Process Scheduling & Inter-Process Communication

Advanced Operating Systems — Week 3

Hunter College, CUNY

Faye

February 2026

Contents

Part I: Process Scheduling

1 Scheduling Basics

1.1 Why Scheduling?

In a multiprogramming environment the CPU is a shared resource. When multiple processes are ready to execute, the operating system must decide *which* process runs next and for *how long*. The component that makes this decision is the **CPU scheduler** (also called the short-term scheduler).

Goals of a scheduler include:

- Maximize CPU utilization — keep the CPU as busy as possible.
- Maximize throughput — complete as many processes per unit time as possible.
- Minimize turnaround time, waiting time, and response time.
- Ensure fairness — every process receives a reasonable share of CPU time.

1.2 Scheduler Hierarchy

Long-term scheduler (job scheduler)
Controls the *degree of multiprogramming* by admitting processes from the job pool into the ready queue. It runs infrequently (seconds to minutes) and balances I/O-bound vs. CPU-bound processes.

Short-term scheduler (CPU scheduler)
Selects the next process from the ready queue and allocates the CPU. It runs very frequently (every few milliseconds) and must therefore be extremely fast.

Medium-term scheduler
Performs *swapping*: temporarily removes processes from memory to reduce the degree of multiprogramming, then reintroduces them later. This helps manage memory pressure.

1.3 Dispatcher

The **dispatcher** is the module that gives control of the CPU to the process selected by the short-term scheduler. Its responsibilities include:

- Context switching (saving/restoring registers, program counter, etc.).
- Switching to user mode.
- Jumping to the appropriate location in the user program.

The time the dispatcher takes to stop one process and start another is called **dispatch latency**. On modern hardware this is typically **1–10 μs**. Minimizing dispatch latency is critical because the dispatcher is invoked on every context switch.

1.4 Non-preemptive vs. Preemptive Scheduling

- **Non-preemptive (cooperative):** Once a process is allocated the CPU, it keeps the CPU until it voluntarily releases it (terminates or performs I/O). Simpler to implement but can lead to poor responsiveness.
- **Preemptive:** The OS can forcibly remove the CPU from a running process (e.g., when a timer interrupt fires or a higher-priority process arrives). Most modern OSes (Linux, Windows, macOS) use preemptive scheduling. Introduces complexity around shared data and synchronization.

2 Scheduling Metrics

Core Scheduling Metrics

$$\text{Turnaround Time} = \text{Completion Time} - \text{Arrival Time} \tag{1}$$

$$\text{Response Time} = \text{First Run Time} - \text{Arrival Time} \tag{2}$$

$$\text{Waiting Time} = \text{Turnaround Time} - \text{Burst Time} \tag{3}$$

$$\text{Throughput} = \frac{\text{Number of Processes Completed}}{\text{Total Time}} \tag{4}$$

$$\text{CPU Utilization} = \frac{\text{Time CPU is Busy}}{\text{Total Time}} \times 100\% \tag{5}$$

- **Turnaround time** measures the total time from submission to completion.
- **Response time** matters for interactive systems — how quickly a process starts producing output.
- **Waiting time** is the total time spent in the ready queue (not executing or doing I/O).
- **Throughput** indicates system efficiency.
- **CPU utilization** in a well-loaded system should be 40%–90%.

3 First-Come, First-Served (FCFS)

FCFS is the simplest scheduling algorithm. Processes are executed in the order they arrive (FIFO queue). It is **non-preemptive**.

Pros: Simple to implement and understand.

Cons: Suffers from the **convoy effect** — a long CPU-bound process can block many short processes behind it, inflating average waiting time dramatically.

FCFS Example

Three processes arrive at time 0 with the following burst times:

Process	Burst Time
P1	24
P2	3
P3	3

Execution order: P1 → P2 → P3

Waiting times:

- P1: 0
- P2: 24
- P3: $24 + 3 = 27$

Average waiting time $= \dfrac{0 + 24 + 27}{3} = 17$

Notice how P1's long burst penalizes P2 and P3. If the order were P2, P3, P1, the average waiting time would be $\frac{0+3+6}{3} = 3$.

4 Shortest Job First (SJF)

SJF selects the process with the **smallest next CPU burst**. In its non-preemptive form, once a process starts, it runs to completion.

SJF is **provably optimal** for minimizing *average turnaround time* among non-preemptive algorithms.

Challenge: We cannot know the exact length of the next CPU burst in advance. The OS must *predict* it.

4.1 Exponential Averaging

The predicted next burst τ_{n+1} is estimated from the actual previous burst t_n and the previous prediction τ_n:

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\, \tau_n, \qquad 0 \le \alpha \le 1$$

- $\alpha = 0$: prediction never changes (ignores history).
- $\alpha = 1$: prediction = last actual burst (no smoothing).
- Common choice: $\alpha = 0.5$.

4.2 Starvation and Aging

SJF can cause **starvation**: a long process may wait indefinitely if short processes keep arriving. The solution is **aging** — gradually increasing the priority of waiting processes over time so that every process eventually gets to run.

SJF Example (Non-preemptive)

Processes arriving at time 0:

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

SJF order: P4(3) → P1(6) → P3(7) → P2(8)
Waiting times: P4 = 0, P1 = 3, P3 = 9, P2 = 16.
Average waiting time $= \dfrac{0+3+9+16}{4} = 7.0$
Compare with FCFS order (P1, P2, P3, P4): average waiting $= \frac{0+6+14+21}{4} = 10.25$.

5 Shortest Remaining Time First (SRTF)

SRTF is the **preemptive** version of SJF. Whenever a new process arrives, the scheduler compares its burst time with the *remaining* time of the currently running process. If the new process has a shorter remaining time, the running process is preempted.

- SRTF is optimal for minimizing **average waiting time**.
- Like SJF, it requires burst-time prediction and can cause starvation.
- Higher overhead due to frequent preemptions and context switches.

SRTF Example

Process	Arrival	Burst
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Timeline:

- $t = 0$: P1 starts (remaining 8).
- $t = 1$: P2 arrives (burst 4 $<$ P1 remaining 7). P2 preempts P1.
- $t = 2$: P3 arrives (burst 9 $>$ P2 remaining 3). P2 continues.
- $t = 3$: P4 arrives (burst 5 $>$ P2 remaining 2). P2 continues.
- $t = 5$: P2 finishes. Remaining: P1(7), P4(5), P3(9). P4 runs.
- $t = 10$: P4 finishes. P1 runs (remaining 7).
- $t = 17$: P1 finishes. P3 runs.
- $t = 26$: P3 finishes.

Average waiting time $= \dfrac{(17-0-8)+(5-1-4)+(26-2-9)+(10-3-5)}{4} = \dfrac{9+0+15+2}{4} = 6.5$

6 Round Robin (RR)

Round Robin assigns each process a fixed **time quantum** q (typically 10–100 ms). Processes are kept in a circular FIFO queue. Each process runs for at most q units; if it does not finish, it is preempted and placed at the back of the queue.

Round Robin Example (q

All processes arrive at time 0:

Process	Burst Time
P1	24
P2	3
P3	3

Execution timeline:

- [0, 4): P1 runs (remaining 20).
- [4, 7): P2 runs (finishes, burst was 3).
- [7, 10): P3 runs (finishes, burst was 3).
- [10, 14): P1 runs (remaining 16).
- [14, 18): P1 runs (remaining 12).
- ... P1 continues in quantum-sized chunks until $t = 30$.

Waiting times: P1 $= 30 - 24 = 6$, P2 $= 4$, P3 $= 7$.

Average waiting time $= \dfrac{6+4+7}{3} \approx 5.67$

Tradeoffs:

- **Small q:** Better responsiveness but more context switches (overhead).
- **Large q:** Approaches FCFS behavior.
- Rule of thumb: q should be large enough that 80% of CPU bursts finish within one quantum.

- RR generally has higher average turnaround than SJF but much better response time.

7 Priority Scheduling

Each process is assigned a **priority** (integer). The CPU is allocated to the process with the highest priority (lower number = higher priority in many systems). Can be preemptive or non-preemptive.

7.1 Priority Sources

- **Internal:** Based on measurable quantities — time limits, memory requirements, ratio of I/O to CPU bursts, open file count.
- **External:** Set by users or administrators — process importance, department, political/-financial factors.

7.2 Starvation and Aging

Like SJF, priority scheduling can cause **starvation** of low-priority processes. **Aging** solves this by gradually increasing the priority of processes that have been waiting for a long time. For example, increase priority by 1 every 15 minutes.

8 Multi-Level Feedback Queue (MLFQ)

MLFQ is the most sophisticated general-purpose scheduling algorithm. It uses multiple queues at different priority levels and adjusts process priority based on observed behavior.

The 5 Rules of MLFQ

1. If Priority(A) > Priority(B), then A runs (B does not).
2. If Priority(A) = Priority(B), then A and B run in Round Robin using the time quantum of that queue.
3. When a job enters the system, it is placed at the **highest priority** (topmost queue).
4. Once a job uses up its total time allotment at a given level (regardless of how many times it has given up the CPU), its priority is **reduced** (moved to the next lower queue).
5. After some time period S (**priority boost**), move *all* jobs to the topmost queue.

How MLFQ learns behavior:

- **Interactive / I/O-bound** processes frequently relinquish the CPU before their quantum expires ⇒ they stay at high priority.
- **CPU-bound** processes consume their full quantum ⇒ they are gradually demoted to lower-priority queues with longer time quanta.
- **Rule 5** (priority boost) prevents starvation and handles processes whose behavior changes over time.

9 Linux Completely Fair Scheduler (CFS)

The Linux CFS (default since kernel 2.6.23) aims to give each process a *fair* share of the CPU proportional to its weight.

- **Virtual runtime (`vruntime`):** Each process tracks how much "virtual" CPU time it has received. CFS always picks the process with the *smallest* `vruntime`.

- **Red-black tree:** Runnable processes are stored in a self-balancing red-black tree keyed by `vruntime`. Selecting the next process (leftmost node) is $O(1)$ (cached); insertion and deletion are $O(\log n)$.
- **Nice values:** Range from -20 (highest priority) to $+19$ (lowest priority). Default is 0. Lower nice values cause `vruntime` to accumulate more slowly, so the process gets scheduled more often.
- **Target latency:** The scheduler period within which every runnable process should get at least one turn (typically 6–24 ms depending on the number of runnable tasks).
- **Minimum granularity:** A floor on the time slice to limit context-switch overhead (typically 0.75 ms).

CFS Replaced by EEVDF in Linux 6.6

Starting with Linux kernel 6.6 (October 2023), CFS has been replaced by the **EEVDF** (Earliest Eligible Virtual Deadline First) scheduler, which provides better latency guarantees while retaining the fairness properties of CFS.

10 Algorithm Comparison

Algorithm	Preemptive?	Optimal?	Starvation?	Complexity
FCFS	No	No	No	$O(1)$
SJF	No	Yes (avg turnaround)	Yes	$O(n)$
SRTF	Yes	Yes (avg waiting)	Yes	$O(n)$
Round Robin	Yes	No	No	$O(1)$
Priority	Both	No	Yes	$O(n)$
MLFQ	Yes	No (adaptive)	No (with boost)	$O(1)$ per queue
Linux CFS	Yes	No (fair)	No	$O(\log n)$

Part II: Inter-Process Communication

11 IPC Overview

Processes may need to cooperate: share data, coordinate actions, or signal events. Since each process has its own isolated address space, the kernel must provide mechanisms for **Inter-Process Communication (IPC)**.

Why IPC?

- Information sharing (e.g., a database accessed by multiple applications).
- Computation speedup via parallelism (divide a task among cooperating processes).
- Modularity (microservice / pipeline architectures).
- Convenience (a user may run multiple tasks that need to interact).

Two Fundamental IPC Models

Message Passing

Processes communicate by exchanging messages through the kernel. No shared address space is required. The kernel copies data between address spaces. Easier to use, especially across networked systems, but involves system-call overhead on every send/receive.

Shared Memory

A region of memory is mapped into the address space of multiple processes. After initial setup (which involves system calls), processes can read/write the shared region directly without kernel intervention. Very fast for large data transfers, but requires explicit synchronization (mutexes, semaphores) to prevent race conditions.

12 Pipes

A **pipe** is the simplest IPC mechanism in Unix. It provides a **unidirectional** byte stream between two related processes (typically parent and child).

- Created with the `pipe(int fd[2])` system call.
- `fd[0]` is the **read end**; `fd[1]` is the **write end**.
- Data written to `fd[1]` can be read from `fd[0]`.
- Pipes are buffered in kernel memory (typically 64 KB on Linux).
- When all writers close `fd[1]`, the reader gets EOF.

Listing 1: Pipe between parent and child

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main(void) {
    int fd[2];
    pipe(fd);  // create the pipe

    pid_t pid = fork();
    if (pid == 0) {
        // Child: read from pipe
        close(fd[1]);           // close unused write end
        char buf[128];
        int n = read(fd[0], buf, sizeof(buf) - 1);
        buf[n] = '\0';
        printf("Child received: %s\n", buf);
        close(fd[0]);
    } else {
        // Parent: write to pipe
        close(fd[0]);           // close unused read end
        const char *msg = "Hello from parent!";
        write(fd[1], msg, strlen(msg));
        close(fd[1]);           // signal EOF to child
    }
    return 0;
}
```

13 Named Pipes (FIFOs)

A **named pipe** (FIFO) is like a regular pipe but has a name in the filesystem, created with `mkfifo()`.

- Appears as a special file (e.g., `/tmp/myfifo`).
- **Unrelated processes** can communicate through it (they just need to know the path).
- Still unidirectional in practice (use two FIFOs for bidirectional communication).
- Persists until explicitly deleted (`unlink`).

Listing 2: Creating and using a named pipe

```c
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

// Writer process
int main(void) {
    mkfifo("/tmp/myfifo", 0666);
    int fd = open("/tmp/myfifo", O_WRONLY);
    write(fd, "Hello FIFO!", 11);
    close(fd);
    unlink("/tmp/myfifo");
    return 0;
}
```

14 Message Queues

POSIX message queues provide a mechanism for processes to exchange discrete **messages** (not byte streams) through the kernel.

- Created/opened with `mq_open()`.
- Messages sent with `mq_send()` and received with `mq_receive()`.
- Each message has a **priority**; higher-priority messages are delivered first.
- Messages persist in the queue until consumed (even if the sender exits).
- Queue has configurable maximum message count and message size.

Listing 3: POSIX message queue example

```c
#include <mqueue.h>
#include <string.h>
#include <stdio.h>

int main(void) {
    struct mq_attr attr = { .mq_maxmsg = 10, .mq_msgsize = 256 };
    mqd_t mq = mq_open("/myqueue", O_CREAT | O_RDWR, 0644, &attr);

    // Send
    const char *msg = "Hello MQ!";
    mq_send(mq, msg, strlen(msg) + 1, /* priority */ 0);

    // Receive
    char buf[256];
    unsigned int prio;
    mq_receive(mq, buf, sizeof(buf), &prio);
    printf("Received: %s (priority %u)\n", buf, prio);

    mq_close(mq);
```

```
    mq_unlink("/myqueue");
    return 0;
}
```

15 Shared Memory

Shared memory is the **fastest** IPC mechanism because, after the initial setup, data transfer does not involve system calls — processes read and write the shared region directly.

- `shm_open()` creates or opens a shared memory object.
- `ftruncate()` sets the size.
- `mmap()` maps it into the process's address space.
- Multiple processes can map the same object.
- **Requires synchronization** (mutexes, semaphores) to prevent race conditions.

Listing 4: POSIX shared memory

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>

int main(void) {
    const char *name = "/my_shm";
    const int SIZE = 4096;

    // Create shared memory
    int fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    ftruncate(fd, SIZE);

    // Map into address space
    char *ptr = mmap(NULL, SIZE, PROT_READ | PROT_WRITE,
                     MAP_SHARED, fd, 0);

    // Write data
    strcpy(ptr, "Hello from shared memory!");

    // Another process can shm_open + mmap the same name to read it
    printf("Wrote: %s\n", ptr);

    munmap(ptr, SIZE);
    shm_unlink(name);
    return 0;
}
```

16 Signals

Signals are software interrupts delivered to a process to notify it of an event. They are the oldest IPC mechanism in Unix.

16.1 Key Signals

Signal	Number	Description	Catchable?
SIGINT	2	Interrupt from keyboard (Ctrl+C)	Yes
SIGKILL	9	Unconditional termination	**No**
SIGSEGV	11	Segmentation fault	Yes
SIGTERM	15	Polite termination request	Yes
SIGCHLD	17	Child process stopped or terminated	Yes

16.2 Signal Handler Example

Listing 5: Custom signal handler

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void handler(int sig) {
    printf("Caught signal %d (SIGINT)\n", sig);
}

int main(void) {
    struct sigaction sa;
    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGINT, &sa, NULL);

    printf("Press Ctrl+C to trigger the handler...\n");
    while (1)
        pause();  // wait for signals
    return 0;
}
```

17 IPC Mechanism Comparison

Mechanism	Direction	Related Only?	Notes
Pipe	Unidirectional	Yes (parent–child)	Byte stream, simple, kernel-buffered
Named Pipe	Unidirectional	No (filesystem name)	Like pipe but accessible to unrelated processes
Message Queue	Bidirectional	No	Discrete messages with priority, kernel-managed
Shared Memory	Bidirectional	No	Fastest — direct memory access, needs synchronization
Signals	Unidirectional	No	Asynchronous notification, no data payload (only signal number)
Sockets	Bidirectional	No	Works across network, most flexible, higher overhead

18 Key Takeaways

Key Takeaways — Scheduling & IPC

1. **FCFS** is simple but suffers from the convoy effect. **SJF/SRTF** are optimal but require burst prediction and can starve long jobs.
2. **Round Robin** provides good response time for interactive systems; the time quantum is a key tuning parameter.
3. **MLFQ** adapts to process behavior by promoting I/O-bound processes and demoting CPU-bound ones. Priority boosts prevent starvation.
4. **Linux CFS** achieves fairness via virtual runtime and a red-black tree. Nice values weight CPU shares.
5. **Message passing** is simpler and safer (kernel mediates); **shared memory** is faster but requires explicit synchronization.
6. **Pipes** are for related processes; **named pipes**, **message queues**, and **shared memory** work between unrelated processes.
7. **Signals** are asynchronous notifications, not data-transfer mechanisms. `SIGKILL` and `SIGSTOP` cannot be caught.