Chapter 1

Introduction to CPU Scheduling

1.1 The Need for Scheduling

In modern computing systems, physical resources are limited, necessitating efficient methods to share these resources among multiple processes and threads. CPU scheduling is a fundamental concept in operating systems that addresses how to allocate processor time among competing tasks.

1.1.1 Resource Sharing Approaches

There are two main approaches to achieve resource sharing in computing systems:

- **Time Sharing** — Running one task at a time and rapidly switching among multiple tasks. In this approach, each task gets exclusive access to the resource for a limited time period.
- **Space Sharing** — Dividing the available resource so that each task receives a portion of the total space simultaneously.

1.2 Fundamentals of CPU Scheduling

The primary goal of CPU scheduling is to create the illusion that each thread has exclusive use of the processor, while in reality, the CPU is being shared among multiple threads. This illusion is maintained through efficient time sharing of the CPU resource.

1.2.1 Thread Types and Scheduling

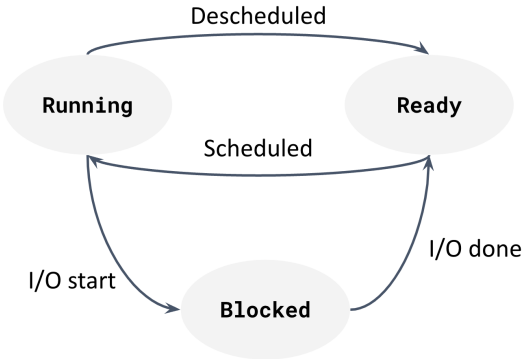
Threads can be categorized based on their operation patterns:

- **CPU-bound threads**: Perform computations with minimal I/O calls (e.g., calculating Fibonacci numbers)
- **I/O-bound threads**: Frequently make I/O calls (e.g., reading from disk, waiting for network)

1.2.2 Thread States

Thread states represent the different operational conditions a thread can be in during its lifecycle.

- **Running**: The thread is currently being executed by the CPU.
- **Ready**: The thread is waiting to be executed by the CPU.
- **Blocked**: The thread is waiting for an event to occur (e.g., I/O completion, message arrival).



1.2.3 Role of the Operating System Scheduler

The operating system's scheduler is responsible for:

- Maintaining a list of all threads in the system
- Tracking each thread's state (running, ready, blocked, etc.)
- Selecting which thread to run next according to a defined scheduling policy
- Managing context switches to give each thread its turn on the CPU

1.2.4 Reasons for Thread Scheduling

The operating system may need to schedule a new thread for various reasons:

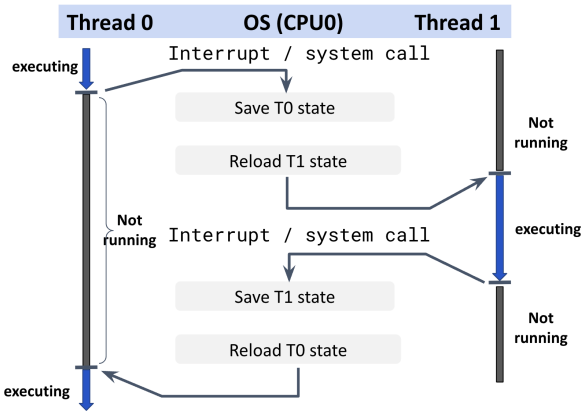
- The current thread has completed execution or terminated (e.g., due to invalid operations)
- The thread has made a system call (e.g., I/O operation) and must wait for its completion
- The OS scheduler has determined another thread should run (e.g., time slice expired)
- Other threads with higher priority are present in the ready queue

1.2.5 Context Switching

Context switching is the process of saving the state of a currently running thread and loading the state of another thread. This mechanism enables time-sharing of the CPU resource.

The OS performs the following operations during a context switch:

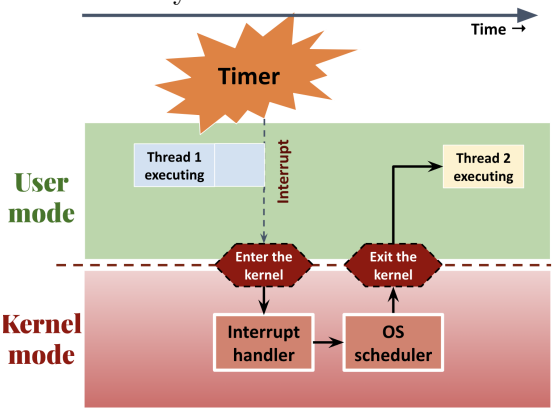
1. Saves the running thread's execution state (registers, program counter, etc.) in memory
2. Selects the next thread to run according to the scheduling policy
3. Restores the execution state of the selected thread
4. Passes control to the thread using a return-from-trap mechanism



1.2.6 Handling Misbehaving Threads

A critical challenge in operating systems is managing threads that may:

- Refuse to give up CPU control on their own
- Run in infinite loops without performing I/O operations
- Attempt to monopolize system resources



To address this challenge, operating systems implement preemptive scheduling using hardware timer interrupts. The process works as follows:

1. The OS sets a hardware timer before scheduling a thread
2. When the timer expires, the CPU is interrupted
3. The current thread is suspended and the system switches to kernel mode
4. The interrupt handler invokes the OS scheduler
5. The scheduler selects the next thread and performs a context switch

This mechanism ensures that no single thread can monopolize the CPU indefinitely, maintaining fairness in resource allocation.

1.3 Scheduling Policies

The scheduling policy is a key component of the operating system that determines which thread should run next. When a system has multiple threads competing for CPU time, the policy establishes the order in which threads execute.

1.3.1 Scheduling Metrics

To evaluate and compare different scheduling policies, we use performance metrics that quantify system behavior. Two fundamental metrics are:

1. **CPU Utilization**: The fraction of time the CPU is executing thread code.
   - **Goal**: Maximize CPU utilization (keep the CPU as busy as possible)
   - Measured as a percentage from 0% (idle) to 100% (fully utilized)
2. **Turnaround Time**: The total time from a thread's arrival to its completion.
   - **Goal**: Minimize turnaround time for better system responsiveness
   - Mathematical definition: $T_{turnaround} = T_{completion} - T_{arrival}$

1.3.2 First In, First Out (FIFO)

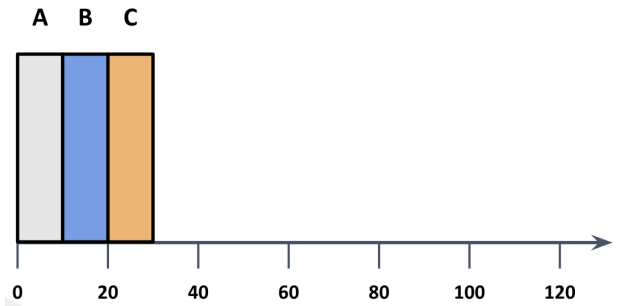
FIFO (also known as First Come, First Served) is the simplest scheduling algorithm where threads are executed in the order they arrive in the ready queue.

FIFO with Equal Run Times

Consider a scenario with three threads (A, B, C) that arrive simultaneously and each requires 10 seconds of CPU time

Assumptions:

- Each thread runs for the same time (10s)
- All threads arrive at the same time ($T_{arrival} = 0$)
- Each thread runs to completion
- Run-time of threads is known in advance



Calculations:

$$T_{arrival} = 0$$
$$T_{completion}(A) = 10$$
$$T_{completion}(B) = 20$$
$$T_{completion}(C) = 30$$

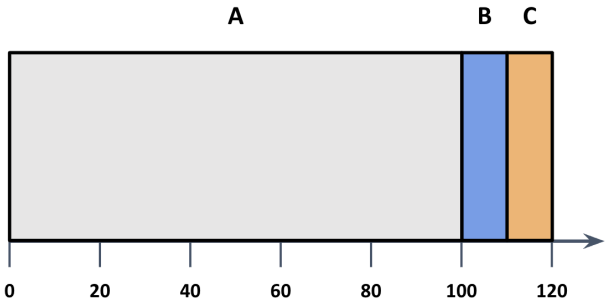
Average turnaround time $= \frac{10+20+30}{3} = 20$ seconds

FIFO with Varied Run Times

Now consider a scenario where thread A requires much more CPU time than the others:

Assumptions:

- Threads have different run times
(A: 100s, B: 10s, C: 10s)
- All threads arrive at the same time ($T_{arrival} = 0$)
- Each thread runs to completion
- Run-time of threads is known in advance



Calculations

$$T_{arrival} = 0$$
$$T_{completion}(A) = 100$$
$$T_{completion}(B) = 110$$
$$T_{completion}(C) = 120$$

Average turnaround time $= \frac{100+110+120}{3} = 110$ seconds

In FIFO scheduling, long-running threads can significantly delay shorter threads, causing poor average turnaround times. This is known as the "convoy effect" and represents a major limitation of the FIFO scheduling policy

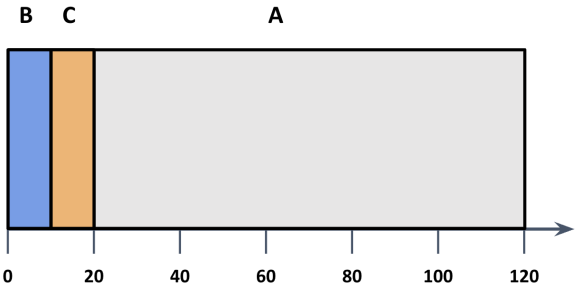
basically, the classic - someone in a supermarket queue is buying a lot of stuff and you're waiting for them to finish even though you have less items than them

1.3.3 Shortest Job First (SJF)

Choose ready threads with shortest running time

Assumptions:

Assume 3 threads (A, B, C): A runs for 100 seconds, while B and C run 10 seconds



Calculations:

$$T_{arrival} = 0$$
$$T_{completion}(A) = 120$$
$$T_{completion}(B) = 10$$
$$T_{completion}(C) = 20$$

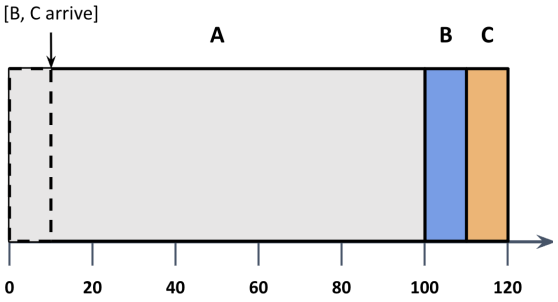
Average turnaround time $= \frac{120+10+20}{3} = 50$ seconds

Turnaround time improves by almost 50%

Issue with SJF

A runs for 100 seconds, while B and C run 10 seconds **Assumptions:**

- Threads do not need to run for same time
- Threads do not need to arrive at same time
- Each thread runs to completion
- Run-time of threads is known



Calculations:

$$T_{arrival}(A) = 0$$
$$T_{arrival}(B) = T_{arrival}(C) = 10$$
$$T_{completion}(A) = 100$$
$$T_{completion}(B) = 110$$
$$T_{completion}(C) = 120$$

Average turnaround time $= \frac{100+(110-10)+(120-10)}{3} = 103.3$

Remark: Long running threads cannot be interrupted, leading to convoy effect

1.3.4 Polite vs. forced scheduling

Non-preemptive Scheduling:

- Previous schedulers (FIFO, SJF) are non-preemptive
- Only switch to other threads once the current thread finishes its whole execution (run-to-completion)
- OS has no control on a thread's completion time

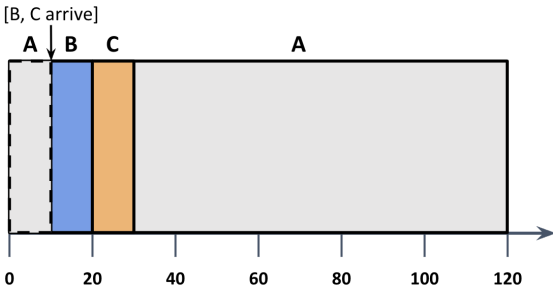
Preemptive Scheduling:

- Stops the execution of the current thread and switches to other ready thread forcibly
- OS avoids CPU monopolization and maintains control (thread create/destroy, timer interrupts)
- **This removes the assumption that each job runs to completion**

1.3.5 Shortest Time to Completion First (STCF)

STCF extends the SJF by adding preemption
Assumptions:

- Any time a new thread is created:
- STCF scheduler determines which of the remaining jobs (including new job) has the least time left
- STCF then schedules the shortest job first



Calculations:

$$\begin{aligned} T_{arrival}(A) &= 0 \\ T_{arrival}(B) &= T_{arrival}(C) = 10 \\ T_{turnaround}(A) &= 120 \\ T_{turnaround}(B) &= (20 - 10) = 10 \\ T_{turnaround}(C) &= (30 - 10) = 20 \end{aligned}$$

Average turnaround time $= \frac{120+10+20}{3} = 50$ seconds
Remark: Reschedule when new threads arrive, prioritize short running threads

1.3.6 New Metric - Response Time

Previous metrics:

- Focused only on turnaround time (i.e., completing the threads' execution as fast as possible)
- Turnaround time is important for batch jobs (non-interactive tasks)

New metric:

- Response time became equally important
- Defined as how long it takes until a thread is scheduled for the first time

STCF with Response Time

For example, for the STCF
Response time: Time from when the job arrives in the system to the first time it is scheduled:
$T_{response} = T_{firstrun} - T_{arrival}$

Calculations:

$$\begin{aligned} T_{arrival}(A) &= 0 \\ T_{arrival}(B) &= T_{arrival}(C) = 10 \\ T_{response}(A) &= (0 - 0) = 0 \\ T_{response}(B) &= (10 - 10) = 0 \\ T_{response}(C) &= (20 - 10) = 10 \end{aligned}$$

Average response time $= \frac{0+0+10}{3} = 3.3$ seconds

STCF is still not perfect

Prior scheduling policies are not good for response time:

Consider 3 jobs arrive at T=0 with same running time, the third job has to wait for the previous two jobs before getting scheduled!
This is great for turnaround time, but bad for interactivity.
Another way to think: typing on a keyboard and waiting **seconds** for the character to show up on the screen...

1.3.7 Round Robin Scheduling

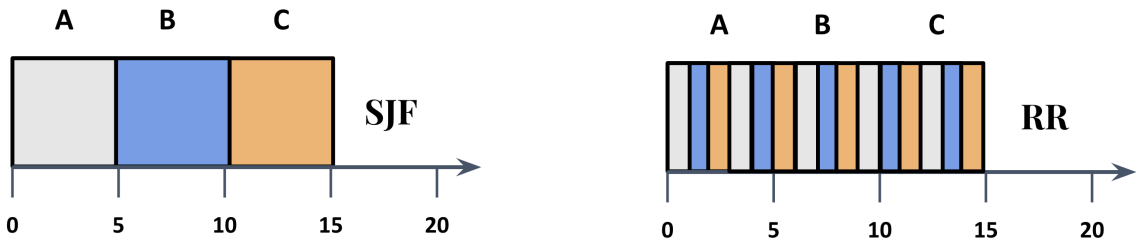
Instead of running threads to completion, RR schedules a thread for a fixed interval (or a time-slice) and then switches to the next thread Alternate ready threads every fixed-length time slice.

Round Robin vs STCF

Threads A, B, and C run for 5 seconds each and arrive at time 0, $T_{arrival} = 0$

- In SJF, each thread runs to completion before running another
- Average response time = (0 + 5 + 10) / 3 = 5
- Average turnaround time = (5 + 10 + 15) / 3 = 10

- In RR, time slice is 1 second and it will run each thread every second
- Average response time = (0 + 1 + 2) / 3 = 1
- Average turnaround time = (13 + 14 + 15) / 3 = 14



Remark: Responsiveness increases turnaround (for equally long running threads)

1.3.8 IO Request Scheduling

In operating systems, we need to consider how IO operations affect CPU scheduling. Let's examine a simple example:
Assume we have two threads (A and B):

- Thread A needs 40 ms of CPU time and makes 3 IO requests of 10 ms each
- Thread B needs 40 ms of CPU time and makes no IO requests
- Thread A issues an IO request every 10 ms of CPU time

When thread A issues IO requests, the CPU is not utilized efficiently if the scheduler doesn't account for this behavior.

Scheduling with IO Awareness

Question: For a Shortest Time-to-Completion First (STCF) scheduler, how should we handle thread A's 4 sub-jobs (10 ms each) versus thread B's single 40 ms job?

Answer: A better approach is for the scheduler to account for both IO and CPU time to improve resource utilization:

- Treat each of A's 10 ms segments as independent sub-jobs
- Consider B as a whole 40 ms job
- STCF chooses A's first sub-job (10 ms) and then schedules B
- When A's next sub-job becomes ready, the scheduler preempts B
- While A waits for IO completion, B can run on the CPU
- This leads to better CPU utilization through overlapping CPU and IO operations

1.3.9 Multi-level Queue Scheduling (MLFQ)

The Scheduling Challenge

A general-purpose scheduler must support different types of threads:

- **Batch-processing threads:** Long-running background processes that need lots of CPU time but where response time is not critical
- **Interactive threads:** Foreground processes that require low latency and run in short bursts (need frequent but small amounts of CPU time)

MLFQ Approach

MLFQ aims to optimize for both types of threads:

- First, it tries to optimize turnaround time (important for batch threads)
- Then, it minimizes response time for better interactivity

The challenge is that the scheduler doesn't know the total runtime of a thread in advance (which would be needed for SJF or STCF). The insight of MLFQ is to use past behavior as a predictor for future behavior.

MLFQ Implementation

MLFQ uses multiple levels of Round Robin queues:

- Each level has a different priority
- Higher levels preempt lower levels
- Higher levels have shorter time slices
- Lower levels have longer time slices

MLFQ Rules

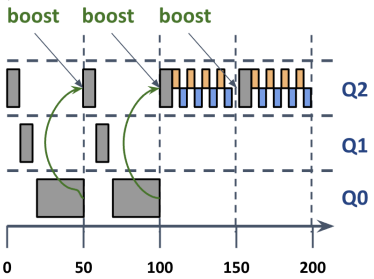
The scheduler follows these rules to adjust priorities dynamically:

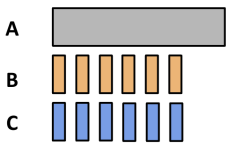
1. If priority(A) > priority(B), then A runs
2. If priority(A) = priority(B), then A and B run in Round Robin
3. All threads start at the highest priority level
4. If a thread uses its entire time slice, the scheduler lowers its priority
5. Periodically, all threads are moved back to the highest priority queue (called "priority boosting")

The periodic priority boosting (rule 5) prevents starvation, which could happen when multiple IO-bound threads at high priority might prevent a CPU-bound thread at low priority from ever running.

Example 1.3.9.1 (MLFQ in Action)**.** Let's see how MLFQ works with a concrete example. Assume we have a system with three priority queues (Q2, Q1, Q0), where Q2 is the highest priority:

- Each queue uses Round Robin scheduling
- Time slice for Q2 = 10 ms, Q1 = 10 ms, Q0 = 30 ms
- Priority boost happens every 50 ms





Phase 1: Single CPU-bound process

1. Process A begins in Q2 (highest priority)
2. A uses its entire 10 ms time slice in Q2 and gets demoted to Q1
3. A uses its entire 10 ms time slice in Q1 and gets demoted to Q0
4. A runs in Q0 for 30 ms until the priority boost occurs
5. After the boost, A returns to Q2 and the cycle repeats

Phase 2: Interactive processes arrive

1. After A has been running for 100 ms, processes B and C join the system
2. All three processes (A, B, C) are now in Q2
3. A runs for 10 ms and gets demoted to Q1
4. Now B runs for a short time but issues an IO request before using its full time slice
5. C also runs briefly before issuing an IO request

6. Since B and C issue frequent IO requests, they never use their full time slices
7. B and C therefore remain in Q2, while A continues to be demoted

Result: Interactive processes (B and C) get quick response times by staying in the high-priority queue, while the CPU-intensive process (A) is given fair access through priority boosting. This demonstrates how MLFQ adapts to different process types without requiring advance knowledge of their behavior.

Good Luck for the midterm !