7

Scheduling: Introduction

By now low-level **mechanisms** of running processes (e.g., context switching) should be clear. However, we have yet to understand the high-level **policies** that an OS scheduler employs. We will now do just that, presenting a series of **scheduling policies** (sometimes called **disciplines**) that people have developed over the years.

The origins of scheduling, in fact, predate computer systems; early approaches were taken from the field of operations management and applied to computers. This should be no surprise: assembly lines and many other human constructions also require scheduling.

THE CRUX: HOW TO DEVELOP SCHEDULING POLICY

How should we develop a basic framework for thinking about scheduling policies? What are the key assumptions? What metrics are important? What basic approaches have been used in the earliest of computer systems?

7.1 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 clearly unrealistic, but that is alright (for now), because we will relax them as we go and eventually develop what we will refer to as ... *(dramatic pause) ...*

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 of course 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 finally completes minus the time at which the job arrived in the system. More formally, the turnaround time $T_{turnaround}$ is:

$$T_{turnaround} = T_{completion} - T_{arrival} \tag{7.1}$$

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.

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

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

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** (**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?

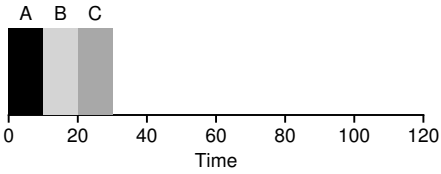

Figure 7.1: FIFO Simple Example

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.

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 while B and C run for 10 each.

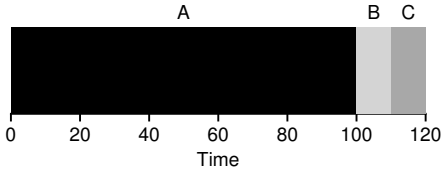

Figure 7.2: Why FIFO Is Not That Great

As you can see from Figure 7.2, Job 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: 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 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.

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.

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.

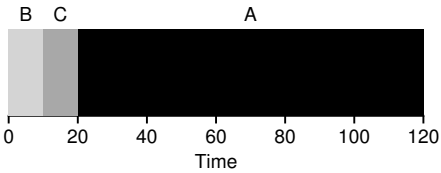

Figure 7.3: SJF Simple Example

In fact, given our assumptions about jobs all arriving at the same time, we could prove that SJF is indeed an **optimal** scheduling algorithm. However, 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? (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 schedule seen in Figure 7.4.

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?

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.

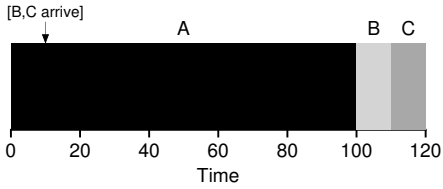

Figure 7.4: SJF With Late Arrivals From B and C

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.

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.

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.

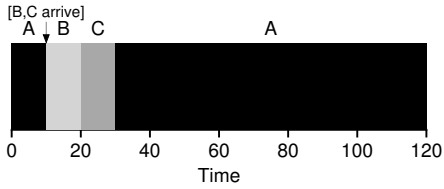

Figure 7.5: STCF Simple Example

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:

$$T_{response} = T_{firstrun} - T_{arrival} \tag{7.2}$$

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?

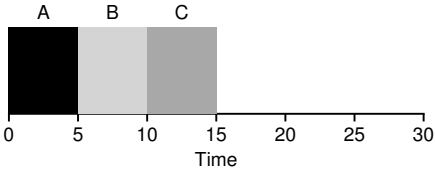

Figure 7.6: SJF Again (Bad for Response Time)

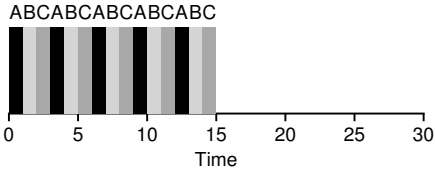

Figure 7.7: Round Robin

7.6 Round Robin

To solve this problem, we will introduce a new scheduling algorithm. This approach is classically known as **Round-Robin (RR)** scheduling [K64]. 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 runs each job to completion before running another (Figure 7.6). In contrast, RR with a time-slice of 1 second would cycle through the jobs quite quickly (Figure 7.7).

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

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.

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 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 [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 runtime 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 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 ms 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. Imagine the scheduler decides to run A first, then B after (Figure 7.8).

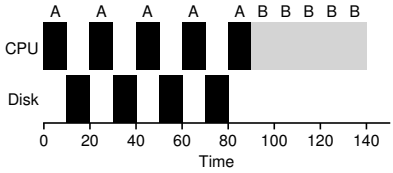

Figure 7.8: Poor Use of Resources

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.

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** to occur, with the CPU being used by one process while waiting for the I/O of another process to complete; the system is thus better utilized (see Figure 7.9).

And thus we see how a scheduler might incorporate I/O. By treating each CPU burst as a job, the scheduler makes sure processes that are "interactive" get run frequently. While those interactive jobs are performing I/O, other CPU-intensive jobs run, thus better utilizing the processor.

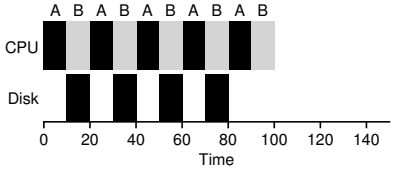

Figure 7.9: Overlap Allows Better Use of Resources

7.8 No More Oracle

With a basic approach to I/O in place, we come to our 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?

TIP: OVERLAP ENABLES HIGHER UTILIZATION

When possible, **overlap** operations to maximize the utilization of systems. Overlap is useful in many different domains, including when performing disk I/O or sending messages to remote machines; in either case, starting the operation and then switching to other work is a good idea, and improved the overall utilization and efficiency of the system.

7.9 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, an inherent trade-off common in systems. 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 recent past to predict the future. This scheduler is known as the **multi-level feedback queue**, and it is the topic of the next chapter.

References

[B+79] "The Convoy Phenomenon"
M. Blasgen, J. Gray, M. Mitoma, T. Price
ACM Operating Systems Review, 13:2, April 1979
Perhaps the first reference to convoys, which occurs in databases as well as the OS.

[C54] "Priority Assignment in Waiting Line Problems"
A. Cobham
Journal of Operations Research, 2:70, pages 70–76, 1954
The pioneering paper on using an SJF approach in scheduling the repair of machines.

[K64] "Analysis of a Time-Shared Processor"
Leonard Kleinrock
Naval Research Logistics Quarterly, 11:1, pages 59–73, March 1964
May be the first reference to the round-robin scheduling algorithm; certainly one of the first analyses of said approach to scheduling a time-shared system.

[CK68] "Computer Scheduling Methods and their Countermeasures"
Edward G. Coffman and Leonard Kleinrock
AFIPS '68 (Spring), April 1968
An excellent early introduction to and analysis of a number of basic scheduling disciplines.

[J91] "The Art of Computer Systems Performance Analysis:
Techniques for Experimental Design, Measurement, Simulation, and Modeling"
R. Jain
Interscience, New York, April 1991
The standard text on computer systems measurement. A great reference for your library, for sure.

[O45] "Animal Farm"
George Orwell
Secker and Warburg (London), 1945
A great but depressing allegorical book about power and its corruptions. Some say it is a critique of Stalin and the pre-WWII Stalin era in the U.S.S.R; we say it's a critique of pigs.

[PV56] "Machine Repair as a Priority Waiting-Line Problem"
Thomas E. Phipps Jr. and W. R. Van Voorhis
Operations Research, 4:1, pages 76–86, February 1956
Follow-on work that generalizes the SJF approach to machine repair from Cobham's original work; also postulates the utility of an STCF approach in such an environment. Specifically, "There are certain types of repair work, ... involving much dismantling and covering the floor with nuts and bolts, which certainly should not be interrupted once undertaken; in other cases it would be inadvisable to continue work on a long job if one or more short ones became available (p.81)."

[MB91] "The effect of context switches on cache performance"
Jeffrey C. Mogul and Anita Borg
ASPLOS, 1991
A nice study on how cache performance can be affected by context switching; less of an issue in today's systems where processors issue billions of instructions per second but context-switches still happen in the millisecond time range.

Homework

ASIDE: **SIMULATION HOMEWORKS**
Simulation homeworks come in the form of simulators you run to make sure you understand some piece of the material. The simulators are generally python programs that enable you both to *generate* different problems (using different random seeds) as well as to have the program solve the problem for you (with the `-c` flag) so that you can check your answers. Running any simulator with a `-h` or `--help` flag will provide with more information as to all the options the simulator gives you.

This program, `scheduler.py`, allows you to see how different schedulers perform under scheduling metrics such as response time, turnaround time, and total wait time. Three schedulers are "implemented": FIFO, SJF, and RR.

There are two steps to running the program.

First, run without the -c flag: this shows you what problem to solve without revealing the answers. For example, if you want to compute response, turnaround, and wait for three jobs using the FIFO policy, run this:

```
./scheduler.py -p FIFO -j 3 -s 100
```

If that doesn't work, try this:

```
python ./scheduler.py -p FIFO -j 3 -s 100
```

This specifies the FIFO policy with three jobs, and, importantly, a specific random seed of 100. If you want to see the solution for this exact problem, you have to specify this exact same random seed again. Let's run it and see what happens. Figure 7.10 shows the output you should see.

As you can see from this example, three jobs are generated: job 0 of length 1, job 1 of length 4, and job 2 of length 7. As the program states, you can now use this to compute some statistics and see if you have a grip on the basic concepts.

Once you are done, you can use the same program to *solve* the problem and see if you did your work correctly. To do so, use the `-c` flag. Figure 7.11 shows the output.

```
prompt> ./scheduler.py -p FIFO -j 3 -s 100
ARG policy FIFO
ARG jobs 3
ARG maxlen 10
ARG seed 100

Here is the job list, with the run time of each job:
  Job 0 (length = 1)
  Job 1 (length = 4)
  Job 2 (length = 7)

Compute the turnaround time, response time, and wait time for each job.
When you are done, run this program again, with the same arguments,
but with -c, which will thus provide you with the answers. You can use
to use -s <somenumber> or your own job list (-l 10,15,20 for example)
to generate different problems for yourself.
```

Figure 7.10: Homework Output

```
prompt> ./scheduler.py -p FIFO -j 3 -s 100 -c
ARG policy FIFO
ARG jobs 3
ARG maxlen 10
ARG seed 100

Here is the job list, with the run time of each job:
  Job 0 (length = 1)
  Job 1 (length = 4)
  Job 2 (length = 7)

** Solutions **

Execution trace:
  [time   0] Run job 0 for 1.00 secs (DONE)
  [time   1] Run job 1 for 4.00 secs (DONE)
  [time   5] Run job 2 for 7.00 secs (DONE)

Final statistics:
  Job   0 -- Response: 0.00  Turnaround 1.00  Wait 0.00
  Job   1 -- Response: 1.00  Turnaround 5.00  Wait 1.00
  Job   2 -- Response: 5.00  Turnaround 12.00  Wait 5.00

  Average -- Response: 2.00  Turnaround 6.00  Wait 2.00
```

Figure 7.11: Generating Homework Solutions

As you can see from the figure, the `-c` flag shows you what happened. Job 0 ran first for 1 second, Job 1 ran second for 4, and then Job 2 ran for 7 seconds. Not too hard; it is FIFO, after all! The execution trace shows these results.

The final statistics are useful too: they compute the **response time** (the time a job spends waiting after arrival before first running), the **turnaround time** (the time it took to complete the job since first arrival), and the total **wait time** (any time spent ready but not running). The stats are shown per job and then as an average across all jobs. Of course, you should have computed these things all before running with the `-c` flag!

If you want to try the same type of problem but with different inputs, try changing the number of jobs or the random seed or both. Different random seeds basically give you a way to generate an infinite number of different problems for yourself, and the `-c` flag lets you check your own work. Keep doing this until you feel like you really understand the concepts.

One other useful flag is `-l` (that's a lower-case L), which lets you specify the exact jobs you wish to see scheduled. For example, if you want to find out how SJF would perform with three jobs of lengths 5, 10, and 15, you can run:

```
prompt> ./scheduler.py -p SJF -l 5,10,15
ARG policy SJF
ARG jlist 5,10,15

Here is the job list, with the run time of each job:
  Job 0 (length = 5.0)
  Job 1 (length = 10.0)
  Job 2 (length = 15.0)
...
```

And then you can use `-c` to solve it again. Note that when you specify the exact jobs, there is no need to specify a random seed or the number of jobs: the jobs lengths are taken from your comma-separated list.

Of course, more interesting things happen when you use SJF (shortest-job first) or even RR (round robin) schedulers. Try them and see!

And you can always run

```
./scheduler.py -h
```

to get a complete list of flags and options (including options such as setting the time quantum for the RR scheduler).

Questions

1. Compute the response time and turnaround time when running three jobs of length 200 with the SJF and FIFO schedulers.
2. Now do the same but with jobs of different lengths: 100, 200, and 300.
3. Now do the same, but also with the RR scheduler and a time-slice of 1.
4. For what types of workloads does SJF deliver the same turnaround times as FIFO?
5. For what types of workloads and quantum lengths does SJF deliver the same response times as RR?
6. What happens to response time with SJF as job lengths increase? Can you use the simulator to demonstrate the trend?
7. What happens to response time with RR as quantum lengths increase? Can you write an equation that gives the worst-case response time, given N jobs?