Operating System
Lecture 6 - Process Scheduling

Yonghao Lee

May 9, 2025

Introduction to Schedulers

Schedulers are fundamental components of operating systems that manage shared resources. They determine how and when processes gain access to system resources, particularly the CPU.

Types of Schedulers

Schedulers exist in various contexts wherever resources need to be shared:

- **CPU scheduler** (short-term scheduler): Manages CPU time allocation among ready processes
- **Mid/long-term scheduler**: Manages which processes reside in memory versus on disk
- **I/O scheduler**: Manages access to I/O devices like disks and printers
- **Network scheduler**: Manages packet transmission in networks
- **Web server scheduler**: Manages request handling in web servers

These scheduling concepts extend beyond computing to real-world scenarios like supermarkets and post offices.

CPU Scheduler's Role

In the process state diagram, the CPU scheduler specifically handles two crucial transitions:

- **Ready → Running**: Selecting the next process to execute when the CPU becomes available
- **Running → Ready**: Deciding when to preempt a running process (in preemptive systems)

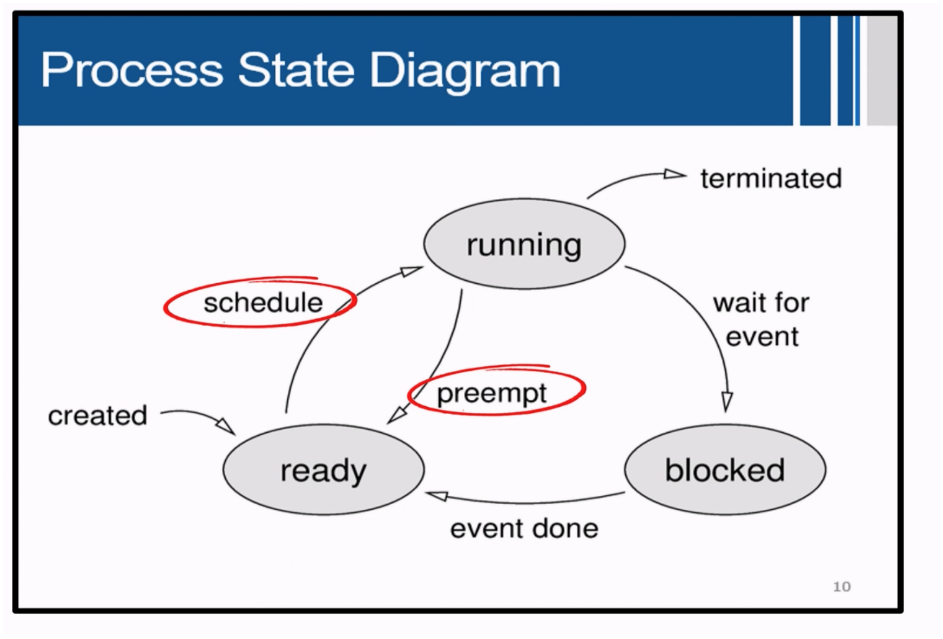


Figure 1: CPU Scheduler's Role in Process State Transitions

Key Terminology

Definition 1: *The **CPU scheduler** (short-term scheduler) is the component that selects which ready process receives CPU time next.*

Definition 2: *The **Dispatcher** is the module that implements the CPU scheduler's decisions by performing context switches, switching to user mode, and setting the program counter to the appropriate instruction.*

Process Lifecycle Metrics

Understanding the timing metrics of a process helps evaluate scheduling algorithms.

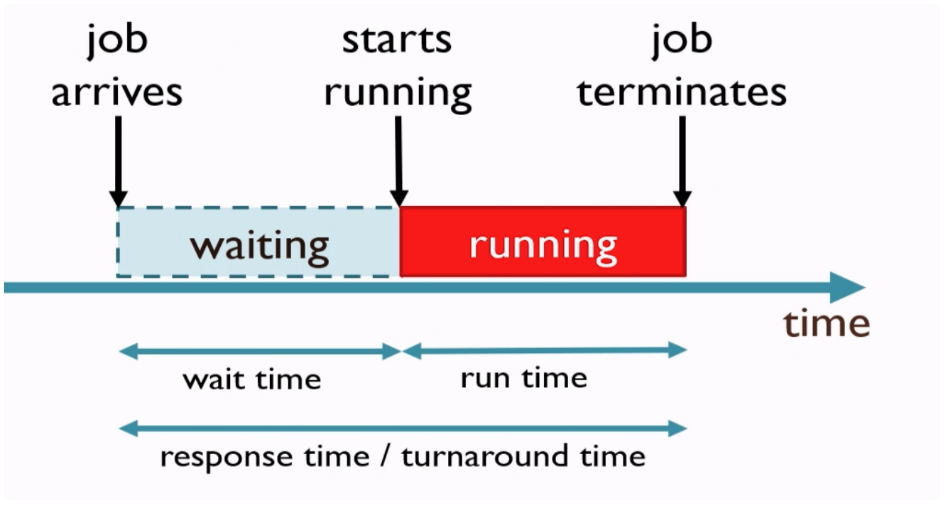


Figure 2: Process Timeline and Key Metrics

Key Time Metrics

Definition 3: ***Arrival Time*** *is when a process enters the system.*

Definition 4: ***Start Time*** *is when a process first begins execution on the CPU.*

Definition 5: ***Completion Time*** *is when a process finishes execution completely.*

Definition 6: ***Run Time*** *is the total time a process spends executing on the CPU.*

Definition 7: ***Wait Time*** *is the total time a process spends in the ready queue.*

Definition 8: ***Response Time*** *(or Turnaround Time) is the total elapsed time from process arrival until completion (Wait Time + Run Time).*

CPU Bursts and I/O Waits

Process execution typically follows a pattern of alternating CPU usage and I/O waits.

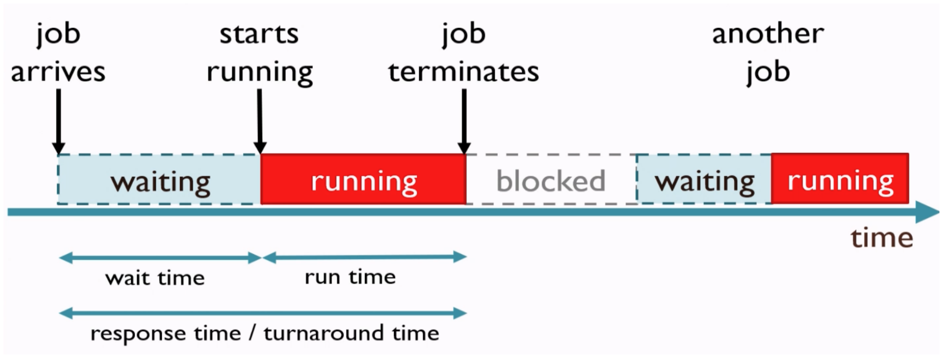


Figure 3: CPU Burst Patterns in Process Execution

Key Concept 1: *Most processes, especially interactive ones, alternate between:*

- ***CPU bursts****: Periods of active computation*
- ***I/O bursts****: Periods waiting for I/O operations to complete*

Example 1: *When using a web browser:*

- *Reading a page: The process waits for user input (I/O wait)*
- *Clicking a link: The browser processes the action (CPU burst)*
- *Loading the new page: The process waits for network data (I/O wait)*
- *Rendering the page: The browser formats and displays content (CPU burst)*

This pattern affects scheduling decisions, as schedulers often consider each CPU burst as a separate schedulable unit.

Scheduling Framework

Definition 9: *A scheduler is fundamentally an algorithm that takes a list of jobs as input and outputs decisions about which job to run when.*

Scheduling Objectives

Schedulers aim to optimize various performance metrics, often with competing priorities:

- **Low Response Time**: Minimize the total time from job arrival to completion
- **Low Wait Time**: Minimize time spent in the ready queue
- **Low Slowdown**: Minimize the ratio of response time to run time Slowdown $= \frac{\text{Response Time}}{\text{Run Time}}$
- **High Throughput**: Maximize the number of jobs completed per time unit
- **High CPU Utilization**: Maximize productive CPU usage
- **Fairness**: Ensure equitable resource distribution and prevent starvation

Note: *These objectives often conflict. For example, optimizing for throughput might suggest minimizing context switches by running jobs to completion, while optimizing response time might require frequent context switches to start new jobs promptly.*

Scheduling Information Models

The information available to the scheduler significantly impacts algorithm design and performance.

Off-line vs. On-line Scheduling

Definition 10: ***Off-line Scheduling*** *occurs when complete information about all jobs is known before any scheduling decisions are made.*

Definition 11: ***On-line Scheduling*** *occurs when the scheduler must make decisions without complete future information, learning about jobs dynamically as they arrive.*

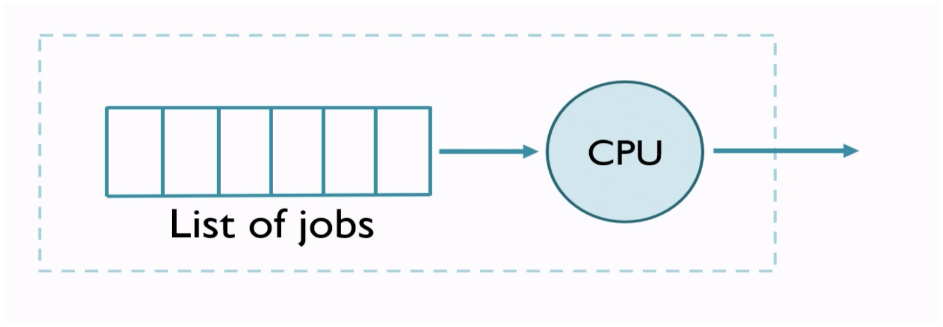
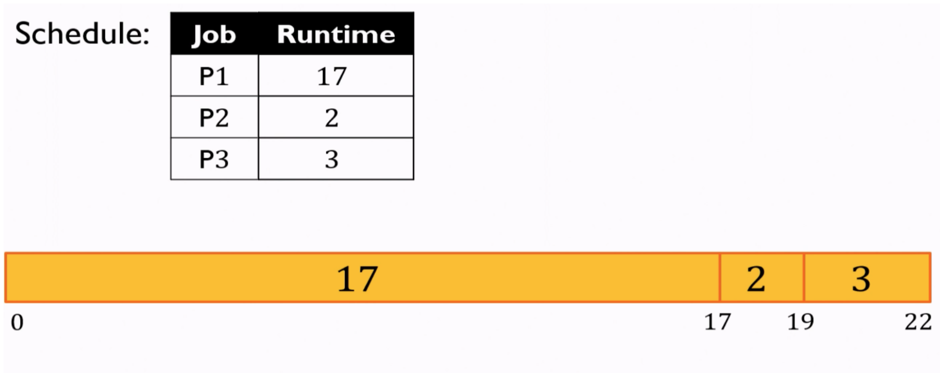


Figure 4: Off-line Scheduling Model

Off-line Scheduling Algorithms

First Come First Served (FCFS)

Definition 12: ***First Come First Served (FCFS)*** *schedules jobs in the exact order they are given to the system.*


Schedule:

Job	Runtime
P1	17
P2	2
P3	3

Figure 5: FCFS Scheduling Example

Example 2: *Consider three jobs:*

- *P1: Runtime = 17 units*
- *P2: Runtime = 2 units*
- *P3: Runtime = 3 units*

With FCFS scheduling:

- *P1: Wait time = 0, Response time = 17*
- *P2: Wait time = 17, Response time = 19*
- *P3: Wait time = 19, Response time = 22*

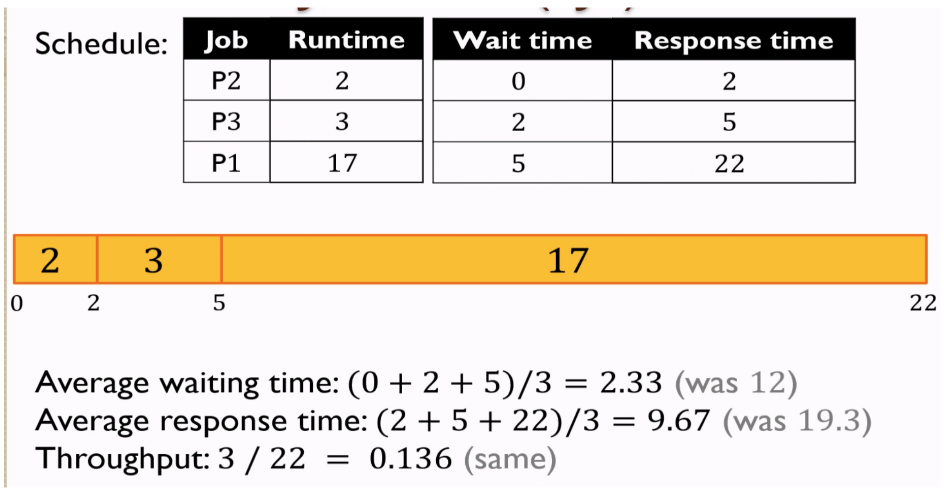
Resulting in:

- *Average wait time:* $(0 + 17 + 19)/3 = 12$ *units*
- *Average response time:* $(17 + 19 + 22)/3 \approx 19.33$ *units*
- *Throughput:* 3 *jobs*$/22$ *time units* ≈ 0.136 *jobs/unit time*

Key Concept 2: *A major drawback of FCFS is the* ***convoy effect****, where short jobs get stuck waiting behind long ones.*

Shortest Job First (SJF)

Definition 13: ***Shortest Job First (SJF)*** *schedules jobs in order of increasing runtime.*



Schedule:

Job	Runtime	Wait time	Response time
P2	2	0	2
P3	3	2	5
P1	17	5	22

Average waiting time: (0 + 2 + 5)/3 = 2.33 (was 12)
Average response time: (2 + 5 + 22)/3 = 9.67 (was 19.3)
Throughput: 3 / 22 = 0.136 (same)

Figure 6: SJF Scheduling Example

Example 3: *Scheduling the same jobs with SJF (P2, P3, P1):*

- *P2 (0-2): Wait time = 0, Response time = 2*
- *P3 (2-5): Wait time = 2, Response time = 5*
- *P1 (5-22): Wait time = 5, Response time = 22*

Resulting in:

- *Average wait time:* $(0 + 2 + 5)/3 \approx 2.33$ *units*
- *Average response time:* $(2 + 5 + 22)/3 \approx 9.67$ *units*

Key Concept 3: *In an off-line, non-preemptive setting, SJF is **provably optimal** for minimizing average wait time and average response time.*

Proof. Proof of SJF optimality:

1. Assume schedule S has the optimal (lowest) average wait time.
2. If S is not an SJF schedule, then there must exist adjacent jobs p_i and p_{i+1} where p_i.runtime $>$ p_{i+1}.runtime.
3. Swapping these jobs reduces their combined contribution to total wait time without affecting other jobs.
4. This contradicts the assumption that S was optimal.
5. Therefore, an optimal schedule must follow SJF ordering.

□

On-line Scheduling Algorithms

In real systems, schedulers must operate without complete future knowledge, making on-line algorithms necessary.

On-line System Model

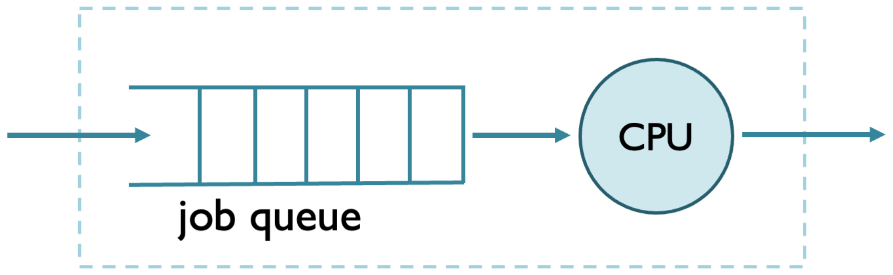
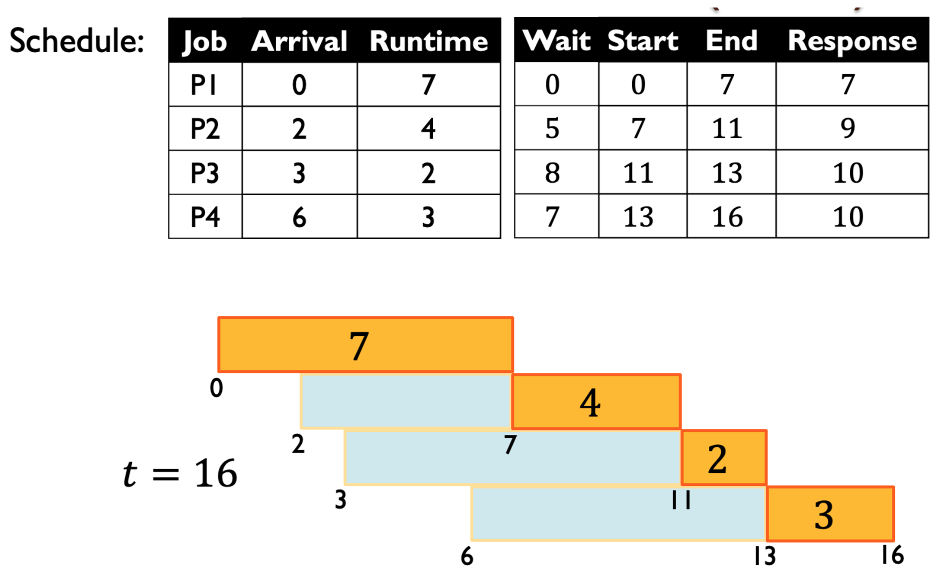


Figure 7: On-line Scheduling Model

The simplified on-line model has these characteristics:

- Jobs arrive at unpredictable times
- Job runtimes become known upon arrival
- No preemption is initially allowed

On-line First Come First Served (FCFS)



Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	0	0	7	7
P2	2	4	5	7	11	9
P3	3	2	8	11	13	10
P4	6	3	7	13	16	10

Figure 8: On-line FCFS Example

Example 4: *Consider jobs with arrival times and runtimes:*

- *P1: Arrival = 0, Runtime = 7*
- *P2: Arrival = 2, Runtime = 4*
- *P3: Arrival = 3, Runtime = 2*
- *P4: Arrival = 6, Runtime = 3*

With on-line FCFS:

- *Average Wait Time: 5 units*
- *Average Response Time: 9 units*
- *Throughput: 0.25 jobs/unit time*

Key Concept 4: *In on-line FCFS, even when shorter jobs arrive while a long job is running, the scheduler cannot make adjustments without preemption.*

On-line Shortest Job First (Non-preemptive)

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	0	0	7	7
P2	2	4	10	12	16	14
P3	3	2	4	7	9	6
P4	6	3	3	9	12	6

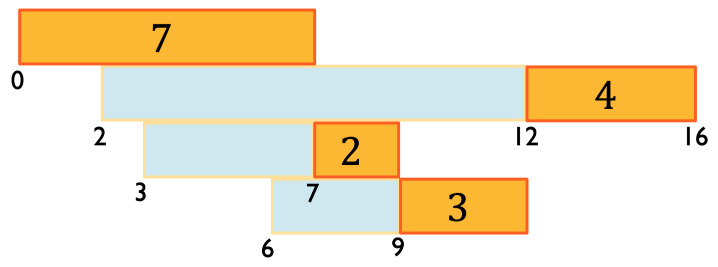


Figure 9: On-line SJF (Non-preemptive) Example

Key Concept 5: *In non-preemptive on-line SJF:*

- *When the CPU becomes free, select the waiting job with the shortest runtime*
- *Once a job starts, it runs to completion regardless of new arrivals*

Example 5: *For the same jobs (P1, P2, P3, P4):*

- *At t=0: P1 starts (only job available)*
- *At t=7: P1 finishes, ready queue has P2(4), P3(2), P4(3); P3 starts*
- *At t=9: P3 finishes, ready queue has P2(4), P4(3); P4 starts*
- *At t=12: P4 finishes, P2 starts*
- *At t=16: P2 finishes, all jobs complete*

Resulting in:

- *Average Wait Time: 4.25 units (improved from 5)*
- *Average Response Time: 8.25 units (improved from 9)*
- *Throughput: 0.25 jobs/unit time (unchanged)*

Limitations of Non-preemptive Scheduling

Key Concept 6: *Non-preemptive on-line scheduling has a fundamental limitation: once a scheduling decision is made, it cannot be reversed, even if it becomes suboptimal due to new arrivals.*

Note: *When P1 starts at t=0, the scheduler commits to running it for its full 7 units, even though shorter jobs P2 and P3 arrive soon after. This constraint prevents achieving optimal scheduling in an on-line context. The solution is to use preemption.*

The Role of Preemption

Preemption allows the scheduler to revise earlier decisions when they prove suboptimal:

- **Advantage**: Can compensate for lack of future knowledge
- **Cost**: Additional context switches increase overhead

Note: *Without preemption, n processes require exactly n-1 context switches. With preemption, this number can be significantly higher.*

Preemptive Online Scheduling

Shortest Remaining Time First (SRTF)

Definition 14: ***Shortest Remaining Time First (SRTF)*** *is the preemptive variant of SJF where at any point (job arrival or completion), the scheduler selects the process with the smallest remaining execution time.*

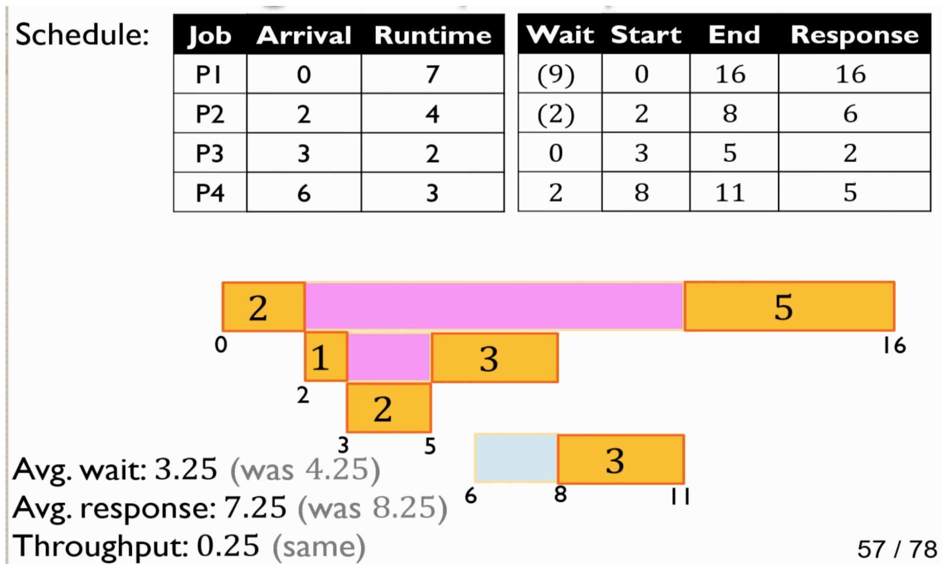
Key Concept 7: *In SRTF, when a new job arrives, it preempts the current job if its total runtime is less than the remaining runtime of the current job. This maximizes responsiveness for short jobs.*

Example 6: *Consider our standard job set: P1(0,7), P2(2,4), P3(3,2), P4(6,3):*

- *t=0: P1 starts (only job, remaining time=7)*
- *t=2: P2 arrives (runtime=4); P1 has remaining time=5; P1 is preempted, P2 starts*
- *t=3: P3 arrives (runtime=2); P2 has remaining time=3; P2 is preempted, P3 starts*
- *t=5: P3 completes; ready queue has P1(5), P2(3); P2 resumes*
- *t=6: P4 arrives (runtime=3); P2 has remaining time=2; P2 continues*
- *t=8: P2 completes; ready queue has P1(5), P4(3); P4 starts*
- *t=11: P4 completes; P1 resumes*
- *t=16: P1 completes*

Performance metrics:

- *Average Wait Time: 3.25 units (improved from 4.25 in non-preemptive SJF)*
- *Average Response Time: 7.25 units (improved from 8.25)*
- *Throughput: 0.25 jobs/unit time (unchanged)*



Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	(9)	0	16	16
P2	2	4	(2)	2	8	6
P3	3	2	0	3	5	2
P4	6	3	2	8	11	5

Avg. wait: 3.25 (was 4.25)
Avg. response: 7.25 (was 8.25)
Throughput: 0.25 (same)

Figure 10: SRTF Scheduling Example

Realistic System Model and Runtime Estimation

In real operating systems, jobs arrive at unknown times and their runtimes are not known in advance. This creates a fundamental challenge: when a job arrives, the scheduler has no way to know whether it will be short or long, making true SJF/SRTF implementation impossible.

Exponentially Weighted Average Method

To address this challenge, operating systems can estimate future CPU burst lengths based on historical behavior:

Definition 15: *The **Exponentially Weighted Average** technique predicts a process's next CPU burst length by combining its past behavior, giving more weight to recent bursts while still considering older patterns.*

- Let t_n be the actual measured length of the n^{th} CPU burst
- Let τ_{n+1} be the predicted length for the next CPU burst
- The prediction formula is: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$

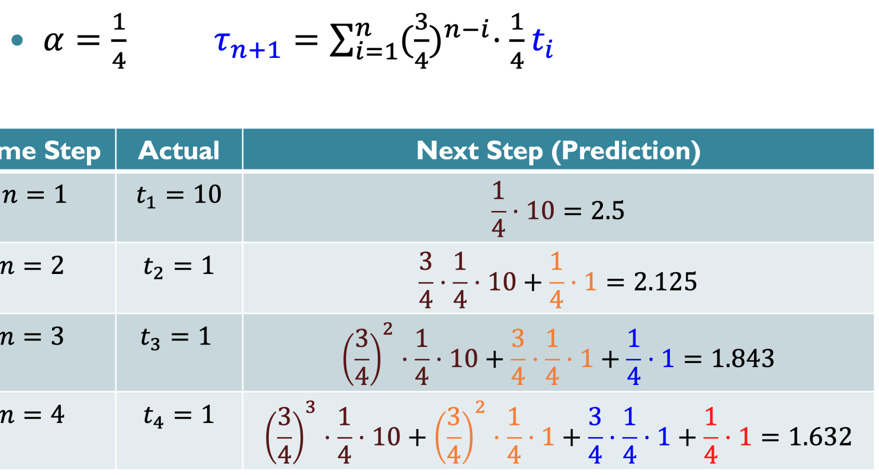
- Where α is a weighting factor with $0 \leqslant \alpha \leqslant 1$

The parameter α controls the balance between recency and history:

- $\alpha = 1$: Only the most recent burst matters ($\tau_{n+1} = t_n$)
- $\alpha = 0$: Only the previous prediction matters ($\tau_{n+1} = \tau_n$)
- Typical values (e.g., $\alpha = 0.5$) balance responsiveness with stability

The expanded form shows how all previous bursts are weighted:

$$\tau_{n+1} = \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} t_i + (1-\alpha)^n \tau_1$$



- $\alpha = \frac{1}{4}$ $\qquad \tau_{n+1} = \sum_{i=1}^{n} (\frac{3}{4})^{n-i} \cdot \frac{1}{4} t_i$

Time Step	Actual	Next Step (Prediction)
$n = 1$	$t_1 = 10$	$\frac{1}{4} \cdot 10 = 2.5$
$n = 2$	$t_2 = 1$	$\frac{3}{4} \cdot \frac{1}{4} \cdot 10 + \frac{1}{4} \cdot 1 = 2.125$
$n = 3$	$t_3 = 1$	$\left(\frac{3}{4}\right)^2 \cdot \frac{1}{4} \cdot 10 + \frac{3}{4} \cdot \frac{1}{4} \cdot 1 + \frac{1}{4} \cdot 1 = 1.843$
$n = 4$	$t_4 = 1$	$\left(\frac{3}{4}\right)^3 \cdot \frac{1}{4} \cdot 10 + \left(\frac{3}{4}\right)^2 \cdot \frac{1}{4} \cdot 1 + \frac{3}{4} \cdot \frac{1}{4} \cdot 1 + \frac{1}{4} \cdot 1 = 1.632$

Figure 11: Example of Exponential Averaging with $\alpha = 1/4$

Key Concept 8: *While this method enables practical implementations of SJF/SRTF in real systems, it has important limitations:*

- ***Prediction accuracy:*** *Estimates may be inaccurate when processes change behavior patterns*
- ***Computational overhead:*** *The system must track history and recalculate estimates for every process*
- ***Cold start problem:*** *New processes lack historical data, requiring default initial values*

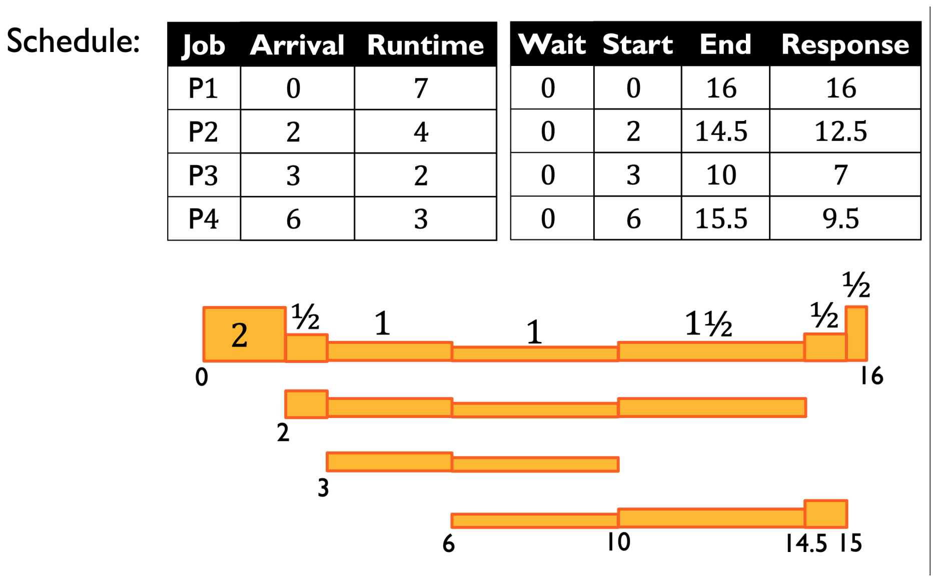
Processor Sharing and Round Robin

Processor Sharing (PS)

Definition 16: ***Processor Sharing (PS)*** *is a theoretical scheduling model where all ready processes receive equal, simultaneous portions of the CPU, creating the illusion that each process has its own virtual processor running at a fraction of the full speed.*

Key Concept 9: *If k processes are running simultaneously:*

- *Each process effectively runs at $\frac{1}{k}$ of the CPU's speed*
- *Short jobs make progress immediately rather than waiting behind long ones*
- *As processes complete, remaining processes receive larger CPU shares*



Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	0	0	16	16
P2	2	4	0	2	14.5	12.5
P3	3	2	0	3	10	7
P4	6	3	0	6	15.5	9.5

Figure 12: Processor Sharing Concept

Workload Characteristics and PS Effectiveness Processor Sharing performs best with **skewed workload distributions**, specifically when:

- Many processes have very short runtimes, while a few have very long runtimes
- The coefficient of variation (CV $= \frac{\text{standard deviation}}{\text{mean}}$) of job lengths exceeds 1
- Process runtimes follow heavy-tailed distributions (like Pareto distributions)

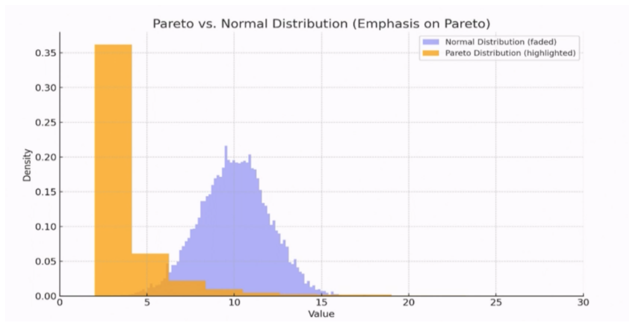


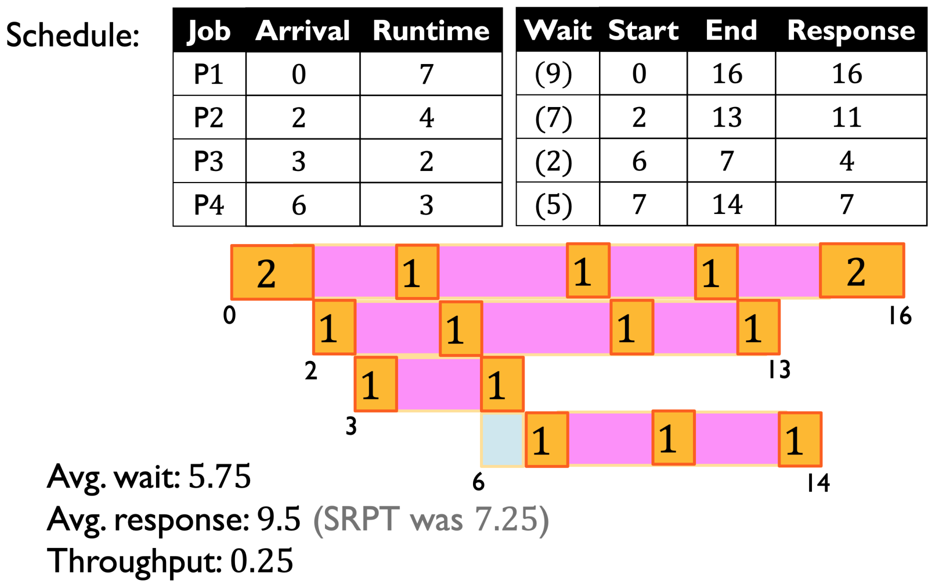
Figure 13: Pareto vs. Normal Distribution (Illustrating Runtime Skewness)

Round Robin (RR) Scheduling

Definition 17: ***Round Robin (RR)*** *is a preemptive scheduling algorithm that approximates Processor Sharing by allocating small time slices to each process in a circular manner, ensuring that all processes get regular access to the CPU.*

Key Concept 10: *The core mechanism of Round Robin consists of:*

- *A* ***time quantum*** *(q): A fixed maximum CPU time allocated to each process*
- *A* ***circular ready queue****: Processes are arranged in FIFO order*
- ***Preemption****: When a process uses its entire quantum without completing, it is moved to the back of the queue*



Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	(9)	0	16	16
P2	2	4	(7)	2	13	11
P3	3	2	(2)	6	7	4
P4	6	3	(5)	7	14	7

Avg. wait: 5.75
Avg. response: 9.5 (SRPT was 7.25)
Throughput: 0.25

Figure 14: Round Robin Scheduling Example

Time Quantum Selection The time quantum (q) significantly affects RR performance:

- **Large quantum:** Reduces context switching overhead but approaches FCFS behavior
- **Small quantum:** Better approximates Processor Sharing but increases overhead
- **Optimal selection:** Must balance responsiveness with overhead
- With n processes, no process waits more than $(n-1)q$ time units for CPU access

Key Concept 11: *The context switch overhead can be quantified: if a context switch takes time c, the fraction of CPU time wasted is $\frac{c}{q+c}$. As quantum q decreases, overhead increases.*

Key Characteristics of Round Robin

- **On-line algorithm:** Requires no knowledge of future arrivals or runtimes
- **Preemptive:** Uses timer interrupts to enforce time quanta
- **Fairness:** Provides uniform CPU access to all ready processes
- **Effective for interactive systems:** Short burst processes complete quickly
- **Hardware support:** Relies on timer interrupts for preemption

Note: *Round Robin has become the foundation for many practical CPU scheduling algorithms in modern operating systems, often enhanced with priority mechanisms and adaptive time quanta.*

Effective Quantum Data and Process Behavior

Real-world process behavior shows dramatic differences between application types:

- **Interactive and multimedia applications** (like web browsers, media players):
  - Use extremely short CPU bursts, typically below 0.01 ms
  - Voluntarily release the CPU after using less than 1/1000 of the allocated quantum
  - This pattern reflects their I/O-bound nature, where processes quickly block waiting for external events
- **System utilities** (like file operations) use moderate bursts
- **Development tools** (like compiler operations) show longer and more variable CPU usage patterns
- **CPU-bound applications** display fundamentally different behavior:
  - Approximately 40% use their full quantum allocation
  - Another 40% are interrupted before completion but still use more than half the quantum

  - Only rarely do they voluntarily yield the processor

This data has significant implications for scheduler design. Setting a single fixed quantum size is inherently inefficient—interactive applications naturally yield the CPU long before their quantum expires, while CPU-bound jobs typically consume their entire allocation.