

a **fully-operational scheduling discipline**¹.

We will make the following assumptions about the processes, sometimes called **jobs**, 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 just as some animals are more equal than others in Orwell's *Animal Farm* [O45], some assumptions are more unrealistic than others in this chapter. 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.

7.2 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 use to *measure* something, and there are 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 completes minus the time at which the job arrived in the system. More formally, the turnaround time $T_{\text{turnaround}}$ is:

$$T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}} \quad (7.1)$$

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

You should note that turnaround time is a **performance** metric, which will be our primary focus this chapter. Another metric of interest is **fairness**, as measured (for example) by **Jain's Fairness Index** [J91]. Performance and fairness are often at odds in scheduling; a scheduler, for example, may optimize performance but at the cost of preventing a few jobs from running, thus decreasing fairness. This conundrum shows us that life isn't always perfect.

7.3 First In, First Out (FIFO)

The most basic algorithm a scheduler can implement is known as **First In, First Out (FIFO)** scheduling or sometimes **First Come, First Served**

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

(FCFS). FIFO has a number of positive properties: it is clearly very simple and thus easy to implement. 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?

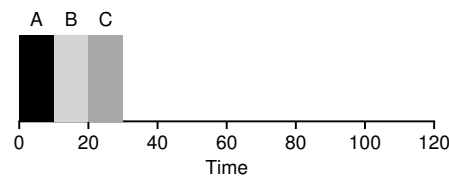


Figure 7.1: FIFO Simple Example

From Figure 7.1, 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.

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 ... keep thinking ... got it?!)

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 while B and C run for 10 each.

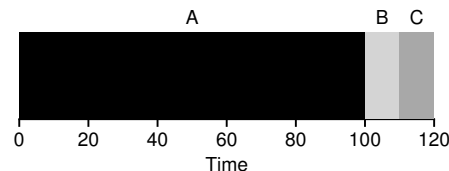


Figure 7.2: Why FIFO Is Not That Great

As you can see in Figure 7.2, Job A runs first for the full 100 seconds before B or C even get a chance to run. Thus, the average turnaround time for the system is high: a painful 110 seconds ($\frac{100+110+120}{3} = 110$).

This problem is generally referred to as the **convoy effect** [B+79], where a number of relatively-short potential consumers of a resource get queued behind a heavyweight resource consumer. This scheduling scenario might remind you of a single line at a grocery store and what you feel like when

TIP: THE 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, a 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 preparing for some upcoming nuclear winter.

you see the person in front of you with three carts full of provisions and a checkbook out; it’s going to be a while².

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.

7.4 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 [C54,PV56] 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.

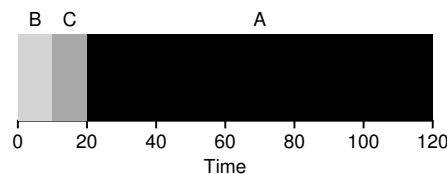


Figure 7.3: SJF Simple Example

Let’s take our example above but with SJF as our scheduling policy. Figure 7.3 shows the results of running A, B, and C. 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 ($\frac{10+20+120}{3} = 50$), 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. How-

²Recommended action in this case: either quickly switch to a different line, or take a long, deep, and relaxing breath. That’s right, breathe in, breathe out. It will be OK, don’t worry.

ASIDE: PREEMPTIVE SCHEDULERS

In the old days of batch computing, a number of **non-preemptive** schedulers were developed; such systems would run each job to completion before considering whether to run a new job. Virtually all modern schedulers are **preemptive**, and quite willing to stop one process from running in order to run another. This implies that the scheduler employs the mechanisms we learned about previously; in particular, the scheduler can perform a **context switch**, stopping one running process temporarily and resuming (or starting) another.

ever, you are in a systems class, not theory or operations research; 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?

(Another pause to think ... are you thinking? Come on, you can do it)

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 schedule seen in Figure 7.4.

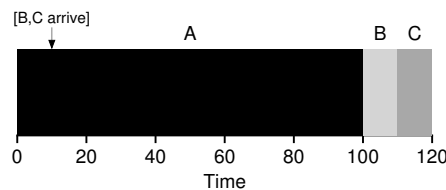


Figure 7.4: SJF With Late Arrivals From B and C

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 ($\frac{100 + (110 - 10) + (120 - 10)}{3}$). What can a scheduler do?

7.5 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, perhaps continuing A later. SJF by our definition is a **non-preemptive** scheduler, and thus suffers from the problems described above.

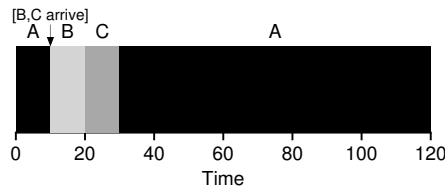


Figure 7.5: STCF Simple Example

Fortunately, there is a scheduler which does exactly that: add preemption to SJF, known as the **Shortest Time-to-Completion First (STCF)** or **Preemptive Shortest Job First (PSJF)** scheduler [CK68]. 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. Figure 7.5 shows an example.

The result is a much-improved average turnaround time: 50 seconds $(\frac{(120-0)+(20-10)+(30-10)}{3})$. 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**.

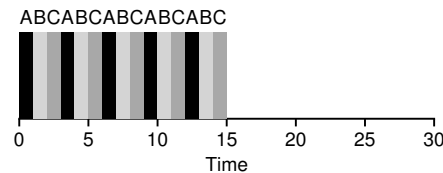
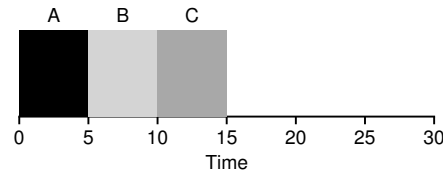
Response time is defined as the time from when the job arrives in a system to the first time it is scheduled. More formally:

$$T_{\text{response}} = T_{\text{firstrun}} - T_{\text{arrival}} \quad (7.2)$$

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 B, and 10 for 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 at 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.

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



7.6 Round Robin

TIP: AMORTIZATION CAN REDUCE COSTS

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.

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 [MB91].

RR, with a reasonable time slice, is thus an excellent scheduler if response time is our only metric. But what about our old friend 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.

More generally, any policy (such as RR) that is **fair**, i.e., that evenly divides the CPU among active processes on a small time scale, will perform poorly on metrics such as turnaround time. Indeed, this is an inherent trade-off: if you are willing to be unfair, you can run shorter jobs to completion, but at the cost of response time; if you instead value fairness, response time is lowered, but at the cost of turnaround time. This type of **trade-off** is common in systems; you can't have your cake and eat it too.

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 next.

7.7 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, with no one to see it; 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 a few milliseconds or longer, depending on the current I/O load of the drive. 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 state. Of course, it could even decide to run the job at that point. How should the OS treat each job?

To understand this issue better, let us assume we have two jobs, A and B, which each need 50 ms of CPU time. However, there is one obvious difference: A runs for 10 ms and then issues an I/O request (assume here that I/Os each take 10 ms), whereas B simply uses the CPU for 50 ms and performs no I/O. The scheduler runs A first, then B after (Figure 7.8).

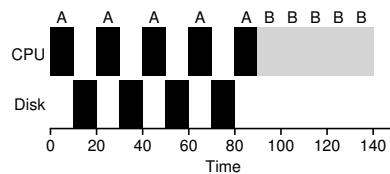


Figure 7.8: **Poor Use of Resources**

Assume we are trying to build a STCF scheduler. How should such a scheduler account for the fact that A is broken up into 5 10-ms sub-jobs, whereas B is just a single 50-ms CPU demand? Clearly, just running one job and then the other without considering how to take I/O into account makes little sense.

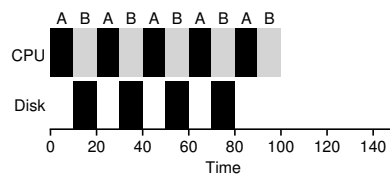


Figure 7.9: **Overlap Allows Better Use of Resources**

A common approach is to treat each 10-ms sub-job of A as an independent job. Thus, when the system starts, its choice is whether to schedule a 10-ms A or a 50-ms 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 10 ms. Doing so allows for **overlap**, with