

# Resource Management and Scheduling

---

## Single-Server Models

To understand how to manage resources across a massive data center, we first need to understand the fundamentals of performance on a single machine.

This section explores the classic queueing theory models that describe how tasks are processed by a **single server**.

### Preliminary Results: Queueing Theory Fundamentals

The performance of any system where tasks arrive, wait for service, and then depart can be analyzed using **queueing theory**.

---

#### Little's Law

A foundational result in queueing theory is **Little's Law**.

- It provides **relationship** between the *average number of customers* in a stable system, their *arrival rate*, and the *average time* they spend in the system.

Given a system at statistical equilibrium, a system that has been running long enough to have settled into a **stable, predictable pattern of behavior**, the law states:

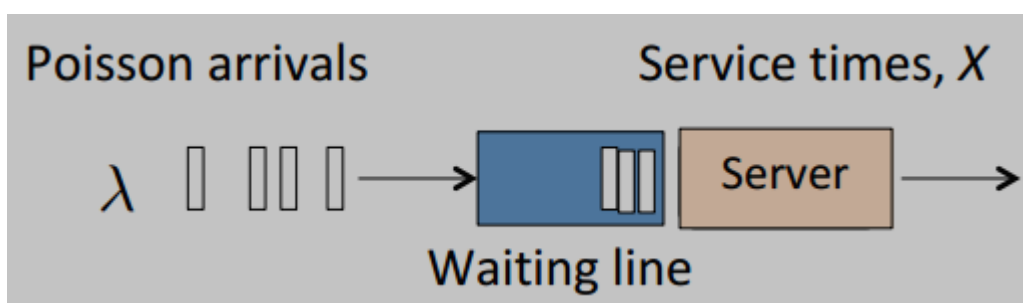
$$E[Q] = \lambda E[S]$$

Where:

- $E[Q]$  is the **average number of customers** (or tasks) in the system.
  - $\lambda$  is the **mean arrival rate** of customers into the system.
  - $E[S]$  is the **average time** a customer spends in the system (waiting time + service time).
- 

#### The M/G/1 Queue Model

To understand how to manage tasks on a single machine, we start with a fundamental concept from performance analysis: the **single-server queueing system**. The diagram on slide 157 illustrates this model, which serves as the theoretical foundation for this entire section of the course.



This diagram shows a standard queueing system with three main components:

- **Poisson Arrivals ( $\lambda$ ):** This represents the **flow of incoming tasks**.
  - The term "**Poisson**" signifies that arrivals are random and independent, but they occur at a predictable average rate, denoted by the Greek letter  $\lambda$  (lambda).
  - This corresponds to the "**M**" in the **M/G/1** model notation.
- **Waiting Line (The Queue):** This is the buffer where tasks wait if the server is busy.
  - The time tasks spend here is the primary source of queuing delay.
- **The Server and Service Time ( $X$ ):** This is the single resource that performs the work. It can only handle one task at a time.
  - The time it takes to complete a task is the **Service Time**, represented by the random variable  $X$ .
  - The model is general, meaning the service time can follow any probability distribution, which corresponds to the "**G**" in **M/G/1**.
  - The "**1**" signifies that there is a single server.

### A Building Block for All Single-Server Analysis

This M/G/1 model is not just an introductory example; it is the **basic building block** upon which all the following analyses of single-server performance are built. Every subsequent topic in this section refers back to this fundamental structure:

- **Waiting Time Formulas:** The detailed formulas for mean waiting time ( $E[W]$ ) derived in the following slides are specifically for this M/G/1 system.
- **Resource Pooling Comparison:** The analysis of whether to use one fast server or many slow ones compares different specific instances of this model (e.g., M/M/1 and M/M/m).
- **Scheduling Policies:** All the different scheduling algorithms you will learn about—**FCFS**, **SJF**, **SRPT**, **Processor Sharing (PS)**, and others—are simply different strategies for managing the "Waiting Line" and deciding the order in which the "Server" processes tasks within this exact model.

### Mean Waiting Time of the M/G/1 Queue

For a single-server system using a **First-Come, First-Served (FCFS)** policy, the time an arriving task must wait before its own service begins ( $W$ ) is the sum of two components:

1. The **residual service time** ( $\tilde{X}_0$ ) of the task currently being served (if any).
2. The **full service times** ( $X_j$ ) of all the  $N$  tasks already ahead of it in the queue.

This gives the formula:

$$W = \tilde{X}_0 + \sum_{j=1}^N X_j$$

From this, we can derive the formula for the *mean* waiting time, known as the **Pollaczek-Khinchine formula**, in a few clear steps.

### Step 1: Taking the Average

We start by taking the average (the expected value,  $E[\dots]$ ) of the expression for  $W$ . Using the principles of **Linearity of Expectation** and **Wald's Identity**, we get:

$$E[W] = E[\tilde{X}_0] + E[N]E[X]$$

We can write this more simply by defining  $W_0 = E[\tilde{X}_0]$ , which gives us our intermediate equation:

☑ \quad \quad \text{(Equation 1)}

This equation gives us the mean waiting time, but it depends on **another unknown**:  $E[N]$ , the **average number of tasks** in the queue.

### Step 2: Applying Little's Law to Find $E[N]$

To solve for  $E[N]$ , we now apply **Little's Law** ( $E[Q] = \lambda E[S]$ ) specifically to the **waiting line itself**.

- The "system" is the queue.
- The average number of items in this system is  $E[N]$ .
- The arrival rate into this system is  $\lambda$ .
- The average time an item spends in this system is, by definition, the mean waiting time,  $E[W]$ .

Applying Little's Law ( $E[Q] = \lambda E[S]$ ) to these terms gives us:

$$E[N] = \lambda E[W] \quad \text{(Equation 2)}$$

### Step 3: Substituting and Solving for $E[W]$

Now we can substitute the result from Little's Law (**Equation 2**) back into our intermediate equation (**Equation 1**):

$$E[W] = W_0 + (\lambda E[W])E[X]$$

We can rearrange the terms and recall the **definition of server utilization**,  $\rho = \lambda E[X]$ :

$$E[W] = W_0 + (\lambda E[X])E[W]$$

$$E[W] = W_0 + \rho E[W]$$

Finally, we solve this equation for  $E[W]$  through simple algebra:

$$E[W] - \rho E[W] = W_0$$

$$E[W](1 - \rho) = W_0$$

$$E[W] = \frac{W_0}{1 - \rho}$$

This is the final **Pollaczek-Khinchine formula** for the mean waiting time in an M/G/1 queue, where:

- $\rho = \lambda E[X]$  is the **utilization coefficient** of the server (*the fraction of time it is busy*).
- $W_0 = E[\tilde{X}_0]$  is the **mean residual service time**, which is the average time an arriving customer has to wait for the *current* service to finish. It is given by:

$$W_0 = \frac{\lambda E[X^2]}{2}$$

This formula is crucial because it shows that the mean waiting time depends not only on the arrival rate and mean service time ( $E[X]$ ) but also on the *second moment* of the service time ( $E[X^2]$ ), which is related to its **variability**. Higher service time variability leads to longer average waits.

### Deriving the Mean Residual Service Time ( $W_0$ )

To find the value of  $W_0$ , we need to consider the **two possibilities** an arriving task might encounter. We can model this using a binary random variable,  $B$ :

- $B = 1$ : The server is **busy**. The probability of this is the server utilization,  $P(B = 1) = \rho$ .
- $B = 0$ : The server is **idle**. The probability of this is  $P(B = 0) = 1 - \rho$ .

The average residual service time,  $W_0$ , is the **weighted average of the outcomes** in these two states:

1. **If the server is idle ( $B = 0$ ):** An arriving task finds an empty server and can begin service **immediately**.
  - The residual service time is **0**.
2. **If the server is busy ( $B = 1$ ):** An arriving task **must wait** for the current task to finish.
  - Due to a property of Poisson arrivals known as the *inspection paradox*, the average remaining service time for the task in progress is given by a specific formula from renewal theory:

$$E[\text{remaining time} | \text{server busy}] = \frac{E[X^2]}{2E[X]}$$

Combining these using the law of total expectation:

$$W_0 = (P(B = 0) \times 0) + (P(B = 1) \times \frac{E[X^2]}{2E[X]})$$

$$W_0 = ((1 - \rho) \times 0) + (\rho \times \frac{E[X^2]}{2E[X]})$$

$$W_0 = \frac{\rho E[X^2]}{2E[X]}$$

Finally, by substituting the definition of utilization ( $\rho = \lambda E[X]$ ), we get the form used in the final calculation:

$$W_0 = \frac{(\lambda E[X]) E[X^2]}{2E[X]} = \frac{\lambda E[X^2]}{2}$$

### Resource Separation vs. Pooling

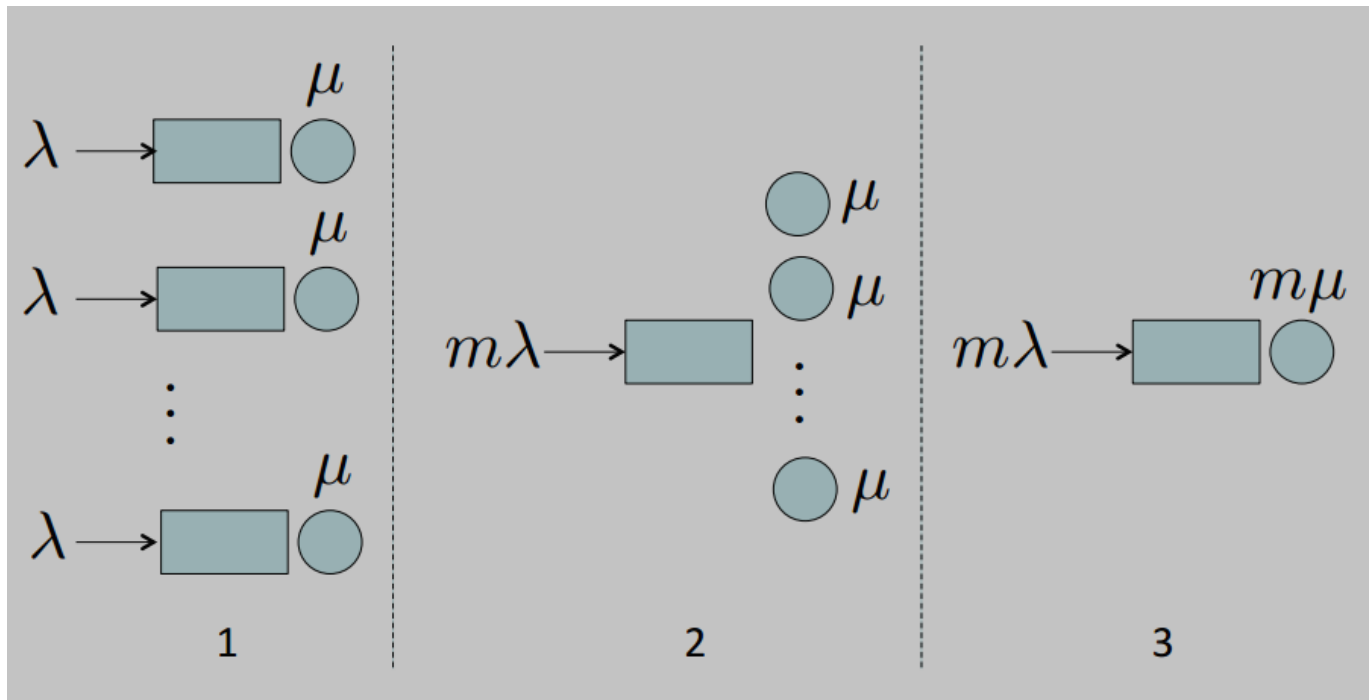
This queueing theory foundation allows us to answer a critical design question: **for a given total processing capacity, is it better to have one single, powerful server or to separate the capacity into multiple,**

**slower servers?**

Let's consider a system with:

- **Total arrival rate** of  $m\lambda$
- **Total service capacity** of  $m\mu$ .

We can arrange this in **three ways**:



1. **Multi-Queue (Separated):**  $m$  separate servers, each with service rate  $\mu$  and handling an arrival rate of  $\lambda$ . Each server is an **independent M/M/1 queue**.

- **Mean System Time:**  $E[S] = \frac{1}{\mu - \lambda}$

2. **Multi-Server (Pooled):**  $m$  servers with rate  $\mu$  sharing a single queue. This is an **M/M/m queue**.

- **Mean System Time:**  $E[S] = \frac{C(m, m\lambda/\mu)}{m\mu - m\lambda} + \frac{1}{\mu}$

Where  $C(m, A)$  is the **Erlang C formula**, which gives the **probability** that an **arriving customer has to wait in the queue**.

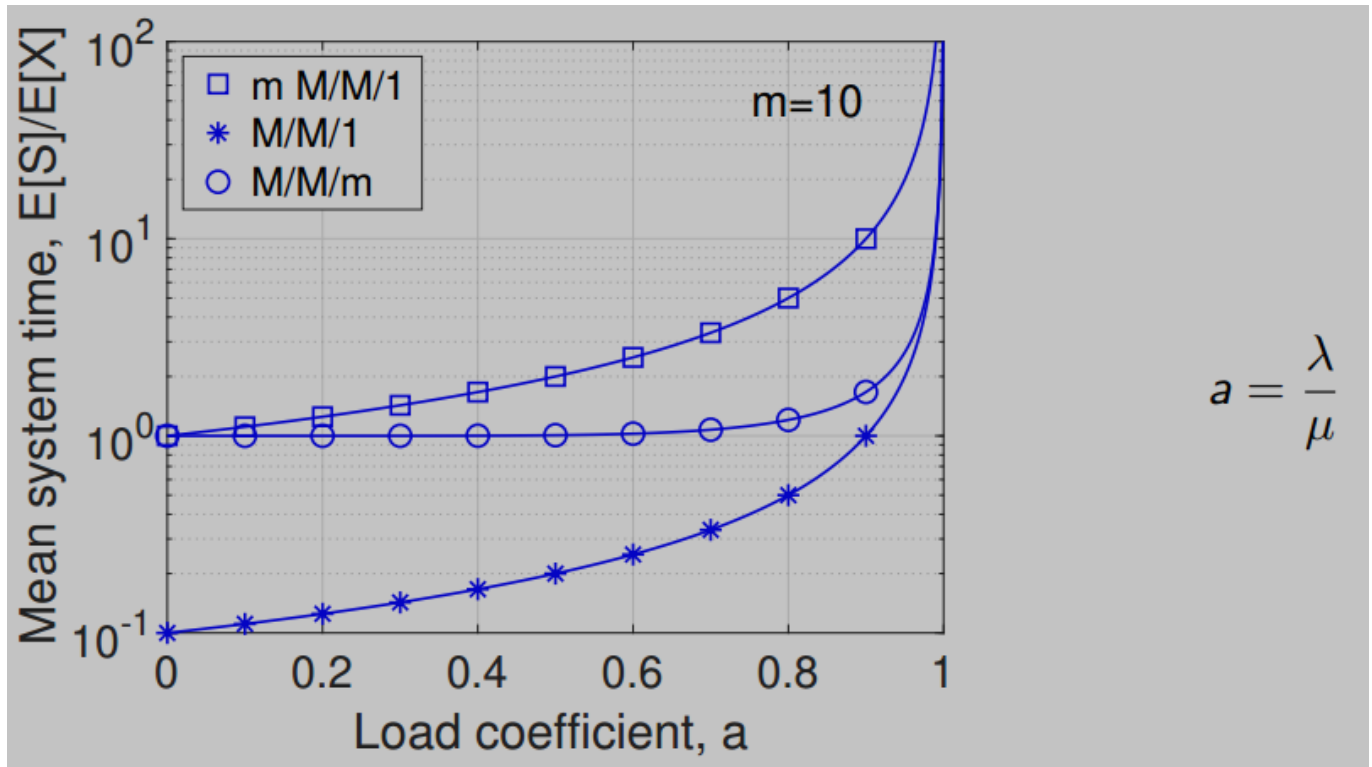
3. **Single-Server (Super-Server):** One single, fast server with a service rate of  $m\mu$  handling the total arrival rate of  $m\lambda$ . This is a **single M/M/1 queue**.

- **Mean System Time:**  $E[S] = \frac{1}{m\mu - m\lambda}$

**Numerical Comparison and Conclusion**

The graph on the slide compares the performance of these three models for  $m=10$  servers as the load ( $a = \lambda/\mu$ ) increases.

- The **squared line** ( $m$  M/M/1) represents the separated multi-queue system.
  - Its performance is the **worst**.
- The **circle line** (M/M/ $m$ ) represents the pooled multi-server system.
  - It performs **significantly better**.
- The **star line** (M/M/1) represents the single "super-server".
  - It provides the **best performance** of all.



**The conclusion is clear: pooling resources is always better.**

A single, faster server will always provide lower average response times than multiple, separate slower servers with the same total capacity.

- This is a fundamental reason why cloud computing, which pools massive resources together, is **so effective**.

## An Introduction to Scheduling Policies and Terminology

A **scheduling algorithm** is a policy for managing the capacity of a single server. It defines the order in which tasks waiting in the queue are served.

To compare different scheduling strategies, we first need a framework for classifying them. Scheduling policies can be categorized along three key dimensions: **Sharing**, **Preemption**, and knowledge of **Job Size**.

### Priority Terminology

- **Sharing**
  - **Head-of-Line (HOL):** The server is dedicated to at most a **single customer** at a time.
    - It completes one task (or a part of it) before even considering the next.
    - This is a "**one-at-a-time**" service model.

- **Server Sharing:** The server's capacity is conceptually "sliced" and **shared among multiple customers** simultaneously.
  - More than one customer can be partially served at any given time.
- **Preemption**
  - **Non-preemptive Priority:** Once a task begins service, it **runs to completion and cannot be interrupted**, even if a higher-priority task arrives.
  - **Preemptive Priority:** A task currently being served **can be interrupted (*preempted*)** if a task with a higher priority arrives.
    - The interrupted task is typically returned to the queue to resume later.
- **Job Size**
  - **Non-anticipative:** The scheduling algorithm has **no prior knowledge** of how long a task will take to complete (its "job size"). It can only react to a job's properties as it observes them.
  - **Anticipative:** The scheduling algorithm **knows the size** of each job upon its arrival and can use this information to make smarter scheduling decisions.

## Overview of Scheduling Policies

Using the terminology above, the slides will cover the following key scheduling strategies:

- **Head-Of-Line (HOL) Strategies:**
  - **Non-preemptive, Non-anticipative:**
    - *First-Come, First-Served* (FCFS)
    - *Static Priority*
  - **Non-preemptive, Anticipative:**
    - *Shortest Job First* (SJF)
  - **Preemptive, Non-anticipative:**
    - *Least Attained Service* (LAS)
  - **Preemptive, Anticipative:**
    - *Shortest Remaining Processing Time* (SRPT)
- **Server Sharing Policies:**
  - **Processor Sharing (PS):** The theoretical model of perfectly fair sharing.
    - *Practical Implementation:* **Round Robin (RR)**.
  - **Generalized Processor Sharing (GPS):** The theoretical model for weighted fair sharing.
    - *Practical Implementation:* **Credit-Based Scheduling**.

## Notation for Analysis

To analyze these policies mathematically, we will use the following standard notation for a single-server system fed by  $P$  different classes of customers:

- **Priority:** Class  $j$  has priority over class  $i$  for any  $j < i$ . (Class 1 is the highest priority).
- **Arrivals:**
  - $T_j$ : The *inter-arrival time* for tasks of class  $j$ .

- We assume arrivals follow a **Poisson process**, unless otherwise stated.
- $\lambda_j$ : The **mean arrival rate** of class  $j$ .
- **Service Times:**
  - $X_j$ : The **service time** for a task of class  $j$ , which can follow a general probability distribution with CDF  $F_{X_j}(t)$ .
  - $E[X_j] = 1/\mu_j$ : The **mean service time** for class  $j$ .
  - $E[X_j^2] = \sigma_{X_j}^2 + 1/\mu_j^2$ : The **second moment of the service time** for class  $j$ .
- **Utilization (Load):**
  - $\rho_j = \lambda_j E[X_j]$ : The **utilization coefficient** for class  $j$ 
    - It is the fraction of the server's time spent on class  $j$  tasks.
  - $\rho = \rho_1 + \dots + \rho_P$ : The total utilization of the server. For the system to be stable, we must have  $\rho < 1$ .

## The Conservation Law for HOL Systems

Before analyzing specific policies, it's important to understand a fundamental principle that governs all **work-conserving**, Head-of-Line (HOL) systems.

- A system is **work-conserving** if the server is never idle when there are tasks in the queue.

**Theorem (Conservation Law):** For any work-conserving HOL priority system **in equilibrium**, the weighted sum of the mean waiting times across all customer classes is constant and equal to the mean waiting time of a simple FCFS system.

$$\sum_{j=1}^P \frac{\rho_j}{\rho} E[W_j] = \frac{W_0}{1 - \rho} = E[W_{FCFS}]$$

Where:

- $\rho_j$  is the **utilization** from class  $j$ .
- $\rho$  is the **total server utilization**.
- $E[W_j]$  is the **mean waiting time** for class  $j$ .
- $W_0$  is the **mean residual service time** for the system as a whole.

**Key takeaway:** You cannot improve the waiting time for one class of tasks without worsening the waiting time for another.

- Any scheduling policy is simply a **trade-off**, redistributing the total waiting time among the different classes.

## Static Non-Preemptive HOL Priorities

This is one of the simplest priority scheduling schemes.

- **Policy:** Arriving tasks are assigned a **fixed priority "label"**
  - *Example:* from 1 to  $P$ , where 1 is the highest priority.
  - When the server becomes free, it serves the task with the highest priority from the queue.



- Because it is **non-preemptive**, once a task begins service, it runs to completion, even if a higher-priority task arrives.
- **Mean Waiting Time:** If the **arrival are Poisson**, the mean waiting time for a task of class  $j$  is given by:

$$E[W_j] = \frac{W_0}{(1 - \sum_{i=1}^{j-1} \rho_i)(1 - \sum_{i=1}^j \rho_i)}$$

- **Key Implication:** Note that the mean residual service time,  $W_0 = \frac{1}{2} \sum_{i=1}^P \lambda_i E[X_i^2]$ , is the same for all classes.
  - This means that a long, high-variance job from a *low-priority* class can significantly delay all *high-priority* jobs if it happens to occupy the server when they arrive.
- **Optimal Priority Assignment:** If there is a cost  $c_j$  associated with each unit of time a class  $j$  task waits, the total cost is minimized by assigning priorities in decreasing order of the ratio  $c_j / E[X_j]$ .
  - This is known as the **cm rule**: give higher priority to tasks that are "cheaper" to serve
  - Have a **small** mean service time ( $E[X_j]$ ) and/or have a **high** cost of waiting ( $c_j$ ).

## Example

Consider two flows of jobs with  $\lambda_1 = \lambda_2$  and where type-2 jobs take 10 times longer on average ( $E[X_2] = 10E[X_1]$ ). The total load is  $\rho = 0.9$ .

- This implies  $\rho_1 \approx 0.082$  and  $\rho_2 \approx 0.818$ .
- Giving class 1 (the shorter jobs) higher priority results in:
  - $E[W_1] \approx 1.089W_0$
  - $E[W_2] \approx 10.89W_0$
- In a standard FCFS system, the overall mean wait would be  $E[W_{FCFS}] = 10W_0$ .
- **Key Takeaway:** The priority scheme significantly **improves performance** for the high-priority short jobs, at a **slight cost** to the low-priority long jobs.

## Shortest Job First (SJF)

SJF is an **anticipative** policy that aims to minimize overall waiting time by **prioritizing shorter jobs**.

- **Policy: Non-preemptive** priority policy where a job's priority is determined by its **service time**.
  - When a job of size  $x$  arrives, it is placed in the queue ahead of any waiting jobs with a service time greater than  $x$ .
- **Conditional Mean Waiting Time:** The mean waiting time for a job, given that it requires  $x$  amount of service, is:

$$W(x) = E[W|X = x] = \frac{W_0}{(1 - \lambda \int_0^x t dF_X(t))^2}$$

- Where the **integral term** represents the portion of the server load contributed by jobs smaller than  $x$ .
- **Performance:** SJF always results in a **lower overall average waiting time** than FCFS:  $E[W_{SJF}] < E[W_{FCFS}]$ .

## Least Attained Service (LAS)

What if you want to prioritize short jobs but don't know their size in advance? **Least Attained Service (LAS)**, also known as **Foreground-Background (FB)**, is a clever **non-anticipative** policy that **approximates SJF**.

- **Policy:** The server always gives **priority** to the task that has so far received the **least amount of service**.
- **Intuition:** The "age" or "attained service" of a job is used as a proxy for its total size.
  - A job that has only received a **small amount of service** is likely to be a short job, so it is **given priority**.
  - A job that has already run for a **long time** is likely a long job and is **de-prioritized**.
- **Implementation:** This is often implemented with a **multi-level feedback queue**.
  - **New jobs** enter the **highest-priority queue**.
  - If a job uses up its time slice in that queue **without finishing**, it is **moved to a lower-priority queue**.
- **Conditional Response Time:** The response time for a job of size  $x$  is:

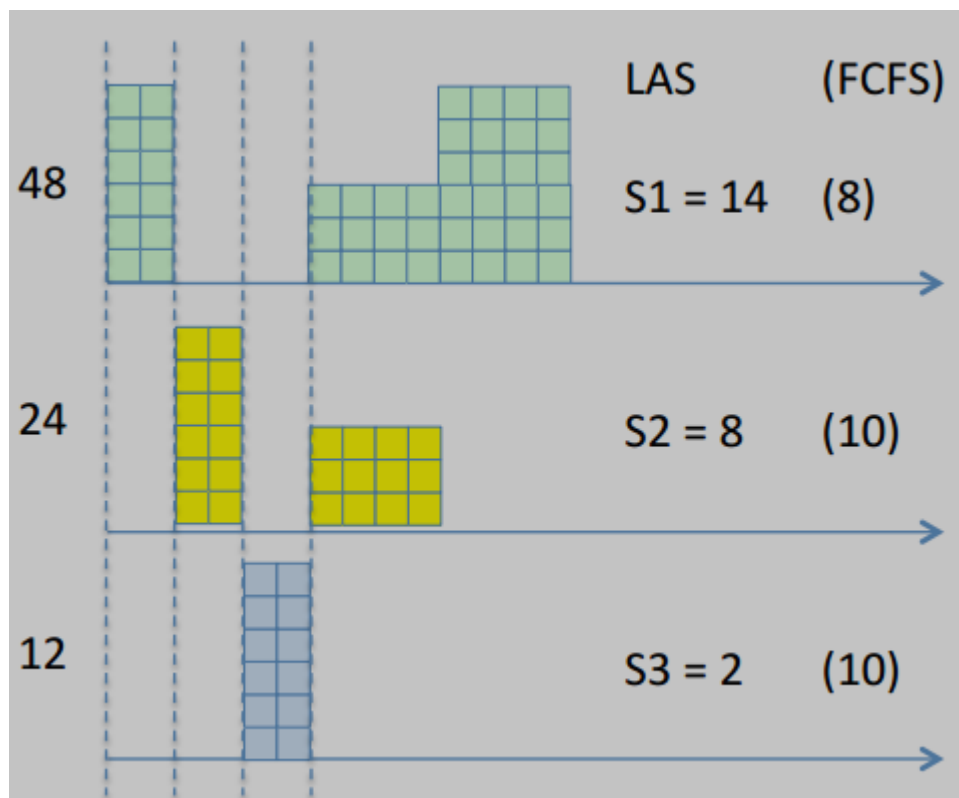
$$S(x) = \frac{\lambda m_2(x)}{2[1 - \lambda m_1(x)]^2} + \frac{x}{1 - \lambda m_1(x)}$$

Where  $m_h(x)$  is the  $h$ -th moment of the service time, truncated at  $x$ .

### LAS Numerical Examples

To better understand how the Least Attained Service (LAS) policy works in practice, we can look at two complementary examples: a step-by-step trace and a graph of its general performance.

#### Example 1: A Step-by-Step Trace (Slide 179)

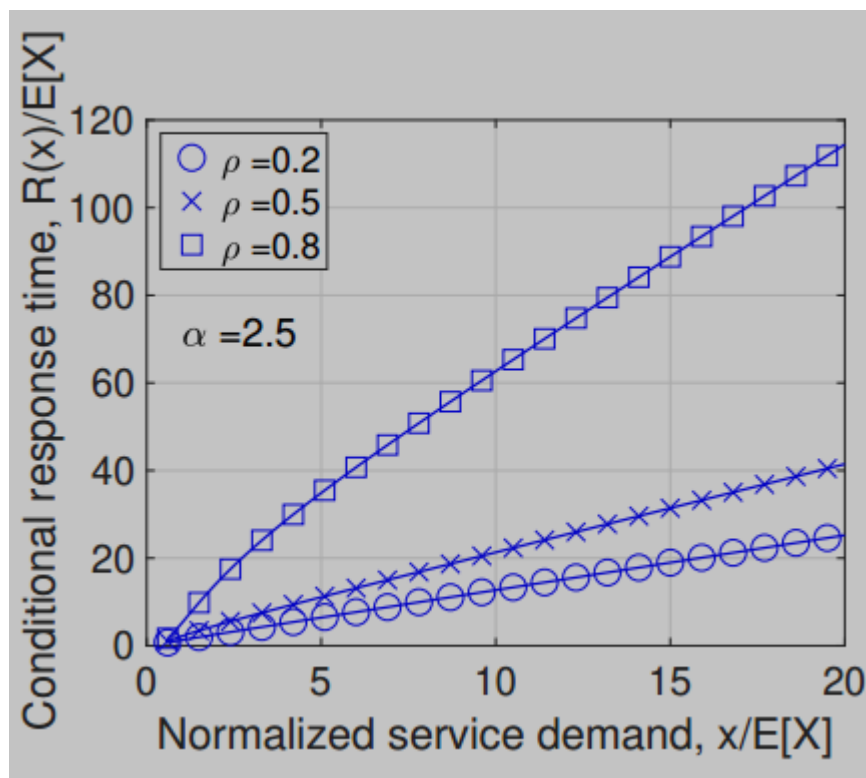


This example traces the execution of three jobs with different service times arriving at different moments, showing how LAS dynamically changes which job is being served.

- **Scenario:**
  - Job 3 (service time = 2) arrives at time 12.
  - Job 2 (service time = 8) arrives at time 24.
  - Job 1 (service time = 4) arrives at time 48.
- **Execution Trace:**
  1. At time 12, Job 3 arrives and starts running.
  2. At time 24, Job 2 arrives. Since Job 2 has received **zero** service so far, it has the "least attained service," and the server preempts Job 3 to start working on Job 2.
  3. At time 48, Job 1 arrives. It also has zero attained service, so it has priority. The server preempts Job 2 and starts working on Job 1.
  4. The server will continue to work on whichever active job has the least accumulated service time, switching between them until they are all complete.
- **Result vs. FCFS:**
  - **With LAS:** The completion times are  $S_1 = 14$ ,  $S_2 = 8$ ,  $S_3 = 2$ . The policy manages to finish the two smaller jobs (2 and 3) very quickly.
  - **With FCFS:** The completion times would have been  $S_1 = 8$ ,  $S_2 = 10$ ,  $S_3 = 10$ . The jobs would be served strictly in their arrival order.

This trace clearly shows how LAS reorders execution to prioritize jobs that are presumed to be short, even if they arrive later.

### Example 2: General Performance Characteristics (Slide 180)



This graph illustrates the general performance of LAS by plotting the conditional response time as a function of the job's size, for different system loads ( $\rho$ ).

- **X-axis:** The normalized service demand ( $x/E[X]$ ), representing how large a job is compared to the average job size.
- **Y-axis:** The conditional response time ( $R(x)/E[X]$ ), representing how long a job of a certain size takes to complete, normalized by the average service time.
- **Interpretation:**
  - **Favors Short Jobs:** For small job sizes (left side of the graph), the response time is very low.
    - LAS is extremely efficient for short tasks.
  - **Penalizes Long Jobs:** For large job sizes (right side of the graph), the response time grows dramatically.
    - LAS will constantly preempt long jobs to service newly arriving short jobs, so large jobs can take a very long time to complete.
  - **Impact of Load:** This effect becomes much more extreme as the system load ( $\rho$ ) increases.
    - At high loads (e.g.,  $\rho = 0.8$ ), the penalty for long jobs is severe.

**Conclusion from both examples:** Together, these examples demonstrate both the *mechanism* of LAS (always prioritizing the job with the least completed work) and its *performance consequence* (excellent response times for short jobs at the expense of very long delays for large ones).

## Shortest Remaining Processing Time (SRPT)

SRPT is the **preemptive** version of SJF and is the gold standard for minimizing delay.

- **Policy:** The server always works on the job with the **shortest remaining processing time**.
  - If a new job arrives whose *total* size is less than the *remaining* time of the job currently in service, the new job preempts the running job.
- **Optimality:** It is proven that SRPT **minimizes the mean system time** ( $E[S]$ ) for a single-server queue among all HOL policies.
- **Performance Ranking:** The mean system times for these policies are ordered as follows:

$$E[S_{SRPT}] \leq E[S_{SJF}] \leq E[S_{FCFS}]$$

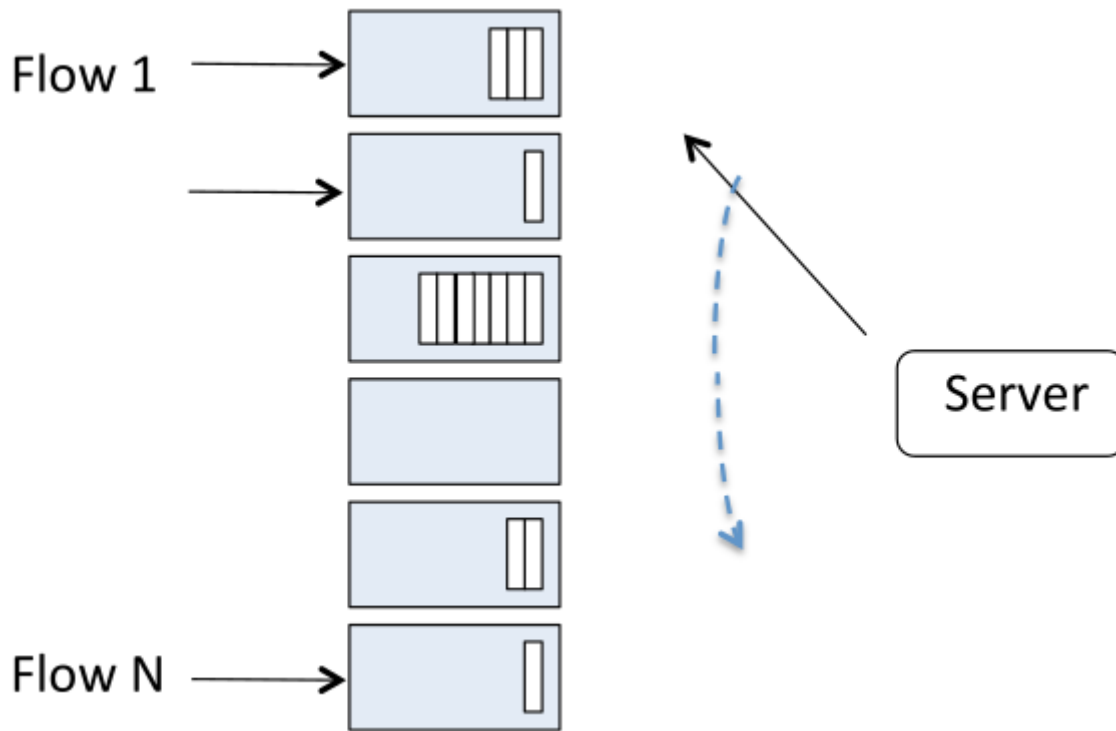
## Server Sharing Policies

Unlike Head-of-Line (HOL) policies that serve one task at a time, server sharing policies conceptually **divide the server's capacity among multiple tasks simultaneously**.

The **primary motivation** for this approach is to provide **isolation**, which serves two main purposes:

- To **protect** well-behaved applications from being starved by aggressive, resource-hungry flows.
- To provide **differentiated** and **guaranteed service** levels to different classes of users.

Processor Sharing (PS): The Ideal of Perfect Fair Sharing



**Processor Sharing (PS)** is a theoretical model that represents the ideal form of perfect fairness.

- **The Concept:** If there are  $n$  tasks in the queue at any given moment, the PS model assumes that the server's capacity (*mean service rate*:  $\mu$ ) is instantly and perfectly divided, with **each task receiving exactly  $1/n$  of the total capacity**. If a new task arrives, the shares are instantaneously recalculated.
- **Performance:** The mean system time (wait + service) in a PS system is given by the simple formula:

$$E[S] = \frac{E[X]}{1 - \rho}$$

A key feature of PS is that its performance depends only on the mean service time ( $E[X]$ ), making it **insensitive to service time variability**. This is a major advantage over FCFS, whose performance degrades significantly as service time variance increases.

- **Practical Implementation - Round-Robin (RR):** In the real world, a CPU cannot truly work on multiple tasks at the exact same instant. **Round-Robin (RR)** scheduling is the practical approximation of the PS ideal. The server gives a small, fixed-size time slice (called a *quantum*) to each task in the queue, cycling through them in a circular order. If the time quantum is very small compared to the job sizes, the effect is very close to perfect, simultaneous sharing.

### Generalized Processor Sharing (GPS): Weighted Fair Sharing

**Generalized Processor Sharing (GPS)** extends the concept of PS to allow for *differentiated* service rather than strictly equal service. This is the model for **weighted fair sharing**.

- **The Concept:** Instead of giving every task an equal  $1/n$  share, GPS assigns a **weight** (e.g.,  $\phi_i$ ) to each task or class of tasks. The amount of service a task receives is **proportional to its weight** relative to the sum of the weights of all other tasks currently in the system ( $B(t)$ ):

$$\text{Share for task } i = \mu \times \frac{\phi_i}{\sum_{j \in B(t)} \phi_j}$$

- **The Goal:** GPS is designed to provide predictable and controllable service differentiation. For instance, a cloud provider can offer different tiers of service by assigning different weights to customers.
- **Practical Implementation - Weighted Fair Queueing (WFQ) / Credit-Based Scheduling (CBS):** Just as RR implements PS, more advanced algorithms are needed to approximate GPS. **Weighted Fair Queueing (WFQ)** and **Credit-Based Scheduling (CBS)** are two such practical implementations that use various mechanisms to ensure that, over time, each task receives its designated proportional share of the server's capacity.

## Performance Comparison, Fairness, and the All-Can-Win Theorem

### Overall Policy Comparison

The graphs on slide 194 compare the mean response time of all the policies we've discussed. The key takeaway is that as service time variance increases, policies like **FCFS degrade dramatically**, while **PS is unaffected** and **SRPT remains the best performer**.

### The Fairness Dilemma

This leads to a critical dilemma:

- **SRPT** provides the best overall mean response time but seems "unfair" because it heavily prioritizes short jobs at the expense of long ones.
- **PS** is perfectly "fair" in the sense that all jobs experience the same slowdown factor, but its average performance is worse than SRPT's.

### The All-Can-Win Theorem

This apparent trade-off between performance and fairness is resolved by the **All-Can-Win Theorem**.

- **Theorem:** In an M/G/1 system, if the server load is sufficiently light (specifically, if  $\rho < 1/2$ ), then the response time under **SRPT is better than the response time under PS for all job sizes**.

$$R_{SRPT}(x) \leq R_{PS}(x), \quad \forall x \quad (\text{if } \rho < 1/2)$$

- **Intuition:** The main penalty for a large job under SRPT is waiting for small jobs to be served first. In a lightly loaded system, an arriving large job is likely to find the server idle, so its initial wait is small. Once it begins service, it gets the *full* server capacity, whereas under PS it would have to *share* that capacity. This makes its service phase much faster, more than compensating for any small initial wait.
- **Lesson Learned:** This theorem teaches a powerful lesson: sometimes, instead of designing complex scheduling policies, it's more effective to simply **ensure that servers are not too heavily loaded**. Keeping utilization below a certain threshold can unlock significant performance benefits and even resolve fairness concerns.

### Case Study: Applying SRPT to Web Server Farms

These theoretical concepts were applied in a real-world system to improve web server performance.

- **The Problem:** Traditional web servers use a PS-like scheduling policy to share the limited ISP bandwidth among thousands of concurrent client requests.
- **The SRPT Solution:** Researchers identified the "job" as a file transfer and the "job size" as the file's size. They modified the server's kernel to prioritize the TCP socket corresponding to the file with the **smallest remaining data to transmit**—a direct implementation of SRPT.
- **The Result:** The graph on slide 201 shows the outcome. The SRPT-based scheduler achieved a **significantly lower mean response time** compared to the standard PS scheduler across all load levels, demonstrating the practical power of size-based scheduling.