a "ten-items-or-less" line to ensure that shoppers with only a few things to purchase don't get stuck behind the family of eight restocking for the winter. You can probably think of related real-life examples of the SJF principle applied to make people happier; what are they?

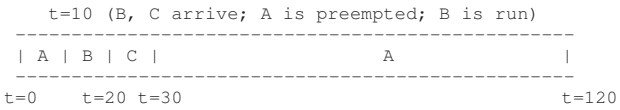
As you can see from the figure, even though B and C arrived shortly after A, they still are forced to wait until A has completed, and thus suffer the same convoy problem; average turnaround time for these three jobs is 103.33 seconds (A: 100; B: 110-10=100; C: 120-10=110). What can a scheduler do?

6.5 Shortest Time-to-Completion First (STCF)

As you might have guessed, given our previous discussion about mechanisms such as timer interrupts and context switching, the scheduler can certainly do something else when B and C arrive: it can **preempt** job A and decide to run another job. SJF by our definition is a **non-preemptive** scheduler, and thus suffers from the problems described above.

Fortunately, there is a scheduler which does exactly that: add preemption to SJF. It is known as the **Shortest Time-to-Completion First (STCF)** scheduler, and any time a new job enters the system, it determines of the remaining jobs and new job, which has the least time left, and then schedules that one. Thus, in our example, STCF would preempt A and run B and C to completion; only when they are finished would A's remaining time be

scheduled.



The result is a much-improved average turnaround time: 50 seconds (A: 120; B: 20-10=10; C: 30-10=20). And as before, given our new assumptions, STCF is provably optimal; given that SJF is optimal if all jobs arrive at the same time, you should probably be able to see the intuition behind the optimality of STCF.

Thus, if we knew that job lengths, and jobs only used the CPU, and our only metric was turnaround time, STCF would be a great policy. In fact, for a number of early batch computing systems, these types of scheduling algorithms made some sense. However, the introduction of time-shared machines changed all that. Now users would sit at a terminal and demand interactive performance from the system as well. And thus, a new metric was born: **response time**.

Response time is defined as the time from when the job arrives in a system to the first time it is scheduled. More formally:

$$T_{response} = T_{firstrun} - T_{arrival} \tag{6.2}$$

For example, if we had the schedule above (with A arriving at time 0, and B and C at time 10), the response time of each job is as follows: 0 for job A, 0 for job B, and 10 for job C (average: 3.33).

As you might be thinking, STCF and related disciplines are not particularly good for response time. If three jobs arrive at the same time, for example, the third job has to wait for the previous two jobs to run *in their entirety* before being scheduled just once. While great for turnaround time, this approach is quite bad for response time and interactivity. Indeed, imagine sitting a terminal, typing, and having to wait 10 seconds to see a response from the system just because some other job got scheduled in front of yours: not too pleasant a thought.

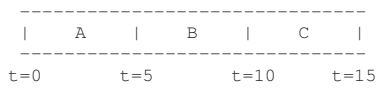
Thus, we are left with another problem: how can we build a scheduler that is sensitive to response time?

6.6 Round Robin

To begin to solve this problem, we will introduce a new scheduling algorithm. This approach is known as **Round-Robin (RR)** scheduling.

The basic idea is simple: instead of running jobs to completion, RR runs a job for a **time slice** (sometimes called a **scheduling quantum**) and then switches to the next job in the run queue. It repeatedly does so until the jobs are finished. For this reason, RR is sometimes called **time-slicing**. Note that the length of a time slice must be a multiple of the timer-interrupt period; thus if the timer interrupts every 10 milliseconds, the time slice could be 10, 20, or any other multiple of 10 ms.

To understand RR in more detail, let's look at an example. Assume three jobs A, B, and C arrive at the same time in the system, and that they each wish to run for 5 seconds. An SJF scheduler would lead to this schedule:



In contrast, RR with a time-slice of 1 second would cycle through the jobs quite quickly:

The average response time of RR is: $\frac{0+1+2}{3} = 1$; for SJF, average response time is: $\frac{0+5+10}{3} = 5$.

As you can see, the length of the time slice is critical for RR. The shorter it is, the better the performance of RR under the

DESIGN TIP: AMORTIZATION
The general technique of **amortization** is commonly used in systems when there is a fixed cost to some operation. By incurring that cost less often (i.e., by performing the operation fewer times), the total cost to the system is reduced. For example, if the time slice is set to 10 ms, and the context-switch cost is 1 ms, roughly 10% of time is spent context switching and is thus wasted. If we want to *amortize* this cost, we can increase the time slice, e.g., to 100 ms. In this case, less than 1% of time is spent context switching, and thus the cost of time-slicing has been amortized.

response-time metric. However, making the time slice too short is problematic: suddenly the cost of context switching will dominate overall performance. Thus, deciding on the length of the time slice presents a trade-off to a system designer, making it long enough to **amortize** the cost of switching without making it so long that the system is no longer responsive.

Note that the cost of context switching does not arise solely from the OS actions of saving and restoring a few registers. When programs run, they build up a great deal of state in CPU caches, TLBs, branch predictors, and other on-chip hardware. Switching to another job causes this state to be flushed and new state relevant to the currently-running job to be brought in, which may exact a noticeable performance cost [4].

RR, with a reasonable time slice, is thus an excellent scheduler if response time is our only metric. But what about turnaround time? Let's look at our example above again. A, B, and C, each with running times of 5 seconds, arrive at the same time, and RR is the scheduler with a (long) 1-second time slice. We can see from the picture above that A finishes at 13, B at 14, and C at 15, for an average of 14. Pretty awful!

It is not surprising, then, that RR is indeed one of the *worst* policies if turnaround time is our metric. Intuitively, this should

make sense: what RR is doing is stretching out each job as long as it can, by only running each job for a short bit before moving to the next. Because turnaround time only cares about when jobs finish, RR is nearly pessimal, even worse than simple FIFO in many cases.

Thus far we have developed two types of schedulers. The first type (SJF, STCF) optimizes turnaround time, but is bad for response time. The second type (RR) optimizes response time but is bad for turnaround. And we still have two assumptions which need to be relaxed: assumption 3 (that jobs do no I/O), and assumption 4 (that the run-time of each job is known). Let's tackle those assumptions now.

6.7 Incorporating I/O

First we will relax assumption 3. Of course all programs perform I/O. Imagine a program that didn't take any input: it would produce the same output each time. Imagine one without output: it is the proverbial tree falling in the forest without an observer; it doesn't matter that it ran.

A scheduler clearly has a decision to make when a job initiates an I/O request, because the currently-running job won't be using the CPU during the I/O; it is **blocked** waiting for I/O completion. If the I/O is sent to a hard disk drive, the process might be blocked for 10 or 20 ms. Thus, the scheduler should probably schedule another job on the CPU at that time.

The scheduler also has to make a decision when the I/O completes. When that occurs, an interrupt is raised, and the OS runs and moves the process that issued the I/O from blocked back to the ready queue. Of course, it could even decide to run the job at that point. Thus, with these possibilities, how should the OS treat each job?

To understand this issue better, let us assume we have two jobs, A and B, that each need 1 second of CPU time. However, there is one obvious difference: A runs for 100 ms and then issues an I/O request that takes 100 ms, whereas B simply uses

the CPU and performs no I/O. A pictorial depiction of A and B yields:

```
CPU   |A| |A| |A| |A| |A| |A| |A| |A| |A| |A|
Disk    |A| |A| |A| |A| |A| |A| |A| |A| |A| |A|

CPU   |         B         |
Disk
```

Assume we are trying to build a STCF scheduler. How should such a scheduler account for the fact that A is broken up into 10 100-ms sub-jobs, whereas B is just a single 1-second CPU demand?

A common approach is to treat each 100-ms sub-job of A as an independent job. Thus, when the system starts, its choice is whether to schedule a 100-ms A or a 1-second B. With STCF, the choice is clear: choose the shorter one, in this case A. Then, when the first sub-job of A has completed, only B is left, and it begins running. Then a new sub-job of A is submitted, and it preempts B and runs for 100 ms. And so on. The outcome would look like this:

```
CPU   |A|B|A|B|A|B|A|B|A|B|A|B|A|B|A|B|A|B|A|B|A|B|
Disk    |A| |A| |A| |A| |A| |A| |A| |A| |A| |A| |A|
```

6.8 No More Oracle

With a basic approach to I/O in place, we come to the final assumption: that the scheduler knows the length of each job. As we said before, this is likely the worst assumption we could make. In fact, in a general-purpose OS (like the ones we care about), the OS usually knows very little about the length of each job. Thus, how can we build an approach that behaves like SJF/STCF without such *a priori* knowledge? Further, how can we incorporate some of the ideas we have seen with the RR scheduler so that response time is also quite good? Alas, these mysteries will remain until the next note.

6.9 Summary

We have introduced the basic ideas behind scheduling and developed two families of approaches. The first runs the shortest job remaining and thus optimizes turnaround time; the second alternates between all jobs and thus optimizes response time. Both are bad where the other is good, alas. We have also seen how we might incorporate I/O into the picture, but have still not solved the problem of the fundamental inability of the OS to see into the future. Shortly, we will see how to overcome this problem, by building a scheduler that uses the past to predict the future. This scheduler is known as the **multi-level feedback queue**, and it is the topic of the next note.

References

[1] "Policy/mechanism separation in Hydra"
R. Levin, E. Cohen, W. Corwin, F. Pollack, W. Wulf.
SOSP 1975.

[2] I'm still looking for a good reference on this, other than some book. Where was it first discovered?

[3] "Machine Repair as a Priority Waiting-Line Problem"
Thomas E. Phipps Jr. and W. R. Van Voorhis
Operations Research, Vol. 4, No. 1 (Feb. 1956), pp. 76-86.
Interestingly, the earliest reference to shortest-job first I can find is to this paper in an operations research journal. The idea clearly pre-dates computer systems!

[4] "The effect of context switches on cache performance"
Jeffrey C. Mogul and Anita Borg
ASPLOS, 1991.