

UNIT 3

Process Management

(15 Hrs) — Operating System (CACS251)

3.1 Process Concepts (3 Hrs)

3.1.1 Definition of a Process

A **process** is a **program in execution**. A program is a passive entity (an executable file stored on disk); a process is an active entity with a program counter, registers, stack, data section and heap. Two processes may run the same program (e.g., two users running a text editor) yet they are separate processes with separate data.

Program	Process
Passive — a file on disk	Active — an instance of a program being executed
Exists until deleted	Exists only while executing
No resources allocated	Uses CPU, memory, files and I/O devices
Single copy on disk	Many processes can be created from one program

3.1.2 The Process Model

In the process model all the runnable software on the computer is organised into a number of **sequential processes**. Conceptually each process has its own virtual CPU. In reality, the real CPU switches back and forth from process to process (multiprogramming). This rapid switching is called a **context switch**. At any instant only one process is running on a single-CPU machine, but over a period all processes make progress, giving the illusion of parallelism (pseudo-parallelism).

A process in memory has four parts: the **text** section (program code), the **data** section (global variables), the **heap** (dynamically allocated memory) and the **stack** (function parameters, return addresses and local variables).

3.1.3 Process States and State Transitions

As a process executes it changes state. A process may be in one of the following states:

State	Meaning
New	The process is being created.
Ready	The process is in memory, waiting to be assigned to the CPU.
Running	Instructions are being executed on the CPU.
Waiting (Blocked)	The process is waiting for some event (I/O completion, a signal, a resource).
Terminated	The process has finished execution.

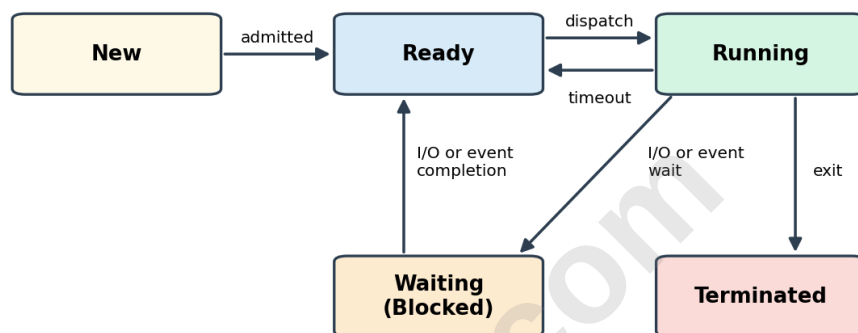


Figure 3.1 — Five-state process transition diagram

Transition	Cause
New → Ready	OS admits the process into the pool of executable processes.
Ready → Running	The scheduler dispatches the process to the CPU.
Running → Ready	Time slice expires (timeout) or a higher-priority process arrives (preemption).
Running → Waiting	Process requests I/O or waits for an event.
Waiting → Ready	The awaited I/O or event completes.
Running → Terminated	Process finishes or is killed.

3.1.4 The Process Control Block (PCB)

The OS represents each process by a **Process Control Block** (also called a task control block). It is a data structure containing all the information needed to manage the process and to restart it correctly after it has been interrupted.

PCB field	Contents
Process ID (PID)	Unique identifier of the process

PCB field	Contents
Process state	New, ready, running, waiting, terminated
Program counter	Address of the next instruction to execute
CPU registers	Accumulators, index registers, stack pointers — saved on interrupt
CPU-scheduling information	Priority, pointers to scheduling queues
Memory-management information	Base/limit registers, page tables, segment tables
Accounting information	CPU time used, time limits, job numbers
I/O status information	List of open files, allocated I/O devices

Note: During a context switch the OS saves the state of the old process into its PCB and loads the saved state from the PCB of the new process. Context-switch time is pure overhead.

3.1.5 Operations on Processes

(a) Process Creation

A process creates new processes by a **create-process system call** (fork in UNIX, CreateProcess in Windows). Events that cause creation: system initialisation, execution of a process-creation system call by a running process, a user request to start a new program, and initiation of a batch job. The creating process is the **parent**; the new ones are its **children**. The parent may run concurrently with the child or wait until it terminates; the child may be a duplicate of the parent (fork) or load a new program (exec).

(b) Process Termination

A process terminates when it executes its last statement and asks the OS to delete it (exit). Reasons: **normal exit** (voluntary), **error exit** (voluntary), **fatal error** (involuntary — illegal instruction, divide by zero) and **killed by another process** (involuntary). A parent may terminate a child if the child exceeds its resources or is no longer needed. The OS reclaims all resources of the terminated process.

(c) Process Hierarchies

In UNIX a process and all its descendants form a **process group** — a tree rooted at **init** (PID 1). If a parent terminates before its children, the children are adopted by init. In Windows there is no hierarchy: the parent holds a *handle* to the child, which it may pass on to another process.

(d) Implementation of Processes

The OS maintains a **process table** — an array of PCBs, one entry per process. When an interrupt occurs the hardware pushes the program counter and state on the stack; an assembly routine then saves the registers into the PCB, the interrupt-service routine runs, and finally the scheduler picks the process to resume.

3.1.6 Cooperating Processes

Processes executing concurrently may be **independent** (cannot affect or be affected by other processes) or **cooperating** (can affect or be affected by others, usually by sharing data). Reasons for cooperation: **information sharing**, **computation speed-up** (parallel sub-tasks), **modularity** and **convenience**. Cooperating processes need **inter-process communication (IPC)** — shared memory or message passing — and must be synchronised.

Example — Producer–Consumer problem

A **producer** process generates items (e.g., compiler producing assembly code) and a **consumer** process uses them (assembler). They share a bounded buffer. The producer must wait when the buffer is full; the consumer must wait when it is empty.

3.1.7 System Calls

System calls provide the interface between a running program and the operating system. Each call has a number; the library routine places parameters in registers, executes a **trap** instruction to switch to kernel mode, and the kernel dispatches the call.

Category	Windows	UNIX / Linux
Process management	CreateProcess(), ExitProcess(), WaitForSingleObject()	fork(), exec(), exit(), wait(), kill()
File management	CreateFile(), ReadFile(), WriteFile(), CloseHandle()	open(), read(), write(), close(), lseek()
Directory management	CreateDirectory(), RemoveDirectory(), SetCurrentDirectory()	mkdir(), rmdir(), link(), unlink(), chdir()
Device management	SetConsoleMode(), ReadConsole()	ioctl(), read(), write()
Communication	CreatePipe(), CreateFileMapping()	pipe(), shmget(), msgsnd(), socket()

```

pid = fork();           /* create a child process          */
if (pid == 0)           /* child              */
    execlp("/bin/ls", "ls", NULL); /* replace with new program */
else                    /* parent            */
    wait(NULL);         /* wait for child to finish */

```

3.2 Threads (1 Hr)

3.2.1 Definition of a Thread

A **thread** is a basic unit of CPU utilisation — a **lightweight process**. It comprises a thread ID, program counter, register set and stack. Threads belonging to the same process share the code section, data section and other OS resources (open files, signals). A traditional (heavyweight) process has a single thread of control.

3.2.2 Single-threaded and Multithreaded Processes

Item	Single-threaded process	Multithreaded process
Threads of control	One	Two or more
Code, data, files	Owned by the process	Shared by all threads
Registers, stack, PC	One set	One set <i>per thread</i>
If one activity blocks	Whole process blocks	Other threads can continue

3.2.3 Benefits of Multithreading

- **Responsiveness** — a program continues even if part of it is blocked (a browser loads a page while the user scrolls).
- **Resource sharing** — threads share memory and resources by default.
- **Economy** — creating and context-switching threads is much cheaper than for processes.
- **Scalability / utilisation of multiprocessor architectures** — threads can run truly in parallel on different cores.

3.2.4 Multithreading Models

Threads may be supported at the **user level** (by a thread library, without kernel knowledge) or at the **kernel level** (directly by the OS). A relationship must exist between user threads and kernel threads.

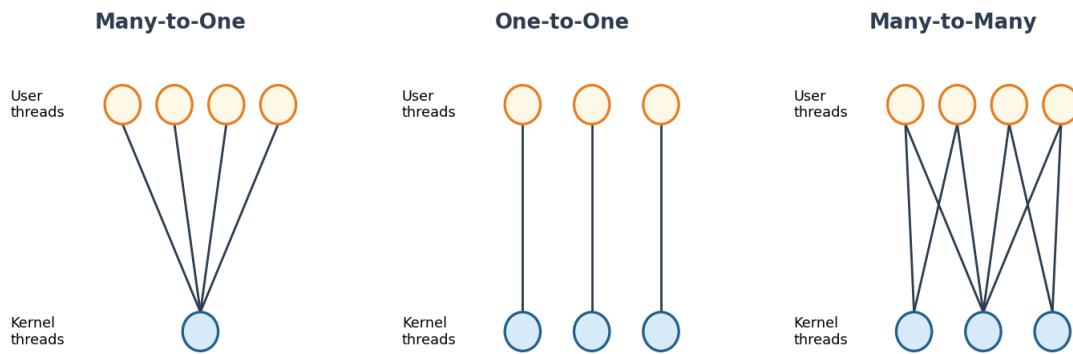


Figure 3.2 — Multithreading models

Model	Description	Advantage / Drawback
Many-to-One	Many user threads mapped to a single kernel thread	Efficient thread management in user space; but one blocking call blocks all threads and no parallelism on multiprocessors
One-to-One	Each user thread mapped to its own kernel thread (Windows, Linux)	More concurrency and no blocking of others; but creating many threads is costly, limits threads per process
Many-to-Many	Many user threads multiplexed onto a smaller or equal number of kernel threads	Developer can create as many user threads as needed; kernel threads run in parallel; more complex

3.3 Inter-Process Communication and Synchronization (6 Hrs)

3.3.1 Introduction

Processes frequently need to communicate with one another and must not get in each other's way. Three issues arise: (1) how one process passes information to another; (2) making sure two or more processes do not get into each other's way when engaging in critical activities; and (3) proper sequencing when dependencies are present (e.g., producer before consumer). The same issues apply to threads.

3.3.2 Race Condition

A **race condition** occurs when two or more processes access shared data at the same time and the final result depends on the **exact order** in which their instructions are interleaved.

Example — Print spooler race

A spooler directory has slots numbered 0, 1, 2 ... and shared variables **out** (next file to print) and **in** (next free slot). Suppose **in** = 7.

Process A reads **in** = 7 and is interrupted before writing. Process B reads **in** = 7, stores its file in slot 7 and sets **in** = 8. Process A resumes, writes its file into slot 7 (overwriting B's file!) and sets **in** = 8.

Result: B's file is never printed. The outcome depends on timing — a race condition.

A similar problem appears for **counter++** and **counter--** executed concurrently because each expands into three machine instructions (load, modify, store).

3.3.3 Critical Regions (Critical Sections)

The part of a program where shared memory is accessed is called the **critical region** (or critical section). If we ensure that no two processes are ever in their critical regions at the same time, race conditions are avoided.

Avoiding Critical Region Problems: Mutual Exclusion and Serializability

Mutual exclusion is a way of making sure that if one process is using a shared variable or file, the other processes are excluded from doing the same thing.

3.3.4 Conditions for a Good Solution (Mutual Exclusion Conditions)

#	Condition	Meaning
1	Mutual exclusion	No two processes may be simultaneously inside their critical regions.
2	Progress	No process running outside its critical region may block other processes.
3	Bounded waiting	No process should have to wait forever to enter its critical region (no starvation).
4	No assumptions	No assumptions may be made about speeds or the number of CPUs.

3.3.5 Proposals for Achieving Mutual Exclusion

(a) Disabling Interrupts

Each process disables all interrupts just after entering its critical region and re-enables them just before leaving. With interrupts off, the CPU cannot be switched to another process. *Drawbacks:* giving user processes power to disable interrupts is dangerous (if a process never re-enables them the system halts); it does not work on multiprocessors (other CPUs continue). It is, however, used inside the kernel itself.

(b) Lock Variables

A single shared variable **lock** is initially 0. A process wanting to enter tests the lock; if it is 0 it sets it to 1 and enters; if 1 it waits. *Flaw:* the test-and-set is not atomic — one process may read 0, get pre-empted before setting it to 1, and another process reads 0 too; both enter. The same race condition just moves to the lock variable.

```
while (lock != 0) ;    /* busy wait - spin */
lock = 1;              /* NOT atomic with the test above! */
    critical_region();
lock = 0;
```

(c) Strict Alternation

Two processes share an integer **turn** which tells whose turn it is to enter. Process 0 enters when $\text{turn} = 0$ and, after finishing, sets $\text{turn} = 1$; process 1 does the opposite. **Busy waiting** (spin lock) is used while waiting. *Flaw:* violates the **progress** condition — a slow process outside its critical region can block a fast one.

```
/* Process 0 */
while (TRUE) {
    while (turn != 0) ;
    critical_region();
    turn = 1;
    noncritical_region();
}

/* Process 1 */
while (TRUE) {
    while (turn != 1) ;
    critical_region();
    turn = 0;
    noncritical_region();
}
```

(d) Peterson's Solution

Peterson (1981) combined the ideas of turn and lock variables into a correct software solution for two processes (numbered 0 and 1). It uses an array **interested[2]** and a variable **turn**. Before entering, a process announces its interest and gives the turn to the other process; it enters only if the other is not interested or it is its own turn.

```

int turn;  int interested[2];

void enter_region(int process) {          /* process = 0 or 1 */
    int other = 1 - process;
    interested[process] = TRUE;          /* show interest    */
    turn = process;                      /* set the flag      */
    while (turn == process && interested[other] == TRUE) ; /* wait */
}
void leave_region(int process) {
    interested[process] = FALSE;          /* leaving          */
}

```

Peterson's solution satisfies mutual exclusion, progress and bounded waiting without special hardware, but it uses busy waiting and is only for two processes.

(e) The TSL Instruction (Hardware Support)

TSL RX, LOCK (Test and Set Lock) reads the contents of memory word LOCK into register RX and then stores a non-zero value at LOCK. The read and write are **indivisible (atomic)** — the memory bus is locked until the instruction finishes, so no other CPU can access the word. To use TSL a shared variable LOCK is used: when LOCK = 0 any process may set it to 1 with TSL and enter the critical region.

```

enter_region:
    TSL REGISTER, LOCK      | copy LOCK to register and set LOCK to 1
    CMP REGISTER, #0        | was LOCK zero?
    JNE enter_region       | if non-zero, lock was set, so loop
    RET                    | return; critical region entered

leave_region:
    MOVE LOCK, #0          | store 0 in LOCK
    RET

```

An alternative atomic instruction is **XCHG** (swap register and memory word). Both rely on busy waiting (spin locks), which wastes CPU and can cause the *priority inversion problem*.

(f) Sleep and Wakeup

Instead of wasting CPU time in busy waiting, a process that cannot enter its critical region calls **sleep** (blocks) and is woken up with **wakeup** by the process that leaves. Consider the **producer-consumer (bounded buffer)** problem with N slots and a shared variable **count**:

```

void producer() {
    while (TRUE) {
        item = produce_item();
        if (count == N) sleep();
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer);
    }
}

void consumer() {
    while (TRUE) {
        if (count == 0) sleep();
        item = remove_item();
        count = count - 1;
        if (count == N-1) wakeup(producer);
        consume_item(item);
    }
}

```

Fatal race: the consumer reads `count = 0` but, before it calls `sleep`, the scheduler switches to the producer. The producer inserts an item and sends `wakeup`, but the consumer is not yet asleep, so the signal is **lost**. The consumer then sleeps forever, and eventually the producer fills the buffer and also sleeps — both sleep forever (deadlock). A *wakeup-waiting bit* is a patch but does not scale; semaphores solve this properly.

3.3.6 Types of Mutual Exclusion (Synchronisation Primitives)

(a) Semaphores

Proposed by **Dijkstra (1965)**. A **semaphore** is an integer variable accessed only through two **atomic** operations: **down (P, wait)** and **up (V, signal)**.

```

down(S):  if (S > 0)  S = S - 1;          /* proceed          */
           else      block the process on S; /* put to sleep      */

up(S):    if (some process is blocked on S) wake one up;
           else      S = S + 1;

```

- **Binary semaphore** (value 0 or 1) — used for mutual exclusion (also called a mutex lock).
- **Counting semaphore** (value ≥ 0) — controls access to a resource that has several identical instances.
- Down and up are implemented as system calls with interrupts disabled briefly (or using TSL) so they are atomic.

Solving the bounded-buffer problem with semaphores — three semaphores are used: **mutex** (= 1) guards the buffer, **empty** (= N) counts empty slots, and **full** (= 0) counts filled slots.

```

semaphore mutex = 1, empty = N, full = 0;

void producer() {
    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer() {
    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}

```

Note: The order of the down operations matters: if the producer executed down(mutex) before down(empty) and the buffer were full, it would hold the mutex while sleeping and the consumer could never enter — deadlock.

(b) Monitors

Semaphores are error-prone. Hoare (1974) and Brinch Hansen (1975) proposed a higher-level primitive, the **monitor** — a collection of procedures, variables and data structures grouped together in a special kind of module or package. **Only one process can be active inside a monitor at any instant**; the compiler enforces this mutual exclusion automatically. Synchronisation inside monitors is done with **condition variables** and two operations: **wait(c)** blocks the caller on condition c (releasing the monitor), and **signal(c)** wakes a process waiting on c. Monitors are a programming-language construct (Java synchronized, Pascal-Concurrent, Ada).

```

monitor ProducerConsumer
    condition full, empty; integer count;
    procedure insert(item) {
        if (count == N) wait(full);
        insert_item(item); count++;
        if (count == 1) signal(empty);
    }
    procedure remove() {
        if (count == 0) wait(empty);
        item = remove_item(); count--;
        if (count == N-1) signal(full);
    }
end monitor;

```

(c) Mutexes

A **mutex** is a simplified semaphore that can only be in one of two states: **unlocked** or **locked**. **mutex_lock** tries to acquire the lock; if it is already locked the caller is blocked (unlike a spin lock, the thread yields the CPU). **mutex_unlock** releases it, and one waiting thread is woken up. Mutexes are used for mutual exclusion between threads (e.g., Pthreads `pthread_mutex_lock()`). Unlike a semaphore, a mutex has an *owner* — only the process that locked it should unlock it.

```
mutex_lock:
    TSL REGISTER, MUTEX      | copy mutex to register, set mutex to 1
    CMP REGISTER, #0         | was mutex zero?
    JZE ok                   | if zero, mutex was unlocked: return
    CALL thread_yield        | mutex busy: schedule another thread
    JMP mutex_lock           | try again later
ok: RET
mutex_unlock:
    MOVE MUTEX, #0           | store 0 in mutex
    RET
```

(d) Message Passing

Uses two primitives — **send(destination, &message)** and **receive(source, &message)** — which, unlike monitors, work across machines. *Design issues*: lost messages (acknowledgements and timeouts), authentication, naming (mailboxes/ports), and the choice of **blocking (synchronous)** vs **non-blocking (asynchronous)** operation. In **rendezvous** both sender and receiver block until the message is transferred. The producer–consumer problem is easily solved: the consumer sends *N* empty messages to the producer; the producer fills one with data and sends it back, so that the total number of messages stays constant and the buffer is bounded.

(e) Bounded Buffer

The bounded buffer is a shared queue of fixed size *N* used to connect producers and consumers. It requires (1) mutual exclusion on the buffer, (2) blocking of the producer when the buffer is full and (3) blocking of the consumer when it is empty. It is solved with semaphores (above), monitors or message passing, and is the model for pipes and spoolers.

Primitive	Idea	Advantage	Limitation
Semaphore	Integer with atomic down/up	General; no busy waiting	Easy to misuse; deadlock-prone
Monitor	Language module with one active process	Compiler-enforced safety	Needs language support
Mutex	Two-state lock	Simple and fast for threads	No counting

Primitive	Idea	Advantage	Limitation
Message passing	send / receive	Works on distributed systems	Slower; message loss issues

3.3.7 Serializability

Serializability is a correctness criterion for **concurrent transactions**. A concurrent schedule is **serializable** if its result is equivalent to that of some *serial* execution (transactions executed one after another without interleaving). A **transaction** is a collection of read and write operations that performs a single logical function and must be *atomic* — either all operations occur or none do (commit or abort/rollback).

(a) Locking Protocols

Each data item has a lock. A transaction must acquire a **shared (S) lock** to read and an **exclusive (X) lock** to write. The **Two-Phase Locking (2PL)** protocol ensures serializability by requiring each transaction to issue lock and unlock in two phases: a **growing phase** (may obtain locks but may not release any) and a **shrinking phase** (may release locks but may not obtain new ones). 2PL guarantees serializability but does not prevent deadlock. *Strict 2PL* holds all exclusive locks until commit, avoiding cascading aborts.

(b) Timestamp Protocols

Each transaction T receives a unique **timestamp** $TS(T)$ when it enters the system (older = smaller). For each data item Q the system keeps **W-timestamp(Q)** (largest timestamp of a successful write) and **R-timestamp(Q)** (largest timestamp of a successful read). Conflicting operations are executed in timestamp order:

- **T wants to read Q:** if $TS(T) < W\text{-timestamp}(Q)$, T would read an already overwritten value → **reject and roll back T**. Otherwise the read is executed and $R\text{-timestamp}(Q) = \max(R\text{-timestamp}(Q), TS(T))$.
- **T wants to write Q:** if $TS(T) < R\text{-timestamp}(Q)$ (a later transaction already read the old value) or $TS(T) < W\text{-timestamp}(Q)$ (obsolete write) → **reject and roll back T**. Otherwise execute the write and set $W\text{-timestamp}(Q) = TS(T)$.

Timestamp ordering guarantees conflict serializability and is free of deadlock, but may cause starvation of long transactions.

3.3.8 Classical IPC Problems

(a) The Dining Philosophers Problem

Five philosophers sit around a circular table; each has a plate of spaghetti and needs **two forks** (one on each side) to eat. There are only five forks. A philosopher alternately thinks and eats. Problem: design a protocol so that no philosopher starves and no deadlock occurs.

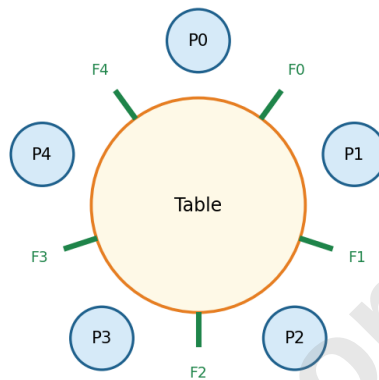


Figure 3.3 — Dining philosophers

Naïve solution: take the left fork, then the right fork. If all five take their left fork simultaneously, everyone waits forever for the right — **deadlock**. If they back off and retry at the same time, **starvation** results.

Correct semaphore solution: uses a *state* array (THINKING, HUNGRY, EATING), a mutex protecting it, and one semaphore per philosopher. A philosopher may start eating only if neither neighbour is eating.

```

#define N 5
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N
int state[N]; semaphore mutex = 1, s[N]; /* s[i] initially 0 */

void philosopher(int i) {
    while (TRUE) { think(); take_forks(i); eat(); put_forks(i); }
}
void take_forks(int i) {
    down(&mutex); state[i] = HUNGRY; test(i); up(&mutex);
    down(&s[i]); /* block if forks not acquired */
}
void put_forks(int i) {
    down(&mutex); state[i] = THINKING;
    test(LEFT); test(RIGHT); up(&mutex);
}
void test(int i) {
    if (state[i]==HUNGRY && state[LEFT]!=EATING && state[RIGHT]!=EATING)
    { state[i] = EATING; up(&s[i]); }
}

```

Other remedies: allow at most four philosophers to sit at once; make one philosopher pick right fork first (asymmetric); or pick both forks only if both are available.

(b) The Readers and Writers Problem

Models access to a database: many processes may **read** simultaneously, but if one process is **writing**, no other process (reader or writer) may access it. In the first readers–writers solution readers have priority: the first reader locks the database (**db** semaphore) and the last reader unlocks it; a shared counter **rc** tracks the number of readers, protected by **mutex**.

```

semaphore mutex = 1, db = 1; int rc = 0;

void reader() {
    down(&mutex);
    rc = rc + 1;
    if (rc == 1) down(&db);
    up(&mutex);
    read_data_base();
    down(&mutex);
    rc = rc - 1;
    if (rc == 0) up(&db);
    up(&mutex);
    use_data_read();
}

void writer() {
    while (TRUE) {
        think_up_data();
        down(&db);
        write_data_base();
        up(&db);
    }
}

```

Note: Drawback: a continuous stream of readers can starve the writer. A writer-priority version has the opposite problem.

(c) The Sleeping Barber Problem

A barber shop has one barber, one barber chair and **n waiting chairs**. If there are no customers the barber sleeps. A customer entering wakes the barber if he is asleep; if the barber is busy the customer sits in a waiting chair if one is free, otherwise **leaves**. Semaphores used: **customers** (= 0, number of waiting customers), **barbers** (= 0, number of idle barbers ready to cut) and **mutex** (= 1) protecting the variable **waiting**.

```
semaphore customers = 0, barbers = 0, mutex = 1;  int waiting = 0;

void barber() {
    while (TRUE) {
        down(&customers); /* sleep if none */
        down(&mutex);
        waiting = waiting - 1;
        up(&barbers);      /* ready to cut */
        up(&mutex);
        cut_hair();
    }
}

void customer() {
    down(&mutex);
    if (waiting < CHAIRS) {
        waiting = waiting + 1;
        up(&customers); /* wake barber */
        up(&mutex);
        down(&barbers); /* wait turn */
        get_haircut();
    } else up(&mutex); /* shop full: leave */
}
```

3.4 Process Scheduling (5 Hrs)

3.4.1 Basic Concept

When a computer is multiprogrammed, several processes compete for the CPU at the same time. The part of the operating system that decides which process to run next is the **scheduler**, and the algorithm it uses is the **scheduling algorithm**. The main aim is to maximise CPU utilisation. The three levels of scheduling are:

Scheduler	Also called	Function	Frequency
Long-term	Job scheduler	Selects which jobs are admitted from the pool into memory; controls degree of multiprogramming	Seconds–minutes (infrequent)
Medium-term	Swapper	Swaps processes out of / into memory to reduce load	Occasional
Short-term	CPU scheduler	Selects a ready process and allocates the CPU	Milliseconds (very frequent)

The **dispatcher** gives control of the CPU to the selected process: it performs the context switch, switches to user mode and jumps to the proper location in the program. The time it takes is the **dispatch latency**. CPU scheduling decisions may take place when a process (1) switches from running to waiting, (2) switches from running to ready, (3) switches from waiting to ready, or (4) terminates.

A process alternates between **CPU bursts** and **I/O bursts**. A **CPU-bound** process has long CPU bursts; an **I/O-bound** process has many short CPU bursts.

3.4.2 Types of Scheduling

(a) Preemptive vs Non-preemptive Scheduling

Non-preemptive	Preemptive
Once the CPU is given to a process it keeps it until it terminates or blocks (waits for I/O)	The CPU can be taken away from a running process (timer interrupt or arrival of a higher-priority process)
Simple; low overhead; no race on shared data	More overhead (context switches); needs care with shared data
Poor response time; short jobs may wait behind long ones	Better response time and fairness
Examples: FCFS, non-preemptive SJF	Examples: Round Robin, SRTN, preemptive priority

(b) Categories of Scheduling Environments

- **Batch:** no users waiting for quick response; non-preemptive or long-quantum preemptive algorithms are acceptable. Goals: throughput, turnaround time, CPU utilisation.
- **Interactive:** preemption is essential so that one process does not hog the CPU. Goals: response time, proportionality. Examples: Round Robin, priority, multilevel queue.
- **Real-time:** deadlines must be met; preemption is sometimes unnecessary because processes are short. Goals: meeting deadlines, predictability. May be hard or soft real-time.

3.4.3 Scheduling Criteria (Performance Analysis)

Criterion	Definition	Goal
CPU utilisation	Percentage of time the CPU is busy	Maximise (40% – 90%)
Throughput	Number of processes completed per unit time	Maximise
Turnaround time (TAT)	Completion time – arrival time	Minimise
Waiting time (WT)	Time spent in the ready queue = TAT – burst time	Minimise
Response time (RT)	Time from submission to the first response (first CPU allocation – arrival)	Minimise
Fairness	Each process gets a fair share of CPU	Maximise

Formulae used in numerical problems:

- Turnaround Time = Completion Time (CT) – Arrival Time (AT)
- Waiting Time = Turnaround Time – Burst Time (BT)
- Response Time = Time of first CPU allocation – Arrival Time
- Average WT = $(\sum WT) / n$; Average TAT = $(\sum TAT) / n$

3.4.4 Scheduling Algorithms

(1) First-Come, First-Served (FCFS)

Processes are executed in the order of arrival, using a simple FIFO queue. Non-preemptive. *Advantages:* easy to implement and fair in arrival order. *Disadvantages:* average waiting time is often long; suffers the **convoy effect** — many short processes wait behind one long CPU-bound process.

(2) Shortest-Job-First (SJF) / Shortest Process Next (SPN)

The CPU is given to the process with the **smallest next CPU burst**. **SJF is provably optimal** — it gives the minimum average waiting time for a given set of processes (when all are available simultaneously). The difficulty is knowing the length of the next burst; it is estimated by **exponential averaging** of previous bursts: $\tau_{n+1} = \alpha \cdot t_n + (1 - \alpha) \cdot \tau_n$. Non-preemptive SJF is often called **Shortest Process Next**. *Disadvantage:* long jobs may **starve**.

(3) Shortest Remaining Time Next (SRTN / SRTF)

The preemptive version of SJF. Whenever a new process arrives, its burst time is compared with the *remaining* time of the running process; if the new one is shorter, the running process is pre-empted. Gives minimum average waiting time among all algorithms but has higher overhead and starvation of long jobs.

(4) Round-Robin (RR)

Designed for time-sharing systems. Each process is given a small unit of CPU time called a **quantum** (time slice, typically 10–100 ms). When the quantum expires the process is pre-empted and put at the **end of the ready queue**. The ready queue is a circular FIFO queue. Performance depends on the quantum: if it is **too large** RR degenerates to FCFS; if it is **too small** the context-switch overhead dominates. A common rule of thumb is that 80% of CPU bursts should be shorter than the quantum.

(5) Priority Scheduling

A **priority** is associated with each process and the CPU is allocated to the highest-priority ready process (equal priorities in FCFS order). Priorities may be **static** or **dynamic**, and internal (time limits, memory needs) or external (importance). It can be preemptive or non-preemptive.

Problem: indefinite blocking (starvation) of low-priority processes. **Solution: aging** — gradually increase the priority of processes that wait a long time. SJF is a special case of priority scheduling where $\text{priority} = 1 / (\text{predicted burst})$.

(6) Multiple Queues / Multilevel Queue Scheduling

The ready queue is partitioned into **separate queues**, for example system processes, interactive (foreground) processes, interactive editing, batch (background) processes and student processes. Each process is permanently assigned to one queue; each queue has its own algorithm (e.g., RR for foreground, FCFS for background). Scheduling among the queues is either **fixed priority** (serve foreground fully before background — possible starvation) or **time slicing** (e.g., 80% of CPU to foreground RR, 20% to background FCFS).

(7) Multilevel Feedback Queue Scheduling

Like multilevel queues, but a process **can move between queues**. A process that uses too much CPU time is moved to a lower-priority queue; I/O-bound and interactive processes stay in the higher-priority queues; a process that waits too long in a lower queue is moved up (aging). A typical example: Queue 0 — RR with quantum 8 ms; Queue 1 — RR with quantum 16 ms; Queue 2 — FCFS. A new job enters queue 0; if it does not finish in 8 ms it moves to queue 1, and so on. Defined by number of queues, algorithm of each queue, and rules for upgrading, demoting and initial placing. It is the most general and complex scheduler (used in Unix BSD, Windows).

(8) Highest Response Ratio Next (HRN)

A **non-preemptive** algorithm that corrects SJF's bias towards short jobs and avoids starvation. The priority of each process is calculated as:

$$\text{Response Ratio} = (\text{Waiting Time} + \text{Burst Time}) / \text{Burst Time}$$

At each scheduling decision the process with the **highest response ratio** runs next. A long-waiting job's ratio grows, so it eventually gets the CPU, while short jobs still have an advantage.

(9) Guaranteed Scheduling

Makes real promises to users about performance. If n users are logged in, each is guaranteed $1/n$ of the CPU power. The system computes for each process the ratio (**actual CPU time consumed**) / (**CPU time entitled**). A ratio of 0.5 means the process has had only half of what it deserves; 2.0 means twice. The scheduler runs the process with the **lowest ratio** until its ratio passes that of its closest competitor.

(10) Lottery Scheduling

Processes are given lottery **tickets** for system resources such as CPU time. When a scheduling decision is made a ticket is picked at random and the process holding it gets the CPU. A process holding a fraction f of the tickets will receive about a fraction f of the CPU. Important processes can be given more tickets; cooperating processes can exchange tickets. Simple, highly responsive, and starvation-free.

(11) Fair-Share Scheduling

Scheduling takes into account **who owns** the process. If user 1 has nine processes and user 2 has one, plain RR would give user 1 90% of the CPU. Fair-share allocates CPU so that each *user* (not each process) receives a defined share — e.g., two users each get 50% regardless of how many processes they run.

(12) Real-Time Scheduling

In a real-time system, time plays an essential role. Events may be **periodic** (occur at regular intervals) or **aperiodic**. A set of m periodic events with period P_i and CPU time C_i is **schedulable** only if the sum of C/P over all events is ≤ 1 . Two well-known algorithms: **Rate Monotonic Scheduling (RMS)** — static priority, shorter period = higher priority; and **Earliest Deadline First (EDF)** — dynamic priority, the process with the nearest deadline runs first (optimal for uniprocessors).

Algorithm	Type	Advantage	Disadvantage
FCFS	Non-pre.	Simple, fair by arrival	Convoy effect, long average WT
SJF / SPN	Non-pre.	Minimum average WT	Starvation; burst length must be predicted
SRTN	Preemptive	Best average WT	Starvation; overhead
Round Robin	Preemptive	Good response; no starvation	Quantum choice is critical
Priority	Either	Important jobs first	Starvation (use aging)
Multilevel queue	Preemptive	Different policy per class	Inflexible; starvation possible

Algorithm	Type	Advantage	Disadvantage
Multilevel feedback	Preemptive	Flexible, adapts to behaviour	Complex to tune
HRN	Non-pre.	No starvation; favours short jobs	Overhead of computing ratios
Lottery / Guaranteed / Fair share	Preemptive	Proportional fairness	Probabilistic or complex accounting

3.4.5 Numerical Examples on Scheduling

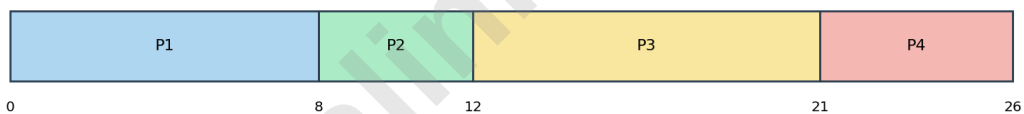
Example 1 — FCFS, SJF, SRTN and Round Robin

Consider four processes:

P1: AT = 0, BT = 8 **P2:** AT = 1, BT = 4 **P3:** AT = 2, BT = 9 **P4:** AT = 3, BT = 5

Draw the Gantt charts and find the average waiting time and turnaround time for FCFS, non-preemptive SJF, SRTN and Round Robin (quantum = 4).

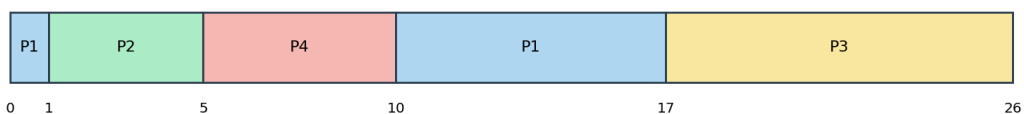
FCFS (avg WT = 8.75, avg TAT = 15.25)



SJF non-preemptive (avg WT = 7.75, avg TAT = 14.25)



SRTN preemptive (avg WT = 6.5, avg TAT = 13)



Round Robin, q = 4 (avg WT = 11.75, avg TAT = 18.25)

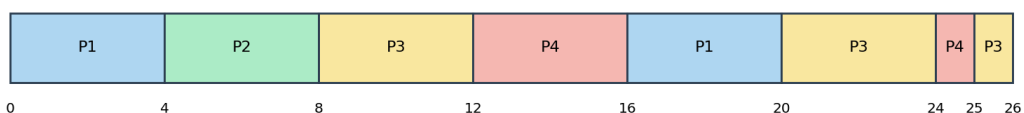


Figure 3.4 — Gantt charts for Example 1

Solution — FCFS

Order: P1, P2, P3, P4. Completion times: 8, 12, 21, 26.

TAT = CT - AT: P1 = 8, P2 = 11, P3 = 19, P4 = 23 → average = $61/4 = 15.25$

WT = TAT - BT: P1 = 0, P2 = 7, P3 = 10, P4 = 18 → average = $35/4 = 8.75$

Solution — SJF (non-preemptive)

At $t = 0$ only P1 is present, so it runs to 8. Then the shortest of P2 (4), P3 (9), P4 (5) → P2, then P4, then P3.

CT: P1 = 8, P2 = 12, P4 = 17, P3 = 26. TAT: 8, 11, 14, 24 → average = $57/4 = 14.25$

WT: P1 = 0, P2 = 7, P4 = 9, P3 = 15 → average = $31/4 = 7.75$

Solution — SRTN (preemptive SJF)

$t = 1$: P2 (4) arrives, shorter than P1's remaining 7 → P1 pre-empted. P2 runs 1–5. At 5: remaining P1 = 7, P3 = 9, P4 = 5 → P4 runs 5–10, then P1 10–17, then P3 17–26.

CT: P1 = 17, P2 = 5, P3 = 26, P4 = 10. TAT: 17, 4, 24, 7 → average = $52/4 = 13$

WT: P1 = 9, P2 = 0, P3 = 15, P4 = 2 → average = $26/4 = 6.5$

Solution — Round Robin ($q = 4$)

Ready queue after each slice: P1 (0–4) → P2 (4–8, done) → P3 (8–12) → P4 (12–16) → P1 (16–20, done) → P3 (20–24) → P4 (24–25, done) → P3 (25–26, done).

CT: P1 = 20, P2 = 8, P3 = 26, P4 = 25. TAT: 20, 7, 24, 22 → average = $73/4 = 18.25$

WT: P1 = 12, P2 = 3, P3 = 15, P4 = 17 → average = $47/4 = 11.75$; Response times: 0, 3, 6, 9 → average 4.5

Example 2 — Priority scheduling (non-preemptive, lower number = higher priority)

All processes arrive at $t = 0$.

P1: BT = 10, priority 3 **P2**: BT = 1, priority 1 **P3**: BT = 2, priority 4 **P4**: BT = 1, priority 5

P5: BT = 5, priority 2

Solution

Order by priority: P2 (0–1), P5 (1–6), P1 (6–16), P3 (16–18), P4 (18–19).

WT: P1 = 6, P2 = 0, P3 = 16, P4 = 18, P5 = 1 → average = $41/5 = 8.2$

TAT: P1 = 16, P2 = 1, P3 = 18, P4 = 19, P5 = 6 → average = $60/5 = 12$

Example 3 — Highest Response Ratio Next

A: AT = 0, BT = 3 B: AT = 2, BT = 6 C: AT = 4, BT = 4 D: AT = 6, BT = 5 E: AT = 8, BT = 2

Solution

$t = 0$: only A $\rightarrow$ runs 0–3. At $t = 3$ only B has arrived $\rightarrow$ B runs 3–9.

At $t = 9$: C: $(5 + 4)/4 = 2.25$; D: $(3 + 5)/5 = 1.6$; E: $(1 + 2)/2 = 1.5 \rightarrow$ C runs 9–13.

At $t = 13$: D: $(7 + 5)/5 = 2.4$; E: $(5 + 2)/2 = 3.5 \rightarrow$ E runs 13–15; then D runs 15–20.

Order: A, B, C, E, D. CT: A = 3, B = 9, C = 13, E = 15, D = 20.

Key Takeaways

- ✓ A **process** is a program in execution; its states are new, ready, running, waiting and terminated; the OS tracks it in a **PCB**.
- ✓ A **thread** is a lightweight process; models: many-to-one, one-to-one, many-to-many.
- ✓ A **race condition** arises from unsynchronised access to shared data; solutions must give mutual exclusion, progress and bounded waiting.
- ✓ Software solutions: lock variable (flawed), strict alternation (violates progress), **Peterson's** (correct for 2 processes); hardware: **TSL**.
- ✓ Sleep/wakeup can lose signals; **semaphores**, **monitors**, **mutexes** and **message passing** are the standard primitives.
- ✓ Serializability is ensured by **two-phase locking** or **timestamp ordering**.
- ✓ Classical problems: dining philosophers, readers–writers, sleeping barber.
- ✓ Scheduling criteria: CPU utilisation, throughput, TAT, WT, RT. SJF/SRTN give minimum average WT; RR is the standard for time-sharing; HRN, aging and MLFQ prevent starvation.

— End of Unit 3 —