Overview

In this section, we'll explore **scheduling policies**

This section should help us answer the following questions:

- How should we think about scheduling policies?
- What are the key assumptions?
- What are the important measurements?

Introduction to CPU Scheduling

We have an idea of how the low-level mechanisms of running processes work, but we still don't know how an OS **scheduler** works.

Let's explore a series of **scheduling policies** (also known as disciplines) that have been developed over time.

Early techniques to scheduling were drawn from operations management and applied to computers. It's no surprise that assembly lines and other human efforts require **scheduling**, and have many of the same concerns, including a laser-like quest for efficiency.

Workload Assumptions

Let us first make some basic **assumptions** about the processes occurring in the system, frequently referred to as the **workload**. The more you know about **workload** you're dealing with, the more fine-tuned your policy can be.

We make some **unrealistic workload assumptions** here, but that's okay (for now) because we'll ease them off as we go, and eventually we'll develop what we'll call a fully operational scheduling discipline.

Our **assumptions** about the processes, sometimes called jobs, that run in the system are as follows:

1. Every job runs for the same amount of time.

2. All jobs arrive at the same time.

3. When a job is started, it runs to completion.

4. All jobs use the CPU only (i.e., they do no I/O).

5. Each job's run-time is known.

We mentioned many of these assumptions are **unrealistic**, but some are more unrealistic than others.

It may bother you that each job's run-time is known: this would make the scheduler omniscient, which would be amazing, but unlikely.

Which of the following is NOT an initial assumption we will make about the workload?

Scheduling Metrics

Aside from workload assumptions, we need a scheduling metric to compare different scheduling policies. A metric is a unit of measurement, and there are many different metrics that make sense in scheduling.

For now, let's focusing on one metric: turnaround time. The turnaround time of a job is the time it takes to complete minus the time it arrived in the system. So:

$$ Tturnaround = Tcompletion - Tarrival$$

We have assumed all jobs arrive at the same time, so $Tarrival = 0$ for now and $Tturnaround = Tcompletion$. This will be our initial assumptions.

Our focus in this section will be on **turnaround time** as a **performance metric**.

Performance and **fairness metrics** are sometimes at odds in scheduling; a scheduler may enhance performance at the expense of fairness by prohibiting some activities from running.

Complete the formula for computing Turnaround Time

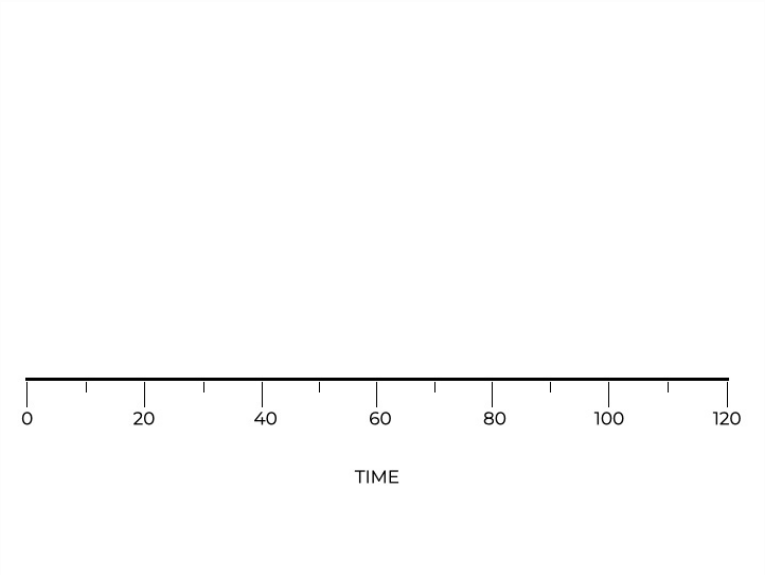
FIFO: First In, First Out

First In, First Out (FIFO) scheduling or First Come, First Served (FCFS) is our first algorithm.

For example, assume:
* Three jobs, A, B, and C, arrive in the system at the same time ($Tarrival = 0$).
* While they all arrived “at the same time”, A arrived just before B, who came just before C.
* Each job lasts 10 seconds.

What’s the average turnaround time for these jobs?

The figure below shows that:
* A finished at 10 seconds,
* B at 20, and
* C at 30.



.guides/img/fifo1

So, computing **average turnaround time** is as straightforward as:

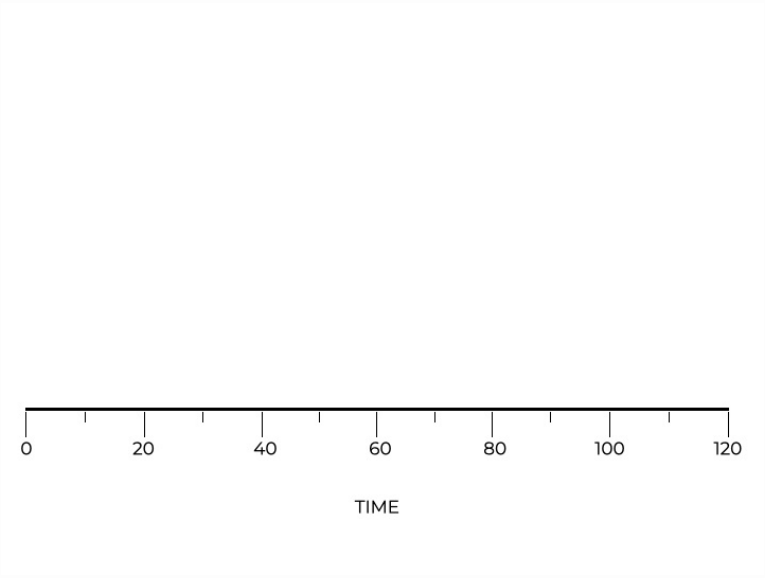
$$\frac{10+20+30}{3} = 20$$

Relaxing Assumption #1

Let's no longer assume that **all of our processes run for the same amount of time.**

How does FIFO perform now? What kind of workload could you create to weaken FIFO's performance?

Assume three jobs (A, B, and C), except this time A will run for 100 seconds, while B and C will each run for 10 seconds.



.guides/img/fifo2

Job A runs for the entire 100 seconds before B or C have a chance to run. As a result, the system's **average turnaround time** is long:

$$\frac{100+110+120}{3} = 110$$

The **convoy effect** occurs when a large number of small prospective resource consumers are waiting behind a heavyweight resource consumer.

Like having one shopping line open and the person in front of you has a huge order.

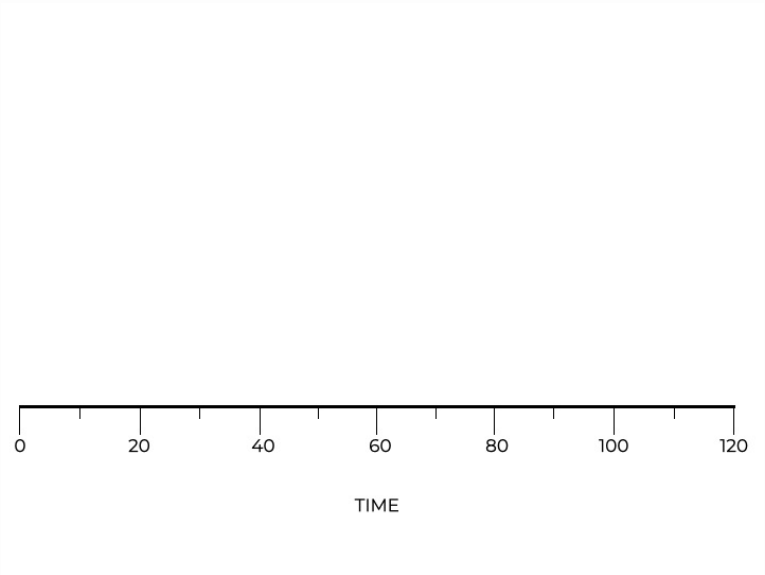
How can we improve our algorithms to deal with jobs that run for varying lengths of time?

SJF: Shortest Job First

This problem can be solved using **Shortest Job First (SJF)**. This policy runs the shortest job first, then the next shortest, and so on.

So, let's use the same example as before, but with **SJF**.

SJF's superior turnaround time is shown in the graphic below.



.guides/img/sjf1

SJF simply runs B and C before A. This reduces the average turnaround time down to 50 seconds.

$$\frac{10+20+120}{3} = 50$$

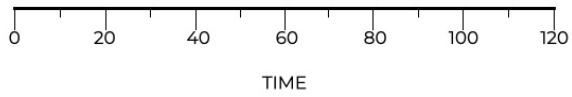
Our approach to scheduling using **SJF** is nice, however our assumptions are still unrealistic.

Relaxing Assumption #2

Let's assume now that **jobs can arrive at any time rather than all at once.**

What issues does this raise?

Let's say A comes at time $t = 0$ and must run for **100 seconds**, while B and C arrive at time $t = 10$.



.guides/img/sjflate

Even though B and C arrived shortly after A, they must wait for A to finish, causing the same **convoy issue**.

The average turnaround time for these jobs is **103.33 seconds**.

$$\frac{100+(110-10)+(120-10)}{3} = 103.33$$

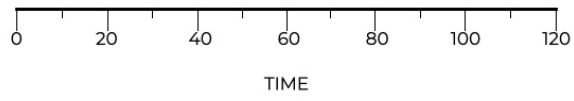
STCF: Shortest Time-to-Completion First

Relaxing Assumption #3

To address this issue, let's relax assumption 3, **that jobs have to run until completed**. Additionally, we need some machinery within the scheduler itself. The scheduler can **preempt**, or interrupt, job A and perform another job, possibly resuming A later. SJF is a non-preemptive scheduler, and so has the issues listed in the previous section.

The **Shortest Time-to-Completion First (STCF)** or Preemptive Shortest Job First (PSJF) scheduler adds **preemption** to SJF.

When a new job enters the system, the **STCF** scheduler looks at the remaining jobs and schedules the one with the least time remaining. In our case, **STCF** would preempt A and complete B and C first, then schedule A's remaining time.



.guides/img/sjcf

This improves the **average turnaround time** to **50 seconds**.

$$\frac{(120-0)+(20-10)+(30-10)}{3} = 50$$

Given that SJF is optimal if all jobs arrive at the same time, you should be able to see the intuition underlying STCF's design.

A New Metric: Response Time

If we knew all job lengths, jobs only used the CPU, and our only metric was turnaround time, STCF would be a fantastic policy.

The response time is the time from when the job arrives in the system to when it is scheduled for the first time

$ Tresponse = Tfirstrun - Tarrival $

If we have the schedule from our previous graphic with A coming at 0 and B and C at 10 our response time would be **3.33**.

Shortest Time to Completion First and related policies not great for response time. If three jobs arrive at the same time, the third has to wait for the previous two to finish before being scheduled.

This method is good for turnaround time but awful for response time and interaction.

Imagine typing at a terminal and waiting 10 seconds for a response from the system because another job was scheduled ahead of yours: not fun.

So now we have another issue: **How to build a scheduler that is responsive?**

Complete the formula for computing Response Time

Round Robin

To address this issue, we propose a new scheduling technique known as **Round-Robin (RR)** scheduling.
Instead of running jobs to completion, **RR runs them for a slice of time (also known as a scheduling quantum) before moving on to the next job in the queue**.

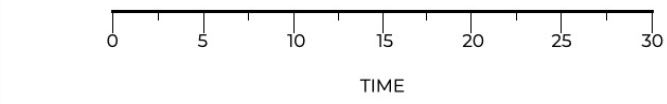
It does so until all of the jobs are done.

If the timer interrupts every 10 milliseconds, a time slice needs to be a multiple of 10 ms; for example, a time slice could be 10, 20, or any other multiple of 10 ms.

Three jobs A, B, and C arrive in the system at the same time. Each of them wants to run for 5 seconds.

An SJF scheduler runs each job all the way through before running another, having an average response time of:

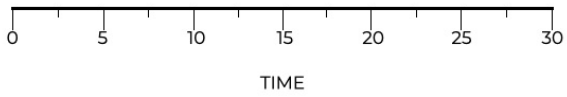
$$\frac{(0)+(5)+(10)}{3} = 5$$



.guides/img/rr1

RR with a time-slice of 1 second would, instead, cycle through the jobs quickly. This would have an average response time of:

$$\frac{(0)+(1)+(2)}{3} = 1$$



.guides/img/rr2

For RR, the slice length is crucial. A shorter response time leads to better performance of RR. The problem with cutting the time slice too short is that context switching costs soon dominate performance.

A system designer must decide on the length of the time slice that will reduce the switching cost while not making it so long that it becomes unresponsive.

In other words, saving and restoring a few registers does not fully cover the cost of context switching.

Programs generate a lot of state in CPU caches, TLBs, branch predictors, and other hardware.

When switching jobs, this data is flushed and fresh state relevant to the current job is brought in, causing a performance hit.

Trade-Offs

If response time is our only metric, RR is a great scheduler, but what about our old friend turnaround time?

Let’s re-examine our previous example.

A, B, and C arrive at the same time, and RR is the scheduler with a 1-second time slice.

As shown in the graphic above, A finishes at 13, B 14, and C 15, for an average of 14.
Horrible!

RR is one of the poorest policies in terms of **turnaround time**. This makes sense because RR is dragging out each job as long as it can by just running it briefly before moving on. Because turnaround time solely worries about project completion, RR is often worse than FIFO.

Generally, a fair policy (like RR) will do poorly on metrics like turnaround time. This is the trade-off we have to consider:

- If you're willing to be unfair, you can complete a job quicker, but at the expense of response time
- If you value fairness, however, response time is lowered, but turnaround time is increased.

There are always **trade-offs** in systems; you, unfortunately, can't have everything.

Incorporating I/O

Relaxing Assumption #4

Let's address the assumption that the **programs don't request any I/O**. All programs perform I/O!

A scheduler has to make a decision when a job requests I/O since the currently running job can't access the CPU while the I/O is being performed.

If the I/O is sent to a hard disk drive, the process could be blocked for a few milliseconds or more depending on the current I/O load.

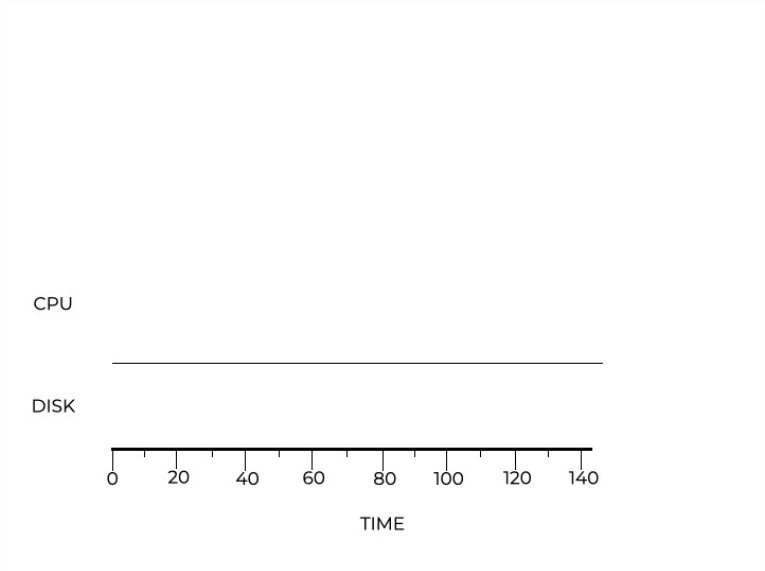
So, scheduling another CPU job at that time might be wise. The scheduler also has to decide when the I/O is complete.

When this happens, an interrupt is raised, and the OS executes, putting the process that issued the I/O back into the ready state.

How should the OS handle each job?

Consider two jobs, A and B, each requiring 50 milliseconds. There is one major distinction between A and B:

A runs for 10 ms before making an I/O request (let us assume here that I/O takes 10 milliseconds each), while B uses the CPU for 50 ms without I/O. The scheduler starts with A and ends with B as in the graphic below.



.guides/img/io1

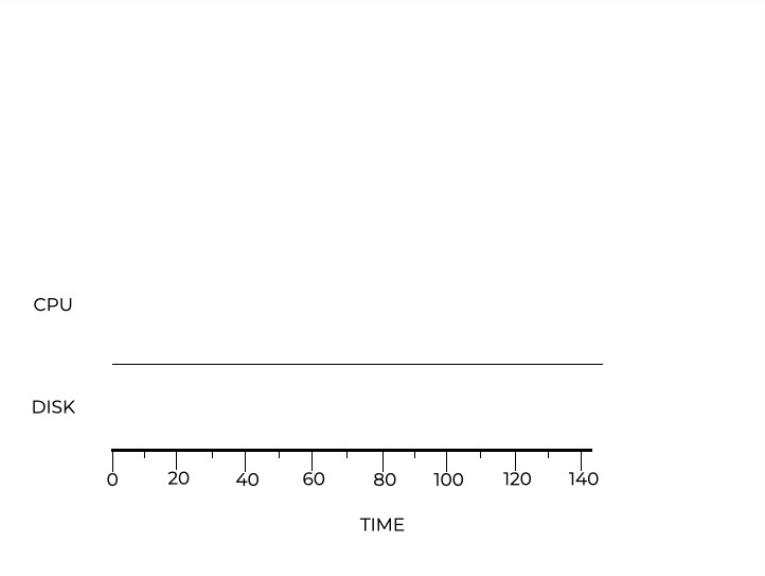
Assume we're writing an STCF scheduler. In that situation, how can this scheduler account for A having five tasks each lasting 10ms while B just has one task lasting 50ms?

Running jobs one after the other without regard for I/O makes little sense.

Each 10-ms subjob of A is usually handled as a standalone job. The system can choose between a 10-ms A or a 50-ms B.

The system can choose between a 10-ms A or a 50-ms B. If STCF is used, the shortest one should be chosen, in this case A. After A's first subjob, B begins.

In this case, A is a sub-job that runs for 10 ms and preempts B. This allows one process to use the CPU while another is delayed by I/O, making the system more efficient.



.guides/img/io2

This example shows how I/O can be scheduled. By treating each CPU burst as a separate job, it enables the scheduler to run interactive processes frequently. Other CPU-intensive jobs can run while interactive jobs are doing I/O, better leveraging the processor.

The End of the Oracle

Relaxing Assumption #5

Now to relax our last assumption that **the scheduler knows the length of every job**.

That may be the worst assumption we could make. In truth, a general-purpose OS knows very little about job length.

- So, how can we construct an **SJF/STCF** strategy without such prior knowledge?
- How can we use the **RR** scheduler's techniques to improve response time?

We will solve this issue by creating a scheduler that predicts the future based on the recent past. The **multi-level feedback queue** scheduler is the subject of the next section.

Summary

We've covered the basics of scheduling and explores two approaches.

- The first runs the shortest job remaining, minimizing turnaround time.
- The second alternates between all jobs, minimizing response time.

Both are bad where the other is good, a systemic trade-off. We also investigated incorporating I/O into our jobs.

Lab Intro

For this lab, will be running a simulator, `scheduler.py`, that lets us see how different schedulers perform under scheduling metrics like:
* Response time,
* Turnaround time, and
* Total wait time.

There are two steps are running this simulator:

1. Run the program without the `-c` flag.

This shows you what problem to solve without showing the answer.

- For example, run this in the terminal to compute **response**, **turnaround**, and **wait times** for three jobs using FIFO:

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

Here, we've defined:
* the `FIFO` policy
* `3` jobs, and
* a random seed of `100`.

To see the solution to this exact problem, you have to use the same random seed.

You should see some thing similar to the output below:

```
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 = 2 ) Job 1 ( length = 5 ) Job 2 ( length = 8 )
```

Three jobs were generated:
* `Job 0` of length `2`,
* `Job 1` of length `5`, and
* `Job 2` of length `8`.

You can use this to compute the response time, turnaround time, and wait time for each job and see if you have a

2. Run the program again with the same arguments but include `-c`

This gives you the solutions to the calculations.

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

Your output should look similar to the following:

```
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 = 2 ) Job 1 ( length = 5 ) Job 2 ( length = 8 ) **
```

The **execution trace** shows us that:
1. `Job 0` ran for `2` second
2. `Job 1` ran for `5` seconds, then
3. `Job 2` ran for `8` seconds.

Finally, the **statistics** compute:
* **response time** - (time a job spends waiting after arrival before running)
* **turnaround time** - (time it takes to complete the job from first arrival)

* **total wait time** - (any time spent ready but not running). The stats are broken down by job, then by average.

You should calculate these yourself before running with the `-c` flag!

Change the number of jobs, the random seed, or both to test the problem with different inputs. The `-c` flag allows you to check your own work. Repeat until you feel confident in your understanding of the principles.

The `-l` and `-h` Flag

Another handy flag is `-l` (lowercase L), which allows you to define which jobs to schedule. For example, to see how SJF performs with three

```
./scheduler.py -p SJF -l 5,10,15
```

You can use the `-h` flag to get a full list of flags and options for this simulator.

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

Lab

Use the simulator to answer the following questions. Enter responses rounded to the nearest hundredth.

Compute the average response time and turnaround time to the nearest hundredth when running 3 jobs of length 200, with the SJF and FIFO Schedulers.

Now, compute the same metrics for the same schedulers, but with different lengths of 100, 200, and 300 for the jobs.

Compute the same metrics with the RR scheduler and a time slice of 1.

For what kind of workloads does SJF deliver the same turn around times as FIFO?