Unit – 1 :: Chapter - 1
Operating Systems Overview

1. Introduction

1.1. Operating System – Definition:

Operating System is a program that acts as intermediary between user programs and computer hardware.

{OR}

An operating system is a program that manages the computer hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware.

Examples:

Microsoft Windows, Linux, iOS, Mac OS, Andriod, etc…,

1.2. Computer System Structure:

Computer system can be divided into four components, as shown in below figure:

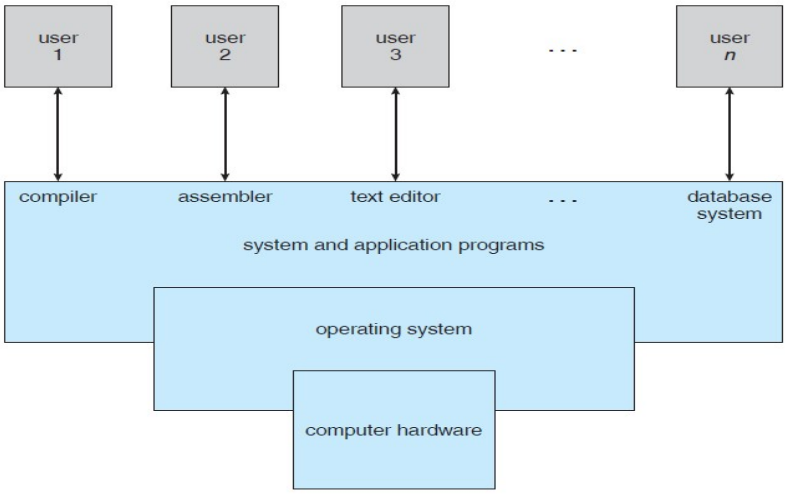


Figure Abstract view of the components of a computer system

1. **Hardware** – provides basic computing resources CPU, memory, I/O devices.
2. **Operating system –** Controls and coordinates use of hardware among various applications and users.
3. **Application programs** –Application programs are performing specific task. Examples are word processors, compilers, web browsers, database systems, video games etc...,.
4. **Users** of the computer.

1.3. Operating system goals:

1. Execute user programs and make solving user problems easier.
2. Make the computer system convenient to use. (Ease of use)
3. Use the computer hardware in an efficient manner. (Resource utilization)

2. Operating system functions

2.1. Operating System Functions:

Operating systems have many functions:

1. **User interface**
   - Provides an interface for computer interaction.
   - Control of inputs and outputs.
   - Without it, most computers would be too difficult for the average person to use.
2. **Management of hardware and peripherals**
   - The operating system is responsible for controlling all the devices connected to the computer. It tells them how to interact and operate correctly.
   - Device drivers are used to manage these connections.
3. **Processor management**
   - The operating system manages the CPU.
   - When software is opened, the OS finds it and loads it into memory (RAM).
   - The CPU can then be instructed to execute the program.
   - The operating system will manage the sharing of processor time.
4. **Interrupt and error handling**
   - Several programs can be stored in RAM at the same time, but the processor can only process one at a time.
   - Through the use of regular interruption signals, the OS can prioritise the requests made to the processor.
   - This gives the illusion that the CPU is actually dealing with more than one program or task at once (multitasking).
   - Errors are spotted and usually alert the user.
5. **Memory management**
   - The operating system is responsible for transferring programs to and from memory.
   - It keeps track of memory usage, and decides how much should be given to each program.
   - The OS also decides what happens if there is not enough memory.
6. **File management**
   - A file system is created to organise files and directories.
   - This gives programs a consistent way to store and retrieve data.
   - The OS is also responsible for the naming, sorting, deleting, moving and copying of files (at the request of the user).
   - Look-up tables are used to relate file names to storage locations.
7. **Security**
   - The operating system is responsible for the creation and application of user accounts and passwords.
   - An OS also comes with many utility programs, including firewalls.

3. Operating systems operations

3.1. Operating systems operations

Modern operating systems are **interrupt driven**. If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen.

An **interrupt** is a signal emitted by hardware or software when a process or an event needs immediate attention.

There are two types of interrupts, they are:

1. **Hardware Interrupts**
2. **Software Interrupts** (also called **trap or exception**)

Hardware interrupt is an interrupt generated from an external device or hardware. For example, pressing a keyboard key or moving mouse that causes the processor to read the keystroke or mouse position

A **trap (or an exception or software interrupt)** is a software-generated interrupt caused either by an error. For example, division by zero or invalid memory access.

The interrupt-driven nature of an operating system defines that system's general structure. For each type of interruption, separate segments of code in the operating system determine which action to take. An **interrupt service routine** is provided that is responsible for dealing with the interrupt.

The operating systems share the hardware and software resources of the computer system to user programs. If there is an error in a user program could cause problems only for the one program or many processes could be affected. For example, if a process gets stuck in an infinite loop, this loop could prevent the correct operation of many other processes.

More subtle errors can occur in a multiprogramming system, where one erroneous program might modify the data of another program, or even the operating system itself.

To protect these types of errors, operating system will run in **dual-mode operation**.

1. Dual-Mode Operation

Dual-mode operation allows OS to protect itself and other system components. Dual-mode operation consists of two modes:

1. User Mode
2. Kernel Mode (also called Supervisor Mode, System Mode, Or Privileged Mode)

A **mode bit** is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1).With the mode bit, we are able to distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user.

At system boot time the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the

operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program. This is shown in below figure.

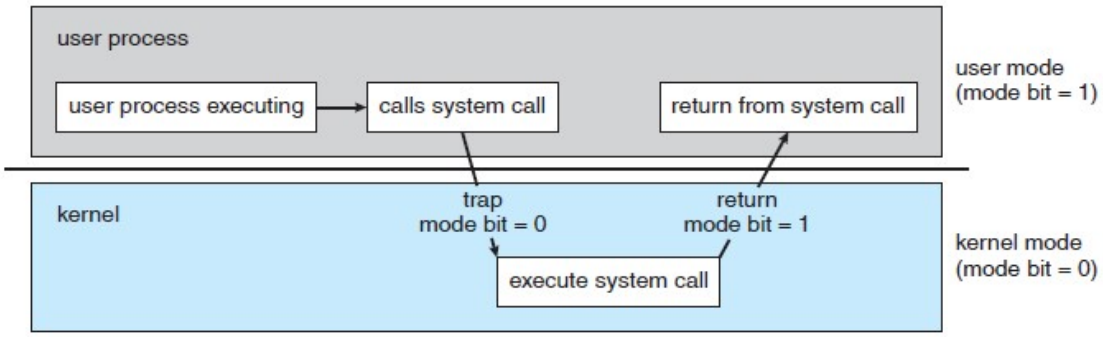


Figure: Transition from user to kernel mode.

The dual mode of operation protects the operating system from errant users. We accomplish this protection by specifying some machine instructions that may cause harm as **privileged instructions**. The instruction to switch to kernel mode, I/O control, timer management, and interrupt management are few examples of a privileged instruction.

The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

Life cycle of instruction:

Initial control resides in the operating system, where instructions are executed in kernel mode. When control is given to a user application, the mode is set to user mode. Eventually, control is switched back to the operating system via an interrupt, a trap, or a system call.

2. Timer

Timer prevents a user program from running too long.

A timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second).

A variable timer is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs.

3.2. Differences between hardware interrupt and software interrupt:

S.No.	Hardware Interrupt	Software Interrupt
1	Hardware interrupt is an interrupt generated from an external device or hardware.	Software interrupt is the interrupt that is caused either by an error.
2	It does not increment the program counter.	It increment the program counter.
3	It has lowest priority than software interrupts	It has highest priority among all interrupts.
4	Hardware interrupt is triggered by external hardware and is considered one of the ways to communicate with the outside peripherals, hardware.	Software interrupt is triggered by software and considered one of the ways to communicate with kernel or to trigger system calls.
5	It is an asynchronous event.	It is synchronous event.
6	Keystrokes and mouse movements are examples of hardware interrupt.	All system calls are examples of software interrupts

4. Computing environments

4.1. Computing Environments:

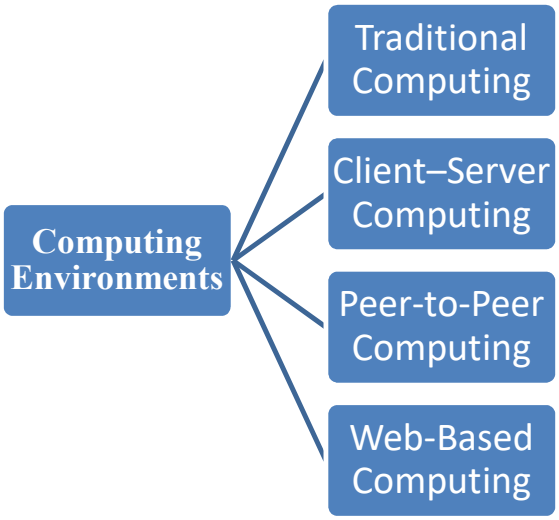


Figure: Types of Computing Environments

1. **Traditional Computing:**

As computing matures, the lines separating many of the traditional computing environments are blurring.

Office environment

A few years ago, this environment consisted of PCs connected to a network, with servers providing file and print services. Terminals attached to mainframes providing batch and timesharing.

Now a day's the environment is **web based computing**, Companies establish portals, which provide web accessibility to their internal servers. Network computers are the terminals.

Home networks

At home, most users had a single computer with an internet connection to the office. Some homes even have firewalls to protect their networks from security breaches

Time-sharing systems

Time-sharing systems used a timer and scheduling algorithms to cycle processes rapidly through the CPU, giving each user a share of the resources.

2. **Client–Server Computing:**

Personal Computers (Clients) connected to Servers (Centralized Systems). Server systems satisfy requests generated by client systems. This is called a client–server system, has the general structure depicted in below Figure.

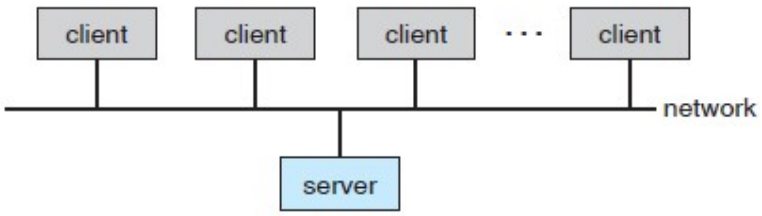


Figure : General structure of a client–server system.

Types of Server Systems

Server systems can be broadly categorized into 2 types:

1. Compute-Server System
2. File-Server System

1. **Compute-Server System:**

In compute-server system, a client can send a request to perform an action; the server executes the action and sends back results to the client. For example read data.

These are the database systems.

2. **File-Server System:**

In file-server system, clients can create, update, read, and delete files. An example of such a system is a Web server that delivers files to clients running Web browsers.

3. Peer-to-Peer Computing:

In this model, clients and servers are not distinguished from one another; instead, all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service.

To participate in a peer-to-peer system, a node must first join the network of peers. Once a node has joined the network, it can begin providing services to other nodes and requesting services from other nodes in the network.

The services availability is determined one of two ways:

1. Registers its service with central lookup service on network, or
2. Broadcast request for service and respond to requests for service via discovery protocol

Examples peer-to-peer computing are Napster and Gnutella.

4. Web-Based Computing:

- The Web has become ubiquitous. PCs, PDAs, cell phones are most prevalent access devices.
- More devices becoming networked to allow web access.
- New category of devices to manage web traffic among similar servers: load balancers
- Use of operating systems like Windows 95, client-side, have evolved into Linux and Windows XP, which can be clients and servers

5. Open-Source Operating Systems

5.1. Open-Source Operating System Definition:

- Open-source operating systems are available in source-code format rather than as compiled binary code.
- **Examples:** Linux, BSD UNIX, Solaris, etc.

5.2. Benefits of Open-Source Operating System:

1. Programmers, who contribute to the code by helping to debug it, analyze it, provide support, and suggest changes.
2. With the source code in hand, a student can modify the operating system and then compile and run the code to try out those changes, which is another excellent learning tool.
3. Open-source code is more secure than closed-source code.
4. Companies that earn revenue from selling their programs tend to be hesitant to open-source their code.

5.3. History:

- Richard Stallman in 1983 started the GNU project to create a free, open-source UNIX compatible operating system.
- In 1985, he published the GNU Manifesto, which argues that all software should be free and open-sourced.
- He also formed the **Free Software Foundation (FSF)** with the goal of encouraging the free exchange of software source code and the free use of that software.
- Rather than copyright its software, the FSF "copylefts" the software to encourage sharing and improvement.
- The GNU General Public License (GPL) codifies copylefting and is common license under which free software is released.

5.4. Linux:

- As an example of an open-source operating system, consider GNU/Linux.
- The GNU project produced many UNIX-compatible tools, including compilers, editors, and utilities, but never released a kernel.
- In 1991, a student in Finland, Linus Torvalds, released a rudimentary UNIX-like kernel using the GNU compilers and tools and invited contributions worldwide.
- The advent of the Internet meant that anyone interested could download the source code, modify it, and submit changes to Torvalds.
- Releasing updates once a week allowed this so-called Linux operating system to grow rapidly, enhanced by several thousand programmers.

5.5. BSD UNIX:

- BSD UNIX has a longer and more complicated history than Linux.
- It started in 1978 as a derivative of AT&T's UNIX.
- Releases from the University of California at Berkeley came in source and binary form, but they were not open-source because a license from AT&T was required.
- BSDUNIX's development was slowed by a lawsuit by AT&T, but eventually a fully functional open-source version, 4.4BSD-lite, was released in 1994.

5.6. Solaris:

- Solaris is the commercial UNIX-based operating system of Sun Microsystems.
- Originally, Sun's SunOS operating system was based on BSD UNIX.
- Sun moved to AT&T's System V UNIX as its base in 1991.
- In 2005, Sun open-sourced some of the Solaris code, and over time, the company has added more and more to that open-source code base.
- Unfortunately, not all of Solaris is open-sourced, because some of the code is still owned by AT&T and other companies.
- However, Solaris can be compiled from the open source and linked with binaries of the close-sourced components, so it can still be explored, modified, compiled, and tested.

5.7. Differences between Open Source Software and Free Source Software:

S.No.	Open Source Software	Free Source Software
1	Freely available and Source also available (can be edited to make your own copyright)	Freely available but copyrighted.
2	It has distribution of License.	Freedom to run program for any purpose.
3	All Open Source Software under free software terminology.	But all free software does not come under open source terminology.
4	It is good for your business.	It is good for morality of society.

5.8. Differences between Open Source Software and Closed Source Software:

S.No.	Open Source Software	Closed Source Software
1	Open source software refers to the computer software which source is open means the general public can access and use.	Closed source software refers to the computer software which source code is closes means public is not given access to the source code.
2	This code can be modified by other users and organizations means that the source code is available for anyone to look at.	The only individual or organization who has created the software can only modify the code.
3	The price of open source software is very less.	The price of closed source software is high.
4	Programmers freely provide improvement for recognition if their improvement is accepted.	Programmers are hired by the software firm/organization to improve the software.
5	Open software can be installed into any computer.	Closed software needs have a valid license before installation into any computer.
6	Open source software fails fast and fix faster.	Closed source software has no room for failure.
7	In opened source software no one is responsible for the software.	In closed source software the vendor is responsible if anything happened to software.
8	**Examples of Open-source Operating System:** Linux, BSD UNIX, Solaris, etc.	**Examples of Closed-source Operating System:** Microsoft Windows etc.
9	**Examples of Open-source software** are Firefox, OpenOffice, Gimp, Alfresco, Android, Zimbra, Thunderbird, MySQL, Mailman, Moodle, TeX, Samba, Perl, PHP, KDE etc.	**Examples of Closed-source software** are Skype, Google earth, Java, Adobe Flash, Virtual Box, Adobe Reader, Microsoft office, , WinRAR, mac OS, Adobe Flash Player etc.0

Unit – 1 :: Chapter - 2
System Structures

1. Operating System Services

1.1. Operating System Services:

Operating System provides certain services to programs and to the users of those programs. Below Figure shows one view of the various operating-system services and how they interrelate.

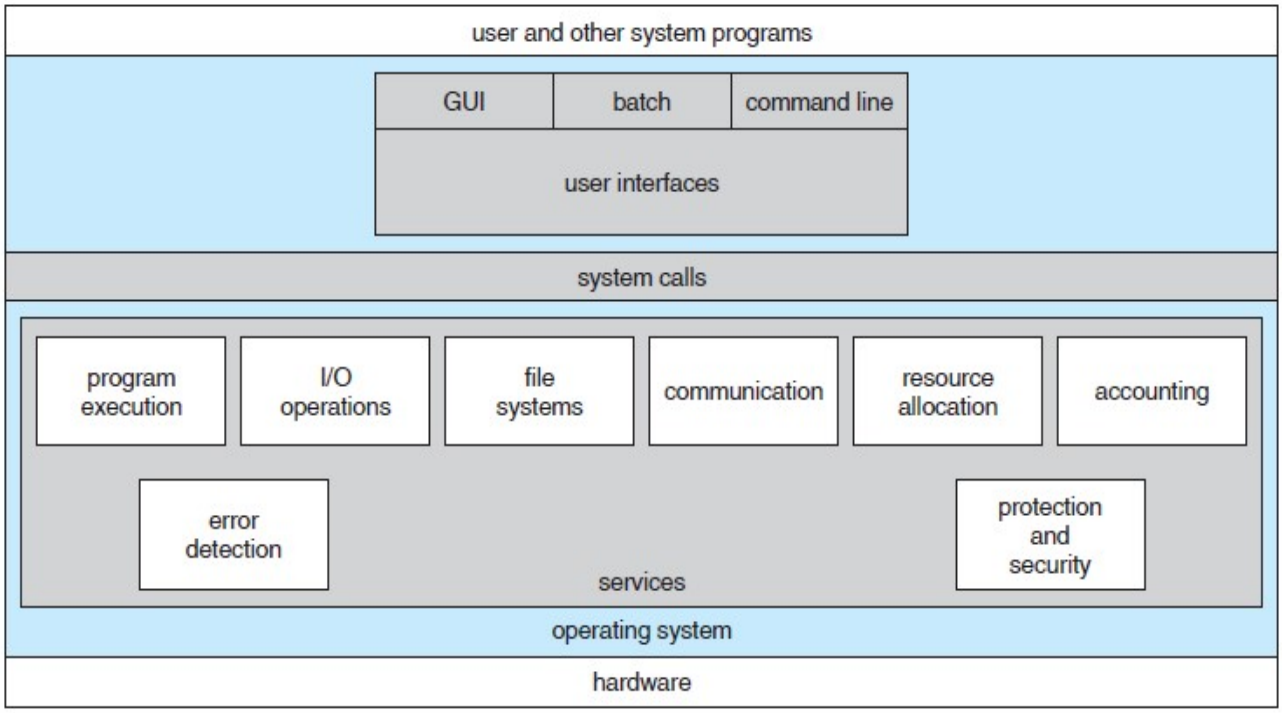


Figure: A view of operating system services.

1. **User interface**:

Almost all operating systems have a user interface (UI). This interface can take several forms.

1. **Command-line interface** (CLI), which uses text commands and a method for entering them.
2. **Batch interface**, in which commands and directives to control those commands are entered into files, and those files are executed.
3. **Graphical user interface (GUI)** is used. Here, the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text.

Some systems provide two or all three of these variations.

2. **Program execution:**

The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).

3. **I/O operations:**

A running program may require I/O, which may involve a file or an I/O device. For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.

4. **File-system manipulation:**

Programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information.

Finally, some programs include permissions management to allow or deny access to files or directories.

Many operating systems provide a variety of file systems, sometimes to allow personal choice, and sometimes to provide specific features or performance characteristics.

5. **Communications:**

Let consider, two or more processes are executing on a same computer system or different computer systems connected together by a computer network. If one process needs to exchange information with another process, this communication could be implemented by operating system via shared memory or through message passing.

6. **Error detection:**

The operating system needs to be constantly aware of possible errors. Errors may occur:

1. In the CPU and memory hardware (such as a memory error or a power failure),
2. In I/O devices (such as a parity error on tape, a connection failure on a network, or lack of paper in the printer), and
3. In the user program (such as an arithmetic overflow, an attempt to access an illegal memory location, or a too-great use of CPU time).

For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.

7. **Resource allocation:**

When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. Many different types of resources are managed by the operating system, such as CPU cycles, main memory, file storage, I/O devices, printers, modems, USB storage drives, and other peripheral devices.

8. **Accounting:**

We want to keep track of which users use how much and what kinds of computer resources. This record keeping may be used for accounting or accumulating usage statistics.

9. **Protection and security:**

- **Protection –** any mechanism for controlling access of processes or users to resources defined by the OS.
- **Security –** defines of the system against internal and external attacks.
- If a system is to be protected and secure, precautions must be instituted throughout it.

2. User and Operating-System Interface

2.1. User and Operating System Interface:

Here, we discuss two fundamental user and operating system interfaces:

1. Command Interpreter or Command Line Interface(CLI) or Character User Interface(CUI):

- It allows users to directly enter commands to be performed by the operating system
- In some systems command interpreters are known as shells. For example on UNIX and Linux systems has different shells such as Bourne shell, C shell, Bourne-Again shell, Korn shell, and others. User may choose any one.
- Command interpreter primarily fetches a command from user and executes it. Sometimes commands built-in, sometimes just names of programs.
- Example of CLI based Operating Systems: MS DOS, Unix, etc.

Advantages:

1. It uses less memory compare to GUI.
2. It is less expensive.

Disadvantages:

1. In CLI, only one task can do at a time.
2. Only supports the usage of keyboard.
3. User must remember the commands to use the system.

2. Graphical User Interfaces:

- It allows users to interface with the operating system via a graphical user interface, or GUI.
- It Provides User-friendly desktop metaphor interface - The user moves the mouse to position its pointer on images, or icons, on the screen (the desktop) that represent programs, files, directories, and system functions.
- Depending on the mouse pointer's location, clicking a button on the mouse can invoke a program, select a file or directory—known as a folder—or pull down a menu that contains commands.
- Various mouse buttons over objects in the interface cause various actions.
- Example of GUI based Operating Systems:
    - Microsoft Windows, macOS, Ubuntu Unity, and GNOME Shell for desktops.
    - Android, Apple's iOS, Black Berry OS, Windows 10 Mobile, Palm OS-WebOS, and Firefox OS for smart phones.

Advantages:

1. Easy to use.
2. More than one task can do at a time (Multiprocessing).

Disadvantages:

1. Uses more memory compare to CLI.
2. Development of OS is more complex.
3. More expensive.

2.2. Differences between CUI and GUI

S.No.	CUI (Character User Interface)	GUI (Graphical User Interface)
1	The user interacts with the computer using commands like text.	The user interacts with the system using Graphics like icons, images.
2	Difficult to use.	Easy to use.
3	It has high speed.	It has a low speed.
4	It has a low memory requirement.	It has a high memory requirement.
5	Keyboard is used typing commands.	Keyboard and Mouse are used to navigate.

Note:

Many systems now include both CLI and GUI interfaces:

1. Microsoft Windows is GUI with CLI "command" shell.
2. Apple Mac OS X as "Aqua" GUI interface with UNIX kernel underneath and shells available.
3. Solaris is CLI with optional GUI interfaces (Java Desktop, KDE).

3. Systems calls

3.1. System Calls:

Definition:

- **System calls** provide an interface to the services provided by an operating system
- Typically system calls written in a high-level language (C or C++).
- **Example:**
  - Let consider, copy one file to another file. The sequence of system calls invoked in this example is shown in below figure:

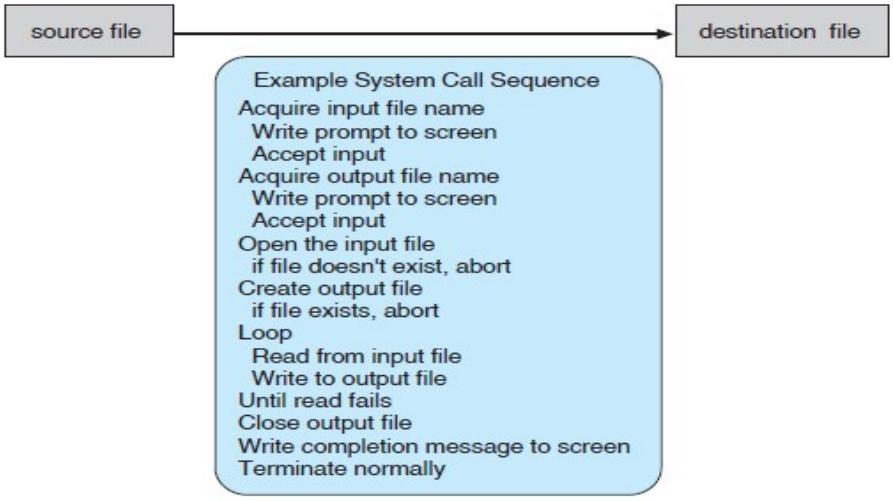


Figure: Example of how system calls are used.

Why use APIs rather than system calls?

- The developers design system calls using Application Program Interface (API) rather than direct development.
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
- **API Example:**

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `ReadFile()` function in the Win32 API—a function for reading from a file. The API for this function appears in Figure 2.5.

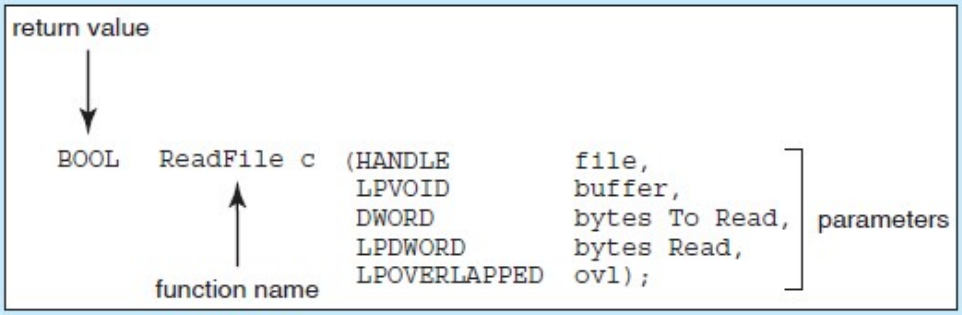


Figure: The API for the `ReadFile()` function.

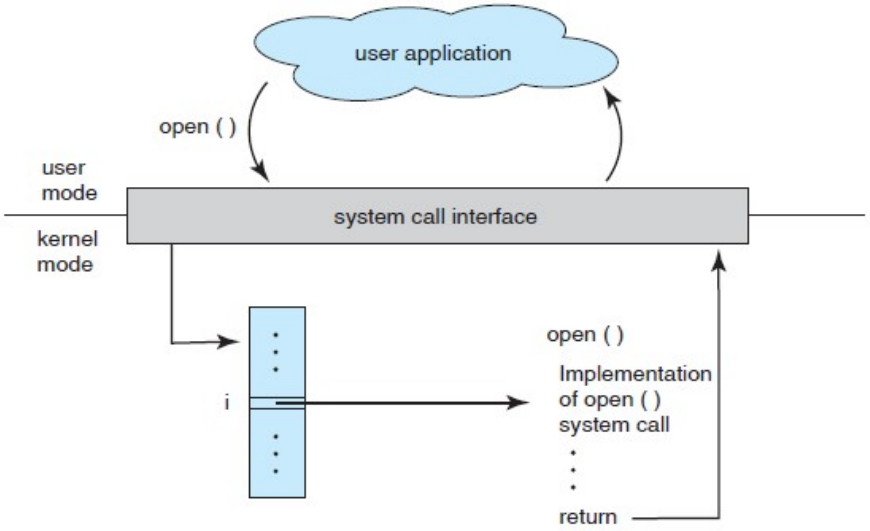
A description of the parameters passed to `ReadFile()` is as follows:

- HANDLE file—the file to be read
- LPVOID buffer—a buffer where the data will be read into and written from
- DWORD bytesToRead—the number of bytes to be read into the buffer
- LPDWORD bytesRead—the number of bytes read during the last read
- LPOVERLAPPED ovl—indicates if overlapped I/O is being used

System Call Implementation:

- Typically, a number associated with each system call.
- System-call interface maintains a table indexed according to these Numbers
- The **system call interface** invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
- Just needs to obey API and understand what OS will do as a result call.
- Most details of OS interface hidden from programmer by API and managed by run-time support library (set of functions built into libraries included with compiler).

- The below figure shows API – System call – OS Relationship.



System Call Parameter Passing

- Often, more information is required than simply identity of desired system call.
- Exact type and amount of information vary according to OS and call
- **Three general methods** used to **pass parameters to the OS**
  1. Simplest approach is to pass the parameters in registers.
  2. In some cases, may be more parameters than registers, parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register.
     This approach taken by Linux and Solaris.
  3. Parameters are placed, or pushed onto the stack by the program and popped off the stack by the operating system.
- Block and stack methods do not limit the number or length of parameters being passed.
- Parameter Passing via Table is shown in below figure:

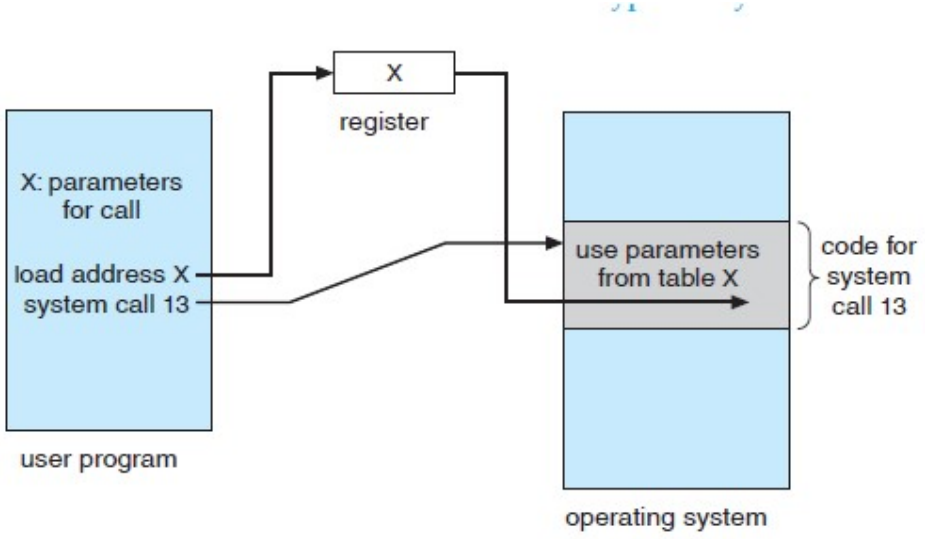


Figure: Passing of parameters as a table.

4. Types of System Calls

4.1. Types of System Calls:

System calls are categorised into 6 types:

1. Process control
2. File management
3. Device management
4. Information maintenance
5. Communications
6. Protection

1. Process Control:

This system calls perform the task of process creation, process termination, etc.

Functions:

- End and Abort
- Load and Execute
- Create Process and Terminate Process
- Wait and Signed Event
- Allocate and free memory

2. File management:

File management system calls handle file manipulation jobs like creating a file, reading, and writing, etc.

Functions:

- Create a file
- Delete file
- Open and close file
- Read, write, and reposition
- Get and set file attributes

3. Device management:

A process may need several resources to execute—main memory, disk drives, access to files, and so on. The various resources controlled by the operating system can be thought of as devices. Some of these devices are physical devices (for example, disk drives), and others are abstract or virtual devices (for example, files).

Functions:

- request device, release device
- read, write, reposition
- get device attributes, set device attributes
- logically attach or detach devices

4. Information maintenance:

It handles information and its transfer between the OS and the user program.

Functions:

- Get or set time and date
- Get process and device attributes

5. Communications:

There are two common models of inter-process communication:

1. Message passing model
2. Shared-memory model.

In the message-passing model, the communicating processes exchange messages with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox.

In the shared-memory model, processes use shared memory for communication. create and attach shared memory system calls are used to create and gain access to regions of memory owned by other processes.

Functions:

- create, delete communication connection
- send, receive messages
- transfer status information
- Attach or detach remote devices

6. Protection:

Protection provides a mechanism for controlling access to the resources provided by a computer system.

System calls providing protection include **set permission and get permission**, which manipulate the permission settings of resources such as files and disks.

The **allow user and deny user** system calls specify whether particular users can—or cannot—be allowed access to certain resources.

4.2. Process Control in Single Tasking (MS DOS) and Multi-Tasking (FreeBSD) Systems:

Process Control in MS DOS:

- The MS-DOS operating system is an example of a **single-tasking system**.
- It has a command interpreter that is invoked when the computer is started, shown in below figure.

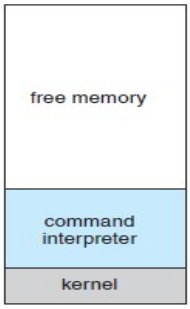


Figure : MS-DOS execution :: At system startup

- Because MS-DOS is single-tasking, it uses a simple method to run a program and does not create a new process. It loads the program into memory, writing over most of itself to give the program as much memory as possible, shown in below figure.

Figure : MS-DOS execution :: Running a program.

- Next, it sets the instruction pointer to the first instruction of the program. The program then runs, and either an error cause a trap, or the program executes a system call to terminate.
- In either case, the error code is saved in the system memory for later use.
- Following this action, the small portion of the command interpreter that was not overwritten resumes execution.
- Its first task is to reload the rest of the command interpreter from disk. Then the command interpreter makes the previous error code available to the user or to the next program.

Process Control in FreeBSD:

- FreeBSD (derived from Berkeley UNIX) is an example of a **multitasking system**.
- When a user logs on to the system, the shell of the user's choice is run. This shell is similar to the MS-DOS shell in that it accepts commands and executes programs that the user requests.
- Since FreeBSD is a multitasking system, the command interpreter may continue running while another program is executed, as shown in below figure.

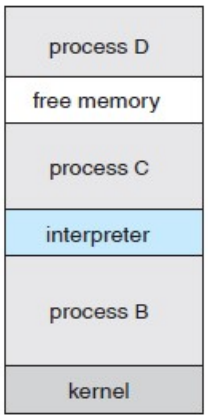


Figure : FreeBSD running multiple programs.

- To start a new process, the shell executes a fork() system call. Then, the selected program is loaded into memory via an exec() system call, and the program is executed.
- Depending on the way the command was issued, the shell then either waits for the process to finish or runs the process "in the background."
- When the process is done, it executes an exit() system call to terminate, returning to the invoking process a status code of 0 or a nonzero error code.
- This status or error code is then available to the shell or other programs.

5. System programs

5.1. System Programs:

- A modern system is the collection of system programs. See this below figure:

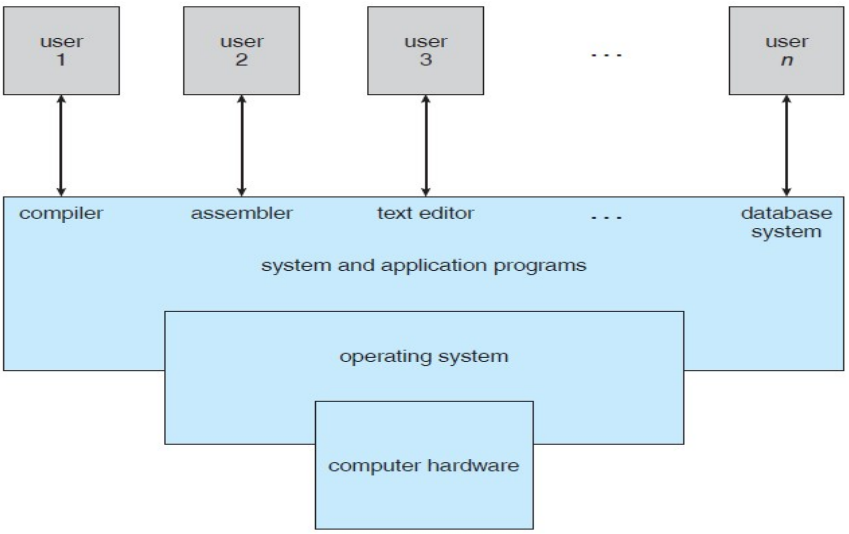


Figure Abstract view of the components of a computer system

- **System programs, also known as system utilities**, provide a convenient environment for program development and execution.
- Some of them are simply user interfaces to system calls; others are considerably more complex. They can be divided into these categories:
    1. File management
    2. Status information
    3. File modification
    4. Programming-language support
    5. Program loading and execution
    6. Communications

1. File management:

These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.

2. Status information:

Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed performance, logging, and debugging information.

Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a registry, which is used to store and retrieve configuration information.

3. File modification:

Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.

4. Programming-language support:

Compilers, assemblers, debuggers and interpreters for common programming languages (such as C, C++, Java, Visual Basic, and PERL) are often provided to the user with the operating system.

5. Program loading and execution:

Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, re-locatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed as well.

6. Communications:

These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse web pages, to send electronic-mail messages, to log in remotely, or to transfer files from one machine to another.

- In addition to systems programs, most operating systems are supplied with programs that are useful in solving common problems or performing common operations. Such programs include web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and statistical-analysis packages, and games. These programs are known as **system utilities or application programs**.

6. Operating system Design and Implementation

6.1. Operating system Design and Implementation:

In this section, we discuss problems we face in designing and implementing an operating system.

1. Design Goals:

- The first problem in designing a system is to define goals and specifications.
- At the highest level, the design of the system will be affected by the choice of hardware and the type of system: batch, time shared, single user, multiuser, distributed, real time, or general purpose.
- Beyond this highest design level, the requirements may be much harder to specify. The requirements can be divided into two basic groups: user goals and system goals.
- User goals are:
    - Convenient to use
    - Easy to learn and to use
    - Reliable, safe, and fast
- System goals are:
    - Easy to design, implement, and maintain
    - Flexible, reliable, error free, and efficient.
- No unique solution to the problem of defining the requirements for an operating system.
- Specifying and designing an operating system is a highly creative task. General principles have been developed in the field of software engineering.

2. Mechanisms and Policies

- One important principle is the separation of policy from mechanism. **Mechanisms determine how to do something; policies determine what will be done**.
- For example, the timer construct is a mechanism for ensuring CPU protection, but deciding how long the timer is to be set for a particular user is a policy decision.
- Policies are likely to change across places or over time. In the worst case, each change in policy would require a change in the underlying mechanism.
- A change in policy would then require redefinition of only certain parameters of the system.
- Microkernel-based operating systems take the separation of mechanism and policy to one extreme by implementing a basic set of primitive building blocks. These blocks are almost policy free and policies to be added via user-created kernel modules or via user programs themselves.
- Policy decisions are important for all resource allocation. Whenever it is necessary to decide whether a resource to allocate or not.
- Whenever the question is how rather than what, it is a mechanism that must be determined.

3. Implementation

- Once an operating system is designed, it must be implemented. Now days, the operating systems are written in higher-level languages such as C or C++.
- The first operating system that was written in high level language was Master Control Program (MCP) for Burroughs computers. MCP was written in a variant of ALGOL. MULTICS, developed at MIT.
- The Linux and Windows operating systems are written mostly in C, although there are some small sections of assembly code for device drivers and registers.
- The advantages of implementing an operating system in a higher-level language are:
    - The code can be written faster
    - More compact
    - Easier to understand and debug.
    - OS is easier to port—to move to some other hardware—
- For example, MS-DOS was written in Intel 8088 assembly language. Consequently, it runs natively only on the Intel X86 family of CPUs.
- The Linux operating system, is written mostly in C and is available natively on a number of different CPUs, including Intel X86, Sun SPARC, and IBMPowerPC.
- The disadvantages of implementing an operating system in a higher-level language are:
    - Increased storage requirements..
- An operating system performance is improved when data structures and algorithms are implemented in it.
- Operating systems are large, only a small amount of the code is critical to high performance;
    - The memory manager
    - The CPU scheduler

7. Operating system structure

7.1. Operating system structure

- Operating system can be implemented with the help of various structures.
- The **operating system defines** how the various components of the operating system are interconnected and melded into the kernel.
- Depending on this we have following structures of the operating system:
    1. Simple Structure
    2. Layered Approach
    3. Micro-kernels
    4. Modules

1. Simple Structure:

- The Simple Structured operating systems do not have well defined structure and are small, simple and limited systems.
- The interfaces and levels of functionality are not well separated.
- MS-DOS is an example of such operating system.
- In MS-DOS, application programs are able to access the basic I/O routines to write directly to the display and disk drives.
- These types of operating system cause the entire system to crash if one of the user programs fails.
- The below figure illustrates structure of MS-DOS:

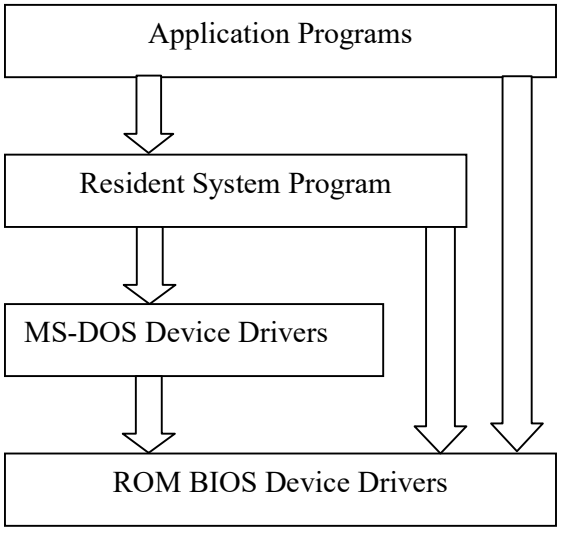


Figure: MS DOS Layer Structure

- Another example of simple structured operating system is the original UNIX operating system.
- UNIX operating system consists of two parts: the kernel and the system programs.
- The kernel is further separated into a series of interfaces and device drivers, as shown in below figure:

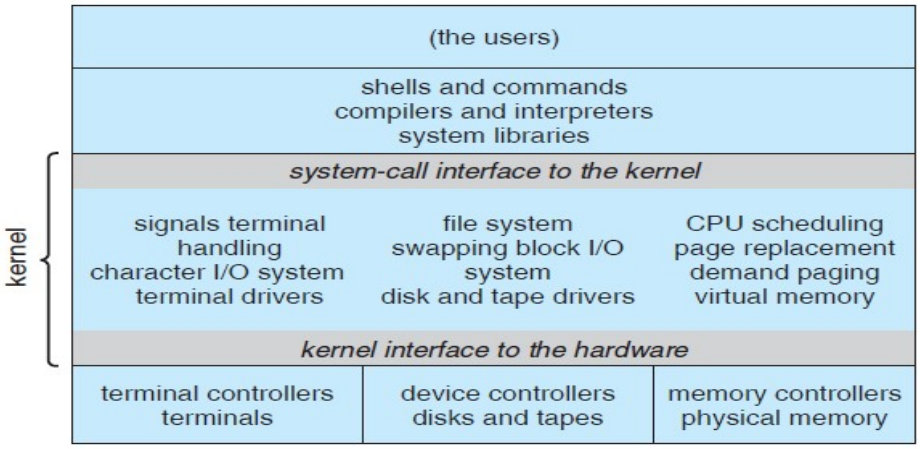


Figure: Traditional UNIX system structure.

2. Layered Approach:

- In this layered approach, structure the operating system is broken into number of layers (levels).
- The bottom layer (layer 0) is the hardware and the topmost layer (layer N) is the user interface.
- Each layer uses the functions of the lower level layers only.
- The below figure illustrates the layered structure of the operating system.

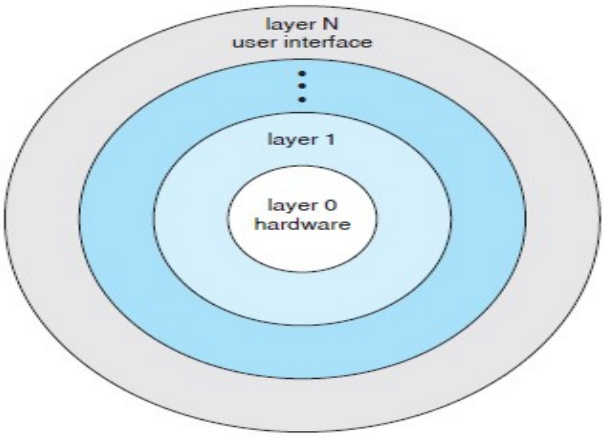


Figure: A layered operating system.

- **Advantages:**
  - This simplifies the debugging process as if lower level layers are debugged and an error occurs during debugging then the error must be on that layer only.
  - Operating system can retain greater control over the computer and over the applications that make use of the computer.
- **Disadvantages:**
  - Each layer, the data needs to be modified and passed on which adds overhead to the system.
  - Less efficient than other structures
  - Careful planning of the layers and implementation is necessary.

3. **Micro-kernels:**

- The micro-kernels structure designs the operating system by removing all non-essential components from the kernel and implementing them as system and user programs. This result in a smaller kernel called the micro-kernel.
- Communication takes place between user modules using message passing.
- Tru64 UNIX is implemented with Mach kernel and Mac OS X kernel (also known as Darwin) is implemented with Mach micro kernel.
- Mac OS X structure is illustrated in below figure:

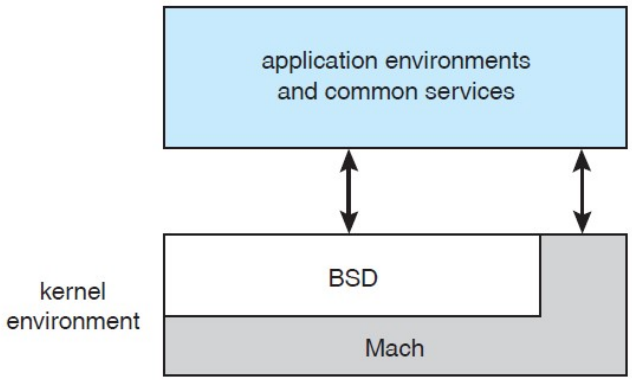


Figure: The Mac OS X structure.

- In the above figure:
  - The top layers include application environments and a set of services providing a graphical interface to applications.
  - Below these layers is the kernel environment, which consists primarily of the Mach microkernel and the BSD kernel.
  - Mach provides memory management, support for remote procedure calls (RPCs) and inter-process communication (IPC).
  - The BSD component provides a BSD command line interface, support for networking and file systems, and an implementation of POSIX APIs, including Pthreads
- **Advantages**:
  - All new services need to be added to user space and does not require the kernel to be modified.
  - More secure and reliable (less code is running in kernel mode).
  - Easier to extend a microkernel.
- .**Disadvantages:**
  - Performance and communication overhead of user space to kernel space.

4. **Modules:**
   - Most modern operating systems implement kernel modules Uses object-oriented approach to create modular kernel.
   - The kernel has only set of core components and other services are added as dynamically loadable modules to the kernel either during run time or boot time.
   - It resembles layered structure
   - Each core component is separate; each talks to the others over known interfaces.
   - Modular structure is similar to layered structure but with more flexible.
   - For example, the Solaris operating system structure, shown in below figure, is organized around a core kernel with seven types of loadable kernel modules.

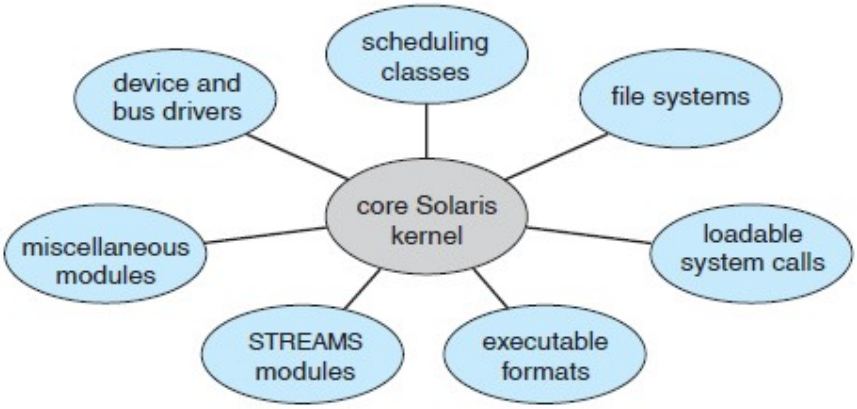


Figure: Solaris loadable modules.

8. Operating system debugging

8.1. Operating system debugging:

- Debugging is the process of finding and fixing errors, or bugs, in a system.
- Debugging seeks to find and fix errors in both hardware and software.
- Performance problems are considered bugs, so debugging can also include performance tuning, which seeks to improve performance by removing bottlenecks in the processing taking place within a system.
- In this section, we explore debugging kernel and process errors and performance problems.

1. **Failure Analysis:**
   - If a process fails, operating systems write the error information to a **log file** to alert system operators or users that the problem occurred.
   - The operating system can also take a **core dump**—the memory state of the process.
   - A kernel failure is called a **crash**.

- When kernel fails, error information is saved to a log file, and the memory state is saved to a **crash dump**.
- Operating system debugging uses a variety of tools and technologies.
    - A common technique is to save the kernel's memory state to a section of disk.
- If the kernel detects an unrecoverable error, it writes the entire contents of memory, or at least the kernel-owned parts of the system memory, to the disk area.
- When the system reboots, a process runs to gather the data from that area and write it to a crash dump file within a file system for analysis.

2. Performance Tuning:

- To identify bottlenecks, we must be able to monitor system performance.
- The operating system performs the tracing and events are written in to a log file.
- Later, an analysis program can process the log file to determine system performance and to identify bottlenecks and inefficiencies.
- The cycle of initiating tracing when system problems occur and then analyzing traces is broken by the new generation of kernel-enabled performance analysis tools.
- The Solaris 10 **DTrace** dynamic tracing facility is a leading example of such a tool.

3. DTrace:

- DTrace adds probes dynamically to the running system in user processes and kernels.
- These probes can be queried via the D programming language to determine an surpricing amount about the kernel, the system state, and process activities.
- DTrace tool provides a dynamic, safe, low-impact debugging environment.

9. System Boot.

9.1. System Boot:

- **Booting:**
    - The procedure of starting a computer by loading the kernel is known as **booting** the system.
- **Bootstrap Loader:**
    - A small piece of code known as the **bootstrap program or bootstrap loader** locates the kernel, loads it into main memory, and starts its execution.
    - Bootstrap loader is stored in ROM (Read-Only Memory).
    - ROM is convenient because it needs no initialization and cannot be infected by a computer virus.
- **Booting Process in computers:**
    - When a CPU receives a reset event—for instance, when it is powered up or rebooted—the instruction register is loaded with a bootstrap loader.

- The bootstrap program can perform a variety of tasks:
  - It runs diagnostics to determine the state of the machine. If the diagnostics pass, the program can continue with the booting steps.
  - It can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory.
  - Loads the operating system from **boot block** into the RAM.
  - Boot block is a fixed location in hard disk, where the operating located in hard disk.
- Once the operating system is loaded into the RAM, it will take over the system to perform the tasks.

- **Booting Process in cellular phones, PDAs, and game consoles:**
  - Store the entire operating system in ROM.
  - Storing the operating system in ROM is suitable for small operating systems, simple supporting hardware, and rugged operation.
  - A problem with this approach is that changing the bootstrap code requires changing the ROM hardware chips.
  - Some systems resolve this problem by using **erasable programmable read-only memory (EPROM).**

2 Marks Questions

1. Define operating system. Give example.
2. What are the goals of operating system?
3. Mention operating system functions.
4. What is an interrupt? Define different types of interrupt with example.
5. Define mode bit.
6. Describe about different types of servers.
7. Define open source software.
8. Write differences between open source and closed source software.
9. Write differences between open source and free source software.
10. Write differences between command interface and GUI
11. Define system call.
12. Define debugging.
13. Define system boot.

Important - 5 Marks Questions

1. Explain different functions of operating systems.
2. Define operating system. Elaborate operating system operations with example.
3. List and explain different computing environments.
4. Discuss various services provided by an operating system.
5. Illustrate user and operating system interface in detail.
6. What is system call? Discuss major system calls of operating system.
7. Enumerate operating system design and implementation.
8. Describe operating system structure in detail.
9. Explain the process of doing operating system debugging.

Unit 2 :: Process Concept, Multithreaded Programming, Process Scheduling, Inter-process Communication

Chapter-1 Processes	Chapter-2 Multithreaded Programming
1. Process Concept 2. Process scheduling 3. Operations on processes 4. Inter-process communication 5. Communication in client server systems	1. Multithreading models 2. Thread libraries 3. Threading issues 4. Examples
Chapter-3 **Process Scheduling**	**Chapter-4** **Inter-process Communication**
1. Basic concepts 2. Scheduling criteria 3. Scheduling algorithms 4. Multiple processor scheduling 5. Thread scheduling 6. Examples	1. Race conditions 2. Critical Regions 3. Mutual exclusion with busy waiting 4. Sleep and wakeup 5. Semaphores 6. Mutexes 7. Monitors 8. Message passing 9. Barriers 10. Classical IPC Problems 10.1. Dining philosophers problem 10.2. Readers and writers problem

Unit – 2 :: Chapter – 1
Processes

1. Process Concept

1.1. The Process:

- **Definition:** A process is a program in execution.

Process in Memory:

A process includes

- **Text section,** which contains program code.
- **Process stack**, which contains temporary data (such as function parameters, return addresses, and local variables).
- **Data section**, which contains global variables.
- **Heap**, which is memory that is dynamically allocated during process run time.

The structure of a process in memory is shown in below figure.

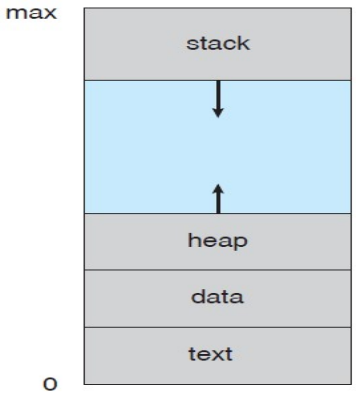


Figure: Process in memory.

Program VS Process:

- A program is a **passive entity**, such as a file containing a list of instructions stored on disk (also called an executable file).
- A process is an **active entity**, with a program counter specifying the next instruction to execute and a set of associated resources.

S.No.	Program	Process
1.	Program contains a set of instructions designed to complete a specific task.	Process is an instance of an executing program.
2.	Program is a passive entity as it resides in the secondary memory.	Process is an active entity as it is created during execution and loaded into the main memory.

S.No.	Program	Process
3.	Program exists at a single place and continues to exist until it is deleted.	Process exists for a limited span of time as it gets terminated after the completion of task.
4.	Program is a static entity.	Process is a dynamic entity.
5.	Program does not have any resource requirement; it only requires memory space for storing the instructions.	Process has a high resource requirement; it needs resources like CPU, memory address, I/O during its lifetime.
6.	Program does not have any control block.	Process has its own control block called Process Control Block.
7.	Program has two logical components: code and data.	In addition to program data, a process also requires additional information required for the management and execution.
8.	Program does not change itself.	Many processes may execute a single program. There program code may be the same but program data may be different. These are never same.

1.2. Process State:

- The state of a process **defines current activity of that process**.
- Each process may be in one of the following states:
  - **New:** The process is being created.
  - **Running:** Instructions are being executed.
  - **Waiting:** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
  - **Ready:** The process is waiting to be assigned to a processor.
  - **Terminated:** The process has finished execution.
- The state diagram of the process (life cycle of process) is illustrated below figure.

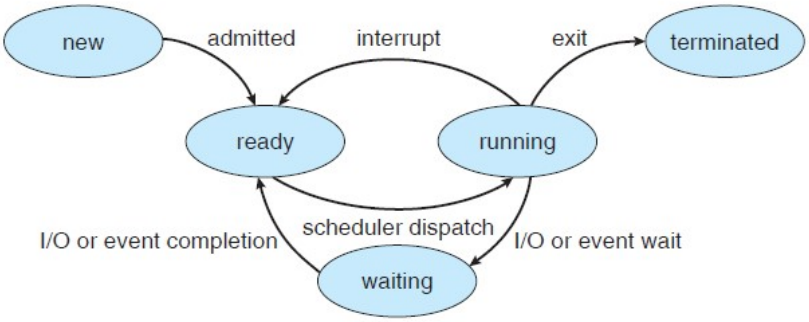


Figure: Diagram of process state.

1.3. Process Control Block

- Each process is represented in the operating system by a process control block (PCB).
- PCB also called a **task control block**.
- A PCB is shown in below figure:

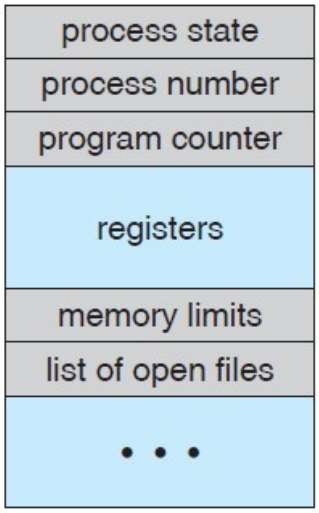


Figure: Process control block (PCB).

- Many pieces of information associated with a specific process, including these:
  1. **Process state:** The state may be new, ready, running, waiting, halted, and so on.
  2. **Program counter:** The counter indicates the address of the next instruction to be executed for this process.
  3. **CPU registers:** They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.

     Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward, see the below figure:

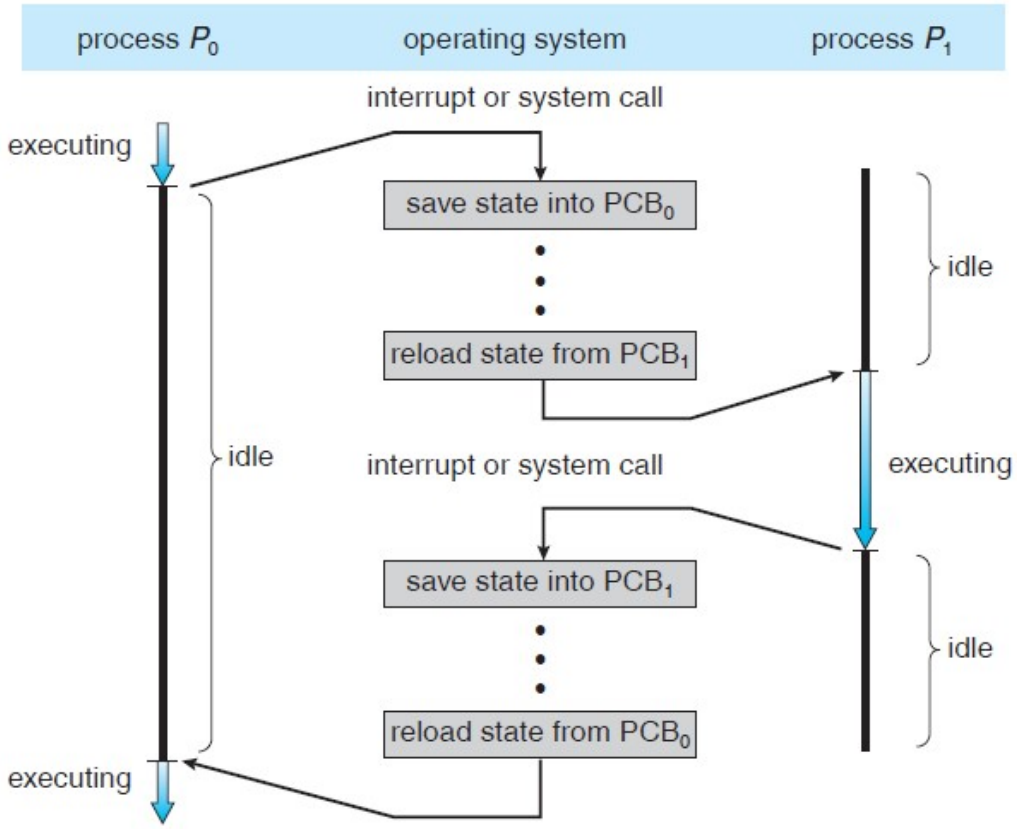


Figure: Diagram showing CPU switch from process to process.

4. **CPU-scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
5. **Memory-management information:** This information may include such information as the value of the base and limit registers, the page tables, or the segment tables.
6. **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
7. **I/O status information:** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

2. Process scheduling

2.1. Process Scheduling:

- The **goal of multiprogramming** is to **maximize CPU usage**, with certain processes running at all times.
- The **goal of time sharing** is to **change the CPU frequently between processes** so that users can interact with each program while it is running.
- To achieve these objectives, the **process scheduler** selects the process available for execution on the CPU.
- For a single-processor system, there will be one processor and one process will execute at a time, the rest will have to wait until the CPU is free and rescheduled.

2.1.1. Scheduling Queues

- As processes enter the system, they are inserting into a job queue.
- **Job queue** consists of all processes in the system.
- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**.
- A ready-queue is implemented using linked list, contains header points to the first and final PCBs in the list.
- Each PCB includes a pointer field that points to the next PCB in the ready queue.
- When a process is allocated the CPU, it executes for a while and waits for the occurrence of a particular event, such as the completion of an I/O request.
- The list of processes waiting for a particular I/O device is called a **device queue**. Each device has its own device queue. See the below figure:

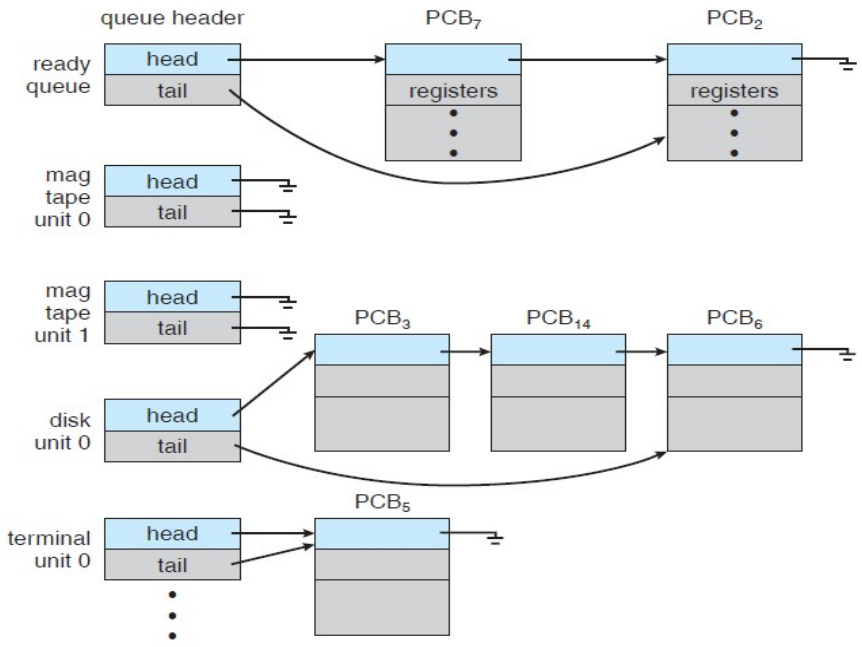


Figure: The ready queue and various I/O device queues.

- The representation of process scheduling is a **queueing diagram**, illustrated in below figure.

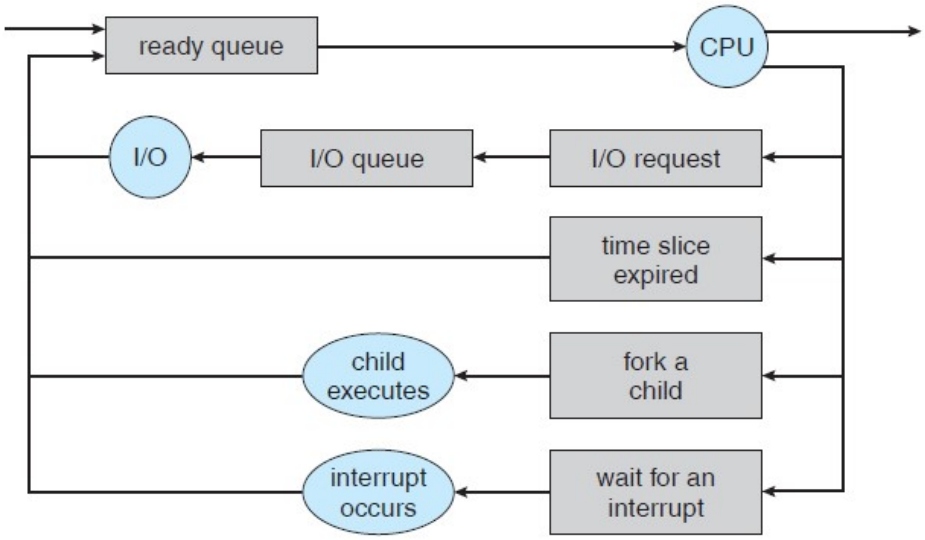


Figure: Queueing-diagram representation of process scheduling.

- In the above figure:
  - Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues.
  - The circles represent the resources that serve the queues
  - And the arrows indicate the flow of processes in the system.
- **Process Execution:**
  - A new process is initially put in the ready queue.
  - It waits there until it is selected for execution.
  - Once the process is allocated the CPU and is executing, one of several events could occur:
    1. The process could issue an I/O request and then be placed in an I/O queue.
    2. The process could create a new sub-process and wait for the sub-process's termination.
    3. The process could be removed forcibly from the CPU as a result of an interrupt and be put back in the ready queue.
  - In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue.
  - A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources de-allocated.

2.1.2. Schedulers

A process migrates among the various scheduling queues throughout its lifetime. The operating system must select processes from these queues for scheduling in some fashion. The selection process is carried out by the appropriate **scheduler.**

In the batch system, immediately executable processes are spooled to the mass-storage device (usually the disk), where they stored for further execution. The **long-term scheduler**, or **job scheduler**, selects processes from this pool and loads them into memory for execution. The **short-term scheduler**, or **CPU scheduler**, selects processes from memory that are ready to execute and allocates the CPU to one of them.

Short-term scheduler is invoked very frequently (milliseconds), must be fast. Long-term scheduler is invoked very infrequently (seconds, minutes), may be slow. The **long-term scheduler controls the degree of multiprogramming.**

Most processes can be described as either **I/O bound or CPU bound**. I/O-bound process spends more time doing I/O than it spends doing computations. A CPU-bound process generates I/O requests infrequently, using more of its time doing computations. It is important that the long-term scheduler select a good process mix of I/O-bound and CPU-bound processes.

If **all processes are I/O bound**, the **ready queue** will almost always be **empty**, and the short-term scheduler will have little to do. If **all processes are CPU bound**, the **I/O waiting queue** will almost always be **empty**, devices will go unused, and again the system will be unbalanced. The **best performing system is a combination of CPU-bound and I / O-bound processes.**

In some time sharing systems like MS Windows and UNIX have no long-term scheduler. All the processors are available to short-term scheduler. The stability of these systems depends either on a physical limitation (such as the number of available terminals) or on the self-adjusting nature of human users.

In some other timesharing systems, may added **medium-term scheduler.** The diagrammatic representation of medium-term scheduler is shown in below figure.

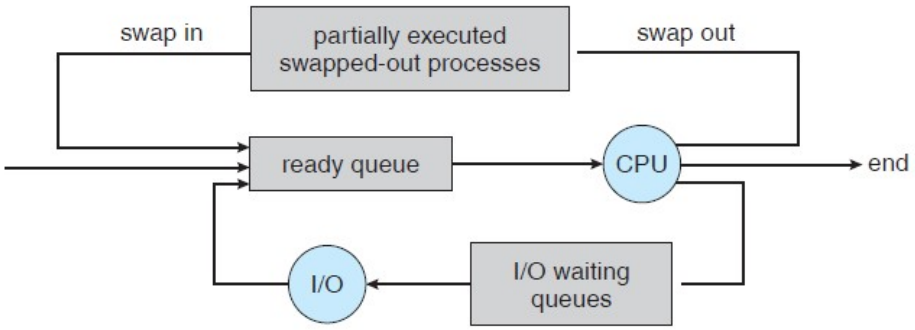


Figure: Addition of medium-term scheduling to the queueing diagram.

The main idea behind the medium-term scheduler is that sometimes it is beneficial to remove processes from memory (and from active contention for the CPU) and thereby reduce the level of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**. The process is swapped out, and is later swapped in, by the medium-term scheduler.

2.1.3. Context Switch

When CPU switching to another process, the system must save the state of the current process and restore the saved state for the new process. This task is known as a **Context switch**.

When a context switch occurs, the kernel saves the state of the old process in its PCB and loads the saved context of the new process scheduled to run.

Context-switch time is pure overhead, because the system does no useful work while switching.

Context-switch times are highly dependent on hardware support (like, memory speed, the number of registers, and special instructions).

3. Operations on processes

3.1. Operations on Processes:

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination.

3.1.1. Process Creation:

- Process may create several new processes, via a create-process system call, during the execution.
- The creating process is called a **parent process**, and the new processes are called the **children** of that process.
- Each new process may create other processes, forming a tree of processes.
- Generally, process identified and managed via a **process identifier (pid).**
- **pid** is unique integer value.
- **Resource Sharing:**
  - Parent and children share all resources.
  - Children share subset of parent's resources.
- **Execution:**
  - Parent and children execute concurrently.
  - Parent waits until children terminate.
- **Address space:**
  - The child process is a duplicate of the parent process.
  - The child process has a new program loaded into it.
- **Example:**
  - **Unix Operating System:**
    1. Each process is identified by its process identifier (pid).
    2. A new process is created by the **fork() system call**.

3. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process.
4. Both processes (the parent and the child) continue execution at the instruction after the fork(), with one difference: the return value for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.
5. The **exec()** system call is used after a fork() system call by one of the two processes to replace the process's memory space with a new program.
6. The parent waits for the child process to complete with the **wait() system call**.
7. When the child process completes the parent process resumes from the call to wait(), where it completes using the exit() system call.
8. This is also illustrated in below figure.

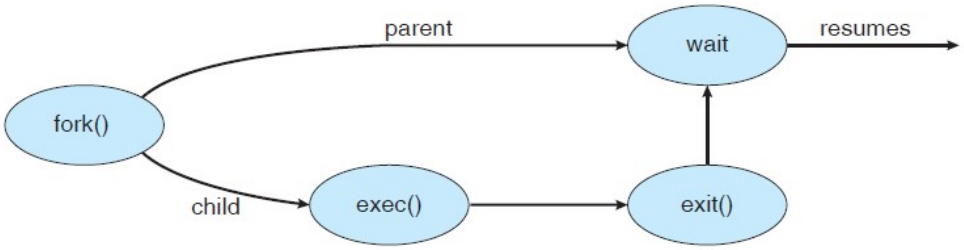


Figure: Process creation using `fork()` system call.

3.1.2. Process Termination:

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the **exit() system** call.
- At that point, the process may return a status value (an integer) to its parent process.
- Process' resources are de-allocated by operating system.
- Parent may terminate execution of children processes using **abort system call**.
- A parent may terminate the execution of one of its children for a variety of reasons, such as these:
  - The child has exceeded its usage of some of the resources that it has been allocated.
  - 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 if its parent terminates. This phenomenon, is called **cascading termination.**

4. Inter-process communication

4.1. Inter-process communication (IPC):

- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
- **Independent process:**
  - Any process that does not share data with any other process is called independent process.
  - An independent process cannot affect or be affected by the other processes executing in the system.
- **Cooperating process:**
  - Any process that shares data with other processes is called cooperating process.
  - A cooperating process can affect or be affected by the other processes executing in the system.
- There are several **reasons for providing an environment** that allows **process cooperation**:
  - **Information sharing:** Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
  - **Computation speedup**: If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).
  - **Modularity:** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
  - **Convenience:** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.
- **Inter-process communication (IPC):**
  - Cooperating processes require an inter-process communication (IPC) that will allow the processes to exchange data and information.
- There are **two fundamental models of inter-process communication**:
  1. Shared memory
  2. Message passing

1. Shared Memory:

- In the shared-memory model, a region of shared memory is established by cooperating processes.
- Processes can then exchange information by reading and writing data to the shared region.
- The shared memory is illustrated in below figure:

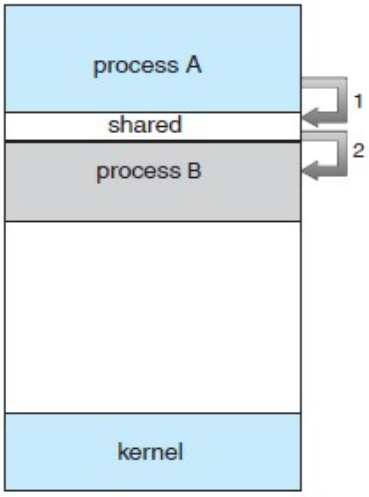


Figure: IPC - Shared Memory Model

- **Producer-Consumer Problem:**
  - Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process.
  - **For example,**
    1. A compiler may produce assembly code, which is consumed by an assembler.
    2. The assembler may produce object modules, which are consumed by the loader.
    3. In client–server paradigm, A Web server produces HTML files, images consumed by client.
- In IPC, shared memory is the solution for producer-consumer problem.
- A producer can produce one item while the consumer is consuming another item.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.
- Two types of buffers can be used for producer-consumer problem:
  1. **Unbounded buffer:**
     - Unbounded buffer has **no limit** on the size of the buffer.
     - The consumer may have to wait for new items, but the producer can always produce new items.
  2. **Bounded buffer:**
     - Bounded buffer has **limit** on the size of the buffer, which is fixed buffer size.
     - In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

- **Note:**

One issue is that, when both the producer process and the consumer process attempt to access the shared buffer concurrently will be discussed in process synchronization.

2. Message Passing:

- In the message passing model, communication takes place by means of messages exchanged between the cooperating processes.
- The message passing is illustrated in below figure:

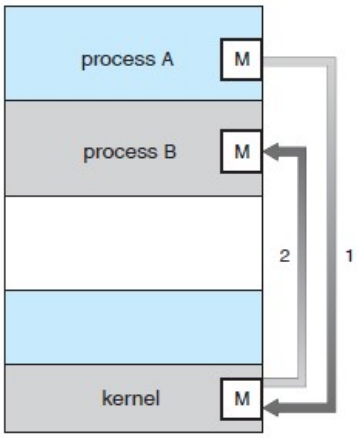


Figure: IPC : Message - Passing Systems

- This model is useful in distributed environment, where the communicating processes may reside on different computers connected by a network.
- For example, a **chat** program used on the World Wide Web could be designed so that chat participants communicate with one another by exchanging messages.
- A message-passing facility provides at least two operations:
  1. Send message
  2. Receive message
- Messages sent by a process can be of either **fixed or variable size**.
- In fixed-sized messages, the system-level implementation is straightforward, but the programming task becomes complex.
- In Variable-sized messages, the system-level implementation is more complex, but the programming task becomes simpler.
- A **communication link** must exist between two processes to send and receive messages from each other.
- A communication link is implemented using following methods:
  1. Naming
  2. Synchronization
  3. Buffering

1. **Naming:**
   - Processes must name each other explicitly to communicate.
   - **Direct Communication:**
     - **Symmetric scheme:**
       - In this, each process that wants to communicate must explicit name of the recipient or sender of the communication.
       - In Symmetric scheme, the send() and receive() primitives are defined as:
         - send(P, message)—Send a message to process P.
         - receive(Q, message)—Receive a message from process Q.
     - **Asymmetric scheme:**
       - In this, only the sender names the recipient; the recipient is not required to name the sender.
       - In Asymmetric scheme, the send() and receive() primitives are defined as follows:
         - send(P, message)—Send a message to process P.
         - receive(id, message)—Receive a message from any process; the variable id is set to the name of the process with which communication has taken place.
     - A communication link in this scheme has the following properties:
       - A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
       - A link is associated with exactly two processes.
       - Between each pair of processes, there exists exactly one link.
     - The drawback of both symmetric and asymmetric schemes is that: Changing the identifier of a process may examine all other process definitions.

   - **Indirect Communication:**
     - In indirect communication, the messages are sent to and received from mailboxes, or ports.
     - The send() and receive() primitives are defined as:
       - send(A, message)—Send a message to mailbox A.
       - receive(A, message)—Receive a message from mailbox A.
     - A communication link in this scheme has the following properties:
       - Link is established between a pair of processes only if both members of the pair have a shared mailbox.

- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.

2. **Synchronization:**
   - Message passing may be blocking or non-blocking — also known as synchronous and asynchronous.
   - **Blocking send:** The sending process is blocked until the message is received by the receiving process or by the mailbox.
   - **Non-blocking send:** The sending process sends the message and resumes operation.
   - **Blocking receive:** The receiver blocks until a message is available.
   - **Non-blocking receive:** The receiver retrieves either a valid message or a null.
3. **Buffering:**
   - Whether communication is direct or in-direct, messages exchanged by communicating processes reside in a temporary queue.
   - Basically, such queues can be implemented in three ways:
     1. **Zero capacity:** The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
     2. **Bounded capacity**. The queue has finite length **n**; thus, at most **n** messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue, and the sender can continue execution without must block until space is available in the queue.
     3. **Unbounded capacity.** The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.
   - The zero-capacity case is sometimes referred to as a message system with **no buffering**; the other cases are referred to as systems with **automatic buffering**.

5. Communication in client server systems

5.1. Communication in client server systems:

- Client – Server communication is an example of IPC: Shared memory model.
- In this section, we discuss two other strategies for communication in client–server systems:
    1. Sockets
    2. Remote Procedure Calls

1. Sockets:

- A socket is defined as an endpoint for communication.
- A pair of processes communicating over a network employ a pair of sockets — one for each process.
- A socket is identified by an IP address concatenated with a port number.
- The server waits for incoming client requests by listening to a specified port.
- Once a request is received, the server accepts a connection from the client socket to complete the connection.
- Servers implementing specific services (such as telnet, FTP, and HTTP) listen to well-known ports (a telnet server listens to port 23; an FTP server listens to port 21; and a Web, or HTTP, server listens to port 80).
- All ports below 1024 are considered well known; we can use them to implement standard services.
- When a client process initiates a request for a connection, it is assigned a port by its host computer. This port is some arbitrary number greater than 1024.
- For example, if a client on host X with IP address 146.86.5.20 wishes to establish a connection with a Web server (which is listening on port 80) at address 161.25.19.8, host X may be assigned port 1625.
- The connection will consist of a pair of sockets: (146.86.5.20:1625) on host X and (161.25.19.8:80) on the Web server. This situation is illustrated in below figure.
- The packets travelling between the hosts are delivered to the appropriate process based on the destination port number.

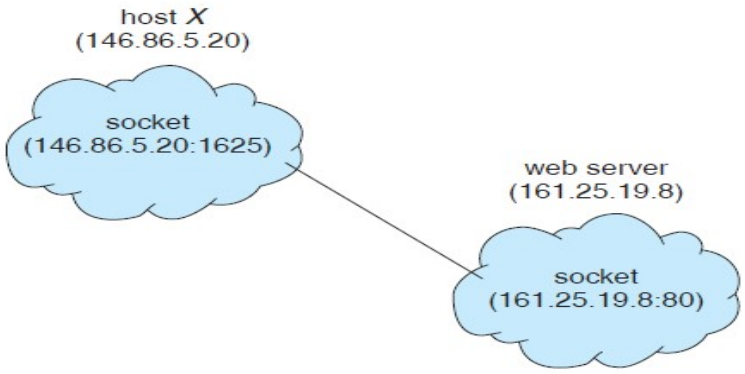


Figure: Communication using sockets.

- All connections must be unique. Therefore, if another process also on host X wished to establish another connection with the same Web server, it would be assigned a port number greater than 1024 and not equal to 1625.

2. Remote Procedure Calls (RPC):

- Remote procedure calls (RPC) are abstracts procedure calls between processes on networked systems.
- RPCs allow a client to invoke a procedure on a remote host because it invokes a procedure locally.
- The RPC system hides the detail that allows communication to take place by providing a **stub** on the client side.
- When the client invokes a remote procedure, the RPC system calls the appropriate stub. This stub locates the port on the server and marshals the parameters.
- Parameter marshalling involves packaging the parameters into a form that can be transmitted over a network.
- The stub then transmits a message to the server using message passing.
- A similar stub on the server side receives this message and invokes the procedure on the server. If necessary, return values are passed back to the client using the same technique.
- To identify the port number of the server the client uses matchmaker daemon on a fixed RPC port.
- The RPC is illustrated in below figure:

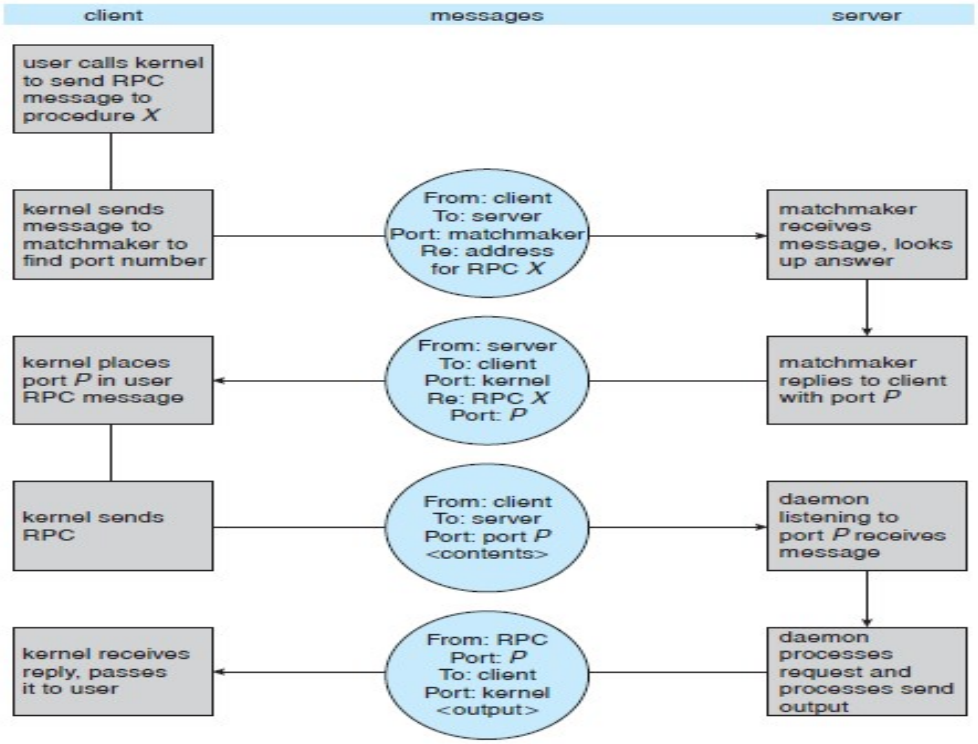


Figure : Execution of a remote procedure call (RPC).

Unit – 2 :: Chapter – 2
Multithreaded Programming

Overview:

Thread - Definition:

- A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack.

Single Thread and Multithreaded Process:

- A traditional (or 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.
- Below figure illustrates the difference between a traditional single-threaded process and a multithreaded process.

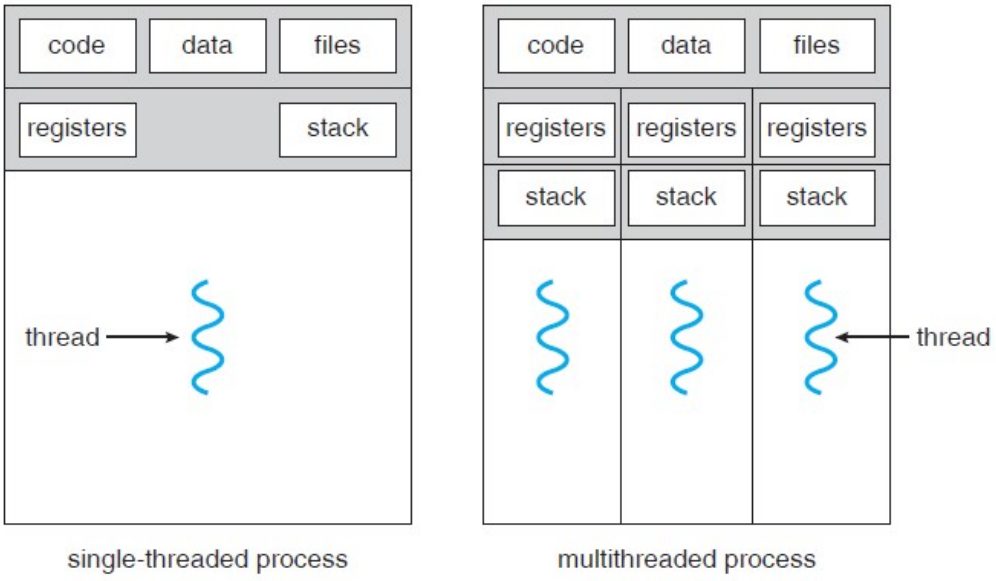


Figure:Single-threaded and multithreaded processes.

Multithreaded Server Architecture:

- Let Consider, a Web server accepts client requests for Web pages, images, sound, and so forth.
- A busy Web server may have several (perhaps thousands of) clients concurrently accessing it.
- If the Web server ran as a traditional single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced.
- If the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. In fact, Process creation is time consuming and resource intensive, however. The new process will perform the same tasks as the existing process, this is an overhead?
- It is more efficient to use one process that contains multiple threads.
- If the Web-server process is multithreaded, the server will create a separate thread that listens for client requests. When a request is made, rather than creating another process, the server will create a

new thread to service the request and resume listening for additional requests. This is illustrated in below figure.

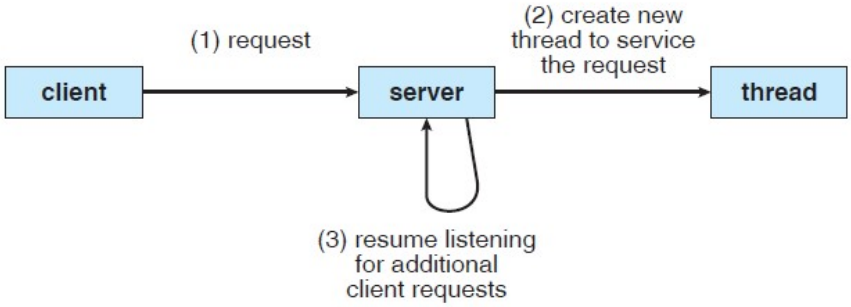


Figure: Multithreaded server architecture.

Benefits of Multithreading:

1. Responsiveness
2. Resource sharing
3. Economy
4. Scalability

Single core VS Multi-core system:

Figure: Concurrent execution on a single-core system.

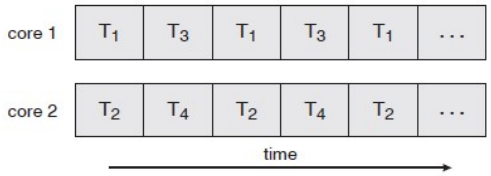


Figure: Parallel execution on a multicore system.

Process VS Thread

S.NO	Process	Thread
1.	Process means any program is in execution.	Thread means a segment of a process.
2.	The process takes more time to terminate.	The thread takes less time to terminate.
3.	It takes more time for creation.	It takes less time for creation.

S.NO	Process	Thread
4.	It also takes more time for context switching.	It takes less time for context switching.
5.	The process is less efficient in terms of communication.	Thread is more efficient in terms of communication.
6.	Multiprogramming holds the concepts of multi-process.	We don't need multi programs in action for multiple threads because a single process consists of multiple threads.
7.	The process is isolated.	Threads share memory.
8.	The process is called the heavyweight process.	A Thread is lightweight as each thread in a process shares code, data, and resources.
9.	Process switching uses an interface in an operating system.	Thread switching does not require calling an operating system and causes an interrupt to the kernel.
10.	If one process is blocked then it will not affect the execution of other processes	If a user-level thread is blocked, then all other user-level threads are blocked.
11.	The process has its own Process Control Block, Stack, and Address Space.	Thread has Parents' PCB, its own Thread Control Block, and Stack and common Address space.
12.	Changes to the parent process do not affect child processes.	Since all threads of the same process share address space and other resources so any changes to the main thread may affect the behavior of the other threads of the process.

1. Multithreading models

1.1. Multithreaded Programming:

- **User Threads:**
  - User threads supported at user level, and are managed without kernel support.
- **Kernel Threads:**
  - Kernel threads are supported and managed directly by the operating system.
  - **Example:**
    - Operating systems—including Windows, Linux, Mac OS X, Solaris, and Tru64 UNIX (formerly Digital UNIX)—support kernel threads.

- There are 3 ways of establishing relationship between user threads and kernel threads.
  1. Many-to-One Model
  2. One-to-One Model
  3. Many-to-Many Model

1. Many-to One Model:

- Many-to-One model is illustrated in below figure:

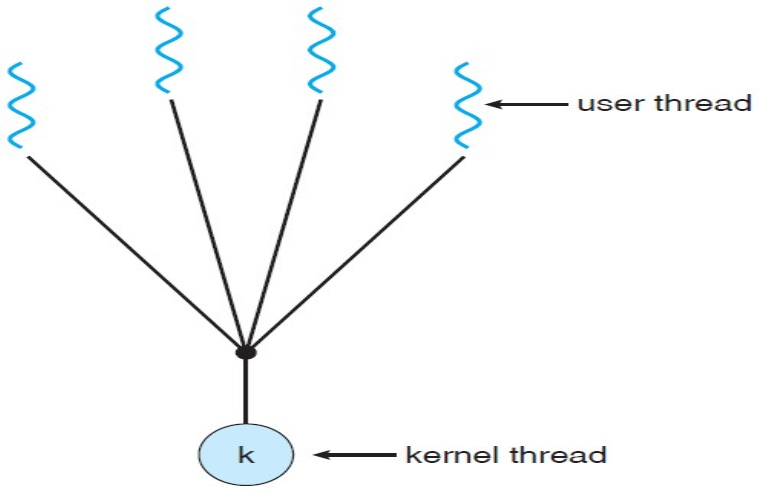


Figure: Many-to-one model.

- 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; but the entire process will block if a thread makes a blocking system call.
- Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.
- **Green threads** — a thread library available for Solaris—uses this model.

2. **One-to-One Model:**
   - One-to-One model is illustrated in below figure:

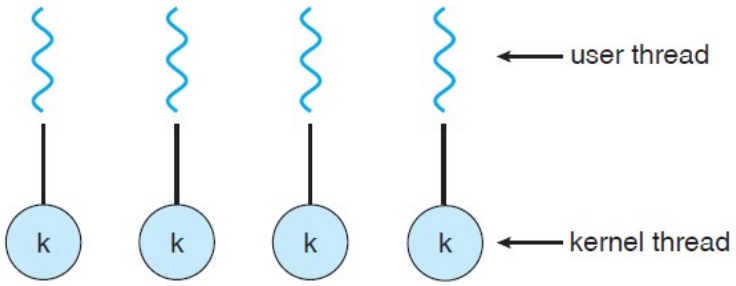


Figure: One-to-one model.

   - The one-to-one model maps each user thread to a kernel thread.
   - It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call; it also allows multiple threads to run in parallel on multiprocessors.
   - The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.
     - Because the overhead of creating kernel threads can burden the performance of an application.
   - Linux, along with the family of Windows operating systems, implement the one-to-one model.

3. **Many-to-Many Model:**
   - Many-to-Many model is illustrated in below figure:

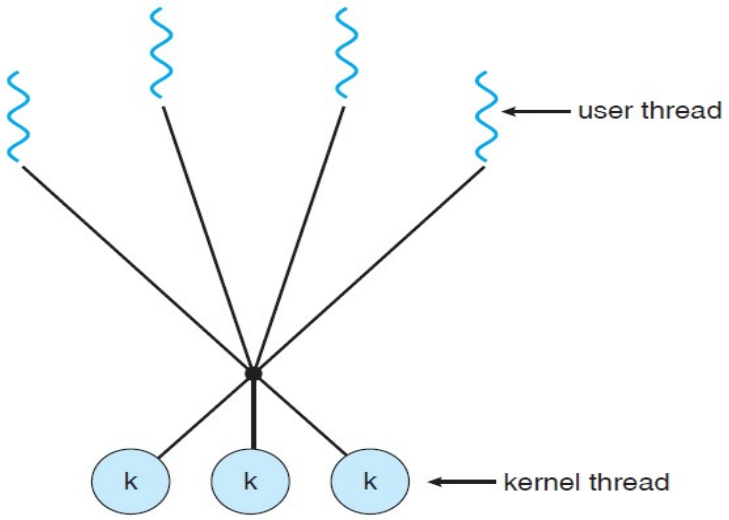


Figure: Many-to-many model.

- 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.
- Whereas the many-to-one model allows the developer to create as many user threads as she wishes, true concurrency is not gained because the kernel can schedule only one thread at a time.
- Solaris prior to version 9 uses this model.
- **Two Level Model:**
  - Similar to Many to Many, except that it allows a user thread to be bound to kernel thread.
  - IRIX, HP-UX, Tru64 UNIX are uses this model.
  - This model is illustrated in below figure:

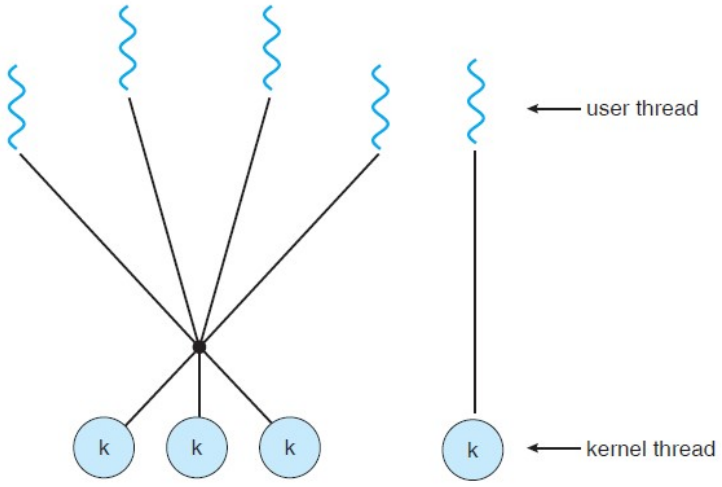


Figure : Two-level model.

2. Thread libraries

2.1. Thread Libraries:

- A **thread library** provides the programmer with an API for creating and managing threads.
- There are **two ways** of implementing a thread library.
- **First approach:**
  - It is to provide a library entirely in user space with no kernel support.
  - All code and data structures for the library exist in user space.
  - Invoking a function in the library results in a local function call in user space and not a system call.
- **Second approach:**
  - It is to implement a kernel-level library supported directly by the operating system.
  - All code and data structures for the library exist in kernel space.
  - Invoking a function in the API for the library results in a system call to the kernel.
- **Three main thread libraries** are in use today:
  1. POSIX Pthreads
  2. Win32 threads
  3. Java threads

1. POSIX Pthreads:

- May be provided either as user-level or kernel-level.
- A POSIX standard (IEEE 1003.1c) defining API for thread creation and synchronization.
- API specifies behaviour of the thread library, implementation is up to development of the library Common in Solaris, Linux, Mac OS X andTru64 UNIX.
- pthread_create( ) function is used to create pthreads.

2. Win32 threads:

- The technique for creating threads using the Win32 thread library is similar to the Pthreads technique in several ways.
- We must include the windows.h header file when using the Win32 API.
- Threads are created in theWin32 API using the CreateThread() function.

3. Java threads:

- Threads are the fundamental model of program execution in a Java program,
- The Java language and its API provide a rich set of features for the creation and management of threads.
- All Java programs comprise at least a single thread of control—even a simple Java program consisting of only a main() method runs as a single thread in the JVM.
- There are two techniques for creating threads in a Java program.
  1. One approach is to create a new class that is derived from the Thread class and to override its run() method.
  2. The Second Approach is —to define a class that implements the Runnable interface.

- Creating a Thread object in java does not specifically create the new thread; rather, it is the start() method that creates the new thread.
- Calling the start() method for the new object does two things:
    1. It allocates memory and initializes a new thread in the JVM.
    2. It calls the run() method, making the thread eligible to be run by the JVM.
- When the summation java program runs, two threads are created by the JVM.
    1. The first is the parent thread, which starts execution in the main() method.
    2. The second thread is created when the start() method on the Thread object is invoked.

3. Threading issues

3.1. Threading Issues:

- The following issues are associated with multithreaded programs.
    1. The fork() and exec() System Calls
    2. Cancellation
    3. Signal Handling
    4. Thread Pools
    5. Thread-Specific Data
    6. Scheduler Activations

1. The fork() and exec() System Calls:

- The fork() system call is used to create a separate, duplicate process.
- The exec() system call is used after a fork() system call by one of the two processes(Parent/Child) to replace the process's memory space with a new program.
- In Multiple Threading the **problem** is that: If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded?
- Here is the **solution** is:
    - If exec() is called immediately after forking, then duplicating all threads is unnecessary.
    - If exec()does not call after forking, the separate process should duplicate all threads.

2. Cancellation:

- Thread cancellation is the task of terminating a thread before it has completed.
- **For example**, when a user presses a button on a Web browser that stops a Web page from loading any further. Often, a Web page is loaded using several threads—each image is loaded

in a separate thread. When a user presses the stop button on the browser, all threads loading the page are cancelled.

- A thread that is to be cancelled is referred as the **target thread**.
- **Cancellation of a target thread** may occur in **two different scenarios**:
  1. **Asynchronous cancellation:** Terminates the target thread immediately.
  2. **Deferred cancellation:** The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself.
- Cancelling a thread asynchronously may not free necessary system resources.
- With deferred cancellation, cancellation occurs only after the target thread has checked a flag to determine whether or not it should be cancelled.

3. <u>Signal Handling:</u>

- A signal is used in UNIX systems to notify a process that a particular event has occurred.
- A signal may be received either synchronously or asynchronously, depending on the source and reason of signalled event.
- **Examples** of **synchronous signals** include illegal memory access and division by 0.
- **Examples** of **asynchronous signals** include terminating a process with specific keystrokes and having a timer expire.
- All signals, whether synchronous or asynchronous, follow the same pattern:
  1. A signal is generated by the occurrence of a particular event.
  2. A generated signal is delivered to a process.
  3. Once delivered, the signal must be handled.
- A signal may be handled by one of two possible handlers:
  1. A default signal handler
  2. A user-defined signal handler
- Every signal has a default signal handler that is run by the kernel when handling that signal.
- The default action can be overridden by a user-defined signal handler.
- Handling signals in single-threaded programs is straightforward: signals are always delivered to a process.
- Handling signals in multithreaded programs is complicated, where a process may have several threads.
- In multithreaded programs, signal haling has the following options:
  1. Deliver the signal to the thread to which the signal applies.
  2. Deliver the signal to every thread in the process.
  3. Deliver the signal to certain threads in the process.
  4. Assign a specific thread to receive all signals for the process.
- **For example**,

- Synchronous signals are delivered to the thread causing the signal and not to other threads in the process.
- Asynchronous signals such as a signal that terminates a process should be sent to all threads.

- In UNIX, the signal kill specifies the process to which a signal is delivered.
- Windows does not explicitly provide support for signals; they can be emulated using asynchronous procedure calls (APCs).

4. Thread Pools:

- Unlimited threads in server could exhaust system resources, such as CPU time or memory.
- One solution to this problem is to use a thread pool.
- Thread pool create a number of threads at process start-up and place them into a pool, where they sit and wait for work.
- When a server receives a request, it awakens a thread from this pool—if one is available—and passes it the request for service.
- Once the thread completes its service, it returns to the pool and awaits more work.
- If the pool contains no available thread, the server waits until one becomes free.
- Thread pools offer these benefits:
  1. Usually slightly faster to service a request with an existing thread than create a new thread
  2. A thread pool limits the number of threads that exist at any one point.

5. Thread-Specific Data:

- Process-related threads share process data.
- In some cases, each thread needs its own copy of certain data. We will call such data **thread-specific data**.
- **For example**, in a transaction-processing system, each transaction served in a separate thread.
- Win32 and Pthreads—provide some form of support for thread-specific data. Java provides support as well.

6. Scheduler Activations:

- Many systems implementing either the many-to-many or the two-level model with data structure for communication between the user and kernel threads.
- This data structure known as a lightweight process, or LWP—is shown in below figure.

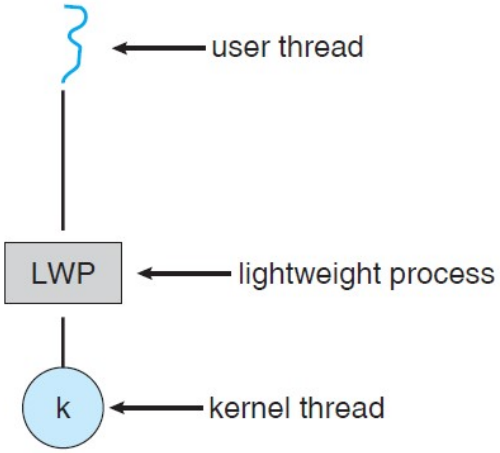


Figure Lightweight process (LWP).

- To the user-thread library, the LWP appears as a virtual processor on which the application can schedule a user thread to run.
- Communication between the user-thread library and the kernel is known as **scheduler activation.**
- It works as follows:
  - The kernel provides an application with a set of virtual processors (LWPs), and the application can schedule user threads onto an available virtual processor.
  - The kernel must inform an application about certain events. This procedure is known as an **upcall**.
  - Upcalls are handled by the thread library with an upcall handler, and upcall handlers must run on a virtual processor.
  - One event that triggers an upcall occurs when an application thread is about to block.
  - In this scenario, the kernel makes an upcall to the application informing it that a thread is about to block and identifying the specific thread.
  - The kernel then allocates a new virtual processor to the application.
  - The application runs an upcall handler on this new virtual processor, which saves the state of the blocking.
  - The upcall handler then schedules another thread that is eligible to run on the new virtual processor.

4. Examples

4.1. Windows Threads

- Windows implements theWin32 API as its primary API.
- A Windows application runs as a separate process, and each process may contain one or more threads.
- Windows uses the one-to-one mapping, where each user-level thread maps to an associated kernel thread.
- Windows also provides support for a fiber library, which provides the functionality of the many-to-many model.
- By using the thread library, any thread belonging to a process can access the address space of the process.
- The general components of a thread include:
    1. A thread ID uniquely identifying the thread
    2. A register set representing the status of the processor
    3. A user stack, employed when the thread is running in user mode, and a kernel stack, employed when the thread is running in kernel mode
    4. A private storage area used by various run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the context of the thread.
- The primary data structures of a thread include:
    1. ETHREAD—executive thread block
    2. KTHREAD—kernel thread block
    3. TEB—thread environment block
- The ETHREAD and the KTHREAD exist entirely in kernel space; this means that only the kernel can access them.
- ETHREAD includes:
    - A pointer to the process to which the thread belongs
    - The address of the routine in which the thread starts control.
    - A pointer to the corresponding KTHREAD.
- The KTHREAD includes:
    - Scheduling and synchronization information for the thread.
    - The kernel stack (used when the thread is running in kernel mode)
    - A pointer to the TEB.
- The TEB is a user-space data structure that is accessed when the thread is running in user mode.
- The TEB includes:
    - Thread identifier
    - User-mode stack

- An array for thread specific data (which Windows terms thread-local storage).

- The structure of a Windows thread is illustrated in below figure.

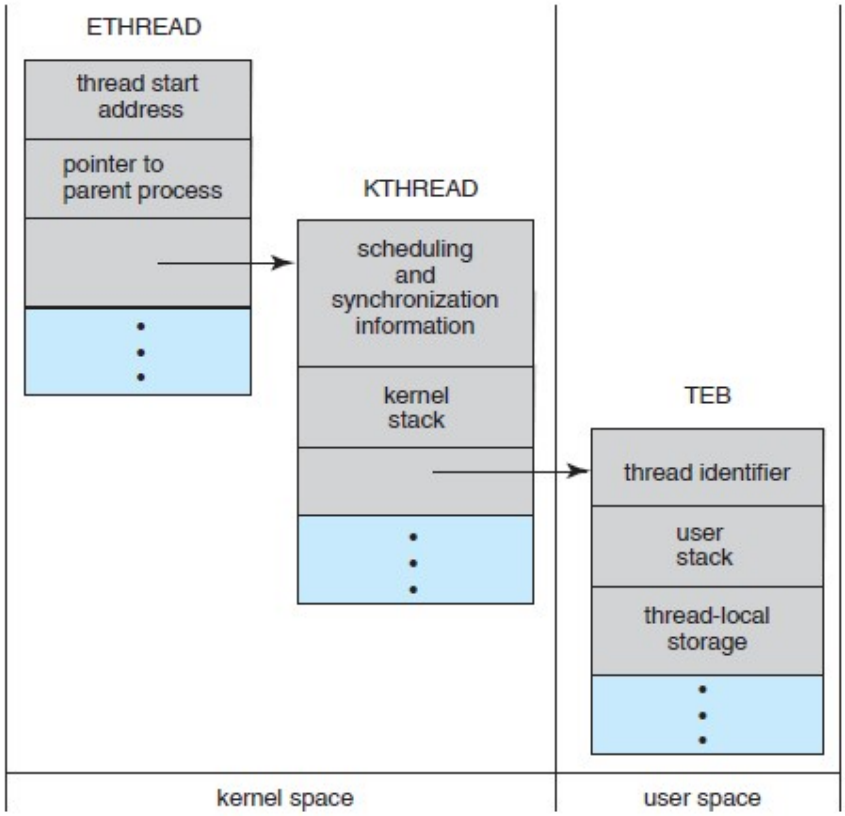


Figure Data structures of a Windows thread.

4.2. Linux Threads

- Linux provides the fork() system call for duplicating a process.
- Linux also provides clone()system call to create threads.
- Linux generally uses the term **task**—rather than process or thread.
- When clone() is invoked, it is passed a set of flags that determine how much sharing is to take place between the parent and child tasks.
- Some of these flags are listed below:

Flag	Meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- In Linux, a unique kernel data structure (specifically, struct task struct) exists for each task in the system.

- This data structure, instead of storing data for the task, contains pointers to other data structures where these data are stored—for example, data structures that represent the list of open files, signal-handling information, and virtual memory.
- Rather than copying all data structures, the new task points to the data structures of the parent task, depending on the set of flags passed to clone().
- Linux kernel includes the NPTL (Native POSIX Thread Library) thread library.
- NPTL provides a POSIX compliant thread model for Linux systems along with several features, such as:
    - Better support for SMP systems, as well as taking advantage of NUMA support.
    - The start-up cost for creating a thread is lower with NPTL than with traditional Linux threads.
- NPTL support hundreds of thousands of threads. Such support becomes more important with the growth of multi-core and other SMP systems.

Unit – 2 :: Chapter – 3
Process Scheduling

1. Basic concepts

Single-Programming Systems (or) Single -Processing Systems (or) Single-Tasking Systems:

- In a single- tasking system, only one process can run at a time; any others must wait until the CPU is free and can be rescheduled.
- A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. This is overcome in multi-programming systems.

Multi-Programming Systems (or) Multi-Processing Systems (or) Multi-Tasking Systems:

- In multi-programming, some process running at all times, in order to maximize CPU utilization.
- In this, several processes are kept in memory at one time. Every time a process has to wait, another process can capture CPU usage.

1.1. CPU–I/O Burst Cycle:

- The below figure illustrates CPU-I/O Burst Cycle.

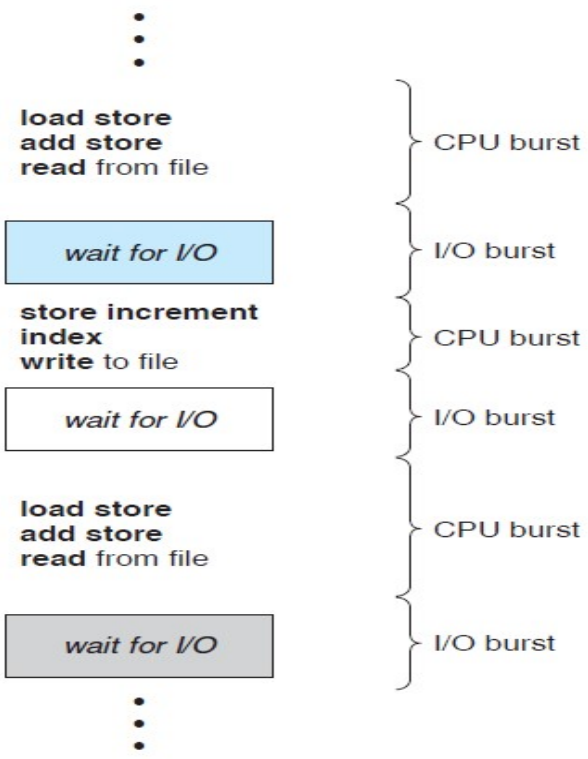


Figure Alternating sequence of CPU and I/O bursts.

- Process execution consists of a cycle of CPU execution and I/O wait.
- Processes alternate between these two states.
- Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on.
- Eventually, the final CPU burst ends with a system request to terminate execution.

1.2. CPU Scheduler:

- Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the **short-term scheduler or CPU scheduler.**

1.3. Preemptive and Non-Preemptive Scheduling:

- CPU scheduling decisions may take place, when a process:
    1. Switches from running to waiting state
    2. Switches from running to ready state
    3. Switches from waiting to ready
    4. Terminates
- When scheduling takes place only under 1 and 4, we say that the scheduling scheme is **non-pre-emptive** or **cooperative**; otherwise, it is **pre-emptive**.

1.4. Dispatcher:

- The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler.
- This function involves the following:
    - Switching context
    - Switching to user mode
    - Jumping to the proper location in the user program to restart the program
- The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

2. Scheduling criteria

- Different CPU-scheduling algorithms have different properties.
- The following characteristics are used for comparison of scheduling algorithms.

1. **CPU utilization:**
    - We want to keep the CPU as busy as possible.
    - In a real system, it should range from 40% (for a lightly loaded system) to 90% (for a heavily used system).
2. **Throughput:**
    - The number of processes that are completed per time unit, called **throughput**.
    - For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.

3. **Turnaround time:**
    - The interval from the time of submission of a process to the time of completion is the **turnaround time.**
    - Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
4. **Waiting time:**
    - **Waiting time** is the total time spent in the ready queue.
5. **Response time:**
    - **Response time** is an amount of time it takes from when a request was submitted until the first response is produced, not output

Note:

- The response time is better than turnaround time, because turnaround time is generally limited by the speed of the output device.
- The best scheduling algorithm is to **maximize CPU utilization** and **throughput** and to **minimize turnaround time, waiting time,** and **response time.**
- **Gantt chart,** which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes.

3. Scheduling algorithms

- CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.
- There are many different CPU-scheduling algorithms. They are:
    - First-Come, First-Served Scheduling
    - Shortest-Job-First Scheduling
    - Priority Scheduling
    - Round-Robin Scheduling
    - Multilevel Queue Scheduling
    - Multilevel Feedback Queue Scheduling

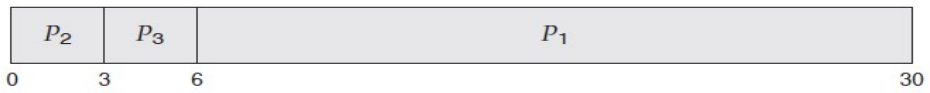
1. **First-Come, First-Served Scheduling (FCFS):**
   - The **simplest** CPU-scheduling algorithm is the **first-come, first-served (FCFS)** scheduling algorithm.
   - With this scheme, the CPU is allocated to the process that request CPU first.
   - **FIFO Queue** is used in implementation of the FCFS.
   - When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.
   - **Example:**
     1. Consider the following set of processes that **arrive at time 0**, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P1	24
P2	3
P3	3

     2. Suppose that **the processes arrive in the order: P1, P2, and P3**. The Gantt Chart for the schedule is:

        - Waiting time for P1 = 0; P2 = 24; P3 = 27.
        - Average waiting time: (0 + 24 + 27)/3 = 17 milliseconds

     3. Suppose that **the processes arrive in the order P2, P3, P1.** The Gantt chart for the schedule is:



        - Waiting time for P1 = 6; P2 = 0; P3 = 3
        - Average waiting time: (6 + 0 + 3)/3 = 3 milliseconds
   - **Advantages of FCFS**
     1. Simple
     2. Easy
     3. First come, First serve
   - **Disadvantages of FCFS**
     1. The scheduling method is non-pre-emptive; the process will run to the completion.
     2. Due to the non-pre-emptive nature of the algorithm, the problem of starvation (Convoy Effect) may occur.
     3. Although it is easy to implement, but it is poor in performance since the average waiting time is higher as compare to other scheduling algorithms.

- **Problems:**

1. Consider the following set of processes that **arrive at time 0**, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P1	24
P2	3
P3	3

Calculate Completion Time, Turn Around Time, Waiting Time and Average Waiting Time using FCFS Algorithm.

2. Consider the following set of processes with different arrival times and with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P1	0	2
P2	1	6
P3	2	4
P4	3	9
P5	6	12

Calculate Completion Time, Turn Around Time, Waiting Time and Average Waiting Time using FCFS Algorithm.

Note:

Turn Around Time = Completion Time - Arrival Time

Waiting Time = Turnaround time - Burst Time

- **Convoy Effect:**

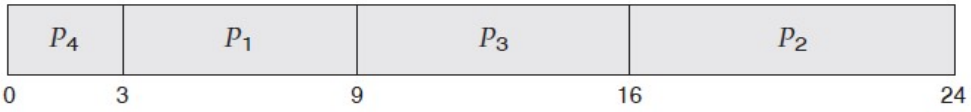
1. If the CPU gets the processes of the higher burst time at the front end of the ready queue then the processes of lower burst time may get blocked which means they may never get the CPU if the job in the execution has a very high burst time. This is called **convoy effect** or **starvation**.

2. Shortest-Job-First Scheduling (SJFS):

- SJF scheduling algorithm, schedules the processes according to their burst time.
- In SJF scheduling, the process with the lowest burst time, among the list of available processes in the ready queue, is going to be scheduled next.
- However, it is very difficult to predict the burst time needed for a process hence this algorithm is very difficult to implement in the system.
- **Example:**
  1. Consider the following set of processes that **arrive at time 0**, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

  2. The Gantt Chart for the SJFS is:



     - Waiting time for P1 = 3; P2 = 16; P3 = 9; P4 = 0
     - Average waiting time: (3 + 16 + 9 + 0)/4 = 7 milliseconds
- **Advantages:**
  1. Maximum throughput
  2. Minimum average waiting and turnaround time
- **Disadvantages:**
  1. May suffer with the problem of starvation
  2. It is not implementable because the exact Burst time for a process can't be known in advance.
- **Problems:**
  1. Consider the following set of processes that **arrive at time 0**, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P1	24
P2	3
P3	3

     Calculate Completion Time, Turn Around Time, Waiting Time and Average Waiting Time using SJFS Algorithm.

2. Consider the following set of processes with different arrival times and with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P1	0	2
P2	1	6
P3	2	4
P4	3	9
P5	6	12

Calculate Completion Time, Turn Around Time, Waiting Time and Average Waiting Time using SJFS Algorithm.

3. Consider the following set of processes with different arrival times and with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P1	1	7
P2	3	3
P3	6	2
P4	7	10
P5	9	8

Calculate Completion Time, Turn Around Time, Waiting Time and Average Waiting Time using SJFS Algorithm

4. Consider the following set of processes with different arrival times and with the length of the CPU burst given in milliseconds:

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

Calculate Completion Time, Turn Around Time, Waiting Time and Average Waiting Time using

1. Pre-emptive SJFS (shortest-remaining-time-first scheduling) Algorithm
2. Non-Pre-emptive SJFS Algorithm

3. Priority Scheduling:

- In Priority scheduling, there is a priority number assigned to each process.
- In some systems, the lower number has higher priority. While, in the others, the higher number has higher priority.
- The Process with the higher priority among the available processes is given the CPU.
- There are two types of priority scheduling algorithm:
  - **Non Pre-emptive** priority scheduling
  - **Pre-emptive** Priority scheduling.

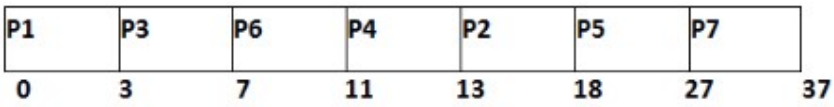
1. Non Pre-emptive priority scheduling:

2. In the Non Preemptive Priority scheduling, The Processes are scheduled according to the priority number assigned to them.
3. Once the process gets scheduled, it will run till the completion.
4. **Example**
   - In the Example, there are 7 processes P1, P2, P3, P4, P5, P6 and P7. Their priorities, Arrival Time and burst time are given in the table.

Process ID	Priority	Arrival Time	Burst Time
1	2	0	3
2	6	2	5
3	3	1	4
4	5	4	2
5	7	6	9
6	4	5	4
7	10	7	10

Solution:

- We can prepare the Gantt chart according to the Non Pre-emptive priority scheduling.
- The Process P1 arrives at time 0 with the burst time of 3 units and the priority number 2. Since No other process has arrived till now hence the OS will schedule it immediately.
- Meanwhile the execution of P1, two more Processes P2 and P3 are arrived. Since the priority of P3 is 3 hence the CPU will execute P3 over P2.
- Meanwhile the execution of P3, All the processes get available in the ready queue. The Process with the lowest priority number will be given the priority. Since P6 has priority number assigned as 4 hence it will be executed just after P3.
- After P6, P4 has the least priority number among the available processes; it will get executed for the whole burst time.
- Since all the jobs are available in the ready queue hence All the Jobs will get executed according to their priorities. If two jobs have similar priority number assigned to them, the one with the least arrival time will be executed.
- **Gantt chart:**



- From the Gantt chart prepared, we can determine the completion time of every process. The turnaround time, waiting time and response time will be determined.
- Turn Around Time = Completion Time - Arrival Time
- Waiting Time = Turn Around Time - Burst Time

Process Id	Priority	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time	Response Time
1	2	0	3	3	3	0	0
2	6	2	5	18	16	11	13
3	3	1	4	7	6	2	3
4	5	4	2	13	9	7	11
5	7	6	9	27	21	12	18
6	4	5	4	11	6	2	7
7	10	7	10	37	30	20	27

- Average Waiting Time = (0+11+2+7+12+2+20)/7 = 54/7 = 7.71 units

2. <u>Pre-emptive Priority scheduling:</u>

5. In Pre-emptive Priority Scheduling, at the time of arrival of a process in the ready queue, its Priority is compared with the priority of the other processes present in the ready queue as well as with the one which is being executed by the CPU at that point of time.
6. The One with the highest priority among all the available processes will be given the CPU next.
7. The difference between pre-emptive priority scheduling and non preemptive priority scheduling is that, in the preemptive priority scheduling, the job which is being executed can be stopped at the arrival of a higher priority job.

8. <u>Example</u>

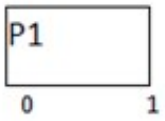
- There are 7 processes P1, P2, P3, P4, P5, P6 and P7 given. Their respective priorities, Arrival Times and Burst times are given in the table below.

Process Id	Priority	Arrival Time	Burst Time
1	2	0	1
2	6	1	7
3	3	2	3
4	5	3	6
5	4	4	5
6	10	5	15
7	9	15	8

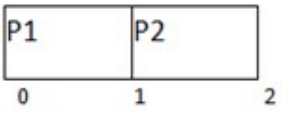
Solution

GANTT chart Preparation

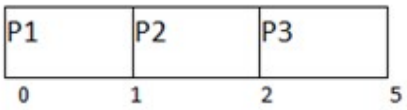
- At time 0, P1 arrives with the burst time of 1 units and priority 2. Since no other process is available hence this will be scheduled till next job arrives or its completion (whichever is lesser).



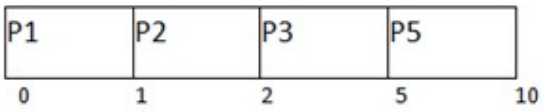
- At time 1, P2 arrives. P1 has completed its execution and no other process is available at this time hence the Operating system has to schedule it regardless of the priority assigned to it.



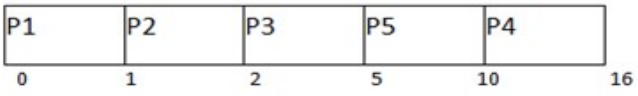
- The Next process P3 arrives at time unit 2, the priority of P3 is higher to P2. Hence the execution of P2 will be stopped and P3 will be scheduled on the CPU.



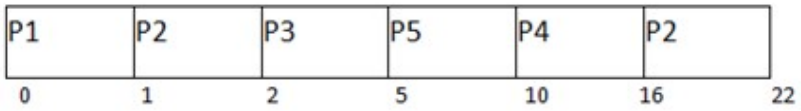
- During the execution of P3, three more processes P4, P5 and P6 becomes available. Since, all these three have the priority lower to the process in execution so PS can't preempt the process. P3 will complete its execution and then P5 will be scheduled with the priority highest among the available processes.



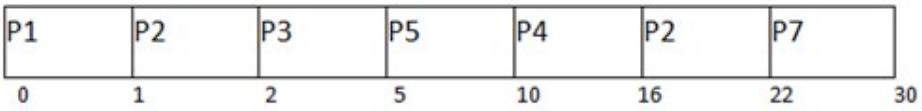
- Meanwhile the execution of P5, all the processes got available in the ready queue. At this point, the algorithm will start behaving as Non Preemptive Priority Scheduling. Hence now, once all the processes get available in the ready queue, the OS just took the process with the highest priority and execute that process till completion. In this case, P4 will be scheduled and will be executed till the completion.



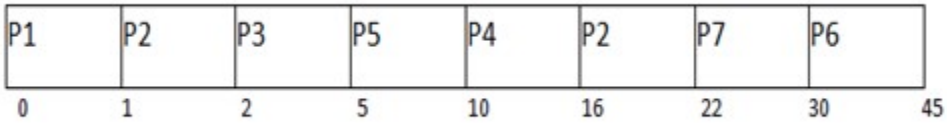
- Since P4 is completed, the other process with the highest priority available in the ready queue is P2. Hence P2 will be scheduled next.



- P2 is given the CPU till the completion. Since its remaining burst time is 6 units hence P7 will be scheduled after this.



- The only remaining process is P6 with the least priority, the Operating System has no choice unless of executing it. This will be executed at the last.



- The Completion Time of each process is determined with the help of GANTT chart. The turnaround time and the waiting time can be calculated by the following formula.
- Turnaround Time = Completion Time - Arrival Time
- Waiting Time = Turn Around Time - Burst Time

Process Id	Priority	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
1	2	0	1	1	1	0
2	6	1	7	22	21	14
3	3	2	3	5	3	0
4	5	3	6	16	13	7
5	4	4	5	10	6	1
6	10	5	15	45	40	25
7	9	15	8	30	20	12

- Average Waiting Time = (0+14+0+7+1+25+12)/7 = 8.43 units

Drawback with priority scheduling algorithms:

- A major problem with priority scheduling algorithms **is indefinite blocking, or starvation**.
  - A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low priority processes waiting indefinitely.

4. Round-Robin (RR) Scheduling:

- The round-robin (RR) scheduling algorithm is designed especially for timesharing systems.
- It is similar to FCFS scheduling, but **pre-emption** is added to enable the system to switch between processes.
- A small unit of time, called a **time quantum** or **time slice**, is defined.
- A time quantum is generally from 10 to 100 milliseconds in length.
- The ready queue is treated as a circular queue.
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.
- **Implementation of RR Scheduling:**
  1. To implement RR scheduling, we keep the ready queue as a FIFO queue of processes.
  2. New processes are added to the tail of the ready queue.
  3. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.
  4. One of two things will then happen.
     - The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue.
     - Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.
- **Advantages:**
  1. It can be actually implementable in the system because it is not depending on the burst time.
  2. It doesn't suffer from the problem of starvation or convoy effect.
  3. All the jobs get a fare allocation of CPU.
- **Disadvantages:**
  1. The higher the time quantum, the higher the response time in the system.
  2. The lower the time quantum, the higher the context switching overhead in the system.
  3. Deciding a perfect time quantum is really a very difficult task in the system.
- **Example:**
  1. In the following example, there are six processes named as P1, P2, P3, P4, P5 and P6. Their arrival time and burst time are given below in the table. The time quantum of the system is 4 units.

Process ID	Arrival Time	Burst Time
1	0	5
2	1	6
3	2	3
4	3	1
5	4	5
6	6	4

Solution:

- According to the algorithm, we have to maintain the ready queue and the Gantt chart. The structure of both the data structures will be changed after every scheduling.

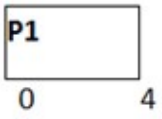
Ready Queue:

- Initially, at time 0, process P1 arrives which will be scheduled for the time slice 4 units. Hence in the ready queue, there will be only one process P1 at starting with CPU burst time 5 units.

P1
5

GANTT chart

- The P1 will be executed for 4 units first.



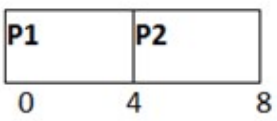
Ready Queue

- Meanwhile the execution of P1, four more processes P2, P3, P4 and P5 arrives in the ready queue. P1 has not completed yet, it needs another 1 unit of time hence it will also be added back to the ready queue.

Process	P2	P3	P4	P5	P1
Remaining Burst Time	6	3	1	5	1

GANTT chart

- After P1, P2 will be executed for 4 units of time which is shown in the Gantt chart.



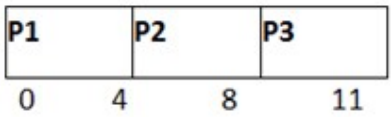
Ready Queue

- During the execution of P2, one more process P6 is arrived in the ready queue. Since P2 has not completed yet hence, P2 will also be added back to the ready queue with the remaining burst time 2 units.

Process	P3	P4	P5	P1	P6	P2
Remaining Burst Time	3	1	5	1	4	2

GANTT chart

- After P1 and P2, P3 will get executed for 3 units of time since its CPU burst time is only 3 seconds.



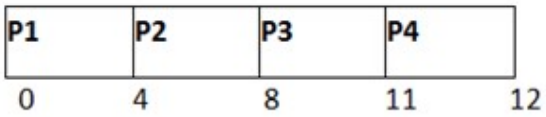
Ready Queue

- Since P3 has been completed, hence it will be terminated and not be added to the ready queue. The next process will be executed is P4.

Process	P4	P5	P1	P6	P2
Remaining Burst Time	1	5	1	4	2

GANTT chart

- After, P1, P2 and P3, P4 will get executed. Its burst time is only 1 unit which is lesser then the time quantum hence it will be completed.



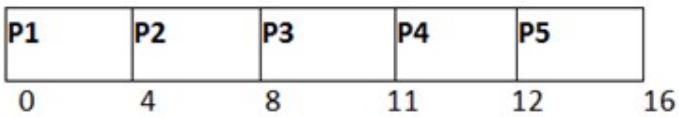
Ready Queue

- The next process in the ready queue is P5 with 5 units of burst time. Since P4 is completed hence it will not be added back to the queue.

Process	P5	P1	P6	P2
Remaining Burst Time	5	1	4	2

GANTT chart

- P5 will be executed for the whole time slice because it requires 5 units of burst time which is higher than the time slice.



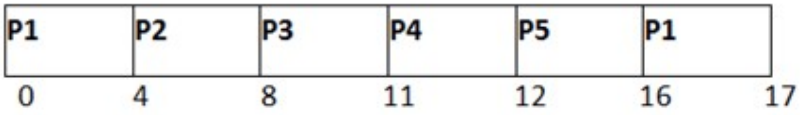
Ready Queue

- P5 has not been completed yet; it will be added back to the queue with the remaining burst time of 1 unit.

Process	P1	P6	P2	P5
Remaining Burst Time	1	4	2	1

GANTT Chart

- The process P1 will be given the next turn to complete its execution. Since it only requires 1 unit of burst time hence it will be completed.



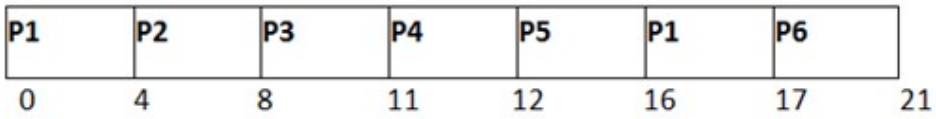
Ready Queue

- P1 is completed and will not be added back to the ready queue. The next process P6 requires only 4 units of burst time and it will be executed next.

Process	P6	P2	P5
Remaining Burst Time	4	2	1

GANTT chart

- P6 will be executed for 4 units of time till completion.



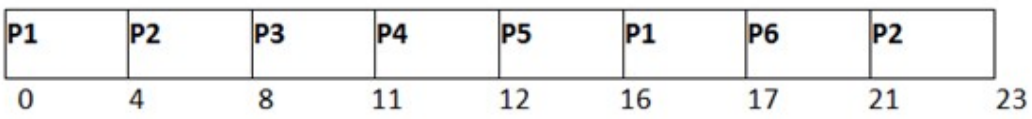
Ready Queue

- Since P6 is completed, hence it will not be added again to the queue. There are only two processes present in the ready queue. The Next process P2 requires only 2 units of time.

Process	P2	P5
Remaining Burst Time	2	1

GANTT Chart

- P2 will get executed again, since it only requires only 2 units of time hence this will be completed.



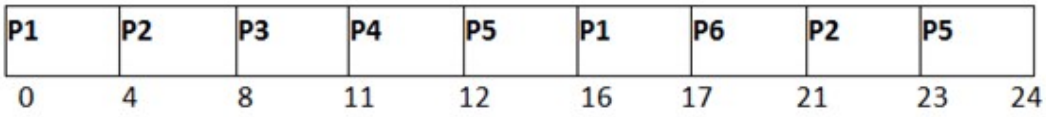
Ready Queue

- Now, the only available process in the queue is P5 which requires 1 unit of burst time. Since the time slice is of 4 units hence it will be completed in the next burst.

Process	P5
Remaining Burst Time	1

GANTT chart

- P5 will get executed till completion.



- The completion time, Turnaround time and waiting time will be calculated as shown in the table below.
- Turn Around Time = Completion Time - Arrival Time
- Waiting Time = Turn Around Time - Burst Time

Process ID	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
1	0	5	17	17	12
2	1	6	23	22	16
3	2	3	11	9	6
4	3	1	12	9	8
5	4	5	24	20	15
6	6	4	21	15	11

- Average Waiting Time = (12+16+6+8+15+11)/6 = 76/6 = 11.33 units

5. Multilevel Queue Scheduling:

- Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups.
- For example, a common distribution is made between foreground (interactive) processes and background (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs.
- In addition, foreground processes may have priority (externally defined) over background processes.
- A **multilevel queue scheduling algorithm** partitions the ready queue into several separate queues as shown in below figure:

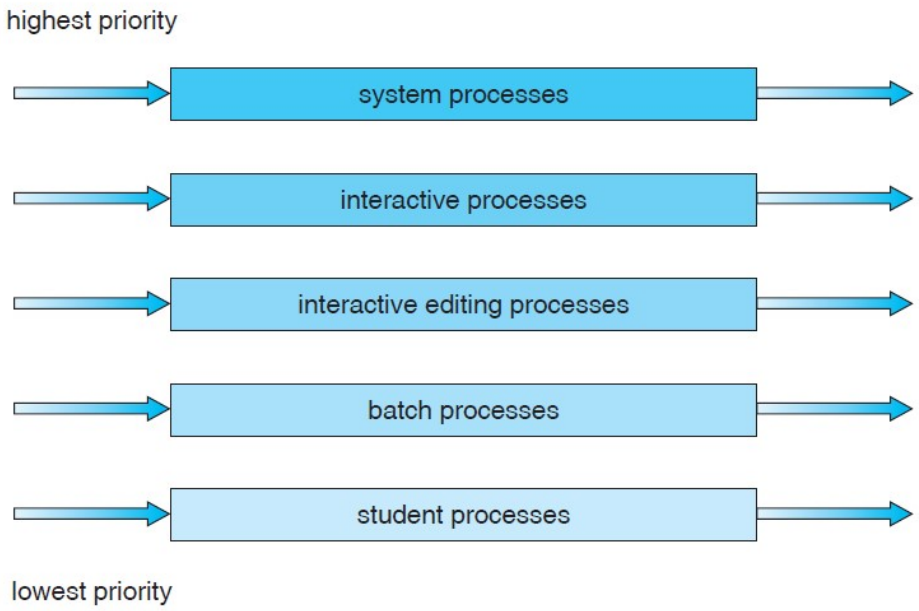


Figure Multilevel queue scheduling.

- The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type.
- Each queue has its own scheduling algorithm.
- For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.
- In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority pre-emptive scheduling.
- For example, the foreground queue may have absolute priority over the background queue.
- Let's look at an example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes

- Each queue has absolute priority over lower-priority queues.
- There is a time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes.
- For instance, in the foreground–background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, whereas the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.
- **In multilevel queue scheduling algorithm processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but it is inflexible.**

6. Multilevel Feedback Queue Scheduling:

- The multilevel feedback queue scheduling algorithm allows a process to move between queues.
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
- In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue.
- This form of aging prevents starvation.
- **For example**, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2. See the below figure:

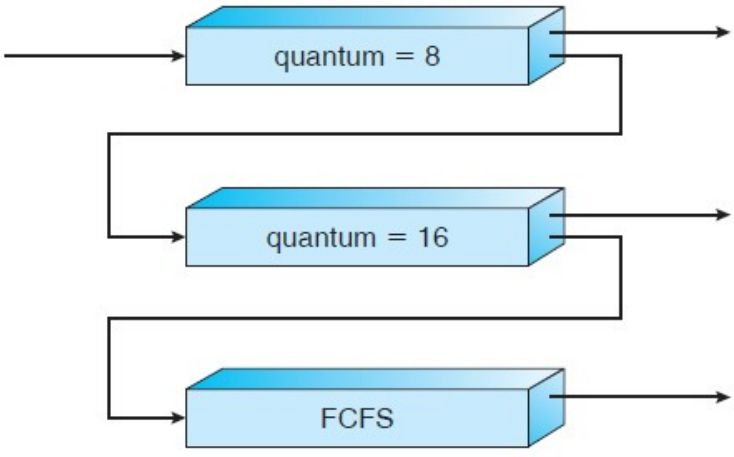


Figure Multilevel feedback queues.

- The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1.
- Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty.
- A process that arrives for queue 1 will pre-empt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.
- In general, a multilevel feedback queue scheduler is defined by the following parameters:
  1. The number of queues
  2. The scheduling algorithm for each queue
  3. The method used to determine when to upgrade a process to a higher priority queue
  4. The method used to determine when to demote a process to a lower priority queue
  5. The method used to determine which queue a process will enter when that process needs service

4. Multiple processor scheduling

- Our discussion so far has focused on issues with CPU scheduling System with a single processor.
- If multiple CPUs are available, load sharing Is possible; However, the schedule problem becomes more complicated.
- Many possibilities have been tried and as we have seen with a single processor CPU scheduling, there is no best solution.
- Here, we discuss many concerns in multiprocessor scheduling.
- We focus on systems where the processors are homogeneous in terms of functionality; We can use any available processor to execute any process in the queue.

1. Approaches to Multiple-Processor Scheduling:

- Two approaches for multiple-processor scheduling:
  1. **Asymmetric Multiprocessing:**
     - In this approach, all scheduling decisions, I/O processing, and other system activities handled by a single processor—the master server. The other processors execute only user code.
     - This is simple because only one processor accesses the system data structures, reducing the need for data sharing.
  2. **Symmetric Multiprocessing (SMP):**
     - In this approach, each processor is self-scheduling.
     - All processes may be in a common ready queue, or each processor may have its own private queue of ready processes.
- If we have multiple processors trying to access and update a common data structure, the scheduler must be programmed carefully.

- We must ensure that two processors do not choose the same process and that processes are not lost from the queue.
- Virtually all modern operating systems support SMP, including Windows, Solaris, Linux, and Mac OS X.

2. Processor Affinity:

- When a processor running on a processor, the data most recently accessed by the process is populate the cache of that respective processor.
- Now consider, if the process migrates to another processor. The contents of cache memory must be invalidated for the first processor, and the cache for the second processor must be repopulated.
- Because of the high cost of invalidating and repopulating caches, most SMP systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor. This is known as **processor affinity.**
- **Soft affinity:**
  1. When an operating system has a policy of attempting to keep a process running on the same processor, this situation is known as soft affinity.
  2. **Example:** Solaris allows processes to be assigned to processor sets, limiting which processes can run on which CPUs.
- **Hard affinity:**
  1. Here, it is possible to migrate a process between processors. Some systems —Like Linux — also provide system calls that support hard affinity, thereby allowing a process to specify that it is not to migrate to other processors.
- The main-memory architecture of a system can affect processor affinity issues.
- Below Figure illustrates an architecture featuring non-uniform memory access (NUMA), in which a CPU has faster access to some parts of main memory than to other parts.

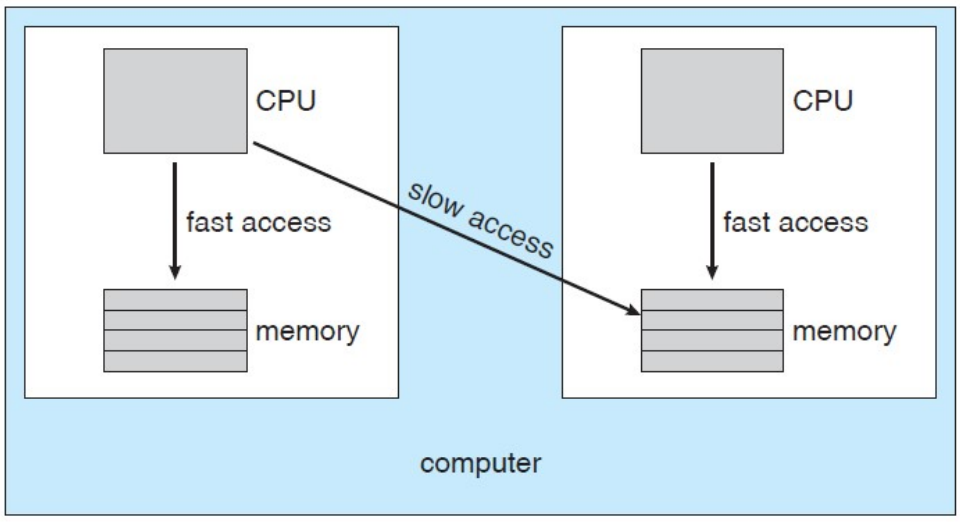


Figure NUMA and CPU scheduling.

- Typically, this occurs in systems containing combined CPU and memory boards.
- The CPUs on a board can access the memory on that board with less delay than they can access memory on other boards in the system.
- If the operating system's CPU scheduler and memory-placement algorithms work together, then a process that is assigned affinity to a particular CPU can be allocated memory on the board where that CPU resides.

3. Load Balancing:

- Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system.
- Load balancing is required in the systems where each processor has its own private queue.
- There are two approaches to load balancing:
    1. Push Migration
    2. Pull Migration

1. Push Migration:

- In push migration, a specific task periodically checks the load on each processor and—if it finds an imbalance—evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors

2. Pull Migration:

- Pull Migration occurs when an idle processor pulls a waiting from a busy processor.

- Push and pull migration need not be mutually exclusive and are in fact often implemented in parallel on load-balancing systems.
- For example, the Linux scheduler and the ULE scheduler available for FreeBSD systems implement both techniques. Linux runs its load balancing algorithm every 200 milliseconds (push migration) or whenever the run queue for a processor is empty (pull migration).

4. Multicore Processors:

- Placing multiple processor cores on the same physical chip, yields a multicore processor.
- Each core has a register set to maintain its architectural state and thus appears to the operating system to be a separate physical processor.
- SMP systems that use multicore processors are faster and consume less power than systems in which each processor has its own physical chip.
- One problem with multicore processors is **memory stall.**
- **A memory stall may occur for various reasons, such as a cache misses (accessing data that is not in cache memory).**
- Below Figure illustrates a memory stall.

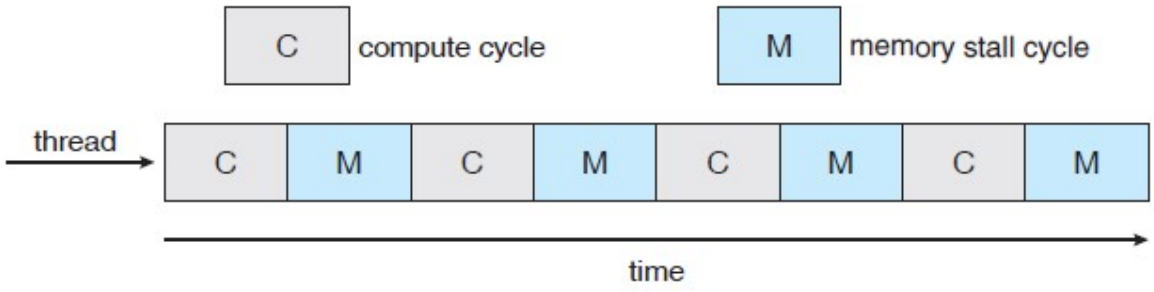


Figure Memory stall.

- In the above figure, the processor can spend up to 50 percent of its time waiting for data to become available from memory.
-

5. **Virtualization and Scheduling:**
   - A system with virtualization, even a single-CPU system, frequently acts like a multiprocessor system.
   - The virtualization software presents one or more virtual CPUs to each of the virtual machines running on the system and then schedules the use of the physical CPUs among the virtual machines.
   - In general, most virtualized environments have one host operating system and many guest operating systems.
   - The host operating system creates and manages the virtual machines, and each virtual machine has a guest operating system installed and applications running within that guest.
   - Depending on how busy the system is, the time slice may take a second or more, resulting in very poor response times for users logged into that virtual machine.
   - Commonly, virtual machines are incorrect because timers take longer to trigger than they would on dedicated CPUs.
   - Virtualization can thus undo the good scheduling-algorithm efforts of the operating systems within virtual machines.

5. Thread scheduling

- On operating systems that support kernel-level threads—not processes—that are being scheduled by the operating system.
- User-level threads are managed by a thread library, and the kernel is unaware of them.
- To run on a CPU, user-level threads must map to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP).
- In this section, we explore **thread scheduling issues** involving user-level and kernel-level threads.

1. **Contention Scope:**
    - On systems implementing the many-to-one and many-to-many models,
        1. The thread library schedules user-level threads to run on an available LWP, a scheme known as **process-contention scope (PCS)**.
        2. When we say the thread library schedules user threads onto available LWPs, we do not mean that the thread is actually running on a CPU; this would require the operating system to schedule the kernel thread onto a physical CPU.
        3. To decide which kernel thread to schedule onto a CPU, the kernel uses **system-contention scope (SCS)**.
        4. Competition for the CPU with SCS scheduling takes place among all threads in the system.
    - Systems using the one to- one model, such as Windows, Solaris, and Linux, schedule threads using only SCS.
    - PCS is done according to priority—the scheduler selects the runnable thread with the highest priority to run.
    - User-level thread priorities are set by the programmer and some thread libraries may allow the programmer to change the priority of a thread.
    - PCS will pre-empt the thread currently running in favour of a higher-priority thread; however, there is no guarantee of time slicing among threads of equal priority.
2. **Pthread Scheduling:**
    - POSIX Pthread API allows either PCS or SCS during thread creation.
    - Pthreads identifies the following contention scope values:
        1. PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling.
        2. PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling.
    - On systems implementing the many-to-many model, the PTHREAD_SCOPE_PROCESS policy schedules user-level threads onto available LWPs.
    - The number of LWPs is maintained by the thread library.
    - The PTHREAD_SCOPE_SYSTEM scheduling policy will create and bind an LWP for each user-level thread on many-to-many systems, effectively mapping threads using the one-to-one policy.

6. Examples

1. **Solaris Scheduling:**
   - Solaris uses **priority-based thread** scheduling where each thread belongs to one of six classes:
     1. Time sharing (TS)
     2. Interactive (IA)
     3. Real time (RT)
     4. System (SYS)
     5. Fair share (FSS)
     6. Fixed priority (FP)
   - Within each class there are different priorities and different scheduling algorithms.

   1. Time sharing (TS):
   - The **default scheduling class** for a process is **time sharing**.
   - The **scheduling policy** for the **time-sharing class** dynamically alters priorities and assigns time slices of different lengths using a multilevel feedback queue.
   - By default, there is an **inverse relationship between priorities and time slices**. The **higher the priority, the smaller the time slice; the lower the priority, the larger the time slice**.

   2. Interactive (IA):
   - Interactive processes have a higher priority, CPU-bound processes, a lower priority.
   - This scheduling policy gives good response time for interactive processes and good throughput for CPU-bound processes.
   - The **interactive class uses the same scheduling policy as the time-sharing class**, but it gives windowing applications—such as those created by the KDE or GNOME window managers—a higher priority for better performance.

   3. Real time (RT):
   - Threads in the real-time class are given the highest priority.
   - This assignment allows a real-time process to have a guaranteed response from the system within a bounded period of time.
   - A real-time process will run before a process in any other class.

   4. System (SYS):
   - Solaris uses the system class to run kernel threads, such as the scheduler and paging daemon.
   - Once established, the priority of a system thread does not change.
   - The system class is reserved for kernel use (user processes running in kernel mode are not in the system class).

5. **Fair share (FSS):**
   - The fair-share class was introduced with Solaris 9.
   - The fair-share scheduling class uses CPU shares instead of priorities to make scheduling decisions.
   - CPU shares indicate entitlement to available CPU resources and are allocated to a set of processes (known as a project).
6. **Fixed priority (FP)**
   - The fixed-priority and fair-share classes were introduced with Solaris 9.
   - Threads in the fixed-priority class have the same priority range as those in the time-sharing class; however, their priorities are not dynamically adjusted.

- **Scheduling Process:**
  - Each scheduling class includes a set of priorities.
  - However, the scheduler converts the class-specific priorities into global priorities and selects the thread with the highest global priority to run.
  - The selected thread runs on the CPU until it (1) blocks, (2) uses its time slice, or (3) is preempted by a higher-priority thread.
  - If there are multiple threads with the same priority, the scheduler uses a round-robin queue.
  - Below illustrates how the six scheduling classes relate to one another and how they map to global priorities.
  - Notice that the kernel maintains 10 threads for servicing interrupts. These threads do not belong to any scheduling class and execute at the highest priority.
  - As mentioned, Solaris has traditionally used the many-to-many model but switched to the one-to-one model beginning with Solaris 9.

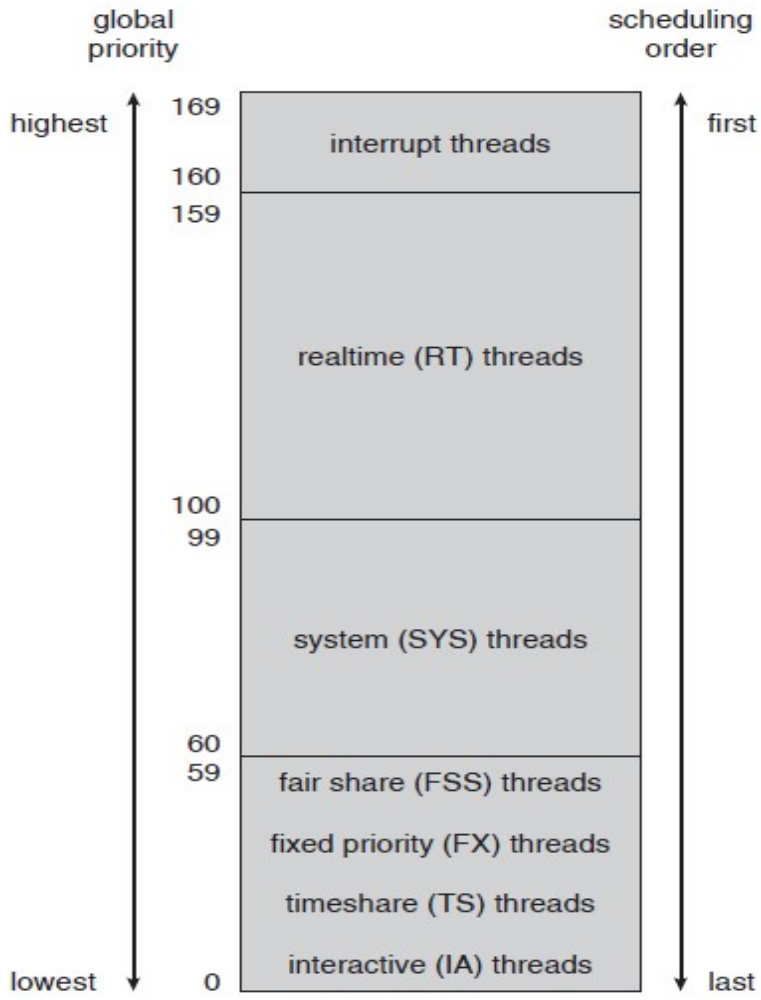


Figure Solaris scheduling.

2. Windows Scheduling

- Windows schedules threads using a **priority-based preemptive** scheduling algorithm.
- The Windows scheduler ensures that the highest-priority thread will always run.
- The portion of the Windows kernel that handles scheduling is called the **dispatcher**.
- A thread selected to run by the dispatcher will run until it is preempted by a higher-priority thread, until it terminates, until its time quantum ends, or until it calls a blocking system call, such as for I/O.
- If a higher-priority real-time thread becomes ready while a lower-priority thread is running, the lower-priority thread will be preempted.
- This preemption gives a real-time thread preferential access to the CPU when the thread needs such access.
- The dispatcher uses a 32-level priority scheme to determine the order of thread execution.
- Priorities are divided into two classes.
    1. The **variable class** contains threads having priorities from1 to 15, and
    2. The **real-time class** contains threads with priorities ranging from 16 to 31. (There is also a thread running at priority 0 that is used for memory management.)
- The dispatcher uses a **queue** for each scheduling priority and traverses the set of queues from highest to lowest until it finds a thread that is ready to run.
- If no ready thread is found, the dispatcher will execute a special thread called the **idle thread**.
- There is a relationship between the numeric priorities of the Windows kernel and the Win32 API.
- The Win32 API identifies several priority classes to which a process can belong. These include:
    1. Real-time Priority Class
    2. High Priority Class
    3. Above Normal Priority Class
    4. Normal Priority Class
    5. Below Normal Priority Class
    6. Idle Priority Class
- A thread within a given priority classes also has a relative priority. The values for relative priorities include:
    1. Time Critical
    2. Highest
    3. Above Normal
    4. Normal
    5. Below Normal

6. Lowest
7. Idle

- The priority of each thread is based on both the priority class it belongs to and its relative priority within that class.
- Each thread has a base priority representing a value in the priority range for the class the thread belongs to.
- By default, the base priority is the value of the NORMAL relative priority for that class.
- The base priorities for each priority class are:
    1. Real-time Priority Class—24
    2. High Priority Class—13
    3. Above Normal Priority Class—10
    4. Normal Priority Class—8
    5. Below Normal Priority Class—6
    6. Idle Priority Class—4
- When a thread's time quantum runs out, that thread is interrupted; if the thread is in the variable-priority class, its priority is lowered. The priority is never lowered below the base priority
- When a user is running an interactive program, the system needs to provide good performance. For this reason, Windows has a special scheduling rule for processes in the NORMAL PRIORITY CLASS.
- Windows distinguishes between the foreground process that is currently selected on the screen and the background processes that are not currently selected.
- When a process moves into the foreground, Windows increases the scheduling quantum by some factor— typically by 3.
- This increase gives the foreground process three times longer to run before a time-sharing preemption occurs.

3. Linux Scheduling

- Prior to Version 2.5, the Linux kernel ran a variation of the traditional UNIXscheduling algorithm.
- Two problems with the traditional UNIX scheduler are that
  1. It does not provide adequate support for SMP systems and
  2. It does not scale well as the number of tasks on the system grows.
- With Version 2.5, the scheduler was overhauled, and the kernel now provides a scheduling algorithm that runs in constant time—known as O(1)—regardless of the number of tasks on the system.
- The new scheduler also provides increased support for SMP, including processor affinity and load balancing, as well as providing fairness and support for interactive tasks.
- The Linux scheduler is a preemptive, priority-based algorithm with two separate priority ranges: a real-time range from0 to 99 and a nice value ranging from 100 to 140.
- These two ranges map into a global priority scheme wherein numerically lower values indicate higher priorities.
- Unlike schedulers for many other systems, including Solaris and Windows, Linux assigns higher-priority tasks longer time quantum and lower-priority tasks shorter time quantum.
- The relationship between priorities and time-slice length is shown in below Figure.

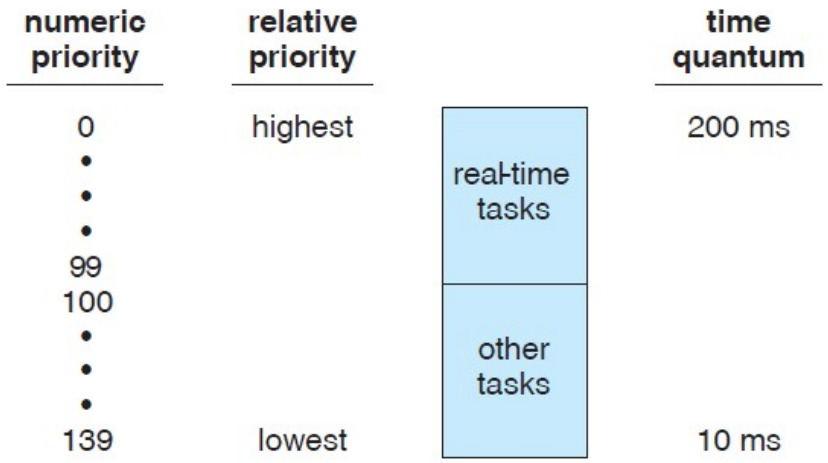


Figure The relationship between priorities and time-slice length.

- A runnable task is considered eligible for execution on the CPU as long as it has time remaining in its time slice.
- When a task has exhausted its time slice, it is considered expired and is not eligible for execution again until all other tasks have also exhausted their time quantum.
- The kernel maintains a list of all runnable tasks in a **run queue** data structure.
- Because of its support for SMP, each processor maintains its own run queue and schedules itself independently.

- Each run queue contains two priority arrays: **active and expired**.
- The **active array** contains all tasks with time remaining in their time slices, and the **expired array** contains all expired tasks.
- Each of these priority arrays contains a list of tasks indexed according to priority see the below figure:

Figure List of tasks indexed according to priority.

- The scheduler chooses the task with the highest priority from the active array for execution on the CPU.
- On multiprocessor machines, this means that each processor is scheduling the highest-priority task from its own run queue structure.
- When all tasks have exhausted their time slices (that is, the active array is empty), the two priority arrays are exchanged: the expired array becomes the active array, and vice versa.
- Linux implements real-time scheduling where Real-time tasks are assigned static priorities.
- All other tasks have dynamic priorities that are based on their nice values plus or minus the value 5.
- A task's dynamic priority is recalculated when the task has exhausted its time quantum and is to be moved to the expired array.
- Thus, when the two arrays are exchanged, all tasks in the new active array have been assigned new priorities and corresponding time slices.

Unit – 2 :: Chapter – 4
Inter-process Communication

1. Race conditions
2. Critical Regions
3. Mutual exclusion with busy waiting
4. Sleep and wakeup
5. Semaphores
6. Mutexes
7. Monitors
8. Message passing
9. Barriers
10. Classical IPC Problems

10.1. Dining philosophers problem

10.2. Readers and writers problem

2.3 INTERPROCESS COMMUNICATION

Processes frequently need to communicate with other processes. For example, in a shell pipeline, the output of the first process must be passed to the second process, and so on down the line. Thus there is a need for communication between processes, preferably in a well-structured way not using interrupts. In the following sections we will look at some of the issues related to this **InterProcess Communication**, or **IPC**.

Very briefly, there are three issues here. The first was alluded to above: how one process can pass information to another. The second has to do with making sure two or more processes do not get in each other's way, for example, two processes in an airline reservation system each trying to grab the last seat on a plane for a different customer. The third concerns proper sequencing when dependencies are present: if process *A* produces data and process *B* prints them, *B* has to wait until *A* has produced some data before starting to print. We will examine all three of these issues starting in the next section.

It is also important to mention that two of these issues apply equally well to threads. The first one—passing information—is easy for threads since they share a common address space (threads in different address spaces that need to communicate fall under the heading of communicating processes). However, the other two—keeping out of each other's hair and proper sequencing—apply equally well to threads. The same problems exist and the same solutions apply. Below we will discuss the problem in the context of processes, but please keep in mind that the same problems and solutions also apply to threads.

2.3.1 Race Conditions

In some operating systems, processes that are working together may share some common storage that each one can read and write. The shared storage may be in main memory (possibly in a kernel data structure) or it may be a shared file; the location of the shared memory does not change the nature of the communication or the problems that arise. To see how interprocess communication works in practice, let us now consider a simple but common example: a print spooler. When a process

wants to print a file, it enters the file name in a special **spooler directory**. Another process, the **printer daemon**, periodically checks to see if there are any files to be printed, and if there are, it prints them and then removes their names from the directory.

Imagine that our spooler directory has a very large number of slots, numbered 0, 1, 2, ..., each one capable of holding a file name. Also imagine that there are two shared variables, *out*, which points to the next file to be printed, and *in*, which points to the next free slot in the directory. These two variables might well be kept in a two-word file available to all processes. At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files queued for printing). More or less simultaneously, processes *A* and *B* decide they want to queue a file for printing. This situation is shown in Fig. 2-21.

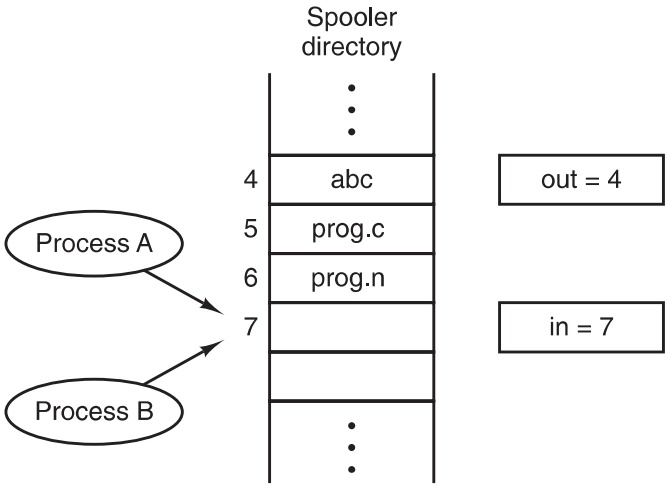


Figure 2-21. Two processes want to access shared memory at the same time.

In jurisdictions where Murphy's law† is applicable, the following could happen. Process *A* reads *in* and stores the value, 7, in a local variable called *next_free_slot*. Just then a clock interrupt occurs and the CPU decides that process *A* has run long enough, so it switches to process *B*. Process *B* also reads *in* and also gets a 7. It, too, stores it in *its* local variable *next_free_slot*. At this instant both processes think that the next available slot is 7.

Process *B* now continues to run. It stores the name of its file in slot 7 and updates *in* to be an 8. Then it goes off and does other things.

Eventually, process *A* runs again, starting from the place it left off. It looks at *next_free_slot*, finds a 7 there, and writes its file name in slot 7, erasing the name that process *B* just put there. Then it computes *next_free_slot* + 1, which is 8, and sets *in* to 8. The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process *B* will never receive any output. User *B* will hang around the printer for years, wistfully hoping for output that

† If something can go wrong, it will.

never comes. Situations like this, where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called **race conditions**. Debugging programs containing race conditions is no fun at all. The results of most test runs are fine, but once in a blue moon something weird and unexplained happens. Unfortunately, with increasing parallelism due to increasing numbers of cores, race condition are becoming more common.

2.3.2 Critical Regions

How do we avoid race conditions? The key to preventing trouble here and in many other situations involving shared memory, shared files, and shared everything else is to find some way to prohibit more than one process from reading and writing the shared data at the same time. Put in other words, what we need is **mutual exclusion**, that is, some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing. The difficulty above occurred because process *B* started using one of the shared variables before process *A* was finished with it. The choice of appropriate primitive operations for achieving mutual exclusion is a major design issue in any operating system, and a subject that we will examine in great detail in the following sections.

The problem of avoiding race conditions can also be formulated in an abstract way. Part of the time, a process is busy doing internal computations and other things that do not lead to race conditions. However, sometimes a process has to access shared memory or files, or do other critical things that can lead to races. That part of the program where the shared memory is accessed is called the **critical region** or **critical section**. If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid races.

Although this requirement avoids race conditions, it is not sufficient for having parallel processes cooperate correctly and efficiently using shared data. We need four conditions to hold to have a good solution:

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block any process.
4. No process should have to wait forever to enter its critical region.

In an abstract sense, the behavior that we want is shown in Fig. 2-22. Here process *A* enters its critical region at time T_1. A little later, at time T_2 process *B* attempts to enter its critical region but fails because another process is already in its critical region and we allow only one at a time. Consequently, *B* is temporarily suspended until time T_3 when *A* leaves its critical region, allowing *B* to enter immediately. Eventually *B* leaves (at T_4) and we are back to the original situation with no processes in their critical regions.

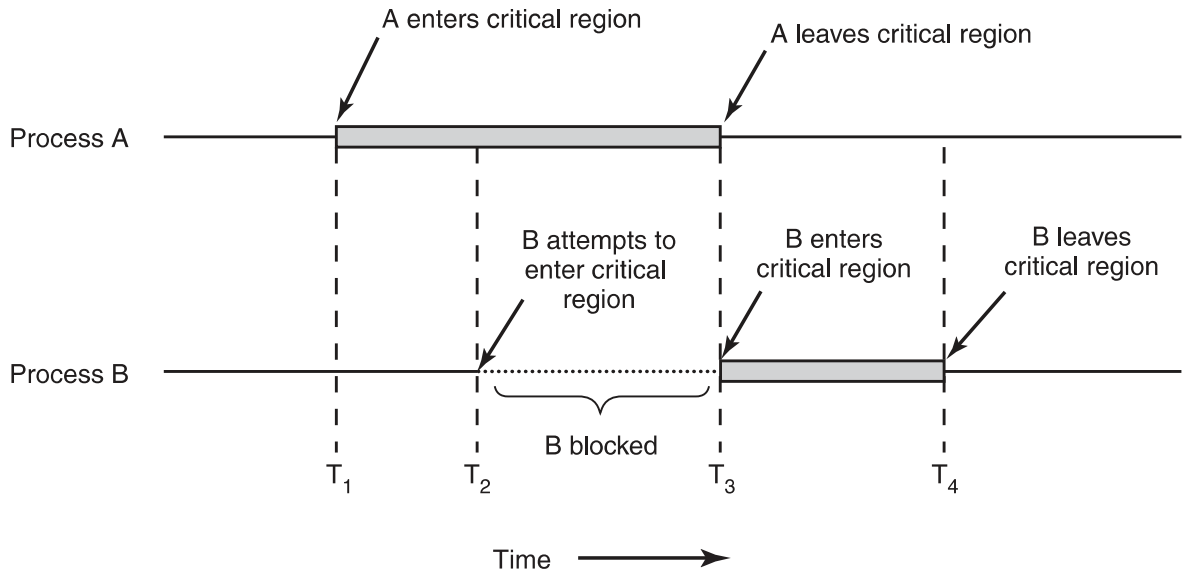


Figure 2-22. Mutual exclusion using critical regions.

2.3.3 Mutual Exclusion with Busy Waiting

In this section we will examine various proposals for achieving mutual exclusion, so that while one process is busy updating shared memory in its critical region, no other process will enter *its* critical region and cause trouble.

Disabling Interrupts

On a single-processor system, the simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it. With interrupts disabled, no clock interrupts can occur. The CPU is only switched from process to process as a result of clock or other interrupts, after all, and with interrupts turned off the CPU will not be switched to another process. Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will intervene.

This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts. What if one of them did it, and never turned them on again? That could be the end of the system. Furthermore, if the system is a multiprocessor (with two or more CPUs) disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue running and can access the shared memory.

On the other hand, it is frequently convenient for the kernel itself to disable interrupts for a few instructions while it is updating variables or especially lists. If an interrupt occurrs while the list of ready processes, for example, is in an inconsistent state, race conditions could occur. The conclusion is: disabling interrupts is

often a useful technique within the operating system itself but is not appropriate as a general mutual exclusion mechanism for user processes.

The possibility of achieving mutual exclusion by disabling interrupts—even within the kernel—is becoming less every day due to the increasing number of multicore chips even in low-end PCs. Two cores are already common, four are present in many machines, and eight, 16, or 32 are not far behind. In a multicore (i.e., multiprocessor system) disabling the interrupts of one CPU does not prevent other CPUs from interfering with operations the first CPU is performing. Consequently, more sophisticated schemes are needed.

Lock Variables

As a second attempt, let us look for a software solution. Consider having a single, shared (lock) variable, initially 0. When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.

Unfortunately, this idea contains exactly the same fatal flaw that we saw in the spooler directory. Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.

Now you might think that we could get around this problem by first reading out the lock value, then checking it again just before storing into it, but that really does not help. The race now occurs if the second process modifies the lock just after the first process has finished its second check.

Strict Alternation

A third approach to the mutual exclusion problem is shown in Fig. 2-23. This program fragment, like nearly all the others in this book, is written in C. C was chosen here because real operating systems are virtually always written in C (or occasionally C++), but hardly ever in languages like Java, Python, or Haskell. C is powerful, efficient, and predictable, characteristics critical for writing operating systems. Java, for example, is not predictable because it might run out of storage at a critical moment and need to invoke the garbage collector to reclaim memory at a most inopportune time. This cannot happen in C because there is no garbage collection in C. A quantitative comparison of C, C++, Java, and four other languages is given by Prechelt (2000).

In Fig. 2-23, the integer variable *turn*, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory. Initially, process 0 inspects *turn*, finds it to be 0, and enters its critical region. Process 1 also

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

(a)

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

(b)

Figure 2-23. A proposed solution to the critical-region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

finds it to be 0 and therefore sits in a tight loop continually testing *turn* to see when it becomes 1. Continuously testing a variable until some value appears is called **busy waiting**. It should usually be avoided, since it wastes CPU time. Only when there is a reasonable expectation that the wait will be short is busy waiting used. A lock that uses busy waiting is called a **spin lock**.

When process 0 leaves the critical region, it sets *turn* to 1, to allow process 1 to enter its critical region. Suppose that process 1 finishes its critical region quickly, so that both processes are in their noncritical regions, with *turn* set to 0. Now process 0 executes its whole loop quickly, exiting its critical region and setting *turn* to 1. At this point *turn* is 1 and both processes are executing in their noncritical regions.

Suddenly, process 0 finishes its noncritical region and goes back to the top of its loop. Unfortunately, it is not permitted to enter its critical region now, because *turn* is 1 and process 1 is busy with its noncritical region. It hangs in its while loop until process 1 sets *turn* to 0. Put differently, taking turns is not a good idea when one of the processes is much slower than the other.

This situation violates condition 3 set out above: process 0 is being blocked by a process not in its critical region. Going back to the spooler directory discussed above, if we now associate the critical region with reading and writing the spooler directory, process 0 would not be allowed to print another file because process 1 was doing something else.

In fact, this solution requires that the two processes strictly alternate in entering their critical regions, for example, in spooling files. Neither one would be permitted to spool two in a row. While this algorithm does avoid all races, it is not really a serious candidate as a solution because it violates condition 3.

Peterson's Solution

By combining the idea of taking turns with the idea of lock variables and warning variables, a Dutch mathematician, T. Dekker, was the first one to devise a software solution to the mutual exclusion problem that does not require strict alternation. For a discussion of Dekker's algorithm, see Dijkstra (1965).

In 1981, G. L. Peterson discovered a much simpler way to achieve mutual exclusion, thus rendering Dekker's solution obsolete. Peterson's algorithm is shown in Fig. 2-24. This algorithm consists of two procedures written in ANSI C, which means that function prototypes should be supplied for all the functions defined and used. However, to save space, we will not show prototypes here or later.

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

Figure 2-24. Peterson's solution for achieving mutual exclusion.

Before using the shared variables (i.e., before entering its critical region), each process calls *enter_region* with its own process number, 0 or 1, as parameter. This call will cause it to wait, if need be, until it is safe to enter. After it has finished with the shared variables, the process calls *leave_region* to indicate that it is done and to allow the other process to enter, if it so desires.

Let us see how this solution works. Initially neither process is in its critical region. Now process 0 calls *enter_region*. It indicates its interest by setting its array element and sets *turn* to 0. Since process 1 is not interested, *enter_region* returns immediately. If process 1 now makes a call to *enter_region*, it will hang there until *interested*[0] goes to *FALSE*, an event that happens only when process 0 calls *leave_region* to exit the critical region.

Now consider the case that both processes call *enter_region* almost simultaneously. Both will store their process number in *turn*. Whichever store is done last is the one that counts; the first one is overwritten and lost. Suppose that process 1 stores last, so *turn* is 1. When both processes come to the while statement, process 0 executes it zero times and enters its critical region. Process 1 loops and does not enter its critical region until process 0 exits its critical region.

The TSL Instruction

Now let us look at a proposal that requires a little help from the hardware. Some computers, especially those designed with multiple processors in mind, have an instruction like

```
TSL RX,LOCK
```

(Test and Set Lock) that works as follows. It reads the contents of the memory word *lock* into register RX and then stores a nonzero value at the memory address *lock*. The operations of reading the word and storing into it are guaranteed to be indivisible—no other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

It is important to note that locking the memory bus is very different from disabling interrupts. Disabling interrupts then performing a read on a memory word followed by a write does not prevent a second processor on the bus from accessing the word between the read and the write. In fact, disabling interrupts on processor 1 has no effect at all on processor 2. The only way to keep processor 2 out of the memory until processor 1 is finished is to lock the bus, which requires a special hardware facility (basically, a bus line asserting that the bus is locked and not available to processors other than the one that locked it).

To use the TSL instruction, we will use a shared variable, *lock*, to coordinate access to shared memory. When *lock* is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory. When it is done, the process sets *lock* back to 0 using an ordinary move instruction.

How can this instruction be used to prevent two processes from simultaneously entering their critical regions? The solution is given in Fig. 2-25. There a four-instruction subroutine in a fictitious (but typical) assembly language is shown. The first instruction copies the old value of *lock* to the register and then sets *lock* to 1. Then the old value is compared with 0. If it is nonzero, the lock was already set, so the program just goes back to the beginning and tests it again. Sooner or later it will become 0 (when the process currently in its critical region is done with its critical region), and the subroutine returns, with the lock set. Clearing the lock is very simple. The program just stores a 0 in *lock*. No special synchronization instructions are needed.

One solution to the critical-region problem is now easy. Before entering its critical region, a process calls *enter_region*, which does busy waiting until the lock is free; then it acquires the lock and returns. After leaving the critical region the process calls *leave_region*, which stores a 0 in *lock*. As with all solutions based on critical regions, the processes must call *enter_region* and *leave_region* at the correct times for the method to work. If one process cheats, the mutual exclusion will fail. In other words, critical regions work only if the processes cooperate.

```
enter_region:
        TSL REGISTER,LOCK       | copy lock to register and set lock to 1
        CMP REGISTER,#0         | was lock zero?
        JNE enter_region        | if it was not zero, lock was set, so loop
        RET                     | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0            | store a 0 in lock
        RET                     | return to caller
```

Figure 2-25. Entering and leaving a critical region using the TSL instruction.

An alternative instruction to TSL is XCHG, which exchanges the contents of two locations atomically, for example, a register and a memory word. The code is shown in Fig. 2-26, and, as can be seen, is essentially the same as the solution with TSL. All Intel x86 CPUs use XCHG instruction for low-level synchronization.

```
enter_region:
        MOVE REGISTER,#1        | put a 1 in the register
        XCHG REGISTER,LOCK      | swap the contents of the register and lock variable
        CMP REGISTER,#0         | was lock zero?
        JNE enter_region        | if it was non zero, lock was set, so loop
        RET                     | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0            | store a 0 in lock
        RET                     | return to caller
```

Figure 2-26. Entering and leaving a critical region using the XCHG instruction.

2.3.4 Sleep and Wakeup

Both Peterson's solution and the solutions using TSL or XCHG are correct, but both have the defect of requiring busy waiting. In essence, what these solutions do is this: when a process wants to enter its critical region, it checks to see if the entry is allowed. If it is not, the process just sits in a tight loop waiting until it is.

Not only does this approach waste CPU time, but it can also have unexpected effects. Consider a computer with two processes, *H*, with high priority, and *L*, with low priority. The scheduling rules are such that *H* is run whenever it is in ready state. At a certain moment, with *L* in its critical region, *H* becomes ready to run (e.g., an I/O operation completes). *H* now begins busy waiting, but since *L* is never

scheduled while *H* is running, *L* never gets the chance to leave its critical region, so *H* loops forever. This situation is sometimes referred to as the **priority inversion problem**.

Now let us look at some interprocess communication primitives that block instead of wasting CPU time when they are not allowed to enter their critical regions. One of the simplest is the pair sleep and wakeup. Sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up. The wakeup call has one parameter, the process to be awakened. Alternatively, both sleep and wakeup each have one parameter, a memory address used to match up sleeps with wakeups.

The Producer-Consumer Problem

As an example of how these primitives can be used, let us consider the **producer-consumer** problem (also known as the **bounded-buffer** problem). Two processes share a common, fixed-size buffer. One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out. (It is also possible to generalize the problem to have *m* producers and *n* consumers, but we will consider only the case of one producer and one consumer because this assumption simplifies the solutions.)

Trouble arises when the producer wants to put a new item in the buffer, but it is already full. The solution is for the producer to go to sleep, to be awakened when the consumer has removed one or more items. Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up.

This approach sounds simple enough, but it leads to the same kinds of race conditions we saw earlier with the spooler directory. To keep track of the number of items in the buffer, we will need a variable, *count*. If the maximum number of items the buffer can hold is *N*, the producer's code will first test to see if *count* is *N*. If it is, the producer will go to sleep; if it is not, the producer will add an item and increment *count*.

The consumer's code is similar: first test *count* to see if it is 0. If it is, go to sleep; if it is nonzero, remove an item and decrement the counter. Each of the processes also tests to see if the other should be awakened, and if so, wakes it up. The code for both producer and consumer is shown in Fig. 2-27.

To express system calls such as sleep and wakeup in C, we will show them as calls to library routines. They are not part of the standard C library but presumably would be made available on any system that actually had these system calls. The procedures *insert_item* and *remove_item*, which are not shown, handle the bookkeeping of putting items into the buffer and taking items out of the buffer.

Now let us get back to the race condition. It can occur because access to *count* is unconstrained. As a consequence, the following situation could possibly occur. The buffer is empty and the consumer has just read *count* to see if it is 0. At that

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Figure 2-27. The producer-consumer problem with a fatal race condition.

instant, the scheduler decides to stop running the consumer temporarily and start running the producer. The producer inserts an item in the buffer, increments *count*, and notices that it is now 1. Reasoning that *count* was just 0, and thus the consumer must be sleeping, the producer calls *wakeup* to wake the consumer up.

Unfortunately, the consumer is not yet logically asleep, so the wakeup signal is lost. When the consumer next runs, it will test the value of *count* it previously read, find it to be 0, and go to sleep. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

The essence of the problem here is that a wakeup sent to a process that is not (yet) sleeping is lost. If it were not lost, everything would work. A quick fix is to modify the rules to add a **wakeup waiting bit** to the picture. When a wakeup is sent to a process that is still awake, this bit is set. Later, when the process tries to go to sleep, if the wakeup waiting bit is on, it will be turned off, but the process will stay awake. The wakeup waiting bit is a piggy bank for storing wakeup signals. The consumer clears the wakeup waiting bit in every iteration of the loop.

While the wakeup waiting bit saves the day in this simple example, it is easy to construct examples with three or more processes in which one wakeup waiting bit is insufficient. We could make another patch and add a second wakeup waiting bit, or maybe 8 or 32 of them, but in principle the problem is still there.

2.3.5 Semaphores

This was the situation in 1965, when E. W. Dijkstra (1965) suggested using an integer variable to count the number of wakeups saved for future use. In his proposal, a new variable type, which he called a **semaphore**, was introduced. A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending.

Dijkstra proposed having two operations on semaphores, now usually called down and up (generalizations of sleep and wakeup, respectively). The down operation on a semaphore checks to see if the value is greater than 0. If so, it decrements the value (i.e., uses up one stored wakeup) and just continues. If the value is 0, the process is put to sleep without completing the down for the moment. Checking the value, changing it, and possibly going to sleep, are all done as a single, indivisible **atomic action**. It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked. This atomicity is absolutely essential to solving synchronization problems and avoiding race conditions. Atomic actions, in which a group of related operations are either all performed without interruption or not performed at all, are extremely important in many other areas of computer science as well.

The up operation increments the value of the semaphore addressed. If one or more processes were sleeping on that semaphore, unable to complete an earlier down operation, one of them is chosen by the system (e.g., at random) and is allowed to complete its down. Thus, after an up on a semaphore with processes sleeping on it, the semaphore will still be 0, but there will be one fewer process sleeping on it. The operation of incrementing the semaphore and waking up one process is also indivisible. No process ever blocks doing an up, just as no process ever blocks doing a wakeup in the earlier model.

As an aside, in Dijkstra's original paper, he used the names P and V instead of down and up, respectively. Since these have no mnemonic significance to people who do not speak Dutch and only marginal significance to those who do—*Proberen* (try) and *Verhogen* (raise, make higher)—we will use the terms down and up instead. These were first introduced in the Algol 68 programming language.

Solving the Producer-Consumer Problem Using Semaphores

Semaphores solve the lost-wakeup problem, as shown in Fig. 2-28. To make them work correctly, it is essential that they be implemented in an indivisible way. The normal way is to implement up and down as system calls, with the operating

system briefly disabling all interrupts while it is testing the semaphore, updating it, and putting the process to sleep, if necessary. As all of these actions take only a few instructions, no harm is done in disabling interrupts. If multiple CPUs are being used, each semaphore should be protected by a lock variable, with the TSL or XCHG instructions used to make sure that only one CPU at a time examines the semaphore.

Be sure you understand that using TSL or XCHG to prevent several CPUs from accessing the semaphore at the same time is quite different from the producer or consumer busy waiting for the other to empty or fill the buffer. The semaphore operation will take only a few microseconds, whereas the producer or consumer might take arbitrarily long.

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );               /* generate something to put in buffer */
          down(&empty);                         /* decrement empty count */
          down(&mutex);                         /* enter critical region */
          insert_item(item);                    /* put new item in buffer */
          up(&mutex);                           /* leave critical region */
          up(&full);                            /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                          /* decrement full count */
          down(&mutex);                         /* enter critical region */
          item = remove_item( );                /* take item from buffer */
          up(&mutex);                           /* leave critical region */
          up(&empty);                           /* increment count of empty slots */
          consume_item(item);                   /* do something with the item */
     }
}
```

Figure 2-28. The producer-consumer problem using semaphores.

This solution uses three semaphores: one called *full* for counting the number of slots that are full, one called *empty* for counting the number of slots that are empty, and one called *mutex* to make sure the producer and consumer do not access the buffer at the same time. *Full* is initially 0, *empty* is initially equal to the number of slots in the buffer, and *mutex* is initially 1. Semaphores that are initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical region at the same time are called **binary semaphores**. If each process does a down just before entering its critical region and an up just after leaving it, mutual exclusion is guaranteed.

Now that we have a good interprocess communication primitive at our disposal, let us go back and look at the interrupt sequence of Fig. 2-5 again. In a system using semaphores, the natural way to hide interrupts is to have a semaphore, initially set to 0, associated with each I/O device. Just after starting an I/O device, the managing process does a down on the associated semaphore, thus blocking immediately. When the interrupt comes in, the interrupt handler then does an up on the associated semaphore, which makes the relevant process ready to run again. In this model, step 5 in Fig. 2-5 consists of doing an up on the device's semaphore, so that in step 6 the scheduler will be able to run the device manager. Of course, if several processes are now ready, the scheduler may choose to run an even more important process next. We will look at some of the algorithms used for scheduling later on in this chapter.

In the example of Fig. 2-28, we have actually used semaphores in two different ways. This difference is important enough to make explicit. The *mutex* semaphore is used for mutual exclusion. It is designed to guarantee that only one process at a time will be reading or writing the buffer and the associated variables. This mutual exclusion is required to prevent chaos. We will study mutual exclusion and how to achieve it in the next section.

The other use of semaphores is for **synchronization**. The *full* and *empty* semaphores are needed to guarantee that certain event sequences do or do not occur. In this case, they ensure that the producer stops running when the buffer is full, and that the consumer stops running when it is empty. This use is different from mutual exclusion.

2.3.6 Mutexes

When the semaphore's ability to count is not needed, a simplified version of the semaphore, called a mutex, is sometimes used. Mutexes are good only for managing mutual exclusion to some shared resource or piece of code. They are easy and efficient to implement, which makes them especially useful in thread packages that are implemented entirely in user space.

A **mutex** is a shared variable that can be in one of two states: unlocked or locked. Consequently, only 1 bit is required to represent it, but in practice an integer often is used, with 0 meaning unlocked and all other values meaning locked.

Two procedures are used with mutexes. When a thread (or process) needs access to a critical region, it calls *mutex_lock*. If the mutex is currently unlocked (meaning that the critical region is available), the call succeeds and the calling thread is free to enter the critical region.

On the other hand, if the mutex is already locked, the calling thread is blocked until the thread in the critical region is finished and calls *mutex_unlock*. If multiple threads are blocked on the mutex, one of them is chosen at random and allowed to acquire the lock.

Because mutexes are so simple, they can easily be implemented in user space provided that a TSL or XCHG instruction is available. The code for *mutex_lock* and *mutex_unlock* for use with a user-level threads package are shown in Fig. 2-29. The solution with XCHG is essentially the same.

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

Figure 2-29. Implementation of *mutex_lock* and *mutex_unlock*.

The code of *mutex_lock* is similar to the code of *enter_region* of Fig. 2-25 but with a crucial difference. When *enter_region* fails to enter the critical region, it keeps testing the lock repeatedly (busy waiting). Eventually, the clock runs out and some other process is scheduled to run. Sooner or later the process holding the lock gets to run and releases it.

With (user) threads, the situation is different because there is no clock that stops threads that have run too long. Consequently, a thread that tries to acquire a lock by busy waiting will loop forever and never acquire the lock because it never allows any other thread to run and release the lock.

That is where the difference between *enter_region* and *mutex_lock* comes in. When the later fails to acquire a lock, it calls *thread_yield* to give up the CPU to another thread. Consequently there is no busy waiting. When the thread runs the next time, it tests the lock again.

Since *thread_yield* is just a call to the thread scheduler in user space, it is very fast. As a consequence, neither *mutex_lock* nor *mutex_unlock* requires any kernel calls. Using them, user-level threads can synchronize entirely in user space using procedures that require only a handful of instructions.

The mutex system that we have described above is a bare-bones set of calls. With all software, there is always a demand for more features, and synchronization primitives are no exception. For example, sometimes a thread package offers a call *mutex_trylock* that either acquires the lock or returns a code for failure, but does not block. This call gives the thread the flexibility to decide what to do next if there are alternatives to just waiting.

There is a subtle issue that up until now we have glossed over but which is worth at least making explicit. With a user-space threads package there is no problem with multiple threads having access to the same mutex, since all the threads operate in a common address space. However, with most of the earlier solutions, such as Peterson's algorithm and semaphores, there is an unspoken assumption that multiple processes have access to at least some shared memory, perhaps only one word, but something. If processes have disjoint address spaces, as we have consistently said, how can they share the *turn* variable in Peterson's algorithm, or semaphores or a common buffer?

There are two answers. First, some of the shared data structures, such as the semaphores, can be stored in the kernel and accessed only by means of system calls. This approach eliminates the problem. Second, most modern operating systems (including UNIX and Windows) offer a way for processes to share some portion of their address space with other processes. In this way, buffers and other data structures can be shared. In the worst case, that nothing else is possible, a shared file can be used.

If two or more processes share most or all of their address spaces, the distinction between processes and threads becomes somewhat blurred but is nevertheless present. Two processes that share a common address space still have different open files, alarm timers, and other per-process properties, whereas the threads within a single process share them. And it is always true that multiple processes sharing a common address space never have the efficiency of user-level threads since the kernel is deeply involved in their management.

Futexes

With increasing parallelism, efficient synchronization and locking is very important for performance. Spin locks are fast if the wait is short, but waste CPU cycles if not. If there is much contention, it is therefore more efficient to block the process and let the kernel unblock it only when the lock is free. Unfortunately, this has the inverse problem: it works well under heavy contention, but continuously switching to the kernel is expensive if there is very little contention to begin with. To make matters worse, it may not be easy to predict the amount of lock contention.

One interesting solution that tries to combine the best of both worlds is known as **futex**, or "fast user space mutex." A futex is a feature of Linux that implements basic locking (much like a mutex) but avoids dropping into the kernel unless it

really has to. Since switching to the kernel and back is quite expensive, doing so improves performance considerably. A futex consists of two parts: a kernel service and a user library. The kernel service provides a "wait queue" that allows multiple processes to wait on a lock. They will not run, unless the kernel explicitly unblocks them. For a process to be put on the wait queue requires an (expensive) system call and should be avoided. In the absence of contention, therefore, the futex works completely in user space. Specifically, the processes share a common lock variable—a fancy name for an aligned 32-bit integer that serves as the lock. Suppose the lock is initially 1—which we assume to mean that the lock is free. A thread grabs the lock by performing an atomic "decrement and test" (atomic functions in Linux consist of inline assembly wrapped in C functions and are defined in header files). Next, the thread inspects the result to see whether or not the lock was free. If it was not in the locked state, all is well and our thread has successfully grabbed the lock. However, if the lock is held by another thread, our thread has to wait. In that case, the futex library does not spin, but uses a system call to put the thread on the wait queue in the kernel. Hopefully, the cost of the switch to the kernel is now justified, because the thread was blocked anyway. When a thread is done with the lock, it releases the lock with an atomic "increment and test" and checks the result to see if any processes are still blocked on the kernel wait queue. If so, it will let the kernel know that it may unblock one or more of these processes. If there is no contention, the kernel is not involved at all.

Mutexes in Pthreads

Pthreads provides a number of functions that can be used to synchronize threads. The basic mechanism uses a mutex variable, which can be locked or unlocked, to guard each critical region. A thread wishing to enter a critical region first tries to lock the associated mutex. If the mutex is unlocked, the thread can enter immediately and the lock is atomically set, preventing other threads from entering. If the mutex is already locked, the calling thread is blocked until it is unlocked. If multiple threads are waiting on the same mutex, when it is unlocked, only one of them is allowed to continue and relock it. These locks are not mandatory. It is up to the programmer to make sure threads use them correctly.

The major calls relating to mutexes are shown in Fig. 2-30. As expected, mutexes can be created and destroyed. The calls for performing these operations are *pthread_mutex_init* and *pthread_mutex_destroy*, respectively. They can also be locked—by *pthread_mutex_lock*—which tries to acquire the lock and blocks if is already locked. There is also an option for trying to lock a mutex and failing with an error code instead of blocking if it is already blocked. This call is *pthread_mutex_trylock*. This call allows a thread to effectively do busy waiting if that is ever needed. Finally, *pthread_mutex_unlock* unlocks a mutex and releases exactly one thread if one or more are waiting on it. Mutexes can also have attributes, but these are used only for specialized purposes.

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Figure 2-30. Some of the Pthreads calls relating to mutexes.

In addition to mutexes, Pthreads offers a second synchronization mechanism: **condition variables**. Mutexes are good for allowing or blocking access to a critical region. Condition variables allow threads to block due to some condition not being met. Almost always the two methods are used together. Let us now look at the interaction of threads, mutexes, and condition variables in a bit more detail.

As a simple example, consider the producer-consumer scenario again: one thread puts things in a buffer and another one takes them out. If the producer discovers that there are no more free slots available in the buffer, it has to block until one becomes available. Mutexes make it possible to do the check atomically without interference from other threads, but having discovered that the buffer is full, the producer needs a way to block and be awakened later. This is what condition variables allow.

The most important calls related to condition variables are shown in Fig. 2-31. As you would probably expect, there are calls to create and destroy condition variables. They can have attributes and there are various calls for managing them (not shown). The primary operations on condition variables are *pthread_cond_wait* and *pthread_cond_signal*. The former blocks the calling thread until some other thread signals it (using the latter call). The reasons for blocking and waiting are not part of the waiting and signaling protocol, of course. The blocking thread often is waiting for the signaling thread to do some work, release some resource, or perform some other activity. Only then can the blocking thread continue. The condition variables allow this waiting and blocking to be done atomically. The *pthread_cond_broadcast* call is used when there are multiple threads potentially all blocked and waiting for the same signal.

Condition variables and mutexes are always used together. The pattern is for one thread to lock a mutex, then wait on a conditional variable when it cannot get what it needs. Eventually another thread will signal it and it can continue. The *pthread_cond_wait* call atomically unlocks the mutex it is holding. For this reason, the mutex is one of the parameters.

It is also worth noting that condition variables (unlike semaphores) have no memory. If a signal is sent to a condition variable on which no thread is waiting, the signal is lost. Programmers have to be careful not to lose signals.

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

Figure 2-31. Some of the Pthreads calls relating to condition variables.

As an example of how mutexes and condition variables are used, Fig. 2-32 shows a very simple producer-consumer problem with a single buffer. When the producer has filled the buffer, it must wait until the consumer empties it before producing the next item. Similarly, when the consumer has removed an item, it must wait until the producer has produced another one. While very simple, this example illustrates the basic mechanisms. The statement that puts a thread to sleep should always check the condition to make sure it is satisfied before continuing, as the thread might have been awakened due to a UNIX signal or some other reason.

2.3.7 Monitors

With semaphores and mutexes interprocess communication looks easy, right? Forget it. Look closely at the order of the downs before inserting or removing items from the buffer in Fig. 2-28. Suppose that the two downs in the producer's code were reversed in order, so *mutex* was decremented before *empty* instead of after it. If the buffer were completely full, the producer would block, with *mutex* set to 0. Consequently, the next time the consumer tried to access the buffer, it would do a down on *mutex*, now 0, and block too. Both processes would stay blocked forever and no more work would ever be done. This unfortunate situation is called a deadlock. We will study deadlocks in detail in Chap. 6.

This problem is pointed out to show how careful you must be when using semaphores. One subtle error and everything comes to a grinding halt. It is like programming in assembly language, only worse, because the errors are race conditions, deadlocks, and other forms of unpredictable and irreproducible behavior.

To make it easier to write correct programs, Brinch Hansen (1973) and Hoare (1974) proposed a higher-level synchronization primitive called a **monitor**. Their proposals differed slightly, as described below. A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package. Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor. Figure 2-33 illustrates a monitor written in an imaginary language, Pidgin Pascal. C cannot be used here because monitors are a *language* concept and C does not have them.

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

#define MAX 1000000000                          /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;                    /* used for signaling */
int buffer = 0;                                 /* buffer used between producer and consumer */

void *producer(void *ptr)                       /* produce data */
{       int i;

        for (i= 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);     /* get exclusive access to buffer */
                while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
                buffer = i;                         /* put item in buffer */
                pthread_cond_signal(&condc);        /* wake up consumer */
                pthread_mutex_unlock(&the_mutex); /* release access to buffer */
        }
        pthread_exit(0);
}

void *consumer(void *ptr)                       /* consume data */
{       int i;

        for (i = 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);     /* get exclusive access to buffer */
                while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
                buffer = 0;                         /* take item out of buffer */
                pthread_cond_signal(&condp);        /* wake up producer */
                pthread_mutex_unlock(&the_mutex); /* release access to buffer */
        }
        pthread_exit(0);
}

int main(int argc, char **argv)
{
        pthread_t pro, con;
        pthread_mutex_init(&the_mutex, 0);
        pthread_cond_init(&condc, 0);
        pthread_cond_init(&condp, 0);
        pthread_create(&con, 0, consumer, 0);
        pthread_create(&pro, 0, producer, 0);
        pthread_join(pro, 0);
        pthread_join(con, 0);
        pthread_cond_destroy(&condc);
        pthread_cond_destroy(&condp);
        pthread_mutex_destroy(&the_mutex);
}
```

Figure 2-32. Using threads to solve the producer-consumer problem.

Monitors have an important property that makes them useful for achieving mutual exclusion: only one process can be active in a monitor at any instant. Monitors are a programming-language construct, so the compiler knows they are special and can handle calls to monitor procedures differently from other procedure calls. Typically, when a process calls a monitor procedure, the first few instructions of the procedure will check to see if any other process is currently active within the monitor. If so, the calling process will be suspended until the other process has left the monitor. If no other process is using the monitor, the calling process may enter.

It is up to the compiler to implement mutual exclusion on monitor entries, but a common way is to use a mutex or a binary semaphore. Because the compiler, not the programmer, is arranging for the mutual exclusion, it is much less likely that something will go wrong. In any event, the person writing the monitor does not have to be aware of how the compiler arranges for mutual exclusion. It is sufficient to know that by turning all the critical regions into monitor procedures, no two processes will ever execute their critical regions at the same time.

Although monitors provide an easy way to achieve mutual exclusion, as we have seen above, that is not enough. We also need a way for processes to block when they cannot proceed. In the producer-consumer problem, it is easy enough to put all the tests for buffer-full and buffer-empty in monitor procedures, but how should the producer block when it finds the buffer full?

The solution lies in the introduction of **condition variables**, along with two operations on them, wait and signal. When a monitor procedure discovers that it cannot continue (e.g., the producer finds the buffer full), it does a wait on some condition variable, say, *full*. This action causes the calling process to block. It also allows another process that had been previously prohibited from entering the monitor to enter now. We saw condition variables and these operations in the context of Pthreads earlier.

This other process, for example, the consumer, can wake up its sleeping partner by doing a signal on the condition variable that its partner is waiting on. To avoid having two active processes in the monitor at the same time, we need a rule telling what happens after a signal. Hoare proposed letting the newly awakened process run, suspending the other one. Brinch Hansen proposed finessing the problem by requiring that a process doing a signal *must* exit the monitor immediately. In other words, a signal statement may appear only as the final statement in a monitor procedure. We will use Brinch Hansen's proposal because it is conceptually simpler and is also easier to implement. If a signal is done on a condition variable on which several processes are waiting, only one of them, determined by the system scheduler, is revived.

As an aside, there is also a third solution, not proposed by either Hoare or Brinch Hansen. This is to let the signaler continue to run and allow the waiting process to start running only after the signaler has exited the monitor.

Condition variables are not counters. They do not accumulate signals for later use the way semaphores do. Thus, if a condition variable is signaled with no one

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .           .           .
        end;
end monitor;
```

Figure 2-33. A monitor.

waiting on it, the signal is lost forever. In other words, the wait must come before the signal. This rule makes the implementation much simpler. In practice, it is not a problem because it is easy to keep track of the state of each process with variables, if need be. A process that might otherwise do a signal can see that this operation is not necessary by looking at the variables.

A skeleton of the producer-consumer problem with monitors is given in Fig. 2-34 in an imaginary language, Pidgin Pascal. The advantage of using Pidgin Pascal here is that it is pure and simple and follows the Hoare/Brinch Hansen model exactly.

You may be thinking that the operations wait and signal look similar to sleep and wakeup, which we saw earlier had fatal race conditions. Well, they *are* very similar, but with one crucial difference: sleep and wakeup failed because while one process was trying to go to sleep, the other one was trying to wake it up. With monitors, that cannot happen. The automatic mutual exclusion on monitor procedures guarantees that if, say, the producer inside a monitor procedure discovers that the buffer is full, it will be able to complete the wait operation without having to worry about the possibility that the scheduler may switch to the consumer just before the wait completes. The consumer will not even be let into the monitor at all until the wait is finished and the producer has been marked as no longer runnable.

Although Pidgin Pascal is an imaginary language, some real programming languages also support monitors, although not always in the form designed by Hoare and Brinch Hansen. One such language is Java. Java is an object-oriented language that supports user-level threads and also allows methods (procedures) to be grouped together into classes. By adding the keyword synchronized to a method declaration, Java guarantees that once any thread has started executing that method, no other thread will be allowed to start executing any other synchronized method of that object. Without synchronized, there are no guarantees about interleaving.

```
monitor ProducerConsumer
        condition full, empty;
        integer count;

        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;

        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;

        count := 0;
end monitor;

procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;

procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Figure 2-34. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N slots.

A solution to the producer-consumer problem using monitors in Java is given in Fig. 2-35. Our solution has four classes. The outer class, *ProducerConsumer*, creates and starts two threads, p and c. The second and third classes, *producer* and *consumer*, respectively, contain the code for the producer and consumer. Finally, the class *our_monitor*, is the monitor. It contains two synchronized threads that are used for actually inserting items into the shared buffer and taking them out. Unlike the previous examples, here we have the full code of *insert* and *remove*.

```
public class ProducerConsumer {
        static final int N = 100;       // constant giving the buffer size
        static producer p = new producer( );    // instantiate a new producer thread
        static consumer c = new consumer( ); // instantiate a new consumer thread
        static our_monitor mon = new our_monitor( );       // instantiate a new monitor

        public static void main(String args[ ]) {
           p.start( );       // start the producer thread
           c.start( );       // start the consumer thread
        }

        static class producer extends Thread {
           public void run( )  { // run method contains the thread code
              int item;
              while (true) {      // producer loop
                 item = produce_item( );
                 mon.insert(item);
              }
           }
           private int produce_item( ) { ... }       // actually produce
        }

        static class consumer extends Thread {
           public void run( )  { run method contains the thread code
              int item;
              while (true) {      // consumer loop
                 item = mon.remove( );
                 consume_item (item);
              }
           }
           private void consume_item(int item) { ... }// actually consume
        }

        static class our_monitor {   // this is a monitor
           private int buffer[ ] = new int[N];
           private int count = 0, lo = 0, hi = 0;   // counters and indices

           public synchronized void insert(int val) {
              if (count == N) go_to_sleep( );     // if the buffer is full, go to sleep
              buffer [hi] = val; // insert an item into the buffer
              hi = (hi + 1) % N;        // slot to place next item in
              count = count + 1;      // one more item in the buffer now
              if (count == 1) notify( );        // if consumer was sleeping, wake it up
           }

           public synchronized int remove( ) {
              int val;
              if (count == 0) go_to_sleep( );     // if the buffer is empty, go to sleep
              val = buffer [lo]; // fetch an item from the buffer
              lo = (lo + 1) % N;        // slot to fetch next item from
              count = count – 1;      // one few items in the buffer
              if (count == N – 1) notify( ); // if producer was sleeping, wake it up
              return val;
           }
           private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
        }
}
```

Figure 2-35. A solution to the producer-consumer problem in Java.

The producer and consumer threads are functionally identical to their counterparts in all our previous examples. The producer has an infinite loop generating data and putting it into the common buffer. The consumer has an equally infinite loop taking data out of the common buffer and doing some fun thing with it.

The interesting part of this program is the class *our_monitor*, which holds the buffer, the administration variables, and two synchronized methods. When the producer is active inside *insert*, it knows for sure that the consumer cannot be active inside *remove*, making it safe to update the variables and the buffer without fear of race conditions. The variable *count* keeps track of how many items are in the buffer. It can take on any value from 0 through and including $N - 1$. The variable *lo* is the index of the buffer slot where the next item is to be fetched. Similarly, *hi* is the index of the buffer slot where the next item is to be placed. It is permitted that $lo = hi$, which means that either 0 items or N items are in the buffer. The value of *count* tells which case holds.

Synchronized methods in Java differ from classical monitors in an essential way: Java does not have condition variables built in. Instead, it offers two procedures, *wait* and *notify*, which are the equivalent of *sleep* and *wakeup* except that when they are used inside synchronized methods, they are not subject to race conditions. In theory, the method *wait* can be interrupted, which is what the code surrounding it is all about. Java requires that the exception handling be made explicit. For our purposes, just imagine that *go_to_sleep* is the way to go to sleep.

By making the mutual exclusion of critical regions automatic, monitors make parallel programming much less error prone than using semaphores. Nevertheless, they too have some drawbacks. It is not for nothing that our two examples of monitors were in Pidgin Pascal instead of C, as are the other examples in this book. As we said earlier, monitors are a programming-language concept. The compiler must recognize them and arrange for the mutual exclusion somehow or other. C, Pascal, and most other languages do not have monitors, so it is unreasonable to expect their compilers to enforce any mutual exclusion rules. In fact, how could the compiler even know which procedures were in monitors and which were not?

These same languages do not have semaphores either, but adding semaphores is easy: all you need to do is add two short assembly-code routines to the library to issue the `up` and `down` system calls. The compilers do not even have to know that they exist. Of course, the operating systems have to know about the semaphores, but at least if you have a semaphore-based operating system, you can still write the user programs for it in C or C++ (or even assembly language if you are masochistic enough). With monitors, you need a language that has them built in.

Another problem with monitors, and also with semaphores, is that they were designed for solving the mutual exclusion problem on one or more CPUs that all have access to a common memory. By putting the semaphores in the shared memory and protecting them with `TSL` or `XCHG` instructions, we can avoid races. When we move to a distributed system consisting of multiple CPUs, each with its own private memory and connected by a local area network, these primitives become

inapplicable. The conclusion is that semaphores are too low level and monitors are not usable except in a few programming languages. Also, none of the primitives allow information exchange between machines. Something else is needed.

2.3.8 Message Passing

That something else is **message passing**. This method of interprocess communication uses two primitives, send and receive, which, like semaphores and unlike monitors, are system calls rather than language constructs. As such, they can easily be put into library procedures, such as

```
send(destination, &message);
```

and

```
receive(source, &message);
```

The former call sends a message to a given destination and the latter one receives a message from a given source (or from *ANY*, if the receiver does not care). If no message is available, the receiver can block until one arrives. Alternatively, it can return immediately with an error code.

Design Issues for Message-Passing Systems

Message-passing systems have many problems and design issues that do not arise with semaphores or with monitors, especially if the communicating processes are on different machines connected by a network. For example, messages can be lost by the network. To guard against lost messages, the sender and receiver can agree that as soon as a message has been received, the receiver will send back a special **acknowledgement** message. If the sender has not received the acknowledgement within a certain time interval, it retransmits the message.

Now consider what happens if the message is received correctly, but the acknowledgement back to the sender is lost. The sender will retransmit the message, so the receiver will get it twice. It is essential that the receiver be able to distinguish a new message from the retransmission of an old one. Usually, this problem is solved by putting consecutive sequence numbers in each original message. If the receiver gets a message bearing the same sequence number as the previous message, it knows that the message is a duplicate that can be ignored. Successfully communicating in the face of unreliable message passing is a major part of the study of computer networks. For more information, see Tanenbaum and Wetherall (2010).

Message systems also have to deal with the question of how processes are named, so that the process specified in a send or receive call is unambiguous. **Authentication** is also an issue in message systems: how can the client tell that it is communicating with the real file server, and not with an imposter?

At the other end of the spectrum, there are also design issues that are important when the sender and receiver are on the same machine. One of these is performance. Copying messages from one process to another is always slower than doing a semaphore operation or entering a monitor. Much work has gone into making message passing efficient.

The Producer-Consumer Problem with Message Passing

Now let us see how the producer-consumer problem can be solved with message passing and no shared memory. A solution is given in Fig. 2-36. We assume that all messages are the same size and that messages sent but not yet received are buffered automatically by the operating system. In this solution, a total of *N* messages is used, analogous to the *N* slots in a shared-memory buffer. The consumer starts out by sending *N* empty messages to the producer. Whenever the producer has an item to give to the consumer, it takes an empty message and sends back a full one. In this way, the total number of messages in the system remains constant in time, so they can be stored in a given amount of memory known in advance.

If the producer works faster than the consumer, all the messages will end up full, waiting for the consumer; the producer will be blocked, waiting for an empty to come back. If the consumer works faster, then the reverse happens: all the messages will be empties waiting for the producer to fill them up; the consumer will be blocked, waiting for a full message.

Many variants are possible with message passing. For starters, let us look at how messages are addressed. One way is to assign each process a unique address and have messages be addressed to processes. A different way is to invent a new data structure, called a **mailbox**. A mailbox is a place to buffer a certain number of messages, typically specified when the mailbox is created. When mailboxes are used, the address parameters in the send and receive calls are mailboxes, not processes. When a process tries to send to a mailbox that is full, it is suspended until a message is removed from that mailbox, making room for a new one.

For the producer-consumer problem, both the producer and consumer would create mailboxes large enough to hold *N* messages. The producer would send messages containing actual data to the consumer's mailbox, and the consumer would send empty messages to the producer's mailbox. When mailboxes are used, the buffering mechanism is clear: the destination mailbox holds messages that have been sent to the destination process but have not yet been accepted.

The other extreme from having mailboxes is to eliminate all buffering. When this approach is taken, if the send is done before the receive, the sending process is blocked until the receive happens, at which time the message can be copied directly from the sender to the receiver, with no buffering. Similarly, if the receive is done first, the receiver is blocked until a send happens. This strategy is often known as a **rendezvous**. It is easier to implement than a buffered message scheme but is less flexible since the sender and receiver are forced to run in lockstep.

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```

Figure 2-36. The producer-consumer problem with *N* messages.

Message passing is commonly used in parallel programming systems. One well-known message-passing system, for example, is **MPI** (**Message-Passing Interface**). It is widely used for scientific computing. For more information about it, see for example Gropp et al. (1994), and Snir et al. (1996).

2.3.9 Barriers

Our last synchronization mechanism is intended for groups of processes rather than two-process producer-consumer type situations. Some applications are divided into phases and have the rule that no process may proceed into the next phase until all processes are ready to proceed to the next phase. This behavior may be achieved by placing a **barrier** at the end of each phase. When a process reaches the barrier, it is blocked until all processes have reached the barrier. This allows groups of processes to synchronize. Barrier operation is illustrated in Fig. 2-37.

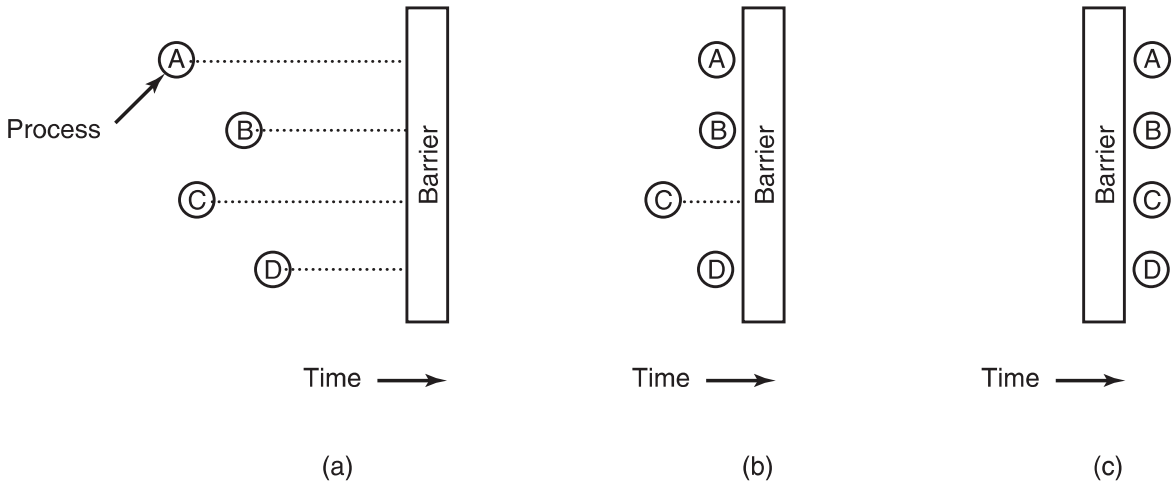


Figure 2-37. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

In Fig. 2-37(a) we see four processes approaching a barrier. What this means is that they are just computing and have not reached the end of the current phase yet. After a while, the first process finishes all the computing required of it during the first phase. It then executes the barrier primitive, generally by calling a library procedure. The process is then suspended. A little later, a second and then a third process finish the first phase and also execute the barrier primitive. This situation is illustrated in Fig. 2-37(b). Finally, when the last process, *C*, hits the barrier, all the processes are released, as shown in Fig. 2-37(c).

As an example of a problem requiring barriers, consider a common relaxation problem in physics or engineering. There is typically a matrix that contains some initial values. The values might represent temperatures at various points on a sheet of metal. The idea might be to calculate how long it takes for the effect of a flame placed at one corner to propagate throughout the sheet.

Starting with the current values, a transformation is applied to the matrix to get the second version of the matrix, for example, by applying the laws of thermodynamics to see what all the temperatures are ΔT later. Then the process is repeated over and over, giving the temperatures at the sample points as a function of time as the sheet heats up. The algorithm produces a sequence of matrices over time, each one for a given point in time.

Now imagine that the matrix is very large (for example, 1 million by 1 million), so that parallel processes are needed (possibly on a multiprocessor) to speed up the calculation. Different processes work on different parts of the matrix, calculating the new matrix elements from the old ones according to the laws of physics. However, no process may start on iteration $n + 1$ until iteration n is complete, that is, until all processes have finished their current work. The way to achieve this goal

is to program each process to execute a barrier operation after it has finished its part of the current iteration. When all of them are done, the new matrix (the input to the next iteration) will be finished, and all processes will be simultaneously released to start the next iteration.

2.5 CLASSICAL IPC PROBLEMS

The operating systems literature is full of interesting problems that have been widely discussed and analyzed using a variety of synchronization methods. In the following sections we will examine three of the better-known problems.

2.5.1 The Dining Philosophers Problem

In 1965, Dijkstra posed and then solved a synchronization problem he called the **dining philosophers problem**. Since that time, everyone inventing yet another synchronization primitive has felt obligated to demonstrate how wonderful the new

primitive is by showing how elegantly it solves the dining philosophers problem. The problem can be stated quite simply as follows. Five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork. The layout of the table is illustrated in Fig. 2-45.

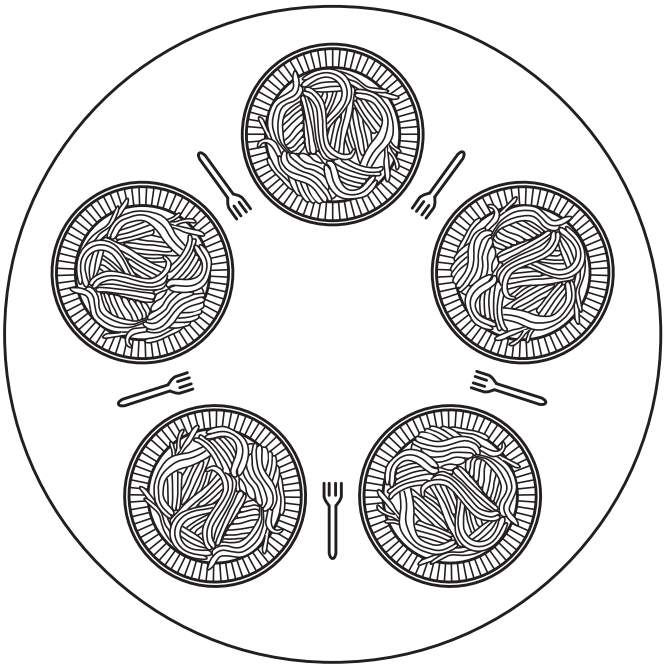


Figure 2-45. Lunch time in the Philosophy Department.

The life of a philosopher consists of alternating periods of eating and thinking. (This is something of an abstraction, even for philosophers, but the other activities are irrelevant here.) When a philosopher gets sufficiently hungry, she tries to acquire her left and right forks, one at a time, in either order. If successful in acquiring two forks, she eats for a while, then puts down the forks, and continues to think. The key question is: Can you write a program for each philosopher that does what it is supposed to do and never gets stuck? (It has been pointed out that the two-fork requirement is somewhat artificial; perhaps we should switch from Italian food to Chinese food, substituting rice for spaghetti and chopsticks for forks.)

Figure 2-46 shows the obvious solution. The procedure *take_fork* waits until the specified fork is available and then seizes it. Unfortunately, the obvious solution is wrong. Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a deadlock.

We could easily modify the program so that after taking the left fork, the program checks to see if the right fork is available. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process. This proposal too, fails, although for a different reason. With a little bit of bad luck, all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks,

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
     while (TRUE) {
          think( );                             /* philosopher is thinking */
          take_fork(i);                         /* take left fork */
          take_fork((i+1) % N);                 /* take right fork; % is modulo operator */
          eat( );                               /* yum-yum, spaghetti */
          put_fork(i);                          /* put left fork back on the table */
          put_fork((i+1) % N);                  /* put right fork back on the table */
     }
}
```

Figure 2-46. A nonsolution to the dining philosophers problem.

waiting, picking up their left forks again simultaneously, and so on, forever. A situation like this, in which all the programs continue to run indefinitely but fail to make any progress, is called **starvation**. (It is called starvation even when the problem does not occur in an Italian or a Chinese restaurant.)

Now you might think that if the philosophers would just wait a random time instead of the same time after failing to acquire the right-hand fork, the chance that everything would continue in lockstep for even an hour is very small. This observation is true, and in nearly all applications trying again later is not a problem. For example, in the popular Ethernet local area network, if two computers send a packet at the same time, each one waits a random time and tries again; in practice this solution works fine. However, in a few applications one would prefer a solution that always works and cannot fail due to an unlikely series of random numbers. Think about safety control in a nuclear power plant.

One improvement to Fig. 2-46 that has no deadlock and no starvation is to protect the five statements following the call to *think* by a binary semaphore. Before starting to acquire forks, a philosopher would do a down on *mutex*. After replacing the forks, she would do an up on *mutex*. From a theoretical viewpoint, this solution is adequate. From a practical one, it has a performance bug: only one philosopher can be eating at any instant. With five forks available, we should be able to allow two philosophers to eat at the same time.

The solution presented in Fig. 2-47 is deadlock-free and allows the maximum parallelism for an arbitrary number of philosophers. It uses an array, *state*, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks). A philosopher may move into eating state only if neither neighbor is eating. Philosopher *i*'s neighbors are defined by the macros *LEFT* and *RIGHT*. In other words, if *i* is 2, *LEFT* is 1 and *RIGHT* is 3.

The program uses an array of semaphores, one per philosopher, so hungry philosophers can block if the needed forks are busy. Note that each process runs the procedure *philosopher* as its main code, but the other procedures, *take_forks*, *put_forks*, and *test*, are ordinary procedures and not separate processes.

```
#define N               5                   /* number of philosophers */
#define LEFT            (i+N–1)%N           /* number of i's left neighbor */
#define RIGHT           (i+1)%N             /* number of i's right neighbor */
#define THINKING        0                   /* philosopher is thinking */
#define HUNGRY          1                   /* philosopher is trying to get forks */
#define EATING          2                   /* philosopher is eating */

typedef int semaphore;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */


void philosopher(int i)                     /* i: philosopher number, from 0 to N–1 */
{
      while (TRUE) {                        /* repeat forever */
            think( );                       /* philosopher is thinking */
            take_forks(i);                  /* acquire two forks or block */
            eat( );                         /* yum-yum, spaghetti */
            put_forks(i);                   /* put both forks back on table */
      }
}

void take_forks(int i)                      /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                         /* enter critical region */
      state[i] = HUNGRY;                    /* record fact that philosopher i is hungry */
      test(i);                              /* try to acquire 2 forks */
      up(&mutex);                           /* exit critical region */
      down(&s[i]);                          /* block if forks were not acquired */
}

void put_forks(i)                           /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                         /* enter critical region */
      state[i] = THINKING;                  /* philosopher has finished eating */
      test(LEFT);                           /* see if left neighbor can now eat */
      test(RIGHT);                          /* see if right neighbor can now eat */
      up(&mutex);                           /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
      if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
            state[i] = EATING;
            up(&s[i]);
      }
}
```

Figure 2-47. A solution to the dining philosophers problem.

2.5.2 The Readers and Writers Problem

The dining philosophers problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices. Another famous problem is the readers and writers problem (Courtois et al., 1971), which models access to a database. Imagine, for example, an airline reservation system, with many competing processes wishing to read and write it. It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other processes may have access to the database, not even readers. The question is how do you program the readers and the writers? One solution is shown in Fig. 2-48.

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
     while (TRUE) {                     /* repeat forever */
          down(&mutex);                 /* get exclusive access to rc */
          rc = rc + 1;                  /* one reader more now */
          if (rc == 1) down(&db);       /* if this is the first reader ... */
          up(&mutex);                   /* release exclusive access to rc */
          read_data_base( );            /* access the data */
          down(&mutex);                 /* get exclusive access to rc */
          rc = rc – 1;                  /* one reader fewer now */
          if (rc == 0) up(&db);         /* if this is the last reader ... */
          up(&mutex);                   /* release exclusive access to rc */
          use_data_read( );             /* noncritical region */
     }
}


void writer(void)
{
     while (TRUE) {                     /* repeat forever */
          think_up_data( );             /* noncritical region */
          down(&db);                    /* get exclusive access */
          write_data_base( );           /* update the data */
          up(&db);                      /* release exclusive access */
     }
}
```

Figure 2-48. A solution to the readers and writers problem.

In this solution, the first reader to get access to the database does a down on the semaphore *db*. Subsequent readers merely increment a counter, *rc*. As readers

leave, they decrement the counter, and the last to leave does an up on the semaphore, allowing a blocked writer, if there is one, to get in.

The solution presented here implicitly contains a subtle decision worth noting. Suppose that while a reader is using the database, another reader comes along. Since having two readers at the same time is not a problem, the second reader is admitted. Additional readers can also be admitted if they come along.

Now suppose a writer shows up. The writer may not be admitted to the database, since writers must have exclusive access, so the writer is suspended. Later, additional readers show up. As long as at least one reader is still active, subsequent readers are admitted. As a consequence of this strategy, as long as there is a steady supply of readers, they will all get in as soon as they arrive. The writer will be kept suspended until no reader is present. If a new reader arrives, say, every 2 sec, and each reader takes 5 sec to do its work, the writer will never get in.

To avoid this situation, the program could be written slightly differently: when a reader arrives and a writer is waiting, the reader is suspended behind the writer instead of being admitted immediately. In this way, a writer has to wait for readers that were active when it arrived to finish but does not have to wait for readers that came along after it. The disadvantage of this solution is that it achieves less concurrency and thus lower performance. Courtois et al. present a solution that gives priority to writers. For details, we refer you to the paper.