Operating System
Lecture 7 - Process Scheduling

Yonghao Lee

May 27, 2025

Recap

We will first go over the terminology used in the last lecture.

In the following graph, there is a time axis. Jobs that are ready to be run compete for shared resources, such as the CPU. When a job arrives, it does not necessarily start running immediately; instead, it waits. This duration is shown in the graph as **waiting** or **wait time**. Then, the job starts running, and this period is the **run time**. The sum of the wait time and the run time is called the **response time** or **turnaround time**.

We discussed that interactive programs often have multiple CPU bursts, one after each event (e.g., an I/O operation). This means the program runs for a short period and then becomes blocked. Typically, these individual bursts from the same job are regarded as separate jobs for scheduling purposes.

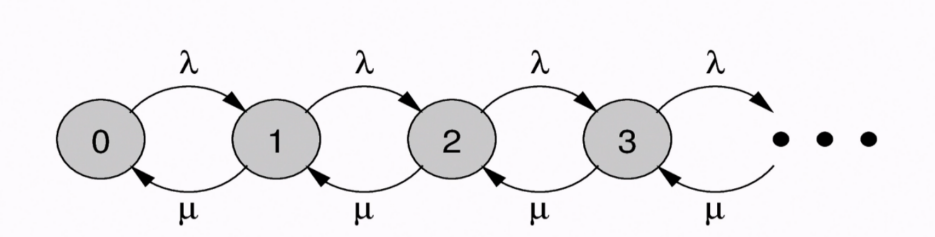

Figure 1: Timeline illustrating job states: waiting, running, and associated times.

Offline Schedulers

The most basic offline scheduler uses the First-Come, First-Served (FCFS) algorithm. With FCFS, jobs are run in the order in which they arrive. This approach can suffer from the **convoy effect**, where short jobs are delayed if a long job is scheduled first.

Shortest Job First (SJF) is an improved offline algorithm compared to FCFS. It achieves the optimal average wait time by always selecting the shortest job to run next.

However, these offline schedulers operate under somewhat unrealistic assumptions, such as knowing the exact number of jobs and their run times beforehand.

Online Schedulers

A basic online scheduling algorithm is Shortest Remaining Processing Time (SRPT). When arrival times and the total number of jobs are unknown, preemption is used. Preemption allows the scheduler to reconsider its decisions, which is valuable due to the lack of foresight. SRPT always runs the job that currently has the shortest remaining time.

The following graph demonstrates SRPT. We observe that after Job 2 arrives (2 seconds after Job 1 started running), a preemption occurs to switch to Job 2 because its run time is currently lower than Job 1's remaining time.

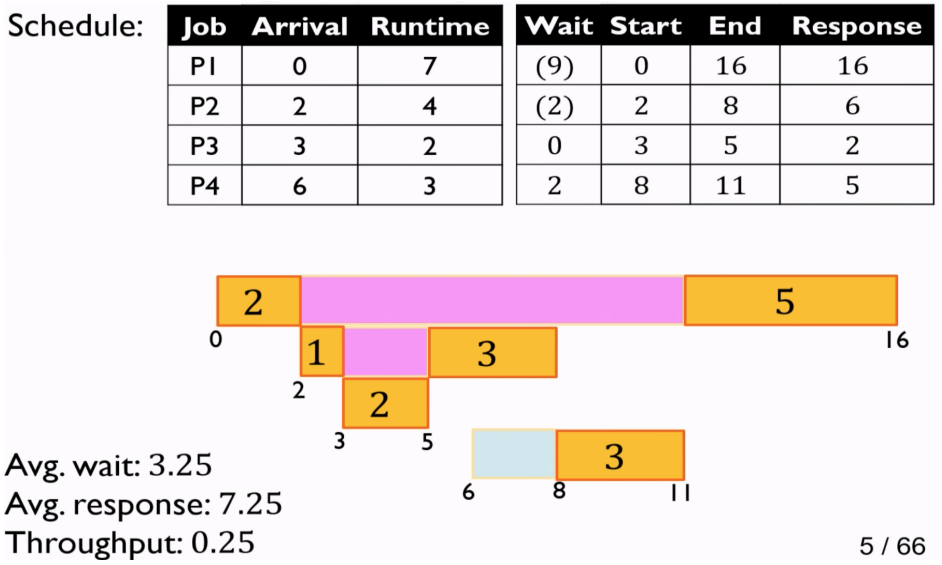

Figure 2: Example of Shortest Remaining Processing Time (SRPT) with preemption.

Processor Sharing Model

The Processor Sharing (PS) model is an idealized concept that cannot be perfectly implemented in practice. Theoretically, it assumes the processor can be divided into infinitesimally small slices, with each of the N active processes receiving $1/N$ of the processor's power simultaneously. As illustrated in the diagram below, when new processes arrive, each existing and new process receives a progressively smaller, equal share of the CPU. For instance, if four processes are active, each effectively uses one-quarter of the CPU's capacity at the same time.

The primary advantage of this model is its fairness and responsiveness for short jobs. Even if a very long job is running, newly arrived short jobs can start processing immediately in parallel, albeit at a reduced rate for all jobs. Consequently, short jobs do not need to wait for long jobs to complete.

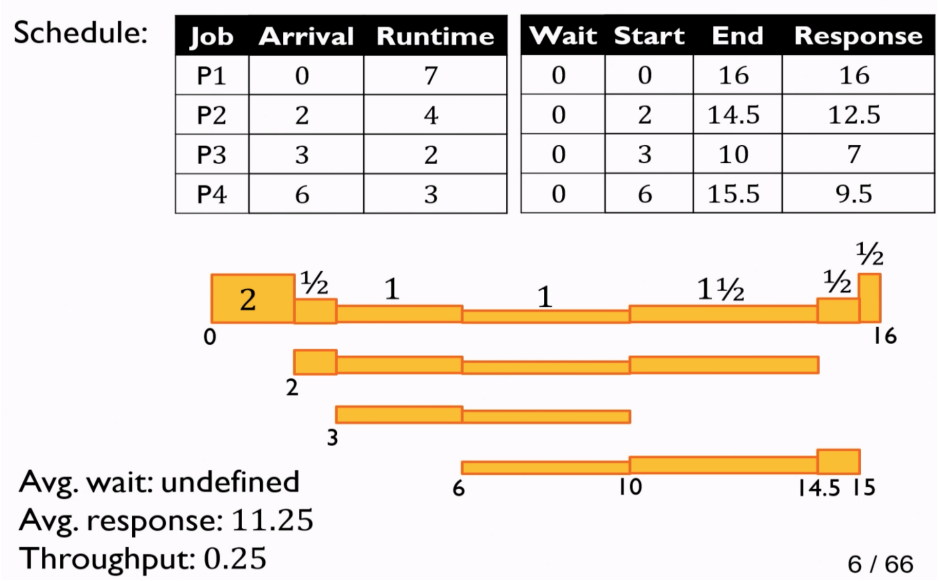

Figure 3: Conceptual illustration of the Processor Sharing model.

Round-Robin (RR) Scheduling

Round-Robin (RR) scheduling is a practical algorithm that aims to approximate the behavior of the ideal Processor Sharing model. In RR, each process is allocated a small, predefined unit of CPU time called a **time quantum** (or time slice). Processes are typically managed in a circular queue. A process runs until its time quantum expires or it blocks (e.g., for I/O). If the quantum expires and the process has not completed, it is preempted and placed at the end of the queue.

A notable disadvantage of RR is the overhead associated with context switches. If the time quantum is too small, frequent context switches can consume a significant portion of CPU time, degrading overall system performance.

It's important to note that RR operates in an online setting, using preemption to adapt to a lack of complete future knowledge about job arrivals or lengths. It provides uniform treatment to all jobs, ensuring that each gets a turn. While fair, this uniform treatment might not always be optimal, suggesting there is often room for further improvement with more advanced scheduling algorithms.

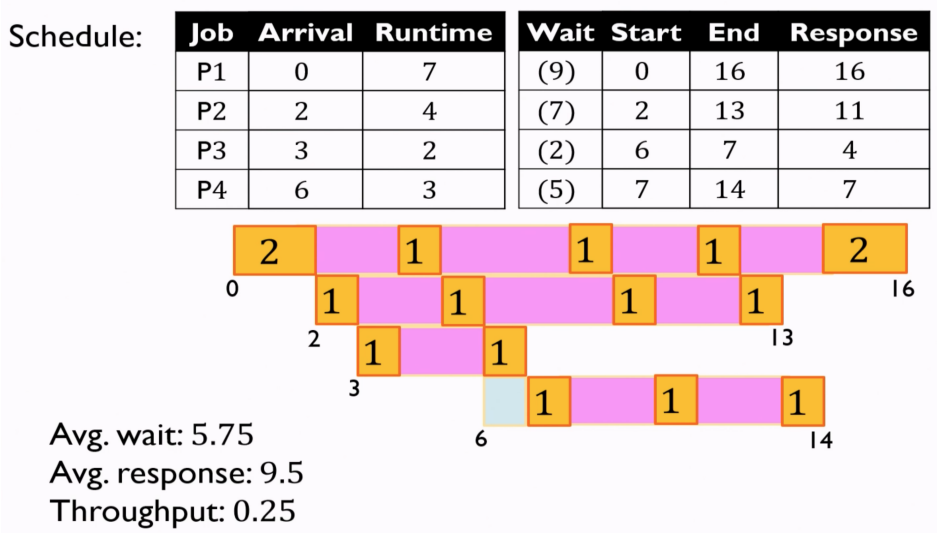

Figure 4: Round Robin Scheduling example showing how processes take turns with the CPU.

Using Accounting Data

Ideally, we would like to use an algorithm like Shortest Remaining Processing Time (SRPT), or at least approximate its behavior, to optimize for metrics like average response time. However, a significant challenge is that we often do not know the exact run times of incoming jobs in advance. At best, we might only have a very coarse-grained estimation based on past behavior or other "accounting data."

This lack of perfect information can lead to suboptimal scheduling decisions. For instance, a process that will eventually be very long might exhibit short CPU bursts initially. If our estimation mistakenly classifies this long process as short, it could be prioritized over genuinely short processes. This would delay the execution of jobs that should have been completed quickly.

To mitigate such scenarios, a common strategy is to "hedge" our scheduling decisions, often using Round Robin (RR) as a component. The approach could be:

- Attempt to approximate SRPT by estimating run times and prioritizing jobs predicted to be the shortest.
- Concurrently, employ Round Robin principles. If estimations are incorrect or if it's difficult to distinguish job lengths, RR ensures that all processes receive a fair share of CPU time. This acts as an "insurance policy," preventing any single job (especially one mistakenly identified as short) from monopolizing the CPU and starving other jobs.

In this way, while we strive for the efficiency of SRPT through estimation, Round Robin provides a safety net against the inevitable errors in those estimations.

Heavy-Tail Distribution Properties

Consider a scenario in our system: jobs arrive to be run, but we do not know their exact run times (i.e., whether they are short or long). We might only know that, on average, jobs typically run for 10 seconds. Now, suppose a particular job has already been running for 9 seconds. A critical question arises: should we allow this job to continue, hoping it's short and about to finish, or should we consider preempting it because it might actually be a very long job, and continuing it could delay other, potentially shorter, jobs?

To address this, we examine the properties of **heavy-tail distributions**, which are often characteristic of job runtimes in computer systems:

- **Definition of a Heavy Tail:** A distribution is said to have a "heavy tail" if the probability of observing very large values (long runtimes) decreases much more slowly than it would for an exponential distribution (e.g., slower than e^{-kx}). This means that while extremely long jobs are not the most common, they occur frequently enough to significantly impact system behavior. The "tail" of the probability distribution (representing these long jobs) is thus "heavier" or "thicker."

- **Coefficient of Variation (CV):** The coefficient of variation is defined as:

  $$CV = \frac{\text{standard deviation}}{\text{mean}}$$

  For distributions characterizing job runtimes, a heavy tail usually implies a $CV > 1$. This indicates a large spread in job sizes relative to the average; there are many short jobs but also a significant number of very long jobs.

- **Expected Remaining Time:** This is perhaps the most crucial property for scheduling decisions. We are interested in the conditional expectation:

  $$\mathbb{E}[X - x_0 \mid X \geqslant x_0]$$

  This formula represents the average remaining processing time $(X - x_0)$ for a job, given that it has already run for a duration of x_0 seconds (where X is the job's total runtime). For a heavy-tail distribution, this expected remaining time typically exhibits a counter-intuitive behavior: it is **monotonously increasing** with x_0.

Distribution Examples and Their Properties

To better understand the implications of different runtime distributions on scheduling decisions, let's examine several common distribution types and their key properties.

Dirac (δ) Distribution Example The Dirac delta distribution, $\delta(x - a)$, represents an idealized scenario where a job's runtime is known with absolute certainty to be a single, specific value 'a'. It is a **generalized function**.

Definition:

$$\delta(x - a) = \begin{cases} 0 & \text{if } x \neq a \\ \infty & \text{if } x = a \end{cases}$$

This means the entire probability is concentrated at the single point $x = a$. **Key Properties:**

- **No Tail:** Since the runtime is exactly 'a', there's no possibility of it being longer. Thus, the distribution has no "tail" extending to larger values. This is the opposite of heavy-tailed distributions.
- **Expected Value (Mean):** a If every job runs for exactly 'a' seconds, the average runtime is simply 'a'.
- **Standard Deviation:** 0 As there's no variation from the mean (all runtimes are exactly 'a'), the spread (standard deviation) is zero.
- **Coefficient of Variation (CV):** $CV = \frac{\text{Standard Deviation}}{\text{Mean}} = \frac{0}{a} = 0$ (assuming $a > 0$). Since $CV < 1$, this clearly indicates it's not a heavy-tailed distribution.
- **Expected Remaining Time:** $E(X - x_0 \mid X \geqslant x_0) = a - x_0$ If a job (which always runs for 'a' seconds) has already run for x_0 seconds, its remaining runtime is precisely $a - x_0$.
- **Decreasing with x_0:** The expected remaining time $(a - x_0)$ gets smaller as x_0 (the time already run) increases. The longer it has run, the less time is left. This is intuitive for a fixed-length job and contrasts with the behavior of heavy-tailed distributions.

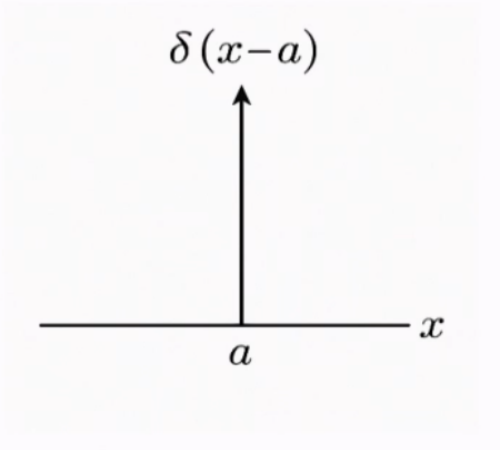

Figure 5: The Dirac delta function $\delta(x - a)$, concentrated at $x = a$.

Uniform Distribution Example The uniform distribution represents a scenario where a job's runtime is equally likely to be any value between 0 and a. This models situations where we know the maximum possible runtime, but have no reason to expect any particular value within that range.

Definition:

$$f(x) = \frac{1}{a} \quad \text{for } 0 \leqslant x \leqslant a$$

This means the probability density is constant across the entire range from 0 to a.

Key Properties:

- **No Tail:** Since the runtime has an upper bound of 'a', there's no possibility of it being longer. Thus, the distribution has no "tail" extending beyond a.
- **Expected Value (Mean):** $\frac{a}{2}$ The average runtime is halfway between 0 and a.
- **Standard Deviation:** $\frac{a}{\sqrt{12}}$ This is derived from the variance formula for uniform distributions, which is $\frac{(b-a)^2}{12}$. In our case, $b = a$ and $a = 0$, so the variance is $\frac{a^2}{12}$.
- **Coefficient of Variation (CV):** $CV = \frac{\text{Standard Deviation}}{\text{Mean}} = \frac{a/\sqrt{12}}{a/2} = \frac{1}{\sqrt{3}} \approx 0.577 < 1$ Since $CV < 1$, this indicates it's not a heavy-tailed distribution.
- **Expected Remaining Time:** $E(X - x_0 \mid X \geqslant x_0) = \frac{a-x_0}{2}$ If a job has already run for x_0 seconds (and we know it hasn't finished yet), the expected remaining runtime is half the distance from x_0 to a. This is because for $X \geqslant x_0$, X is uniformly distributed between x_0 and a.
- **Decreasing with x_0:** The expected remaining time ($\frac{a-x_0}{2}$) decreases as x_0 increases. This is intuitive: the longer a job has run, the less time we expect it to continue running.

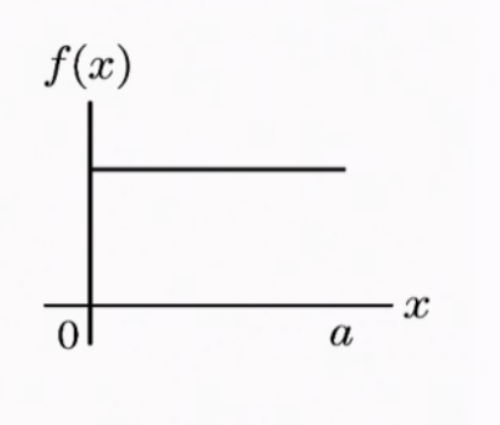

Figure 6: The uniform distribution function $f(x) = \frac{1}{a}$ for $0 \leqslant x \leqslant a$.

Normal Distribution Example The normal (Gaussian) distribution represents a scenario where a job's runtime tends to cluster around a central value, with symmetric variation in both directions. In practice, we often truncate at zero since negative runtimes are impossible.

Definition:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

where μ is the mean and σ is the standard deviation of the distribution.

Key Properties:

- **Light Tail:** The normal distribution has exponentially decreasing tails, meaning the probability of extremely long runtimes diminishes very rapidly. This contrasts with heavy-tailed distributions where extremely long runtimes have a significant probability.
- **Expected Value (Mean):** μ The average runtime centers around μ.
- **Standard Deviation:** σ This parameter controls how dispersed the runtimes are around the mean.
- **Coefficient of Variation (CV):** $CV = \frac{\sigma}{\mu}$ For typical process runtimes, $\sigma < \mu$, so $CV < 1$, indicating it's not a heavy-tailed distribution.
- **Expected Remaining Time:** $E(X - x_0 \mid X \geqslant x_0)$ is decreasing with x_0. If a job has already run for x_0 time units and hasn't completed, the expected additional time it will take decreases as x_0 increases. This property is characteristic of light-tailed distributions.

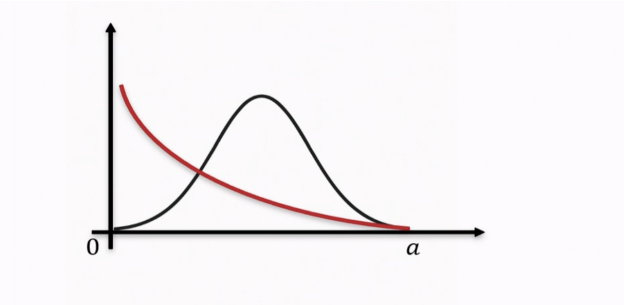

Figure 7: The normal distribution showing a black curve for the probability density function and a red curve illustrating how the expected remaining time $E(X - x_0 \mid X \geqslant x_0)$ decreases as x_0 increases.

Exponential Distribution Example The exponential distribution represents a scenario where job runtimes have a constant "hazard rate," meaning the probability of a job completing in the next instant is independent of how long it has already run. This distribution is uniquely characterized by the memoryless property.

Definition:

$$f(x) = \lambda e^{-\lambda x} \quad \text{for } 0 \leqslant x < \infty$$

where $\lambda > 0$ is the rate parameter, representing events (job completions) per unit of time.

Key Properties:

- **Rate Parameter λ:** The parameter λ represents the rate at which events occur per unit time. For instance, if $\lambda = 2$, we expect 2 job completions per time unit on average.

- **"Transition" Tail:** The exponential distribution sits at the boundary between light-tailed and heavy-tailed distributions. Its tail decreases fast enough to have finite moments of all orders, but slower than distributions like the normal distribution.
- **Expected Value (Mean):** $\frac{1}{\lambda}$ The average runtime of a job is the reciprocal of the rate parameter. If jobs complete at a rate of $\lambda = 0.1$ per second, the average runtime is $\frac{1}{0.1} = 10$ seconds.
- **Standard Deviation:** $\frac{1}{\lambda}$ Interestingly, the standard deviation equals the mean, indicating a significant level of variability.
- **Coefficient of Variation (CV):** $CV = \frac{\text{Standard Deviation}}{\text{Mean}} = \frac{1/\lambda}{1/\lambda} = 1$ With $CV = 1$, the exponential distribution marks the threshold between light-tailed ($CV < 1$) and heavy-tailed ($CV > 1$) distributions.
- **Memoryless Property:** $E(X - x_0 \mid X \geqslant x_0) = \frac{1}{\lambda}$ This is the exponential distribution's most distinctive characteristic. The expected remaining time is constant ($\frac{1}{\lambda}$) regardless of how long the job has already run. A job that has been running for 10 seconds has the same expected remaining time as one that just started.

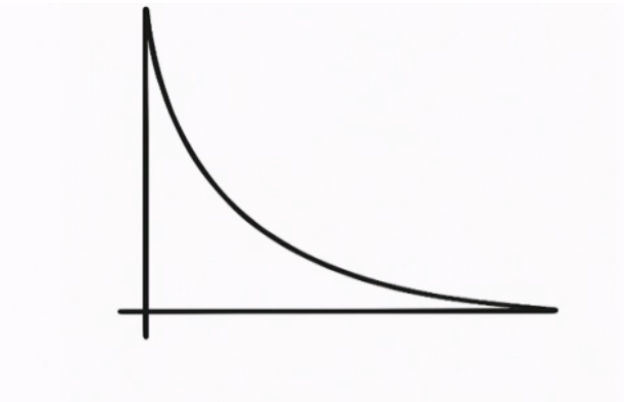

Figure 8: The exponential distribution with parameter λ. The curve shows the probability density function $f(x) = \lambda e^{-\lambda x}$, which starts at height λ when $x = 0$ and decreases exponentially.

Power Law (Pareto) Distribution Example The Power Law distribution (often referred to as Pareto in this context) is crucial for modeling phenomena where a small number of items are high-frequency/high-magnitude, while a large number of items are low-frequency/low-magnitude (e.g., many short computing jobs and a few very long ones).

Definition (Probability Density Function):

$$f(x) = \alpha x^{-(\alpha+1)} \quad \text{for } \alpha > 1, \text{ and } 1 \leqslant x < \infty$$

where α is the shape parameter that determines how quickly the tail of the distribution decays.

Key Properties:

- **Heavy Tail:** The Pareto distribution is the classic example of a heavy-tailed distribution. The probability of extremely large values decreases much more slowly than in exponential or normal distributions, making very long runtimes significantly more probable.
- **Coefficient of Variation (CV):** $CV = \infty > 1$ (for $\alpha \leqslant 2$) When $\alpha \leqslant 2$, the variance is infinite, making the CV infinite. Even for $\alpha > 2$, the CV remains greater than 1, which is characteristic of heavy-tailed distributions.
- **Expected Value (Mean):** $\mathbb{E}[X] = \frac{\alpha}{\alpha-1}$ for $\alpha > 1$ If $\alpha \leqslant 1$, the mean is infinite.
- **Variance:** $\text{Var}(X) = \frac{\alpha}{(\alpha-1)^2(\alpha-2)}$ for $\alpha > 2$ If $\alpha \leqslant 2$, the variance is infinite.
- **Expected Remaining Time:** $E(X - x_0 \mid X \geqslant x_0) = \frac{x_0}{\alpha-1}$ This formula reveals a critical property of heavy-tailed distributions: the expected remaining time *increases* with x_0. This means that the longer a job has been running, the longer you should expect it to continue running.
- **Increases with x_0:** Unlike light-tailed distributions where remaining time decreases with job age, in Pareto distributions, the expected remaining time grows linearly with how long the job has already run. This counterintuitive property is fundamental to understanding the scheduling challenges posed by heavy-tailed workloads.

Scheduling Implications: For skewed distributions with heavy tails (CV > 1), the paradoxical property emerges: the longer a job survives (continues running), the longer it still has to run (on average).

Learning About Jobs from Runtime Behavior

A key takeaway from our discussion of heavy-tail distributions is the principle: the longer a job has been running, the longer its expected remaining runtime becomes (assuming a heavy-tailed distribution of job sizes).

This has direct implications for scheduling:

- If a job completes its assigned time quantum without finishing, we gain information. This survival suggests an increased likelihood that the job is inherently long.
- Consequently, it often makes sense to assign such a job a lower priority compared to new jobs. New jobs, about which we initially know nothing, are, on average (before any execution), more likely to be short than a job that has already proven it can run for at least one full quantum.

Multi-Level Feedback Queues (MLFQ)

The insights from "learning about jobs" lead directly to the design of Multi-Level Feedback Queues (MLFQ). Instead of returning a job to the same run queue after it completes its time quantum (as in a simple Round Robin system), an MLFQ employs multiple queues, typically representing different priority levels.

- When a job uses its entire quantum in a higher-priority queue (implying it's longer than initially hoped), it is demoted to a "lower" or "slower" queue.
- The system might have several such queues: e.g., a "fast" queue for very short jobs, a "medium" queue, and one or more "slow" queues for long-running jobs.
- The general scheduling principle is to always pick jobs from the highest-priority queue that is not empty. Longer jobs in lower-priority queues only run when no shorter (or higher-priority) jobs are ready.

This mechanism allows the scheduler to adaptively categorize jobs based on their observed behavior.

Multi-Level Feedback Queue (MLFQ) Demonstration: Path of a New Job

This explanation details how a new job (which we'll refer to as the "red ball," as per the visual in the lecture slides) progresses through a Multi-Level Feedback Queue (MLFQ) system.

The MLFQ Setup:

- Multiple queues are used, ordered by priority (e.g., Queue 0 = highest, Queue 1 = next, etc.).
- Each queue i is assigned a specific time quantum Q_i. In our example:
  - Queue 0: Quantum $Q_0 = 10$ time units.
  - Queue 1: Quantum $Q_1 = 20$ time units.
  - Queue 2: Quantum $Q_2 = 40$ time units.
  - Queue 3: Quantum $Q_3 = 80$ time units.
- **Key Scheduling Rules:**
  (a) New jobs enter the highest-priority queue (Queue 0).
  (b) The scheduler always executes jobs from the highest-priority non-empty queue.
  (c) If a job in queue i uses its entire time quantum Q_i without completing or blocking, it is preempted and demoted to the next lower-priority queue $(i + 1)$.
  (d) If a job completes or blocks before its quantum expires, it leaves the CPU (and potentially the queueing system or moves to a wait state).

(e) Jobs within the same queue are scheduled using Round Robin if that queue contains multiple jobs.

Tracing the "Red Ball" (a New Job) Through the MLFQ:
Let's trace the path of our "red ball," assuming it's a new job entering the system:

1. **Arrival into Queue 0:** The "red ball" enters the system and is placed in Queue 0, the queue with the highest priority.

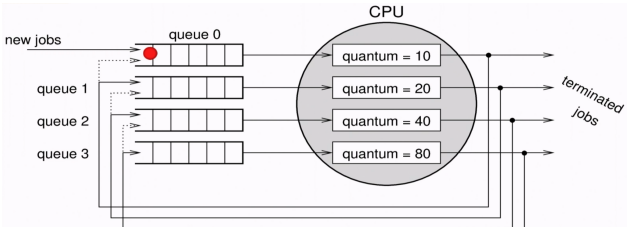

Figure 9: Stage 1: New job enters the highest-priority queue

2. **Execution from Queue 0:** When Queue 0 is selected (i.e., it's the highest-priority queue with jobs), the "red ball" runs on the CPU for up to its assigned quantum of $Q_0 = 10$ time units.

   - **If the job is short** (requires $\leqslant 10$ units): It completes and exits the system. It received fast service.
   - **If the job is longer** (requires > 10 units): It uses all 10 units of its quantum, is preempted, and is then demoted to Queue 1.

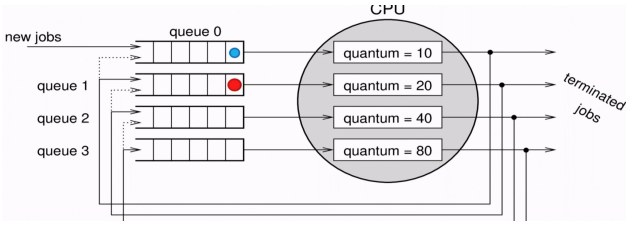

Figure 10: Stage 2: Job is demoted to Queue 1 after using its full quantum in Queue 0

3. **Execution from Queue 1:** The "red ball" now resides in Queue 1. It will only run if Queue 0 is empty. When selected, it runs for up to its quantum of $Q_1 = 20$ time units.

   - **If the job finishes** (needs $\leqslant 20$ additional units): It completes and exits.
   - **If the job is still longer**: It uses all 20 units, is preempted, and is demoted to Queue 2.

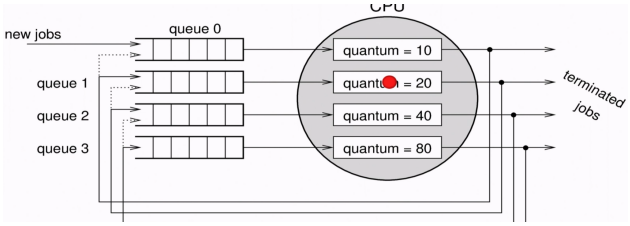

Figure 11: Stage 3: Job is further demoted to Queue 2 after using its full quantum in Queue 1

4. **Execution from Queue 2:** In Queue 2 (which runs only if Queues 0 and 1 are empty), the "red ball" gets a quantum of $Q_2 = 40$ time units.
   - **If the job finishes**: It completes and exits.
   - **If the job is still longer**: It's demoted to Queue 3.
5. **Execution from Queue 3 (Lowest Priority):** In Queue 3 (runs only if Queues 0, 1, and 2 are empty), the "red ball" gets a larger quantum of $Q_3 = 80$ time units.
   - **If the job finishes**: It completes and exits.
   - **If the job is even longer / uses its full quantum**: It will typically remain in Queue 3 and share CPU time with other jobs in Queue 3 (often via Round Robin) whenever the higher-priority queues are all empty.

Summary of MLFQ Rules:

- New jobs always enter the highest-priority queue.
- The scheduler invariably selects jobs from the highest-priority non-empty queue. Jobs in lower-priority queues are not considered for execution if higher-priority queues have ready jobs.
- If a job exhausts its allocated time quantum in its current queue without completing, it is demoted to the next lower-priority queue.
- Within a single queue, jobs are typically scheduled using a Round Robin approach if multiple jobs are present and ready.

Addressing Starvation in MLFQ

A potential issue with the basic MLFQ strategy is **starvation**. What if new, short jobs (which enter the highest-priority queues) continue to arrive incessantly? In such a scenario, longer jobs in lower-priority queues might be starved—they might never get a chance to run.

Several solutions can mitigate starvation:

Solution 1: CPU Time Allocations

One approach is to assign a relative percentage of CPU time to each queue:

- Higher-priority queues (containing presumably shorter jobs) receive a larger percentage of CPU time.
- Crucially, lower-priority queues (for longer jobs) receive a non-zero allocation. This ensures that even the lowest-priority jobs will eventually get some CPU time, preventing complete starvation.

For example: A scheduler might allocate 60% of CPU time to Queue 0, 25% to Queue 1, 10% to Queue 2, and 5% to Queue 3. Once Queue 0 has used its 60% budget (within a larger scheduling interval), the scheduler moves to consider Queue 1 with its budget, and so on. This guarantees that Queue 3 will eventually receive its 5% share.

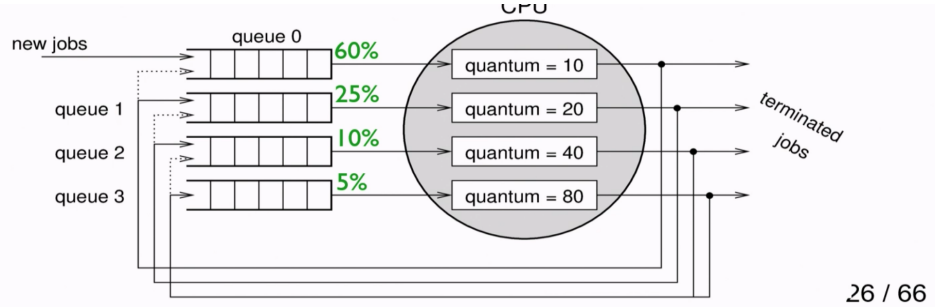

Figure 12: Example of CPU Time Allocations across Queues in MLFQ.

Solution 2: Aging

Aging is another common technique to prevent starvation. It works on a "negative feedback" principle for priority based on runtime, and a "positive feedback" principle for priority based on waiting time:

- **Running reduces priority (Negative Feedback):** As a job consumes CPU time, its priority might effectively decrease (e.g., by being moved to a lower-priority queue, as in standard MLFQ).
- **Waiting increases priority (Positive Feedback):** Conversely, if a job has been waiting in a lower-priority queue for a long time without running, its priority is gradually increased. This might eventually elevate it to a higher-priority queue where it has a better chance of being scheduled.

System Overload Management

An interesting perspective on scheduling is how systems handle overload conditions:

- **Overload Definition:** "Load" is the fraction of system capacity that users want. Overload occurs when demand exceeds 100% of capacity.

- **Inevitable Reality:** When short jobs continuously arrive (a common scenario), the system becomes overloaded, making it impossible to run everything immediately.
- **Strategic Sacrifice:** The system must decide what not to run. Sacrificing long-running jobs often makes strategic sense because:
  - They are typically non-interactive (nobody is actively waiting for them)
  - One long job can be traded for many short jobs, improving overall throughput and user satisfaction
  - Long jobs can eventually run when the load abates (e.g., at night)
- **Feature, Not Bug:** What appears as "starvation" of long jobs can be viewed as an intentional design choice to optimize overall system responsiveness under high load.

Classic Unix Scheduler

The classic Unix scheduler provides an instructive example of how these scheduling principles are applied in practice:

- **Priority Levels:** The system uses 128 queues corresponding to 128 priority levels:
  - Levels 0-49: Reserved for kernel processes
  - Levels 50-127: Used for user mode processes
- **Priority Formula:** For user processes, priority is calculated approximately as:
  - User priority $\approx$ cpu_usage + base (where base = 50)
  - Lower numerical values represent higher priorities
- **Clock-Based Accounting:** The Unix scheduler updates priorities based on clock ticks:
  - On each clock tick (1/100th of a second):
    - Add 1 to the running process's cpu_usage (stored in the Process Control Block)
    - If a higher priority process exists, switch to it
  - At the end of a burst (quantum = 10 ticks or when process blocks):
    - Switch to the next process at the same priority level (Round Robin)
  - Every second (100 clock ticks):
    - Divide the cpu_usage of all processes by 2 (effectively increasing their priority)
    - Recalculate priorities
- **Aging Mechanism:** The system incorporates both aging and feedback:
  - cpu_usage is incremented when a process runs, reducing its priority

  - cpu_usage is divided by 2 every second the process does not run, increasing its priority
  - This creates an "exponential aging" effect for waiting processes

- **Scheduling Policy:** The system always schedules the highest-priority process that is ready to run
- **Priority Assignment for Kernel Processes:** Priorities are based on the reason for sleep:
  - Disk I/O: Priority level 20 (higher priority)
  - Terminal I/O: Priority level 28 (medium-high priority)
  - Network I/O: Priority level 32 (medium priority)
- **Time-Sharing:** Within each priority level, processes share the CPU using a time-slicing approach

Computational vs. Interactive Process Treatment

Modern schedulers, including MLFQ implementations, must distinguish between different types of processes:

- **Computational Processes:** These CPU-intensive processes typically receive lower priority as they continue to consume their full time quanta.
- **Simple Interactive Processes:** Programs like text editors that wait for user input receive higher priority and run immediately when ready, ensuring responsive user experience.
- **Complex Interactive Processes:** Applications like 3D games or graphics-intensive programs require substantial CPU for rendering but are still interactive. These present a scheduling challenge and may receive lower priority under naive scheduling algorithms.
- **Modern Enhancements:** Contemporary schedulers employ additional heuristics to better identify and prioritize interactive processes, such as:
  - Identifying the currently active window or application
  - Tracking I/O patterns characteristic of interactive use
  - Maintaining responsive user interfaces even for CPU-intensive applications

This design reflects a broader understanding that modern operating systems must balance CPU-bound jobs with I/O-bound jobs, moving beyond the simple CPU burst job model discussed in earlier sections. The Unix scheduler accomplishes this by giving preference to interactive and I/O-bound processes (which tend to have shorter CPU bursts) while still ensuring that CPU-intensive processes make progress over time.

Performance Evaluation of Schedulers

After exploring various scheduling algorithms, we need methods to evaluate their performance. This section outlines approaches for analyzing scheduler effectiveness.

Methods of Scheduler Evaluation

There are three principal approaches to evaluate scheduling algorithms:

- **Queuing Models:**
  - Use queuing theory to solve a stochastic model and derive average performance metrics
  - Based on simplifying mathematical assumptions
- **Simulation:**
  - Use a program that implements a model of a computer system
  - Model is a simplification of reality
- **Implementation:**
  - Put the actual algorithm in a real system for evaluation under real operating conditions

Understanding Queuing System Models

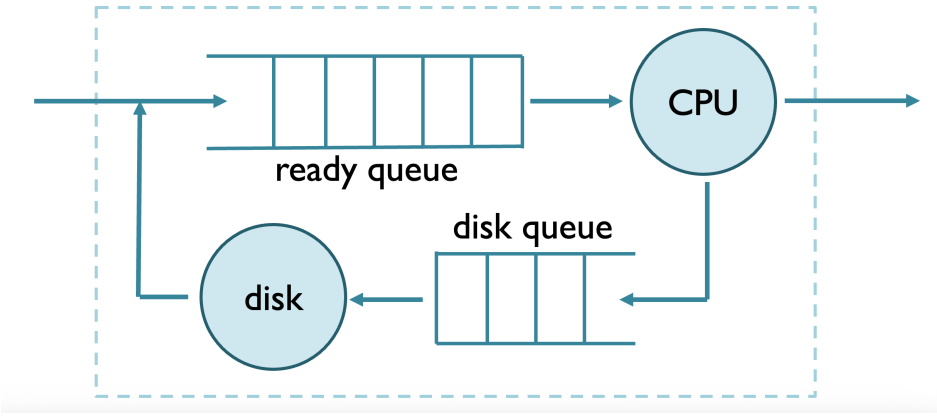

Figure 13: System Model with CPU and Disk Queues

The diagram illustrates several key components of a queuing system model:

- **Ready Queue:** Where processes wait to use the CPU
- **CPU:** The server that processes jobs from the ready queue

- **Disk Queue:** Where processes wait for disk I/O operations
- **Disk:** The server that handles I/O requests

The arrows show the flow of processes through the system:

- Processes enter the system and join the ready queue
- The CPU takes processes from the ready queue
- After CPU processing, some processes may need disk I/O and move to the disk queue
- The disk handles the I/O requests
- After disk operations complete, processes return to the ready queue
- This creates a feedback loop where processes can cycle between CPU and disk
- Eventually, processes complete and exit the system

This model, while simplified, captures the essential dynamics of how processes flow through a computer system, competing for limited resources and waiting in queues when resources are busy.

Mathematical Analysis of Schedulers

To analyze a scheduler like FCFS (First-Come-First-Served), we make specific assumptions about the system:

- **Assumption on the incoming job rate:**
  - Memoryless, fixed average rate λ (lambda)
  - This typically refers to a Poisson arrival process, where the time between arrivals follows an exponential distribution
- **Assumption on the processing time and order:**
  - Memoryless, fixed average rate μ (mu) with unlimited queue capacity
  - "Memoryless" here refers to service times following an exponential distribution
  - FCFS as the service discipline
- **Goal:** Find the average response time (queuing analysis)
  - First find the average queue length using Markov chains

A Note on Notation

In queuing theory, we use Kendall's notation to describe queuing systems:

- **M/M/1:**
  - **Arrivals:** M = Memoryless
  - **Processing time:** M = Memoryless
  - **Number of servers:** 1

The letter "M" stands for "Memoryless," which is a property of the exponential distribution. This property means that the future behavior of the system depends only on its current state, not on its past history.

In an M/M/1 queuing system:

- The first "M" indicates that job arrivals follow a memoryless process (specifically, a Poisson process with rate λ)
- The second "M" indicates that service times also follow a memoryless distribution (exponential distribution with average rate μ)
- The "1" indicates there is a single server processing jobs

M/M/1 Queue Analysis

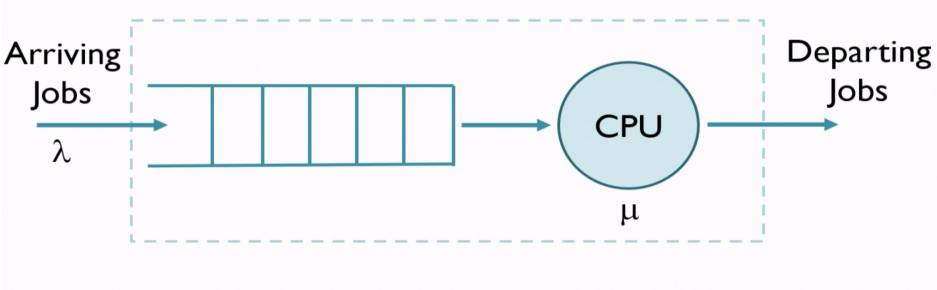

Figure 14: M/M/1 Queue with Arrival Rate λ and Service Rate μ

In our example, we have a single server, arrivals at rate λ, and service at rate μ. For stability, we require that $\lambda \leqslant \mu$.

For simplicity in our analysis, we use the exponential distribution which, while not strictly heavy-tailed, is sufficiently close for our purposes.

Little's Law

We know λ and μ, and want to find $\bar{r}$ (average response time). We can use Little's Law:

$$\bar{n} = \lambda \cdot \bar{r}$$

Where $\bar{n}$ is the average number of processes in the system.

To calculate the average queue length, we model the system as a Markov chain. The states of the chain represent the total number of processes in the system (both running and waiting).

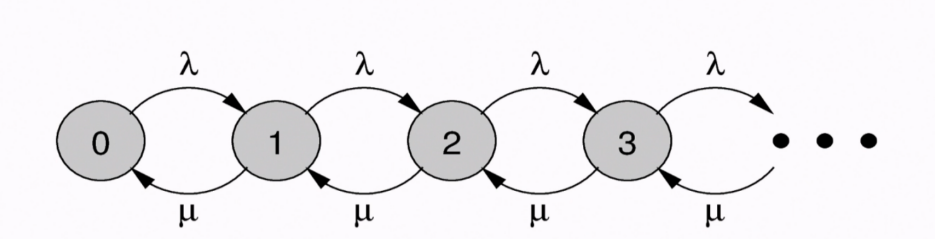

Figure 15: Markov Chain Representing M/M/1 Queue States

The system moves from state i to state $i+1$ at rate λ and from state $i+1$ to state i at rate μ. For a stable system, we expect to spend most time near state 0, since $\lambda \leqslant \mu$.

Markov Chain Analysis

The states of Markov chains have limiting probabilities. That is, if we let the system run for a sufficiently long time, we have a specific probability of being in each state. This is a property of ergodic Markov chains (connected and without periodic cycles).

If we denote the probability of being in state i as π_i, we can calculate these probabilities using flow balance equations. The flow from state i to $i+1$ equals the flow in the opposite direction:

$$\pi_0 \cdot \lambda = \pi_1 \cdot \mu \Rightarrow \pi_1 = \frac{\lambda}{\mu}\pi_0$$

Continuing this pattern, we get:

$$\pi_i = \left(\frac{\lambda}{\mu}\right)^i \pi_0$$

Let's look at a concrete example with $\lambda = 1$ and $\mu = 2$:

Example: Calculating State Probabilities for M/M/1

Let's calculate the steady-state probabilities π_i for an M/M/1 queue with the following parameters:

- Arrival rate, $\lambda = 1$ job per unit time.
- Service rate, $\mu = 2$ jobs per unit time.

Step 1: Calculate the System Load (ρ) The system load (or utilization factor) ρ is given by:

$$\rho = \frac{\lambda}{\mu} = \frac{1}{2} = 0.5$$

Step 2: Determine the Probability of an Empty System (π_0) For a stable M/M/1 queue ($\rho < 1$), the probability that the system is empty is:

$$\pi_0 = 1 - \rho = 1 - \frac{1}{2} = \frac{1}{2}$$

Step 3: Calculate Probabilities for Other States The probability of being in state i is:

$$\pi_i = \rho^i \pi_0$$

- **For State 1:** $\pi_1 = (0.5)(0.5) = \frac{1}{4}$
- **For State 2:** $\pi_2 = (0.5)^2(0.5) = \frac{1}{8}$
- **For State 3:** $\pi_3 = (0.5)^3(0.5) = \frac{1}{16}$
- **For State 4:** $\pi_4 = (0.5)^4(0.5) = \frac{1}{32}$

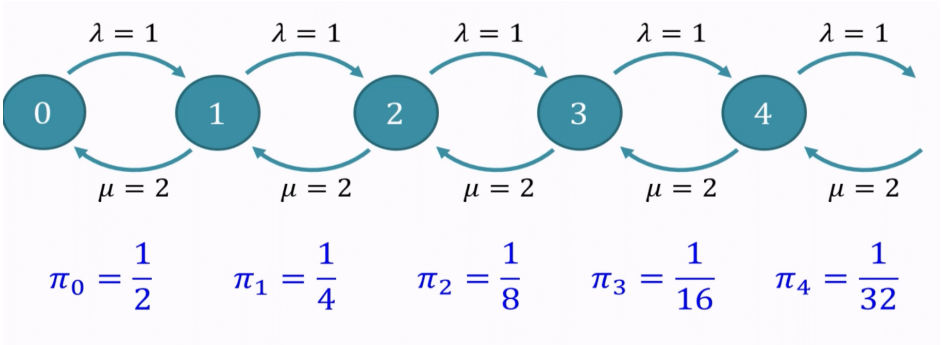

Figure 16: Example Calculation of State Probabilities

With these probabilities, we can derive the general formula for the expected number of jobs in the system:

$$\bar{n} = \sum_{i=0}^{\infty} i \cdot \pi_i = \sum_{i=0}^{\infty} i(1-\rho)\rho^i = \frac{\rho}{1-\rho}$$

And using Little's Law, the average response time is:

$$\bar{r} = \frac{\bar{n}}{\lambda} = \frac{\rho}{\lambda(1-\rho)} = \frac{1/\mu}{1-\rho}$$

Understanding the "Probabilistic Nature" of the M/M/1 Result

The M/M/1 queuing model gives us the result $\bar{r} = \frac{1/\mu}{1-\rho}$, which produces a characteristic response time curve that rises sharply as utilization approaches 100%. This behavior is fundamentally based on probability and has important implications for system design.

- **Randomness is Intrinsic:** The "M/M" in M/M/1 signifies that both job arrivals and service times follow memoryless (exponential) distributions. This means:
  - Arrivals may occur in unpredictable bursts
  - Service times vary randomly around the mean
  - System state fluctuates stochastically
- **Behavior at Different Utilization Levels:**
  - **Low Utilization ($\rho \approx 0.2$):** Ample spare capacity means minimal queuing
  - **Moderate Utilization ($\rho \approx 0.8$):** Response times remain manageable but with increased sensitivity to random fluctuations
  - **High Utilization ($\rho \geqslant 0.8$):** The "skyrocket effect" occurs - temporary arrival bursts or longer service times can cause queue lengths to explode
- **Real-World Example: TCP Congestion Collapse** This mathematical insight explains the phenomenon observed in network systems:
  1. **High Network Utilization:** Router or link operating near capacity
  2. **Packet Loss:** Random bursts cause buffer overflows and dropped packets
  3. **TCP Retransmissions:** Lost packets trigger retransmissions
  4. **Feedback Loop:** Retransmissions increase load, causing more packet loss
  5. **Collapse:** The system enters a vicious cycle where throughput plummets despite high utilization

The key insight from queuing theory is that due to the probabilistic nature of arrivals and service times, wait times become unbounded as utilization approaches 100%. Therefore, practical systems must operate below maximum theoretical capacity to maintain reasonable response times.

Additional Scheduling Concepts

Open, Closed, and Combined Systems

System models significantly influence scheduling approaches and performance metrics. Understanding these models is crucial for designing effective schedulers.

Open vs. Closed System Models

- **Open System Model:**
  - Has an infinite population of potential jobs that arrive from external sources
  - Jobs enter the system, receive service, and then depart

- Characterized by fluctuating workloads and unpredictable arrival patterns
- Typically operates at less than 100% load
- Optimized for **response time**
- Examples: Web servers, customer-facing applications

- **Closed System Model:**
  - Contains a fixed number of jobs that cycle through the system
  - When a job completes, it re-enters the system as a new job (feedback loop)
  - No external arrivals or final departures
  - Typically operates at 100% load
  - Optimized for **throughput**
  - Examples: Batch processing systems, controllers

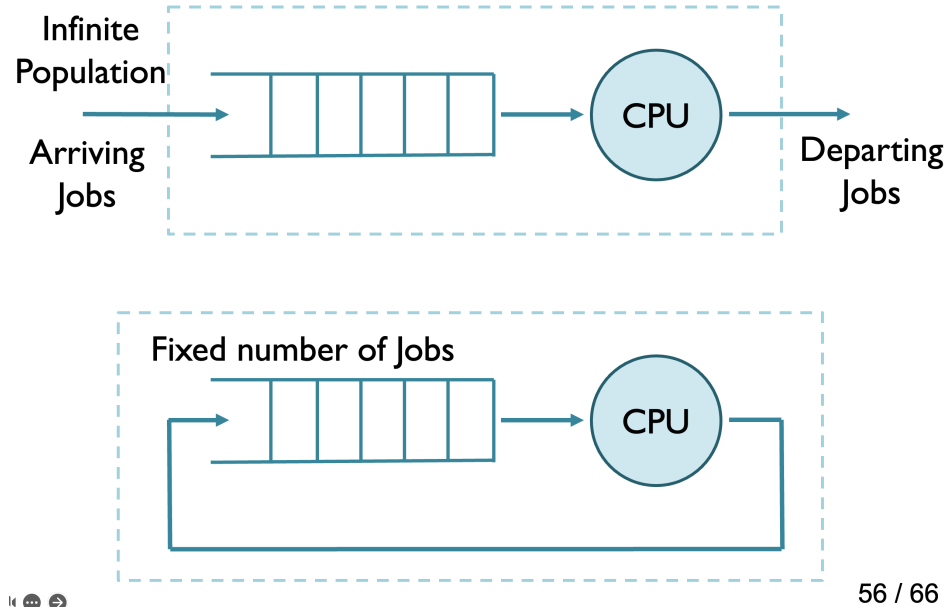

Figure 17: Open Vs. Closed System Models

Combined System Model

Real-world operating systems typically implement a combined model that incorporates elements of both open and closed systems:

- External users arrive and depart (open system aspect)
- Processes cycle between different resources within the system (closed system aspect)
- Multiple resources (CPU, disk, terminal) with associated queues

- Complex interactions between different system components

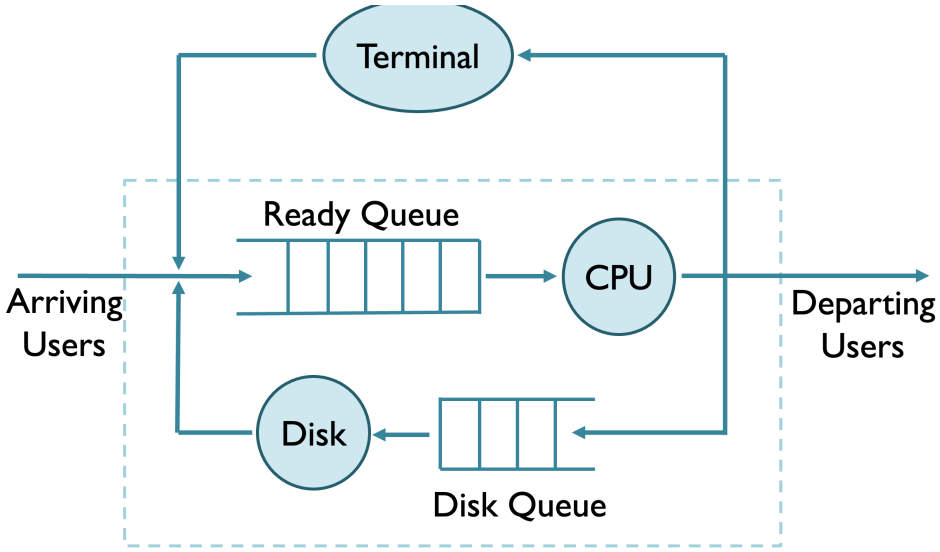

Figure 18: Combined Model

In this combined model, processes move between different states:

- Terminal wait state (user thinking or typing)
- Ready queue (waiting for CPU)
- CPU execution
- Disk queue (waiting for I/O)
- Disk I/O operation

Bottlenecks in System Performance

A key insight for efficient scheduling is understanding bottlenecks in the system:

- **Definition:** In a queueing network, performance is primarily dictated by the bottleneck device—the resource with the highest utilization relative to its capacity.
- **Bottleneck Characteristics:**
  - If jobs are predominantly **compute-bound**, the CPU becomes the bottleneck
    - CPU utilization approaches 100%
    - Disk remains mostly idle
  - If jobs are predominantly **I/O-bound**, the disk becomes the bottleneck
    - Disk utilization approaches 100%

    - CPU remains mostly idle

- **Scheduling Implication:** Only the scheduling of the bottleneck device significantly impacts overall system performance. Optimizing the scheduling of non-bottleneck resources yields minimal benefits.

Long-Term Scheduling and Memory Management

While short-term scheduling focuses on CPU allocation, long-term scheduling addresses broader resource constraints, particularly memory:

- **Memory Constraints:**
  - Processes require memory space to run
  - Physical memory is limited and may not accommodate all processes simultaneously
- **Swapping Mechanism:**
  - When memory is insufficient, some processes must be **swapped out**
  - Swapped-out processes are temporarily stored on disk
  - Processes are later **swapped in** when memory becomes available
- **Key Considerations:**
  - **Prioritize interactive jobs:** Keep them in memory to maintain system responsiveness
  - **Create a balanced job mix:** Ensure complementary resource usage

The goal of long-term scheduling is to maintain an optimal set of processes in memory to maximize resource utilization while ensuring responsive system operation.

Job/Process Characterization

Understanding the behavior patterns of different process types is essential for effective scheduling:

- **CPU-Bound Processes:**
  - Spend most of their time performing computations
  - Have few but very long CPU bursts
  - Examples: Scientific calculations, simulations, rendering
  - Characterized by high CPU utilization and minimal I/O operations
- **I/O-Bound Processes:**

- Spend more time doing I/O than computations
- Have many short CPU bursts interspersed with I/O requests
- Examples: Database operations, text editors, web browsers
- Characterized by frequent I/O operations with minimal CPU requirements

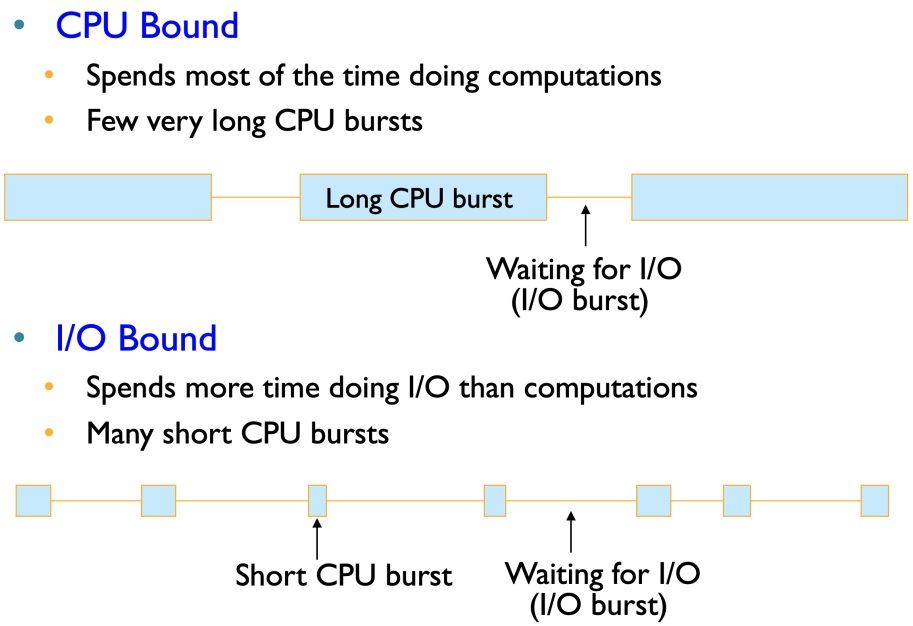

Figure 19: CPU-Bound Vs. I/O-Bound

Optimizing the Job Mix

A diverse job mix can lead to more efficient resource utilization:

- **Resource Utilization Challenge:**
  - Systems have multiple resources (CPU, disk, memory, network)
  - Any single resource can become a bottleneck
  - When jobs wait for the bottleneck resource, other resources may be underutilized
- **Benefits of a Balanced Job Mix:**
  - **Complementary resource usage:** CPU-bound jobs utilize the CPU while I/O-bound jobs utilize I/O devices
  - **Higher throughput:** More work completed per unit time
  - **Better resource utilization:** Fewer idle resources
  - **Improved system responsiveness:** Interactive jobs remain responsive

An optimal job mix allows the system to approach full utilization of all resources simultaneously, avoiding single-resource bottlenecks.

Fair Share Scheduling

Fairness in resource allocation is a common goal in multi-user systems:

- **Fairness Principles:**
  - "Fair" does not necessarily mean "equal"
  - Fairness is determined by allocation policies (often a "political decision")
  - Different users or groups may be entitled to different resource proportions
- **Implementation Approaches:**
  - **Virtual Time Scheduling:**
    - Count time "faster" for low-priority processes
    - Apply virtual time "penalties" per quantum for processes that exceed their allocation
  - **Lottery Scheduling:**
    - Distribute lottery tickets to processes based on their resource allocation
    - Randomly select a ticket when scheduling decisions are needed
    - The process holding the selected ticket receives the resource
    - Processes with more tickets have proportionally higher chances of being selected

Fair share scheduling ensures that resource distribution aligns with organizational policies while preventing any single user or process group from monopolizing system resources.