Operating System
Tutorial 6 - Process Scheduling

Yonghao Lee

May 10, 2025

Scheduling Criteria

Scheduling determines which process runs on each available CPU at any given time. This is the operating system's responsibility, where:

- The **scheduler** decides which job to run next
- The **dispatcher** handles the actual context switching

Optimization Goals

We aim to optimize several metrics:

- **CPU Utilization:** Maximize CPU time spent on relevant instructions, keeping processors as busy as possible
- **Throughput:** Maximize the number of completed processes per time unit
- **Waiting Time:** Minimize the time processes spend waiting in the ready queue
- **Turnaround Time:** Minimize the total time from process submission to completion

Challenges in Scheduling

Scheduling is difficult for several reasons:

- Each process has a requested running time that is often unknown in advance
- We cannot reliably estimate exact running times
- Processes rarely arrive simultaneously
- Even with perfect information (known running times and arrival times), optimal scheduling with 2 or more CPUs is NP-hard

Due to these computational limitations, we must rely on heuristic approaches rather than optimal algorithms.

Scheduling Algorithm Properties

When designing scheduling algorithms, we aim to ensure the following properties:

- **Fairness:** All processes should receive a fair allocation of CPU time, preventing any process from being unfairly prioritized or neglected
- **Starvation Prevention:** The system must avoid scenarios where processes are perpetually denied CPU access due to scheduling decisions
- **Preemption Policy:** We must determine whether the scheduler can forcibly interrupt executing processes (preemptive) or must wait for processes to voluntarily release the CPU (non-preemptive)
- **Information Model (Online vs. Offline):**
  - *Online algorithms* make decisions based only on currently available information, without knowledge of future process arrivals or behaviors
  - *Offline algorithms* have complete information about all processes (arrival times, execution requirements) before scheduling begins

Types of Scheduling

- **Batch:** The jobs do not need an intervention of the user, for example: training a neural network
- **Interactive:** Users wait behind the screen, supplying input or expecting immediate output, this scheduling focuses on how to respond quickly, for example, web browsers, IDEs
- **Real Time:** The execution should be completed before a given time (a deadline). Time matters here, sometimes bad things can happen if we do not meet the deadline

We rarely talk about real time scheduling in this course, but we do need to remember the existence of it.

Basic Scheduling Algorithms

First-Come First-Serve (FCFS)

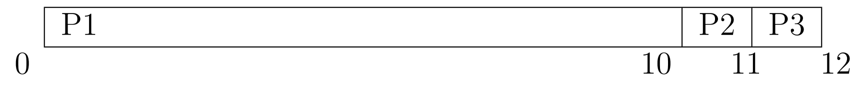
The process that asks for the CPU first gets the CPU first, then the second that asked and so on. This is the simplest CPU-scheduling algorithm, this can be implemented using a queue, we add to the tail of the queue and remove from the head of the queue. This algorithm is apparently starvation-free, given that the running time of any process is finite.

FCFS Example 1

Given the following times:

Process	Burst Time
P1	10
P2	1
P3	1

Gantt Chart for processes scheduling:



Waiting times:

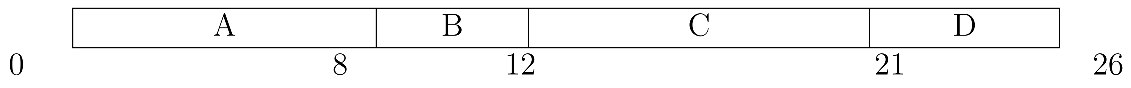
- P1 - 0
- P2 - 10 + cs
- P3 - 10 + cs + 1 + cs = 11 + 2cs
- Average: $\frac{(0)+(10+\text{cs})+(11+2\text{cs})}{3} = 7 + \text{cs}$

FCFS Example 2

Consider the following processes with their burst times:

Process	Burst Time
A	8
B	4
C	9
D	5

Gantt Chart for processes scheduling:



Performance:

- CPU Utilization: $\frac{\text{actual running time}}{\text{total running time}} = \frac{26}{26+3\text{cs}}$
- Average Turn-Around time: $\frac{(8)+(12+\text{cs})+(21+2\text{cs})+(26+3\text{cs})}{4} = 16.75 + 1.5\text{cs}$
- Average Waiting time: $\frac{(0)+(8+\text{cs})+(12+2\text{cs})+(21+3\text{cs})}{4} = 10.25 + 1.5\text{cs}$
- Throughput: $\frac{\text{finished jobs}}{\text{total running time}} = \frac{4}{26+3\text{cs}}$

Shortest Job First/Next - (SJF/SJN)

When the CPU is available, it is assigned to the process that has the smallest next CPU burst time. This algorithm is provably optimal for minimizing average waiting time when all processes are available simultaneously.

SJF can be implemented in two variants:

- **Non-preemptive SJF**: Once a process starts executing, it runs to completion
- **Preemptive SJF** (also called Shortest Remaining Time First or SRTF): If a new process arrives with a shorter burst time than the remaining time of the current process, the current process is preempted

The significant challenges with implementing SJF include:

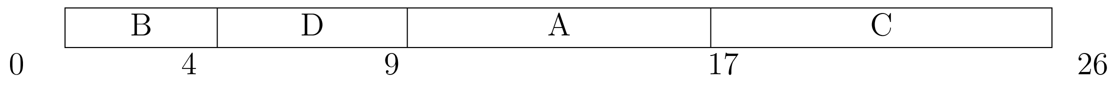
- **Unknown Burst Times**: In real systems, the exact CPU burst time is unknown in advance and must be estimated
- **Starvation Risk**: Long processes may never execute if shorter processes continuously arrive (particularly in online settings)

SJF Example

Consider the following processes with their burst times:

Process	Burst Time
A	8
B	4
C	9
D	5

Gantt Chart for processes scheduling:



Performance:

- CPU Utilization: $\frac{\text{actual running time}}{\text{total running time}} = \frac{26}{26+3cs}$
- Average Turn-Around time: $\frac{(4)+(9+cs)+(17+2cs)+(26+3cs)}{4} = 14 + 1.5cs$
- Average Waiting time: $\frac{(0)+(4+cs)+(9+2cs)+(17+3cs)}{4} = 7.5 + 1.5cs$
- Throughput: $\frac{\text{finished jobs}}{\text{total running time}} = \frac{4}{26+3cs}$

Recall that in FCFS, the average waiting time was higher than SJF.

Shortest Remaining Time First (SRTF)

This is the preemptive variant of SJF. If a new process arrives with a CPU burst time that is smaller than the remaining time of the currently executing process, the current process is preempted. This ensures that shorter processes are handled very quickly.

Key characteristics of SRTF:

- Minimizes average waiting time (proven to be optimal)
- Requires constant monitoring of remaining execution time
- Short processes receive immediate attention
- May lead to starvation of longer processes if shorter jobs continually arrive
- Faces the same challenge as SJF: predicting burst times in advance

SRTF Example

Consider the following processes with their arrival and burst times:

Process	Arrival Time	Burst Time
A	0	8
B	1	4
C	2	9
D	3	5

Process execution sequence: A → B → D → A → C

Time	Process	Reason
0 - 1	A	A starts execution
1 - 5	B	B preempts A (4 < remaining 7 of A)
5 - 10	D	D has shortest remaining time (5)
10 - 17	A	A resumes (remaining time: 7)
17 - 26	C	C executes (remaining time: 9)

Preemption analysis:

- At $t = 0$: Process A begins execution
- At $t = 1$: Process B arrives with burst time 4 < remaining time of A (8-1=7), so A is preempted and B begins
- At $t = 2$: Process C arrives with burst time 9 > remaining time of B (4-1=3), so B continues
- At $t = 3$: Process D arrives with burst time 5 > remaining time of B (4-2=2), so B continues

- At $t = 5$: Process B completes, D has the shortest remaining time (5), so D begins
- At $t = 10$: Process D completes, A has the shortest remaining time (7), so A resumes
- At $t = 17$: Process A completes, only C remains with 9 units of time
- At $t = 26$: Process C completes, all processes finished

Performance:

- CPU Utilization: $\frac{\text{actual running time}}{\text{total running time}} = \frac{26}{26+4\text{cs}}$
- Average Turn-Around time: $\frac{(17-0+3\text{cs})+(5-1+\text{cs})+(26-2+3\text{cs})+(10-3+\text{cs})}{4} = 13 + 2\text{cs}$
- Average Waiting time: $\frac{((0)+(10-1+3\text{cs}))+(\text{cs})+(17+3\text{cs}-2)+(5-3+\text{cs})}{4} = 6.5 + 2\text{cs}$

Priority Scheduling

The CPU is allocated to the process with the highest priority.

A priority is associated with each process, fixed range of numbers and we use low numbers to represent higher priorities. Processes with the same priorities are scheduled in FCFS order. SJF is a special case for priority scheduling, where the priority is the CPU burst.

Priorities can be defined:

- **Internally** – Use a measurable quantity.
  - Time limits, memory requirements, memory consumption.
- **Externally** – Usually by the user.
  - Importance, political factors.

Priority scheduling has a preemptive version.
Major problems:

- Indefinite blocking and starvation.
- Solution: Aging
  - Gradually increase the priority of processes that wait for a long time.

This algorithm can be preemptive but usually not. It is not starvation-free.

Round Robin

A small unit of time, called a quantum is defined.

- Generally, 10-100 milliseconds.

The ready queue is treated as a circular, FIFO queue.
The CPU scheduler goes around the ready queue.

- Allocate the CPU to each process for a time interval of up to 1 time quantum.
- CPU Burst < 1 quantum → release the CPU voluntarily.
- CPU Burst > 1 quantum → context switch.

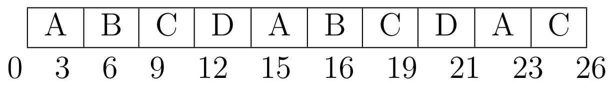
Starvation-Free?	**Preemptive?**
Yes	Yes

Round Robin - Example

Consider the following processes with their burst times:

Process	**Burst Time**
A	8
B	4
C	9
D	5

And a quantum of 3 milliseconds.
Gantt Chart for processes scheduling:



Execution sequence:

- 0-3: Process A (3 of 8 units)
- 3-6: Process B (3 of 4 units)
- 6-9: Process C (3 of 9 units)
- 9-12: Process D (3 of 5 units)
- 12-15: Process A (6 of 8 units)
- 15-16: Process B (completes)
- 16-19: Process C (6 of 9 units)
- 19-21: Process D (completes)

- 21-23: Process A (completes)
- 23-26: Process C (completes)

Performance:

- CPU Utilization: $\frac{\text{actual running time}}{\text{total running time}} = \frac{26}{26+9\text{cs}}$
- Average Turn-Around time: $\frac{(23)+(16)+(26)+(21)}{4} = \frac{86}{4} = 21.5$

  Adding context switching overhead: $\frac{(23+8\text{cs})+(16+5\text{cs})+(26+9\text{cs})+(21+7\text{cs})}{4} = 21.5 + 7.25\text{cs}$
- Average Waiting time (Time spent waiting - not executing): $\frac{(23-8)+(16-4)+(26-9)+(21-5)}{4} = \frac{60}{4} = 15$

  With detailed calculation including context switching: $\frac{(15+8\text{cs})+(12+5\text{cs})+(17+9\text{cs})+(16+7\text{cs})}{4} = 15 + 7.25\text{cs}$
- Throughput: $\frac{\text{finished jobs}}{\text{total running time}} = \frac{4}{26+9\text{cs}}$

Round Robin Analysis

- Performance depends heavily on quantum size:
  - Too large: Degenerates to FCFS.
  - Too small: Excessive context switching overhead.
- Typically, a guideline is that 80% of CPU bursts should be shorter than the time quantum to balance responsiveness and overhead.
- Provides good response time for short processes.
- Fair allocation of CPU among processes.
- Guarantees that no process waits more than $(n-1)q$ time units for its next turn (where n is the number of processes and q is the quantum), assuming processes remain in the ready queue.
- **Impact of Running Time Distribution:**
  - RR performance is poor if the variance in job (CPU burst) lengths is small. For example, if all jobs are of similar long lengths.
  - RR is generally good for "real life" jobs where there's a mix of short and long CPU bursts.
- **Context Switch Overhead:**
  - Context switches significantly hurt performance. This is a critical factor.

  - A context switch involves running other process/thread code (OS kernel code), which changes the content of the CPU cache (e.g., instruction cache, data cache, TLB).
  - This cache pollution means that even if the context switch itself is assumed to take zero time for calculation purposes, the subsequent performance of the switched-in process (and potentially other processes) can be degraded as it needs to rebuild its cache footprint.

Round Robin - Comparative Analysis

When comparing scheduling algorithms, it's important to understand their performance characteristics under different workloads. Let's examine whether Round Robin is always superior to FCFS when we assume context switches have no cost.

Consider a scenario with 10 jobs:

- All jobs start at time 0.
- Each job requires exactly 100 seconds of CPU time.
- Round Robin quantum is set to 1 second.

Job	**FCFS Completion Time**	**RR Completion Time**
1	100	991
2	200	992
...	...	...
10	1000	1000

Key observations:

- Both FCFS and RR finish processing all jobs at the same total time (1000 seconds in this idealized example without context switch overhead).
- Under FCFS, jobs complete sequentially (job 1 at 100s, job 2 at 200s, etc.).
- Under RR, with many long jobs and a small quantum, almost all jobs make incremental progress and tend to finish closer to the overall completion time of the entire batch.
- The average completion time is much worse under RR in this specific scenario of identical long jobs. This is because each job is repeatedly preempted and has to wait for all other $n - 1$ jobs to get their quantum before it gets another turn.
- This occurs because a job can run briefly, then be preempted, and its next execution segment might be scheduled only much later in the future after many other processes have had their turns.

Quantum Size Impact Analysis

Consider four processes with running times: 53, 8, 68, and 24. (Process order for FCFS can significantly affect its performance, "Best FCFS" implies an optimal ordering, likely shortest jobs first if all arrive at t=0, while "Worst FCFS" implies a detrimental order, likely longest jobs first).

Scheduling		P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	$\mathbf{31\frac{1}{4}} \leftarrow$
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	$61\frac{1}{4}$
	Q = 8	80	8	85	56	$57\frac{1}{4}$
	Q = 10	82	10	85	68	$61\frac{1}{4}$
	Q = 20	72	20	85	88	$66\frac{1}{4}$
	Worst FCFS	68	145	0	121	$\mathbf{83\frac{1}{2}} \leftarrow$
Completion Time	Best FCFS	85	8	153	32	$69\frac{1}{2}$
	Q = 1	137	30	153	81	$100\frac{1}{4}$
	Q = 5	135	28	153	82	$99\frac{1}{2}$
	Q = 8	133	16	153	80	$95\frac{1}{2}$
	Q = 10	135	18	153	92	$99\frac{1}{2}$
	Q = 20	125	28	153	112	$104\frac{1}{4}$
	Worst FCFS	121	153	68	145	$121\frac{3}{4}$

This comparison demonstrates that:

- FCFS with an optimal ordering (e.g., shortest jobs first if arrival times are the same, which mimics SJF) can yield better average waiting/completion times than Round Robin.
- Round Robin with an appropriately chosen quantum (e.g., Q=8 in this specific example for wait time) can offer a reasonable trade-off.
- Process ordering dramatically impacts FCFS performance (e.g., average wait time of $31\frac{1}{4}$ for Best FCFS vs $83\frac{1}{2}$ for Worst FCFS).

Advanced Scheduling Techniques

Multilevel Queue Scheduling

This is a multilevel queue scheduling algorithm, where we partition the ready queue into several separate groups. Each group has its own scheduling algorithm, and scheduling is also done among the queues, for instance, using fixed-priority preemptive scheduling or time slices between the queues.

Processes are classified into different groups, such as foreground interactive processes and background batch processes.

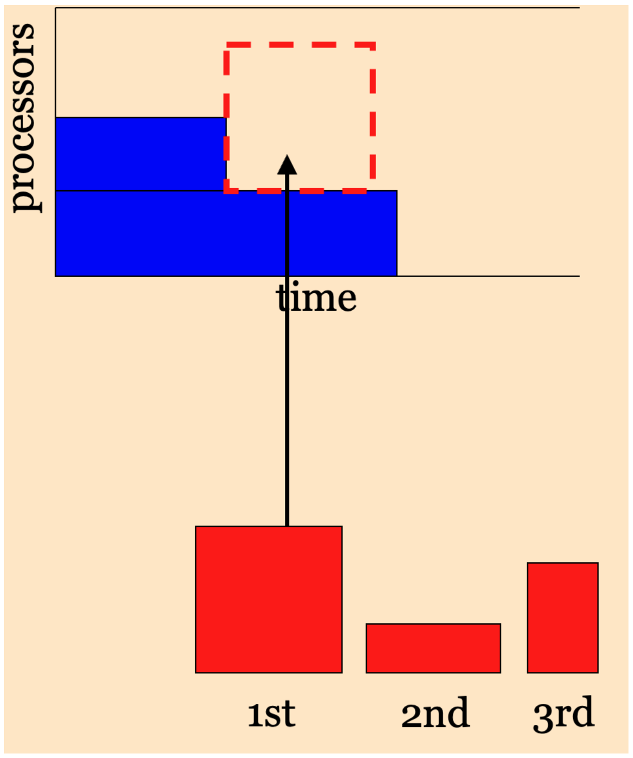


Figure 1: Multilevel queue scheduling

Multi-level Feedback Queues

We can add feedback mechanisms, where when a process finishes its execution, it gets feedback and can move to a different queue. For example, processes that use too much CPU time may move to a lower priority queue, while processes that wait too long may move to a higher priority queue (similar to aging).

The multilevel-feedback queue scheduler is defined by:

- Number of queues
- Scheduling algorithms for each queue
- Procedure used to determine which queue a process enters when it needs service
- Procedures to determine when to upgrade or demote a process

Example: Three-Level Feedback Queue

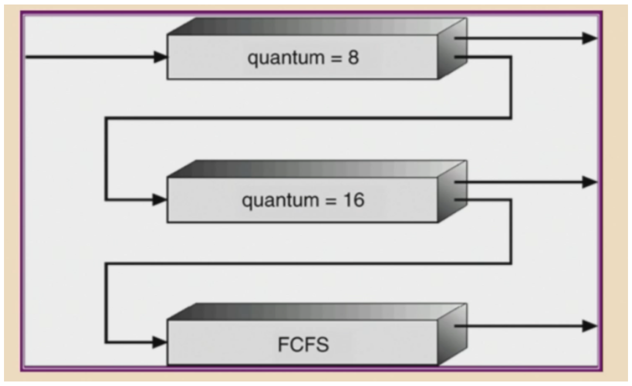


Figure 2: 3 Level Feedback Queue

As shown in the figure, a practical implementation might consist of three queues:

- Q_0 – highest priority queue with a time quantum of 8 milliseconds. Processes enter the system through this queue.
- Q_1 – medium priority queue with a time quantum of 16 milliseconds. Processes that exceed their time quantum in Q_0 are demoted to this queue.
- Q_2 – lowest priority queue using FCFS (First-Come, First-Served) scheduling without time slicing. Processes that exceed their time quantum in Q_1 are demoted to this queue.

This implementation balances responsiveness and throughput by:

- Giving short, interactive processes preferential treatment (they complete quickly in Q_0)
- Providing increasingly longer time slices for medium-length processes in Q_1
- Ensuring CPU-intensive background processes still complete eventually in Q_2

The arrows in the diagram show both the demotion path for processes that exhaust their time quantum and the flow of execution. A process will not be scheduled from a lower-priority queue until all higher-priority queues are empty, enforcing the priority relationship between queues.

Parallel Systems Scheduling

Scheduling on supercomputers often involves a scheduler that is not part of the individual node's operating system (OS). Instead, it's a higher-level service managing resources across all servers. Its primary purpose is to determine which processors will be allocated to each parallel job. A well-known example of such a scheduler is SLURM. Common strategies include First-Come First-Serve (FCFS), and various forms of Backfilling schedulers.

While Shortest Job First (SJF) achieves minimal average waiting time in offline settings for a single processor, the scheduling problem becomes NP-hard for two or more processors. This complexity means that finding optimal solutions is generally infeasible, and approximation algorithms can be either too expensive computationally or perform poorly compared to effective heuristics. The following diagrams illustrate different states of such a parallel system.

Interpreting the Scheduling Graphs

The diagrams below depict the allocation of processors to jobs over time. Here's a general guide to reading them:

- **Vertical Axis (Y-axis):** Represents the total number of available processors in the system.
- **Horizontal Axis (X-axis):** Shows the progression of time, typically moving from left (past/current) to right (future).
- **Colored Blocks:** Each distinct colored block represents an individual parallel job.
  - The *height* of a block corresponds to the number of processors that job requires.
  - The *width* of a block represents the duration or burst time of that job.
- **Job Placement:** Jobs are placed on the graph according to their start time and the processors they are allocated. Blocks stacked vertically are running concurrently on different sets of processors.
- **'Current Time' Marker:** If present, an indicator (like an arrow labeled "Current Time") points to a specific snapshot of the system's state. Jobs to the left or straddling this marker are currently running or have recently completed.
- **Waiting Jobs Queue:** Jobs that have arrived in the system but are awaiting scheduling and execution.
- **Scheduled Future Jobs:** Blocks appearing to the right of the 'Current Time' marker or after the completion of currently running jobs represent jobs that are scheduled to run in the future.
- **Idle Processors (White Space):** Any empty regions on the graph below the maximum processor line, where no colored blocks are present, signify periods when processors are unutilized or idle.

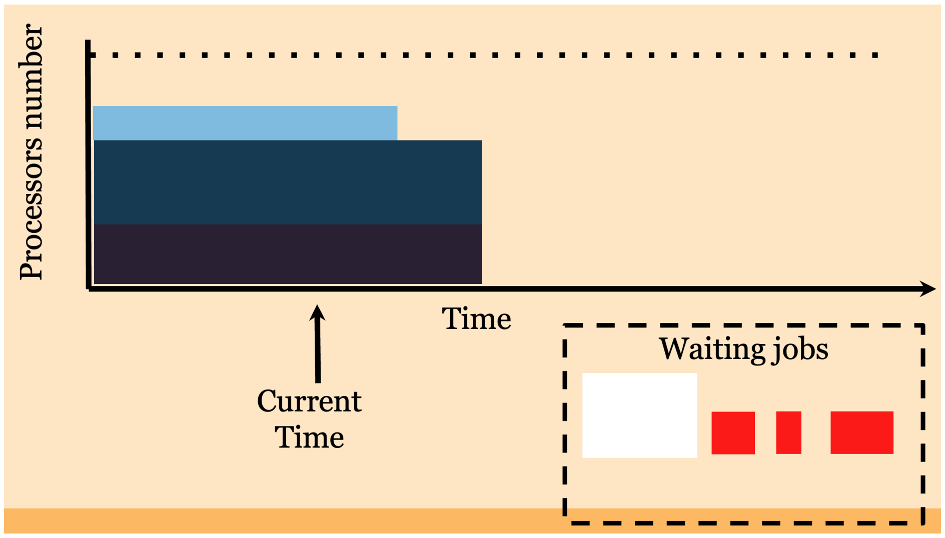


Figure 3: Stage 1 of Parallel Job Scheduling

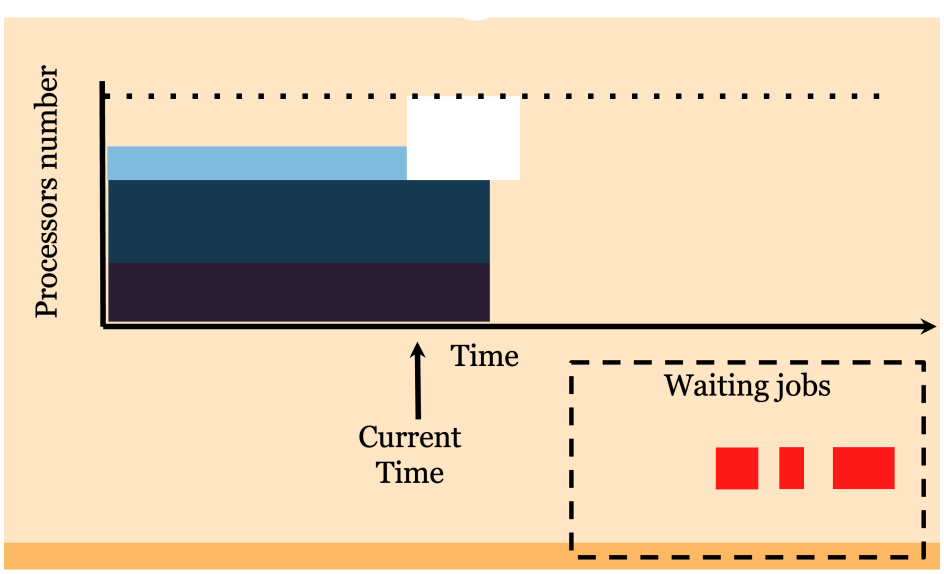


Figure 4: Stage 2 of Parallel Job Scheduling

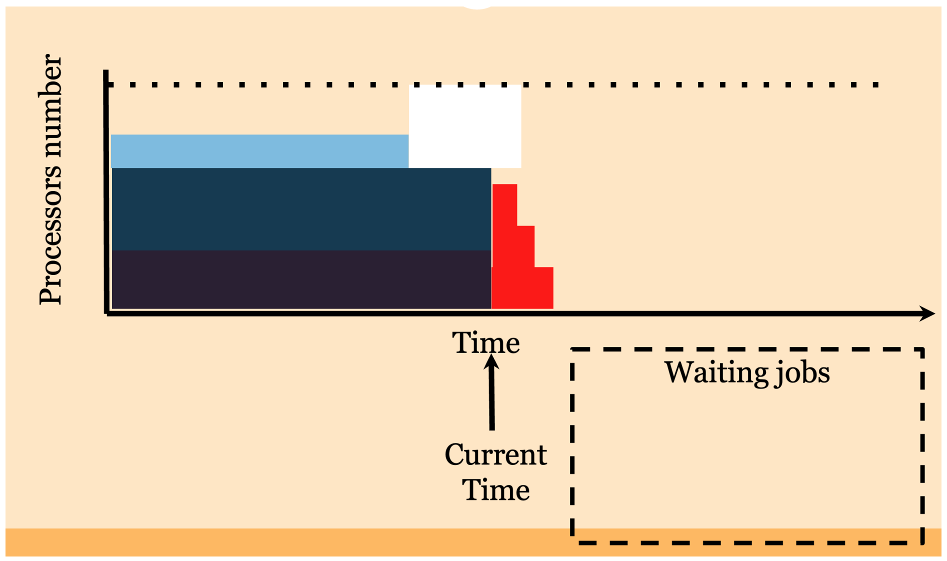


Figure 5: Stage 3 of Parallel Job Scheduling

Backfilling

We can also use a better algorithm called back-filling:

This scheduling heuristic aims to improve resource utilization in parallel systems. When a high-priority job (e.g., at the head of an FCFS queue) must wait for resources, potentially leaving processor slots idle, backfilling allows the scheduler to select smaller, suitable jobs from further down the waiting queue. These selected jobs are then run earlier by fitting them into these empty slots. A critical condition for backfilling is that these "backfilled" jobs must not delay the predetermined start time of any higher-priority job or any job that has an existing reservation. This improves overall system throughput by minimizing wasted CPU cycles.

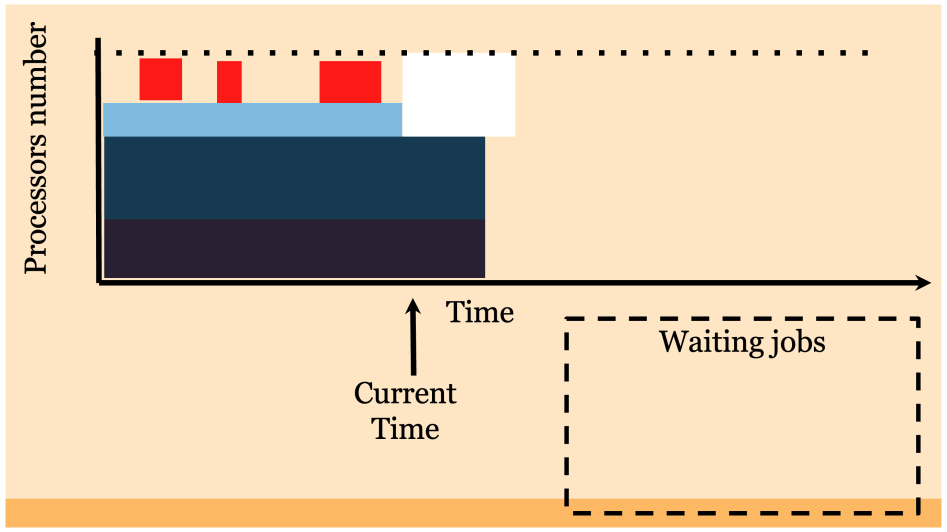


Figure 6: Backfilling Example

The EASY Scheduler Algorithm

The EASY (Extensible Argonne Scheduling sYstem) scheduler is a popular algorithm for parallel systems that seeks to balance the fairness of First-Come First-Serve (FCFS) with the increased system utilization provided by backfilling. It achieves this by making a reservation for the first job in the FCFS queue that cannot currently run, and then attempting to backfill other, smaller jobs around this reservation.

The core logic of the EASY scheduler generally follows these steps:

1. **FCFS Processing:** Initially, jobs from the waiting queue are scheduled on available processors strictly following FCFS order.

2. **Reservation for First Waiting Job:** If the first job in the FCFS queue (let's call it Job_1) cannot be started immediately (e.g., due to a lack of sufficient processors), the EASY scheduler makes a reservation for Job_1. This reservation marks the earliest estimated time that Job_1 will be able to acquire its needed resources and begin execution.

3. **Backfilling Around Reservation:** With Job_1's reservation in place, the scheduler then considers other jobs further down the waiting queue for backfilling. These jobs can be scheduled out of strict FCFS order if they can run and complete *without delaying the reserved start time of Job_1* and without conflicting with any other existing reservations.

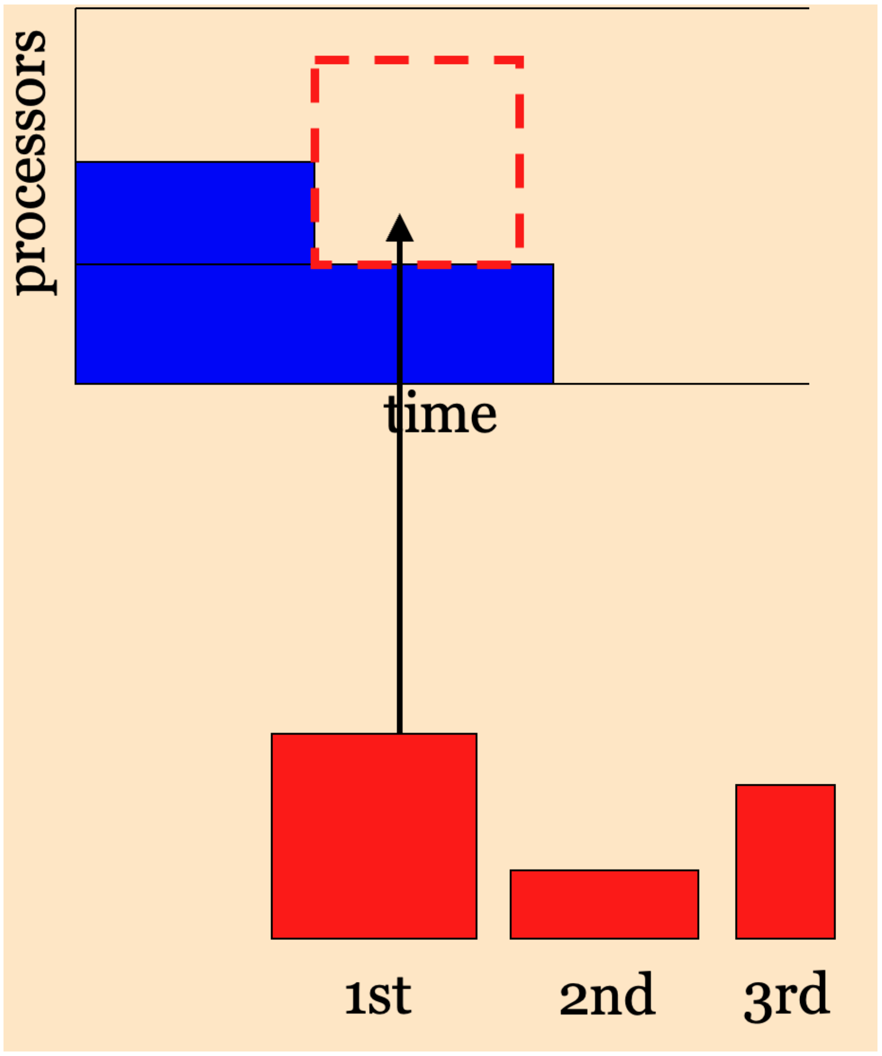


Figure 7: EASY Scheduler: Reservation and Backfilling Example

Conditions for Backfilling in EASY

For a job to be successfully backfilled in a system using reservations (like EASY), it must satisfy specific conditions to ensure that no higher-priority job is unfairly delayed. A candidate job from the waiting queue can be backfilled if it meets **at least one** of the following criteria:

- The backfill job's estimated runtime allows it to complete *before* the reserved start time of the primary waiting job (e.g., Job_1).
- The backfill job will only utilize "extra" processors—that is, processors not allocated to or required by any job with an existing reservation up until that reservation's start time.

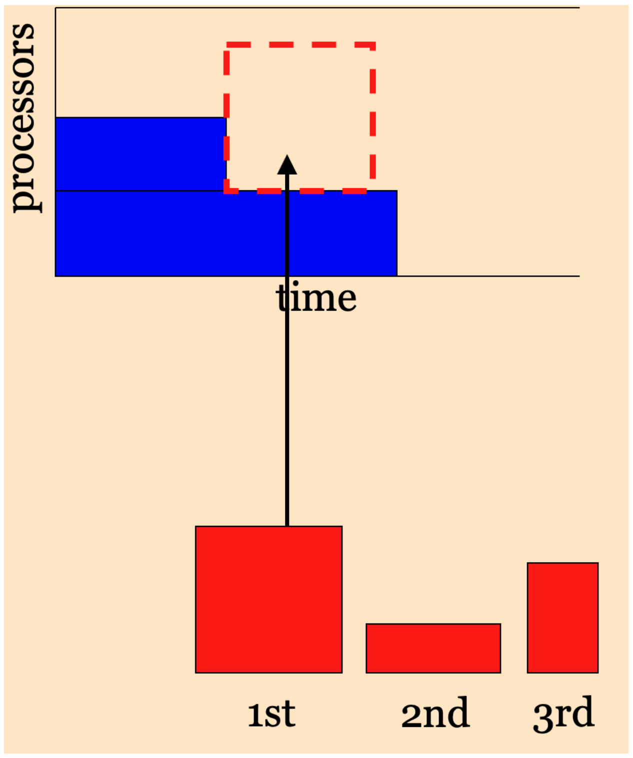


Figure 8: Conditions for Backfilling with Reservations

Role and Importance of Running Time Estimates

The ability of schedulers like EASY to make reservations and perform effective backfilling heavily depends on the runtime estimates provided by users when they submit their jobs.

- **User Input:** When submitting a job, users are typically required to specify both the number of processors needed and an estimate of the job's maximum execution time.
- **Predicting Availability:** The scheduler uses these runtime estimates of currently running jobs to predict when processors will become free, which is essential for calculating reservation times for waiting jobs.
- **Validating Backfill Candidates:** User-provided runtimes are also critical for verifying whether a potential backfill job can indeed finish before an established reservation without causing a delay.
- **Enforcement and System Integrity:** To maintain the reliability of the schedule and ensure fairness, jobs that exceed their stated runtime estimate, particularly if they threaten to delay reserved jobs, may be terminated (killed) by the system. This incentivizes users to provide reasonably accurate estimates.