Chapter 7: Scheduling: Introduction

This chapter introduces the fundamental problem of CPU scheduling: deciding which of the many ready processes should be run on the CPU. We will explore a series of scheduling policies (or disciplines), starting with simple assumptions and gradually making them more realistic to build towards a comprehensive solution.

> **The Crux of the Problem: 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?

1. Foundational Concepts

1.1. Workload Assumptions

To simplify our initial analysis, we start with a highly unrealistic set of assumptions about the processes (or **jobs**) in the system. We will relax these assumptions one by one.

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 (non-preemptive).
4. All jobs only use the CPU (no I/O).
5. The run-time of each job is known in advance.

1.2. Scheduling Metrics

To compare different scheduling policies, we need metrics to measure their performance.

- **Turnaround Time:** The primary performance metric for now. It measures the total time a job spends in the system.
  - **Formula:** $T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}}$
  - Since we assume all jobs arrive at time 0 initially, $T_{\text{turnaround}} = T_{\text{completion}}$.
- **Fairness:** Another important metric, often at odds with performance. A scheduler might optimize performance by starving some jobs, which is unfair.

---

2. Basic Scheduling Algorithms

2.1. First In, First Out (FIFO)

Also known as **First Come, First Served (FCFS)**, this is the simplest scheduling algorithm.

- **Policy:** Jobs are processed in the order they arrive.
- **Example:** Three jobs (A, B, C), each running for 10 seconds, arrive at the same time.
  - A runs from 0-10, B from 10-20, C from 20-30.
  - Average Turnaround Time = $(10 + 20 + 30)/3 = 20$ seconds.
- **The Convoy Effect:** FIFO performs poorly when short jobs get stuck behind a long one.
  - **Example:** Job A runs for 100s, while B and C run for 10s each. If A arrives first, it runs to completion.
  - Average Turnaround Time = $(100 + 110 + 120)/3 = 110$ seconds. This is highly inefficient.

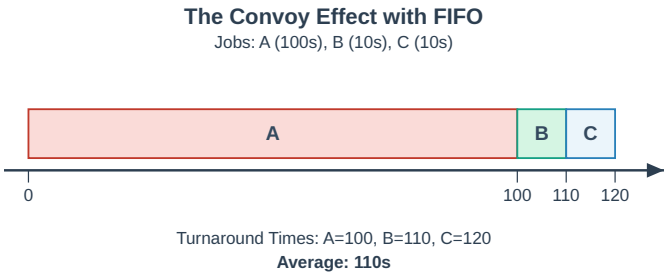


Figure 1: The Convoy Effect in FIFO Scheduling

2.2. Shortest Job First (SJF)

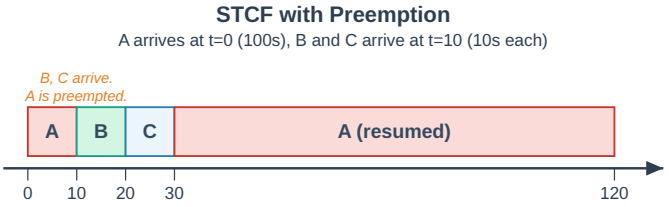
SJF is a non-preemptive policy designed to optimize turnaround time by prioritizing shorter jobs.

- **Policy:** Given a queue of jobs, always run the one with the shortest execution time next.
- **Example (Convoy Problem Solved):** Job A (100s), B (10s), C (10s).
  - SJF runs B (0-10), then C (10-20), then A (20-120).
  - Average Turnaround Time = $(120 + 10 + 20)/3 = 50$ seconds. A massive improvement over FIFO.
- **Optimality:** Given that all jobs arrive at the same time, SJF is a provably optimal algorithm for minimizing average turnaround time.

2.3. Shortest Time-to-Completion First (STCF)

What if jobs arrive at different times? A long job might start running, and then shorter jobs arrive, leading to the convoy effect again. To solve this, we introduce preemption.

- **Policy:** STCF (also known as **Preemptive Shortest Job First, PSJF**) is the preemptive version of SJF. When a new job arrives, the scheduler checks if its remaining run time is less than the remaining time of the *currently running* job. If so, it preempts the current job and runs the new, shorter one.
- **Example:** Job A arrives at t=0 (runs for 100s). Jobs B and C arrive at t=10 (each runs for 10s).
  1. At t=0, A starts running.
  2. At t=10, B and C arrive. A has 90s left. B and C each have 10s left.
  3. STCF preempts A and runs B.
  4. At t=20, B finishes. STCF runs C.
  5. At t=30, C finishes. STCF resumes A, which runs for its remaining 90s.
- **Result:** This approach is also provably optimal for minimizing average turnaround time, even when jobs arrive at different times.



---

3. A New Metric: Response Time

The introduction of interactive systems required a new metric beyond just turnaround time.

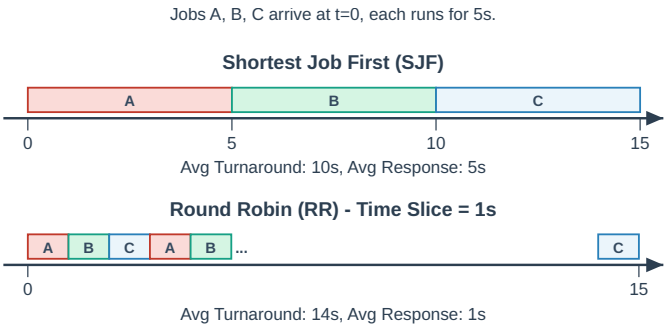
- **Response Time:** Measures how quickly a system responds to a user request.
  - **Formula:** $T_{\text{response}} = T_{\text{first run}} - T_{\text{arrival}}$
- **Problem:** SJF and STCF are terrible for response time. If three jobs arrive at once, the third job has to wait for the first two to complete entirely before it gets to run even once.

4. Round Robin (RR) Scheduling

RR is a preemptive policy designed to optimize for response time.

- **Policy:** RR runs a job for a fixed duration called a **time slice** (or quantum), then switches to the next job in the queue. It cycles through the ready jobs.
- **Example:** Three jobs (A, B, C), each needing 5s of CPU time. Time slice = 1s.
  - **RR Schedule:** A, B, C, A, B, C, ...
  - **Response Times:** A=0, B=1, C=2. Average = 1s.
  - **SJF Response Times:** A=0, B=5, C=10. Average = 5s.
- **Time Slice Length:** This is a critical parameter.
  - A **short** time slice improves response time.
  - A **very short** time slice increases overhead due to frequent context switching. The cost of the context switch must be amortized over a sufficiently long time slice.
- **Turnaround Time Trade-off:** RR is one of the worst policies for turnaround time because it stretches every job out for as long as possible.
  - In the example above, RR average turnaround is $(13+14+15)/3 = 14s$, while SJF is $(5 + 10 + 15)/3 = 10s$.

  Inherent Trade-off: Fairness and good response time (like in RR) often come at the cost of poor turnaround time. Unfair policies that prioritize short jobs (like SJF/STCF) achieve excellent turnaround time at the cost of poor response time for longer jobs.



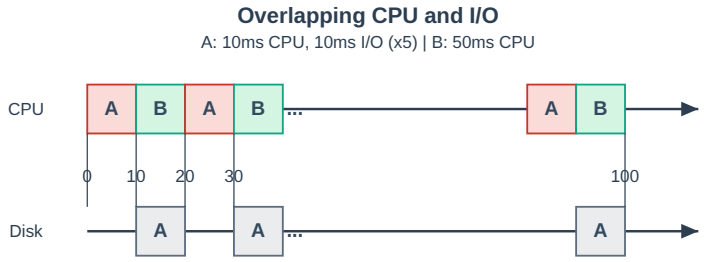
5. Incorporating I/O

So far, we've assumed jobs are CPU-only. Real programs perform I/O.

- **The Problem:** When a process initiates an I/O request, it becomes

blocked and cannot use the CPU. The scheduler should run another process during this time to keep the CPU busy.

- **The Solution:** A scheduler can treat a process as a series of CPU bursts. When a process blocks on I/O, the scheduler can run another job. When the I/O completes, the original process becomes ready again.
- **Example:** Job A (5 CPU bursts of 10ms each, with 10ms of I/O in between) and Job B (one CPU burst of 50ms).
  - An intelligent scheduler (like STCF) would run A's first 10ms burst.
  - While A is blocked on I/O, it would run 10ms of B.
  - When A's I/O completes, it becomes ready. Since its next CPU burst (10ms) is shorter than B's remaining time (40ms), it would preempt B and run.
  - This **overlap** of one process's I/O with another's CPU execution leads to much better system utilization.



6. The Final Challenge: No More Oracle

We have developed sophisticated schedulers, but they all rely on our most unrealistic assumption: that the OS knows the exact run time of each job in advance. In a real general-purpose OS, this is not the case.

The next challenge is to design a scheduler that can achieve the benefits of SJF/STCF (good turnaround time) and RR (good response time) *without* a priori knowledge of job lengths. This leads to the **multi-level feedback queue**, the topic of the next chapter.