# Chapter 7 Scheduling: Introduction

Separation between mechanism and policy

Now we talk about policy (higher-level)

Q1: How should we develop a basic framework for thinking about scheduling policies?

Q2: What are the key assumptions?

Q3: What metrics are important?

Q4: What basic approaches that have been used in other system?

**7.1 Workload Assumptions**

Simplifying assumptions —> collectively called workload

Knowledge about workload —> more fine-tune policy.

Workload assumptions are not realistic —> they are just assumptions

**Unrealistic assumptions** about the processes (sometimes 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., they perform no I/O)
5. The run-time of each job is known. (Would make the scheduler omniscient — not going to happen any time soon)

## 7.2 Scheduling Metrics

To compare different scheduling policies: a scheduling metric

For now we simplify and use only a single metric.

**Def. Turnaround time**: $T_{turnaround} = T_{completion} - T_{arrival}$

For now, $T_{arrival} = 0$.

Hence, $T_{turnaround} = T_{completion}$

Turnaround time is a **performance** metric.

However, we also care about **fairness.**

Often times: performance and fairness are at odds with each other.

If we optimize performance by preventing some processes from running, we sacrifice fairness.

Conundrum: Sometime we must make **tradeoffs.**

## 7.3 First In, First Out (FIFO)

Most basic algorithm: or First Come, First Serve (FCFS)

Ex. Three process A, B, C arrive roughly at the same time ($T_{arrival} = 0$).

Since FIFO has to give an order to the arrival times
Assume A arrive just before B, and B just before C.

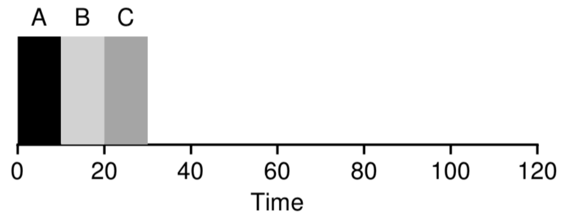


Figure 7.1: **FIFO Simple Example**

Gantt diagram

$T_{turnaround}$ (A) = A finish @ 10

$T_{turnaround}$ (B) = B finish @ 20

$T_{turnaround}$ (C) = C finish @ 30

Avg $T_{turnaround}$ = (10+20+30)/3 = 20 seconds

*Relax assumption 1: jobs may run for different amount of time.*

Q: Can you construct a workload for which FIFO performs poorly?

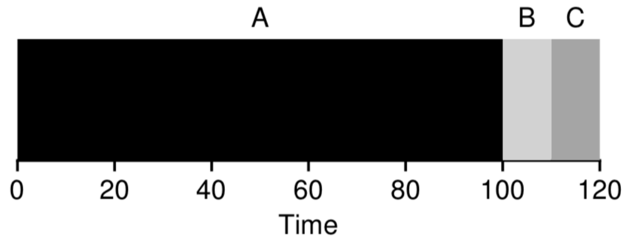


Figure 7.2: **Why FIFO Is Not That Great**

Avg $T_{turnaround}$ = (100+110+120)/3 = 110 seconds

Painfully slow for B and C.

This is called the **Convoy effect**:

A number of relatively short potential consumers of the resource get queued behind a heavyweight resource consumer.

What should we do?

Example from real-world: Grocery stores commonly have a “ten-items-or-less” lines to ensure that shoppers with only a few things don’t get stuck behind the family preparing for some upcoming nuclear winter.

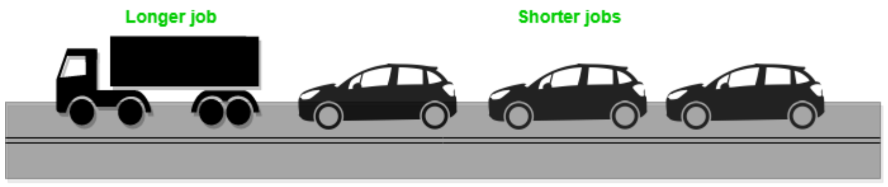


**Figure** - The Convey Effect, Visualized

## 7.4 Shortest Job First (SJF)

Very simple: Runs shortest jobs first, then the next shortest job and so on…

See Fig. 7.3. Each to see that this leads to much better performance

A needs 100 seconds of CPU

B needs 10 seconds of CPU

C needs 10 seconds of CPU

$T_{turnaround}$ (A) = 120

$T_{turnaround}$ (B) = 10

$T_{turnaround}$ (C) = 20

Avg $T_{turnaround}$ = (10+20+120)/3 = 50 seconds

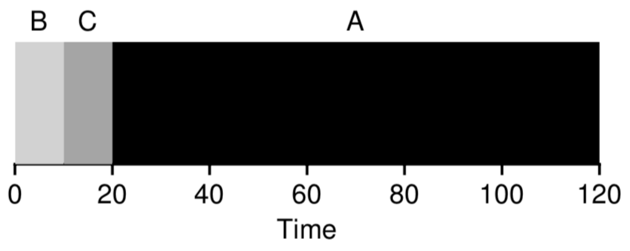


Figure 7.3: **SJF Simple Example**

*Relax assumption 2: Now assume jobs can arrive any time…*

Q: What can happen now? (See Fig 7.4)

$T_{arrival}$ (A) = 0

$T_{arrival}$ (B) = 10

$T_{arrival}$ (C) = 10

A starts to run, and even though B and C arrive early on they both have to wait until A has finished.

Avg $T_{turnaround}$ = (100+100+110)/3 = 103.333 seconds

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

$T_{turnaround}$ (A) = 100 - 0 = 100

$T_{turnaround}$ (B) = 110 - 10 = 100

$T_{turnaround}$ (C) = 120 - 10 = 110

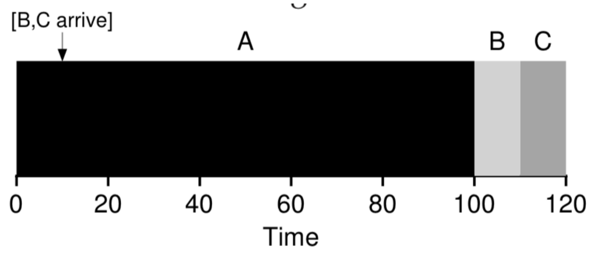


Figure 7.4: **SJF With Late Arrivals From B and C**

## 7.5 Shortest Time-Completion First (STCF)

*Relax assumption 3: Now assume that jobs can be interrupted by the OS…*
*(they don't have to run to completion…)*

**Aside: Preemptive schedulers**

Old days: Batch of jobs run to completion. Used **non-preemptive** schedulers.

Today, virtually all schedulers are **preemptive**.

- OS stops one process and runs another.
- Requires timer interrupts and context switch mechanism (Ch 6)

*A preemptive scheduler can preempt job A and decide to run another job B or C, and continue job A later.*

STCF or Preemptive Shortest Job First (PSFJ)

**STCF Scheduler***: When a new job arrive, find the job that has the least time left, and schedules that job.*

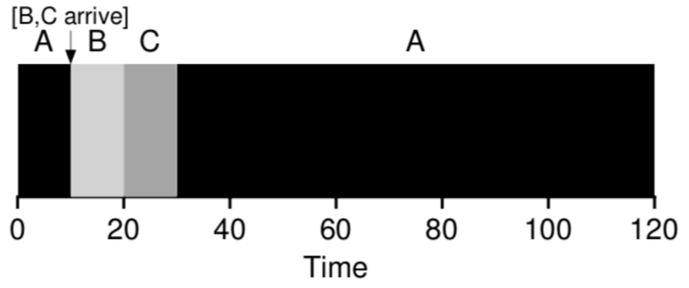


Figure 7.5: **STCF Simple Example**

A needs 100 seconds of CPU

B needs 10 seconds of CPU

C needs 10 seconds of CPU

Avg $T_{turnaround}$ = (120+10+20) / 3 = 50 seconds

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

$T_{turnaround}$ (A) = 120 - 0 = 120

$T_{turnaround}$ (B) = 20 - 10 = 10

$T_{turnaround}$ (C) = 30 - 10 = 20

## 7.5 A New Metric: Response Time

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.

However, turnaround time is not a good metric for users interacting with a system. They want:

**Def. Response Time**: $T_{response} = T_{firstrun} - T_{arrival}$

*The time from when the job first arrives in the system until the first time it is scheduled to run.*

Example:

$T_{arrival}$ (A) = 0

$T_{arrival}$ (B) = 10

$T_{arrival}$ (C) = 10

Response time for each job will be as follows:

$T_{response}$ (A) = 0 - 0 = 0

$T_{response}$ (B) = 10 - 10 = 0

$T_{response}$ (C) = 20 - 10 = 10

Average $T_{response}$ = (0+0+10) / 3 = 10/3 = 3.33 seconds

So with STCF the response time is pretty bad. If you are typing on the keyboard, in process C, you have to wait for 10 seconds for your response (to see the characters appear on the screen).

Q: How can we build a scheduler that is sensitive to response time??

## 7.7 Round Robin (time-slicing)

RR runs a job for a time slice (also called: scheduling/time quantum)
and then switches to the next job in the run queue.

Quantum must be a multiple of the timer-interrupt, e.g., if the timer interrupts
every 10 ms, then the time slice must be either 10, 20, or any other multiple of 10 ms.

Example: A = B = C = 5 seconds

$T_{arrival}$ (A) = 0

$T_{arrival}$ (B) = 0

$T_{arrival}$ (C) = 0

Assume a time slice of $t_s$ = 1 second.

Average response time for SJF (Fig 7.6): $T_{response}$ = (0+5+10) / 3 = 5 seconds

Average response time for RR (Fig 7.7) : $T_{response}$ = (0+1+2) / 3 = 1 seconds

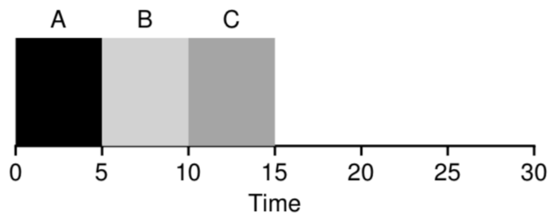


Figure 7.6: **SJF Again (Bad for Response Time)**

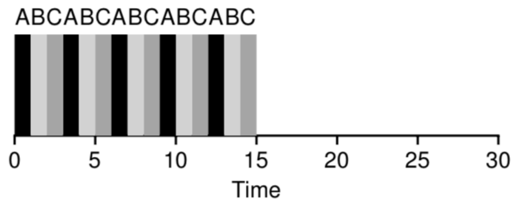


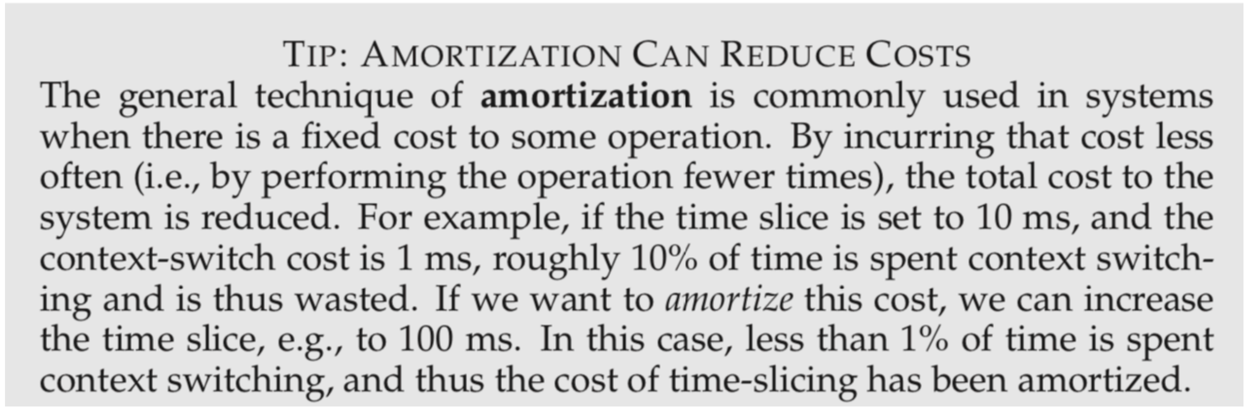
Figure 7.7: **Round Robin (Good For Response Time)**

*The length of the time slice is critical for RR.*

Shorter: better the performance of RR under the response-time metric.

Too short (problematic): the cost of context switching will dominate overall performance.

**Tradeoff for system designer**



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.

Long enough to **amortize** the cost of switching, but

Short enough so that the system is still responsive…

**A note on context switching delay:**

OS actions are not only

- save/restore CPU registers

As a process runs, build up cache state on the CPU

- Caches and TLBs, branch predictors and other on-chip hardware

These caches must be flushed when bringing a new process in.
Noticeable performance cost.

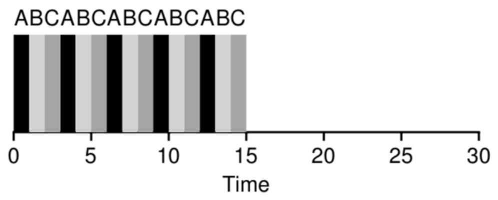


Figure 7.7: **Round Robin (Good For Response Time)**

RR is great for response time, but what about turnaround time?

Example: time slice of $t_s$ = 1 second. Fig. 7.7.

A = B = C need 5 seconds

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

$T_{arrival}$ (A) = 0

$T_{arrival}$ (B) = 0

$T_{arrival}$ (C) = 0

A finish @ 13 sec

B finish @ 14 sec

C finish @ 15 sec

Avg turnaround time = (13+14+15) / 3 = 42/3 = 14 seconds

RR is nearly pessimal (close to the worst we can do) w.r.t. turnaround time.
We are stretching the jobs out for a long time.

Two types of schedulers:

1. SJF/STCF: optimizes turnaround time — at the cost of response time.
2. RR: optimizes response time — at the cost of turnaround time.

RR is a fair scheduling policy, while SJF is performant.

## 7.8 Incorporating I/O

*Relax assumption 4: Jobs can also use I/O…*

When a job requests an IO operation — scheduler knows the job will not use the CPU during the IO.

It is **blocked** waiting for IO completion.
(hard disk drives may block for a few ms)

So scheduler should find another job to run on the CPU.

Example (Fig 7.8 and Fig 7.9):

A = B = 50 ms CPU time

A runs 10 ms on the CPU, then wait for IO that takes 10 ms and so on…

Fig.7.8 where job A is not preempted from CPU.
Hence, we can think of this as five 10 ms sub-jobs of A and one 50 ms job, B.

With STCF:
Run shorter jobs first.
Run first sub-job of A until completion (until it requests IO)
Then run B for 10 ms, until preempted by A's IO request being completed.
And so on…

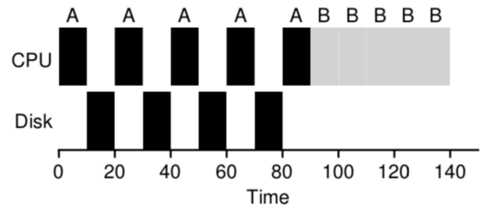


Figure 7.8: **Poor Use Of Resources**

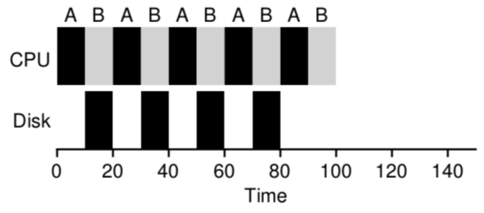


Figure 7.9: **Overlap Allows Better Use Of Resources**

## 7.9 No More Oracle

*Relax assumption 5: Scheduler does not know the length of each job…*

OS usually knows very little about the length of each job.

Cannot predict the future!

## 7.10 Summary

Two approaches

- One optimize turnaround time (SJF/STCF)
- Other optimize response time (RR)
- Both are bad where the other is good
  - Inherent tradeoff common in systems
- OS: Can't see into the future (to determine the length of jobs)
- Next chapter: Multi-level Feedback Queue
  - Use recent past to predict the future…