
OPERATING SYSTEM

Chapter 3 PROCESS MANAGEMENT

Complete Study Notes for Engineering Students

3.1 Processes: Concept, States, Process Control Block

3.2 Process Scheduling: Queues, Schedulers, Context Switch

3.3 Operations on Processes: Creation and Termination

3.4 Inter-Process Communication, Synchronization, Semaphores

3.5 Threads: Benefits, User and Kernel Threads

3.6 Multithreading Models

15 Multiple Choice Questions with Answer Key

10 Long-Answer Questions

PolyNotesHub

Contents

3.1 Processes: Concept, States, Process Control Block	3
3.2 Process Scheduling: Queues, Schedulers, Context Switch	7
3.3 Operations on Processes: Creation and Termination	10
3.4 Inter-Process Communication, Synchronization and Semaphores	12
3.5 Threads: Benefits, User and Kernel Threads	20
3.6 Multithreading Models	23
Quick Revision Summary	25
Multiple Choice Questions (15)	26
Answer Key for MCQs	27
Long-Answer (Broad) Questions (10)	28

How to use these notes

Each section starts with the concept in paragraph form, followed by lists for quick revision, tables for comparisons, and diagrams wherever a picture explains better than words. Revise the **boxed key points** and the **tables** just before exams. The MCQs and long-answer questions at the end follow the pattern of university examinations.

Chapter 3: Process Management

A modern computer runs many activities at the same time: a web browser downloads a file, a music player plays a song, an antivirus scans the disk and the operating system itself performs housekeeping. The operating system must give each of these activities a fair share of the CPU, memory and I/O devices, protect them from each other, and let them cooperate when required. **Process management** is the part of the operating system responsible for all of this.

The major responsibilities of the OS in process management are:

- **Creating and deleting** user and system processes.
- **Scheduling** processes and threads on the CPU(s).
- **Suspending and resuming** processes.
- Providing mechanisms for **process synchronization**.
- Providing mechanisms for **inter-process communication** (IPC).
- Handling **deadlocks** (studied in the next chapter).

3.1 Processes: Concept, States, Process Control Block

3.1.1 Process Concept

A **program** is a passive entity: an executable file stored on disk containing a list of instructions. A **process** is a **program in execution**: an active entity with a program counter specifying the next instruction to execute, a set of associated resources, and its own memory. When an executable file is loaded into memory (by double-clicking it or typing its name on the command line) it becomes a process. One program can give rise to several processes: if three users run the same text editor, there is one program but three distinct processes.

Basis	Program	Process
Nature	Passive entity (file on disk)	Active entity (running instance)
Lifetime	Exists until deleted from disk	Exists only from creation to termination
Resources	Needs no CPU or memory while stored	Needs CPU, memory, files, I/O devices
Components	Only code (and static data)	Code, data, stack, heap, registers, program counter
Count	One program file	Many processes can come from one program

Table 3.1: Program versus process

Structure of a process in memory

The memory image of a process is divided into four sections:

- **Text section:** the executable code. Its size is fixed at compile time.
- **Data section:** global and static variables. Also fixed in size.
- **Heap:** memory that is dynamically allocated at run time (for example by *malloc()* or *new*). It grows upward.
- **Stack:** temporary data when functions are called: function parameters, return addresses and local variables. It grows downward, towards the heap.

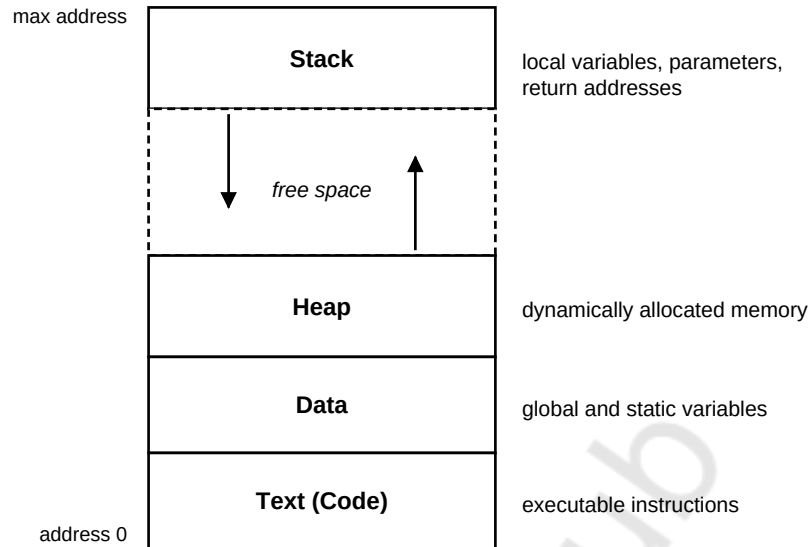


Fig. 3.1: Layout of a process in memory

3.1.2 Process States

As a process executes, it changes **state**. The state of a process is defined in part by its current activity. Every process can be in exactly one of the following states at any instant:

State	Meaning
New	The process is being created. The OS has accepted the program but has not yet admitted it to the pool of executable processes.
Ready	The process is loaded in main memory and has everything it needs except the CPU. It waits in the ready queue to be assigned to a processor.
Running	The CPU is executing the instructions of the process. On a single-core CPU only one process can be in this state at any instant.
Waiting (Blocked)	The process cannot continue until some event occurs, such as completion of an I/O operation or receipt of a signal.
Terminated	The process has finished execution (or was killed). The OS is reclaiming its resources.

Table 3.2: Process states

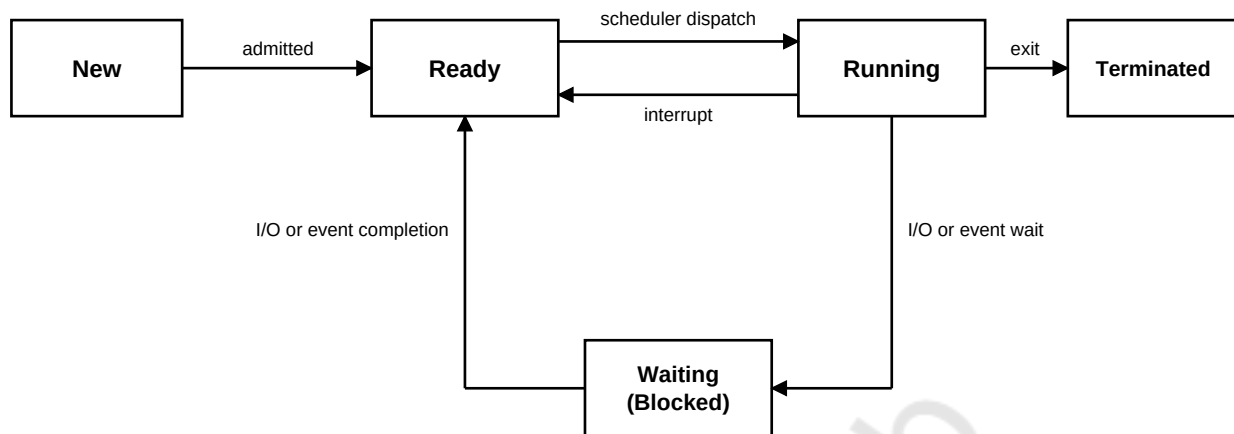


Fig. 3.2: Process state transition diagram

The transitions shown in the diagram are explained below:

- **New to Ready (admitted):** the long-term scheduler admits the process into the ready queue.
- **Ready to Running (scheduler dispatch):** the short-term scheduler selects the process and the dispatcher gives it the CPU.
- **Running to Ready (interrupt):** the time slice expires or a higher-priority process arrives, so the running process is preempted.
- **Running to Waiting (I/O or event wait):** the process requests I/O or waits for an event.
- **Waiting to Ready (I/O or event completion):** the awaited event occurs. Note that the process goes to *ready*, never directly to running.
- **Running to Terminated (exit):** the process executes its last statement or calls *exit()*.

Key point

Only **one** process per CPU core can be *running* at an instant, but many processes can be *ready* or *waiting*. A process never moves directly from waiting to running.

3.1.3 Process Control Block (PCB)

Each process is represented in the operating system by a **Process Control Block**, also called a **task control block**. The PCB is a data structure that stores all the information the OS needs to manage the process. When the CPU switches from one process to another, the PCB of the outgoing process stores its state so that it can be restarted later exactly where it stopped. In Linux the PCB is implemented as the structure *task_struct*.

pointer to PCB	Process state
	Process number (PID)
	Program counter
	CPU registers
	Memory-management information
	CPU-scheduling information (priority, queue pointers)
	Accounting information
	I/O status information (open files, devices)

Fig. 3.3: Process Control Block (PCB)

PCB field	Purpose
Process state	Current state: new, ready, running, waiting, terminated.
Process number (PID)	Unique identifier of the process.
Program counter	Address of the next instruction to be executed for this process.
CPU registers	Accumulators, index registers, stack pointers and general-purpose registers. They must be saved on an interrupt so that the process can continue correctly afterwards.
Memory-management information	Base and limit registers, page tables or segment tables.
CPU-scheduling information	Process priority, pointers to scheduling queues and other scheduling parameters.
Accounting information	CPU time used, real time elapsed, time limits, account numbers, job numbers.
I/O status information	List of I/O devices allocated to the process, list of open files.

Table 3.3: Fields of the PCB

Important: the PCB does *not* hold the program's source code or its data. It holds only the bookkeeping information that the kernel needs to identify, schedule and control the process. The PCBs of all processes together form the **process table**.

3.2 Process Scheduling: Queues, Schedulers, Context Switch

The objective of **multiprogramming** is to have some process running at all times so that CPU utilization is maximized. The objective of **time sharing** is to switch the CPU among processes so frequently that users can interact with each program while it runs. To meet these objectives, the **process scheduler** selects an available process for execution on the CPU. A single-processor system never has more than one running process; extra processes must wait until the CPU is free and can be rescheduled.

3.2.1 Scheduling Queues

As processes enter the system they are placed in queues. Each queue is generally implemented as a linked list of PCBs, with a header that points to the first and last PCB in the list. Each PCB also contains a pointer to the next PCB in the same queue.

Queue	Contents
Job queue	All processes in the system (those that have entered but are not yet in main memory are also kept here on batch systems).
Ready queue	All processes residing in main memory that are ready and waiting to execute on the CPU. It is generally stored as a linked list.
Device (I/O) queues	Processes waiting for a particular I/O device. Each device has its own queue, for example a disk queue or a printer queue.

Table 3.4: Types of scheduling queues

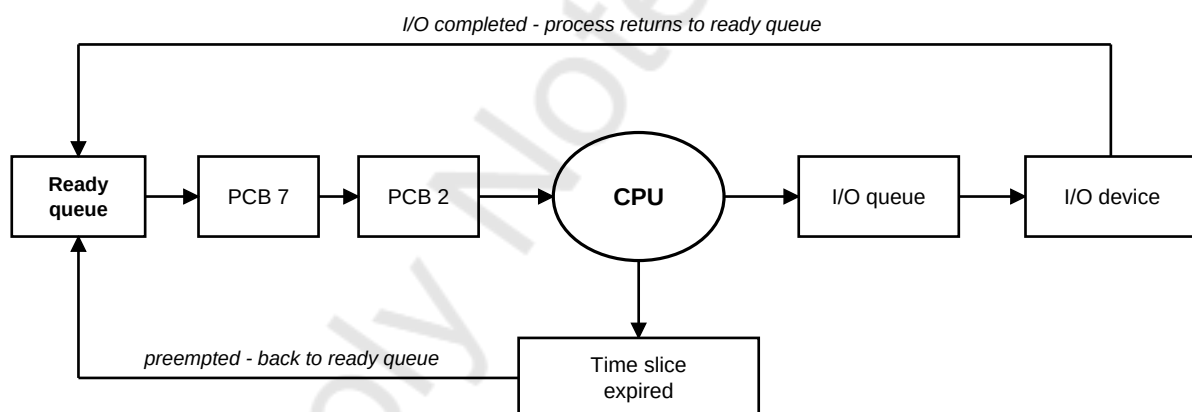


Fig. 3.4: Queueing diagram of process scheduling

A new process is initially put in the ready queue, where it waits until it is selected for execution (dispatched). Once the process is allocated the CPU, one of several events may occur:

- The process issues an **I/O request** and is placed in an I/O (device) queue.
- The process creates a **new child process** and waits for its termination.
- The process is **removed forcibly** from the CPU as a result of an interrupt (for example, expiry of its time slice) and is put back in the ready queue.

- The process **waits for an interrupt** or event.

In the first two cases the process eventually switches from the waiting state to the ready state and is put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and its PCB and resources are deallocated.

3.2.2 Schedulers

A **scheduler** is a system program that selects processes from queues in some order. In a batch system more processes are submitted than can be executed immediately, so they are spooled to disk. The OS uses three kinds of schedulers.

Basis	Long-term scheduler (job scheduler)	Short-term scheduler (CPU scheduler)	Medium-term scheduler
Function	Selects processes from the job pool on disk and loads them into memory (ready queue).	Selects one process from the ready queue and allocates the CPU to it.	Swaps processes out of memory and later back in (swapping).
Frequency	Invoked rarely (seconds or minutes).	Invoked very frequently (every few milliseconds).	Invoked occasionally, when memory is short.
Speed	Slow	Very fast	Medium
Controls	Degree of multiprogramming (number of processes in memory).	Which process runs next.	Reduces the degree of multiprogramming.
In time-sharing systems	Often absent or minimal.	Present and essential.	Present; used for swapping.
Process state change	New to Ready	Ready to Running	Swapped out to Ready (and reverse)

Table 3.5: Comparison of schedulers

The long-term scheduler should select a good **mix of processes**:

- **I/O-bound process**: spends more time doing I/O than computations; has many short CPU bursts.
- **CPU-bound process**: spends more time doing computations; has few very long CPU bursts.
- If all processes are I/O-bound the ready queue is almost always empty and the CPU is under-used. If all are CPU-bound the I/O devices are idle. A balanced mix gives the best performance.

The **medium-term scheduler** is used by time-sharing systems. Its key idea is that it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming. Later the process is reintroduced into memory and its execution is continued where it left off. This scheme is called **swapping**.

Dispatcher. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. Its functions are switching context, switching to user mode, and jumping to the proper location in the user program to restart it. The time the dispatcher takes to stop one process and start another is called **dispatch latency**, and it should be as small as possible.

3.2.3 Context Switch

Interrupts cause the operating system to change a CPU from its current task and to run a kernel routine. When an interrupt occurs, the system needs to save the current **context** of the running process so that it can restore that context when its processing is done. The context is represented in the PCB of the process and includes the value of the CPU registers, the process state and memory-management information.

Switching the CPU from one process to another requires performing a **state save** of the current process and a **state restore** of a different process. This task is known as a **context switch**.

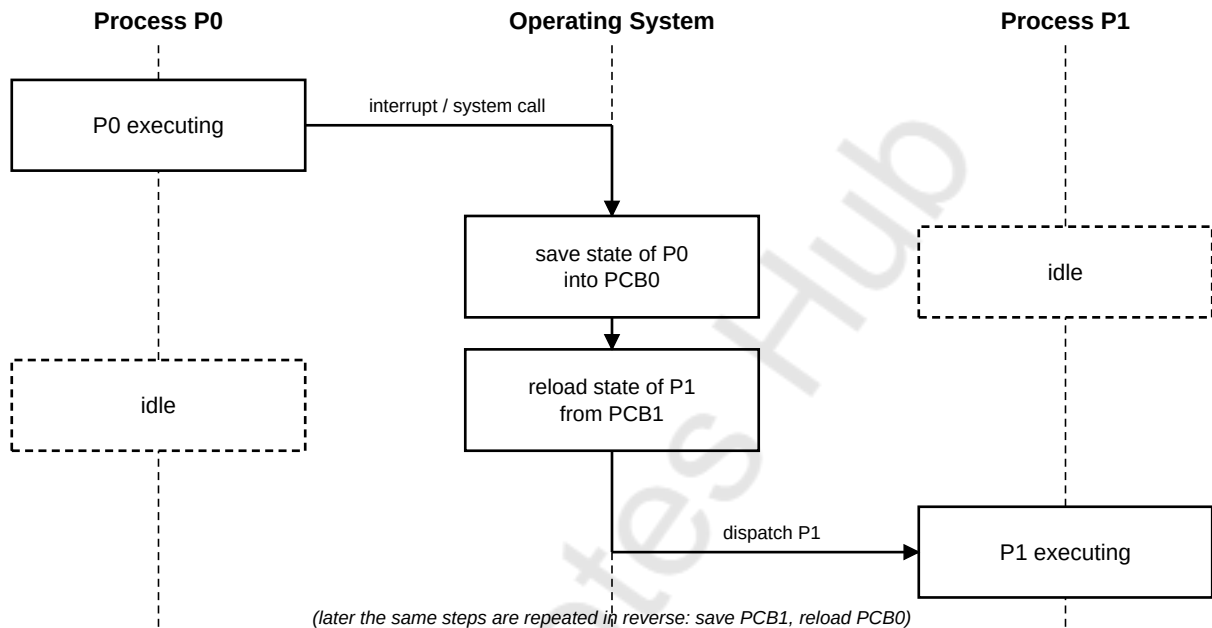


Fig. 3.5: Context switch from process P0 to process P1

Steps in a context switch:

- 1 An interrupt or system call causes the running process P0 to stop.
- 2 The kernel saves the program counter, registers and other state of P0 into PCB0.
- 3 The scheduler selects the next process P1 from the ready queue.
- 4 The kernel loads the saved state of P1 from PCB1 into the CPU registers.
- 5 Control passes to P1, which resumes from the point where it was last interrupted.

Context-switch time is pure overhead

The system does no useful work while switching. Switch time depends on hardware support: memory speed, the number of registers to be copied and the existence of special instructions (such as a single instruction to load or store all registers). Typical switch times are a few microseconds. Some processors provide multiple register sets so that a context switch only changes the pointer to the current register set.

3.3 Operations on Processes: Creation and Termination

3.3.1 Process Creation

A process may create several new processes during its execution through a **create-process system call**. The creating process is called the **parent** and the new processes are the **children**. Each of these new processes may in turn create other processes, forming a **tree of processes**. Processes are identified by a unique **process identifier (pid)**, usually an integer.

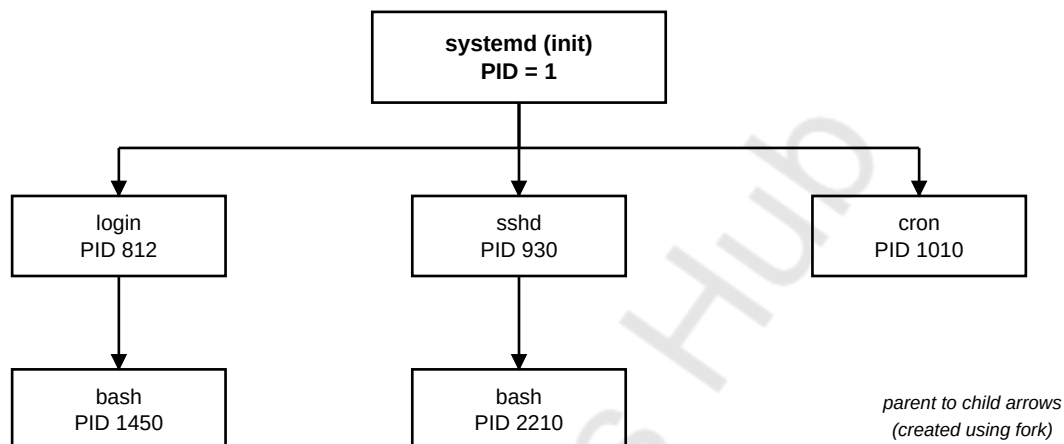


Fig. 3.6: A tree of processes on a typical UNIX/Linux system

When a parent creates a child, the following design choices exist:

Aspect	Possible options
Resource sharing	Parent and child share all resources; or child shares a subset of the parent's resources; or parent and child share no resources.
Execution	Parent and children execute concurrently ; or parent waits until some or all children terminate.
Address space	Child is a duplicate of the parent (same program and data); or child has a new program loaded into it.

Table 3.6: Design choices at process creation

Process creation in UNIX

- **fork()**: creates a new process. The child is a copy of the parent's address space. Both continue execution from the instruction after *fork()*. The return value of *fork()* is **0 in the child**, the **child's pid in the parent**, and **-1** if creation failed.
- **exec() family**: used after *fork* by one of the two processes to replace its memory space with a new program. The call does not return unless an error occurs.
- **wait()**: used by the parent to move itself out of the ready queue until the child terminates.

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main()
{
    pid_t pid;
    pid = fork();                /* create a child process */

    if (pid < 0) {                /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) {          /* child process */
        execlp("/bin/ls", "ls", NULL); /* load a new program */
    }
    else {                        /* parent process */
        wait(NULL);               /* wait for child to finish */
        printf("Child Complete");
    }
    return 0;
}

```

Process creation in Windows uses the *CreateProcess()* function. It behaves like *fork()* followed by *exec()* in a single call, but it requires the name of the program to load and takes ten parameters.

3.3.2 Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it using the **exit()** system call. The process may return a status value (usually an integer) to its parent through *wait()*. All the resources of the process, including physical and virtual memory, open files and I/O buffers, are deallocated by the operating system.

A process can also terminate **another** process through a system call such as *TerminateProcess()* in Windows or *kill()* in UNIX. Usually only the parent of the process can do this. A parent may terminate its child for these reasons:

- The child has **exceeded its allocated resources**.
- The **task assigned to the child is no longer required**.
- The **parent is exiting** and the operating system does not allow a child to continue without its parent. This is called **cascading termination**.
- The child has encountered a **fatal error** (illegal instruction, divide by zero, memory violation) or was killed by a user or another process.

Zombie and orphan processes

Basis	Zombie process	Orphan process
Definition	A child that has terminated but whose parent has not yet called wait(). Its PCB entry still remains in the process table.	A child whose parent has terminated without waiting for it; the child is still running.
State	Terminated (but not yet reaped).	Running or ready.
Resources	Holds only a process-table entry (pid and exit status).	Holds all normal resources.
Handling	Removed when the parent calls wait(). If the parent never does, it stays as long as the parent lives.	Adopted by the <i>init/systemd</i> process, which periodically calls wait() on them.

Table 3.7: Zombie versus orphan processes

3.4 Inter-Process Communication, Synchronization and Semaphores

3.4.1 Inter-Process Communication (IPC)

Processes executing concurrently in the operating system may be either independent or cooperating. An **independent process** cannot affect or be affected by other processes. A **cooperating process** can affect or be affected by others and therefore needs a mechanism to **exchange data and information**. Reasons for providing an environment that allows process cooperation are:

- **Information sharing:** several users may be interested in the same piece of information (a shared file).
- **Computation speedup:** a task is broken into subtasks that run in parallel on multiple cores.
- **Modularity:** the system is built in a modular fashion, dividing functions into separate processes.
- **Convenience:** a user may work on many tasks at the same time (editing, printing, compiling).

There are two fundamental IPC models: **shared memory** and **message passing**.

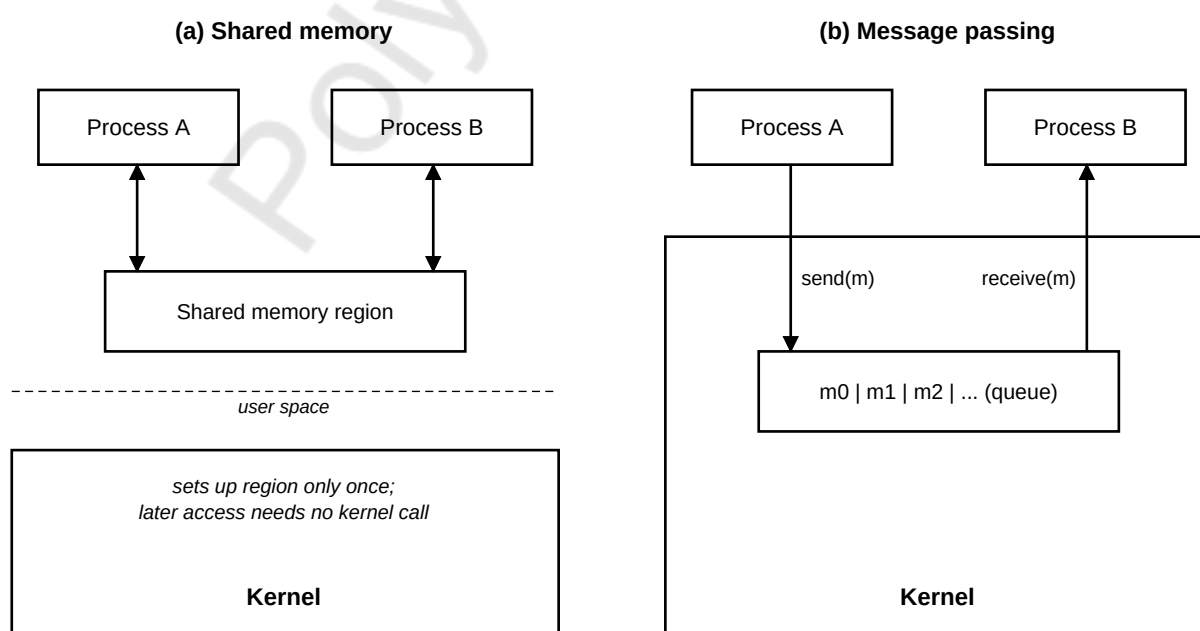


Fig. 3.7: Communication models: (a) shared memory, (b) message passing

Basis	Shared memory	Message passing
Mechanism	A region of memory is shared; processes read and write data in it.	Processes exchange messages using send() and receive() operations.
Kernel involvement	Only to set up the region; later access is done at memory speed.	Kernel is involved in every send and receive (system calls).
Speed	Faster	Slower (kernel intervention)
Synchronization	Programmer must ensure that processes do not write at the same time.	Handled implicitly by the send/receive operations.
Suitable for	Large amounts of data on the same machine.	Small amounts of data; distributed systems; easier to implement.

Table 3.8: Shared memory versus message passing

Shared-memory systems: the producer-consumer problem

A **producer** process produces information that is consumed by a **consumer** process (for example a compiler produces assembly code that an assembler consumes). To let them run concurrently, a **buffer** of items is placed in shared memory. The **unbounded buffer** has no limit on size; the **bounded buffer** assumes a fixed size, so the producer must wait if the buffer is full and the consumer must wait if it is empty. The code below uses a circular array with two pointers, *in* (next free position) and *out* (first full position). It can hold at most BUFFER_SIZE - 1 items.

```
#define BUFFER_SIZE 10
item buffer[BUFFER_SIZE];
int in = 0, out = 0;

/* Producer */
while (true) {
    /* produce an item in next_produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* buffer full: do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}

/* Consumer */
while (true) {
    while (in == out)
        ; /* buffer empty: do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    /* consume the item in next_consumed */
}
```

Message-passing systems

Message passing is useful for exchanging smaller amounts of data and is easier to implement for inter-computer communication. It provides at least two operations: **send(message)** and **receive(message)**. If processes P and Q want to communicate, a **communication link** must exist between them. The link can be classified on three design issues.

Issue	Variants	Description
Naming	Direct communication	Processes name each other explicitly: send(P, message), receive(Q, message). A link is associated with exactly two processes.
	Indirect communication	Messages are sent to and received from mailboxes (ports) . Processes can communicate only if they share a mailbox; a link may involve many processes.
Synchronization	Blocking (synchronous)	Blocking send: sender is blocked until the message is received. Blocking receive: receiver blocks until a message is available.
	Non-blocking (asynchronous)	Sender sends and continues. Receiver gets either a valid message or null.
Buffering	Zero capacity	No messages can wait in the link; the sender must block until the receiver receives (rendezvous).
	Bounded capacity	Queue of finite length n; the sender blocks if the queue is full.
	Unbounded capacity	Infinite queue; the sender never blocks.

Table 3.9: Design issues of message passing

Common IPC mechanisms implemented by operating systems include **pipes** (ordinary pipes between parent-child, named pipes/FIFOs between unrelated processes), **message queues**, **shared memory segments**, **signals**, **sockets** (communication over a network) and **remote procedure calls (RPC)**.

3.4.2 Process Synchronization

Concurrent access to shared data may result in **data inconsistency**. Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes. A situation where several processes access and manipulate the same data concurrently and the outcome depends on the particular **order** in which the accesses take place is called a **race condition**.

Example of a race condition

Let *counter* be a shared variable with value 5. The producer executes *counter++* and the consumer executes *counter--*. Each statement is compiled into three machine instructions: load into a register, modify the register, store back. If the instructions interleave as follows, the final value is **4** although the correct value is 5:

Time	Process	Action	Result
T0	Producer	register1 = counter	register1 = 5
T1	Producer	register1 = register1 + 1	register1 = 6
T2	Consumer	register2 = counter	register2 = 5
T3	Consumer	register2 = register2 - 1	register2 = 4
T4	Producer	counter = register1	counter = 6
T5	Consumer	counter = register2	counter = 4 (wrong)

Table 3.10: Interleaving that causes a race condition

The critical-section problem

Consider a system of n processes. Each process has a segment of code called its **critical section**, in which it may change common variables, update a table or write a file. The important feature is that when one process is executing in its critical section, no other process is allowed to execute in its critical section. The general structure of a process is:

```
do {
    [ entry section ]      /* request permission to enter */
        critical section
    [ exit section ]      /* announce that CS is finished */
        remainder section
} while (true);
```

A solution to the critical-section problem must satisfy three requirements:

Requirement	Meaning
1. Mutual exclusion	If process P_i is executing in its critical section, then no other process can be executing in its critical section.
2. Progress	If no process is in its critical section and some processes wish to enter, only those not in their remainder sections can take part in deciding who enters next, and this decision cannot be postponed indefinitely.
3. Bounded waiting	There is a limit on the number of times other processes are allowed to enter their critical sections after a process has made a request and before that request is granted. This prevents starvation.

Table 3.11: Requirements for a critical-section solution

Peterson's solution

Peterson's solution is a classic **software-based** solution for **two processes** P_0 and P_1 (i denotes the current process and j the other one). The processes share two variables: *int* $turn$, which indicates whose turn it is to enter, and *boolean* $flag[2]$, which indicates whether a process is ready to enter.

```
do {
    flag[i] = true;          /* I am ready to enter          */
    turn = j;                /* but let the other go first */
    while (flag[j] && turn == j)
        ;                  /* busy wait                */

    /* critical section */

    flag[i] = false;         /* I have left the CS        */

    /* remainder section */
} while (true);
```

- **Mutual exclusion:** P_i enters only if $flag[j]$ is false or $turn$ equals i . Since $turn$ can hold only one value at a time, both processes cannot be in the critical section together.
- **Progress and bounded waiting:** P_i can be stopped only while $flag[j]$ is true and $turn$ equals j . Once P_j leaves it sets $flag[j]$ to false, and even if P_j tries again it sets $turn$ to i , so P_i enters after at most one entry by P_j .
- **Limitation:** works for only two processes and may fail on modern processors that reorder instructions.

Hardware support for synchronization

- **Disabling interrupts:** works on a single-processor system but is inefficient and not scalable on multiprocessors.
- **Atomic instructions:** *test_and_set()* and *compare_and_swap()* execute as one uninterruptible unit and can be used to build locks.
- **Mutex locks:** a process calls *acquire()* before the critical section and *release()* after it. A simple implementation uses busy waiting, so the lock is called a **spinlock**.

3.4.3 Semaphores

A **semaphore** *S* is an integer variable that, apart from initialization, is accessed only through two **atomic** operations: **wait()** (originally called **P**, from the Dutch *proberen*, to test) and **signal()** (originally **V**, *verhogen*, to increment). It was proposed by **E. W. Dijkstra**. When one process modifies the semaphore value, no other process can simultaneously modify the same value.

```
wait(S) {                                signal(S) {
    while (S <= 0)                          S++;
    ; /* busy wait */                       }
    S--;
}
```

Type	Description	Typical use
Counting semaphore	Value ranges over an unrestricted integer domain.	Controls access to a resource with a finite number of instances (e.g. 5 printers).
Binary semaphore	Value can be only 0 or 1. Behaves like a mutex lock .	Mutual exclusion on a critical section.

Table 3.12: Types of semaphores

Usage of semaphores

- **Mutual exclusion:** initialize *mutex* = 1. Each process executes *wait(mutex)*; critical section; *signal(mutex)*.
- **Ordering of statements:** to ensure statement *S*₁ of *P*₁ executes before *S*₂ of *P*₂, initialize a semaphore *synch* = 0. *P*₁ executes *S*₁ then *signal(synch)*; *P*₂ executes *wait(synch)* then *S*₂.
- **Resource counting:** initialize the counting semaphore to the number of resource instances. *wait()* acquires an instance and *signal()* releases it.

Implementation without busy waiting

The definition above wastes CPU cycles in the while loop (busy waiting). To overcome this, each semaphore has an associated **waiting queue**. A process that must wait is **blocked** (moved to the waiting state) and placed in the queue; a **signal()** removes a process from the queue and **wakes** it up (moves it to the ready state).

```

typedef struct {
    int value;
    struct process *list;      /* queue of waiting processes */
} semaphore;

wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}

```

In this implementation a **negative semaphore value** indicates the number of processes waiting on it.

Problems with semaphores

Deadlock: two or more processes wait indefinitely for an event that only one of the waiting processes can cause (for example P0 does wait(S) then wait(Q) while P1 does wait(Q) then wait(S)). **Starvation:** a process waits indefinitely in the semaphore queue (for example if the queue is served in LIFO order). **Priority inversion:** a high-priority process waits for a lock held by a low-priority process. Incorrect use, such as signal(mutex) before wait(mutex), violates mutual exclusion.

3.4.4 Classical Problems of Synchronization

The following problems are used to test every new synchronization scheme.

(a) The Bounded-Buffer (Producer-Consumer) Problem

A pool of n buffers, each holding one item, is shared by a producer and a consumer. Three semaphores are used: **mutex** = 1 (mutual exclusion on buffer access), **empty** = n (number of empty buffers) and **full** = 0 (number of full buffers).

```

/* Producer */
do {
    /* produce an item in next_produced */
    wait(empty);
    wait(mutex);
    /* add next_produced to the buffer */
    signal(mutex);
    signal(full);
} while (true);

/* Consumer */
do {
    wait(full);
    wait(mutex);
    /* remove an item from buffer into next_consumed */
    signal(mutex);
    signal(empty);
    /* consume the item in next_consumed */
} while (true);

```

The producer waits when no buffer is empty; the consumer waits when no buffer is full. The symmetry between the two is evident: the producer produces full buffers for the consumer and the consumer produces empty buffers for the producer.

(b) The Readers-Writers Problem

A database is shared among several concurrent processes. **Readers** only read the data; **writers** both read and write. Two readers can access the data at the same time without harm, but if a writer and any other process (reader or writer) access it simultaneously, chaos may result. Writers must therefore have **exclusive access**. In the **first readers-writers problem** no reader is kept waiting unless a writer has already obtained permission to use the database. Shared data: semaphores **rw_mutex = 1** and **mutex = 1**, and integer **read_count = 0**.

```
/* Writer */
do {
    wait(rw_mutex);
    /* writing is performed */
    signal(rw_mutex);
} while (true);

/* Reader */
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);      /* first reader locks writers out */
    signal(mutex);
    /* reading is performed */
    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);    /* last reader lets writers in */
    signal(mutex);
} while (true);
```

Drawback: writers may **starve** because continuous arrival of readers keeps the database busy. The second readers-writers problem gives priority to writers, which can in turn starve readers.

(c) The Dining-Philosophers Problem

Five philosophers spend their lives thinking and eating. They sit around a circular table with a bowl of rice in the centre and **five chopsticks**, one between each pair of neighbours. A hungry philosopher needs **both** the left and the right chopstick to eat, picks up one at a time, and releases both after eating. The problem represents the allocation of several resources among several processes without deadlock and starvation.

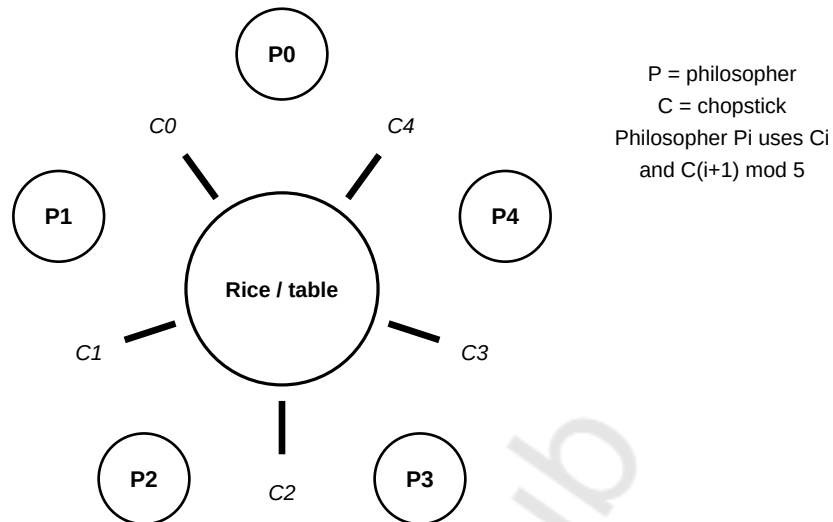


Fig. 3.8: The dining-philosophers problem

Each chopstick is a semaphore: `semaphore chopstick[5]`, all initialized to 1.

```
do {
    wait(chopstick[i]);           /* pick up left chopstick */
    wait(chopstick[(i + 1) % 5]); /* pick up right chopstick */
    /* eat */
    signal(chopstick[i]);         /* put down left */
    signal(chopstick[(i + 1) % 5]); /* put down right */
    /* think */
} while (true);
```

This guarantees that no two neighbours eat together, but it can create a **deadlock**: if all five philosophers become hungry at once and each picks up the left chopstick, every one waits forever for the right one. Possible remedies:

- Allow at most **four** philosophers to sit at the table at the same time.
- Allow a philosopher to pick up chopsticks only if **both** are available (pick them inside a critical section).
- Use an **asymmetric solution**: an odd-numbered philosopher picks up the left chopstick first and then the right, while an even-numbered philosopher picks up the right first and then the left.

Problem	Semaphores / variables used	Main issue illustrated
Bounded buffer	<code>mutex = 1</code> , <code>empty = n</code> , <code>full = 0</code>	Cooperation between producers and consumers on a finite buffer
Readers-writers	<code>rw_mutex = 1</code> , <code>mutex = 1</code> , <code>read_count</code>	Shared reading, exclusive writing; starvation
Dining philosophers	<code>chopstick[5]</code> , each = 1	Deadlock and starvation in resource allocation

Table 3.13: Summary of classical synchronization problems

3.5 Threads: Benefits, User and Kernel Threads

3.5.1 Thread Concept

A **thread** is a basic unit of CPU utilization. It is often called a **lightweight process**. A thread comprises a **thread ID**, a **program counter**, a **register set** and a **stack**. It shares with other threads belonging to the same process its **code section**, **data section** and other operating-system resources such as **open files and signals**. A traditional (heavyweight) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.

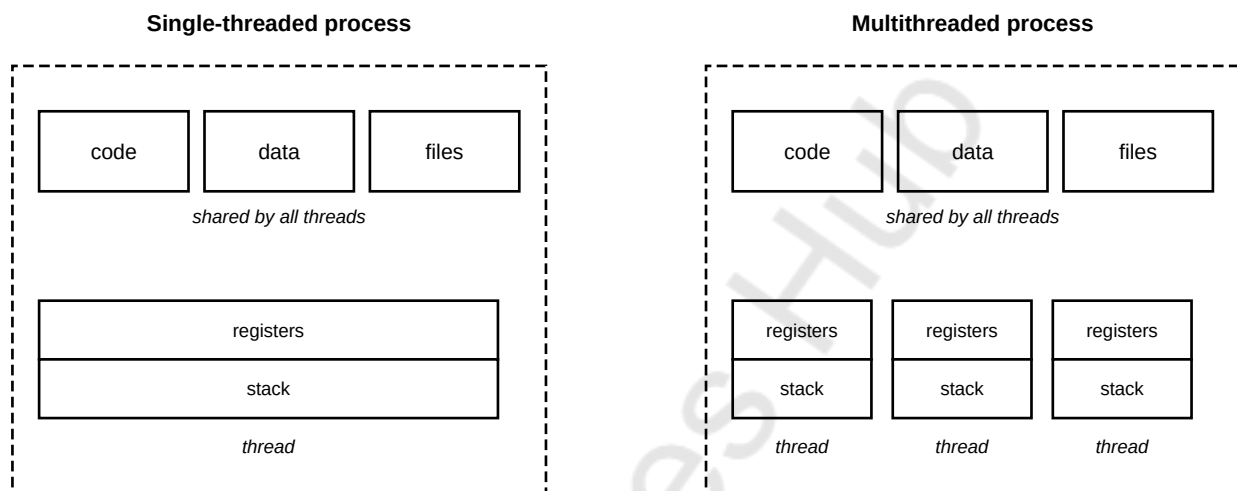


Fig. 3.9: Single-threaded and multithreaded processes

Most modern applications are multithreaded. A web browser may have one thread displaying images or text while another thread retrieves data from the network. A word processor may have one thread for display, another for keystrokes and a third for spelling and grammar checking in the background. A web server creates a separate thread to service each client request instead of creating a new process for each one.

Basis	Process	Thread
Definition	Program in execution; heavyweight.	Unit of execution within a process; lightweight.
Memory	Has its own separate address space.	Shares the address space of its process.
Creation / termination	Slow and costly.	Fast and cheap.
Context switch	Slow (memory maps must be switched).	Fast (only registers, PC and stack).
Communication	Needs IPC mechanisms (pipes, messages, shared memory).	Direct, through shared data and variables.
Isolation	Independent; failure of one rarely affects others.	A faulty thread can crash the whole process.
Own resources	Own code, data, files, registers, stack.	Own registers, program counter and stack only.

Table 3.14: Process versus thread

3.5.2 Benefits of Multithreading

Benefit	Explanation
Responsiveness	A program can continue running even if part of it is blocked or performing a lengthy operation, which increases responsiveness to the user (for example the user interface stays active while a file loads).
Resource sharing	Threads share the memory and resources of their process by default, so no special IPC techniques are required. An application can have several threads of activity in the same address space.
Economy	Creating and context-switching threads is much cheaper than for processes because memory and resources need not be allocated separately. (In Solaris, creating a process is about thirty times slower than creating a thread.)
Scalability	Threads of one process can run in parallel on different processors of a multiprocessor or multicore architecture, whereas a single-threaded process can run on only one CPU.

Table 3.15: Benefits of multithreaded programming

Concurrency versus parallelism. A concurrent system supports more than one task by allowing all of them to make progress (on a single core the CPU rapidly interleaves them). A parallel system can perform more than one task *simultaneously* and therefore needs multiple cores. Thus it is possible to have concurrency without parallelism.

3.5.3 User Threads and Kernel Threads

Support for threads may be provided either at the user level (**user threads**) or by the kernel (**kernel threads**).

User threads are supported **above the kernel** and are managed without kernel support, by a thread library in user space. The library provides support for thread creation, scheduling and management. Since the kernel is unaware of them, it treats the whole process as a single-threaded entity. Examples: POSIX Pthreads (user-level implementations), Java threads on early JVMs (green threads), GNU Portable Threads.

Kernel threads are supported **directly by the operating system**. The kernel performs thread creation, scheduling and management in kernel space. Examples: Windows, Linux, Solaris, Tru64 UNIX, macOS.

Basis	User-level threads	Kernel-level threads
Managed by	User-space thread library.	Operating-system kernel.
Kernel awareness	Kernel does not know about them; sees one process.	Kernel knows and schedules each thread.
Creation and management	Fast; no system call needed.	Slower; a system call is needed.
Context switch	Very fast; no kernel mode switch.	Slower; requires kernel mode switch.
Blocking system call	Blocks the entire process .	Only the calling thread blocks; others can run.
Multiprocessor use	Cannot run threads of one process on multiple CPUs in parallel.	Can run different threads on different CPUs in parallel.
Portability	Runs on any OS (even one without thread support).	OS specific.
Overhead	Low	Higher (kernel data structures for every thread).

Table 3.16: User threads versus kernel threads

Advantages of user threads: very fast creation and switching; can be implemented on an OS that does not support threads; the thread library can use its own scheduling algorithm. **Disadvantages:** a blocking system call by one thread blocks all threads of the process; no true parallelism on multiprocessors.

Advantages of kernel threads: true parallelism; when a thread blocks, the kernel can schedule another thread of the same application. **Disadvantages:** slower to create and manage; each thread consumes kernel resources.

3.5.4 Thread Libraries

A **thread library** provides the programmer with an API for creating and managing threads. The three main libraries are **POSIX Pthreads** (user-level or kernel-level, for UNIX/Linux/macOS), **Windows threads** (kernel-level) and **Java threads** (implemented using the thread library of the host OS). The Pthreads program below creates a thread that computes the sum of the first N integers.

```

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int sum;                                /* shared by all threads */
void *runner(void *param);              /* thread function      */

int main(int argc, char *argv[])
{
    pthread_t tid;                       /* thread identifier */
    pthread_attr_t attr;                 /* thread attributes */

    pthread_attr_init(&attr);            /* default attributes */
    pthread_create(&tid, &attr, runner, argv[1]); /* create the thread */
    pthread_join(tid, NULL);             /* wait for it       */
    printf("sum = %d\n", sum);
}

void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;
    for (i = 1; i <= upper; i++)
        sum += i;
    pthread_exit(0);
}

```

3.6 Multithreading Models

A relationship must exist between user threads and kernel threads. Three common ways of establishing this relationship are the **many-to-one**, **one-to-one** and **many-to-many** models.

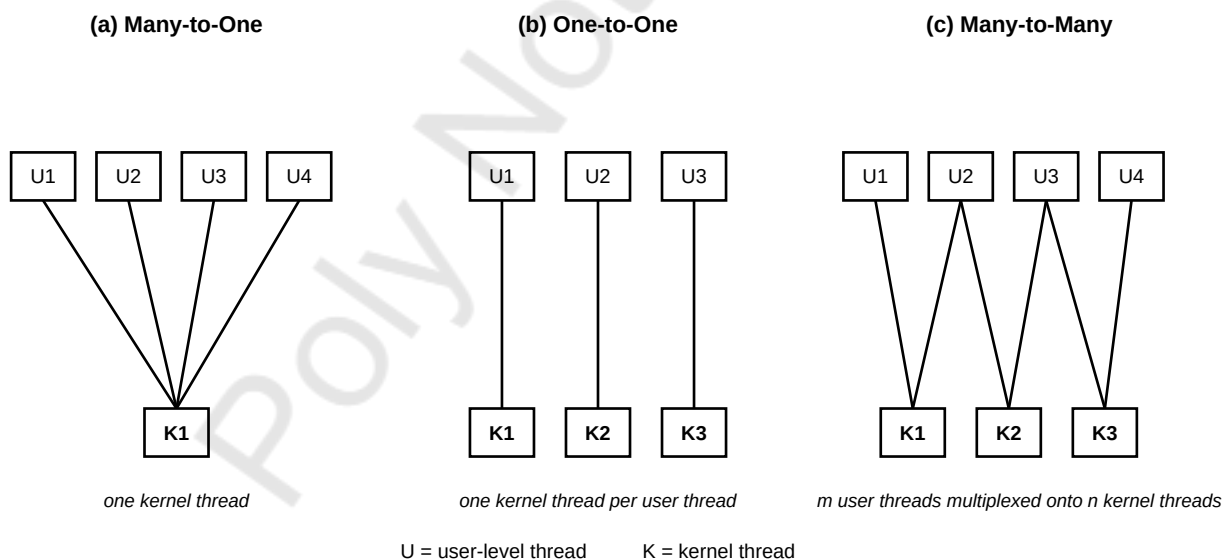


Fig. 3.10: Multithreading models

3.6.1 Many-to-One Model

The many-to-one model maps **many user-level threads to one kernel thread**. Thread management is done by the thread library in user space, so it is efficient.

- **Advantages:** efficient thread management; low overhead; needs no kernel support for threads.
- **Disadvantages:** the entire process **blocks** if one thread makes a blocking system call; because only one thread can access the kernel at a time, multiple threads **cannot run in parallel** on multiprocessors.

- **Examples:** Solaris Green Threads, GNU Portable Threads. Very few systems use this model today.

3.6.2 One-to-One Model

The one-to-one model maps **each user thread to a kernel thread**. Creating a user thread requires creating the corresponding kernel thread.

- **Advantages:** more concurrency than many-to-one because another thread can run when one makes a blocking system call; allows multiple threads to run in **parallel** on multiprocessors.
- **Disadvantages:** creating a user thread requires creating a kernel thread, and this **overhead** can burden the performance of an application; most implementations therefore **restrict the number of threads**.
- **Examples:** Linux, Windows (NT/2000/XP and later), Solaris 9 and later.

3.6.3 Many-to-Many Model

The many-to-many model **multiplexes many user-level threads to a smaller or equal number of kernel threads**. The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a uniprocessor).

- **Advantages:** developers can create as many user threads as necessary; the corresponding kernel threads can run in parallel on a multiprocessor; when a thread performs a blocking system call, the kernel can schedule another thread for execution.
- **Disadvantages:** difficult to implement; the kernel and the library must coordinate.
- **Examples:** Solaris prior to version 9, IRIX, HP-UX, Tru64 UNIX.

Two-level model. A popular variation of many-to-many that still multiplexes many user threads onto a smaller or equal number of kernel threads but also **allows a user thread to be bound to a kernel thread**. It is supported by operating systems such as IRIX, HP-UX and Tru64 UNIX.

Feature	Many-to-One	One-to-One	Many-to-Many
Mapping	Many user to 1 kernel thread	1 user to 1 kernel thread	m user to n kernel threads (m >= n)
Concurrency	Low	High	Medium to high
True parallelism on multiprocessor	No	Yes	Yes
Blocking system call	Blocks the whole process	Blocks only that thread	Blocks only that thread (kernel schedules another)
Thread creation overhead	Low	High	Medium
Limit on number of threads	None practical	Limited by kernel resources	None practical
Implementation	Simple	Simple	Complex
Examples	Green threads, GNU Portable Threads	Linux, Windows	Older Solaris, IRIX, HP-UX

Table 3.17: Comparison of multithreading models

Quick Revision Summary

- A **process** is a program in execution; its parts are text, data, heap and stack.
- Process states: **new, ready, running, waiting, terminated**. The **PCB** stores the state, PID, PC, registers, memory, scheduling, accounting and I/O information.
- Queues: job queue, ready queue, device queues. Schedulers: **long-term** (admits jobs), **short-term** (selects CPU process), **medium-term** (swaps).
- **Context switch** = save state of old process in its PCB + load state of new process; it is pure overhead.
- **fork()** creates a child (returns 0 in the child); **exec()** loads a new program; **wait()** makes the parent wait; **exit()** terminates. Zombie: terminated but not reaped; orphan: parent died.
- IPC models: **shared memory** (fast) and **message passing** (easy, kernel mediated).
- Critical-section solution needs **mutual exclusion, progress, bounded waiting**.
- **Semaphore**: integer with atomic wait() (P) and signal() (V); binary and counting types.
- Classical problems: **bounded buffer, readers-writers, dining philosophers**.
- A **thread** has its own PC, registers and stack, and shares code, data and files. Benefits: responsiveness, resource sharing, economy, scalability.
- Models: **many-to-one** (blocking blocks all), **one-to-one** (parallel but costly), **many-to-many** (flexible but complex).

Multiple Choice Questions (15)

Choose the correct option. The answer key is given after the questions.

Q1. Which of the following is not a valid state of a process?

- (a) Ready
- (b) Running
- (c) Compiling
- (d) Waiting

Q2. Which of the following is not stored in the Process Control Block?

- (a) Program counter
- (b) Process state
- (c) List of open files
- (d) Source-code listing of the program

Q3. The scheduler that controls the degree of multiprogramming is the

- (a) Short-term scheduler
- (b) Long-term scheduler
- (c) Dispatcher
- (d) I/O scheduler

Q4. The time taken by the dispatcher to stop one process and start another is called

- (a) Turnaround time
- (b) Response time
- (c) Dispatch latency
- (d) Waiting time

Q5. Which statement about a context switch is correct?

- (a) It is useful work done by the CPU
- (b) It is pure overhead
- (c) It never saves the PCB
- (d) It occurs only when a process terminates

Q6. In UNIX, the value returned by fork() to the child process is

- (a) The pid of the parent
- (b) The pid of the child
- (c) 0
- (d) -1

Q7. A process that has terminated but whose entry still exists in the process table is a

- (a) Orphan process
- (b) Zombie process
- (c) Daemon process
- (d) Swapped process

Q8. Which IPC mechanism is generally the fastest?

- (a) Message passing
- (b) Pipes
- (c) Shared memory
- (d) Sockets

Q9. Which of the following is not a requirement of a solution to the critical-section problem?

- (a) Mutual exclusion
- (b) Progress
- (c) Bounded waiting
- (d) Preemption

Q10. The wait(S) operation on a semaphore

- (a) Increments S
- (b) Decrements S (blocks the caller if no resource is available)
- (c) Sets S to zero
- (d) Deletes the semaphore

Q11. A binary semaphore can take the values

- (a) 0 and 1
- (b) Any positive integer
- (c) -1 and 1
- (d) Any integer

Q12. In the dining-philosophers problem, deadlock can occur if

- (a) Only one philosopher is hungry
- (b) All philosophers pick up their left chopstick at the same time
- (c) Philosophers pick up both chopsticks together
- (d) There are six chopsticks

Q13. Which of the following is not shared among the threads of the same process?

- (a) Code section
- (b) Data section
- (c) Open files
- (d) Stack

Q14. In which multithreading model does a blocking system call by one thread block the entire process?

- (a) One-to-one
- (b) Many-to-one
- (c) Many-to-many
- (d) Two-level

Q15. Which multithreading model gives true parallelism but requires a kernel thread for every user thread?

- (a) Many-to-one
- (b) One-to-one
- (c) Many-to-many
- (d) None of these

Answer Key for MCQs

Q. No.	Answer	Explanation
1	(c)	Compiling is not a process state; the five states are new, ready, running, waiting, terminated.
2	(d)	The PCB holds control information only, not the program source code.
3	(b)	The long-term (job) scheduler decides how many processes are admitted into memory.
4	(c)	Dispatch latency is the time the dispatcher needs to switch processes.
5	(b)	The system does no useful work during a switch; it is overhead.
6	(c)	fork() returns 0 to the child and the child's pid to the parent.
7	(b)	A zombie has terminated but has not yet been reaped by wait().
8	(c)	Shared memory avoids kernel involvement after setup, so it is fastest.
9	(d)	The three requirements are mutual exclusion, progress and bounded waiting.
10	(b)	wait() decrements the semaphore and blocks if the value becomes unavailable.
11	(a)	A binary semaphore (mutex) takes only the values 0 and 1.
12	(b)	If each philosopher holds the left chopstick, all wait for the right: circular wait.
13	(d)	Each thread has its own stack, registers and program counter.
14	(b)	Many user threads map to one kernel thread, so a blocking call blocks all.
15	(b)	One-to-one gives parallelism but creating a thread creates a kernel thread.

Long-Answer (Broad) Questions (10)

Write detailed answers with neat diagrams wherever applicable.

- Q1.** Define a process. Differentiate between a program and a process. Explain the different states of a process with a neat state transition diagram.
- Q2.** What is a Process Control Block (PCB)? Draw its structure and explain each field in detail.
- Q3.** Explain scheduling queues with the help of a queueing diagram. Describe the long-term, short-term and medium-term schedulers and compare them.
- Q4.** What is a context switch? Explain the steps involved with a diagram. Why is context-switch time considered an overhead? Also explain the role of the dispatcher.
- Q5.** Explain process creation and process termination in UNIX using `fork()`, `exec()`, `wait()` and `exit()`. Write a C program to illustrate. What are zombie and orphan processes?
- Q6.** Explain the two models of inter-process communication, shared memory and message passing, with diagrams. Compare them. Describe the design issues of message passing.
- Q7.** What is a race condition? Explain the critical-section problem and the three requirements for its solution. Explain Peterson's solution and prove that it satisfies these requirements.
- Q8.** What is a semaphore? Explain the `wait()` and `signal()` operations and the types of semaphores. Give a solution to the bounded-buffer, readers-writers and dining-philosophers problems using semaphores.
- Q9.** What is a thread? List the benefits of multithreading. Differentiate between a process and a thread, and between user-level threads and kernel-level threads.
- Q10.** Explain the multithreading models (many-to-one, one-to-one and many-to-many) with neat diagrams, listing the advantages, disadvantages and examples of each.

- End of Chapter 3: Process Management -