**Lecture 1**

**What Is an Operating System?**

A computer system typically consists of software (programs) and hardware (the tangible

machine and its electronic components). The operating system software is the chief

piece of software, the portion of the computing system that manintages all of the hardware

and all of the other software. To be specific, it controls every file, every device,

every section of main memory, and every moment of processing time. It controls who

can use the system and how. In short, the operating system is the boss.

Therefore, each time the user sends a command, the operating system must make sure

that the command is executed, or if it’s not executed, it must arrange for the user to

get a message explaining the error. Remember: this doesn’t necessarily mean that the

operating system executes the command or sends the error message—but it does control

the parts of the system that do.

**Operating System Software**

Operating System Software is the pyramid. At the base of the pyramid of are the four essential managers of every major operating

system: the **Memory Manager, Processor Manager, Device Manager,** and **File Manager.**

These managers and their interactions are discussed in detail in Chapters 1 through 8 of

this book. Each manager works closely with the other managers as each one performs

its unique role. At the top of the pyramid is the User Interface, which allows the user to

issue commands to the operating system. Because this component has specific elements,

in both form and function, it is often very different from one operating system to the

next—sometimes even between different versions of the same operating system.

Operating System Software

Regardless of the size or configuration of the system, the four managers

must, at a minimum, perform the following tasks while collectively keeping

the system working smoothly:

• Monitor the system’s resources

• Enforce the policies that determine what component gets what resources, when, and

how much

• Allocate the resources when appropriate

• Deallocate the resources when appropriate

For example, the Memory Manager must keep track of the status of the computer

system’s main memory space, allocate the correct amount of it to incoming processes,

and deallocate that space when appropriate—all while enforcing the policies that were

established by the designers of the operating system.

An additional management task, networking, was not always an integral part of

operating systems. Today the vast majority of major operating systems incorporate

a **Network Manager** to coordinate the services required for multiple systems to work

cohesively together. For example, the Network Manager must coordinate the workings

of the networked resources, which might include shared access to memory space,

processors, printers, databases, monitors, applications, and more. This can be a complex

balancing act as the number of resources increases, as it often does.

**Main Memory Management**

The Memory Manager is in charge of main memory, widely known as **RAM**

(short for random access memory). The Memory Manager

checks the validity of each request for memory space, and if it is a legal request, allocates

a portion of memory that isn’t already in use. If the m emory space becomes fragmented,

this manager might use policies established by the operating systems designers

to reallocate memory to make more useable space available for other jobs that are

waiting. Finally, when the job or process is finished, the Memory Manager deallocates

its allotted memory space.

A key feature of RAM chips—the hardware that comprises computer memory—is that

they depend on the constant flow of electricity to hold data. When the power fails or is

turned o ff, the contents of RAM is wiped clean. This is one reason why computer system

designers attempt to build elegant shutdown procedures, so the contents of RAM

can be stored on a nonvolatile device, such as a hard drive, before the main memory

chips lose power during computer shutdown.

A critical responsibility of the Memory Manager is to protect all of the space in main

memory, particularly that occupied by the operating system itself—it can’t allow any

part of the operating system to be accidentally or intentionally altered because that

would lead to instability or a system crash.

Another kind of memory that’s critical when the computer is powered on is **Read-**

**Only Memory** (often shortened to **ROM**). This ROM chip holds

software called **firmware**, the programming code that is used to start the computer and

perform other necessary tasks. To put it in simplest form, it describes in prescribed

steps when and how to load each piece of the operating system after the power is

turned on and until the computer is ready for use. The contents of the ROM chip are

nonvolatile, meaning that they are not erased when the power is turned off, unlike the

contents of RAM.

Processor Management

Device Management

The Device Manager (the subject of Chapter 7) is responsible for connecting with

every device that’s available on the system and for choosing the most efficient way to

allocate each of these printers, ports, disk drives, and more, based on the device scheduling

policies selected by the designers of the operating system.

Good device management requires that this part of the operating system uniquely identify

each device, start its operation when appropriate, monitor its progress, and finally

deallocate the device to make the operating system available to the next waiting process.

This isn’t as easy as it sounds because of the exceptionally wide range of devices

resources require management of a vast range of alternative elements, which enormously

complicates the tasks required to add network management capabilities.

Networks can range from a small wireless system that connects a game system to

the Internet, to a private network for a small business, to one that connects multiple

computer systems, devices, and mobile phones to the Internet. Regardless of the size

and complexity of the network, these operating systems must be prepared to properly

manage the available memory, CPUs, devices, and files.

User Interface

The user interface—the portion of the operating system that users interact with

directly—is one of the most unique components of an operating system. Two primary

types are the **graphical user interface** (GUI) shown in Figure 1.4 and the **command line**

**interface.** The GUI relies on input from a pointing device such as a mouse or your finger.

Specific menu options, desktops, and formats often vary widely from one operating

system to another (and sometimes from one version to another).

The alternative to a GUI is a command line interface, which responds to specific commands

typed on a keyboard and displayed on the monitor, as shown in Figure 1.5.

These interfaces accept typed commands and offer skilled users powerful additional

control because typically the commands can be linked together (concatenated) to perform

complex tasks with a single multifunctional command that would require many

mouse clicks to duplicate using a graphical interface.

While a command structure offers powerful functionality, it has strict requirements for

every command: each must be typed accurately, each must be formed in correct syntax,

and combinations of commands must be assembled correctly. In addition, users need to

know how to recover gracefully from any errors they encounter. These command line

interfaces were once standard for operating systems and are still favored by power users

but have largely been supplemented with simple, for giving graphical user interfaces.

Cooperation Issues

None of the elements of an operating system can perform its individual tasks in

isolation—each must also work harmoniously with every other manager. To illustrate

this using a very simplified example, let’s follow the steps as someone chooses a menu

option to open a program. The following series of major steps are typical of the

discrete actions that would occur in fractions of a second as a result:

1. The Device Manager receives the electrical impulse caused by a click of the

mouse, decodes the command by calculating the location of the cursor, and

sends that information through the User Interface, which identifies the requested

command. Immediately, it sends the command to the Processor Manager.

2. The Processor Manager then sends an acknowledgment message (such as

“waiting” or “loading”) to be displayed on the monitor so the user knows that

the command has been sent successfully.

3. The Processor Manager determines whether the user request requires that a file

(in this case a program file) be retrieved from storage or whether it is already in

memory.

4. If the program is in secondary storage (perhaps on a disk), the File Manager calculates

its exact location on the disk and passes this information to the Device

Manager, which retrieves the program and sends it to the Memory Manager.

5. If necessary, the Memory Manager finds space for the program file in main

memory and records its exact location. Once the program file is in memory,

this manager keeps track of its location in memory.

6. When the CPU is ready to run it, the program begins execution by the

Processor Manager. When the program has finished executing, the Processor

Manager relays this information to the other managers.

7. The Processor Manager reassigns the CPU to the next program waiting in line.

the central processing unit (CPU); an important function of the Processor Manager

i resources require management of a vast range of alternative elements, which enormously

complicates the tasks required to add network management capabilities.

Networks can range from a small wireless system that connects a game system to

the Internet, to a private network for a small business, to one that connects multiple

computer systems, devices, and mobile phones to the Internet. Regardless of the size

and complexity of the network, these operating systems must be prepared to properly

manage the available memory, CPUs, devices, and files.

User Interface

The user interface—the portion of the operating system that users interact with

directly—is one of the most unique components of an operating system. Two primary

types are the **graphical user interface** (GUI) shown in Figure 1.4 and the **command line**

**interface.** The GUI relies on input from a pointing device such as a mouse or your finger.

Specific menu options, desktops, and formats often vary widely from one operating

system to another (and sometimes from one version to another).

The alternative to a GUI is a command line interface, which responds to specific commands

typed on a keyboard and displayed on the monitor, as shown in Figure 1.5.

These interfaces accept typed commands and offer skilled users powerful additional

control because typically the commands can be linked together (concatenated) to perform

complex tasks with a single multifunctional command that would require many

mouse clicks to duplicate using a graphical interface.

While a command structure offers powerful functionality, it has strict requirements for

every command: each must be typed accurately, each must be formed in correct syntax,

and combinations of commands must be assembled correctly. In addition, users need to

know how to recover gracefully from any errors they encounter. These command line

interfaces were once standard for operating systems and are still favored by power users

but have largely been supplemented with simple, forgiving graphical user interfaces.

s resources require management of a vast range of alternative elements, which enormously

complicates the tasks required to add network management capabilities.

Networks can range from a small wireless system that connects a game system to

the Internet, to a private network for a small business, to one that connects multiple

computer systems, devices, and mobile phones to the Internet. Regardless of the size

and complexity of the network, these operating systems must be prepared to properly

manage the available memory, CPUs, devices, and files.

User Interface

The user interface—the portion of the operating system that users interact with

directly—is one of the most unique components of an operating system. Two primary

types are the **graphical user interface** (GUI) shown in Figure 1.4 and the **command line**

**interface.** The GUI relies on input from a pointing device such as a mouse or your finger.

Specific menu options, desktops, and formats often vary widely from one operating

system to another (and sometimes from one version to another).

The alternative to a GUI is a command line interface, which responds to specific commands

typed on a keyboard and displayed on the monitor.

These interfaces accept typed commands and offer skilled users powerful additional

control because typically the commands can be linked together (concatenated) to perform

complex tasks with a single multifunctional command that would require many

mouse clicks to duplicate using a graphical interface.

While a command structure offers powerful functionality, it has strict requirements for

every command: each must be typed accurately, each must be formed in correct syntax,

and combinations of commands must be assembled correctly. In addition, users need to

know how to recover gracefully from any errors they encounter. These command line

interfaces were once standard for operating systems and are still favored by power users

but have largely been supplemented with simple, forgiving graphical user interfaces. to keep track of the status of each job, process, thread, and so on. We will discuss

all of these in the chapters that follow, but for this overview, let’s limit our discussion

to **p Types of Operating Systems**

Operating systems fall into several general categories distinguished by the speed of

their response and the method used to enter data into the system. The five categories

are batch, interactive, real-time, hybrid, and embedded systems.

**Batch systems feature jobs that are entered as a whole and in sequence. That is, only**

**one job can be entered at a time, and once a job begins processing, then no other job**

**can start processing until the resident job is finished.** These systems date from early

computers, when each job consisted of a stack of cards—or reels of magnetic tape—

for input and were entered into the system as a unit, called a batch. The efficiency of

a batch system is measured in **throughput**, which is the number of jobs completed in a

given amount of time (usually measured in minutes, hours, or days.)

**Interactive systems allow multiple jobs to begin processing and return results to**

**users with better response times than batch systems, but interactive systems are**

**slower than the real-time systems we talk about next.** Early versions of these operating

systems allowed each user to interact directly with the computer system via

commands entered from a typewriter-like terminal, and the operating system used

complex algorithms to share processing power (often with a single processor)

among the jobs awaiting processing. Interactive systems offered huge improvements

in response over batch-only systems with **turnaround times** in seconds or minutes

instead of hours or days.

**Real-time systems are used in time-critical environments where reliability is critical**

**and data must be processed within a strict time limit.** This time limit need not

be ultra-fast (though it often is), but system response time must meet the deadline

because there are significant consequences of not doing so. They also need to provide

contingencies to fail gracefully—that is, preserve as much of the system’s capabilities

and data as possible to facilitate recovery. Examples of real-time systems

are those used for spacecraft, airport traffic control, fly-by-wire aircraft, critical

industrial processes, and medical systems, to name a few. There are two types of

real-time systems depending on the consequences of missing the deadline: hard and

soft systems.

• Hard real-time systems risk total system failure if the predicted time deadline is missed.

• Soft real-time systems suffer performance degradation, but not total system failure,

as a consequence of a missed deadline.

Although it’s theoretically possible to convert a general-purpose operating system

into a real-time system by merely establishing a deadline, the need to be extremely n

in predictable is not part of the design criteria for most operating systems so they can’t

provide the guaranteed response times that real-time performance requires. Therefore,

most embedded systems (described below) and real-time environments require operating

systems that are specially designed to meet real-time needs.

**Hybrid systems, widely used today, are a combination of batch and interactive. They**

**appear to be interactive because individual users can enter multiple jobs or processes**

**into the system and get fast responses, but these systems actually accept and run**

**batch programs in the background when the interactive load is light. A hybrid system**

**takes advantage of the free time between high-demand usage of the system and lowdemand**

**times.**

**Networks allow users to manipulate resources that may be located over a wide geographical**

**area.** Network operating systems were originally similar to single-processor

operating systems in that each machine ran its own local operating system and served

its own local user group. Now network operating systems make up a special class of

software that allows users to perform their tasks using few, if any, local resources.

One example of this phenomenon is cloud computing.

As shown in Figure 1.7, wireless networking capability is a standard feature in many

computing devices: cell phones, tablets, and other handheld Web browsers.

**Embedded systems are computers that are physically placed inside the products that**

**they operate to add very specific features and capabilities.** For example, embedded systems

can be found in automobiles, digital music players, elevators, and pacemakers, to name a few.

Computers embedded in automobiles facilitate engine performance, braking,

navigation, entertainment systems, and engine diagnostic details.

Operating systems for embedded computers are very different from those for general

computer systems. Each one is designed to perform a set of specific programs, which

are not interchangeable among systems. This permits the designers to make the operating

system more efficient and take advantage of the computer’s limited resources (typically

slower CPUs and smaller memory resources), to their maximum.

Before a general-purpose operating system, such as Linux, UNIX, or Windows, can be

used in an embedded system, the system designers must select which operating system

components are required in that particular environment and which are not. The final

version of this operating system generally includes redundant safety features and only

the necessary elements; any unneeded features or functions are dropped. Therefore,

operating systems with a small kernel (the core portion of the software) and other

functions that can be mixed and matched to meet the embedded system requirements

have potential in this market.

eq**Brief History of Operating Systems Development**

The evolution of operating system software parallels the evolution of the computer

hardware it was designed to control.

1940s

Computers from this time were operated by the programmers presiding from the main

console—this was a hands-on process. In fact, to fix an error in a program, the programmer

would stop the processor, read the contents of each register, make the corrections

in memory, and then resume operation. To run programs on these systems,

the programmers would reserve the entire machine for the entire time they estimated

it would take for the computer to execute their program, and the computer sat idle

between reservations. As a result, the machine was poorly utilized because the processor,

the CPU, was active for only a fraction of the time it was reserved and didn’t work

at all between reservations.

There were a lot of variables that could go wrong with these early computers. For

example, when Harvard’s Mark I computer stopped working one day in 1945, technicians

investigating the cause for the interruption discovered that a moth had short-circuited

Relay 70 in Panel F, giving its life in the process. The researcher, Grace Murray

Hopper, duly placed the dead insect in the system log as shown in Figure 1.8 noting,

“First actual case of bug being found.” The incident spawned the industry-wide use of

the word “bug” to indicate that a system is not working correctly, and the term is still

commonly used today.

1950s

Two improvements were widely adopted during this decade: the task of running programs

was assigned to professional computer operators (instead of individual programmers)

who were assigned to maximize the computer’s operation and schedule the

incoming jobs as efficiently as possible. Job scheduling required that waiting jobs be

put into groups with similar requirements (for example, by grouping all jobs running

with a certain language compiler) so the entire batch of jobs could run faster. But even

with these batching techniques, there were still extensive time lags between the CPUs

that were fast and the I/O devices that ran much slower. Eventually, several factors

helped improve the performance of the CPU and the system.

• The speeds of I/O devices, such as tape drives and disks, gradually increased.

• To use more of the available storage area in these devices, records were grouped

into blocks before they were retrieved or stored. (This is called “blocking,” meaning

that several logical records are grouped within one physical record.

u• To reduce the discrepancy in speed between the I/O and the CPU, an interface called

the control unit was placed between them to act as a buffer. A buffer is an interim storage

area that works as a temporary holding place. As the slow input device reads one

record, the control unit places each character of the record into the buffer. When the

buffer is full, the entire record is quickly transmitted to the CPU. The process is just the

opposite for output devices: the CPU places the entire record into the buffer, which is

then passed on by the control unit at the slower rate required by the output device.

The buffers of this generation were conceptually similar to those now used routinely

by Web browsers to make video and audio playback smoother.at

During the second generation, programs were still assigned to the processor one-ata-

time in sequence. The next step toward better use of the system’s resources was the

move to shared processing.

1960s

Computers in the mid-1960s were designed with faster CPUs, but they still had problems

interacting directly with the relatively slow printers and other I/O devices. The

solution was called multiprogramming, which introduced the concept of loading many

programs at one time and allowing them to share the attention of the single CPU.

The most common mechanism for implementing multiprogramming was the introduction

of the concept of the interrupt, whereby the CPU was notified of events needing

operating systems services. For example, when a program issued a print command,

called input/output (I/O) command, it generated an interrupt, which signaled the

release of the CPU from one job so it could begin execution of the next job. This

was called *passive multiprogramming* because the operating system didn’t control the

interrupts, but instead, it waited for each job to end on its own. This was less than

ideal because if a job was CPU-bound (meaning that it performed a great deal of nonstop

CPU processing before issuing an interrupt), it could monopolize the CPU for a

long time while all other jobs waited, even if they were more important.

To counteract this effect, computer scientists designed *active multiprogramming,*

allowing the operating system a more active role. Each program was initially allowed

to use only a preset slice of CPU time. When time expired, the job was interrupted by

the operating system so another job could begin its execution. The interrupted job

then had to wait until it was allowed to resume execution at some later time. Soon,

this idea, called time slicing, became common in many interactive systems.

1970s

During this decade, computers were built with faster CPUs, creating an even greater

disparity between their rapid processing speed and slower I/O times. However, multiprogramming

schemes to increase CPU use were limited by the physical capacity of

the main memory, which was a limited resource and very expensive. For example, the

first Cray supercomputer, shown in Figure 1.10, was installed at Los Alamos National

Laboratory in 1976 and had only 8 megabytes (MB) of main memory, significantly

less than many computing devices today.

A solution to this physical limitation was the development of virtual memory, which

allowed portions of multiple programs to reside in memory at the same time. In other

words, a virtual memory system could divide each program into parts and keep those

parts in secondary storage, bringing each part into memory only as it was needed.

oVirtual memory soon became standard in operating systems of all sizes and paved the

way to much better use of the CPU.

1980s

Hardware during this time became more flexible, with logical functions that were

built on easily replaceable circuit boards. And because it had become cheaper to create

these circuit boards, more operating system functions were made part of the hardware

itself, giving rise to a new concept—**firmware**, a word used to indicate that a program

is permanently held in read only memory (ROM), as opposed to being held in

secondary storage.

Eventually the industry moved to multiprocessing (having more than one processor),

and more complex languages were designed to coordinate the activities of the multiple

processors servicing a single job. As a result, it became possible to execute two programs

at the same time (in parallel), and eventually operating systems for computers of

every size were routinely expected to accommodate multiprocessing.

f seThe evolution of personal computers and high-speed communications sparked the

move to networked systems and distributed processing, enabling users in remote locations

to share hardware and software resources. These systems required a new kind of

operating system—one capable of managing multiple sets of subsystem managers, as

well as hardware that might reside half a world away.

On the other hand, with distributed operating systems, users could think they were

working with a system using one processor, when in fact they were connected to

a cluster of many processors working closely together. With these systems, users

didn’t need to know which processor was running their applications or which

devices were storing their files. These details were all handled transparently by

the operating system—something that required more than just adding a few lines

of code to a uniprocessor operating system. The disadvantage of such a complex

operating system was the requirement for more complex processor-scheduling

algorithms.

1990s

The overwhelming demand for Internet capability in the mid-1990s sparked the

proliferation of networking capability. The World Wide Web was first described in

a paper by Tim Berners-Lee; his original concept is shown in Figure 1.11. Based on

this research, he designed the first Web server and browser, making it available to the

general public in 1991. While his innovations sparked increased connectivity, it also

increased demand for tighter security to protect system assets from Internet threats.

The decade also introduced a proliferation of multimedia applications demanding

additional power, flexibility, and device compatibility for most operating systems,

as well as large amounts of storage capability (in addition to longer battery life and

cooler operation). These technological advances required commensurate advances by

the operating system.

2000s

The new century emphasized the need for improved flexibility, reliability, and speed.

The concept of virtual machines was expanded to allow computers to accommodate

multiple operating systems that ran at the same time and shared resources. One example

is shown in Figure 1.12.

**Virtualization** allowed separate partitions of a single server to support a different operating

system. In other words, it turned a single physical server into multiple virtual

servers, often with multiple operating systems. Virtualization required the operating

system to have an intermediate manager, to oversee the access of each operating system

to the server’s physical resources.veral

prProcessing speed enjoyed a similar advancement with the commercialization of multicore

processors, which can contain from two to many cores. For example, a chip with

two CPUs (sometimes called a dual-core chip) allows two sets of calculations to run

at the same time, which sometimes leads to faster job completion. It’s almost as if the

user has two separate computers, and thus two processors, cooperating on a single

task. Designers have created chips that have dozens of cores, as shown in Figure 1.13.

Does this hardware innovation affect the operating system software? Absolutely—

because the operating system must now manage the work of each of these processors

and be able to schedule and manage the processing of their multiple tasks.

2010s

Increased mobility and wireless connectivity spawned a proliferation of dual-core,

quad-core, and other multicore CPUs with more than one processor (also called a

core) on a computer chip. Multicore engineering was driven by the problems caused by

nano-sized transistors and their ultra-close placement on a computer chip. Although

chips with millions of transistors that were very close together helped increase system

performance dramatically, the close proximity of these transistors also allowed current

to “leak,” which caused the buildup of heat, as well as other issues. With the development

of multi-core technology, a single chip (one piece of silicon) was equipped with

two or more processor cores. In other words, they replaced a single large processor

with two half-sized processors (called dual core), four quarter-sized processors (quad

core), and so on. This design allowed the same sized chip to produce less heat and

offered the opportunity to permit multiple calculations to take place at the same time.

**Design Considerations**

The people who write operating systems are faced with many choices that can affect

every part of the software and the resources it controls. Before beginning, designers

typically start by asking key questions, using the answers to guide them in their work.

The most common overall goal is to maximize use of the system’s resources (memory,

processing, devices, and files) and minimize downtime, though certain proprietary systems

may have other priorities. Typically, designers include the following factors into

their developmental efforts: the minimum and maximum RAM resources, the number

and type of CPUs available, the variety of devices likely to be connected, the range

of files, networking capability, security requirements, default user interfaces available,

assumed user capabilities, and so on.

For example, a mobile operating system for a tablet might have a single CPU and

need to manage that CPU to minimize the heat it generates. Likewise, if the operating

system manages a real-time system, where deadlines are urgent, designers need

to manage the memory, processor time, devices, and files so urgent deadlines will be

met. For these reasons, operating systems are often complex pieces of software that

juggle numerous applications, access to networked resources, several users, and multiple

CPUs in an effort to keep the system running effectively.

As you might expect, no single operating system is perfect for every environment. Some

systems can be best served with a UNIX system, others benefit from the structure of

a Windows system, and still others work best using Linux, Mac OS, or Android, or

even a custom-built operating system.ocesses

, w**Key Terms**

**batch system:** a type of computing system that executes programs, each of which is

submitted in its entirety, can be grouped into batches, and executed without external

intervention.

**central processing unit (CPU):** a component with circuitry to control the interpretation

and execution of instructions.

**cloud computing:** a multifaceted technology that allows computing, data storage and

retrieval and other computer functions to take place over a large network, typically the

Internet.

**Device Manager:** the section of the operating system responsible for controlling the

use of devices. It monitors every device, channel, and control unit and chooses the

most efficient way to allocate all of the system’s devices.

**embedded computer system:** a dedicated computer system that often is part of a

larger physical system, such as jet aircraft or automobiles. Often, it must be small,

fast, and able to work with real-time constraints, fail-safe execution, and nonstandard

I/O devices.

**File Manager:** the section of the operating system responsible for controlling the use

of files.

**firmware:** software instructions or data that are stored in a fixed or “firm” way, usually

implemented on some type of read only memory (ROM).

**hardware:** the tangible machine and its components, including main memory, I/O

devices, I/O channels, direct access storage devices, and the central processing unit.

**hybrid system:** a computer system that supports both batch and interactive processes.

**interactive system:** a system that allows each user to interact directly with the operating

system.

**kernel:** the primary part of the operating system that remains in random access memory

(RAM) and is charged with performing the system’s most essential tasks, such as

managing main memory and disk access.

**main memory:** the memory unit that works directly with the CPU and in which the

data and instructions must reside in order to be processed. Also called *primary storage,*

*RAM,* or *internal memory.*

e**mainframe:** the historical name given to a large computer system characterized by its

large size, high cost, and relatively fast performance.

**Memory Manager:** the section of the operating system responsible for controlling the

use of memory. It checks the validity of each request for memory space, and if it’s a

legal request, allocates the amount needed to execute the job.

**multiprocessing:** when two or more CPUs share the same main memory, most I/O

devices, and the same control program routines. They service the same job stream and

execute distinct processing programs concurrently.

**multiprogramming:** a technique that allows a single processor to process several programs

residing simultaneously in main memory and interleaving their execution by

overlapping I/O requests with CPU requests.

**network:** a system of interconnected computer systems and peripheral devices that

exchange information with one another.

**operating system:** the primary software on a computing system that manages its

resources, controls the execution of other programs, and manages communications

and data storage.

**Processor Manager:** a composite of two submanagers, the Job Scheduler and the

Process Scheduler, which decides how to allocate the CPU.

**RAM:** random access memory. See *main memory.*

**real-time system:** a computing system used in time-critical environments that require

guaranteed response times, such as navigation systems, rapid transit systems, and

industrial control systems.

**server:** a node that provides to clients various network services, such as file retrieval,

printing, or database access services.

**storage:** the place where data is stored in the computer system. Primary storage is main

memory. Secondary storage is nonvolatile media, such as disks and flash memory.

**Lecture 2**

Most operating systems provide certain basic concepts and abstractions such as

processes, address spaces, and files that are central to understanding them. In the

following sections, we will look at some of these basic concepts ever so briefly, as

an introduction. We will come back to each of them in great detail later in this

book. To illustrate these concepts we will, from time to time, use examples, generally

drawn from UNIX. Similar examples typically exist in other systems as well,

however, and we will study some of them later.

**Processes**

**A key concept in all operating systems is the process. A process is basically a**

**program in execution. Associated with each process is its address space, a list of**

**memory locations from 0 to some maximum, which the process can read and write.**

The address space contains the executable program, the program’s data, and its

stack. Also associated with each process is a set of resources, commonly including

registers (including the program counter and stack pointer), a list of open files, outstanding

alarms, lists of related processes, and all the other information needed to

run the program. A process is fundamentally a container that holds all the information

needed to run a program.

We will come back to the process concept in much more detail in Chap. 2. For

the time being, the easiest way to get a good intuitive feel for a process is to think

about a multiprogramming system. The user may have started a video editing program

and instructed it to convert a one-hour video to a certain format (something

that can take hours) and then gone off to surf the Web. Meanwhile, a background

process that wakes up periodically to check for incoming email may have started

running. Thus we have (at least) three active processes: the video editor, the Web

browser, and the email receiver. Periodically, the operating system decides to stop

running one process and start running another, perhaps because the first one has

used up more than its share of CPU time in the past second or two.

When a process is suspended temporarily like this, it must later be restarted in

exactly the same state it had when it was stopped. This means that all information

about the process must be explicitly saved somewhere during the suspension. For

example, the process may have sev eral files open for reading at once. Associated

with each of these files is a pointer giving the current position (i.e., the number of

the byte or record to be read next). When a process is temporarily suspended, all

these pointers must be saved so that a read call executed after the process is restarted

will read the proper data. In many operating systems, all the information about

each process, other than the contents of its own address space, is stored in an operating

system table called the **process table**, which is an array of structures, one for

each process currently in existence.

Thus, a (suspended) process consists of its address space, usually called the

**core image** (in honor of the magnetic core memories used in days of yore), and its

process table entry, which contains the contents of its registers and many other

items needed to restart the process later.

The key process-management system calls are those dealing with the creation

and termination of processes. Consider a typical example. A process called the

**command interpreter** or shell reads commands from a terminal. The user has just

typed a command requesting that a program be compiled. The shell must now create

a new process that will run the compiler. When that process has finished the

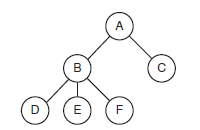
compilation, it executes a system call to terminate itself.

If a process can create one or more other processes (referred to as **child processes**)

and these processes in turn can create child processes, we quickly arrive at

the process tree structure of Fig. Related processes that are cooperating to

get some job

**

A process tree. Process *A* created two child processes, *B* and *C*.

Process *B* created three child processes, *D*, *E*, and *F.*

Other process system calls are available to request more memory (or release

unused memory), wait for a child process to terminate, and overlay its program

with a different one.

Occasionally, there is a need to convey information to a running process that is

not sitting around waiting for this information. For example, a process that is communicating

with another process on a different computer does so by sending messages

to the remote process over a computer network. To guard against the possibility

that a message or its reply is lost, the sender may request that its own operating

system notify it after a specified number of seconds, so that it can retransmit

the message if no acknowledgement has been received yet. After setting this timer,

the program may continue doing other work.

When the specified number of seconds has elapsed, the operating system sends

an **alarm signal** to the process. The signal causes the process to temporarily suspend

whatever it was doing, save its registers on the stack, and start running a special

signal-handling procedure, for example, to retransmit a presumably lost message.

When the signal handler is done, the running process is restarted in the state

it was in just before the signal. Signals are the software analog of hardware interrupts

and can be generated by a variety of causes in addition to timers expiring.

Many traps detected by hardware, such as executing an illegal instruction or using

an invalid address, are also converted into signals to the guilty process.

Each person authorized to use a system is assigned a **UID** (**User IDentification**)

by the system administrator. Every process started has the UID of the person

who started it. A child process has the same UID as its parent. Users can be members

of groups, each of which has a **GID** (**Group IDentification**).

One UID, called the **superuser** (in UNIX), or **Administrator** (in Windows),

has special power and may override many of the protection rules. In large installations,

only the system administrator knows the password needed to become

superuser, but many of the ordinary users (especially students) devote considerable

effort seeking flaws in the system that allow them to become superuser without the

password.

Every computer has some main memory that it uses to hold executing programs.

In a very simple operating system, only one program at a time is in memory.

To run a second program, the first one has to be removed and the second one

placed in memory.

More sophisticated operating systems allow multiple programs to be in memory

at the same time. To keep them from interfering with one another (and with the

operating system), some kind of protection mechanism is needed. While this mechanism

has to be in the hardware, it is controlled by the operating system.

The above viewpoint is concerned with managing and protecting the computer’s

main memory. A different, but equally important, memory-related issue is

managing the address space of the processes. Normally, each process has some set

of addresses it can use, typically running from 0 up to some maximum. In the simplest

case, the maximum amount of address space a process has is less than the

main memory. In this way, a process can fill up its address space and there will be

enough room in main memory to hold it all.

However, on many computers addresses are 32 or 64 bits, giving an address

space of 232 or 264 bytes, respectively. What happens if a process has more address

space than the computer has main memory and the process wants to use it all? In

the first computers, such a process was just out of luck. Nowadays, a technique called

virtual memory exists, as mentioned earlier, in which the operating system

keeps part of the address space in main memory and part on disk and shuttles

pieces back and forth between them as needed. In essence, the operating system

creates the abstraction of an address space as the set of addresses a process may

reference. The address space is decoupled from the machine’s physical memory

and may be either larger or smaller than the physical memory.

Another key concept supported by virtually all operating systems is the file

system. As noted before, a major function of the operating system is to hide the

peculiarities of the disks and other I/O devices and present the programmer with a

nice, clean abstract model of device-independent files. System calls are obviously

needed to create files, remove files, read files, and write files. Before a file can be

read, it must be located on the disk and opened, and after being read it should be

closed, so calls are provided to do these things.

To provide a place to keep files, most PC operating systems have the concept

of a **directory** as a way of grouping files together. A student, for example, might

have one directory for each course he is taking (for the programs needed for that

course), another directory for his electronic mail, and still another directory for his

World Wide Web home page. System calls are then needed to create and remove

directories. Calls are also provided to put an existing file in a directory and to remove

a file from a directory. Directory entries may be either files or other directories.

This model also gives rise to a hierarchy—the file system.

**Virtual memory became possible with the capability of moving pages at will between**

**main memory and secondary storage, and it effectively removed restrictions on maximum**

**program size. With virtual memory, even though only a portion of each program**

**is stored in memory at any given moment, by swapping pages into and out of memory,**

**it gives users the appearance that their programs are completely loaded into main**

**memory during their entire processing time—a feat that would require an incredibly**

**large amount of main memory.**

Virtual memory can be implemented with both paging and segmentation.

Segmentation allows users to share program code. The shared segment contains: (1) an

area where unchangeable code (called **reentrant code**) is stored, and (2) several data

areas, one for each user. In this case, users share the code, which cannot be modified,

but they can modify the information stored in their own data areas as needed without

affecting the data stored in other users’ data areas.

Before virtual memory, sharing meant that copies of files were stored in each user’s

account. This allowed them to load their own copy and work on it at any time.

This kind of sharing created a great deal of unnecessary system cost—the I/O overhead

in loading the copies and the extra secondary storage needed. With virtual

memory, those costs are substantially reduced because shared programs and subroutines

are loaded on demand, satisfactorily reducing the storage requirements

of main memory (although this is accomplished at the expense of the Memory

Map Table).

The use of virtual memory requires cooperation between the Memory Manager (which

tracks each page or segment) and the processor hardware (which issues the interrupt

and resolves the virtual address). For example, when a page that is not already in

memory is needed, a page fault is issued and the Memory Manager chooses a page

frame, loads the page, and updates entries in the Memory Map Table and the Page

Map Tables.

Virtual memory works well in a multiprogramming environment because most programs

spend a lot of time waiting—they wait for I/O to be performed; they wait for

pages to be swapped in or out; and they wait when their turn to use the processor is

expired. In a multiprogramming environment, the waiting time isn’t wasted because

the CPU simply moves to another job.

Virtual memory has increased the use of several programming techniques. For instance,

it aids the development of large software systems, because individual pieces can be

developed independently and linked later on.

Virtual memory management has several advantages:

• A job’s size is no longer restricted to the size of main memory (or worse, to the free

space available within main memory).

• Memory is used more efficiently because the only sections of a job stored in

memory are those needed immediately, while those not needed remain in secondary

storage.

• It allows an unlimited amount of multiprogramming, which can apply to many

jobs, as in dynamic and static partitioning, or to many users.

• It allows the sharing of code and data.

• It facilitates dynamic linking of program segments.

The advantages far outweigh these disadvantages:

• Increased processor hardware costs.

• Increased overhead for handling paging interrupts.

• Increased software complexity to prevent thrashing.

**Cache Memory**

**Cache memory** is based on the concept of using a small, fast, and expensive memory to

supplement the workings of main memory. Because the cache is usually small in capacity

(compared to main memory), it can use more expensive memory chips. These are

five to ten times faster than main memory and match the speed of the CPU. Therefore,

if data or instructions that are frequently used are stored in cache memory, memory

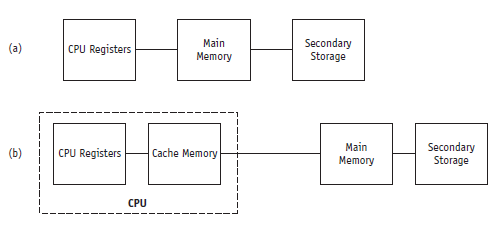
access time can be cut down significantly and the CPU can execute those instructions

faster, thus raising the overall performance of the computer system. (It’s similar to the

role of a frequently called list of telephone numbers in a telephone. By keeping those

numbers in an easy-to-reach place, they can be called much faster than those in a long

contact list.)



**(figure )**

*Comparison of (a) the traditional path used by early computers between main memory and the CPU and (b) the path used*

*by modern computers to connect the main memory and the CPU via cache memory*

As shown in Figure (a), early computers were designed to have data and

instructions transferred from secondary storage to main memory and then to specialpurpose

registers for processing—this path necessarily increased the amount of time

needed to process those instructions. However, because the same instructions are used

repeatedly in most programs, computer system designers thought it would be more

efficient if the system would not use a complete memory cycle every time an instruction

or data value is required. Designers found that this could be done if they placed

repeatedly used data in general-purpose registers instead of in main memory.

To solve this problem, computer systems automatically store frequently used data in

an intermediate memory unit called **cache memory**. This adds a middle layer to the

original hierarchy. Cache memory can be thought of as an intermediary between main

memory and the special-purpose registers, which are the domain of the CPU, as shown

in Figure (b).

A typical microprocessor has two or more levels of caches, such as Level 1 (L1),

Level 2 (L2), and Level 3 (L3), as well as specialized caches.

In a simple configuration with only two cache levels, information enters the processor

through the bus interface unit, which immediately sends one copy to the Level 2 cache,

which is an integral part of the microprocessor and is directly connected to the CPU.

A second copy is sent to one of two Level 1 caches, which are built directly into the

chip. One of these Level 1 caches stores instructions, while the other stores data to be

used by the instructions. If an instruction needs additional data, the instruction is put

on hold while the processor looks for the missing data—first in the data Level 1 cache,

and then in the larger Level 2 cache before searching main memory. Because the Level

2 cache is an integral part of the microprocessor, data moves two to four times faster

between the CPU and the cache than between the CPU and main memory.

To understand the relationship between main memory and cache memory, consider

the relationship between the size of the Web and the size of your private browser

bookmark file. Your bookmark file is small and contains only a tiny fraction of all the

available addresses on the Web; but the chance that you will soon visit a Web site that

is in your bookmark file is high. Therefore, the purpose of your bookmark file is to

keep your most frequently accessed addresses easy to reach so you can access them

quickly. Similarly, the purpose of cache memory is to keep handy the most recently

accessed data and instructions so that the CPU can access them repeatedly without

wasting time.

The movement of data or instructions from main memory to cache memory uses a

method similar to that used in paging algorithms. First, cache memory is divided into

blocks of equal size called slots. Then, when the CPU first requests an instruction or

data from a location in main memory, the requested instruction and several others

around it are transferred from main memory to cache memory, where they are stored

in one of the free slots. Moving a block at a time is based on the principle of locality

of reference, which states that it is very likely that the next CPU request will be

physically close to the one just requested. In addition to the block of data transferred,

the slot also contains a label that indicates the main memory address from which the

block was copied. When the CPU requests additional information from that location

in main memory, cache memory is accessed first; and if the contents of one of

the labels in a slot matches the address requested, then access to main memory is

not required.

The algorithm to execute one of these “transfers from main memory” is simple to

implement (a pseudocode algorithm can be found in Appendix A).

These steps become more complicated when there are no free cache slots, which can

occur because the size of cache memory is smaller than that of main memory—in this

case individual slots cannot be permanently allocated to blocks. To address this contingency,

the system needs a policy for block replacement, which could be similar to

those used in page replacement.

When designing cache memory, one must take into consideration the following four

factors:

• *Cache size.* Studies have shown that having any cache, even a small one, can substantially

improve the performance of the computer system.

• *Block size.* Because of the principle of locality of reference, as block size increases,

the ratio of number of references found in the cache to the total number of references

will tend to increase.

• *Block replacement algorithm.* When all the slots are busy and a new block has to

be brought into the cache, a block that is least likely to be used in the near future

should be selected for replacement. However, as we saw in paging, this is nearly

impossible to predict. A reasonable course of action is to select a block that has not

been used for a long time. Therefore, LRU is the algorithm that is often chosen for

block replacement, which requires a hardware mechanism to determine the least

recently used slot.

• *Rewrite policy.* When the contents of a block residing in cache are changed, it must

be written back to main memory before it is replaced by another block. A rewrite

policy must be in place to determine when this writing will take place. One alternative

is to do this every time that a change occurs, which would increase the number

of memory writes, possibly increasing overhead. A second alternative is to do this

only when the block is replaced or the process is finished, which would minimize

overhead but would leave the block in main memory in an inconsistent state. This

would create problems in multiprocessor environments and in cases where I/O

modules can access main memory directly.

The optimal selection of cache size and replacement algorithm can result in 80 to 90

percent of all requests being in the cache, making for a very efficient memory system.

This measure of efficiency, called the cache hit ratio, is used to determine the performance

of cache memory and, when shown as a percentage, represents the percentage

of total memory requests that are found in the cache. One formula is this:



For example, if the total number of requests is 10, and 6 of those are found in cache

memory, then the hit ratio is 60 percent, which is reasonably good and suggests

improved system performance.

HitRatio = (6/10) \* 100 = 60 %

Likewise, if the total number of requests is 100, and 9 of those are found in cache

memory, then the hit ratio is only 9 percent.

HitRatio = (9/100) \* 100 = 9 %

Another way to measure the efficiency of a system with cache memory, assuming that

the system always checks the cache first, is to compute the average memory access

time (Avg\_Mem**\_**AccTime) using the following formula:

Avg\_Mem\_AccTime = Avg\_Cache\_AccessTime + (1-HitRatio)\*Avg\_MainMem\_AccTime

For example, if we know that the average cache access time is 200 nanoseconds (nsec)

and the average main memory access time is 1000 nsec, then a system with a hit ratio

of 60 percent will have an average memory access time of 600 nsec:

AvgMemAccessTime = 200 + (1 - 0.60) \* 1000 = 600 nsec

A system with a hit ratio of 9 percent will show an average memory access time

of 1110 nsec:

AvgMemAccessTime = 200 + (1 - 0.09) \* 1000 = 1110 nsec

Because of its role in improving system performance, cache is roBecause of its role in improving system performance, cache is routinely added to a wide variety of main memory configurations as well as devices.

Lecture 3

An Operating System provides services to both the users and to the programs.

* It provides programs an environment to execute.
* It provides users the services to execute the programs in a convenient manner.

Following are a few common services provided by an operating system −

* Program execution
* I/O operations
* File System manipulation
* Communication
* Error Detection
* Resource Allocation
* Protection

Program execution

Operating systems handle many kinds of activities from user programs to system programs like printer spooler, name servers, file server, etc. Each of these activities is encapsulated as a process.

A process includes the complete execution context (code to execute, data to manipulate, registers, OS resources in use). Following are the major activities of an operating system with respect to program management −

* Loads a program into memory.
* Executes the program.
* Handles program's execution.
* Provides a mechanism for process synchronization.
* Provides a mechanism for process communication.
* Provides a mechanism for deadlock handling.

I/O Operation

An I/O subsystem comprises of I/O devices and their corresponding driver software. Drivers hide the peculiarities of specific hardware devices from the users.

An Operating System manages the communication between user and device drivers.

* I/O operation means read or write operation with any file or any specific I/O device.
* Operating system provides the access to the required I/O device when required.

File system manipulation

A file represents a collection of related information. Computers can store files on the disk (secondary storage), for long-term storage purpose. Examples of storage media include magnetic tape, magnetic disk and optical disk drives like CD, DVD. Each of these media has its own properties like speed, capacity, data transfer rate and data access methods.

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions. Following are the major activities of an operating system with respect to file management −

* Program needs to read a file or write a file.
* The operating system gives the permission to the program for operation on file.
* Permission varies from read-only, read-write, denied and so on.
* Operating System provides an interface to the user to create/delete files.
* Operating System provides an interface to the user to create/delete directories.
* Operating System provides an interface to create the backup of file system.

Communication

In case of distributed systems which are a collection of processors that do not share memory, peripheral devices, or a clock, the operating system manages communications between all the processes. Multiple processes communicate with one another through communication lines in the network.

The OS handles routing and connection strategies, and the problems of contention and security. Following are the major activities of an operating system with respect to communication −

* Two processes often require data to be transferred between them
* Both the processes can be on one computer or on different computers, but are connected through a computer network.
* Communication may be implemented by two methods, either by Shared Memory or by Message Passing.

Error handling

Errors can occur anytime and anywhere. An error may occur in CPU, in I/O devices or in the memory hardware. Following are the major activities of an operating system with respect to error handling −

* The OS constantly checks for possible errors.
* The OS takes an appropriate action to ensure correct and consistent computing.

Resource Management

In case of multi-user or multi-tasking environment, resources such as main memory, CPU cycles and files storage are to be allocated to each user or job. Following are the major activities of an operating system with respect to resource management −

* The OS manages all kinds of resources using schedulers.
* CPU scheduling algorithms are used for better utilization of CPU.

Protection

Considering a computer system having multiple users and concurrent execution of multiple processes, the various processes must be protected from each other's activities.

Protection refers to a mechanism or a way to control the access of programs, processes, or users to the resources defined by a computer system. Following are the major activities of an operating system with respect to protection −

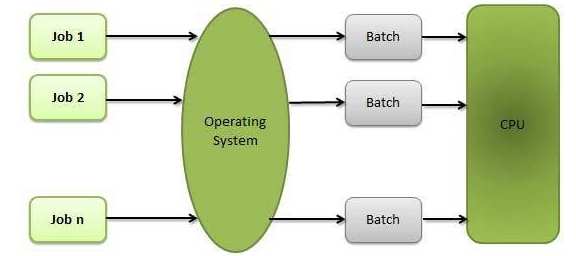
* The OS ensures that all access to system resources is controlled.
* The OS ensures that external I/O devices are protected from invalid access attempts.
* The OS provides authentication features for each user by means of passwords.

**Topic № 4**

## Batch processing

Batch processing is a technique in which an Operating System collects the programs and data together in a batch before processing starts. An operating system does the following activities related to batch processing −

* The OS defines a job which has predefined sequence of commands, programs and data as a single unit.
* The OS keeps a number a jobs in memory and executes them without any manual information.
* Jobs are processed in the order of submission, i.e., first come first served fashion.
* When a job completes its execution, its memory is released and the output for the job gets copied into an output spool for later printing or processing.

****

### Advantages

* Batch processing takes much of the work of the operator to the computer.
* Increased performance as a new job get started as soon as the previous job is finished, without any manual intervention.

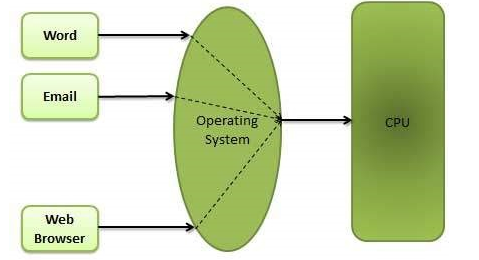
### Disadvantages

* Difficult to debug program.
* A job could enter an infinite loop.
* Due to lack of protection scheme, one batch job can affect pending jobs.

## Multitasking

Multitasking is when multiple jobs are executed by the CPU simultaneously by switching between them. Switches occur so frequently that the users may interact with each program while it is running. An OS does the following activities related to multitasking −

* The user gives instructions to the operating system or to a program directly, and receives an immediate response.
* The OS handles multitasking in the way that it can handle multiple operations/executes multiple programs at a time.
* Multitasking Operating Systems are also known as Time-sharing systems.
* These Operating Systems were developed to provide interactive use of a computer system at a reasonable cost.
* A time-shared operating system uses the concept of CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared CPU.
* Each user has at least one separate program in memory.

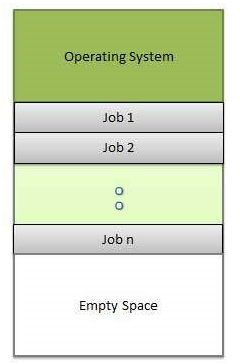


* A program that is loaded into memory and is executing is commonly referred to as a **process**.
* When a process executes, it typically executes for only a very short time before it either finishes or needs to perform I/O.
* Since interactive I/O typically runs at slower speeds, it may take a long time to complete. During this time, a CPU can be utilized by another process.
* The operating system allows the users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user.
* As the system switches CPU rapidly from one user/program to the next, each user is given the impression that he/she has his/her own CPU, whereas actually one CPU is being shared among many users.

## Multiprogramming

Sharing the processor, when two or more programs reside in memory at the same time, is referred as **multiprogramming**. Multiprogramming assumes a single shared processor. Multiprogramming increases CPU utilization by organizing jobs so that the CPU always has one to execute.

The following figure shows the memory layout for a multiprogramming system.



An OS does the following activities related to multiprogramming.

* The operating system keeps several jobs in memory at a time.
* This set of jobs is a subset of the jobs kept in the job pool.
* The operating system picks and begins to execute one of the jobs in the memory.
* Multiprogramming operating systems monitor the state of all active programs and system resources using memory management programs to ensures that the CPU is never idle, unless there are no jobs to process.

### Advantages

* High and efficient CPU utilization.
* User feels that many programs are allotted CPU almost simultaneously.

### Disadvantages

* CPU scheduling is required.
* To accommodate many jobs in memory, memory management is required.

## Interactivity

Interactivity refers to the ability of users to interact with a computer system. An Operating system does the following activities related to interactivity −

* Provides the user an interface to interact with the system.
* Manages input devices to take inputs from the user. For example, keyboard.
* Manages output devices to show outputs to the user. For example, Monitor.

The response time of the OS needs to be short, since the user submits and waits for the result.

## Real Time System

Real-time systems are usually dedicated, embedded systems. An operating system does the following activities related to real-time system activity.

* In such systems, Operating Systems typically read from and react to sensor data.
* The Operating system must guarantee response to events within fixed periods of time to ensure correct performance.

## Distributed Environment

A distributed environment refers to multiple independent CPUs or processors in a computer system. An operating system does the following activities related to distributed environment −

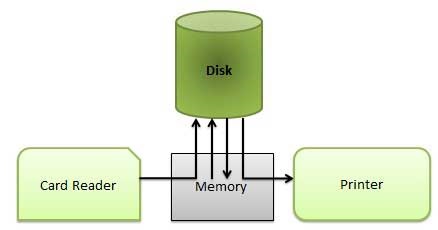
* The OS distributes computation logics among several physical processors.
* The processors do not share memory or a clock. Instead, each processor has its own local memory.
* The OS manages the communications between the processors. They communicate with each other through various communication lines.

## Spooling

Spooling is an acronym for simultaneous peripheral operations on line. Spooling refers to putting data of various I/O jobs in a buffer. This buffer is a special area in memory or hard disk which is accessible to I/O devices.

An operating system does the following activities related to distributed environment −

* Handles I/O device data spooling as devices have different data access rates.
* Maintains the spooling buffer which provides a waiting station where data can rest while the slower device catches up.
* Maintains parallel computation because of spooling process as a computer can perform I/O in parallel fashion. It becomes possible to have the computer read data from a tape, write data to disk and to write out to a tape printer while it is doing its computing task.



### Advantages

* The spooling operation uses a disk as a very large buffer.
* Spooling is capable of overlapping I/O operation for one job with processor operations for another job.

Lecture 5

In a simple system, one with a single user and one processor, the processor is busy

only when it is executing the user’s jobs or system software. However, when there are

many users, such as in a multiprogramming environment, or when there are multiple

processes competing to be run by a single CPU, the processor must be allocated to

each job in a fair and efficient manner. This can be a complex task, as we show in this

chapter, which is devoted to single processor systems.

Before we begin, let’s define some terms. The **processor**, the CPU**,** is the part of the

hardware that performs calculations and executes programs.

A **program** is an inactive unit, such as a file stored on a disk. A program is not a

process. For our discussion, a program or job is a unit of work that has been submitted

by the user.

On the other hand, a **process** is an active entity that requires a set of resources, including

a processor and special registers, to perform its function. A process, sometimes known

as a **task,** is a single instance of a program in execution.

A **thread** is created by a process, and it can be scheduled and executed independently

of its parent process. A process can consist of multiple threads. In a threaded

environment, the process owns the resources that are allocated; it then becomes

a more passive element, so its threads become the elements that use resources

(such as the CPU). Manipulating threads is less time consuming than manipulating

processes, which are more complex. Some operating systems support multiple

threads with a single process, while others support multiple processes with multiple

threads.

Multithreading allows applications to manage a separate process with several threads

of control. Web browsers use multithreading routinely. For instance, one thread

can retrieve images while another sends and retrieves e-mail. Multithreading can

also increase responsiveness in a time-sharing system, increase resource sharing, and

decrease overhead.

Here’s a simplified example. If your single-core system allows its processes to have a

single thread of control and you want to see a series of pictures on a friend’s Web site,

you can instruct the browser to establish one connection between the two sites and

download one picture at a time. However, if your system allows processes to have

multiple threads of control (a more common circumstance), then you can request several

pictures at the same time, and the browser will set up multiple connections and

download several pictures, seemingly at once.

**Multiprogramming** requires that the processor be allocated to each job or to each

process for a period of time and deallocated at an appropriate moment. If the processor

is deallocated during a program’s execution, it must be done in such a way that it

can be restarted later as easily as possible. It’s a delicate procedure. To demonstrate,

let’s look at an everyday example.

Here you are, confident you can assemble a bicycle (perhaps despite the warning that

some assembly is required). Armed with the instructions and lots of patience, you

embark on your task—to read the directions, collect the necessary tools, follow each

step in turn, and turn out the finished bike.

The first step is to join Part A to Part B with a 2-inch screw, and as you complete that

task you check off Step 1. Inspired by your success, you move on to Step 2 and then

Step 3. You’ve only just completed the third step when a neighbor is injured while

working with a power tool and cries for help.

Quickly you check off Step 3 in the directions so you know where you left off, then

you drop your tools and race to your neighbor’s side. After all, someone’s immediate

need is more important than your eventual success with the bicycle. Now you find

yourself engaged in a very different task: following the instructions in a first-aid kit

and using antiseptic and bandages.

Once the injury has been successfully treated, you return to your previous job. As you

pick up your tools, you refer to the instructions and see that you should begin with

Step 4. You then continue with your bike project until it is finally completed.

In operating system terminology, you played the part of the *CPU* or *processor*. There

were two *programs,* or *jobs*—one was the mission to assemble the bike and the second

was to bandage the injury. Each step in assembling the bike (Job A) can be called

a *process*. The call for help was an *interrupt*; when you left the bike to treat your

wounded friend, you left for a *higher priority program*. When you were interrupted,

you performed a *context switch* when you marked Step 3 as the last completed instruction

and put down your tools. Attending to the neighbor’s injury became Job B. While

you were executing the first-aid instructions, each of the steps you executed was again

a *process*. And when each job was completed, each was *finished* or terminated.

The Processor Manager would identify the series of events as follows:

Get the input for Job A (find and read the instructions in the box)

Identify the resources (collect the necessary tools)

Execute the process (follow each step in turn)

Receive the interrupt (receive call for help)

Perform a context

switch to Job B

(mark your place in the assembly

instructions)

Get the input for Job B (find your first-aid kit)

Identify the resources (identify the medical supplies)

Execute the process (follow each first aid step)

Terminate Job B (return home)

Perform context switch

to Job A

(prepare to resume assembly)

Resume executing the

interrupted process

(follow remaining steps in turn)

Terminate Job A (turn out the finished bike)

As we’ve shown, a single processor can be shared by several jobs, or several processes—

but if, and only if, the operating system has a scheduling policy, as well as a scheduling

algorithm, to determine when to stop working on one job and proceed to another.

In this example, the scheduling algorithm was based on priority: you worked on the

processes belonging to Job A (assembling the bicycle) until a higher priority job came

along. Although this was a good algorithm in this case, a priority-based scheduling

algorithm isn’t always best, as we’ll see in this chapter.

About Multi-Core Technologies

A dual-core, quad-core, or other multi-core CPU has more than one processing element

(sometimes called a core) on the computer chip. Multi-core engineering was driven by

the problems caused by nano-sized transistors and their ultra-close placement on a

computer chip. Although such an arrangement helped increase system performance

dramatically, the close proximity of these transistors also caused the unintended loss of

electrical current and excessive heat that can result in circuit failure.

One solution was to create a single chip (one piece of silicon) that housed two or more

processor cores. In other words, a single large processor was replaced with two smaller

processors (dual core), or four even smaller processors (quad core). The combined

multi-core chips are of approximately the same size as a single-processor chip but produce

less current leakage and heat. They also permit multiple calculations to take place

at the same time.

The Processor Manager is a composite of at least two submanagers: one in charge of

job scheduling and the other in charge of process scheduling. They’re known as the

**Job Scheduler** and the **Process Scheduler**.

Typically a user views a job either as a series of global job steps—compilation, loading,

and execution—or as one all-encompassing step—execution. However, the scheduling

of jobs is actually handled on two levels by most operating systems. If we return

to the example presented earlier, we can see that a hierarchy exists between the Job

Scheduler and the Process Scheduler.

The scheduling of the two jobs, to assemble the bike and to bandage the injury, was

on a priority basis. Each job was initiated by the Job Scheduler based on certain criteria.

Once a job was selected for execution, the Process Scheduler determined when

each step, or set of steps, was executed—a decision that was also based on certain

criteria. When you started assembling the bike, each step in the assembly instructions

was selected for execution by the Process Scheduler.

The same concepts apply to computer systems, where each job (or program) passes

through a hierarchy of managers. Since the first one it encounters is the Job Scheduler,

this is also called the **high-level scheduler**. It is concerned only with selecting jobs from

a queue of incoming jobs and placing them in the process queue based on each job’s

characteristics. The Job Scheduler’s goal is to put the jobs (as they still reside on the

disk) in a sequence that best meets the designers or administrator’s goals, such as using

the system’s resources as efficiently as possible.

This is an important function. For example, if the Job Scheduler selected several jobs

to run consecutively and each had a lot of requests for input and output (often abbreviated

as I/O), then the I/O devices would be kept very busy. The CPU might be busy

handling the I/O requests (if an I/O controller were not used), resulting in the completion

of very little computation. On the other hand, if the Job Scheduler selected several

consecutive jobs with a great deal of computation requirements, then the CPU would

be very busy doing that, forcing the I/O devices to remain idle waiting for requests.

Therefore, a major goal of the Job Scheduler is to create an order for the incoming

jobs that has a balanced mix of I/O interaction and computation requirements, thus

balancing the system’s resources. As you might expect, the Job Scheduler’s goal is to

keep most components of the computer system busy most of the time.

**Process Scheduler**

After a job has been accepted by the Job Scheduler to run, the Process Scheduler takes

over that job (and if the operating systems support threads, the Process Scheduler

takes responsibility for that function, too). The Process Scheduler determines which

processes will get the CPU, when, and for how long. It also decides what to do when

processing is interrupted; it determines which waiting lines (queues) the job should be

moved to during its execution; and it recognizes when a job has concluded and should

be terminated.

The Process Scheduler is a **low-level scheduler** that assigns the CPU to execute the individual

actions for those jobs placed on the READY queue by the Job Scheduler. This

becomes crucial when the processing of several jobs has to be orchestrated—just as

when you had to set aside your assembly and rush to help your neighbor.

To schedule the CPU, the Process Scheduler takes advantage of a common trait

among most computer programs: they alternate between CPU cycles and I/O cycles.

Notice that the following job has one relatively long CPU cycle and two very brief

I/O cycles:

Ask Clerk for the first number |

Retrieve the number that’s entered (Input #1) |

on the keyboard |

Ask Clerk for the second number | Brief I/O Cycle

|

Retrieve the second number entered (Input #2)

on the keyboard

Add the two numbers (Input #1 + Input #2) |

Divide the sum from the previous calculation |

and divide by 2 to get the average | CPU Cycle

(Input #1 + Input #2) / 2 |

Multiply the result of the previous calculation |

by 3 to get the average for the quarter

Multiply the result of the previous calculation

by 4 to get the average for the year

Print the average for the quarter (Output #1) | Brief I/O Cycle

|

Print the average for the year (Output #2) |

End

Although the duration and frequency of CPU cycles vary from program to program,

there are some general tendencies that can be exploited when selecting a scheduling

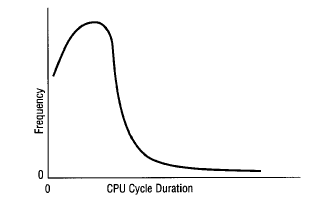
algorithm. Two of these are **I/O-bound** jobs (such as printing a series of documents)

that have long I/O cycles and brief CPU cycles and **CPU-bound** jobs (such as finding

the first 300,000 prime numbers) that have long CPU cycles and shorter I/O cycles.

The total effect of all CPU cycles, from both I/O-bound and CPU-bound jobs, approximates

a curve.



**figure**

Distribution of CPU cycle times. This distribution shows a greater number of jobs requesting short CPU cycles (the frequency peaks close to the low end of the CPU cycle axis), and fewer jobs requesting long CPU cycles.

In a highly interactive environment, there’s also a third layer of the Processor Manager

called the **middle-level scheduler**. In some cases, especially when the system is overloaded,

the middle-level scheduler finds it is advantageous to remove active jobs from

memory to reduce the degree of multiprogramming, which allows other jobs to be

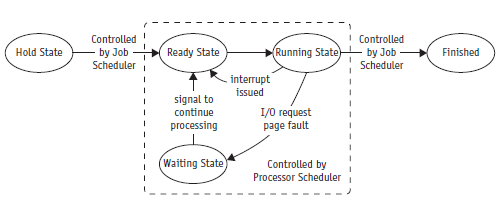
completed faster. The jobs that are swapped out and eventually swapped back in are

managed by the middle-level scheduler.

Job and Process States

As a job, a process, or a thread moves through the system, its status changes, often from

HOLD, to READY, to RUNNING, to WAITING, and eventually to FINISHED . These are called the **job status**, **process status**, or **thread status,** respectively.



**figure 4.**

*A typical job (or process) changes status as it moves through the*

*system from HOLD to FINISHED.*

Here’s how a job status can change when a user submits a job to the system. When

the job is accepted by the system, it’s put on HOLD and placed in a queue. In some

systems, the job spooler (or disk controller) creates a table with the characteristics of

each job in the queue and notes the important features of the job, such as an estimate

of CPU time, priority, special I/O devices required, and maximum memory required.

This table is used by the Job Scheduler to decide which job is to be run next.

The job moves to READY after the interrupts have been resolved. In some systems, the

job (or process) might be placed on the READY list directly. RUNNING, of course,

means that the job is being processed. In a single processor system, this is one “job”

or process. WAITING means that the job can’t continue until a specific resource is

allocated or an I/O operation has finished, and then moves back to READY. Upon

completion, the job is FINISHED and returned to the user.

The transition from one job status, job state, to another is initiated by the Job

Scheduler, and the transition from one process or thread state to another is initiated by

the Process Scheduler. Here’s a simple example:

• The job transition from HOLD to READY is initiated by the Job Scheduler (according

to a policy that’s predefined by the operating system designers). At this point, the

availability of enough main memory and any requested devices is checked.

• The transition from READY to RUNNING is handled by the Process Scheduler

according to a predefined algorithm.

The transition from RUNNING back to READY is handled by the Process Scheduler

according to a predefined time limit or other criterion, such as a priority interrupt.

• The transition from RUNNING to WAITING is handled by the Process Scheduler

and is initiated in response to an instruction in the job such as a command to

READ, WRITE, or other I/O request.

• The transition from WAITING to READY is handled by the Process Scheduler and

is initiated by a signal from the I/O device manager that the I/O request has been

satisfied and the job can continue. In the case of a page fetch, the page fault handler

will signal that the page is now in memory and the process can be placed back in

the READY queue.

• Eventually, the transition from RUNNING to FINISHED is initiated by the Process

Scheduler or the Job Scheduler when (1) the job is successfully completed and it

ends execution, or (2) the operating system indicates that an error has occurred and

the job must be terminated prematurely.

As a thread moves through the system it is in one of five states, not counting its

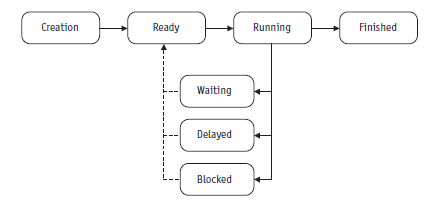
creation and finished states, as shown in Figure . When an application creates

a thread, it is made ready by allocating to it the needed resources and placing it in

the READY queue. The thread state changes from READY to RUNNING when the

Process Scheduler assigns it to a processor. In this chapter we consider systems with

only one processor.



**figure**

*A typical thread changes states from READY to FINISHED as it moves through the system*

A thread transitions from RUNNING to WAITING when it has to wait for an event

outside its control to occur. For example, a mouse click can be the trigger event for a

thread to change states, causing a transition from WAITING to READY. Alternatively,

another thread, having completed its task, can send a signal indicating that the waiting

thread can continue to execute.

When an application has the capability of delaying the processing of a thread by a specified

amount of time, it causes the thread to transition from RUNNING to DELAYED.

When the prescribed time has elapsed, the thread transitions from DELAYED to READY.

For example, when using a word processor, the thread that periodically saves a current

document can be delayed for a period of time after it has completed the save. After the

time has expired, it performs the next save and then is delayed again. If the delay was

not built into the application, this thread would be forced into a loop that would continuously

test to see if it is time to do a save, wasting processor time and reducing system

performance.

A thread transitions from RUNNING to BLOCKED when an I/O request is issued.

After the I/O is completed, the thread returns to the READY state. When a thread

transitions from RUNNING to FINISHED, all of its resources are released; it then

exits the system or is terminated and ceases to exist.

As you can see, the same operations are performed on both traditional processes and

threads. Therefore, the operating system must be able to support:

• Creating new threads

• Setting up a thread so it is ready to execute

• Delaying, or putting to sleep, threads for a specified amount of time

• Blocking, or suspending, threads that are waiting for I/O to be completed

• Setting threads to a WAIT state until a specific event has occurred

• Scheduling threads for execution

• Synchronizing thread execution using semaphores, events, or conditional variables

• Terminating a thread and releasing its resources

To do so, the operating system needs to track the critical information for each thread.

**Scheduling Algorithms**

The Process Scheduler relies on a **scheduling algorithm**, based on a specific scheduling

policy, to allocate the CPU in the best way to move jobs through the system efficiently.

Most systems place an emphasis on fast user **response time**.

To keep this discussion simple, we refer to these algorithms as **process scheduling**

**algorithms**, though they are also used to schedule threads. Here are several algorithms

that have been used extensively for these purposes.

First-Come, First-Served

**First-come, first-served (FCFS)** is a nonpreemptive scheduling algorithm that handles

all incoming objects according to their arrival time: the earlier they arrive, the sooner

they’re served. It’s a very simple algorithm to implement because it uses a First In, First

Out (FIFO) queue. This algorithm is fine for most batch systems, but it is unacceptable

for interactive systems because interactive users expect quick response times.

With FCFS, as a new job enters the system, its PCB is linked to the end of the READY

queue and it is removed from the front of the READY queue after the jobs before it

runs to completion and the processor becomes available—that is, after all of the jobs

before it in the queue have run to completion.

In a strictly FCFS system, there are no WAIT queues (each job is run to completion),

although there may be systems in which control (context) is switched on a natural wait

(I/O request) and then the job resumes on I/O completion.

The following examples presume a strictly FCFS environment (no multiprogramming).

**Turnaround time** (the time required to execute a job and return the output to the user)

is unpredictable with the FCFS policy. For example, consider the following three jobs:

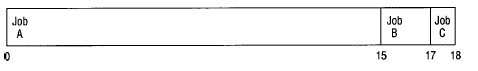
• Job A has a CPU cycle of 15 milliseconds.

• Job B has a CPU cycle of 2 milliseconds.

• Job C has a CPU cycle of 1 millisecond.

For each job, the CPU cycle contains both the actual CPU usage and the I/O requests.

That is, it is the total run time.



all three jobs arrive almost simultaneously (at Time 0), we can calculate that the turnaround

time (the job’s finish time minus arrival time) for Job A is 15, for Job B is 17,

and for Job C is 18. So the average turnaround time is:

((15 - 0) + (17 - 0) + (18 - 0))/3=16.67

However, if the jobs arrived in a different order, say C, B, A, then the results using the

same FCFS algorithm would be as shown in Figure.

If one job monopolizes the system, the extent of its overall effect on system performance

depends on the scheduling policy and whether the job is CPU-bound or I/O-bound.

While a job with a long CPU cycle (in this example, Job A) is using the CPU, the other

jobs in the system are waiting for processing or finishing their I/O requests (if an I/O

controller is used) and joining the READY queue to wait for their turn to use the processor.

If the I/O requests are not being serviced, the I/O queues remain stable while

the READY list grows (with new arrivals). In extreme cases, the READY queue could

fill to capacity while the I/O queues would be empty, or stable, and the I/O devices

would sit idle.

On the other hand, if the job is processing a lengthy I/O cycle, the I/O queues quickly

build to overflowing and the CPU could be sitting idle (if an I/O controller is used).

This situation is eventually resolved when the I/O-bound job finishes its I/O cycle, the

queues start moving again, and the system can recover from the bottleneck.

In a strictly FCFS algorithm, neither situation occurs. However, the turnaround time is

variable (unpredictable). For this reason, FCFS is a less attractive algorithm than one

that would serve the shortest job first, as the next scheduling algorithm does, even in

an environment that doesn’t support multiprogramming.

Priority Scheduling

**Priority scheduling** is one of the most common scheduling algorithms for batch

systems and is a nonpreemptive algorithm (in a batch environment). This algorithm

gives preferential treatment to important jobs. It allows the programs with the highest

priority to be processed first, and they aren’t interrupted until their CPU cycles (run

times) are completed or a natural wait occurs. If two or more jobs with equal priority

are present in the READY queue, the processor is allocated to the one that arrived first

(first-come, first-served within priority).

Priorities can be assigned by a system administrator using characteristics extrinsic

to the jobs. For example, they can be assigned based on the position of the user

(researchers first, students last) or, in commercial environments, they can be purchased

by the users who pay more for higher priority to guarantee the fastest possible

processing of their jobs. With a priority algorithm, jobs are usually linked to one of

several READY queues by the Job Scheduler based on their priority so the Process

Scheduler manages multiple READY queues instead of just one. Details about multiple

queues are presented later in this chapter.

Priorities can also be determined by the Processor Manager based on characteristics

intrinsic to the jobs such as:

• *Memory requirements.* Jobs requiring large amounts of memory could be allocated

lower priorities than those requesting small amounts of memory, or vice versa.

• *Number and type of peripheral devices.* Jobs requiring many peripheral devices

would be allocated lower priorities than those requesting fewer devices.

• *Total CPU time.* Jobs having a long CPU cycle, or estimated run time, would be

given lower priorities than those having a brief estimated run time.

• *Amount of time already spent in the system.* This is the total amount of elapsed

time since the job was accepted for processing. Some systems increase the priority

of jobs that have been in the system for an unusually long time to expedite their

exit. This is known as **aging**.

These criteria are used to determine default priorities in many systems. The default

priorities can be overruled by specific priorities named by users. There are also

preemptive priority schemes. These are discussed later in this chapter in the section on

multiple queues.

Shortest Remaining Time

**Shortest remaining time (SRT)** is a preemptive version of the SJN algorithm. The

processor is allocated to the job closest to completion—but even this job can be

interrupted if a newer job in the READY queue has a time to completion that’s shorter.

This algorithm can’t be implemented in an interactive system because it requires

advance knowledge of the CPU time required to finish each job. It can work well in

batch environments, because it can give preference to short jobs. A disadvantage is that

SRT involves more overhead than SJN—it requires the operating system to frequently

monitor the CPU time for all the jobs in the READY queue and it must perform context

switching for the jobs being swapped at preemption time (not necessarily swapped out

to the disk, although this might occur as well).

Round Robin

**Round Robin** is a preemptive process scheduling algorithm that is used extensively in interactive

systems. It’s the computing version of two children taking turns using the television

remote control. Round Robin is easy to implement. It isn’t based on job characteristics

but on a predetermined slice of time that’s given to each job to ensure that the CPU is

equally shared among all active processes and isn’t monopolized by any one job.

This time slice is called a **time quantum**; its size is crucial to the performance of the

system. It can vary from 100 milliseconds to 1 or 2 seconds.

Jobs are placed in the READY queue using a first-come, first-served scheme. The

Process Scheduler selects the first job from the front of the queue, sets the timer to

the time quantum, and allocates the CPU to this job. If processing isn’t finished when

time expires, the job is preempted and put at the end of the READY queue, and its

information is saved in its PCB.

In the event that the job’s CPU cycle is shorter than the time quantum, one of two

actions will take place: (1) If this is the job’s last CPU cycle and the job is finished,

then all resources allocated to it are released and the completed job is returned to the

user; (2) If the CPU cycle has been interrupted by an I/O request, then information

about the job is saved in its PCB and it is linked at the end of the appropriate I/O

queue. Later, when the I/O request has been satisfied, it is returned to the end of the

READY queue to await allocation of the CPU.

The efficiency of Round Robin depends on the size of the time quantum in relation

to the average CPU cycle. If the quantum is too large—that is, if it’s larger than most

CPU cycles—then the algorithm reduces to the FCFS scheme. If the quantum is too

small, then the amount of context switching slows down the execution of the jobs and

the amount of overhead is dramatically increased, as the three examples in Figure 4.12

demonstrate. Job A has a CPU cycle of 8 milliseconds. The amount of context switching

increases as the time quantum decreases in size.

Multiple-Level Queues

**Multiple-level queues** isn’t really a separate scheduling algorithm, but works in conjunction

with several of the schemes already discussed and is found in systems with

jobs that can be grouped according to a common characteristic. We’ve already introduced

at least one kind of multiple-level queue—that of a priority-based system with a

different queue for each priority level.

Another kind of system might gather all of the CPU-bound jobs in one queue and all

I/O-bound jobs in another. The Process Scheduler then alternately selects jobs from

each queue to keep the system balanced.

A third common example can be used in a hybrid environment that supports both

batch and interactive jobs. The batch jobs are put in one queue, called the background

queue, while the interactive jobs are put in a foreground queue and are treated more

favorably than those on the background queue.

All of these examples have one thing in common: The scheduling policy is based on some

predetermined scheme that allocates special treatment to the jobs in each queue. With

multiple-level queues, the system designers can choose to use different algorithms for different

queues, allowing them to combine the advantages of several algorithms. For example,

within each queue, the jobs are served in FCFS fashion or use some other scheme instead.

Multiple-level queues raise some interesting questions:

• Is the processor allocated to the jobs in the first queue until it is empty before

moving to the next queue, or does it travel from queue to queue until the last job

on the last queue has been served? And then go back to serve the first job on the

first queue? Or something in between?

• Is this fair to those who have earned, or paid for, a higher priority?

• Is it fair to those in a low-priority queue?

• If the processor is allocated to the jobs on the first queue and it never empties out,

when will the jobs in the last queues be served?

• Can the jobs in the last queues get “time off for good behavior” and eventually

move to better queues?

The answers depend on the policy used by the system to service the queues. There

are four primary methods to the movement: not allowing movement between queues,

moving jobs from queue to queue, moving jobs from queue to queue and increasing

the time quantums for lower queues, and giving special treatment to jobs that have

been in the system for a long time (aging). We explore each of these methods with the

following four cases.

Case 1: No Movement Between Queues

No movement between queues is a very simple policy that rewards those who have

high-priority jobs. The processor is allocated to the jobs in the high-priority queue in

FCFS fashion, and it is allocated to jobs in low-priority queues only when the highpriority

queues are empty. This policy can be justified if there are relatively few users

with high-priority jobs so the top queues quickly empty out, allowing the processor to

spend a fair amount of time running the low-priority jobs.

Case 2: Movement Between Queues

Movement between queues is a policy that adjusts the priorities assigned to each job:

High-priority jobs are treated like all the others once they are in the system. (Their

initial priority may be favorable.) When a time quantum interrupt occurs, the job is

preempted and moved to the end of the next lower queue. A job may also have its

priority increased, such as when it issues an I/O request before its time quantum has

expired.

This policy is fairest in a system in which the jobs are handled according to their computing

cycle characteristics: CPU-bound or I/O-bound. This assumes that a job that

exceeds its time quantum is CPU-bound and will require more CPU allocation than

one that requests I/O before the time quantum expires. Therefore, the CPU-bound jobs

are placed at the end of the next lower-level queue when they’re preempted because

of the expiration of the time quantum, while I/O-bound jobs are returned to the end

of the next higher-level queue once their I/O request has finished. This facilitates

I/O-bound jobs and is good in interactive systems.

Case 3: Variable Time Quantum Per Queue

Variable time quantum per queue is a variation of the movement between queues

policy. It allows for faster turnaround of CPU-bound jobs.

In this scheme, each of the queues is given a time quantum twice as long as the

previous queue. The highest queue might have a time quantum of 100 milliseconds.

The second-highest queue would have a time quantum of 200 milliseconds, the third

would have 400 milliseconds, and so on. If there are enough queues, the lowest one

might have a relatively long time quantum of 3 seconds or more.

If a job doesn’t finish its CPU cycle in the first time quantum, it is moved to the end

of the next lower-level queue; and when the processor is next allocated to it, the job

executes for twice as long as before. With this scheme, a CPU-bound job can execute

for longer and longer periods of time, thus improving its chances of finishing

faster.

Case 4: Aging

Aging is used to ensure that jobs in the lower-level queues will eventually complete

their execution. The operating system keeps track of each job’s waiting time, and

when a job gets too old—that is, when it reaches a certain time limit—the system

moves the job to the next highest queue, and so on, until it reaches the top queue.

A more drastic aging policy is one that moves the old job directly from the lowest

queue to the end of the top queue. Regardless of its actual implementation, an

aging policy guards against the indefinite postponement of unwieldy jobs. As you

might expect, **indefinite postponement** means that a job’s execution is delayed for

an undefined amount of time because it is repeatedly preempted so other jobs can be

processed. (We all know examples of an unpleasant task that’s been indefinitely postponed

to make time for a more appealing pastime). Eventually the situation could

lead to the old job’s starvation causing it to never be processed. Indefinite postponement

is a major problem when allocating resources and one that is discussed in detail

in Chapter 5.

Earliest Deadline First

**Earliest Deadline First (EDF),** known as a dynamic priority algorithm, is a preemptive

scheduling algorithm built to address the critical processing requirements of real-time

systems and their pressing deadlines. Contrary to the fixed priority scheme explored

earlier, where the priority of a job does not change after it enters the system, with EDF

the priority can be adjusted as it moves through execution from START to FINISHED.

The primary goal of EDF is to process all jobs in the order that is most likely to allow

each to run to completion before reaching their respective deadlines. Initially, the priority

assigned to each job is based on the amount of time remaining until the job’s

impending deadline—and that priority is inversely proportional to its absolute deadline.

So, in its simplest sense: the closer the deadline, the higher the priority. Sometime two

or more jobs share the same deadline, in which case the tie is broken using any scheme

such as first in, first out. Remember, the goal is to complete all jobs before each one

reaches its deadline.

**Conclusion**

The Processor Manager must allocate the CPU among all the system’s users and

all of their jobs, processes, and threads. In this chapter we’ve made the distinction

between job scheduling (the selection of incoming jobs based on their characteristics)

and process and thread scheduling (the instant-by-instant allocation of the CPU).

We’ve also described how interrupts are generated and resolved by the interrupt

handler.

Each scheduling algorithm presented in this chapter has unique characteristics, objectives,

and applications. A system designer can choose the best policy and algorithm

only after carefully evaluating the strengths and weaknesses of each one that’s available

in the context of the system’s requirements. Table 4.3 shows how the algorithms

presented in this chapter compare.

In the next chapter we explore the demands placed on the Processor Manager as it

attempts to synchronize execution of every job admitted to the system to avoid

deadlocks, livelock, and starvation.

**Process Management**

In computer systems, a deadlock is a system-wide tangle of resource requests that

begins when two or more jobs are put on hold, each waiting for a vital resource

to become available. The problem builds when the resources needed by those jobs

are the resources held by other jobs that are also waiting to run but cannot because

they’re waiting for other unavailable resources. The tangled jobs come to a standstill.

The deadlock is complete if the remainder of the system comes to a standstill as well.

When the situation can’t be resolved by the operating system, then intervention is

required.

Deadlock is more serious than indefinite postponement or starvation because it affects

more than one job. Because resources are being tied up, the entire system (not just a

few programs) is affected. There’s no simple and immediate solution to a deadlock; no

one can move forward until someone moves out of the way, but no one can move out

of the way until either someone advances or everyone behind moves back. Obviously,

it requires outside intervention. Only then can the deadlock be resolved.

Deadlocks became prevalent with the introduction of interactive systems, which generally

improve the use of resources through dynamic resource sharing, but this capability

also increases the possibility of deadlocks.

In some computer systems, deadlocks are regarded as a mere inconvenience that causes

delays. But for real-time systems, deadlocks cause critical situations. For example, a

deadlock in a hospital’s life support system or in the guidance system aboard an aircraft

could endanger lives. Regardless of the environment, the operating system must

either prevent deadlocks or resolve them when they happen.

A deadlock usually occurs when nonsharable, nonpreemptable resources, such as files,

printers, or scanners, are allocated to jobs that eventually require other non sharable,

non preemptive resources—resources that have been locked by other jobs. However,

deadlocks aren’t restricted to files, printers, and scanners. They can also occur on sharable

resources that are locked, such as disks and databases.

Directed graphs visually represent the system’s resources and processes and show how

they are deadlocked. Using a series of squares for resources, circles for processes, and

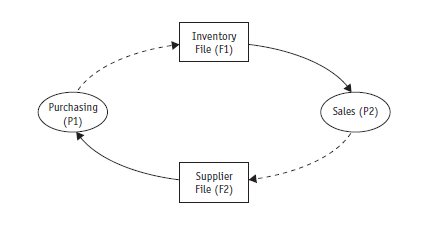
connectors with arrows for requests, directed graphs can be manipulated to understand

how deadlocks occur.

Case 1: Deadlocks on File Requests

If jobs are allowed to request and hold files for the duration of their execution, a deadlock

can occur, as the simplified directed graph shown in Figure illustrates.



**figure**

*Case 1. These two processes, shown as circles, are each waiting for a resource, shown as rectangles, that has*

*already been allocated to the othe r process, thus creating a deadlock.*

For example, consider the case of a home construction company with two application

programs, Purchasing and Sales, which are active at the same time. Both need to access

two separate files, called Inventory and Suppliers, to read and write transactions. One

day the system deadlocks when the following sequence of events takes place:

1. The Purchasing process accesses the Suppliers file to place an order for

more lumber.

2. The Sales process accesses the Inventory file to reserve the parts that will be

required to build the home ordered that day.

3. The Purchasing process doesn’t release the Suppliers file, but it requests the

Inventory file so it can verify the quantity of lumber on hand before placing its

order for more. However, Purchasing is blocked because the Inventory file is

being held by Sales.

4. Meanwhile, the Sales process doesn’t release the Inventory file (because it needs

it), but requests the Suppliers file to check the schedule of a subcontractor.

At this point, the Sales process is also blocked because the Suppliers file is

already being held by the Purchasing process.

In the meantime, any other programs that require the Inventory or Suppliers files will

be put on hold as long as this situation continues. This deadlock will remain until one

of the two programs is closed or forcibly removed and its file is released. Only then

can the other program continue and the system return to normal.

Case 2: Deadlocks in Databases

A deadlock can also occur if two processes access and lock records in a database.

To appreciate the following scenario, remember that database queries and transactions

are often relatively brief processes that either search or modify parts of a database.

Requests usually arrive at random and may be interleaved arbitrarily.

Database **locking** is a technique used to guarantee the integrity of the data through

which the user locks out all other users while working with the database. Locking can

be done at three different levels: the entire database can be locked for the duration of

the request; a subsection of the database can be locked; or only the individual record

can be locked. Locking the entire database (the most extreme and most successful

solution) prevents a deadlock from occurring, but it restricts access to the database

to one user at a time and, in a multiuser environment, response times are significantly

slowed; this is normally an unacceptable solution. When the locking is performed on

only one part of the database, access time is improved, but the possibility of a deadlock

is increased because different processes sometimes need to work with several parts of

the database at the same time.

Here’s a system that locks each record in the database when it is accessed and keeps it

locked until that process is completed. Let’s assume there are two processes (The sales

process and address process), each of which needs to update a different record (the

final exam record and the current address record), and the following sequence leads to

a deadlock:

1. The sales process accesses the first quarter record and locks it.

2. The address process accesses the current address record and locks it.

3. The sales process requests the current address record, which is locked by the

address process.

4. The address process requests the first quarter record, which is locked by the

sales process.

If locks are *not* used to preserve database integrity, the resulting records in the database

might include only some of the data—and their contents would depend on the order in which each process finishes its execution. Known as a **race** between processes.

Case 3: Deadlocks in Dedicated Device Allocation

The use of a group of dedicated devices, such as two audio recorders, can also deadlock

the system. Remember that dedicated devices cannot be shared.

Case 4: Deadlocks in Multiple Device Allocation

Deadlocks aren’t restricted to processes that are contending for the same type of device;

they can happen when several processes request, and hold on to, several dedicated

devices while other processes act in a similar manner, as shown in Figure.



*Three processes, shown as circles, are each waiting for a device that has already been allocated*

*to another process, thus creating a deadlock.*

Consider the case of an engineering design firm with three programs (P1, P2, and P3)

and three dedicated devices: scanner, printer, and plotter. The following sequence of

events will result in deadlock:

1. Program 1 (P1) requests and gets the only scanner.

2. Program 2 requests and gets the only printer.

3. Program 3 requests and gets the only plotter.

4. Now, Program 1 requests the printer but is blocked.

5. Then, Program 2 requests the plotter but is blocked.

6. Finally, Program 3 requests the scanner but is blocked and the deadlock begins.

As was the case in the earlier examples, none of these programs can continue because

each is waiting for a necessary resource that’s already being held by another.

**Case 5: Deadlocks in Spooling**

Although in the previous example the printer was a dedicated device, printers can be

sharable devices and join a category called “virtual devices” that uses high-speed storage

to transfer data between it and the CPU. The spooler accepts output from several

users and acts as a temporary storage area for all output until the printer is ready to

accept it. This process is called **spooling**. However, if the printer needs all of a job’s

output before it will begin printing, but the spooling system fills the available space

with only partially completed output, then a deadlock can occur.

This scenario isn’t limited to printers. Any part of the computing system that relies on

spooling, such as one that handles incoming jobs or transfers files over a network, is

vulnerable to such a deadlock.

Case 6: Deadlocks in a Network

A network that’s congested or has filled a large percentage of its I/O buffer space

can become deadlocked if it doesn’t have protocols to control the flow of messages

through the network.

**Case 7: Deadlocks in Disk Sharing**

Disks are designed to be shared, so it’s not uncommon for two processes to access different

areas of the same disk. This ability to share creates an active type of deadlock,

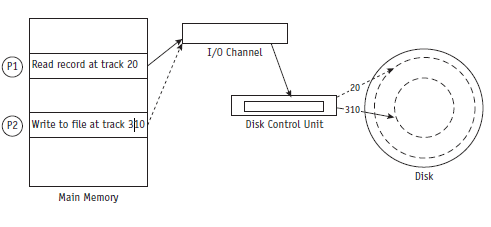
known as livelock. Processes use a form of busy-waiting that’s different from a natural

wait. In this case, it’s waiting to share a resource but never actually gains control of

it. In Figure 5.6, two competing processes are sending conflicting commands, causing

livelock (notice that neither process is blocked, which would cause a deadlock). Instead,

each remains active but without achieving any progress and never reaching fulfillment.



**figure**

*Case . Two processes are each waiting for an I/O request to be filled: one attrack 20 and one at track*

*310. But by the time the read/write arm reaches one track, a competing command for the other track*

*has been issued, so neither command is satisfie.*

Necessary Conditions for Deadlock or Livelock

In each of these seven cases, the deadlock or livelock involved the interaction of several

processes and resources, but each time, it was preceded by the simultaneous

occurrence of four conditions that the operating system (or other systems) could have

recognized: mutual exclusion, resource holding, no preemption, and circular wait. It’s

important to remember that each of these four conditions is necessary for the operating

system to work smoothly. None of them can be removed easily without causing

the system’s overall functioning to suffer. Therefore, the system needs to recognize the

combination of conditions before they occur.

To illustrate these four conditions, let’s revisit the staircase example from the beginning

of the chapter to identify the four conditions required for a locked system. (Note

that deadlock and livelock share the same requirements, so the following discussion

applies to both.)

1. When two people meet on the steps, between landings, they can’t pass because

the steps can hold only one person at a time. **Mutual exclusion**, the act of

allowing only one person (or process) to have access to a step (or a dedicated

resource), is the first condition for deadlock.

2. When two people meet on the stairs and each one holds ground and waits

for the other to retreat, that is an example of **resource holding** (as opposed to

resource sharing), the second condition for deadlock.

3. In this example, each step is dedicated to the climber (or the descender); it is allocated

to the holder for as long as needed. This is called **no preemption**, the lack

of temporary reallocation of resources, and is the third condition for deadlock.

a system’s requests and releases can be received in an unpredictable

order, which makes it very difficult to design a foolproof preventive policy.

In general, operating systems use some combination of several strategies to deal with

deadlocks:

• Prevent one of the four conditions from occurring: **prevention**.

• Avoid the deadlock if it becomes probable: **avoidance**.

• Detect the deadlock when it occurs: **detection**.

• Recover from it gracefully: **recovery**.

Prevention

To prevent a deadlock, the operating system must eliminate one of the four necessary

conditions discussed at the beginning of this chapter, a task complicated by the fact

that the same condition can’t be eliminated from every resource.

Mutual exclusion is necessary in any computer system because some resources, such

as memory, CPU, and dedicated devices, must be exclusively allocated to one user

at a time. In the case of I/O devices, such as printers, the mutual exclusion may be

bypassed by spooling, which allows the output from many jobs to be stored in separate

temporary spool files at the same time, and each complete output file is then

selected for printing when the device is ready. However, we may be trading one type

of deadlock (such as Case 3: Deadlocks in Dedicated Device Allocation) for another

(Case 5: Deadlocks in Spooling).

Resource holding, where a job holds on to one resource while waiting for another

one that’s not yet available, could be sidestepped by forcing each job to request,

at creation time, every resource it will need to run to completion. For example, if

every job in a batch system is given as much memory as it needs, then the number

of active jobs will be dictated by how many can fit in memory—a policy that

would significantly decrease the degree of multiprogramming. In addition, peripheral

devices would be idle because they would be allocated to a job even though

they wouldn’t be used all the time. As we’ve said before, this was used successfully

in batch environments, although it did reduce the effective use of resources and

restricted the amount of multiprogramming. But it doesn’t work as well in interactive

systems.

No preemption could be bypassed by allowing the operating system to deallocate

resources from jobs. This can be done if the state of the job can be easily

saved and restored, as when a job is preempted in a round robin environment or

a page is swapped to secondary storage in a virtual memory system. On the other

hand, preemption of a dedicated I/O device (printer, plotter, scanner, and so on),

or of files during the modification process, can require some extremely unpleasant

recovery tasks.

Circular wait can be bypassed if the operating system prevents the formation of a

circle. One such solution proposed by Havender (1968) is based on a numbering system

for the resources such as: printer = 1, disk = 2, scanner = 3, plotter = 4, and so on.

The system forces each job to request its resources in ascending order: any “number

one” devices required by the job would be requested first; any “number two” devices

would be requested next; and so on. So if a job needed a printer and then a plotter,

it would request them in this order: printer (1) first and then the plotter (4). If the

job required the plotter first and then the printer, it would still request the printer

first (which is a 1) even though it wouldn’t be used right away. A job could request a

printer (1) and then a disk (2) and then a scanner (3); but if it needed another printer

(1) late in its processing, it would still have to anticipate that need when it requested

the first one and before it requested the disk.

This scheme of “hierarchical ordering” removes the possibility of a circular wait and

therefore guarantees the removal of deadlocks. It doesn’t require that jobs state their

maximum needs in advance, but it does require that the jobs anticipate the order in

which they will request resources. One of the difficulties of this scheme is discovering

the best order for the resources so that the needs of the majority of the users are satisfied.

Another difficulty is that of assigning a ranking to nonphysical resources such as

files or locked database records where there is no basis for assigning a higher number

to one over another.

Avoidance

Even if the operating system can’t remove one of the conditions for deadlock, it can

avoid one if the system knows ahead of time the sequence of requests associated with

each of the active processes. As was illustrated in the graphs shown in Figure 5.7

through Figure 5.10, there exists at least one allocation of resources sequence that will

allow jobs to continue without becoming deadlocked.

One such algorithm was proposed by Dijkstra in 1965 to regulate resource allocation

to avoid deadlocks. The Banker’s Algorithm is based on a bank with a fixed amount of

capital that operates on the following principles:

• No customer will be granted a loan exceeding the bank’s total capital.

• All customers will be given a maximum credit limit when opening an account.

• No customer will be allowed to borrow over the limit.

• The sum of all loans won’t exceed the bank’s total capital.

Under these conditions, the bank isn’t required to have on hand the total of all

maximum lending quotas before it can open up for business (we’ll assume the bank

will always have the same fixed total and we’ll disregard interest charged on loans).

For our example, the bank has a total capital fund of $10,000 and has three customers

(#1, #2, and #3), who have maximum credit limits of $4,000, $5,000, and

$8,000, respectively. Table 5.4 illustrates the state of affairs of the bank after some

loans have been granted to customers #2 and #3. This is called a **safe state** because

the bank still has enough money left to satisfy the maximum requests of all three

customers.

✔Recovery

Once a deadlock has been detected, it must be untangled and the system returned to

normal as quickly as possible. There are several **recovery** algorithms, but they all have

one feature in common: They all require at least one **victim**, an expendable job, which,

when removed from the deadlock, will free the system. Unfortunately for the victim,

removal generally requires that the job be restarted from the beginning or from a convenient

midpoint.

The first and simplest recovery method, and the most drastic, is to terminate every job

that’s active in the system and restart them from the beginning.

The second method is to terminate only the jobs involved in the deadlock and ask their

users to resubmit them.

**To re**The third method is to identify which jobs are involved in the deadlock and terminate

them one at a time, checking to see if the deadlock is eliminated after each removal,

until the deadlock has been resolved. Once the system is freed, the remaining jobs are

allowed to complete their processing, and later, the halted jobs are started again from

the beginning.

The fourth method can be put into effect only if the job keeps a record, a snapshot, of

its progress so it can be interrupted and then continued without starting again from the

beginning of its execution. The snapshot is like the landing in our staircase example:

Instead of forcing the deadlocked stair climbers to return to the bottom of the stairs,

they need to retreat only to the nearest landing and wait until the others have passed.

Then the climb can be resumed. In general, this method is favored for long-running

jobs to help them make a speedy recovery.

Until now we’ve offered solutions involving the jobs caught in the deadlock. The next

two methods concentrate on the nondeadlocked jobs and the resources they hold.

One of them, the fifth method in our list, selects a nondeadlocked job, preempts the

resources it’s holding, and allocates them to a deadlocked process so it can resume

execution, thus breaking the deadlock. The sixth method stops new jobs from entering

the system, which allows the nondeadlocked jobs to run to completion so they’ll

release their resources. Eventually, with fewer jobs in the system, competition for

resources is curtailed so the deadlocked processes get the resources they need to run

to completion. This method is the only one listed here that doesn’t rely on a victim,

and it’s not guaranteed to work unless the number of available resources surpasses

that needed by at least one of the deadlocked jobs to run (this is possible with multiple

resources).

Several factors must be considered to select the victim that will have the least-negative

effect on the system. The most common are:

• The priority of the job under consideration—high-priority jobs are usually

untouched.

• CPU time used by the job—jobs close to completion are usually left alone.

• The number of other jobs that would be affected if this job were selected as

the victim.

In addition, programs working with databases also deserve special treatment because

a database that is only partially updated is only partially correct. Therefore, jobs

that are modifying data shouldn’t be selected for termination because the consistency

and validity of the database would be jeopardized. Fortunately, designers of

many database systems have included sophisticated recovery mechanisms so damage

to the database is minimized if a transaction is interrupted or terminated before

completion.

**Starvation**

So far in this chapter, we have concentrated on deadlocks and livelocks, both of which

result from the liberal allocation of resources. At the opposite end is starvation, the

result of conservative allocation of resources, where a single job is prevented from

execution because it’s kept waiting for resources that never become available. To illustrate

this, the case of the Dining Philosophers Problem was introduced by Dijkstra

in 1968.

Five philosophers are sitting at a round table, each deep in thought. In the center of the

table, accessible to everyone, is a bowl of spaghetti. There are five forks on the table—

one between each philosopher, as illustrated in Figure 5.13. Local custom dictates that

each philosopher must use two forks, the forks on either side of the plate, to eat the

spaghetti, but there are only five forks—not the 10 it would require for all five thinkers

to eat at once—and that’s unfortunate for Philosopher 2.

When they sit down to dinner, Philosopher 1 (P1) is the first to take the two forks (both

F1 and F5) on either side of the plate and begins to eat. Inspired by this bold move,

Philosopher 3 (P3) does likewise, acquiring both F2 and F3. Now Philosopher 2 (P2)

** funds to**

**satisfy** decides to begin the meal, but is unable to begin because no forks are available: Fork #1

has been allocated to Philosopher 1, and Fork #2 has been allocated to Philosopher 3.

In fact, by this time, four of the five forks have been allocated. The only remaining

fork is situated between Philosopher 4 and Philosopher 5. So Philosopher 2

must wait.

Soon, Philosopher 3 finishes eating, puts down the two forks, and resumes pondering.

Should the fork that’s now free, Fork #2, be allocated to the hungry Philosopher 2,

even though only that one fork is free? Although it’s tempting, such a move would be

a bad precedent, because if the philosophers are allowed to tie up resources with only

the hope that the other required resource will become available, the dinner could easily

slip into an unsafe state; it would be only a matter of time before each philosopher

held a single fork, and nobody could eat. So the resources are allocated to the philosophers

only when both forks are available at the same time. The status of the “system”

is illustrated in Figure 5.14.

P4 and P5 are quietly thinking and Philosopher 1 is still eating. Philosopher 3 (who

should be full) decides to eat some more and is able to take F2 and F3 once again. Soon

thereafter, Philosopher 1 finishes and releases Fork #1 and Fork #5, but Philosopher 2**the needs**

**of** decides to begin the meal, but is unable to begin because no forks are available: Fork #1

has been allocated to Philosopher 1, and Fork #2 has been allocated to Philosopher 3.

In fact, by this time, four of the five forks have been allocated. The only remaining

fork is situated between Philosopher 4 and Philosopher 5. So Philosopher 2

must wait.

Soon, Philosopher 3 finishes eating, puts down the two forks, and resumes pondering.

Should the fork that’s now free, Fork #2, be allocated to the hungry Philosopher 2,

even though only that one fork is free? Although it’s tempting, such a move would be

a bad precedent, because if the philosophers are allowed to tie up resources with only

the hope that the other required resource will become available, the dinner could easily

slip into an unsafe state; it would be only a matter of time before each philosopher

held a single fork, and nobody could eat. So the resources are allocated to the philosophers

only when both forks are available at the same time. The status of the “system”

is illustrated in Figure 5.14.

P4 and P5 are quietly thinking and Philosopher 1 is still eating. Philosopher 3 (who

should be full) decides to eat some more and is able to take F2 and F3 once again. Soon

thereafter, Philosopher 1 finishes and releases Fork #1 and Fork #5, but Philosopher 2

**t l**is still not able to eat, because Fork #2is now allocated. This scenario could continue

forever; as long as Philosopher 1 and Philosopher 3 alternate their use of the available

resources, Philosopher 2 must wait. Philosophers 1 and 3 can eat any time they wish,

while Philosopher 2 starves, only inches from nourishment.

In a computer environment, the resources are like forks and the competing processes

are like dining philosophers. If the resource manager doesn’t watch for starving processes

and jobs and plan for their eventual completion, they could remain in the system

forever waiting for the right combination of resources.

To address this problem, an algorithm designed to detect starving jobs can be implemented,

which tracks how long each job has been waiting for resources (this is the

same as aging, described in Chapter 4). Once starvation has been detected, the system

can block new jobs until the starving jobs have been satisfied. This algorithm must

be monitored closely: If monitoring is done too often, then new jobs will be blocked

too frequently and throughput will be diminished. If it’s not done often enough, then

starving jobs will remain in the system for an unacceptably long period of time.**e**

**Topic № 6**

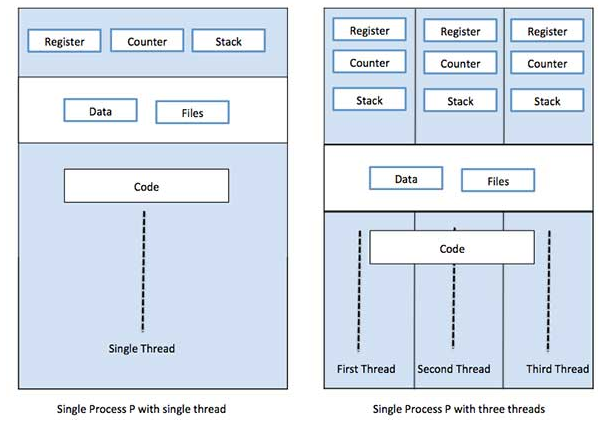
## What is Thread?

A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

A thread is also called a **lightweight process**. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. The following figure shows the working of a single-threaded and a multithreaded process.



Difference between Process and Thread

|  |  |  |
| --- | --- | --- |
| **S.N.** | **Process** | **Thread** |
| 1 | Process is heavy weight or resource intensive. | Thread is light weight, taking lesser resources than a process. |
| 2 | Process switching needs interaction with operating system. | Thread switching does not need to interact with operating system. |
| 3 | In multiple processing environments, each process executes the same code but has its own memory and file resources. | All threads can share same set of open files, child processes. |
| 4 | If one process is blocked, then no other process can execute until the first process is unblocked. | While one thread is blocked and waiting, a second thread in the same task can run. |
| 5 | Multiple processes without using threads use more resources. | Multiple threaded processes use fewer resources. |
| 6 | In multiple processes each process operates independently of the others. | One thread can read, write or change another thread's data. |

Advantages of Thread

* Threads minimize the context switching time.
* Use of threads provides concurrency within a process.
* Efficient communication.
* It is more economical to create and context switch threads.
* Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

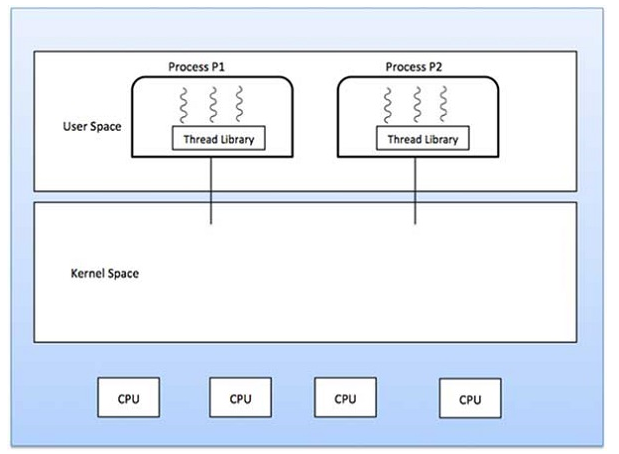
Types of Thread

Threads are implemented in following two ways −

* **User Level Threads** − User managed threads.
* **Kernel Level Threads** − Operating System managed threads acting on kernel, an operating system core.

User Level Threads

In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.



### Advantages

* Thread switching does not require Kernel mode privileges.
* User level thread can run on any operating system.
* Scheduling can be application specific in the user level thread.
* User level threads are fast to create and manage.

### Disadvantages

* In a typical operating system, most system calls are blocking.
* Multithreaded application cannot take advantage of multiprocessing.

## Kernel Level Threads

In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

### Advantages

* Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
* If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
* Kernel routines themselves can be multithreaded.

### Disadvantages

* Kernel threads are generally slower to create and manage than the user threads.
* Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

## Multithreading Models

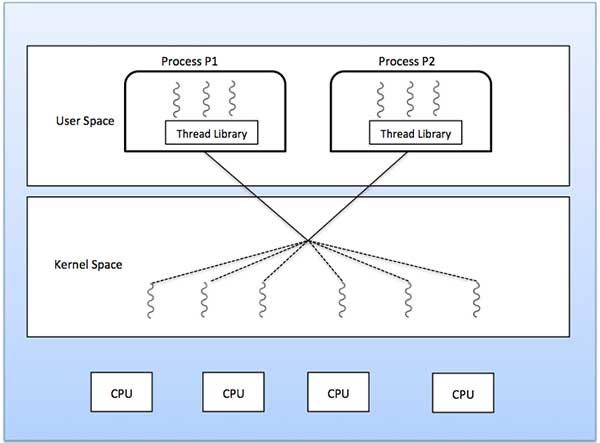
Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types

* Many to many relationship.
* Many to one relationship.
* One to one relationship.

## Many to Many Model

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.

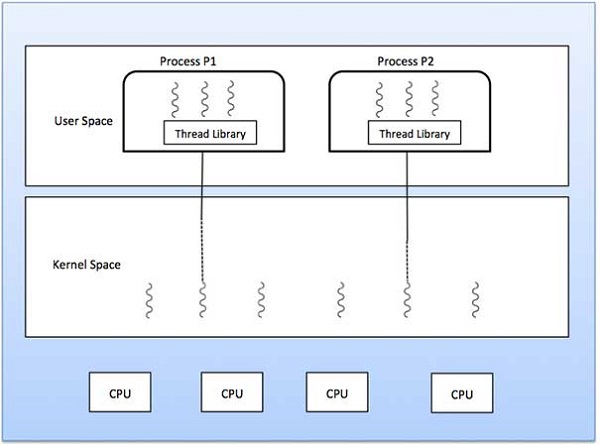
The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.



## Many to One Model

Many-to-one model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

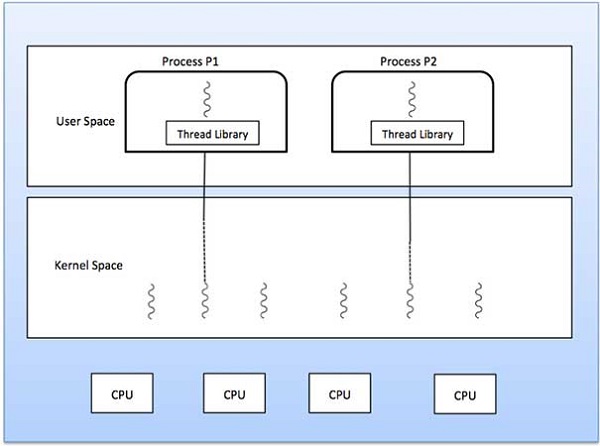
If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.



## One to One Model

There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.



## Difference between User-Level & Kernel-Level Thread

|  |  |  |
| --- | --- | --- |
| **S.N.** | **User-Level Threads** | **Kernel-Level Thread** |
| 1 | User-level threads are faster to create and manage. | Kernel-level threads are slower to create and manage. |
| 2 | Implementation is by a thread library at the user level. | Operating system supports creation of Kernel threads. |
| 3 | User-level thread is generic and can run on any operating system. | Kernel-level thread is specific to the operating system. |
| 4 | Multi-threaded applications cannot take advantage of multiprocessing. | Kernel routines themselves can be multithreaded. |

**Lecture 7**

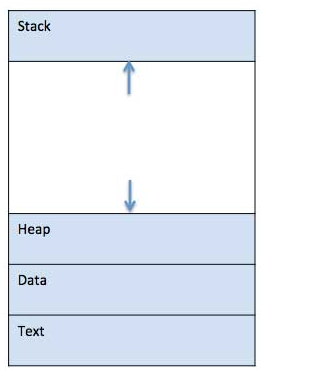
Process

A process is basically a program in execution. The execution of a process must progress in a sequential fashion.

A process is defined as an entity which represents the basic unit of work to be implemented in the system.

To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.

When a program is loaded into the memory and it becomes a process, it can be divided into four sections ─ stack, heap, text and data. The following image shows a simplified layout of a process inside main memory −

****

|  |  |
| --- | --- |
| **S.N.** | **Component & Description** |
| 1 | **Stack**  The process Stack contains the temporary data such as method/function parameters, return address and local variables. |
| 2 | **Heap**  This is dynamically allocated memory to a process during its run time. |
| 3 | **Text**  This includes the current activity represented by the value of Program Counter and the contents of the processor's registers. |
| 4 | **Data**  This section contains the global and static variables. |

Program

A program is a piece of code which may be a single line or millions of lines. A computer program is usually written by a computer programmer in a programming language. For example, here is a simple program written in C programming language −

#include <stdio.h>

int main() {

printf("Hello, World! \n");

return 0;

}

A computer program is a collection of instructions that performs a specific task when executed by a computer. When we compare a program with a process, we can conclude that a process is a dynamic instance of a computer program.

A part of a computer program that performs a well-defined task is known as an **algorithm**. A collection of computer programs, libraries and related data are referred to as a **software**.

Process Life Cycle

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized.

In general, a process can have one of the following five states at a time.

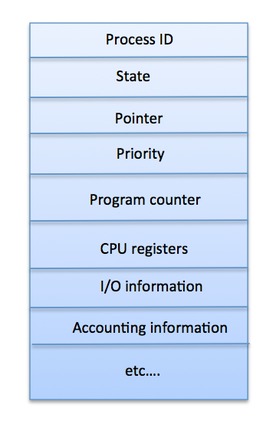
|  |  |
| --- | --- |
| **S.N.** | **State & Description** |
| 1 | **Start**  This is the initial state when a process is first started/created. |
| 2 | **Ready**  The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after **Start** state or while running it by but interrupted by the scheduler to assign CPU to some other process. |
| 3 | **Running**  Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions. |
| 4 | **Waiting**  Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available. |
| 5 | **Terminated or Exit**  Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory. |

Process Control Block (PCB)

A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below in the table −

|  |  |
| --- | --- |
| **S.N.** | **Information & Description** |
| 1 | **Process State**  The current state of the process i.e., whether it is ready, running, waiting, or whatever. |
| 2 | **Process privileges**  This is required to allow/disallow access to system resources. |
| 3 | **Process ID**  Unique identification for each of the process in the operating system. |
| 4 | **Pointer**  A pointer to parent process. |
| 5 | **Program Counter**  Program Counter is a pointer to the address of the next instruction to be executed for this process. |
| 6 | **CPU registers**  Various CPU registers where process need to be stored for execution for running state. |
| 7 | **CPU Scheduling Information**  Process priority and other scheduling information which is required to schedule the process. |
| 8 | **Memory management information**  This includes the information of page table, memory limits, Segment table depending on memory used by the operating system. |
| 9 | **Accounting information**  This includes the amount of CPU used for process execution, time limits, execution ID etc. |
| 10 | **IO status information**  This includes a list of I/O devices allocated to the process. |

The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. Here is a simplified diagram of a PCB −



The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

## Process Scheduling Queues

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues −

* **Job queue** − This queue keeps all the processes in the system.
* **Ready queue** − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
* **Device queues** − The processes which are blocked due to unavailability of an I/O device constitute this queue.

The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

Two-State Process Model

Two-state process model refers to running and non-running states which are described below −

|  |  |
| --- | --- |
| **S.N.** | **State & Description** |
| 1 | **Running**  When a new process is created, it enters into the system as in the running state. |
| 2 | **Not Running**  Processes that are not running are kept in queue, waiting for their turn to execute. Each entry in the queue is a pointer to a particular process. Queue is implemented by using linked list. Use of dispatcher is as follows. When a process is interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded. In either case, the dispatcher then selects a process from the queue to execute. |

Schedulers

Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types −

* Long-Term Scheduler
* Short-Term Scheduler
* Medium-Term Scheduler

## Long Term Scheduler

It is also called a **job scheduler**. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling.

The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

On some systems, the long-term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

## Short Term Scheduler

It is also called as **CPU scheduler**. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.

Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

## Medium Term Scheduler

Medium-term scheduling is a part of **swapping**. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the swapped out-processes.

A running process may become suspended if it makes an I/O request. A suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage. This process is called **swapping**, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

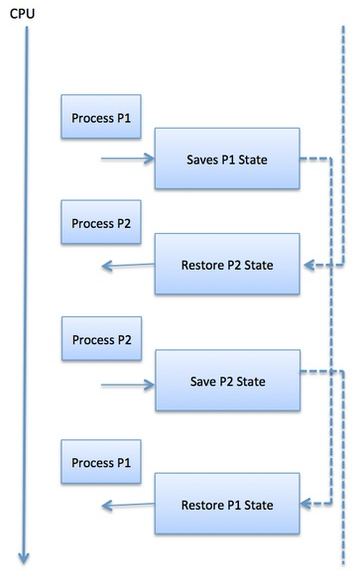
Comparison among Scheduler

|  |  |  |  |
| --- | --- | --- | --- |
| **S.N.** | **Long-Term Scheduler** | **Short-Term Scheduler** | **Medium-Term Scheduler** |
| 1 | It is a job scheduler | It is a CPU scheduler | It is a process swapping scheduler. |
| 2 | Speed is lesser than short term scheduler | Speed is fastest among other two | Speed is in between both short and long term scheduler. |
| 3 | It controls the degree of multiprogramming | It provides lesser control over degree of multiprogramming | It reduces the degree of multiprogramming. |
| 4 | It is almost absent or minimal in time sharing system | It is also minimal in time sharing system | It is a part of Time sharing systems. |
| 5 | It selects processes from pool and loads them into memory for execution | It selects those processes which are ready to execute | It can re-introduce the process into memory and execution can be continued. |

Context Switch

A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.



Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers. When the process is switched, the following information is stored for later use.

* Program Counter
* Scheduling information
* Base and limit register value
* Currently used register
* Changed State
* I/O State information
* Accounting information

**Parallel processing** is a situation in which two or more processors operate in one

system at the same time and may or may not work on related activities. In other words,

two or more CPUs execute instructions simultaneously and the Processor Manager

has to coordinate activities of each processor while synchronizing the cooperative

interactions among all of them.

There are two primary benefits to parallel processing systems: increased reliability and

faster processing.

The reliability stems from the availability of more than one CPU. Theoretically, if one

processor fails, then the others can continue to operate and absorb the load. However,

this capability must be designed into the system so that, first, the failing processor

informs other processors to take over, and second, the operating system dynamically

restructures its available resources and allocation strategies so that the remaining processors

don’t become overloaded.

The increased processing speed can be achieved when instructions or data manipulation

can be processed in parallel, two or more at a time. Some systems allocate a

CPU to each program or job. Others allocate a CPU to each working set or parts of

it. Still others subdivide individual instructions so that each subdivision can be processed

simultaneously (called concurrent programming). And others achieve parallel

processing by allocating several CPUs to perform a set of instructions separately on

vast amounts of data and combine all the results at the end of the job.

Increased flexibility brings increased complexity, however, and two major challenges

remain: how to connect the processors into workable configurations and how to

orchestrate their interaction to achieve efficiencies, which applies to multiple interacting

processes as well. (It might help if you think of each process as being run on a

separate processor.)

The complexities of the Processor Manager’s multiprocessing tasks are easily illustrated

with an example: You’re late for an early afternoon appointment and you’re in

danger of missing lunch, so you get in line for the drive-through window of the local

fast-food shop. When you place your order, the order clerk confirms your request,

tells you how much it will cost, and asks you to drive to the pickup window; there,

a cashier collects your money and hands over your order. All’s well and once again

you’re on your way—driving and thriving. You just witnessed a well-synchronized

multiprocessing system. Although you came in contact with just two processors (the

order clerk and the cashier), there were at least two other processors behind the scenes

who cooperated to make the system work—the cook and the bagger.

A fast food lunch spot is similar to the six-step information retrieval system below.

It is described in a different way in Table 6.1.

a) Processor 1 (the order clerk) accepts the query, checks for errors, and passes

the request on to Processor 2 (the bagger).

b) Processor 2 (the bagger) searches the database for the required information

(the hamburger).

c) Processor 3 (the cook) retrieves the data from the database (the meat to cook

for the hamburger) if it’s kept off-line in secondary storage.

d) Once the data is gathered (the hamburger is cooked), it’s placed where

Processor 2 can get it (in the hamburger bin).

e) Processor 2 (the bagger) retrieves the data (the hamburger) and passes it on to

Processor 4 (the cashier).

f) Processor 4 (the cashier) routes the response (your order) back to the originator

of the request—you.

Levels of Multiprocessing

Multiprocessing can take place at several different levels, each of which tends to require

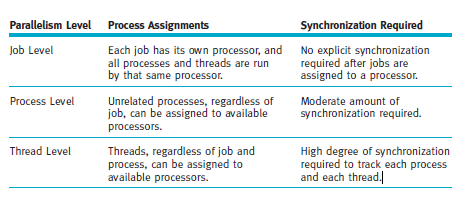
a different frequency of synchronization, as shown in Table 6.2. At the job level,

multiprocessing is fairly benign. It’s as if each job is running on its own workstation

with shared system resources. On the other hand, when multiprocessing takes place at

the thread level, a high degree of synchronization is required to disassemble each process,

perform the thread’s instructions, and then correctly reassemble the process.

****

**Multi-core processors** have several processors on a single chip. As processors became

smaller in size (as predicted by Moore’s Law) and faster in processing speed, CPU

designers began to use nanometer-sized transistors (one nanometer is one billionth of

a meter). Each transistor switches between two positions—0 and 1—as the computer

conducts its binary arithmetic at increasingly fast speeds. However, as transistors

reached nano-sized dimensions and the space between transistors became ever closer,

the quantum physics of electrons got in the way.

In a nutshell, here’s the problem. When transistors are placed extremely close together,

electrons can spontaneously tunnel, at random, from one transistor to another, causing

a tiny but measurable amount of current to leak. The smaller the transistor, the more

significant the leak. (When an electron does this “tunneling,” it seems to spontaneously

disappear from one transistor and appear in another nearby transistor.

A second problem was the heat generated by the chip. As processors became faster, the

heat also climbed and became increasingly difficult to disperse. These heat and tunneling

issues threatened to limit the ability of chip designers to make processors ever

smaller.

One solution was to create a single chip (one piece of silicon) with two “processor

cores” in the same amount of space. With this arrangement, two sets of calculations

can take place at the same time. The two cores on the chip generate less heat than a

single core of the same size, and tunneling is reduced; however, the two cores each run

more slowly than the single core chip. Therefore, to get improved performance from

a dual-core chip, the software has to be structured to take advantage of the double

calculation capability of the new chip design. Building on their success with two-core

chips, designers have created multi-core processors with more than 80 cores on a

single chip. An example is shown in Chapter 1.

Does this hardware innovation affect the operating system? Yes, because it must

manage multiple processors, multiple units of cache and RAM, and the processing of

many tasks at once. However, a dual-core chip is not always faster than a single-core

chip. It depends on the tasks being performed and whether they’re multi-threaded or

sequential.

**Typical Multiprocessing Configurations**

Much depends on how the multiple processors are configured within the system. Three

typical configurations are master/slave, loosely coupled, and symmetric.

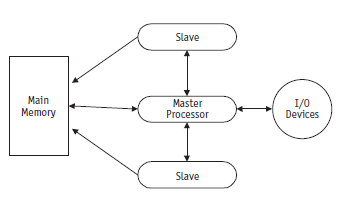
Master/Slave Configuration

The **master/slave** configuration is an asymmetric multiprocessing system. Think of it as

a single-processor system with additional slave processors, each of which is managed

by the primary master processor as shown in Figure

**Software that**

**requires sequentia**

**l***In a master/slave multiprocessing configuration, slave processors can access main memory directly but they must send all*

*I/O requests through the master processor.*

The master processor is responsible for managing the entire system—all files, devices,

memory, and processors. Therefore, it maintains the status of all processes in the

system, performs storage management activities, schedules the work for the other

processors, and executes all control programs. This configuration is well suited for

computing environments in which processing time is divided between front-end and

back-end processors; in these cases, the front-end processor takes care of the interactive

users and quick jobs, and the back-end processor takes care of those with long

jobs using the batch mode.

The primary advantage of this configuration is its simplicity. However, it has three

serious disadvantages:

• Its reliability is no higher than for a single-processor system because if the master

processor fails, none of the slave processors can take over, and the entire system

fails.

• It can lead to poor use of resources because if a slave processor should become free

while the master processor is busy, the slave must wait until the master becomes

free and can assign more work to it.

• It increases the number of interrupts because all slave processors must interrupt the

master processor every time they need operating system intervention, such as for I/O

requests. This creates long queues at the master processor level when there are many

processors and many interrupts.

Loosely Coupled Configuration

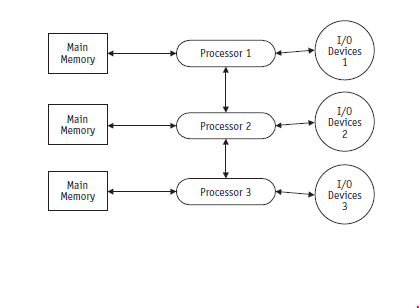
The **loosely coupled configuration** features several complete computer systems, each

with its own memory, I/O devices, CPU, and operating system, as shown in Figure .

This configuration is called loosely coupled because each processor controls its

own resources—its own files, access to memory, and its own I/O devices—and that

means that each processor maintains its own commands and I/O management tables.

**

**calculations can**

**run more slowly**

**on a chip with**

**dual cores than**

*In a loosely coupled multiprocessing configuration, each processor has its own*

*dedicated resources.*

**on a similar chip**

**with one core**

**b**The only difference between a loosely coupled multiprocessing system and a collection

of independent single-processing systems is that each processor can communicate and

cooperate with the others.

When a job arrives for the first time, it’s assigned to one processor. Once allocated, the

job remains with the same processor until it’s finished. Therefore, each processor must

have global tables that indicate where each job has been allocated.

To keep the system well balanced and to ensure the best use of resources, job scheduling

is based on several requirements and policies. For example, new jobs might be

assigned to the processor with the lightest load or the one with the best combination of

output devices available.

This system isn’t prone to catastrophic system failures because even when a single

processor fails, the others can continue to work independently. However, it can be

difficult to detect when a processor has failed.

Symmetric Configuration

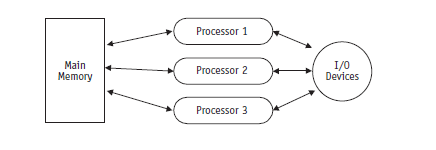
A **symmetric configuration** (as depicted in Figure ) features decentralized processor

scheduling. That is, a single copy of the operating system and a global table listing

each process and its status is stored in a common area of memory so every processor

has access to it. Each processor uses the same scheduling algorithm to select which

process it will run next.**ecause the**

**calcul**

**ations cannot**The symmetric configuration (sometimes called tightly coupled) has four advantages

over loosely coupled configuration:

• It’s more reliable.

• It uses resources effectively.

• It can balance loads well.

• It can degrade gracefully in the event of a failure.

Whenever a process is interrupted, whether because of an I/O request or another type

of interrupt, its processor updates the corresponding entry in the process list and finds

another process to run. This means that the processors are kept quite busy. But it

also means that any given job or task may be executed by several different processors

during its run time. And because each processor has access to all I/O devices and can

reference any storage unit, there are more conflicts as several processors try to access

the same resource at the same time.

This presents the obvious need for algorithms to resolve conflicts among processors.

**Process Synchronization Software**

The success of **process synchronization** hinges on the capability of the operating system

to make a resource unavailable to other processes while it is being used by one

of them. These “resources” can include scanners and other I/O devices, a location in

storage, or a data file, to name a few. In essence, the resource that’s being used must

be locked away from other processes until it is released. Only then is a waiting process

allowed to use the resource. This is where synchronization is critical. A mistake could

leave a job waiting indefinitely, causing starvation, or if it’s a key resource, cause

a deadlock.

Obviously, this situation calls for synchronization. Several synchronization mechanisms

are available to provide cooperation and communication among processes.

The common element in all synchronization schemes is to allow a process to finish

work on a critical part of the program before other processes have access to it. This is

applicable both to multiprocessors and to two or more processes in a single-processor

(time-shared) system. It is called a **critical region** because it is a critical section and

its execution must be handled as a unit. As we’ve seen, the processes within a critical

region can’t be interleaved without threatening the integrity of the operation.

Synchronization is sometimes implemented as a lock-and-key arrangement: Before a

process can work on a critical region, it must get the key. And once it has the key, all

other processes are locked out until it finishes, unlocks the entry to the critical region,

and returns the key so that another process can get the key and begin work.

This sequence consists of two actions: (1) the process must first see if the key is available

and (2) if it is available, the process must pick it up and put it in the lock to make

it unavailable to all other processes. For this scheme to work, both actions must be

performed in a single machine cycle; otherwise it is conceivable that while the first

process is ready to pick up the key, another one would find the key available and prepare

to pick up the key—and each could block the other from proceeding any further.

Several locking mechanisms have been developed, including test-and-set, WAIT and

SIGNAL, and semaphores.

Test-and-Set

**Test-and-set** is a single, indivisible machine instruction known simply as **TS** and was

introduced by IBM for its early multiprocessing computers. In a single machine cycle,

it tests to see if the key is available and, if it is, sets it to unavailable.

The actual key is a single bit in a storage location that can contain a 0

Therefore, a process (Process 1) tests the condition code using the TS instruction before

entering a critical region. If no other process is in this critical region, then Process 1 is

allowed to proceed and the condition code is changed from 0 to 1. Later, when Process 1

exits the critical region, the condition code is reset to 0 so another process is allowed

to enter. On the other hand, if Process 1 finds a busy condition code when it arrives,

then it’s placed in a waiting loop where it continues to test the condition code—when

it’s free, it’s allowed to enter.

Although it’s a simple procedure to implement, and it works well for a small number

of processes, test-and-set has two major drawbacks:

• First, when many processes are waiting to enter a critical region, starvation can

occur because the processes gain access in an arbitrary fashion. Unless a first-come,

first-served policy is set up, some processes could be favored over others.

• Second, the waiting processes remain in unproductive, resource-consuming wait

loops, requiring context switching, because the processes repeatedly check for the

key. This is known as **busy waiting**—which not only consumes valuable processor

time, but also relies on the competing processes to test the key—something that is

best handled by the operating system or the hardware.

WAIT and SIGNAL

**WAIT and SIGNAL** is a modification of test-and-set that’s designed to remove busy

waiting. Two new operations, WAIT and SIGNAL, are mutually exclusive and become

part of the process scheduler’s set of operations.

WAIT is activated when the process encounters a busy condition code. WAIT sets the

process’s process control block (PCB) to the blocked state and links it to the queue of

processes waiting to enter this particular critical region. The Process Scheduler then

selects another process for execution. SIGNAL is activated when a process exits the

critical region and the condition code is set to “free.” It checks the queue of processes

waiting to enter this critical region and selects one, setting it to the READY state.

Eventually the Process Scheduler will choose this process for running. The addition of

the operations WAIT and SIGNAL frees the processes from the busy waiting dilemma

and returns control to the operating system, which can then run other jobs while the

waiting processes are idle (WAIT).

Semaphores

A **semaphore** is a non-negative integer variable that can be used as a binary signal,

a flag. One of the most historically significant semaphores was the signaling device.

In an operating system, a semaphore is set to either zero or one to perform a similar

function: It signals if and when a resource is free and can be used by a process. Dutch

computer scientist Edsger Dijkstra (1965) introduced two operations to overcome the

process synchronization problem we’ve discussed. Dijkstra called them P and V, and

that’s how they’re known today. The P stands for the Dutch word *proberen* (to test)

and the V stands for *verhogen* (to increment). The P and V operations do just that:

They test and increment.

Here’s how semaphores work: If we let *s* be a semaphore variable, then the V operation

on *s* is simply to increment *s* by 1. The action can be stated as:

V(*s*): *s*: = *s* + 1

This in turn necessitates a fetch (to get the current value of *s*), increment (to add one to

*s*), and store sequence. Like the test-and-set operation, the increment operation must

be performed as a single indivisible action to avoid race conditions. And that means

that *s* cannot be accessed by any other process during the operation.

The operation P on *s* is to test the value of *s*, and if it’s not 0, to decrement it by 1. The

action can be stated as:

P(*s*): If *s* > 0 then *s*: = *s −* 1

This involves a test, fetch, decrement, and store sequence. Again, this sequence must

be performed as an indivisible action in a single machine cycle or be arranged so that

the process cannot take action until the operation (test or increment) is finished.

In sequential computations mutual exclusion is achieved automatically because each

operation is handled in order, one at a time. However, in parallel computations, the

order of execution can change, so mutual exclusion must be explicitly stated and

maintained. In fact, the entire premise of parallel processes hinges on the requirement

that all operations on common variables consistently exclude one another over time.

**Process Cooperation**

There are occasions when several processes work directly together to complete a common

task. Two famous examples are the problems of producers and consumers, and

of readers and writers. Each case requires both mutual exclusion and synchronization,

and each is implemented by using semaphores.

Producers and Consumers

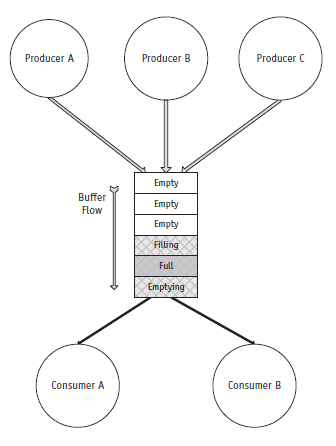
The classic problem of **producers and consumers** concerns one or more processes that

produce data that one or more process consumes later. They do so by using a single

buffer. Let’s begin with the case with one producer and one consumer, although current

versions would almost certainly apply to systems with multiple producers and

consumers, as shown in Figure.

****

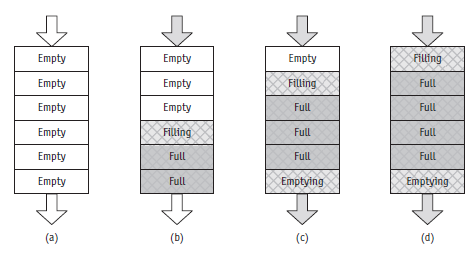
Consider the case of the prolific CPU that can generate output data much faster than

a printer can print it. Therefore, to accommodate the two different speeds, we need

a buffer where the producer can temporarily store data that can be retrieved by the

consumer at a slower speed, freeing the CPU from having to wait unnecessarily. This

buffer can be in many states, from empty to full, as shown in Figure.



The essence of the problem is that the system must make sure that the producer won’t

try to add data to a full buffer, and that the consumer won’t try to make withdrawals

from an empty buffer. Because the buffer can hold only a finite amount of data, the

synchronization process must delay the producer from generating more data when the

buffer is full. It must also be prepared to delay the consumer from retrieving data when

the buffer is empty. This task can be implemented by two counting semaphores—one

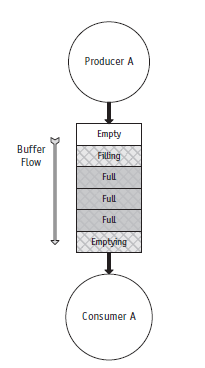
to indicate the number of *full* positions in the buffer and the other to indicate the

number of *empty* positions in the buffer. A third semaphore, mutex, ensures mutual

exclusion between processes to prevent race conditions.

Notice that when there is only one producer and one consumer, this classic problem

becomes a simple case of first in, first out (FIFO), as shown in Figure .



However, when there are several producers and consumers, a deadlock can occur if they are all

paused as they wait to be notified that they can resume their respective deposits and

withdrawals.

Readers and Writers

The problem of **readers and writers** was first formulated by Courtois, Heymans, and

Parnas (1971) and arises when two types of processes need to access a shared resource

such as a file or database. The authors called these processes readers and writers.

An airline reservation system is a good example. The readers are those who want flight

information. They’re called readers because they only read the existing data; they

don’t modify it. And because none of them is changing the database, the system can

allow many readers to be active at the same time—there’s no need to enforce mutual

exclusion among them.

The writers are those who are making reservations on a particular flight. Unlike the

readers, the writers must be carefully accommodated for each flight because they are

modifying existing data in the database. The system can’t allow someone to be writing

while someone else is reading or writing to the exact same file. Therefore, it must

enforce mutual exclusion for any and all writers. Of course, the system must be fair

when it enforces its policy to avoid indefinite postponement of readers or writers.

In the original paper, Courtois, Heymans, and Parnas offered two solutions using

P and V operations.

• The first gives priority to readers over writers so readers are kept waiting only if a

writer is actually modifying the data. However, if there is a continuous stream of

readers, this policy results in writer starvation.

• The second policy gives priority to the writers. In this case, as soon as a writer

arrives, any readers that are already active are allowed to finish processing, but all

additional readers are put on hold until the writer is done. Obviously, if a continuous

stream of writers is present, this policy results in reader starvation.

• Either scenario is unacceptable.

To prevent either type of starvation, Hoare (1974) proposed the following combination

priority policy: When a writer is finished, any and all readers who are waiting or

on hold are allowed to read. Then, when that group of readers is finished, the writer

who is on hold can begin; any *new* readers who arrive in the meantime aren’t allowed

to start until the writer is finished.

The state of the system can be summarized by four counters initialized to 0:

• Number of readers who have *requested* a resource and haven’t yet released it (R1 = 0)

• Number of readers who are *using* a resource and haven’t yet released it (R2 = 0)

• Number of writers who have *requested* a resource and haven’t yet released it (W1 = 0)

• Number of writers who are *using* a resource and haven’t yet released it (W2 = 0)

This can be implemented using two semaphores to ensure mutual exclusion between

readers and writers. A resource can be given to all readers, provided that no writers

are processing (W2 = 0). A resource can be given to a writer, provided that no readers

are reading (R2 = 0) and no writers are writing (W2 = 0).

Readers must always call two procedures: the first checks whether the resources can be

immediately granted for reading; and then, when the resource is released, the second checks

to see if there are any writers waiting. The same holds true for writers. The first procedure

must determine if the resource can be immediately granted for writing, and then, upon

releasing the resource, the second procedure will find out if any readers are waiting.

**Concurrent Programming**

Until now, we’ve looked at multiprocessing as several jobs executing at the same time

on a single processor (which interacts with I/O processors, for example) or on multiprocessors.

Multiprocessing can also refer to one job using several processors to execute

sets of instructions in parallel. The concept isn’t new, but it requires a programming

language and a computer system that can support this type of construct. This type of

system is referred to as a **concurrent processing** system, and it can generally perform

data level parallelism and instruction (or task) level parallelism.

In general, parallel systems can be put into two broad categories: **data level parallelism**

**(DLP),** which refers to systems that can perform on one or more streams or elements

of data, and **instruction (or task) level parallelism (ILP**), which refers to systems that

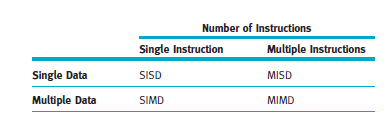
can perform multiple instructions in parallel.

Michael Flynn (1972) published what’s now known as Flynn’s taxonomy, shown in

Table 6.4, describing machine structures that fall into four main classifications of parallel

construction with combinations of single/multiple instruction and single/multiple

data. Each category presents different opportunities for parallelism.

****

Each category presents different opportunities for parallelism.

• The single instruction, single data (SISD) classification is represented by systems

with a single processor and a single stream of data which presents few if any

opportunities for parallelism.

• The multiple instructions, single data (MISD) classification includes systems with

multiple processors (which might allow some level of parallelism) and a single

stream of data. Configurations such as this might allow instruction level parallelism

but little or no data level parallelism without additional software assistance.

• The single instruction, multiple data (SIMD) classification is represented by systems

with a single processor and a multiple data streams.

• The multiple instructions, multiple data (MIMD) classification is represented by systems

with a multiple processors and a multiple data streams. These systems may allow

the most creative use of both instruction level parallelism and data level parallelism.

**Topic № 8**

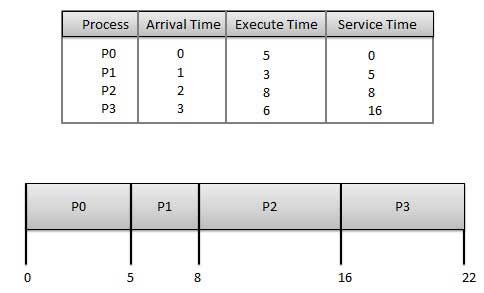
A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms which we are going to discuss in this chapter −

* First-Come, First-Served (FCFS) Scheduling
* Shortest-Job-Next (SJN) Scheduling
* Priority Scheduling
* Shortest Remaining Time
* Round Robin(RR) Scheduling
* Multiple-Level Queues Scheduling

These algorithms are either **non-preemptive or preemptive**. Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

First Come First Serve (FCFS)

* Jobs are executed on first come, first serve basis.
* It is a non-preemptive, pre-emptive scheduling algorithm.
* Easy to understand and implement.
* Its implementation is based on FIFO queue.
* Poor in performance as average wait time is high.



**Wait time** of each process is as follows −

|  |  |
| --- | --- |
| **Process** | **Wait Time : Service Time - Arrival Time** |
| P0 | 0 - 0 = 0 |
| P1 | 5 - 1 = 4 |
| P2 | 8 - 2 = 6 |
| P3 | 16 - 3 = 13 |

Average Wait Time: (0+4+6+13) / 4 = 5.75

Shortest Job Next (SJN)

* This is also known as **shortest job first**, or SJF
* This is a non-preemptive, pre-emptive scheduling algorithm.
* Best approach to minimize waiting time.
* Easy to implement in Batch systems where required CPU time is known in advance.
* Impossible to implement in interactive systems where required CPU time is not known.
* The processer should know in advance how much time process will take.

Given: Table of processes, and their Arrival time, Execution time

|  |  |  |  |
| --- | --- | --- | --- |
| **Process** | **Arrival Time** | **Execution Time** | **Service Time** |
| P0 | 0 | 5 | 0 |
| P1 | 1 | 3 | 5 |
| P2 | 2 | 8 | 14 |
| P3 | 3 | 6 | 8 |

**Waiting time** of each process is as follows −

|  |  |
| --- | --- |
| **Process** | **Waiting Time** |
| P0 | 0 - 0 = 0 |
| P1 | 5 - 1 = 4 |
| P2 | 14 - 2 = 12 |
| P3 | 8 - 3 = 5 |

Average Wait Time: (0 + 4 + 12 + 5)/4 = 21 / 4 = 5.25

Priority Based Scheduling

* Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
* Each process is assigned a priority. Process with highest priority is to be executed first and so on.
* Processes with same priority are executed on first come first served basis.
* Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Given: Table of processes, and their Arrival time, Execution time, and priority. Here we are considering 1 is the lowest priority.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Process** | **Arrival Time** | **Execution Time** | **Priority** | **Service Time** |
| P0 | 0 | 5 | 1 | 0 |
| P1 | 1 | 3 | 2 | 11 |
| P2 | 2 | 8 | 1 | 14 |
| P3 | 3 | 6 | 3 | 5 |

**Waiting time** of each process is as follows −

|  |  |
| --- | --- |
| **Process** | **Waiting Time** |
| P0 | 0 - 0 = 0 |
| P1 | 11 - 1 = 10 |
| P2 | 14 - 2 = 12 |
| P3 | 5 - 3 = 2 |

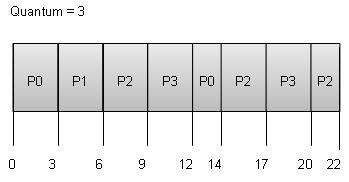
Average Wait Time: (0 + 10 + 12 + 2)/4 = 24 / 4 = 6

Shortest Remaining Time

* Shortest remaining time (SRT) is the preemptive version of the SJN algorithm.
* The processor is allocated to the job closest to completion but it can be preempted by a newer ready job with shorter time to completion.
* Impossible to implement in interactive systems where required CPU time is not known.
* It is often used in batch environments where short jobs need to give preference.

Round Robin Scheduling

* Round Robin is the preemptive process scheduling algorithm.
* Each process is provided a fix time to execute, it is called a **quantum**.
* Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
* Context switching is used to save states of preempted processes.



**Wait time** of each process is as follows −

|  |  |
| --- | --- |
| **Process** | **Wait Time : Service Time - Arrival Time** |
| P0 | (0 - 0) + (12 - 3) = 9 |
| P1 | (3 - 1) = 2 |
| P2 | (6 - 2) + (14 - 9) + (20 - 17) = 12 |
| P3 | (9 - 3) + (17 - 12) = 11 |

Average Wait Time: (9+2+12+11) / 4 = 8.5

Multiple-Level Queues Scheduling

Multiple-level queues are not an independent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics.

* Multiple queues are maintained for processes with common characteristics.
* Each queue can have its own scheduling algorithms.
* Priorities are assigned to each queue.

For example, CPU-bound jobs can be scheduled in one queue and all I/O-bound jobs in another queue. The Process Scheduler then alternately selects jobs from each queue and assigns them to the CPU based on the algorithm assigned to the queue.

**Topic № 9**

**Process Synchronization** is the task of coordinating the execution of processes in a way that no two processes can have access to the same shared data and resources.

It is specially needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or data at the same time.

This can lead to the inconsistency of shared data. So the change made by one process not necessarily reflected when other processes accessed the same shared data. To avoid this type of inconsistency of data, the processes need to be synchronized with each other.

For Example, process A changing the data in a memory location while another process B is trying to read the data from the **same** memory location. There is a high probability that data read by the second process will be erroneous.

## Sections of a Program

Here, are four essential elements of the critical section:

* **Entry Section:** It is part of the process which decides the entry of a particular process.
* **Critical Section:** This part allows one process to enter and modify the shared variable.
* **Exit Section:** Exit section allows the other process that are waiting in the Entry Section, to enter into the Critical Sections. It also checks that a process that finished its execution should be removed through this Section.
* **Remainder Section:**Allother parts of the Code, which is not in Critical, Entry, and Exit Section, are known as the Remainder Section.

## What is Critical Section Problem?

A critical section is a segment of code which can be accessed by a signal process at a specific point of time. The section consists of shared data resources that required to be accessed by other processes.

* The entry to the critical section is handled by the wait() function, and it is represented as P().
* The exit from a critical section is controlled by the signal() function, represented as V().

In the critical section, only a single process can be executed. Other processes, waiting to execute their critical section, need to wait until the current process completes its execution.

## Rules for Critical Section

The critical section need to must enforce all three rules:

* **Mutual Exclusion:** Mutual Exclusion is a special type of binary semaphore which is used for controlling access to the shared resource. It includes a priority inheritance mechanism to avoid extended priority inversion problems. Not more than one process can execute in its critical section at one time.
* **Progress:**This solution is used when no one is in the critical section, and someone wants in. Then those processes not in their reminder section should decide who should go in, in a finite time.
* **Bound Waiting:**When a process makes a request for getting into critical section, there is a specific limit about number of processes can get into their critical section. So, when the limit is reached, the systemmust allow request to the process to get into its critical section.

## Solutions To The Critical Section

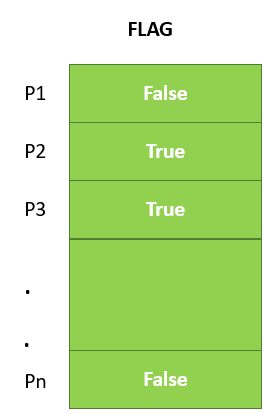
In Process Synchronization, critical section plays the main role so that the problem must be solved.

Here are some widely used methods to solve the critical section problem.

### Peterson Solution

Peterson's solution is widely used solution to critical section problems. This algorithm was developed by a computer scientist Peterson that's why it is named as a Peterson's solution.

In this solution, when a process is executing in a critical state, then the other process only executes the rest of the code, and the opposite can happen. This method also helps to make sure that only a single process runs in the critical section at a specific time.



PROCESS Pi

FLAG[i] = true

while( (turn != i) AND (CS is !free) ){ wait;

}

CRITICAL SECTION FLAG[i] = false

turn = j; //choose another process to go to CS

* Assume there are N processes (P1, P2, ... PN) and every process at some point of time requires to enter the Critical Section
* A FLAG[] array of size N is maintained which is by default false. So, whenever a process requires to enter the critical section, it has to set its flag as true. For example, If Pi wants to enter it will set FLAG[i]=TRUE.
* Another variable called TURN indicates the process number which is currently wating to enter into the CS.
* The process which enters into the critical section while exiting would change the TURN to another number from the list of ready processes.
* Example: turn is 2 then P2 enters the Critical section and while exiting turn=3 and therefore P3 breaks out of wait loop.

### Synchronization Hardware

Sometimes the problems of the Critical Section are also resolved by hardware. Some operating system offers a lock functionality where a Process acquires a lock when entering the Critical section and releases the lock after leaving it.

o when another process is trying to enter the critical section, it will not be able to enter as it is locked. It can only do so if it is free by acquiring the lock itself.

### Mutex Locks

Synchronization hardware not simple method to implement for everyone, so strict software method known as Mutex Locks was also introduced.

In this approach, in the entry section of code, a LOCK is obtained over the critical resources used inside the critical section. In the exit section that lock is released.

### Semaphore Solution

Semaphore is simply a variable that is non-negative and shared between threads. It is another algorithm or solution to the critical section problem. It is a signaling mechanism and a thread that is waiting on a semaphore, which can be signaled by another thread.

It uses two atomic operations, 1)wait, and 2) signal for the process synchronization.

### Example

WAIT ( S ):

while ( S <= 0 );

S = S - 1;

SIGNAL ( S ):

S = S + 1;

### Summary:

* Process synchronization is the task of coordinating the execution of processes in a way that no two processes can have access to the same shared data and resources.
* Four elements of critical section are 1) Entry section 2) Critical section 3) Exit section 4) Reminder section
* A critical section is a segment of code which can be accessed by a signal process at a specific point of time.
* Three must rules which must enforce by critical section are : 1) Mutual Exclusion 2) Process solution 3)Bound waiting
* Mutual Exclusion is a special type of binary semaphore which is used for controlling access to the shared resource.
* Process solution is used when no one is in the critical section, and someone wants in.
* In bound waiting solution, after a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section.
* Peterson's solution is widely used solution to critical section problems.
* Problems of the Critical Section are also resolved by synchronization of hardware
* Synchronization hardware is not a simple method to implement for everyone, so the strict software method known as Mutex Locks was also introduced.
* Semaphore is another algorithm or solution to the critical section problem.

## Process Scheduling Queues

Process Scheduling Queues help you to maintain a distinct queue for each and every process states and PCBs. All the process of the same execution state are placed in the same queue. Therefore, whenever the state of a process is modified, its PCB needs to be unlinked from its existing queue, which moves back to the new state queue.

Three types of operating system queues are:

1. **Job queue** – It helps you to store all the processes in the system.
2. **Ready queue** – This type of queue helps you to set every process residing in the main memory, which is ready and waiting to execute.
3. **Device queues** – It is a process that is blocked because of the absence of an I/O device.

**Scheduling Objectives**

Here, are important objectives of Process scheduling

* Maximize the number of interactive users within acceptable response times.
* Achieve a balance between response and utilization.
* Avoid indefinite postponement and enforce priorities.
* It also should give reference to the processes holding the key resources.

**Type of Process Schedulers**

A scheduler is a type of system software that allows you to handle process scheduling.

There are mainly three types of Process Schedulers:

1. Long Term
2. Short Term
3. Medium Term

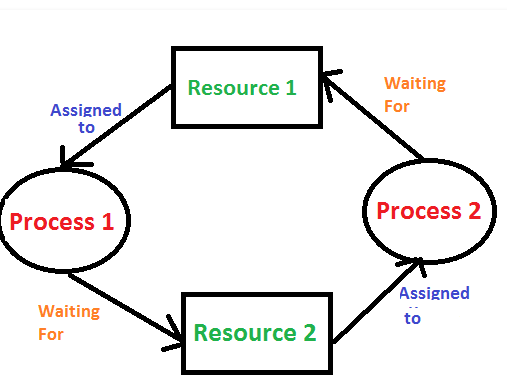
**Long Term Scheduler**

Long term scheduler is also known as a **job scheduler**. This scheduler regulates the program and select process from the queue and loads them into memory for execution. It also regulates the degree of multi-programming.

However, the main goal of this type of scheduler is to offer a balanced mix of jobs, like Processor, I/O jobs., that allows managing multiprogramming.

**Topic № 10**

A process in operating systems uses different resources and uses resources in the following way.   
1) Requests a resource   
2) Use the resource   
2) Releases the resource

***Deadlock***is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.   
Consider an example when two trains are coming toward each other on the same track and there is only one track, none of the trains can move once they are in front of each other. A similar situation occurs in operating systems when there are two or more processes that hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.   
 

**Deadlock can arise if**the **following four conditions hold simultaneously (Necessary Conditions)**  
***Mutual Exclusion:*** One or more than one resource are non-shareable (Only one process can use at a time)   
***Hold and Wait:***A process is holding at least one resource and waiting for resources.   
***No Preemption:*** A resource cannot be taken from a process unless the process releases the resource.   
***Circular Wait:*** A set of processes are waiting for each other in circular form.

**Methods for handling deadlock**   
There are three ways to handle deadlock   
1) Deadlock prevention or avoidance: The idea is to not let the system into a deadlock state.   
One can zoom into each category individually, Prevention is done by negating one of above mentioned necessary conditions for deadlock.   
Avoidance is kind of futuristic in nature. By using strategy of “Avoidance”, we have to make an assumption. We need to ensure that all information about resources which process will need are known to us prior to execution of the process. We use Banker’s algorithm (Which is in-turn a gift from Dijkstra) in order to avoid deadlock.

2) Deadlock detection and recovery: Let deadlock occur, then do preemption to handle it once occurred.

3) Ignore the problem altogether: If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

# What is Banker's Algorithm?

Banker's algorithm is a **deadlock avoidance algorithm**. It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not.

Consider there are n account holders in a bank and the sum of the money in all of their accounts is S. Everytime a loan has to be granted by the bank, it subtracts the **loan amount** from the **total money** the bank has. Then it checks if that difference is greater than S. It is done because, only then, the bank would have enough money even if all the n account holders draw all their money at once.

Banker's algorithm works in a similar way in computers.

Whenever a new process is created, it must specify the maximum instances of each resource type that it needs, exactly.

Let us assume that there are n processes and m resource types. Some data structures that are used to implement the banker's algorithm are:

### 1. Available

It is an **array** of length m. It represents the number of available resources of each type. If Available[j] = k, then there are k instances available, of resource type R(j).

### 2. Max

It is an n x m matrix which represents the maximum number of instances of each resource that a process can request. If Max[i][j] = k, then the process P(i) can request atmost k instances of resource type R(j).

### 3. Allocation

It is an n x m matrix which represents the number of resources of each type currently allocated to each process. If Allocation[i][j] = k, then process P(i) is currently allocated k instances of resource type R(j).

### 4. Need

It is an n x m matrix which indicates the remaining resource needs of each process. If Need[i][j] = k, then process P(i) may need k more instances of resource type R(j) to complete its task.

Need[i][j] = Max[i][j] - Allocation [i][j]

Resource Request Algorithm

This describes the behavior of the system when a process makes a resource request in the form of a request matrix. The steps are:

1. If number of requested instances of each resource is less than the need (which was declared previously by the process), go to step 2.
2. If number of requested instances of each resource type is less than the available resources of each type, go to step 3. If not, the process has to wait because sufficient resources are not available yet.
3. Now, assume that the resources have been allocated. Accordingly do,

Available = Available - Requesti

Allocation(i) = Allocation(i) + Request(i)

Need(i) = Need(i) - Request(i)

This step is done because the system needs to assume that resources have been allocated. So there will be less resources available after allocation. The number of allocated instances will increase. The need of the resources by the process will reduce. That's what is represented by the above three operations.

After completing the above three steps, check if the system is in safe state by applying the safety algorithm. If it is in safe state, proceed to allocate the requested resources. Else, the process has to wait longer.

Safety Algorithm

1. Let Work and Finish be vectors of length **m** and **n**, respectively. Initially,
2. Work = Available
3. Finish[i] =false for i = 0, 1, ... , n - 1.

This means, initially, no process has finished and the number of available resources is represented by the **Available** array.

1. Find an index **i** such that both
2. Finish[i] ==false
3. Needi <= Work

If there is no such i present, then proceed to step 4.

It means, we need to find an unfinished process whose need can be satisfied by the available resources. If no such process exists, just go to step 4.

1. Perform the following:
2. Work = Work + Allocation;
3. Finish[i] = true;

Go to step 2.

When an unfinished process is found, then the resources are allocated and the process is marked finished. And then, the loop is repeated to check the same for all other processes.

1. If Finish[i] == true for all i, then the system is in a safe state.

That means if all processes are finished, then the system is in safe state.

**Lecture 11**

The management of **main memory** is critical. In fact, for many years, the performance

of the *entire* system was directly dependent on two things: How much memory was

available and how that memory was optimized while jobs were being processed.is

**Single-User Contiguous Scheme**

This memory allocation scheme works like this: before execution can begin, each job

or program is loaded in its entirety into memory and allocated as much contiguous

space in memory as it needs. The key words here are *entirety*

and *contiguous*. If the program is too large to fit into the available memory space, it

cannot begin execution.

This scheme demonstrates a significant limiting factor of all computers—they have

only a finite amount of memory. If a program doesn’t fit, then either the size of the

main memory must be increased, or the program must be modified to fit, often by

revising it to be smaller.

Single-user systems in a non-networked environment allocate, to each user, access

to all available main memory for each job, and jobs are processed sequentially, one

after the other. To allocate memory, the amount of work required from the operating

system’s Memory Manager is minimal, as described in these steps:

1. Evaluate the incoming process to see if it is small enough to fit into the

available space. If it is, load it into memory; if not, reject it and evaluate the

next incoming process,

2. Monitor the occupied memory space. When the resident process ends its

execution and no longer needs to be in memory, make the entire amount

of main memory space available and return to Step 1, evaluating the next

incoming process.

Once the program is entirely loaded into memory, it begins its execution and remains

there until execution is complete, either by finishing its work or through the intervention

of the operating system, such as when an error is detected.

One major problem with this type of memory allocation scheme is that it doesn’t support

multiprogramming (multiple jobs or processes occupying memory at the same

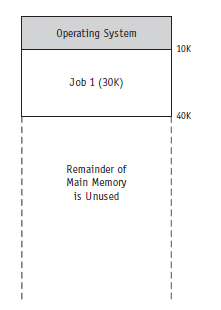
time); it can handle only one at a time. When these single-user configurations were

first made available commercially in the late 1940s and early 1950s, they were used in

research institutions but proved unacceptable for the business community—it wasn’t

cost effective to spend almost $200,000 for a piece of equipment that could be used by

only one person at a time. t

o a *Only one program can fit into memory at a time, even if there is room*

*to accommodate other waiting jobs.*

**Fixed Partitions**

The first attempt to allow for multiprogramming used **fixed partitions** (also known as

static partitions) within main memory—each partition could be assigned to one job.

A system with four partitions could hold four jobs in memory at the same time. One

fact remained the same, however—these partitions were static, so the systems administrator

had to turn off the entire system to reconfigure their sizes, and any job that

couldn’t fit into the largest partition could not be executed.

An important factor was introduced with this scheme: protection of the job’s memory

space. Once a partition was assigned to a job, the jobs in other memory partitions had

to be prevented from invading its boundaries, either accidentally or intentionally. This

problem of partition intrusion didn’t exist in single-user contiguous allocation schemes

because only one job was present in main memory at any given time—only the portion

of main memory that held the operating system had to be protected. However, for

the fixed partition allocation schemes, protection was mandatory for each partition

in main memory. Typically this was the joint responsibility of the hardware of the

computer and of the operating system.

The algorithm used to store jobs in memory requires a few more steps than the one

used for a single-user system because the size of the job must be matched with the size

of the available partitions to make sure it fits completely. (“An Algorithm to Load a

Job in a Fixed Partition” is in Appendix A.) Remember, this scheme also required that

the entire job be loaded into memory before execution could begin.

To do so, the Memory Manager could perform these steps in a two-partition system:

1. Check the incoming job’s memory requirements. If it’s greater than the size of

the largest partition, reject the job and go to the next waiting job. If it’s less

than the largest partition, go to Step 2.

2. Check the job size against the size of the first available partition. If the job is

small enough to fit, see if that partition is free. If it is available, load the job

into that partition. If it’s busy with another job, go to Step 3.

3. Check the job size against the size of the second available partition. If the job is small enough to fit, check to see if that partition is free. If it is available, load

the incoming job into that partition. If not, go to Step 4.

4. Because neither partition is available now, place the incoming job in the

waiting queue for loading at a later time. Return to Step 1 to evaluate the

next incoming job.

tra This partition scheme is more flexible than the single-user scheme because it allows

more than one program to be in memory at the same time. However, it still requires

that the *entire* program be stored *contiguously* and *in memory* from the beginning to

the end of its execution.ffic

co **Dynamic Partitions**ntrolle

r. With the introduction of the **dynamic partition** allocation scheme, memory is allocated to an incoming job in one contiguous block, and each job is given only as much

memory as it requests when it is loaded for processing. Although this is a significant

improvement over fixed partitions because memory is no longer wasted inside each

partition, it introduces another problem.When the proce

ss It works well when the first jobs are loaded. As shown in Figure , a dynamic partition

scheme allocates memory efficiently as each of the first few jobs are loaded, but

when those jobs finish and new jobs enter the system (which are not the same size as

those that just vacated memory), the newer jobs are allocated space in the available

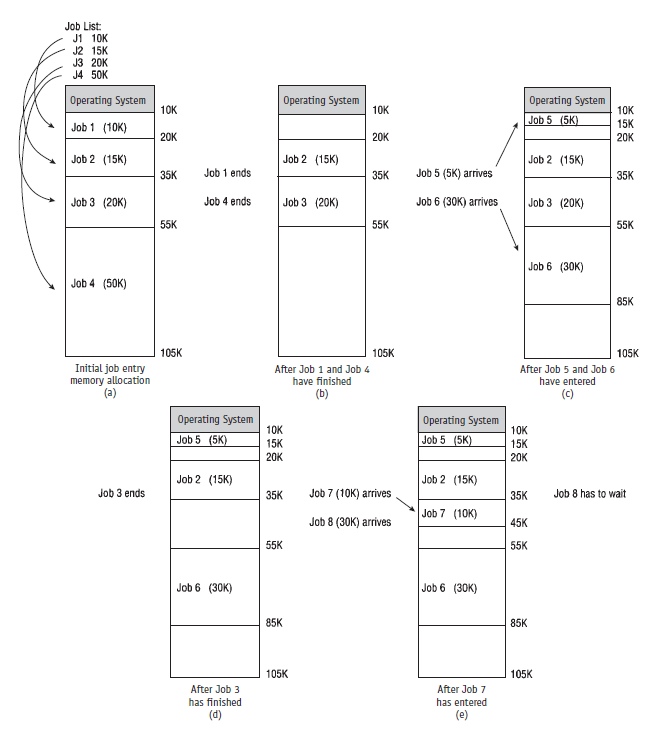
partition spaces on a priority basis. Figure demonstrates first-come, first-served

priority—that is, each job is loaded into the first available partition. Therefore, the

subsequent allocation of memory creates fragments of free memory *between* partitions

of allocated memory. This problem is called **external fragmentation** and, like internal

fragmentation, allows memory to be wasted.is finis

hed, 

*Main memory use during dynamic partition allocation. Five snapshots (a-e) of main memory as eight jobs are submitted for processing and allocated space on the basis of “first come, first served.” Job 8 has to wait (e) even though there’s enough free memory between partitions to accommodate it.*

**Best-Fit and First-Fit Allocation**

For both fixed and dynamic memory allocation schemes, the operating system must

keep lists of each memory location, noting which are free and which are busy. Then,

as new jobs come into the system, the free partitions must be allocated fairly, according

to the policy adopted by the programmers who designed and wrote the operating

system.

Memory partitions may be allocated on the basis of first-fit memory allocation or

best-fit memory allocation. For both schemes, the Memory Manager keeps detailed

lists of the free and busy sections of memory either by size or by location. The **best-fit**

**allocation method keeps** the free/busy lists in order by size, from smallest to largest.

The **first-fit allocation method** keeps the free/busy lists organized by memory locations,

from low-order memory to high-order memory. Each has advantages depending

on the needs of the particular allocation scheme. Best-fit usually makes the best use of

memory space; first-fit is faster.

**Deallocation**

Until now, we’ve considered only the problem of how memory blocks are allocated,

but eventually there comes a time for the release of memory space, called

**deallocation**.

For a fixed partition system, the process is quite straightforward. When the job is completed,

the Memory Manager immediately deallocates it by resetting the status of the

entire memory block from “busy” to “free.” Any code—for example, binary values

with 0 indicating free and 1 indicating busy—may be used, so the mechanical task of

deallocating a block of memory is relatively simple.

A dynamic partition system uses a more complex algorithm (shown in Appendix A)

because it tries to combine free areas of memory whenever possible. Therefore, the

system must be prepared for three alternative situations:

• Case 1: When the block to be deallocated is adjacent to another free block

• Case 2: When the block to be deallocated is between two free blocks

• Case 3: When the block to be deallocated is isolated from other free blocks

Case 1: Joining Two Free Blocks

Using the deallocation algorithm, the system sees that the memory to be released is

next to a free memory block, which starts at location 7800. Therefore, the list must

be changed to reflect the starting address of the new free block, 7600, which was the

address of the first instruction of the job that just released this block. In addition, the

memory block size for this new free space must be changed to show its new size, which

is the combined total of the two free partitions (200 + 5).

Case 2: Joining Three Free Blocks

When the deallocated memory space is between two free memory blocks, the process

is similar.

Using the deallocation algorithm, the system learns that the memory to be deallocated is

between two free blocks of memory. Therefore, the sizes of the three free partitions (20 + 20

+ 205) must be combined and the total stored with the smallest beginning address, 7560.

or w Case 3: Deallocating an Isolated Blockhe

n The third alternative is when the space to be deallocated is isolated from all other

free areas. For this example, we need to know more about how the busy memory

list is configured. To simplify matters, let’s look at the busy list for the memory area

between locations 7560 and 10250. Remember that, starting at 7560, there’s a free

memory block of 245, so the busy memory area includes everything from location

7805 (7560 + 245) to 10250, which is the address of the next free block. the

Using the deallocation algorithm, the system learns that the memory block to be

released is not adjacent to any free blocks of memory; instead it is between two

other busy areas. Therefore, the system must search the table for a null entry.

The scheme presented in this example creates null entries in both the busy and the

free lists during the process of allocation or deallocation of memory. An example

of a null entry occurring as a result of deallocation was presented in Case 2. A null

entry in the busy list occurs when a memory block between two other busy memory

blocks is returned to the free list.

This mechanism ensures that all blocks are entered in the lists according to the beginning address of their memory location from smallest to largest.

When the null entry is found, the beginning memory location of the terminating

job is entered in the beginning address column, the job size is entered under the

memory block size column, and the status is changed from “null entry” to free to

indicate that a new block of memory is available.

**Relocatable Dynamic Partitions**

All of the memory allocation schemes described thus far shared some unacceptable

fragmentation characteristics that had to be resolved as the number of waiting

jobs became unwieldy and demand increased to use all the slivers of memory often

left unused.

The solution to both problems was the development of **relocatable dynamic partitions**.

With this memory allocation scheme, the Memory Manager relocates programs

to gather together all of the empty blocks and compact them to make one block of

memory large enough to accommodate some or all of the jobs waiting to get in.

The **compaction of memory**, sometimes referred to as memory defragmentation, is

performed by the operating system to reclaim fragmented space. Remember our earlier

example of the lending library? If you stopped lending books for a few moments and

rearranged the books in the most effective order, you would be compacting your collection.

But this demonstrates its disadvantage—this is an overhead process that can

take place only while everything else waits.

Compaction isn’t an easy task. Most or all programs in memory must be relocated so

they’re contiguous, and then every address, and every reference to an address, within

each program must be adjusted to account for the program’s new location in memory.

However, all other values within the program (such as data values) must be left alone.

In other words, the operating system must distinguish between addresses and data

values, and these distinctions are not obvious after the program has been loaded into

memory.

To appreciate the complexity of **relocation**, let’s look at a typical program. Remember,

all numbers are stored in memory as binary values (ones and zeros), and in any given

program instruction it’s not uncommon to find addresses as well as data values. For

example, an assembly language program might include the instruction to add the

integer 1 to I. The source code instruction looks like this:

ADDI I, 1

However, after it has been translated into actual code it could be represented like

this (for readability purposes the values are represented here in octal code, not binary

code):

000007 271 01 0 00 000001

The **bounds register** is used to store the highest (or lowest, depending on the specific

system) location in memory accessible by each program. This ensures that during execution,

a program won’t try to access memory locations that don’t belong to it—that

is, those that are out of bounds.

The **relocation register** contains the value that must be added to each address referenced

in the program so that the system will be able to access the correct memory

addresses after relocation. If a program isn’t relocated, the value stored in the program’s

relocation register is zero.

**Paged Memory Allocation**

**Paged memory allocation** is based on the concept of dividing jobs into units of equal

size and each unit is called a **page**. Some operating systems choose a page size that

is the exact same size as a section of main memory, which is called a **page frame.**

Likewise, the sections of a magnetic disk are called **sectors** or blocks. The paged memory

allocation scheme works quite efficiently when the pages, sectors, and page frames

are all the same size. The exact size (the number of bytes that can be stored in each of

them) is usually determined by the disk’s sector size. Therefore, one sector will hold

one page of job instructions (or data) and fit into one page frame of memory, but

because this is the smallest addressable chunk of disk storage, it isn’t subdivided even

for very tiny jobs.

Before executing a program, a basic Memory Manager prepares it by:

1. Determining the number of pages in the program

2. Locating enough empty page frames in main memory

3. Loading all of the program’s pages into those frames

When the program is initially prepared for loading, its pages are in logical sequence—

the first pages contain the first instructions of the program and the last page has the

last instructions. We refer to the program’s instructions as “bytes” or “words.”

The loading process is different from the schemes we studied in Chapter 2 because the

pages do not have to be loaded in adjacent memory blocks. With the paged memory

allocation, they can be loaded in noncontiguous page frames. In fact, each page can be

stored in any available page frame anywhere in main memory.

The primary advantage of storing programs in noncontiguous page frames is that

main memory is used more efficiently because an empty page frame can be used by any

page of any job. In addition, the compaction scheme used for relocatable partitions is

eliminated because there is no external fragmentation between page frames.

However, with every new solution comes a new problem. Because a job’s pages can be

located anywhere in main memory, the Memory Manager now needs a mechanism to

keep track of them—and that means enlarging the size, complexity, and overhead of

the operating system software.

Each active job has its own **Page Map Table (PMT)**, which contains the vital

information for each page—the page number and its corresponding memory address

of the page frame. Actually, the PMT includes only one entry per page. The page

numbers are sequential (Page 0, Page 1, Page 2, and so on), so it isn’t necessary to list

each page number in the PMT. The first entry in the PMT always lists the page frame

memory address for Page 0, the second entry is the address for Page 1, and so on.

A simple **Memory Map Table (MMT)** has one entry for each page frame and shows its

location and its free/busy status.

At compilation time, every job is divided into pages. Using Job 1 from Figure,

we can see how this works:

• Page 0 contains the first hundred bytes.

• Page 1 contains the second hundred bytes.

• Page 2 contains the third hundred bytes.

• Page 3 contains the last 50 bytes.

As you can see, the program has 350 bytes; but when they are stored, the system numbers

them starting from 0 through 349. Therefore, the system refers to them as Byte 0

through Byte 349.

How far away is a certain byte from the beginning of its page frame? This value is

called the **displacement** (also called the offset) and it is a relative factor that’s used to

locate one certain byte within its page frame.

Here’s how it works. In the simplified example shown in Figure 3.2, Bytes 0, 100, 200,

and 300 are the first bytes for pages 0, 1, 2, and 3, respectively, so each has a displacement

of zero. To state it another way, the distance from the beginning of the page

frame to Byte 200 is zero bytes. Likewise, if the Memory Manager needs to access

Byte 314, it can first go to the page frame containing Page 3 and then go to Byte 14

(the fifteenth) in that page frame.

Because the first byte of each page has a displacement of zero, and the last byte has

a displacement of 99, the operating system can access the correct bytes by using its

relative position from the beginning of its page frame.

A huge advantage of a paging scheme is that it allows jobs to be allocated in noncontiguous

memory locations, allowing memory to be used more efficiently.

However, there are disadvantages—overhead is increased and internal fragmentation is still a problem, although it occurs only in the last page of each job. The key to the success of this scheme is the size of the page. A page size that is too small will generate very long PMTs (with a corresponding increase in overhead), while a page size that is too large will result in excessive

internal fragmentation. Determining the best page size is an important policy decision—

there are no hard and fast rules that will guarantee optimal use of resources, and it is a

problem we’ll see again as we examine other paging alternatives. The best size depends

on the nature of the jobs being processed and on the constraints placed on the system.

**Demand Paging Memory Allocation**

**Demand paging** introduced the concept of loading only a part of the program into

memory for processing. It was the first widely used scheme that removed the restriction of having the entire job in memory from the beginning to the end of its processing.

With demand paging, jobs are still divided into equally-sized pages that initially reside

in secondary storage. When the job begins to run, its pages are brought into memory

only as they are needed, and if they’re never needed, they’re never loaded.

Demand paging takes advantage of the fact that programs are written sequentially so

that while one section, or module, is processed, other modules may be idle. Not all the

pages are accessed at the same time, or even sequentially. For example:

• Instructions to handle errors are processed only when a specific error is detected

during execution. For instance, these instructions can indicate that input data was

incorrect or that a computation resulted in an invalid answer. If no error occurs,

and we hope this is generally the case, these instructions are never processed and

never need to be loaded into memory.

• Many program modules are mutually exclusive. For example, while the program is

being loaded (when the input module is active), then the processing module is inactive

because it is generally not performing calculations during the input stage. Similarly, if

the processing module is active, then the output module (such as printing) may be idle.

• Certain program options are either mutually exclusive or not always accessible. For

example, when a program gives the user several menu choices

If the user selects the first option (FILE), then the module with those program instructions is the only one that is being used, so that is the only module that needs to be in memory at this time. The other modules remain in secondary storage until they are “called.” Many tables are

assigned a large fixed amount of address space even though only a fraction of the table

is actually used. For example, a symbol table might be prepared to handle 100 symbols.

If only 10 symbols are used, then 90 percent of the table remains unused.

One of the most important innovations of demand paging was that it made virtual

memory feasible.

How and when the pages are passed (also called swapped) between main memory

and secondary storage depends on predefined policies that determine when to make

room for needed pages and how to do so. The operating system relies on tables (such

as the Job Table, the Page Map Tables, and the Memory Map Table) to implement

the algorithm. These tables are basically the same as for paged memory allocation.

With demand paging, there are three new fields for each page in each PMT: one to

determine if the page being requested is already in memory, a second to determine if

the page contents have been modified while in memory, and a third to determine if the

page has been referenced most recently. The fourth field, which we discussed earlier in

this chapter, shows the page frame number.

****

The memory field tells the system where to find each page. If it is already in memory

(shown here with a Y), the system will be spared the time required to bring it from secondary

storage. As one can imagine, it’s faster for the operating system to scan a table located in main memory than it is to retrieve a page from a disk or other secondary

storage device.

The modified field, noting whether or not the page has been changed, is used to save

time when pages are removed from main memory and returned to secondary storage.

If the contents of the page haven’t been modified, then the page doesn’t need to be

rewritten to secondary storage. The original, the one that is already there, is correct.

The referenced field notes any recent activity and is used to determine which pages

show the most processing activity and which are relatively inactive. This information

is used by several page-swapping policy schemes to determine which pages should

remain in main memory and which should be swapped out when the system needs to

make room for other pages being requested.

The last field shows the page frame number for that page.

For example, in Figure the number of total job pages is 15, and the number of total

page frames available to jobs is 12. (The operating system occupies the first four of the

16 page frames in main memory.)

Assuming the processing status illustrated in Figure , what happens when Job 4

requests that Page 3 be brought into memory, given that there are no empty page

frames available?

To move in a new page, one of the resident pages must be swapped back into secondary

storage. Specifically, that includes copying the resident page to the disk (if it was

modified) and writing the new page into the newly available page frame.

The hardware components generate the address of the required page, find the page

number, and determine whether it is already in memory. Refer to the “Hardware

Instruction Processing Algorithm” shown in Appendix A for more information.

If the Algorithm finds that the page is not in memory, then the operating system

software takes over. The section of the operating system that resolves these problems

is called the **page fault handler**, and an algorithm to perform that task is also

described in Appendix A. The page fault handler determines whether there are empty

page frames in memory so that the requested page can be immediately copied from

secondary storage. If all page frames are busy, then the page fault handler must decide

which page will be swapped out. (This decision is determined by the predefined policy

for page removal.) Then the swap is made.

How many tables are changed because of this page swap? Immediately, three tables

must be updated: the Page Map Tables for both jobs (the PMT with the page that was

swapped out and the PMT with the page that was swapped in) as well as the Memory

Map Table. Finally, the instruction that was interrupted is resumed, and processing

continues.

Although demand paging is a good solution to improve memory use, it is not free of problems.

When there is an excessive amount of **page swapping** between main memory and

secondary storage, the operation becomes inefficient. This phenomenon is called **thrashing**.

Thrashing is similar to the problem that students face when comparing explanations of

a complex problem in two different textbooks. The amount of time you spend switching

back and forth between the two books could cause you to spend more time figuring

out where you left off in each book than you spend actually solving the issue at hand.

Caused when pages are frequently removed from memory but are called back shortly

thereafter, thrashing uses a great deal of the computer’s time and accomplishes very

little. Thrashing can occur across jobs, when a large number of jobs are vying for a

relatively low number of free pages (that is, the ratio of job pages to free memory page

frames is high), or it can happen within a job—for example, in loops that cross page

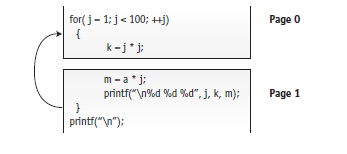
boundaries. We can demonstrate this with a simple example. Suppose that a loop to

perform a certain sequence of steps begins at the bottom of one page and is completed

at the top of the next page, as demonstrated in the C program in Figure . This

requires that the command sequence go from Page 0 to Page 1 to Page 0 to Page 1 to

Page 0, again and again, until the loop is finished.



*An example of demand paging that causes a page swap each time the loop is executed and results in thrashing. If only a single page frame is available, this program will have one*

*page fault each time the loop is executed.*

If there is only one empty page frame available, the first page is loaded into memory

and execution begins. But, in this example, after executing the last instruction on Page

0, that page is swapped out to make room for Page 1. Now execution can continue with

the first instruction on Page 1, but at the “}” symbol, Page 1 must be swapped out so

Page 0 can be brought back in to continue the loop. Before this program is completed,

swapping will have occurred 199 times (unless another page frame becomes free so

both pages can reside in memory at the same time). A failure to find a page in memory

is often called a **page fault**; this example would generate 199 page faults (and swaps).

In such extreme cases, the rate of useful computation could be degraded by at least a

factor of 100. Ideally, a demand paging scheme is most efficient when programmers

are aware of the page size used by their operating system and are careful to design their

programs to keep page faults to a minimum; but in reality, this is not often feasible.

**Lecture 12**

As we just learned, the policy that selects the page to be removed, the **page replacement**

**policy**, is crucial to the efficiency of the system, and the algorithm to do that must be

carefully selected.

Several such algorithms exist, and it is a subject that enjoys a great deal of theoretical

attention and research. Two of the most well-known algorithms are first-in first-out

and least recently used. The **first-in first-out (FIFO) policy** is based on the assumption

that the best page to remove is the one that has been in memory the longest. The **least**

**recently used (LRU) policy** chooses the page least recently accessed to be swapped out.

To illustrate the difference between FIFO and LRU, let us imagine a dresser drawer

filled with your favorite sweaters. The drawer is full, but that didn’t stop you from

buying a new sweater. Now you have to put it away. Obviously it won’t fit in your

sweater drawer unless you take something out—but which sweater should you move

to the storage closet? Your decision will be based on a “sweater removal policy.”

You could take out your oldest sweater (the one that was first in), figuring that you

probably won’t use it again—hoping you won’t discover in the following days that it

is your most used, most treasured possession. Or, you could remove the sweater that

you haven’t worn recently and has been idle for the longest amount of time (the one

that was least recently used). It is readily identifiable because it is at the bottom of the

drawer. But just because it hasn’t been used recently doesn’t mean that a once-a-year

occasion won’t demand its appearance soon!

What guarantee do you have that once you have made your choice you won’t be trekking

to the storage closet to retrieve the sweater you stored there yesterday? You could

become a victim of thrashing, going back and forth to swap out sweaters.

Which is the best policy? It depends on the weather, the wearer, and the wardrobe.

Of course, one option is to get another drawer. For an operating system (or a computer),

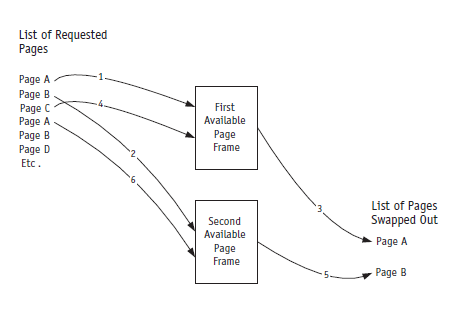
this is the equivalent of adding more accessible memory.

**First-In First-Out**

The first-in first-out (FIFO) page replacement policy will remove the pages that have

been in memory the longest, that is, those that were “first in.” The process of swapping

pages is illustrated in Figure .



• Step 1: Page A is moved into the first available page frame.

• Step 2: Page B is moved into the second available page frame.

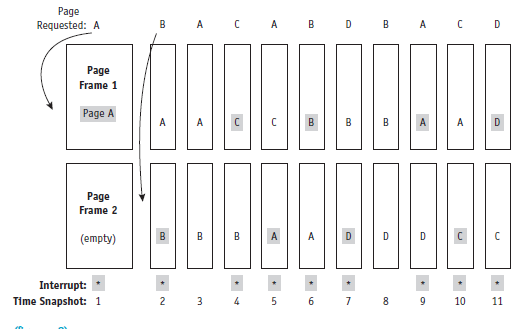
• Step 3: Page A is swapped into secondary storage.

•Step 4: Page C is moved into the first available page frame.

• Step 5: Page B is swapped into secondary storage.

• Step 6: Page A is moved into the second available page frame.

Figure



demonstrates how the FIFO algorithm works by following a job with four

pages (A, B, C, and D) as it is processed by a system with only two available page

frames. The job needs to have its pages processed in the following order: A, B, A, C,

A, B, D, B, A, C, D.

When both page frames are occupied, each new page brought into memory will cause

an interrupt and a page swap into secondary storage. A page interrupt, which we identify

with an asterisk (\*), is generated anytime a page needs to be loaded into memory,

whether a page is swapped out or not. Then we count the number of page interrupts

and compute the failure rate and the success rate.

The efficiency of this configuration is dismal—due to the limited number of page

frames available, there are 9 page interrupts out of 11 page requests. To calculate the

failure rate, we divide the number of interrupts by the number of page requests:

Failure-Rate = Number\_of\_Interrupts / Page\_Requests\_Made

The failure rate of this system is 9/11, which is 82 percent. Stated another way, the

success rate is 2/11, or 18 percent. A failure rate this high is usually unacceptable.

We are not saying FIFO is bad. We chose this example to show how FIFO works, not to

diminish its appeal as a page replacement policy. The high failure rate here is caused by

both the limited amount of memory available and the order in which pages are requested

by the program. Because the job dictates the page order, that order can’t be changed by

the system, although the size of main memory can be changed. However, buying more

memory may not always be the best solution—especially when you have many users

and each one wants an unlimited amount of memory.

When both page frames are occupied, each new page brought into memory will cause

an interrupt and a page swap into secondary storage. A page interrupt, which we identify

with an asterisk (\*), is generated anytime a page needs to be loaded into memory,

whether a page is swapped out or not. Then we count the number of page interrupts

and compute the failure rate and the success rate.

The efficiency of this configuration is dismal—due to the limited number of page

frames available, there are 9 page interrupts out of 11 page requests. To calculate the

failure rate, we divide the number of interrupts by the number of page requests:

Failure-Rate = Number\_of\_Interrupts / Page\_Requests\_Made

The failure rate of this system is 9/11, which is 82 percent. Stated another way, the

success rate is 2/11, or 18 percent. A failure rate this high is usually unacceptable.

We are not saying FIFO is bad. We chose this example to show how FIFO works, not to

diminish its appeal as a page replacement policy. The high failure rate here is caused by

both the limited amount of memory available and the order in which pages are requested

by the program. Because the job dictates the page order, that order can’t be changed by

the system, although the size of main memory can be changed. However, buying more

memory may not always be the best solution—especially when you have many users

and each one wants an unlimited amount of memory. As explained later in this chapter,

there is no guarantee that buying more memory will always result in better performance.

**Least Recently Used**

The least recently used (LRU) page replacement policy swaps out the pages that show

the least recent activity, figuring that these pages are the least likely to be used again

in the immediate future. Conversely, if a page is used, it is likely to be used again soon;

this is based on the theory of locality, explained later in this chapter.

To see how it works, let us follow the same job in Figure but using the LRU policy.

The results are shown in Figure . To implement this policy, a queue of requests is

kept in FIFO order, a time stamp of when the page entered the system is saved, or a

mark in the job’s PMT is made periodically.

The efficiency of the configuration for this example is only slightly better than with

FIFO. Here, there are 8 page interrupts out of 11 page requests, so the failure rate

is 8/11, or 73 percent. In this example, an increase in main memory by one page frame

would increase the success rate of both FIFO and LRU. However, we can’t conclude

on the basis of only one example that one policy is better than the other. In fact, LRU

will never cause an increase in the number of page interrupts.

The same cannot be said about FIFO. It has been shown that *under certain circumstances*

adding more memory can, *in rare cases,* actually cause an increase in page

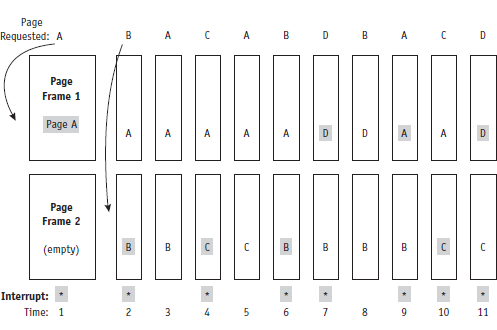
interrupts when using a FIFO policy. This was first demonstrated by Laszlo Belady

in 1969 and is called the **FIFO Anomaly** or the **Belady Anomaly**. Although this is

an unusual occurrence, the fact that it exists, coupled with the fact that pages are

removed regardless of their activity, has removed FIFO

from the most favored policy position.



**Segmented/Demand Paged Memory Allocation**

The **segmented/demand paged memory allocation** scheme evolved from the two we

have just discussed. It is a combination of segmentation and demand paging, and it

offers the logical benefits of segmentation, as well as the physical benefits of paging.

The logic isn’t new. The algorithms used by the demand paging and segmented

memory management schemes are applied here with only minor modifications.

This allocation scheme doesn’t keep each segment as a single contiguous unit, but subdivides

it into pages of equal size that are smaller than most segments and more easily

manipulated than whole segments. Therefore, many of the problems of segmentation

(compaction, external fragmentation, and secondary storage handling) are removed

because the pages are of fixed length.

This scheme, illustrated in Figure , requires four types of tables:

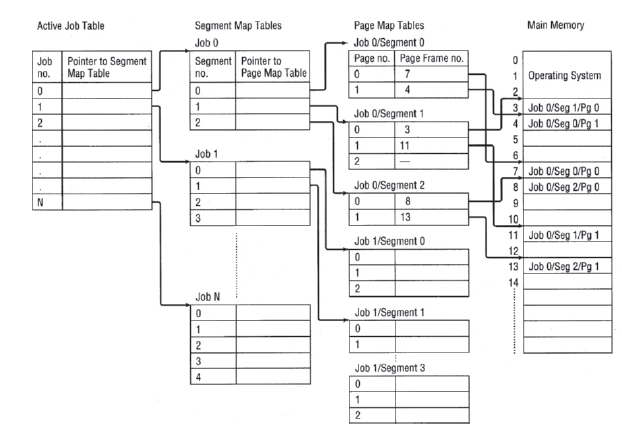
• The Job Table lists every job in process (there’s one JT for the whole system).

• The Segment Map Table lists details about each segment (one SMT for each job).

• The Page Map Table lists details about every page (one PMT for each segment).

• The Memory Map Table monitors the allocation of the page frames in main

memory (there’s one for the whole system).



**(figure )**

*How the Job Table, Segment Map Table, Page Map Table, and main memory interact in a segment/paging*

*scheme.*

Notice that the tables in Figure have been simplified. The SMT actually includes

additional information regarding protection (such as the authority to read, write, and

execute parts of the file), and it also determines which specific users or processes are

allowed to access that segment. In addition, the PMT includes indicators of the page’s

status, last modification, and last reference.

Here’s how it works: To access a certain location in memory, the system locates the

address, which is composed of three entries: the segment number, the page number within

that segment, and the displacement within that page. Therefore, it is a three-dimensional

addressing scheme with the SEGMENT\_NUMBER, the PAGE\_NUMBER, and the

DISPLACEMENT.

The major disadvantages of this memory allocation scheme are twofold: the overhead

that is required to manage the tables (Segment Map Tables and the Page Map Tables),

and the time required to reference them. To minimize the number of references, many

systems take advantage of associative memory to speed up the process.

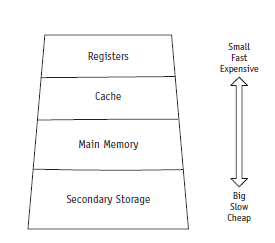
**Associative memory** is a name given to several registers that are allocated to each job

that is active. Their task is to associate several segment and page

numbers belonging to the job being processed with their main memory addresses.

These associative registers are hardware, and the exact number of registers varies from

system to system.



**figure**

*Hierarchy of data storage options, from most expensive at the top to cheapest at the bottom*

To appreciate the role of associative memory, it is important to understand how the

system works with segments and pages. In general, when a job is allocated to the CPU,

its Segment Map Table is loaded into main memory, while its Page Map Tables are

loaded only as they are needed. When pages are swapped between main memory and

secondary storage, all tables are updated.

Here is a typical procedure:

1. When a page is first requested, the job’s SMT is searched to locate its PMT.

2. The PMT is loaded (if necessary) and searched to determine the page’s location

in memory.

a. If the page isn’t in memory, then a page interrupt is issued, the page is

brought into memory, and the table is updated. (As the example indicates,

loading the PMT can cause a page interrupt, or fault, as well.) This process

is just as tedious as it sounds, but it gets easier.

b. Since this segment’s PMT (or part of it) now resides in memory, any other

requests for pages within this segment can be quickly accommodated

because there is no need to bring the PMT into memory. However,

accessing these tables (SMT and PMT) is time consuming.

This illustrates the problem addressed by associative memory, which stores the information

related to the most-recently-used pages. Then when a page request is issued,

two searches begin at once—one through the segment and page map tables and one

through the contents of the associative registers.

If the search of the associative registers is successful, then the search through the tables

is abandoned and the address translation is performed using the information in the

associative registers. However, if the search of associative memory fails, no time is lost

because the search through the SMT and PMTs had already begun (in this schema).

When this search is successful and the main memory address from the PMT has been

determined, the address is used to continue execution of the program while the reference

is also stored in one of the associative registers. If all of the associative registers

are full, then an LRU (or other) algorithm is used and the least-recently-referenced

associative register holds the information on this requested page.

For example, a system with eight associative registers per job will use them to store the

SMT entries and PMT entries for the last eight pages referenced by that job. When an

address needs to be translated from segment and page numbers to a memory location,

the system will look first in the eight associative registers. If a match is found, the

memory location is taken from the associative register; if there is no match, then the

SMT and PMTs will continue to be searched and the new information will be stored

in one of the eight registers as a result. It’s worth noting that in some systems the

searches do not run in parallel, but the search of the SMT and PMTs is performed

after a search of the associative registers fails.

If a job is swapped out to secondary storage during its execution, then all of the information

stored in its associative registers is saved, as well as the current PMT and SMT,

so the displaced job can be resumed quickly when the CPU is reallocated to it. The

primary advantage of a large associative memory is increased speed. The disadvantage

is the high cost of the complex hardware required to perform the parallel searches.

**Topic № 13**

A **file** can be "free formed", indexed or structured collection of related bytes having meaning only to the one who created it. Or in other words an entry in a directory is the file. The file may have attributes like name, creator, date, type, permissions etc.

## File Structure

A file has various kinds of structure. Some of them can be :

* **Simple Record Structure** with lines of fixed or variable lengths.
* **Complex Structures** like formatted document or reloadable load files.
* **No Definite Structure** like sequence of words and bytes etc.

### Attributes of a File

Following are some of the attributes of a file :

* **Name**. It is the only information which is in human-readable form.
* **Identifier**. The file is identified by a unique tag(number) within file system.
* **Type**. It is needed for systems that support different types of files.
* **Location**. Pointer to file location on device.
* **Size**. The current size of the file.
* **Protection**. This controls and assigns the power of reading, writing, executing.
* **Time, date, and user identification**. This is the data for protection, security, and usage monitoring.

## File Access Methods

The way that files are accessed and read into memory is determined by Access methods. Usually a single access method is supported by systems while there are OS's that support multiple access methods.

#### 1. Sequential Access

* Data is accessed one record right after another is an order.
* Read command cause a pointer to be moved ahead by one.
* Write command allocate space for the record and move the pointer to the new End Of File.
* Such a method is reasonable for tape.

#### 2. Direct Access

* This method is useful for disks.
* The file is viewed as a numbered sequence of blocks or records.
* There are no restrictions on which blocks are read/written, it can be dobe in any order.
* User now says "read n" rather than "read next".
* "n" is a number relative to the beginning of file, not relative to an absolute physical disk location.

#### 3. Indexed Sequential Access

* It is built on top of Sequential access.
* It uses an Index to control the pointer while accessing files.

### What is a Directory?

Information about files is maintained by Directories. A directory can contain multiple files. It can even have directories inside of them. In Windows we also call these directories as folders.

Following is the information maintained in a directory :

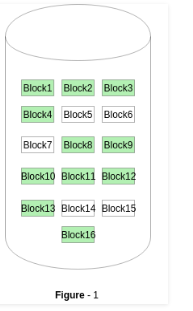
* **Name** : The name visible to user.
* **Type** : Type of the directory.
* **Location** : Device and location on the device where the file header is located.
* **Size** : Number of bytes/words/blocks in the file.
* **Position** : Current next-read/next-write pointers.
* **Protection** : Access control on read/write/execute/delete.
* **Usage** : Time of creation, access, modification etc.
* **Mounting** : When the root of one file system is "grafted" into the existing tree of another file system its called Mounting.

**Topic № 14**

The system keeps tracks of the free disk blocks for allocating space to files when they are created. Also, to reuse the space released from deleting the files, free space management becomes crucial. The system maintains a free space list which keeps track of the disk blocks that are not allocated to some file or directory. The free space list can be implemented mainly as:

1. **Bitmap or Bit vector –**  
   A Bitmap or Bit Vector is series or collection of bits where each bit corresponds to a disk block. The bit can take two values: 0 and 1: *0 indicates that the block is allocated* and 1 indicates a free block.

The given instance of disk blocks on the disk in Figure 1 (where green blocks are allocated) can be represented by a bitmap of 16 bits as:**0000111000000110**.



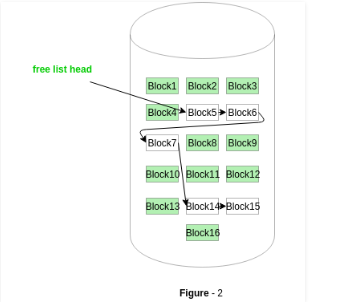
**Advantages –**

* Simple to understand.
* Finding the first free block is efficient. It requires scanning the words (a group of 8 bits) in a bitmap for a non-zero word. (A 0-valued word has all bits 0). The first free block is then found by scanning for the first 1 bit in the non-zero word.

The block number can be calculated as:  
*(number of bits per word) \*(number of 0-values words) + offset of bit first bit 1 in the non-zero word*.

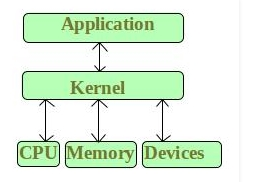
1. For the *Figure-1*, we scan the bitmap sequentially for the first non-zero word.  
   The first group of 8 bits (00001110) constitute a non-zero word since all bits are not 0. After the non-0 word is found, we look for the first 1 bit. This is the 5th bit of the non-zero word. So, offset = 5.  
   Therefore, the first free block number = 8\*0+5 = 5.
2. **Linked List –**  
   In this approach, the free disk blocks are linked together i.e. a free block contains a pointer to the next free block. The block number of the very first disk block is stored at a separate location on disk and is also cached in memory.
3. **Grouping –**  
   This approach stores the address of the free blocks in the first free block. The first free block stores the address of some, say n free blocks. Out of these n blocks, the first n-1 blocks are actually free and the last block contains the address of next free n blocks.  
   An **advantage** of this approach is that the addresses of a group of free disk blocks can be found easily.
4. **Counting –**  
   This approach stores the address of the first free disk block and a number n of free contiguous disk blocks that follow the first block.  
   Every entry in the list would contain:
   1. Address of first free disk block
   2. A number n

For example, *in Figure-1*, the first entry of the free space list would be: ([Address of Block 5], 2), because 2 contiguous free blocks follow block 5.



**Topic № 15**

**Kernel** is the core part of an operating system which manages system resources. It also acts like a bridge between application and hardware of the computer. It is one of the first programs loaded on start-up (after the Bootloader).



**Kernel mode and User mode of CPU operation**  
The CPU can execute certain instruction only when it is in the kernel mode. These instruction are called privilege instruction. They allow implementation of special operation whose execution by the user program could interface with the functioning of operating system or activity of another user program. For example, instruction for managing memory protection.

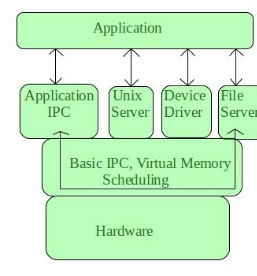
* The operating system puts the CPU in kernel mode when it is executing in the kernel so, that kernel can execute some special operation.
* The operating system puts the CPU in user mode when a user program is in execution so, that user program cannot interface with the operating system program.
* User-level instruction does not require special privilege. Example are ADD,PUSH,etc.

The concept of modes can be extended beyond two, requiring more than a single mode bit CPUs that support virtualization use one of these extra bits to indicate when the virtual machine manager, VMM, is in control of the system. The VMM has more privileges than ordinary user programs, but not so many as the full kernel.

System calls are typically implemented in the form of software interrupts, which causes the hardware’s interrupt handler to transfer control over to an appropriate interrupt handler, which is part of the operating system, switching the mode bit to kernel mode in the process. The interrupt handler checks exactly which interrupt was generated, checks additional parameters ( generally passed through registers ) if appropriate, and then calls the appropriate kernel service routine to handle the service requested by the system call.

User programs’ attempts to execute illegal instructions ( privileged or non-existent instructions ), or to access forbidden memory areas, also generate software interrupts, which are trapped by the interrupt handler and control is transferred to the OS, which issues an appropriate error message, possibly dumps data to a log ( core ) file for later analysis, and then terminates the offending program.

**What is Microkernel?**  
Microkernel is one of the classification of the kernel. Being a kernel it manages all system resources. But in a microkernel, the **user services** and **kernel services** are implemented in different address space. The user services are kept in **user address space**, and kernel services are kept under **kernel address space**, thus also reduces the size of kernel and size of operating system as well.



It provides minimal services of process and memory management. The communication between client program/application and services running in user address space is established through message passing, reducing the speed of execution microkernel. The Operating System **remains unaffected** as user services and kernel services are isolated so if any user service fails it does not affect kernel service. Thus it adds to one of the advantages in a microkernel. It is easily **extendable** i.e. if any new services are to be added they are added to user address space and hence requires no modification in kernel space. It is also portable, secure and reliable.

**Microkernel Architecture –**  
Since kernel is the core part of the operating system, so it is meant for handling the most important services only. Thus in this architecture only the most important services are inside kernel and rest of the OS services are present inside system application program. Thus users are able to interact with those not-so important services within the system application. And the microkernel is solely responsible for the most important services of operating system they are named as follows:

* Inter process-Communication
* Memory Management
* CPU-Scheduling

**Advantages of Microkernel –**

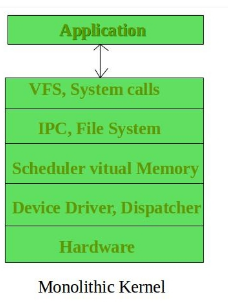
* The architecture of this kernel is small and isolated hence it can function better.
* Expansion of the system is easier, it is simply added in the system application without disturbing the kernel.

**Eclipse IDE** is a good example of Microkernel Architecture.

Apart from microkernel, **Monolithic Kernel** is another classification of Kernel. Like microkernel this one also manages system resources between application and hardware, but **user services** and **kernel services** are implemented under same address space. It increases the size of the kernel, thus increases size of operating system as well.

This kernel provides CPU scheduling, memory management, file management and other operating system functions through system calls. As both services are implemented under same address space, this makes operating system execution faster.

Below is the diagrammatic representation of Monolithic Kernel:



If any service fails the entire system crashes, and it is one of the drawbacks of this kernel. The entire operating system needs modification if user adds a new service.

**Advantages of Monolithic Kernel –**

* One of the major advantage of having monolithic kernel is that it provides CPU scheduling, memory management, file management and other operating system functions through system calls.
* The other one is that it is a single large process running entirely in a single address space.
* It is a single static binary file. Example of some Monolithic Kernel based OSs are: Unix, Linux, Open VMS, XTS-400, z/TPF.

**Disadvantages of Monolithic Kernel –**

* One of the major disadvantage of monolithic kernel is that, if anyone service fails it leads to entire system failure.
* If user has to add any new service. User needs to modify entire operating system.

**Key differences between Monolithic Kernel and Microkernel –**

