Join on Zoom if you are remote (see website for link).

377 Operating Systems

Introduction to CPU Scheduling

Scheduling: Introduction

By now low-level mechanisms of running processes should be clear

If not, go back a chapter or two and read it again, review course notes, etc.

Scheduling: Introduction

By now low-level mechanisms of running processes should be clear

If not, go back a chapter or two and read it again, review course notes, etc.

However, we have yet to understand the high-level policies that an OS scheduler employs.

Scheduling: Introduction

By now low-level mechanisms of running processes should be clear

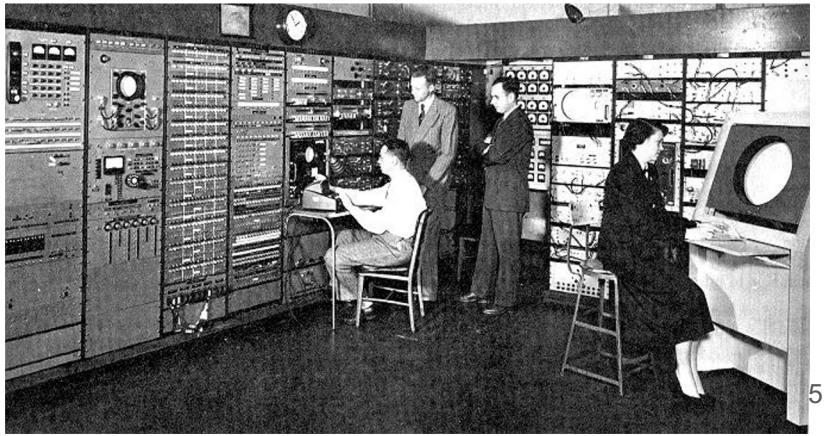
If not, go back a chapter or two and read it again, review course notes, etc.

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 that various smart and hard-working people have developed over the years.

Scheduling: Origins

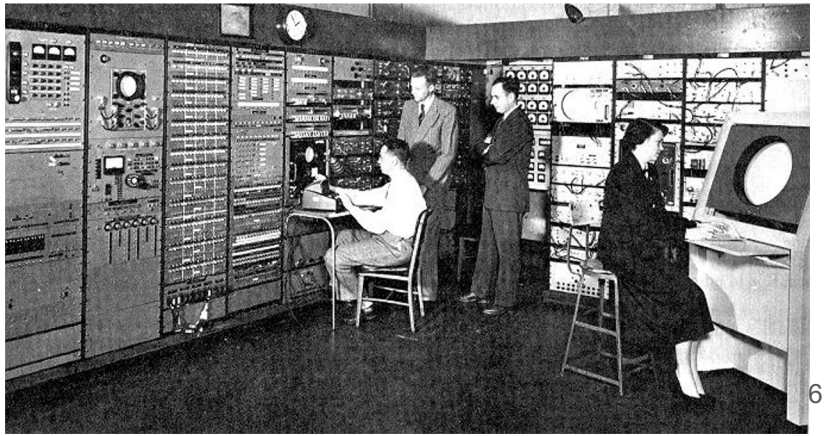
The origins of scheduling, in fact, predate computer systems; early approaches were taken from the field of operations management and applied to computers.



Scheduling: Origins

The origins of scheduling, in fact, predate computer systems; early approaches were taken from the field of operations management and applied to computers.

This reality should be no surprise:
assembly lines and many other human endeavors also require scheduling, and many of the same concerns exist therein, including a laser-like desire for efficiency.



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?

Workload Assumptions

First, we want to make some simplifying assumptions about running processes

This is typically referred to as the workload.

Workload Assumptions

First, we want to make some simplifying assumptions about running processes

This is typically referred to as the workload.

Simplified assumptions about running processes (often called jobs):

Workload Assumptions

First, we want to make some simplifying assumptions about running processes

This is typically referred to as the workload.

Simplified assumptions about running processes (often called jobs):

1. Each job runs for the same amount of time

Workload Assumptions

First, we want to make some simplifying assumptions about running processes

This is typically referred to as the workload.

Simplified assumptions about running processes (often called jobs):

1. Each job runs for the same amount of time
2. All jobs arrive at the same time

Workload Assumptions

First, we want to make some simplifying assumptions about running processes

This is typically referred to as the workload.

Simplified assumptions about running processes (often called jobs):

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. Once started, each job runs to completion

Workload Assumptions

First, we want to make some simplifying assumptions about running processes

This is typically referred to as the workload.

Simplified assumptions about running processes (often called jobs):

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. Once started, each job runs to completion
4. All jobs only use the CPU (i.e., no I/O)

Workload Assumptions

First, we want to make some simplifying assumptions about running processes

This is typically referred to as the workload.

Simplified assumptions about running processes (often called jobs):

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

Workload Assumptions

First, we want to make some simplifying assumptions about running processes

This is typically referred to as the workload.

Simplified assumptions about running processes (often called jobs):

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

Note, this is unrealistic and we will relax these as we go along.

Scheduling Metrics

Beyond making workload assumptions, we also need one more thing to enable us to compare different scheduling policies: a scheduling metric.

For now, however, let us also simplify our life by having a single metric:

turnaround time

Scheduling Metrics

Beyond making workload assumptions, we also need one more thing to enable us to compare different scheduling policies: a scheduling metric.

For now, however, let us also simplify our life by 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.

$$T_{turnaround} = T_{completion} - T_{arrival}$$

Scheduling Metrics

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.

Scheduling Metrics

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 at the moment.

Scheduling Metrics

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 at the moment.

Another metric of interest is fairness.

Scheduling Metrics

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 at the moment.

Another metric of interest is fairness.

Performance and fairness are often at odds in scheduling

Scheduling Metrics

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 at the moment.

Another metric of interest is fairness.

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.

First In, First Out (FIFO)

First In, First Out (FIFO)

The most basic algorithm we 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 simple and thus easy to implement. And, given our assumptions, it works pretty well.

Example 1: First In, First Out (FIFO)

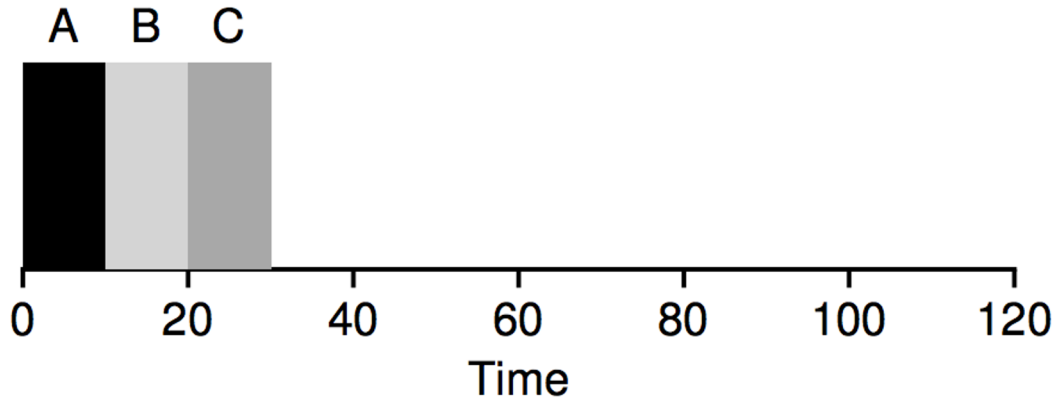
Imagine three jobs arrive in the system, A, B, and C, at roughly the same time ($T_{arrival} = 0$).

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?

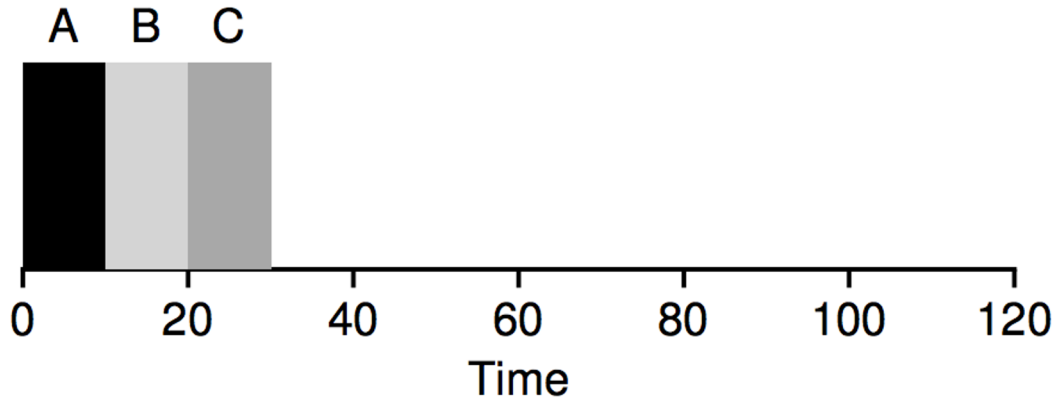
Example 1: First In, First Out (FIFO)



As you can see, A finished at 10, B at 20, and C at 30.

Thus, the average turnaround time is ???

Example 1: First In, First Out (FIFO)



As you can see, A finished at 10, B at 20, and C at 30.

Thus, the average turnaround time is (10+20+30) / 3 = 20.

Workload Assumptions

Simplified assumptions about running processes (often called jobs):

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

What if we relax one of our assumptions?

In particular, **let's relax assumption 1**

no longer assume that each job runs for the same amount of time.

Example 1: First In, First Out (FIFO)

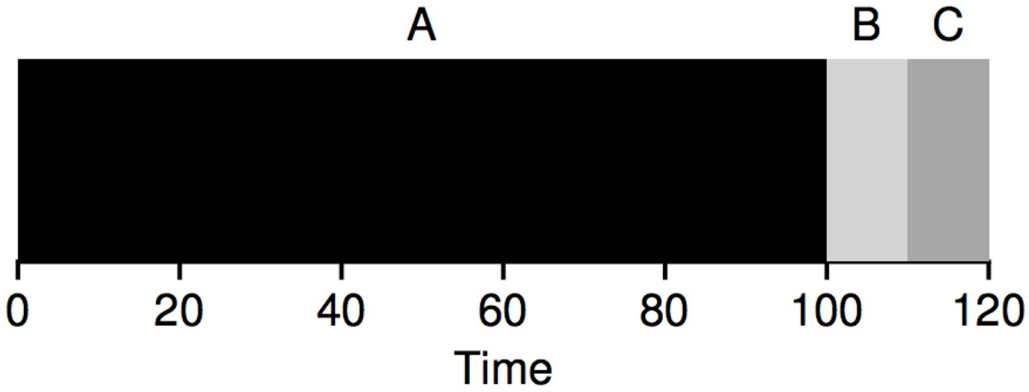
Now, we **no longer assume that each job runs for the same amount of time**.

How does FIFO perform now?

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.

(arriving in that order, one slightly before the next)

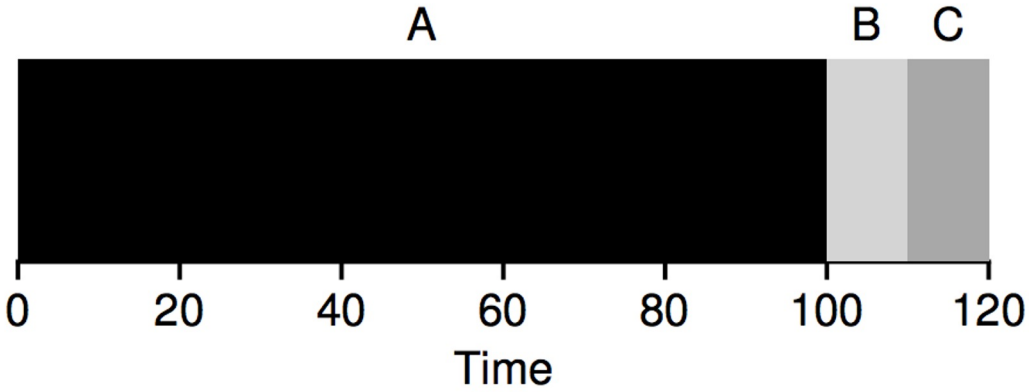
Question?



As you can see, A finished at 100, B at 110, and C at 120.

Thus, the average turnaround time is ???

Answer!



As you can see, A finished at 100, B at 110, and C at 120.

Thus, the average turnaround time is (100+110+120) / 3 = 110.

Convoy Effect

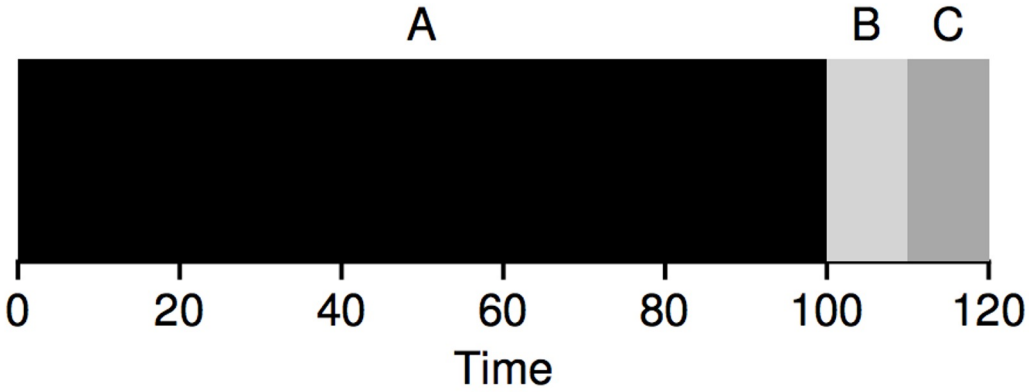
This problem is generally referred to as the **convoy effect**:
A number of relatively-short potential consumers of a resource get queued behind a heavyweight resource consumer.

This scheduling scenario might remind you of a single line at a grocery store and what you feel like when you see the person in front of you with three carts full of provisions and a checkbook out; it's going to be a while.

What can we easily do to fix this?

Shortest Job First

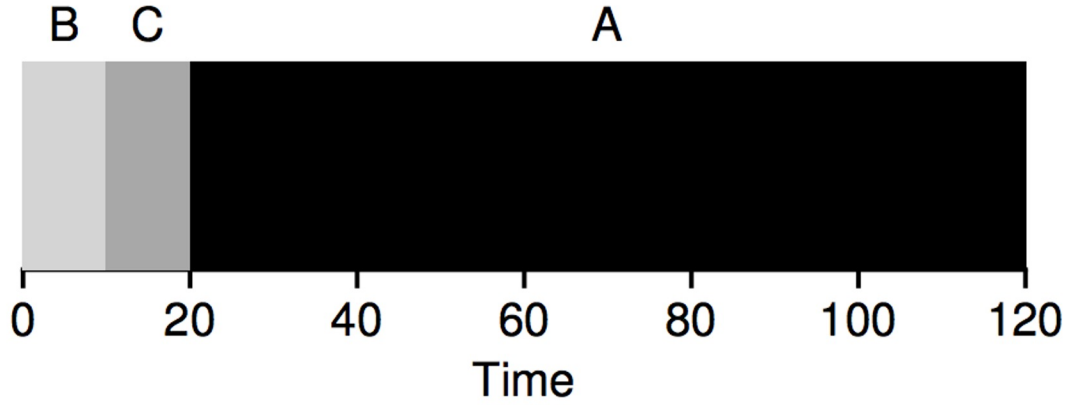
Example 1: Shortest Job First (SJF)



What if we applied SJF to our FIFO example from before?

What is the average turnaround time?

Example 1: Shortest Job First (SJF)

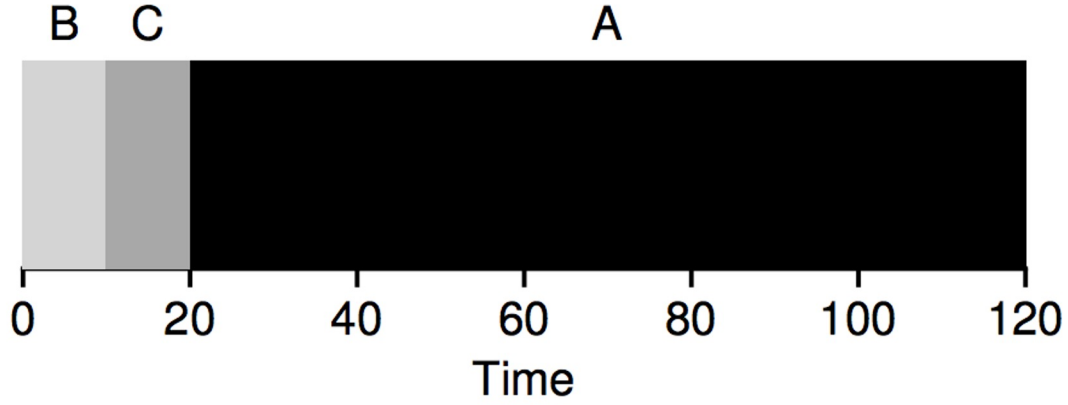


What if we applied SJF to our FIFO example from before?

What is the average turnaround time?

Thus, the average turnaround time is ???

Example 1: Shortest Job First (SJF)



What if we applied SJF to our FIFO example from before?

What is the average turnaround time?

Thus, the <u>average turnaround</u> time is (10+20+120) / 3 = 50.

Workload Assumptions

Simplified assumptions about running processes (often called jobs):

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

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.

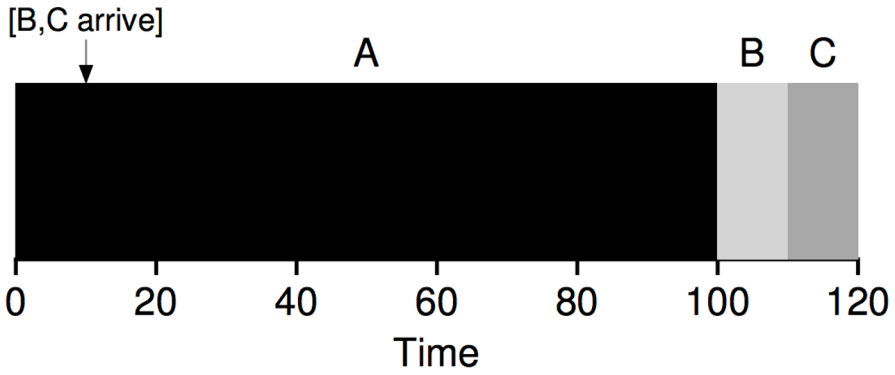
Shortest Job First (SJF): Assumptions

So, now assume that jobs can arrive at any time instead of all at once.

What problems does this lead to?

Shortest Job First Problem

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.



What can we easily do to fix this?

Shortest Time to Completion First

Workload Assumptions

Simplified assumptions about running processes (often called jobs):

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

To address this concern, **we need to relax assumption 3** (that jobs must run to completion).

We also need some machinery within the scheduler itself.

Shortest Time to Completion First (STCF)

As you might have guessed, given our previous discussion about 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, maybe continuing A later.

SJF by our definition is a non-preemptive scheduler, and thus suffers from the problems we just witnessed.

STCF Example #1

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.

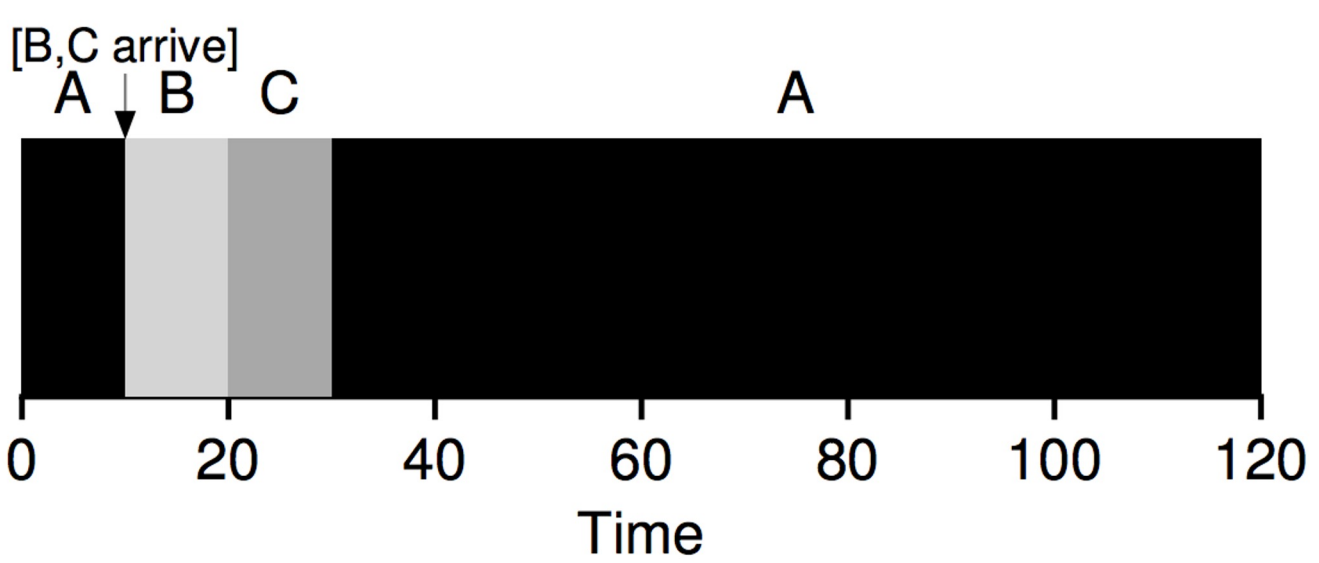
Any time a new job enters the system:

1. STCF scheduler determines which of the remaining jobs (including the new job) has the least time left

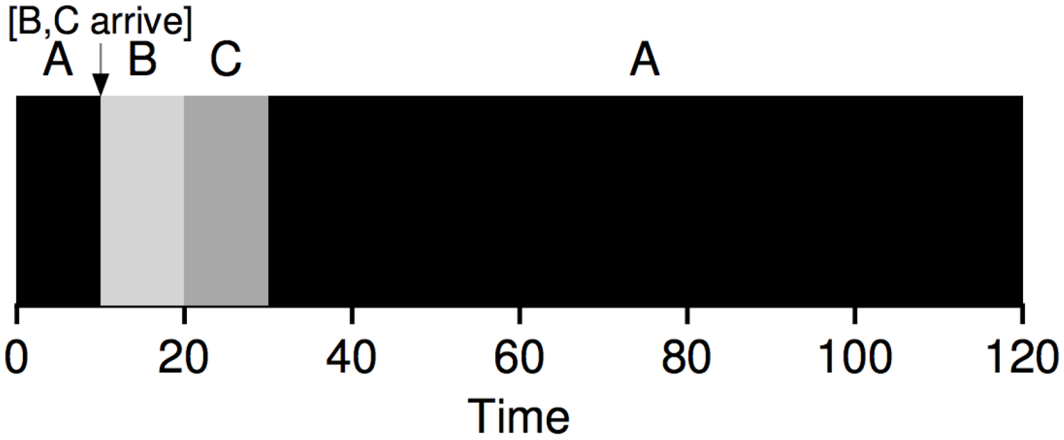
2. STCF schedules the shortest job first.

STCF Example #1

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.



STCF Example #1



$$\left(\frac{(120-0)+(20-10)+(30-10)}{3}\right)$$

The result is a much-improved average turnaround time: **50 seconds**

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.

A New Metric: Response Time

Thus, if we knew job lengths, and that jobs only used the CPU, and our only metric was turnaround time, STCF would be a great policy.

A New Metric: Response Time

Thus, if we knew job lengths, and that 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.

A New Metric: Response Time

Thus, if we knew job lengths, and that 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.

A New Metric: Response Time

Thus, if we knew job lengths, and that 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 Definition

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

Response Time Definition

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

$$T_{response} = T_{firstrun} - T_{arrival}$$

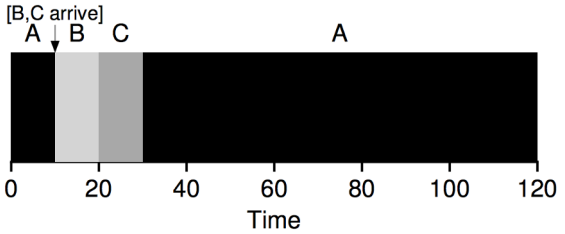
Response Time Definition

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

$$T_{response} = T_{firstrun} - T_{arrival}$$

For example, if we had the previous schedule (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).

((0 - 0) + (10 - 10) + (20 - 10)) / 3 = 3.33



STCF Response Time: Not so good :-(

As you might be thinking, STCF and related disciplines are not particularly good for response time.

STCF Response Time: Not so good :-(

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.

STCF Response Time: Not so good :-(

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…

STCF Response Time: Not so good :-(

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…

Thus, we are left with another problem:

how can we build a scheduler that is sensitive to response time?

What can we easily do to fix this?

Round Robin

Round Robin

The basic idea is simple:

Instead of running jobs to completion,

RR runs a job for a time slice (sometimes called a scheduling quantum),

then switches to the next job in the run queue.

Round Robin

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.

Round Robin

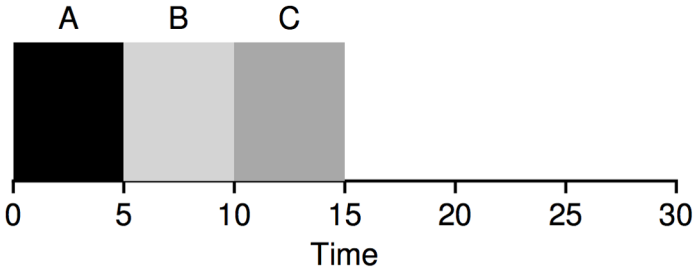


Figure 7.6: **SJF Again (Bad for Response Time)**

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.

Round Robin

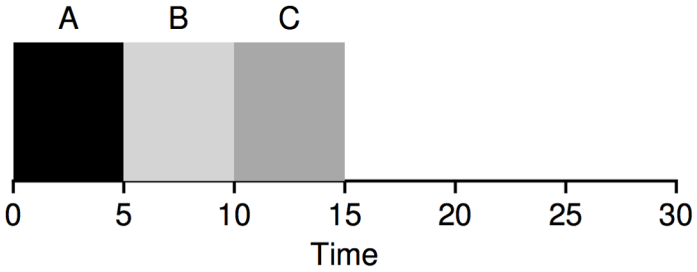


Figure 7.6: **SJF Again (Bad for Response Time)**

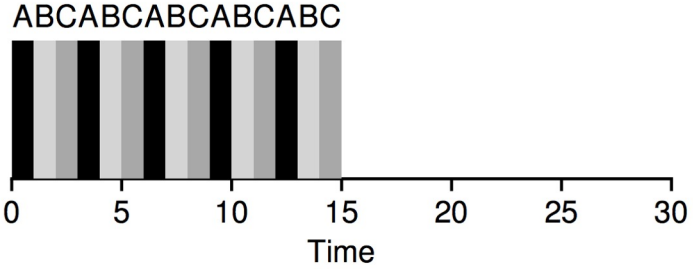


Figure 7.7: **Round Robin (Good for Response Time)**

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.

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

Round Robin

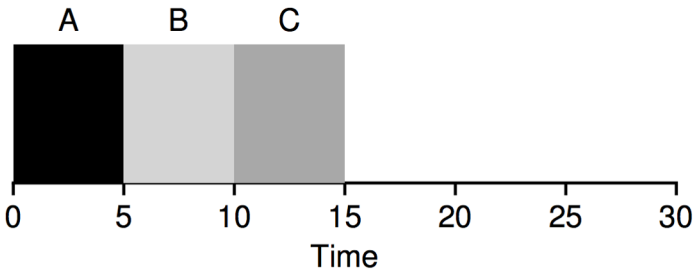


Figure 7.6: **SJF Again (Bad for Response Time)**

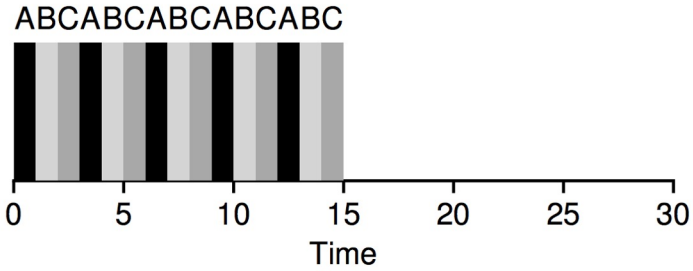


Figure 7.7: **Round Robin (Good for Response Time)**

Average response time for RR: $\frac{0+1+2}{3} = 1$

Average response time for SJF: $\frac{0+5+10}{3} = 5$

RR: Is the grass always greener on the other side?

RR does well for response time.

The shorter the time slice, the better it is for RR.

RR: Is the grass always greener on the other side?

RR does well for response time.

The shorter the time slice, the better it is for RR.

But, what about the overhead of a context switch?

RR: Is the grass always greener on the other side?

RR does well for response time.

The shorter the time slice, the better it is for RR.

But, what about the overhead of a context switch?

A context switch is expensive, so we need to be careful here.
In other words, shorter time slice means *context switch dominates performance*.

And, what about turnaround time?

Round Robin: Turnaround

$$T_{turnaround} = T_{completion} - T_{arrival}$$

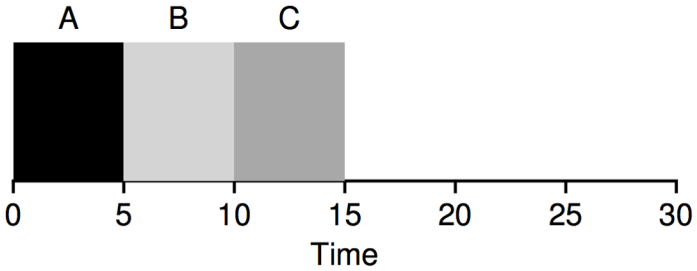


Figure 7.6: **SJF Again (Bad for Response Time)**

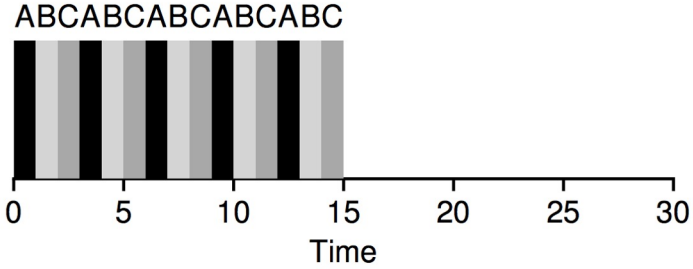


Figure 7.7: **Round Robin (Good for Response Time)**

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! (SJF is 10)

Punch Line

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.

Punch Line

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:

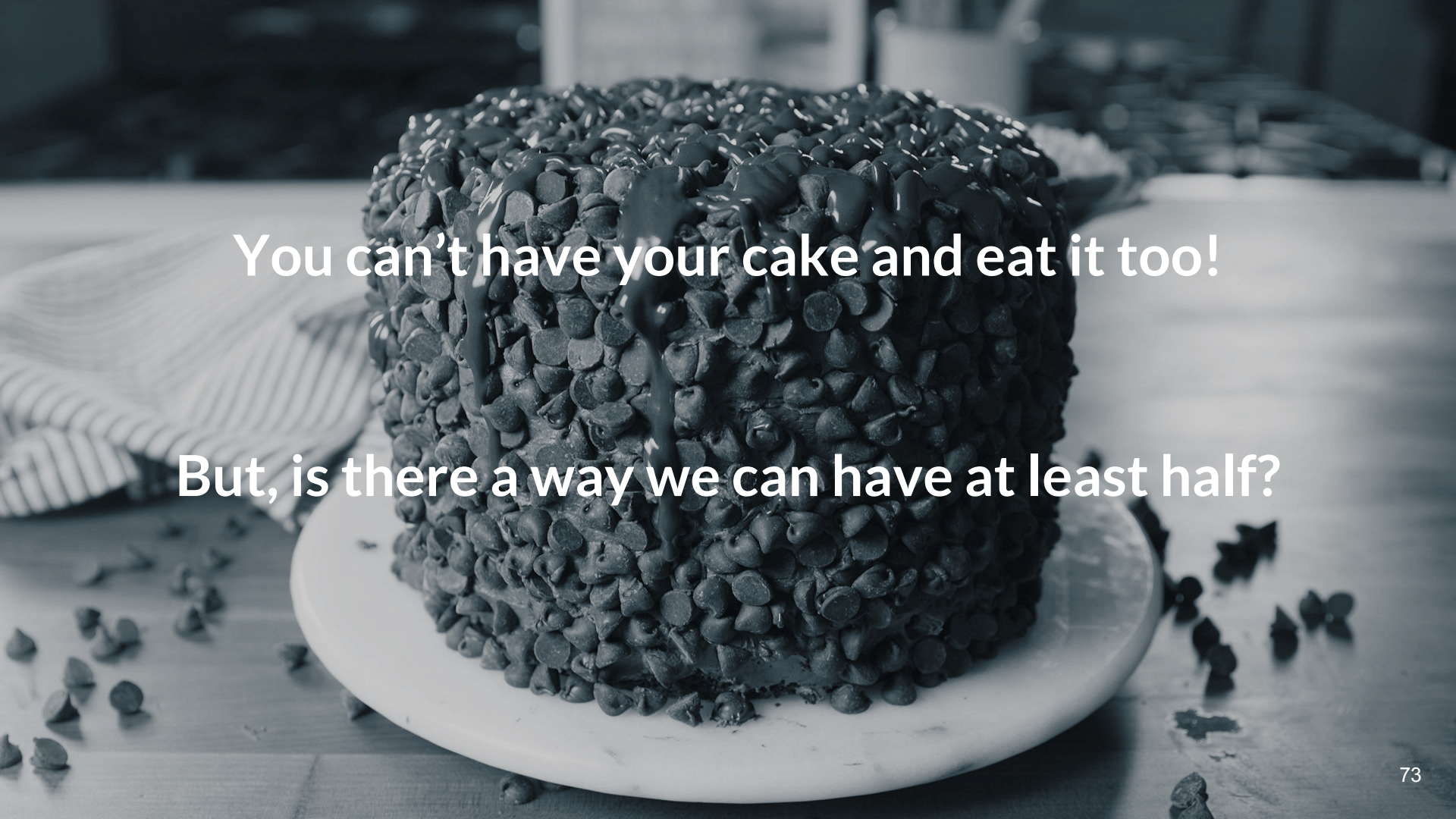
Punch Line

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!


You can’t have your cake and eat it too!
But, is there a way we can have at least half?


The End