(Types of Operating Systems)

The Evolution of Operating Systems:

To understand the key requirements for an operating system and the significance of the major features of an operating system, it is useful to consider how operating systems have evolved over the years.

① Serial processing:

with the Earliest Computers, from the late 1940s to the mid-1950s, the Programmer interacted directly with the Computer hardware, These machines there was no operating System. were run from a Console, Consisting of display lights, toggle switches, some form of input device, and a printer.

Programs in machine code were loaded via the input device (E.g. a card reader).

If an Error halted the program, the Error condition was indicated by the lights.

The programmer could proceed to Examine processor registers and main memory to determine the cause of the Error.

If the program proceeded to a normal completion, the output appeared on the printer.

These Early systems presented two main problems:

→ Scheduling: Most installations used a hardcopy sign-up sheet to reserve machine time. A user might sign up for an hour and finish in 45 minutes, this would result in wasted computer idle time.
(Or), The user might run into problems, and be not finish in the allotted time, and be forced to stop before resolving the problem.

→ Setup time: A single program, called a job, could involve loading the compiler plus the high-level language program (source program) into memory, saving the compiled program (object program) and then loading and linking together the object program and common functions.
Each of these steps could involve mounting or dismounting tapes, or setting up card decks. If an Error occured, the user had to go back to the beginning of the setup sequence. Thus a considerable amount of time was spent just in setting up the program to run.

This mode of operation can be termed as Serial Processing, reflecting the fact that users have access to the Computer in Series.

After some time, various System Software tools were developed to make serial processing more Efficient.

these include libraries of Common functions, linkers, loaders, debuggers, and I/O driver routines that were available as Common Software for all users.

② Simple Batch Systems:

Early machines were very Expensive, and therefore it was important to maximize machine utilization. The wasted time due to Scheduling and setup time was Unaceptable

To improve utilization, the concept of a batch operating system was developed.
(Batch is defined as a group of jobs with similar needs.)

The central idea behind the simple batch processing scheme was the use of a piece of Software known as the monitor.

with the use of this type of operating system, the user no longer has direct access to the machine. Rather, the user submits the job on cards or tape to a computer operator, who batches the jobs togetter sequentially and places the Entine batch on an input device, for use by the monitor.

Each program is constructed to branch back to the monitor when it completes processing, at which point the monitor would automatically begin loading the next program.

In this Execution Environment, CPU is often idle, because the speed of mechanical I/O devices is slower than Electronic device CPU.

In order to overcome the problem of speed mismatch the concept of spooling was introduced. After sometimes, improvement in technology and introduction of disks resulted in faster I/O devices. CPU speed is increased even a greater Extent. so the problem was not resolved.

The introduction of disk technology allows the operating system to keep all jobs on a disk. rather than on a serial card reader. with direct access on jobs operating system performs jobs scheduling to accomplish its task Efficiently.

Spooling: (Simultaneous peripheral operations online) This 'absorbs' surplus processor time by performing I/O transfer for other jobs. I/O data routed via disk files so that these jobs were only required to communicate with disk systems. which are more faster.

Even with the automatic job sequencing provided by a simple batch operating system, the processor is often idle.

The problem is that I/O devices are slow compared to the processor.

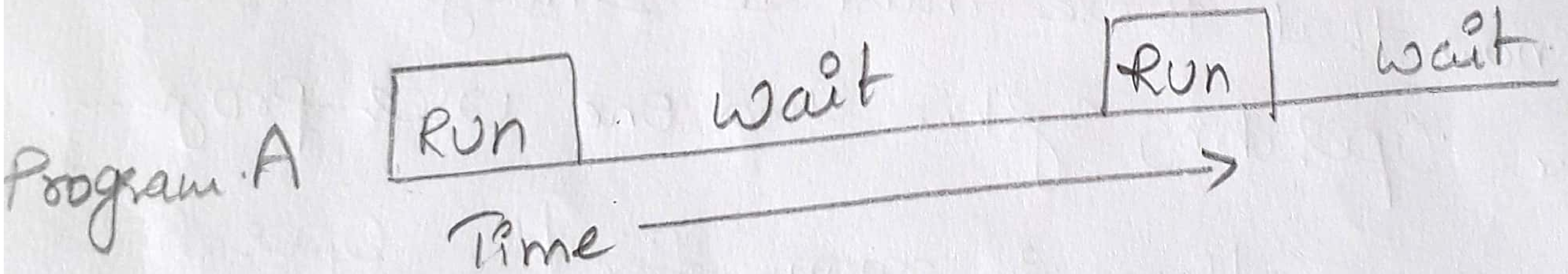
Eg: System Utilization:

Read one record from file	0.0015 seconds
Execute 100 instructions	0.0001 seconds
write one record to file	0.0015 seconds
TOTAL	0.0031 seconds

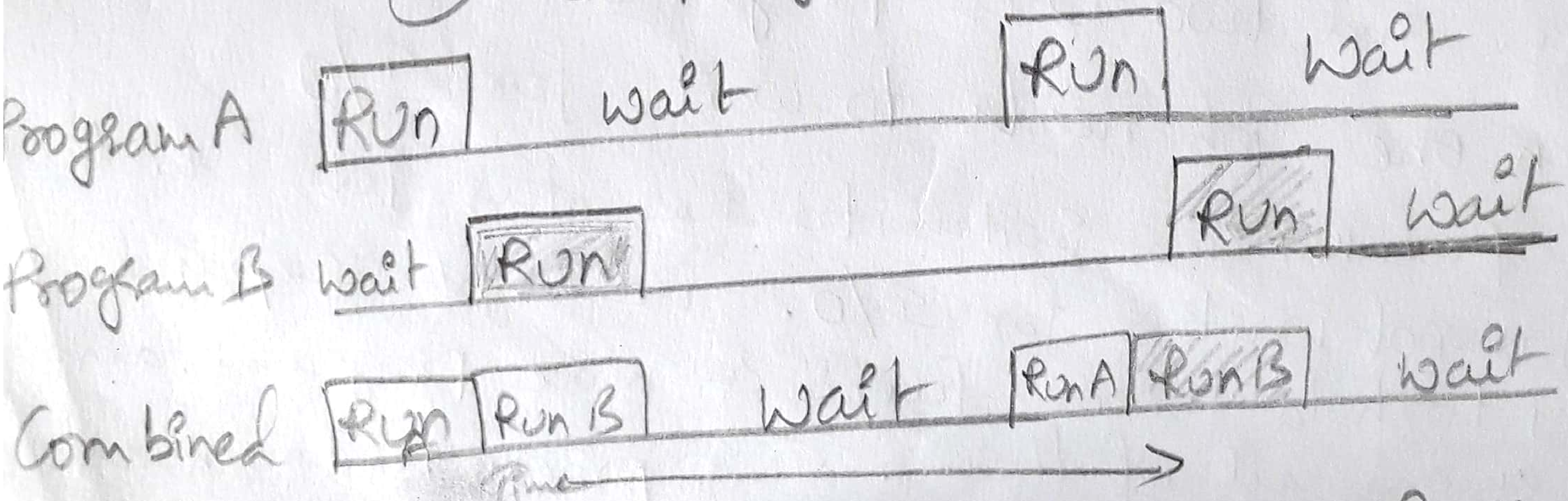
$$\text{Percent CPU Utilization} = \frac{0.0001}{0.0031} = 0.032$$

$$= 3.2\%$$

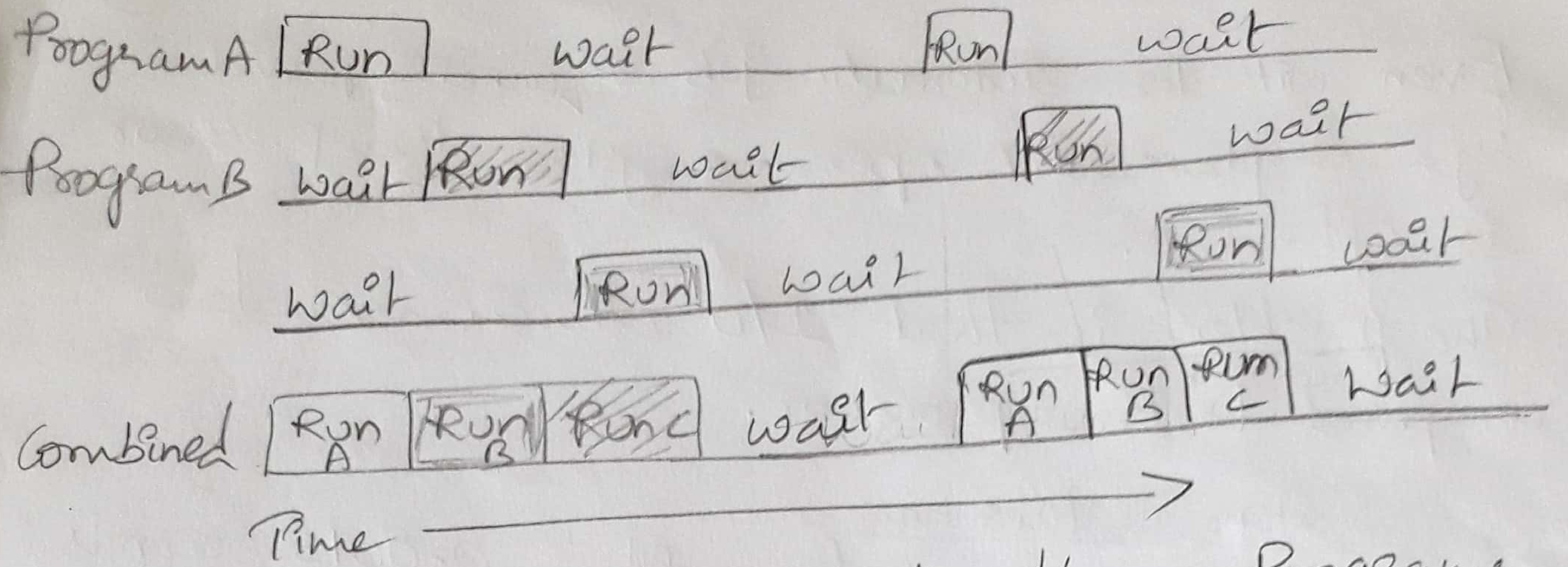
3) Multiprogrammed Batch System:



a) Uniprogramming



b) Multiprogramming with two Programs.



(C) Multiprogramming with three Programs.

The above Multiprogramming Example figure illustrates the Situation, where we have a single program, referred to as Uniprogramming. The processor Spends a certain amount of time Executing, until it reaches an I/O Instruction. It must then wait until that I/O Instruction Concludes before proceeding.

→ we know that there must be Enough memory to hold the Operating system and one user program.

→ Suppose that there is room for the Operating System and two user programs. Now, when one job needs to wait for I/O, the processor can switch to the other job, which likely is not waiting for I/O. we might expand memory to hold three, four, or more programs and switch among all of them. This process is known as multiprogramming, or multitasking.

3 Multiprogrammed Batch Systems:

-A Single program cannot keep either CPU & I/O devices busy at all times. multiprogramming increases CPU utilization by organizing jobs in such a manner that CPU has always one job to Execute.

If Computer is required to run Several programs at the 'same time', the processor could be kept busy for most of the time by switching its attention from one program to the next Additionally I/O transfers could overlap the Processor activity i.e. while one program is awaiting for an I/O transfer, another program can use the processor. So CPU never sits idle or if comes in idle state then after a very small time it is again busy.

Multiprogramming increases CPU utilization by organising jobs (Code and data) so that the CPU always has one to Execute.

The Operating system keeps Several jobs in memory Simultaneously.

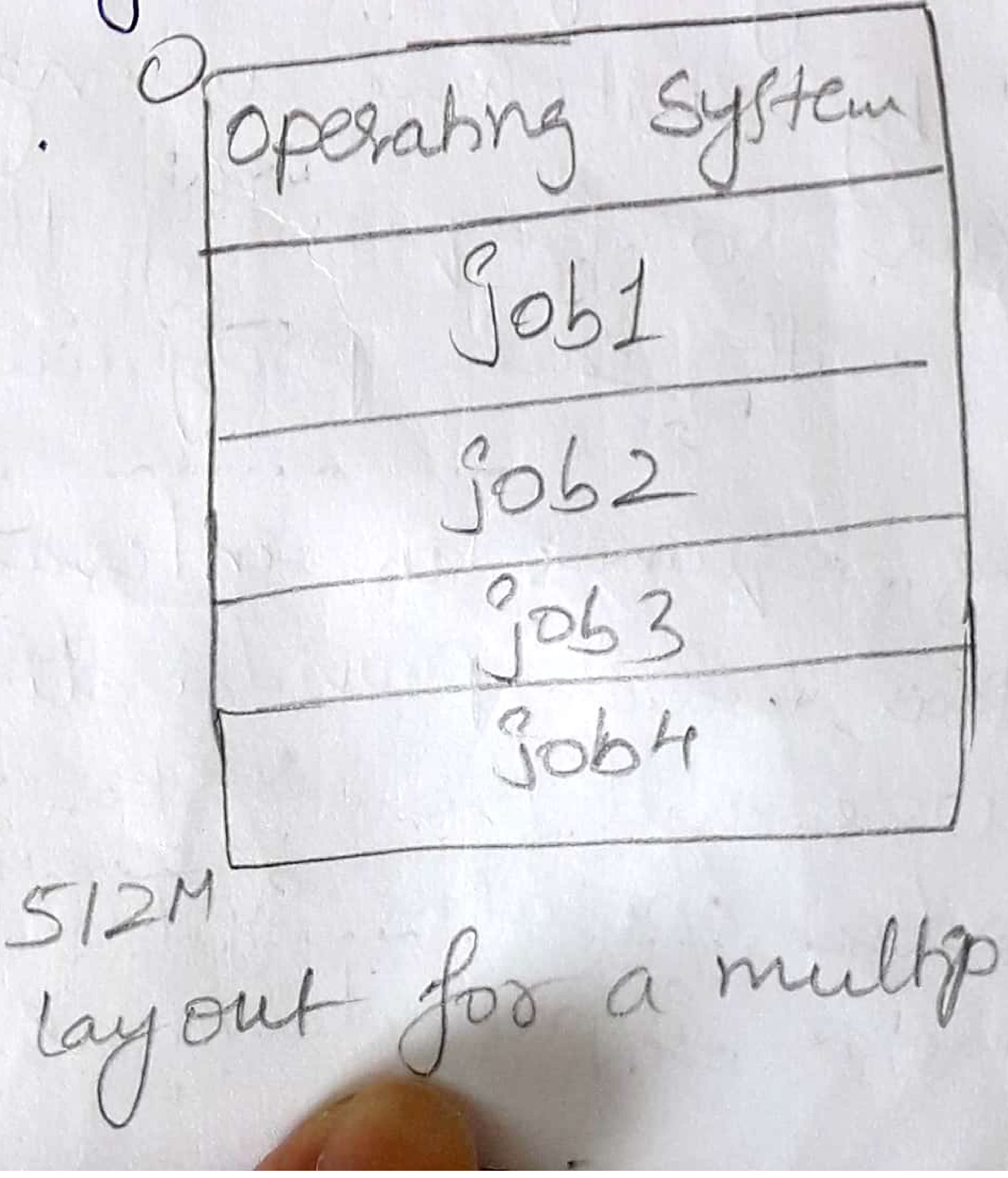


Fig: Memory layout for a multiprogramming system

in general, main memory is too small to accommodate
all jobs, the jobs are kept initially on the
disk in the job pool. This pool consists of
all processes residing on disk awaiting
allocation of main memory.

(4) Time Sharing Systems: (Multitasking)

Multiprogrammed systems provide an Environment
in which the various system resources (Eg: CPU,
memory and peripheral devices) are utilized
effectively, but they do not provide for
user interaction with the computer system.

Time sharing (or multitasking) is a logical
Extension of multiprogramming.

In time-sharing systems, the CPU Executes
multiple jobs by switching among them, but the
switches occur so frequently that the users
can interact with each program while it is running.

Time sharing requires an interactive (or hands-on)
computer system, which provides direct communication
between the user and the system. The user gives
instructions to the operating system or to a program
directly, using a input device such as a keyboard or

a mouse, and waits for immediate results on an output device. Accordingly, the response time should be short — typically less than one second.

A time shared operating system allows many 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 rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use, even though it is being shared among many users.

Each user has at least one separate program in memory. "A program loaded into memory and executing is called a process".

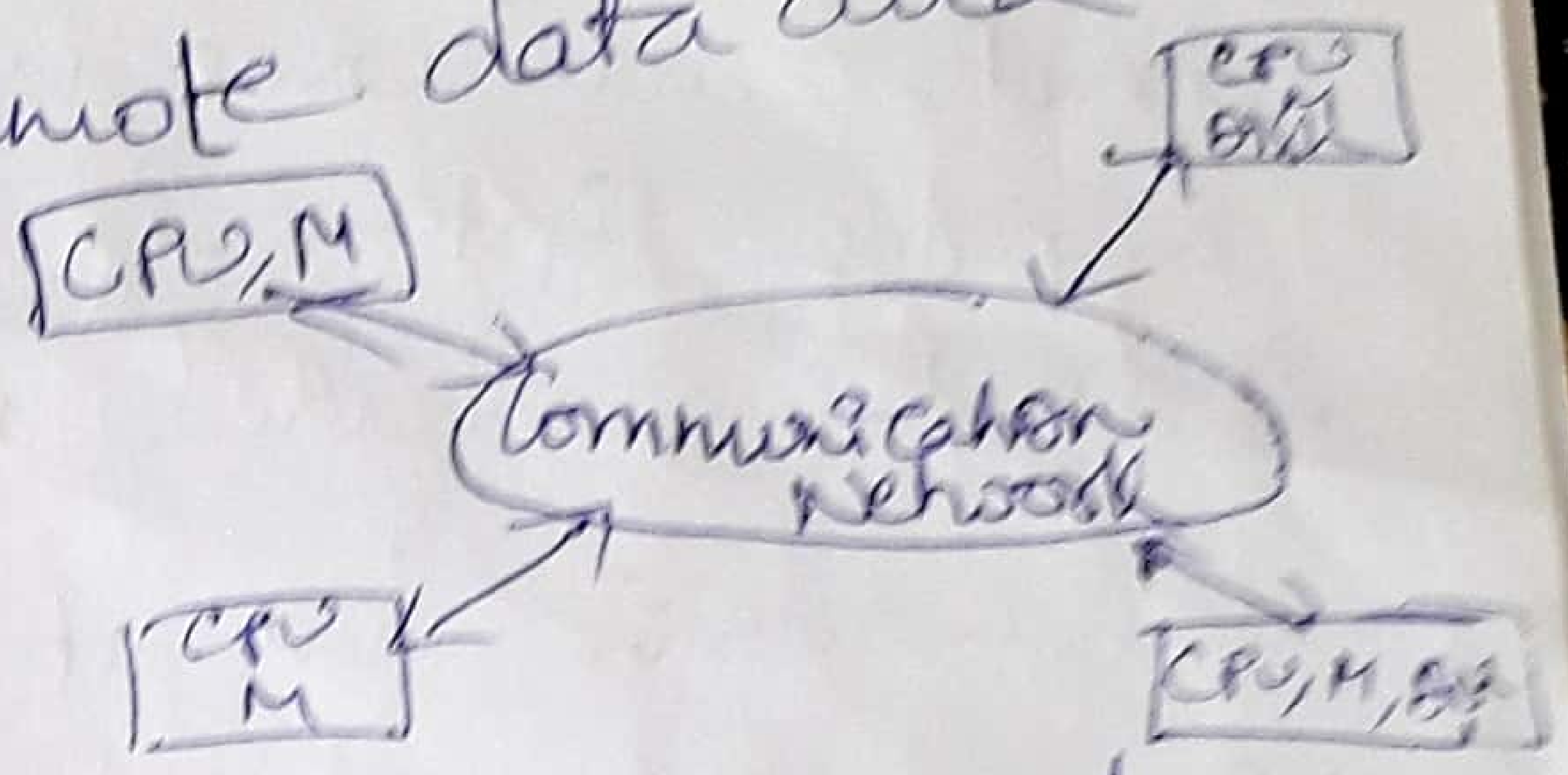
Time sharing and multiprogramming require that several jobs be kept simultaneously in memory. "If several jobs are ready to be brought into memory and if there is not enough room for all of them, ~~then~~ the system must choose among them. making this decision is job scheduling."

when the OS selects a job from the job pool, it loads that job into memory for execution. Having several programs in memory at the same time requires some form of memory management. "If several jobs are ready to run at the same time, the system must choose among them. Making this decision is CPU scheduling".

In a time-sharing system, the OS must ensure reasonable response time, which is sometimes accomplished through swapping, where processes are swapped in and out of main memory to the disk.

(5) A distributed system can be Characterized as collection of multiple autonomous Computers that Communicate over a Communication network such as 'message passing mechanism and having following features:

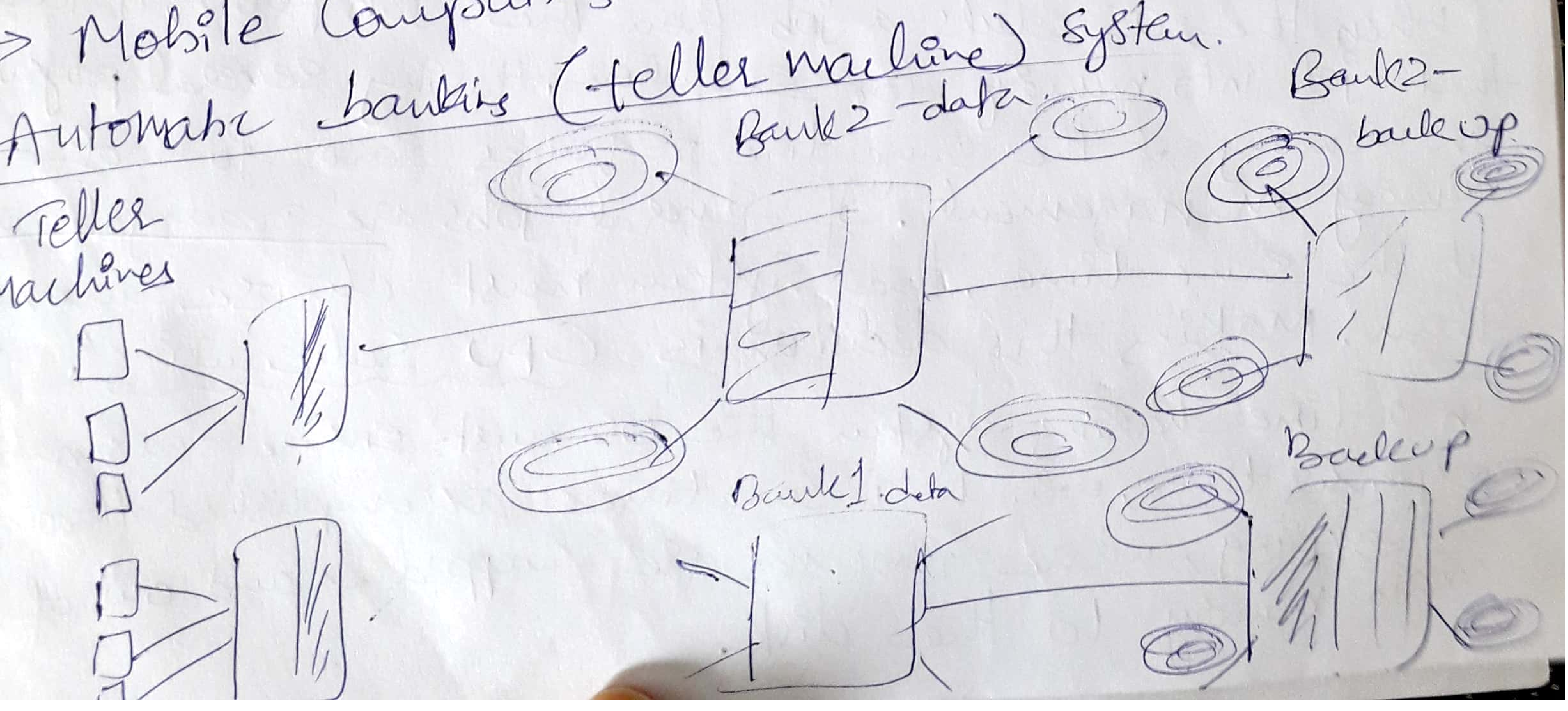
- → NO Common physical clock
- → Enhanced Reliability.
- → Increased performance / Cost ratio
- → Access to geographically remote data and resources
- → Scalability.



Examples of Distributed System!

- → Telephone Networks and Cellular Networks
- → Computer Networks such as internet & intranet
- → ATM (bank) Machines.
- → Distributed database and distributed database management system.
- → Network of workstations.
- → Mobile Computing etc.

Automatic banking (teller machine) system.



→ Primary requirements: Security & reliability

→ Consistency of replicated data.

→ Concurrent transactions (Operations which involve accounts in different banks: simultaneous access from several users, Etc).

→ Fault tolerance.

Advantages:

→ Information sharing among Distributed users

→ Resource sharing.

→ Extensibility & incremental growth

→ Shorter Response time & Higher output

A Distributed operating system controls and manages the hardware and software resources of a distributed system. when a program is executed on a distributed system, user is not aware of where the program is executed on the location of the resources accessed.

The absence of both shared memory and global clock, and unpredictable communication delays make the design of distributed operating system very difficult.

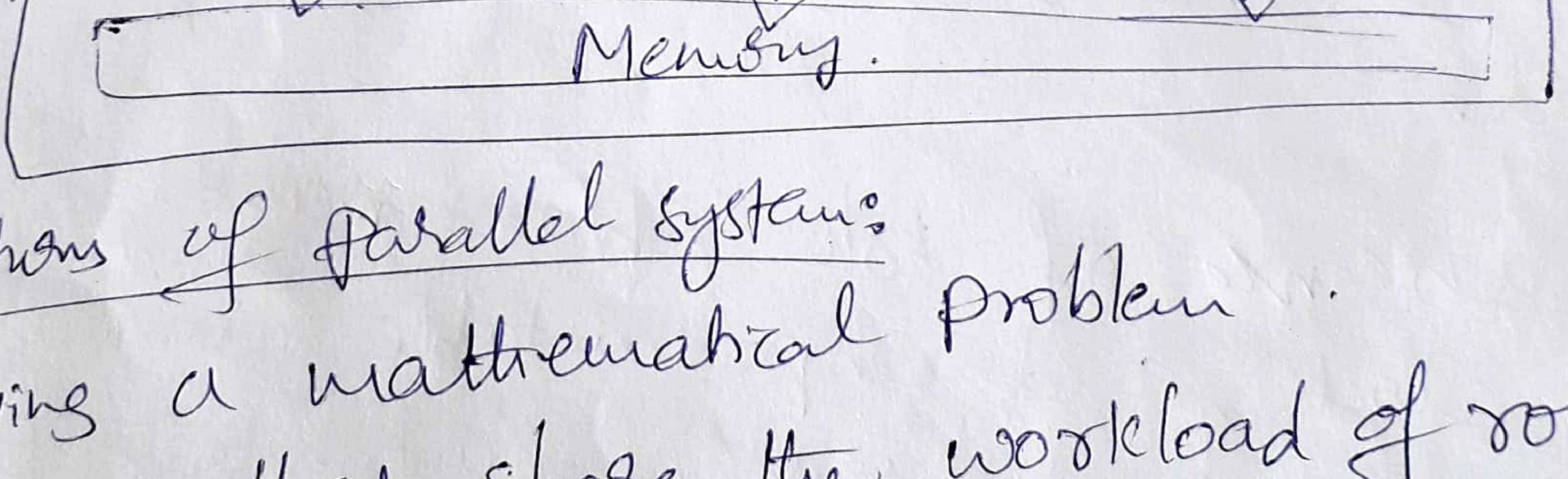
(6) Parallel system. (Multiprocessor operating system)

→A system is said to be a parallel system in which multiple processors have direct access to shared memory which forms a common address space

→usually tightly-Coupled System are referred to as parallel system. In these systems, there is a single system wide primary memory (address space) that is shared by all the processors.

on the other hand Distributed system are loosely-Coupled system.

→ parallel Computing is the use of two or more Processors (cores, Computers) in Combination to solve a single problem.



Applications of parallel systems:

→ solving a mathematical problem.

→ two servers that share the workload of routing mail, managing connections to an accounting system or database.

→ Supercomputers are usually placed in parallel system architecture.

→ Terminals Connected to single server.

Advantages of Parallel System:

→ Provide Concurrency (do multiple things at the same time)

→ Taking advantage of non-local resources

→ Cost savings.

→ Over Coming memory Contraints

→ Global address space provides a user friendly programming perspective to memory.

3 main Advantages

1. Increased throughput. → ↑ no. of proces. get more work in less time

2 Economy of scale →

3- Increased reliability.

Disadvantages.

→ lack of Scalability b/w memory & CPUs

→ Programmer responsibility for Synchronization Constructs that Ensure "Correct" access of global memory.

→ It becomes increasingly difficult and Expensive to design and produce Shared memory machines with Ever increasing numbers of processors.

Pallel Computing is the Simultaneous use of multiple compute resources to solve a Computational problem. - Accomplished by breaking the problem into independent parts so that Each Processing Element Can Execute its part of the algorithm Simultaneously with ~~the~~ the others

Parallel Vs Distributed System

	Parallel Systems	Distributed system
Memory Multiple Processors	Tightly coupled system Shared memory	weakly coupled system Distributed memory
Control	Global clock control	No global clock control
Processor interConnection	Order of Tbps	Order of Gbps
Main focus	Performance Scientific Computing	Performance (cost and Scalability) Reliability/availability information/resource Sharing

Eg: Intel/AMD:

2004: 2 cores per processor
2006: 4 " "
2009: 6 " "

The ability to continue providing service proportional to the level of surviving hardware is called graceful degradation. some systems go beyond graceful degradation and are called fault tolerant, because they can suffer a failure of any single component and still continue operation.

two types of systems:

① Asymmetric multiprocessing — defines a master-slave relationship. the master processor schedules & allocates work to the slave processor

② Symmetric Multiprocessing (smp) — all processors are peers, no master-slave relationship. Exists b/w processors.

7) Real-time operating Systems: are very fast and quick respondent Systems.

these systems are used in an environment where a large number of Events must be accepted and processed in a short time.

Eg. Rocket launchings, flight control, robotics, Defense, application system

Real time processing requires quick transaction and characterized by supplying immediate response.

Primary objective of ROT. is to provide quick response time and thus to meet a scheduling deadline.

Types of Real-time operating system:

① Soft-Real-Time OS:

A process might not be Executed in given deadline. It can be crossed it then Executed next, without harming the system.

Eg: digital camera, mobile phones, Etc. → telephone switches

If certain deadlines are missed then system continues its working with no failure but its performance degrades.

② Hard-Real-time OS:

A process should be Executed in given deadline.

The deadline should not be crossed.

The preemption time for Hard Real time OS is almost less than the few microseconds.

Eg: Airbag Control in Cars,
anti-lock brake,
Engine Control system, Etc.

Applications: Control system, image processing, VOIP(Voice over IP)

Hard real Time System: If any deadline is missed then system will fail to work or does not work properly.

This system guarantees that Critical tasks be Completed on time.

Operating System Structure:

Simple Structure:

Many commercial operating systems do not have well-defined structures. Started with small, simple, and limited systems and then grew beyond their original scope.

Eg: MS-DOS

It was written to provide the most functionality in the least space, so it was not divided into modules carefully.

→ In MS-DOS, the interfaces and levels of functionality are not well separated.

eg: application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to malicious programs, causing entire system crashes when user programs fail.

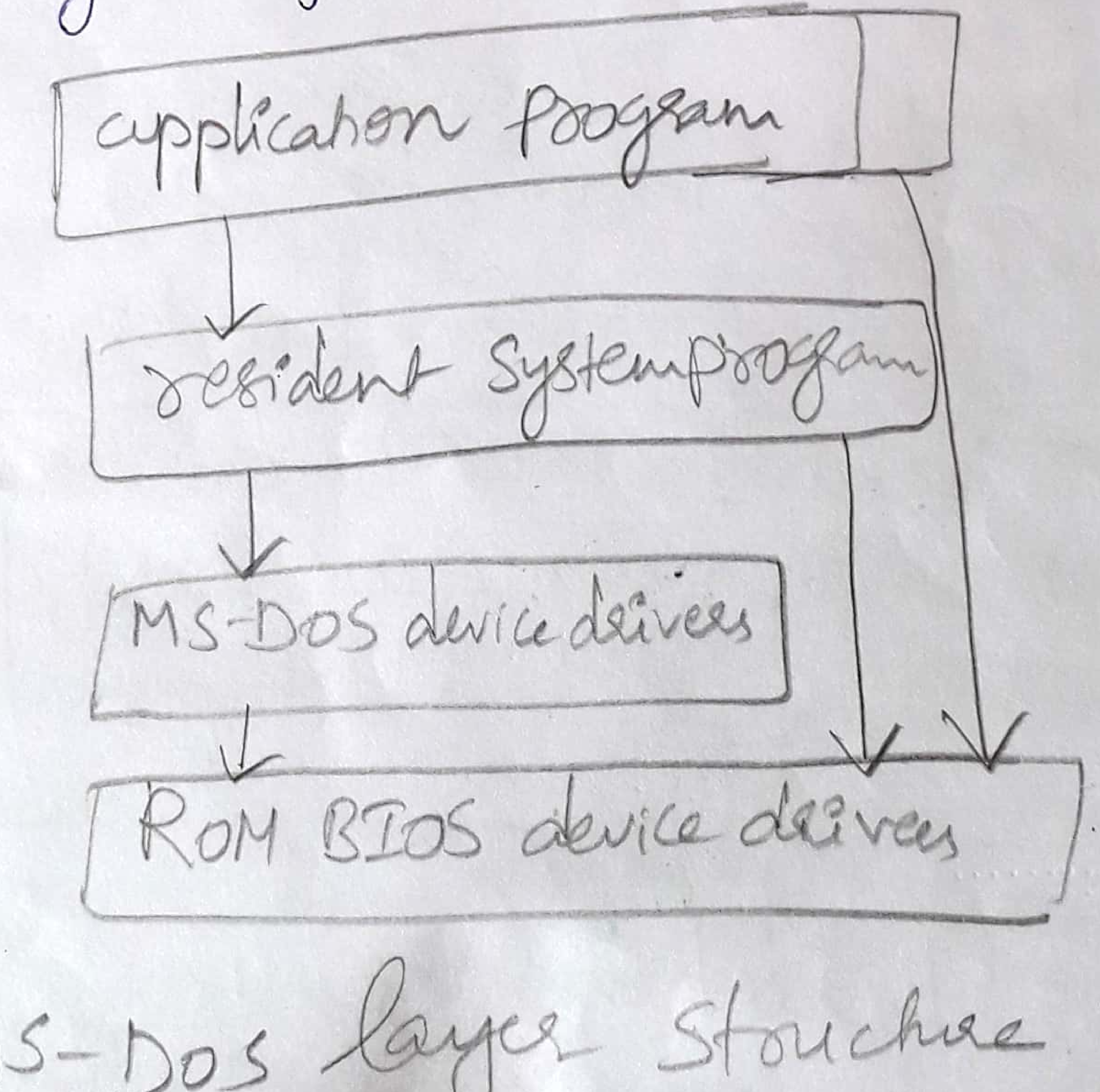


Fig: MS-Dos layer Structure

→ Another Example of limited structuring is the original UNIX operating system.

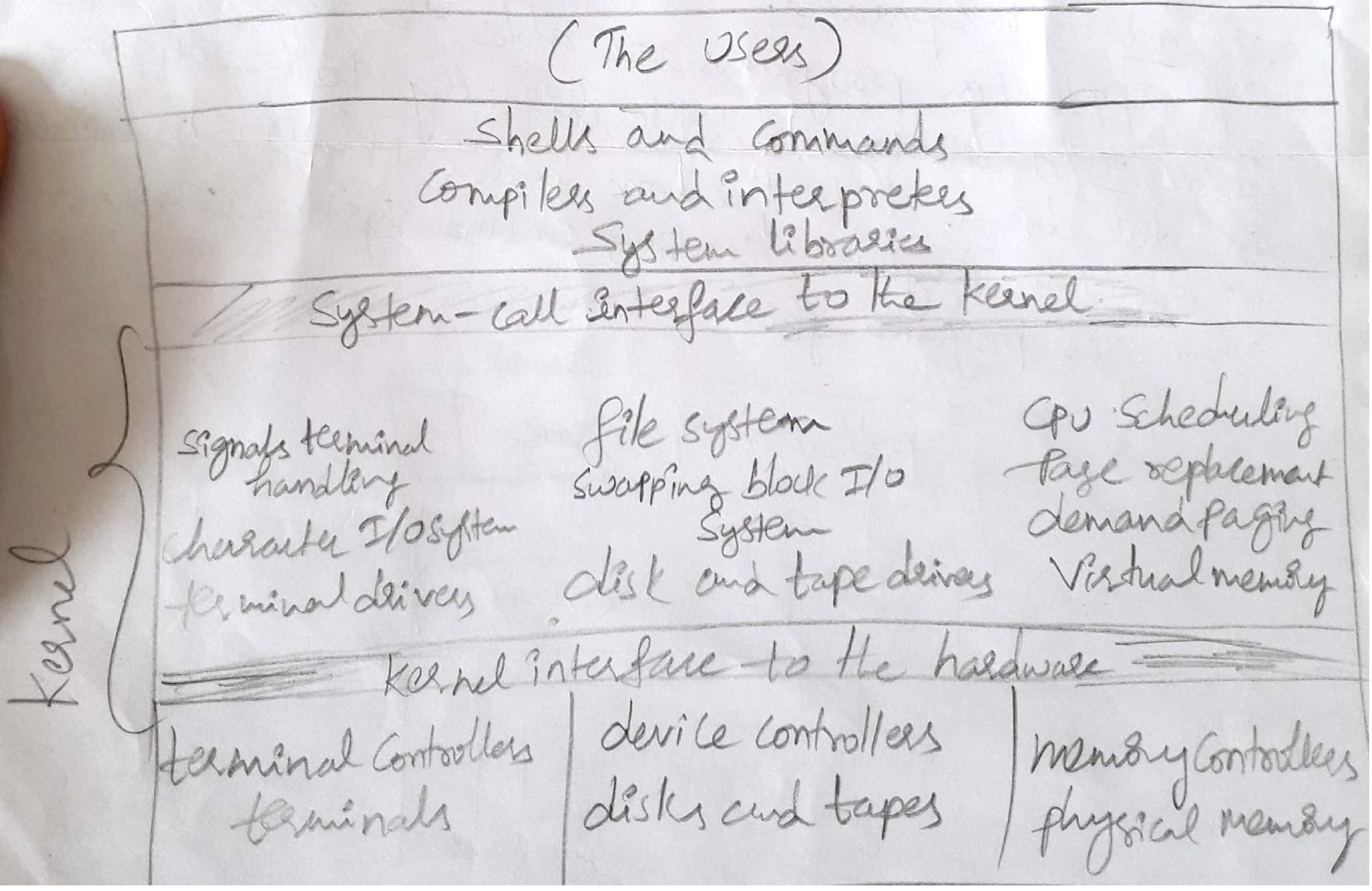
→ Like MS-DOS, UNIX initially - limited by hardware functionality.

It Consists of two-separate parts:-

① The kernel and

② The system programs.

→ The kernel is further separated into a Series of interfaces and device drivers, which have been added and Expanded over the years as UNIX has evolved.



as shown in figure - Traditional UNIX System structure, i.e. Everything below the System-call interface and above the physical hardware is the Kernel.

⇒ The kernel provides the file System, CPU Scheduling, memory management, and other operating-System functions through System calls.

⇒ i.e. vast amount of functionality to be combined into one level. This monolithic structure was difficult to implement and maintain.

② Layered Approach:

The operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware, The highest (layer N) is the user interface.

⇒ The main advantage of the layered approach is Simplicity of Construction and debugging.

⇒ The layers are selected so that Each uses functions (operations) and services of only lower-level layers.

⇒ This approach Simplifies debugging and System verification.

⇒ The first layer can be debugged without any concern for the rest of the system, because, it uses only the basic hardware to implement its functions.

⇒ once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on, If an Error is found during the debugging of a particular layer, the Error must be on that layer, because the layers below it are already debugged.

⇒ Each layer is implemented with only those operations provided by lower-level layers.

⇒ A layer does not need to know how these operations are implemented, it needs to know only what these operations do. Hence, Each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.

⇒ The major difficulty with the layered approach involves appropriately defining the various layers. Because a layer can use only lower-level layers, careful planning is necessary.

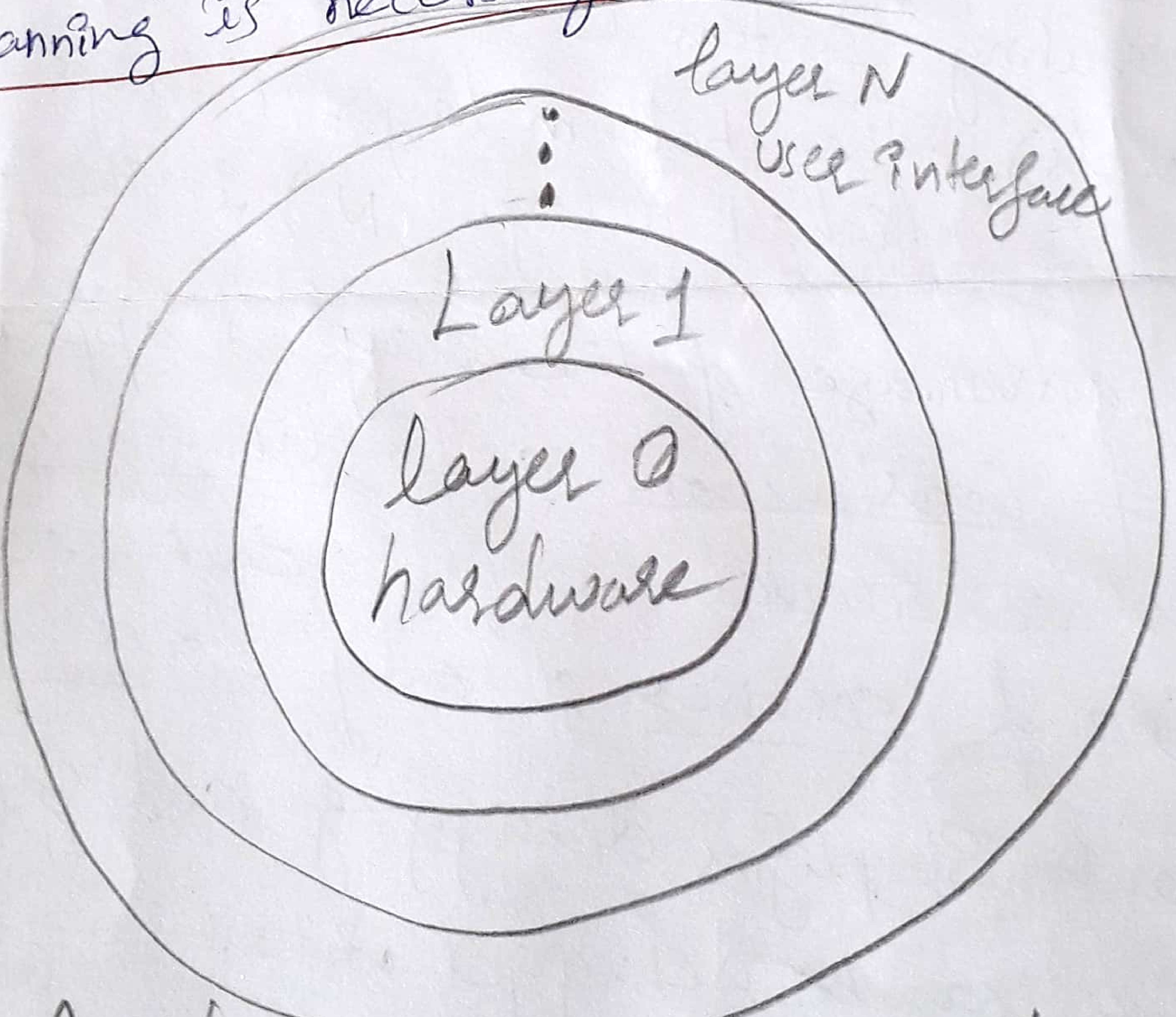


Fig: A layered operating system.

③ Microkernels:

⇒ This method Structures the operating system by removing all nonessential Components from the Kernel and implementing them as System and user-level programs.

⇒ The result is a smaller Kernel. There is little care about which services should remain in the kernel and which should be implemented in user space is need.

⇒ The main function of the microkernel is to provide a Communication facility between the client program and the various Services that are also running in user space.

⇒ Communication is provided by message passing.

⇒ The microkernel also provides more security and reliability, Since most services are running as user- rather than kernel- Processes. If a service fails, the rest of the operating System remains untouched.

⇒ micro kernels can suffer from performance decreases due to increased System function overhead.

④ Modules:

The Kernel has a set of Core Components and links in additional services Either during boot time or during run time. Such a strategy uses dynamically loadable modules and is Common in modern implementations of UNIX, such as Solaris, Linux, and Mac OS X.

Eg: The Solaris operating system structure, is organized around a core kernel with seven types of loadable kernel modules: Scheduling classes, file systems, Loadable system calls

Executable formats, STREAMS modules, Miscellaneous Device and bus drivers.

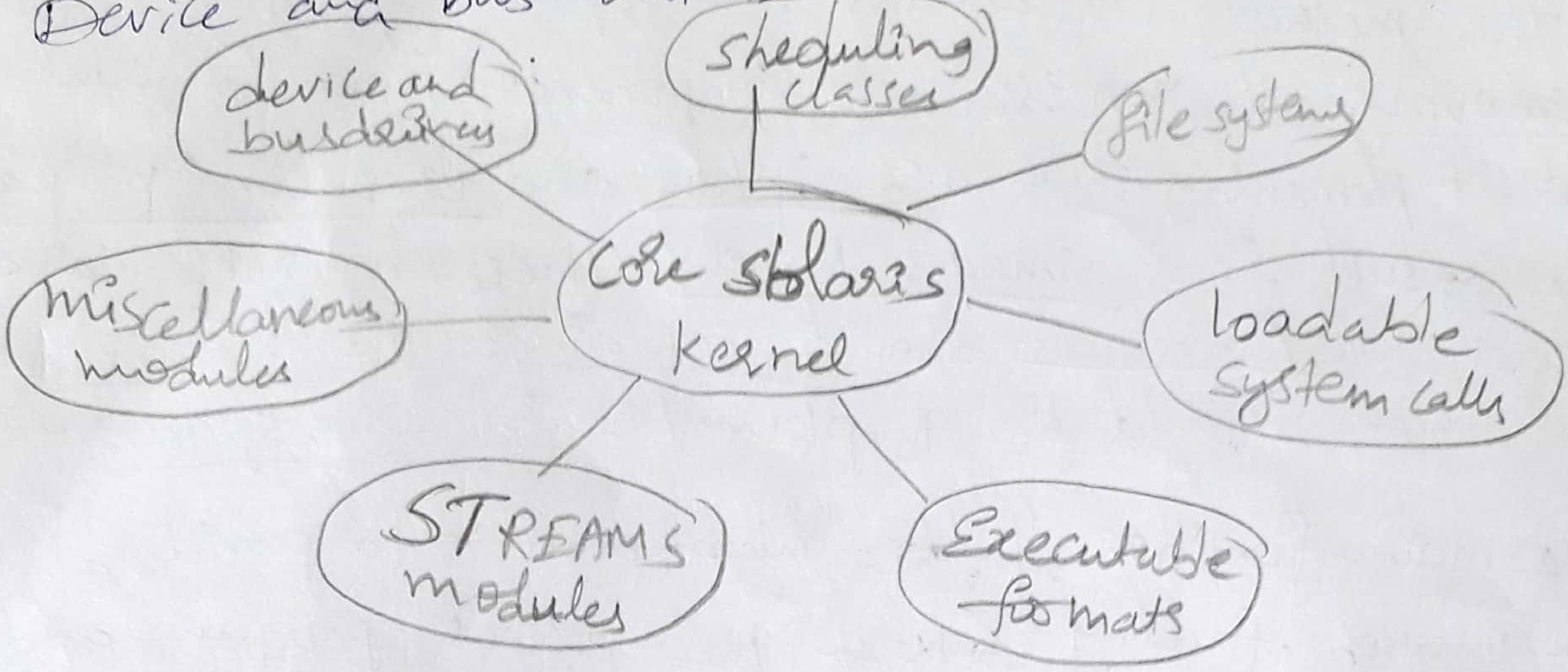


Fig: Solaris loadable modules.

Such a design allows the kernel to provide core services yet also allows certain features to be implemented dynamically.

Eg: device and bus drivers for specific hardware can be added to the kernel, and support for different file systems can be added as loadable modules.

→ It is more flexible than a layered system that any module can call any other module.

→ the approach is like the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules. but it is more Efficient, because modules do not need to invoke message passing in order to communicate

The Apple Mac OS X operating system uses a hybrid structure. It is a layered system in which one layer consists of the Mach micro kernel. The top layers include application Environments and a set of services providing a graphical interface to applications.

Below these layers is the kernel Environment, which consists of Mach micro kernel and BSD kernel.

→ Mach provides — memory management,
Support for remote procedure calls (RPC)
and Interprocess Communication (IPC) facilities, including message passing, and
Thread scheduling.

→ The BSD Component provides - a BSD Command line interface (CLI)
— Support for networking and file systems,
and an implementation of POSIX APIs,
including Pthreads.

In addition to Mach and BSD, the Kernel Environment provides an I/O kit for development of device drivers and dynamically loadable modules (which Mac OS X refers to as Kernel Extensions).

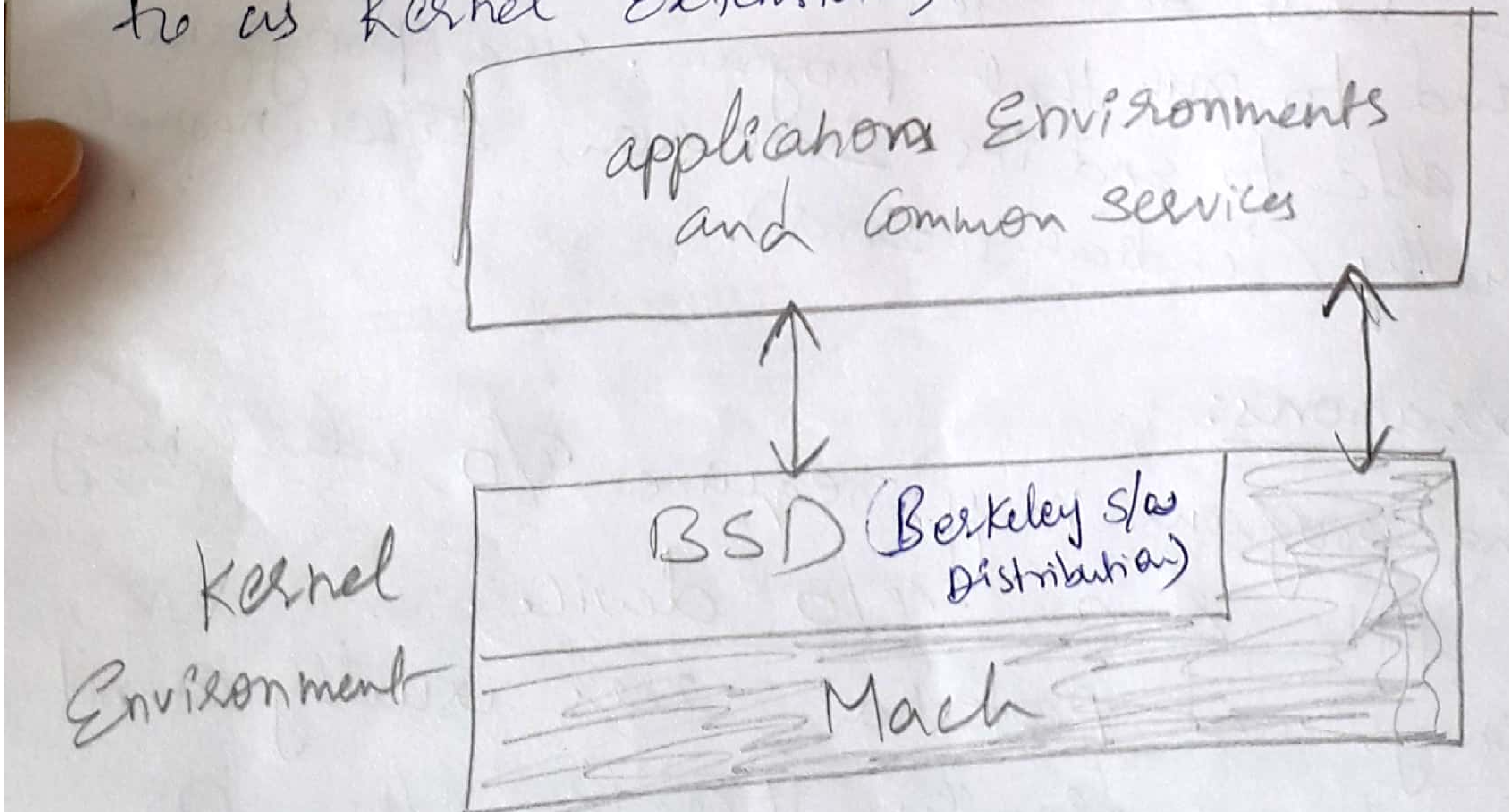

applications Environments and Common services
BSD (Berkeley s/w Distribution)
Mach
Kernel Environment

Fig: The Mac OS X Structure.

Operating - System services:

One set of operating system Services provides functions that are helpful to the user.

① User interface:

→ DTrace Command-line interface (CLI), which uses text Commands and a method for Entering them.

→ a batch interface, in which Commands and directives to control those Commands are Entered into files, and those files are Executed.

→ a graphical user interface (GUI) is used, Here the Enterface is a window system with a pointing device to direct I/o, choose from menus, and make selections and a keyboard to Enter text.

② Program Execution:

The system must be able to load a program into memory and to run that program. The program must be able to End its Execution, Either normally or abnormally (indicating Error).

③ I/O operations:

A running program may require I/O, which may involve a file & an I/O device. for Effiency and protection, users usually canno Control I/O devices directly. therefore, the OS must provide a means to do I/O.

④ File-System manipulation:

Programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information. some programs include permissions management to allow or deny access to files & directories based on file ownership.

⑤ Communications: There are many circumstances in which one process needs to Exchange information with another process. Such communication may occur between processes that are Executing on the same computer or between processes that are Executing on different computer systems tied together by a computer network. Communications may be implemented via shared memory or through message passing, in which packets of information are moved between processes by the operating system.

⑥ Error detection:

Errors may occur in the CPU, memory hardware, in I/O devices (such as a parity error on tape, a connection failure on a network, or lack of paper in the printer) and in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location, or a too-great use of CPU time). for Each type of Error, the OS should take the appropriate action to Ensure correct

and Consistent Computing.
Debugging facilities can greatly Enhance the user's and programmer's abilities to use the System Efficiently.

Another set of OS functions Exists not for helping the user but rather for Ensuring the Efficient operation of the System itself.

① Resource allocation: when there are multiple users or multiple jobs running at the same time, resources must be allocated to Each of them. Many different types of resources are managed by the operating system. Some (such as CPU cycles, main memory, and file storage) may have special allocation code. whereas others (such as I/O devices) may have much more general request and release code.

② Accounting: we want to keep track of which users use how much and what kind of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics. (to reconfigure the system to improve computing services).

③ Protection and Security:
The owners of information stored in a multiuser or networked computer system may want to control use of that information.
Protection involves Ensuring that all access to system resources is controlled.
Security of the system from outsiders is also important. It requires each user to authenticate himself or herself to the system. with a password, to gain access to system resources.

System Components:

1. Process Management:

→ A program does nothing unless its instructions are Executed by a CPU.

→ A Program in Execution is a process.

Eg: • A time shared user program such as a Compiler is a process.

- • A word-processing program being run by an individual user, on a PC is a process.

• A System task, Such as sending output to a printer, can also be a process.

→ A process needs certain resources - including CPU time, memory, files and I/O devices. to accomplish its task.

→ These resources are either given to the process when it is created or allocated to it while it is running.

→ In addition to the various physical and logical resources that a process obtains when it is created, various initialization data (input) may be passed

→ A program by itself is not a process.

→ A program is a passive Entity,

→ A program is a file stored on disk,
like the Contents of a file stored on disk,

→ where as a process is an active Entity.

→ A single-threaded process has one program counter
Specifying the next-instruction to Execute.

→ the Execution of Such a process must be Sequential.

→ The CPU Executes one instruction of the process after another, until the process Completes.

→ Further, at any time, one instruction at most is Executed on behalf of the process. Thus, although two processes may be associated with the Same program, they are considered two Separate Execution Sequences.

→ A process is the unit of work in a System.
A System Consists of a Collection of processes,
Some of which are operating-System processes
(those that Execute System code), and the
rest of which are user processes (those that
Execute user code).

→ All these processes can Execute concurrently
by multiplexing on a Single CPU.

The Operating System is responsible for the following activities in Connection with process Management:

1. → Scheduling processes and threads on the CPUs.
2. → Creating and deleting both user and system Processes
3. → Suspending and resuming processes.
4. → Providing mechanisms for process Synchronization
5. → Providing mechanisms for process Communication.

2. Memory Management:

→ Main Memory is a large array of bytes ranging in Size from hundreds of thousands to billions.

→ Each byte has its own address.

→ Main Memory is a repository of quickly accessible data shared by the CPU and I/O devices.

→ Instruction-fetch cycle – CPU reads instructions from main memory.

→ data-fetch cycle – reads & writes data from main memory.

→ For a Program to be Executed, it must be mapped to absolute addresses and loaded in-to memory.

→ As the program Executes, it accesses program instructions and data from memory by generating these absolute addresses. when the program terminates, its memory space is declared available, and the next program can be loaded and Executed.

→ To improve both the utilization of the CPU and the speed of the Computer's response to its users, Computers must keep several Programs in memory., Creating a need for memory management schemes.

→ The Operating System is responsible for the following activities in Connection with memory management:

1. → Keeping track of which parts of memory are currently being used and who is using them.
2. → Deciding which processes (or parts of processes) and data to move into and out of memory.
3. → Allocating and deallocating memory space as needed.

Policies (what should be done?)

→ Decide when to load each process into memory

→ Decide how much memory space to allocate each process.

→ Decide when a process should be removed from memory.

Mechanism (How it should be done?)

→ keep track of memory in use

→ keep track of unused (free) memory

→ Protect memory space

→ Allocate, deallocate spaces for processes.

→ Swap processes: memory ⟷ disk.

~~1.3~~ 3 Storage Management:

3.1 ~~1.8.1~~ File-System Management

~~1.8.2~~ Mass-storage Management

3.2

3.3 ~~1.8.3~~ Caching

3.4 ~~1.8.4~~ I/O Systems.

To make the Computer system Convenient for users, the operating system provides a uniform, logical view of information storage.

3.1 1.8.1 File System Management:

→ Computers can store information on Several different types of physical media.

Magnetic disk, optical disk, and magnetic tape are lbe most Common.

→ Each of these media has its own characteristics and physical organization.

→ Each medium is Controlled by a device such as a disk drive or tape drive, that also has its own unique characteristics.

→ These properties include access Speed, Capacity data-transfer rate, and access method (Sequential or random).

- A file is a Collection of related information defined by its Creator.

Commonly, files represent programs (both source and object forms) and data.

Data files may be numeric, alphabetic, alphanumeric or binary. Files may be free-form (text files). or they may be formatted (Eg: fixed fields).

⇒ The operating System implements the abstract Concept of a file by managing mass-Storage media, such as tapes and disks, and the devices that Control them.

⇒ files are normally organizeld into directories to make them easier to use.

→ when multiple users have access to files, it may be desirable to control which user may access a file and how that user may access it (for Eg: read, write, append).

The operating System is responsible for the following activities in Connection with file management:

→ Creating and deleting files

→ Creating and deleting directories to organize files

→ Supporting primitives for manipulating files and directories

⇒ Mapping files onto Secondary Storage

→ Backing up files on stable (non volatile) Storage media.

3.2 Mass-storage Management:

→ Most modern Computer systems use disks as the principal on-line Storage medium for both Programs and data.

→ Most programs including Compilers, assemblers, word processors, Editors, and formatters — are stored on a disk until loaded into memory.

→ They then use the disk as both the Source and destination of their processing.

Hence, the proper management of disk storage is important for a Computer System.

→ The operating System is responsible for the following activities in Connection with disk management:

→ · Free-Space management.

→ · Storage allocation.

→ · Disk Scheduling.

Magnetic tape drives and their tapes and CD and DVD drives and platters are tertiary Storage devices.

The media (tapes and optical platters) Vary between WORM (write-once, read-many times) and RW (read-write) formats.

1.3.3 Caching:

→ Information is normally kept in Some Storage System (Such as main memory).

→ As it is used, it is copied into a faster storage system – the Cache – on a temporary basis.

→ when we need a particular piece of information, we first check whether it is in the cache. If it is, we use the information directly from the cache. If it is not, we use the information from the source, putting a copy in the cache under the assumption that we will need it again soon.

→ The programmer implements the register-allocation and register-replacement algorithms to decide which information to keep in registers and which to keep in main memory.

→ Because caches have limited size, Cache managemen is an important design problem. Careful selection of the cache size and of a replacemen policy can result in greatly increased performance.

→ The movement of information between levels of a storage hierarchy may be Either Explicit or implicit, depending on the hardware desig and the Controlling Operating-System Software.

→ For instance, data transfer from cache to CPU and registers is usually a hardware function, with no operating-System intervention.

→ In contrast, transfer of data from disk to memory is usually controlled by the operating System.

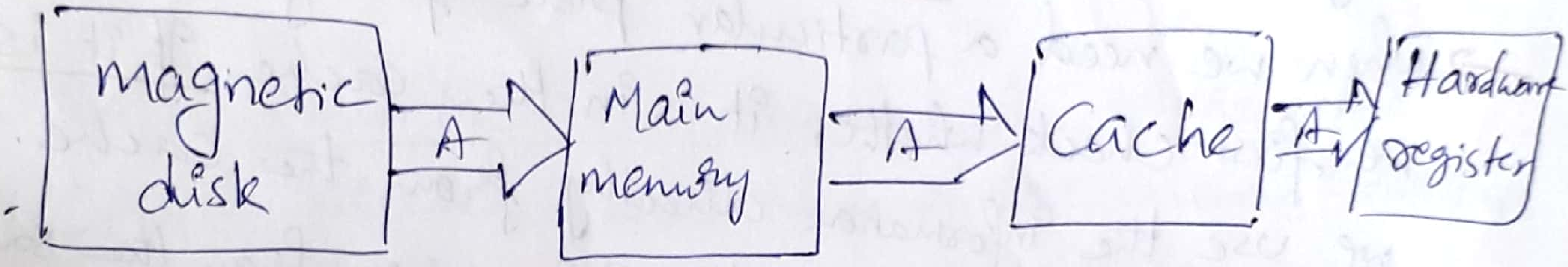


Fig: Migration of integer A from disk to register.

In a hierarchical storage structure, the same data may appear in different levels of the storage System.

Eg: Suppose that an integer A that is to be incremented by 1 is located in file B, and file B resides on magnetic disk.

The increment operation proceeds by first issuing an I/O operation to copy the disk block on which A resides to main memory. This operation is followed by copying A to the cache and to an internal register. Thus, the copy of A appears in several places: Once the increment takes place in the internal register, the value of A differs in the various storage systems. The value of A becomes the same only after the new value of A is written from the internal register back to the magnetic disk.

⇒ In a Computing Environment, where only One Process Executes at a time, this arrangement poses no difficulties, since an access to integer A will always be to the copy at the highest level of the hierarchy.

⇒ In a Multitasking Environment, where the CPU is switched back and forth among various processes, Extreme care must be taken to ensure that, if several processes wish to access A, then each of these processes will obtain the most recently updated value of A.

⇒ In a multiprocessor Environment, where, in addition to maintaining internal registers, Each of the CPUs also contains a local cache. In such an Environment, a copy of A may Exist simultaneously in several caches. since the various CPUs can all Execute in parallel, we must make sure that an update to the value of A in one cache is immediatel reflected in all other caches where A reside This situation is called Cache Coherenc

In a distributed Environment, the situation becomes even more complex, In this Environme several copies (or replicas) of the same file Ca

kept on different computers.
Since the various replicas may be accessed and updated concurrently, some distributed systems ensure that, when a replica is updated in one place, all other replicas are brought up to date as soon as possible.

1.8.4 I/O Systems:

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user.

Eg: in UNIX, the peculiarities of I/O devices are hidden from the bulk of the operating system itself by the I/O subsystem.

The I/O subsystem consists of several components:

- A memory-management component that includes buffering, caching, and spooling (simultaneous peripheral operations online)
- A general device-driver interface.
- Drivers for specific hardware devices.

Only the device driver knows the peculiarities of the specific device to which it is assigned.

3.5. Protection and security:

→ Protection, is any mechanism for controlling the access of processes & users to the resources defined by a computer system.

→ This mechanism must provide means to speify the controls to be imposed and to enforce the controls.

→ Protection can improve reliability by detecting latent errors at the interfaces between component subsystems.

→ Early detection of interface errors can often prevent contamination of a healthy subsystem by another subsystem that is malfunctioning.

→ It is the job of security to defend a system from external and internal attacks.

Such attacks spread across a huge range and include viruses and worms, denial-of-service attacks, identity theft, and theft of service.

→ Protection and security require the system to be able to distinguish among all its users.

→ Most operating systems maintain a list of user names and associated user identifiers (user IDs).

Systems generally first distinguish among users, to determine who can do what.

→ User identities (user IDs, security IDs) include name and associated number, one per user.

→ User ID then associated with all files, processes of that user to determine access control

→ Group identifier (group ID) allows set of users to be defined and controls manage then also associated with each process, fil

→ Privilege Escalation allows user to change to Effective ID with more right

System calls:

⇒ System calls provide an interface to the services made available by an operating system.

⇒ application developers design programs according toan application programming interface (API).

⇒ The API Specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect.

⇒ Three of the most common APIs available to application programmers are the:

- Win32 API for windows systems
- The POSIx API for POSIx-based systems (POSIx - Portable Operating system interface)
- Java API for designing programs that run on the Java Virtual Machine.

System call Implementation:

The run-time support system for most programming languages provides a system-call interface that serves as the link to system calls made available by the operating system.

- The System-call interface intercepts function calls in the API and invokes the necessary System calls within the Operating System.
- A number is associated with each system call and the System-call interface maintains a table indexed, according to these numbers.
- The system call interface then invokes the intended system call in the operating-System kernel and returns the status of the system call and any return values.
- The caller need to know nothing about how the system call is implemented or what it does during Execution. Rather, it need only obey the API and understand what the operating System will do as a result of the Execution of that system call.
- Thus, most of the details of the operating-system interface are hidden from the programmer by the API and are managed by the run-time support library.

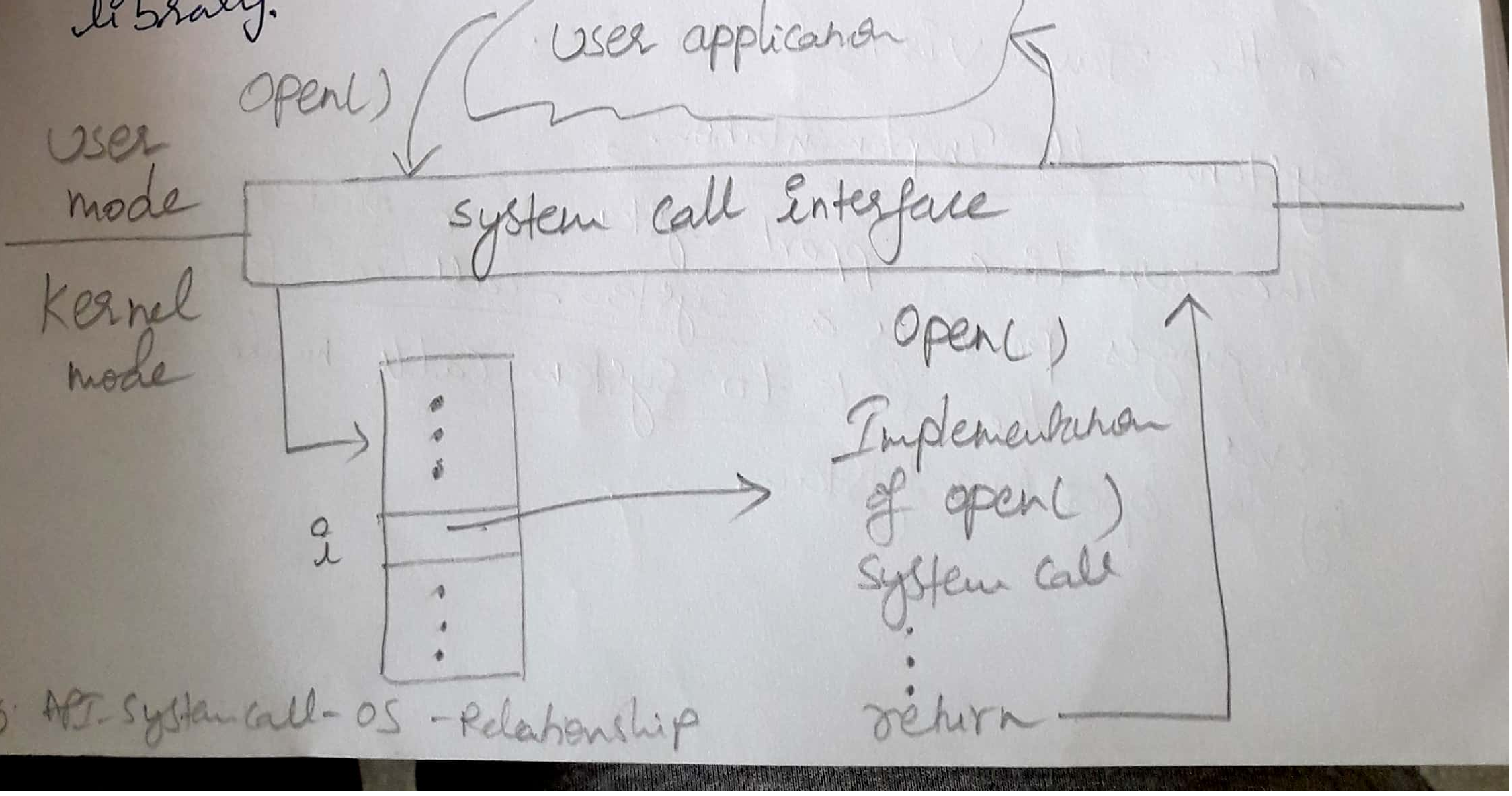


Fig: API-System call-OS-Relationship

System call parameter passing:

- Three general methods are used to pass parameters to the operating system.
- The simplest approach is to pass the parameters in registers.
- In some cases, there may be more parameters than registers. In these cases, the parameters are generally stored in a block, or table, in memory, and the address of the block is passed as a parameter in a register. This is the approach taken by Linux and solaris.
- Parameters also can be placed, or pushed onto the stack by the program and popped off the stack by the operating system.
- Some operating systems prefer the block or stack method because those approaches do not limit the number or length of parameters being passed.

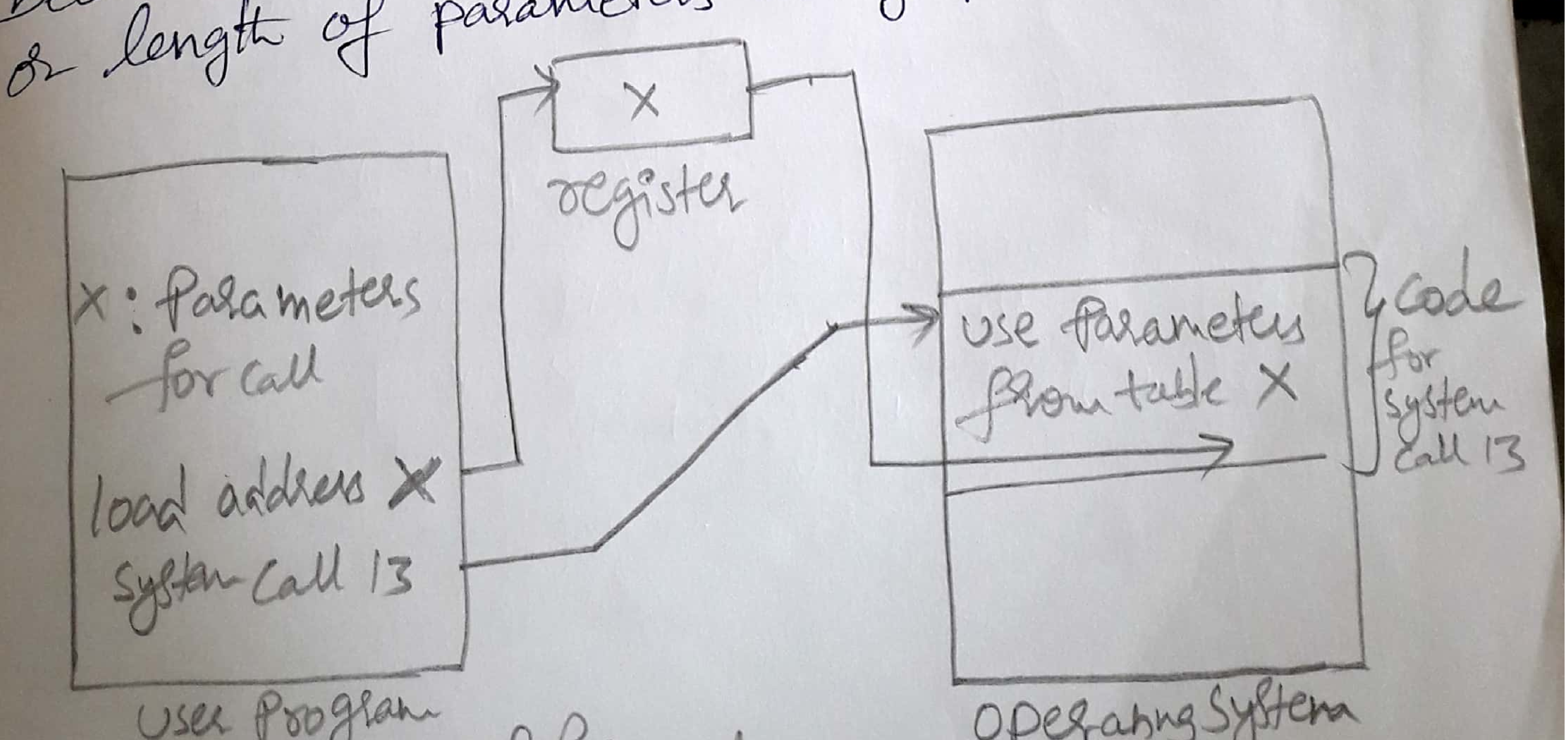


Fig:- Passing of Parameters as a table.

UNIT - III

DEADLOCKS

To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks.
To present a number of different methods for preventing or avoiding deadlocks in a computer system.

The Deadlock Problem

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set

Example

System has 2 disk drives

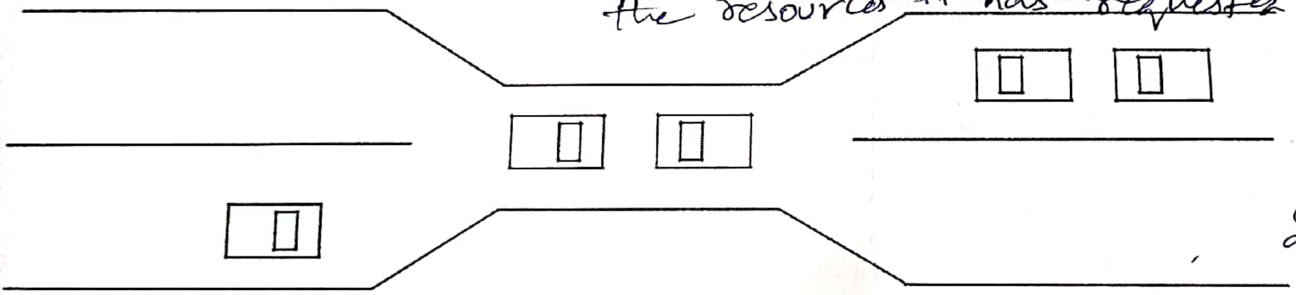
*P*1 and *P*2 each hold one disk drive and each needs another one.Example

semaphores *A* and *B*, initialized to 1

```
 P0        P1
wait (A);    wait(B)
wait (B); wait(A)
```

A process requests resources, if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. this situation is called a deadlock.

Bridge Crossing Example



- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
- Note – Most OSes do not prevent or deal with deadlocks

System Model : A system consists of a finite number of resources to be distributed among a no. of competing processes.

- Resource types *R*1, *R*2, . . ., *R*m
- *CPU cycles, memory space, I/O devices*

- Each resource type R_i has W_i instances. Each process utilizes a resource as follows:

① **request :** The Process requests the resource. If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.

② **use :** → The Process can operate on the resource (eg: Printer)

③ **release:** ③ The Process releases the resource

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously

→ **Mutual exclusion:** only one process at a time can use a resource

→ **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes

→ **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task

→ **Circular wait:** there exists a set $\{P0, P1, \ldots, P0\}$ of waiting processes such that $P0$ is waiting for a resource that is held by $P1$. $P1$ is waiting for a resource that is held by $P2, \ldots, Pn-1$ is waiting for a resource that is held by Pn. and $P0$ is waiting for a resource that is held by $P0$.

All four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent

* **Resource-Allocation Graph**

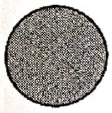
A set of vertices V and a set of edges E.V is partitioned into two types:

V { $P = \{P1, P2, \ldots. Pn\}$, the set consisting of all the processes in the system
$R = \{R1, R2, \ldots. Rm\}$, the set consisting of all resource types in the system

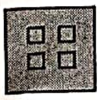
request edge – directed edge $P1 \rightarrow Rj$ $P_i \rightarrow R_j$

assignment edge – directed edge $Rj \rightarrow Pi$ $R_j \rightarrow P_i$

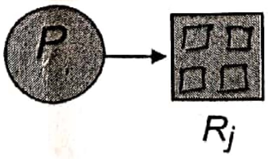
Process



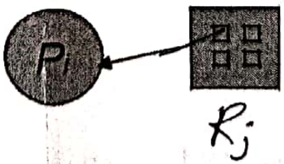
Resource Type with 4 instances



Pi requests instance of *Rjn*



Pi is holding an instance of *Rj*

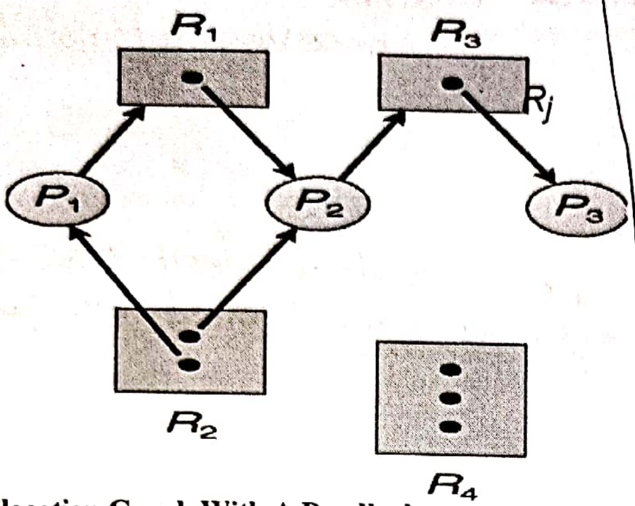


[redacted]

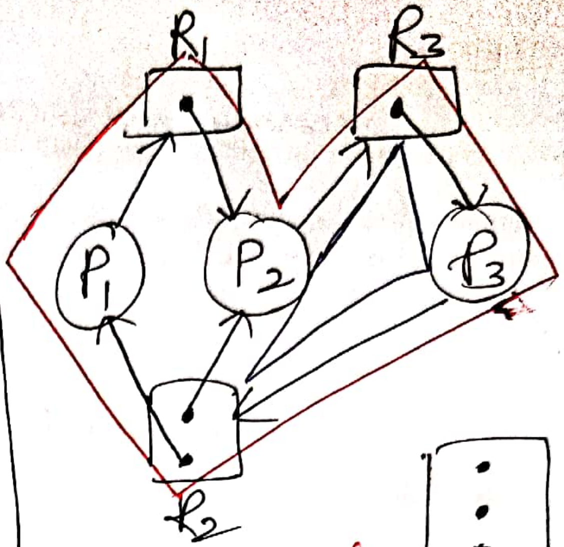
3

[redacted]

(1) Example of a Resource Allocation Graph

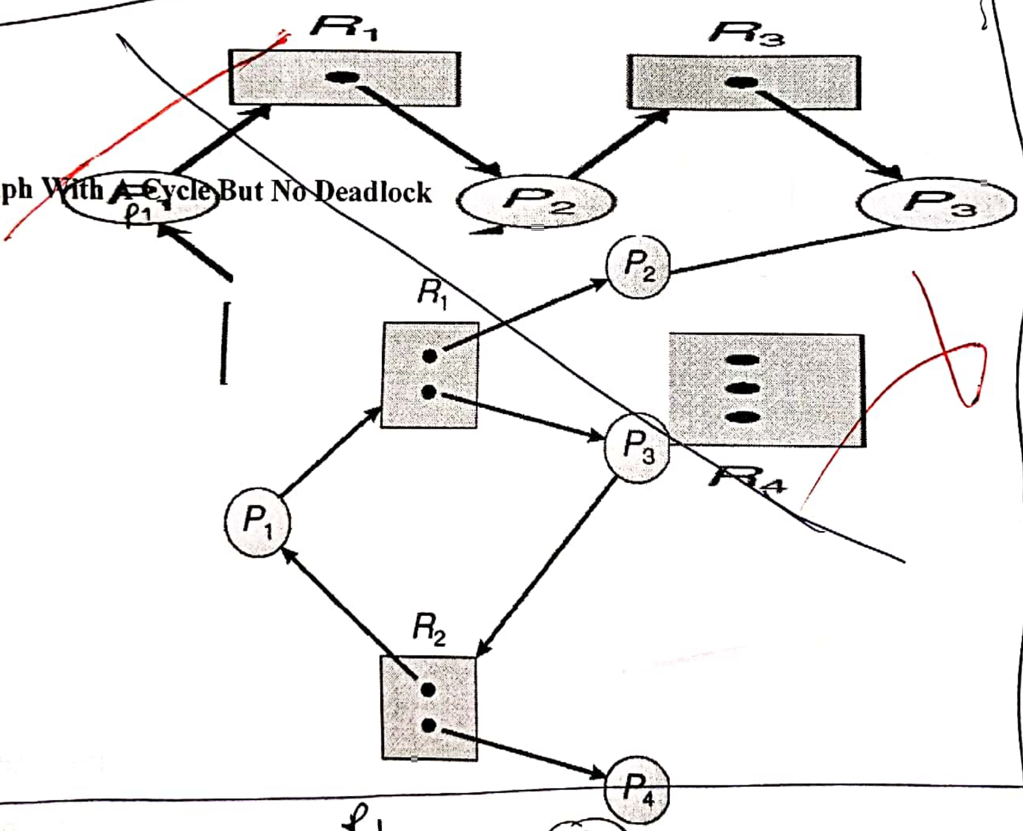


(2) Resource Allocation Graph With A Deadlock

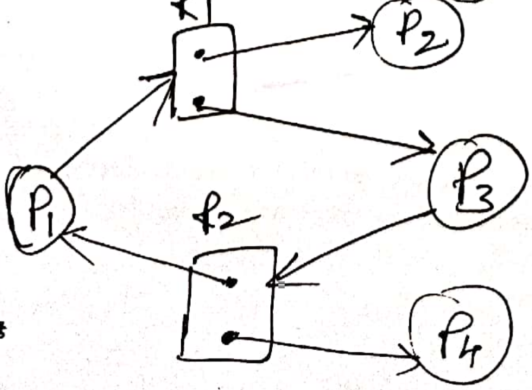


with two minimal cycles.

(3) Graph With A Cycle But No Deadlock



(3)



[redacted]

[redacted]

Fig: Resource-allocation graph with a cycle but no deadlock

Basic Facts

→ If graph contains no cycles no deadlock If graph contains a cycle if only one instance per resource type, then deadlock , necessary and sufficient condition.

→ if several instances per resource type, possibility of deadlock → necessary but not sufficient condition

Methods for Handling Deadlocks

Ensure that the system will *never* enter a deadlock state Allow the system to enter a deadlock state and then recover Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

Restrain the ways request can be made (By ensuring that at least one of these conditions cannot hold, we can prevent the occurence of a deadlock).

1. **Mutual Exclusion** – not required for sharable resources; must hold for non sharable resources
2. **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
   Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
   Low resource utilization; starvation possible are disadvantage.
3. **No Preemption** –
   If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released Preempted resources are added to the list of resources for which the process is waiting Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
4. **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Avoidance

Requires that the system has some additional *a priori* information available

→ Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need

⇒ The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition

⇒ Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

→ **Safe State**

When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state

⇒ System is in safe state if there exists a sequence <*P1*, *P2*, ..., *Pn*> of ALL the processes is the systems such that for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the *Pj*, with *j* < *i* That is:

⇒ If Pi resource needs are not immediately available, then *Pi* can wait until all *Pj* have finished When *Pj* is finished, *Pi* can obtain needed resources, execute, return allocated resources, and terminate When *Pi* terminates, *Pi* +1 can obtain its needed resources, and so on.

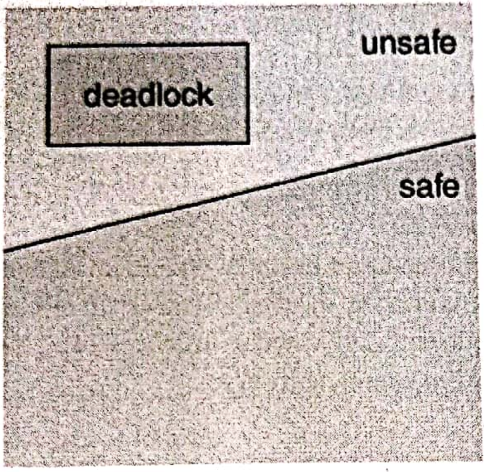
[redacted]

[redacted]

Basic Facts

→ If a system is in safe state ⇒ no deadlocks. If a system is in unsafe state ⇒ possibility of deadlock Avoidance ⇒ ensure that a system will never enter an unsafe state.

Safe, Unsafe , Deadlock State



Note: A safe state is not a deadlock state.

* Conversely, a deadlocked state is an unsafe state.

* Not all unsafe states are deadlocks.

* however, An unsafe state may lead to a deadlock..

* As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states.

Deadlok

Avoidance algorithms

- Single instance of a resource type
- Use a resource-allocation graph
- Multiple instances of a resource type
- Use the banker's algorithm

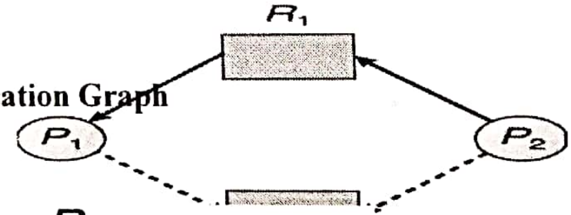
Resource-Allocation Graph Scheme (Variant)

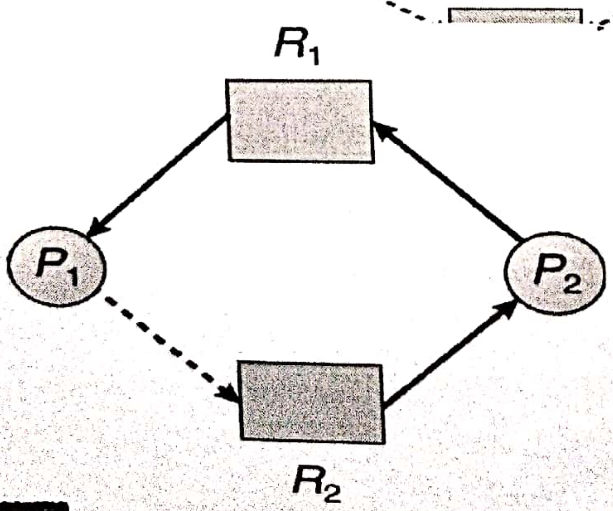
Claim edge *Pi* → *Rj* indicated that process *Pj* may request resource *Rj*; represented by a dashed linen Claim edge converts to request edge when a process requests a resource Request edge converted to an assignment edge when the resource is allocated to the process

→ When a resource is released by a process, assignment edge reconverts to a claim edge Resources must be claimed *a priori* in the system

Resource-Allocation Graph

Unsafe State In Resource-Allocation Graph





[redacted]

[redacted]

[redacted] [redacted]

Resource-Allocation Graph Algorithm

Suppose that process Pi requests a resource Rj

The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker's Algorithm

Multiple instances Each process must a priori claim maximum use When a process requests a resource it may have to wait When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

Available: Vector of length m. If available $[j] = k$, there are k instances of resource type Rj available

Max: $n \times m$ matrix. If $Max\ [i,j] = k$, then process Pi may request at most k instances of resource type Rj

Allocation: $n \times m$ matrix. If Allocation$[i,j] = k$ then Pi is currently allocated k instances of Rj**Need**: $n \times m$ matrix. If $Need[i,j] = k$, then Pi may need k more instances of Rj to complete its task

$$Need\ [i,j] = Max[i,j] - Allocation\ [i,j]$$

Safety Algorithm

```
1. Let Work and Finish be vectors of length m and n, respectively.  Initialize:
     Work = Available
     Finish [i] = false for i = 0, 1, ..., n- 1
2. Find and i such that both:
     (a) Finish [i] = false(b) Needi £ Work
     If no such i exists, go to step 4

3. Work = Work + Allocationi
     Finish[i] = true
     go to step 2

4. If Finish [i] == true for all i, then the system is in a safe state
```

Resource-Request Algorithm for Process P_i

1. *Request* = request vector for process *Pi*. If *Requesti* $[j] = k$ then process *Pi* wants k instances of resource type *Rj*1. If *Requesti* £ *Needi* go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If *Requesti* £ *Available*, go to step 3. Otherwise *Pi* must wait, since resources are not available
3. Pretend to allocate requested resources to *Pi* by modifying the state as follows:

   Available = *Available* – *Request;*

   Allocationi = *Allocationi* + *Requesti;*

   Needi = *Needi* – *Requesti;*

If safe Þ the resources are allocated to Pi

If unsafe Þ Pi must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

n5 processes *P0* through *P4*;

Deadlock Avoidance:
Banker's Algorithm:

⇒ The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.

⇒ The deadlock avoidance algorithm, commonly known as the bankers algorithm is applicable to such a system, but it is less efficient than the resource - allocation graph-scheme.

⇒ When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system.

⇒ when a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.

⇒ If it will, the resources are allocated, otherwise. the process must wait until some other process releases enough resources.

⇒ We need the following data structures, to implement the banker's algorithm, where n is the number of processes in the system and m is the number of resource types:

① Available: A vector of length m indicates the number of available resources of each type. If Available [j], equals k, then k instances of resource type R_j are available.

② **Max**: An $n \times m$ matrix defines the maximum demand of each process. If Max[i][j]. Equals k, then process P_i may request at most k instances of resource type R_j.

③ **Allocation**: An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
If Allocation[i][j] Equals k, then process P_i is currently allocated k instances of resource type R_j.

④ **Need**: An $n \times m$ matrix indicates the remaining resource need of each process. If Need[i][j] Equals k, then process P_i may need k more instances of resource type R_j to Complete its task. Note that

$$\boxed{\text{Need[i][j] Equals Max[i][j]} - \text{Allocation[i][j]}}$$

→ These data structures vary over time in both size & value.

→ The vector Allocation_i specifies the resources currently allocated to process P_i.

→ The vector Need_i specifies the additional resources that process P_i may still request to Complete its task.

I. Safety Algorithm:

This algorithm is for finding out whether or not a system is in a safe state. or unsafe state

1. Let work and Finish be vectors of length m and n, respectively.

   Initialize $\boxed{work = Available}$ and

   $\boxed{Finish[i] = false}$ for $i = 0, 1, \dots, n-1$.

2. Find an index i such that both

   (a) $\boxed{Finish[i] = = false}$

   (b). $\boxed{Need_i \le work}$

   If no such i exists, go to step 4.

3. $work = work + Allocation_i$

   $finish[i] = true$.

   Go to step 2.

4. If $finish[i] == true$ for all i, then the system is in a safe state.

→ This algorithm may require an order of $m \times n^2$ operations to determine whether a state is safe.

II. Resource - Request Algorithm:

This algorithm is for determining whether requests can be safely granted.

⇒ Let $Request_i$ be the request vector for Process i.

⇒ If $Request_i[j] == K$, then Process P_i wants K instances of resource type R_j.

When a request for resources is made by process P_i, the following actions are taken:

1. if $Request_i \leq Need_i$, go to step 2. otherwise, raise an Error condition. since the process has exceeded its maximum claim.

2. If $Request_i \leq Available$, Go to step 3. otherwise, P_i must wait since the resources are not available.

3. Have the system pretend to have allocated the requested resources to Process P_i by modifying the state as follows:

$$\boxed{\begin{aligned} Available &= Available - Request_i;\\ Allocation_i &= Allocation_i + Request_i;\\ Need_i &= Need_i - Request_i; \end{aligned}}$$

⇒ If the resulting resource-allocation state is safe, the transaction is completed, and process P_i is allocated its resources.

⇒ if the new state is unsafe, then P_i must wait for $Request_i$, and the old resource-allocation state is restored.

Deadlock Detection:

① Single Instance of Each Resource Type:

⇒ If all resources have only a single instances then we can define a deadlock-detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph.

⇒ we obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

⇒ an edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs.

⇒ An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges (i) $P_i \rightarrow R_q$ and (ii) $R_q \rightarrow P_j$ for some resource R_q.

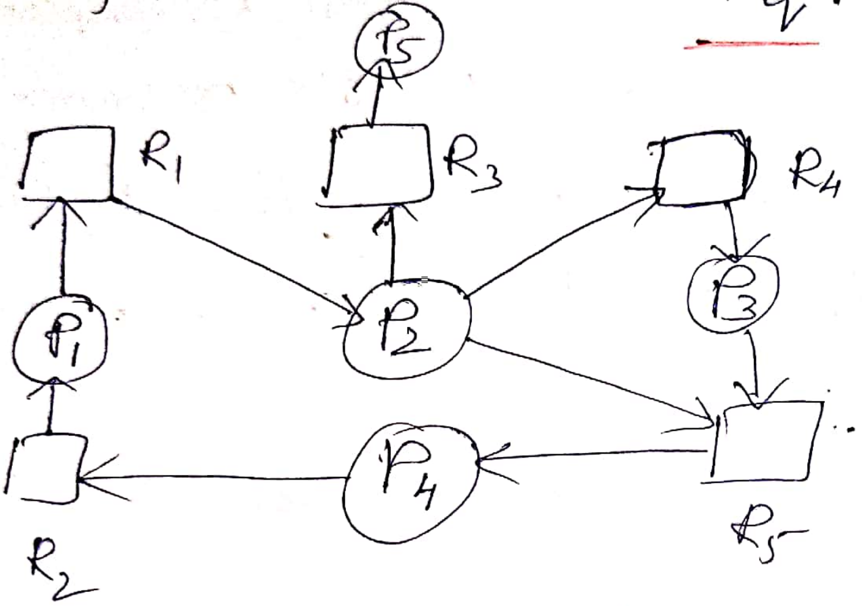


Fig: (a) Resource-allocation graph.

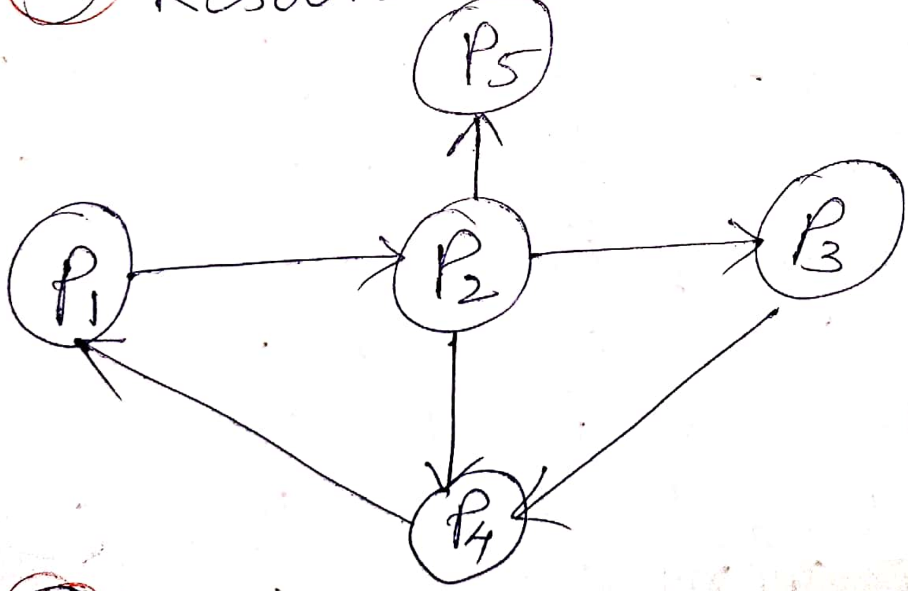


Fig: (b) Corresponding wait-for graph.

⇒ a deadlock Exists in the system if and only if the wait-for graph contains a cycle. ⇒ To detect deadlocks, the system needs to maintain the (i) wait-for graph and periodically invoke an algorithm that (ii) searches for a cycle in the graph.

(2) Several Instances of a Resource Type:

⇒ The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.

Deadlock-Detection algorithm:

Data structures:

(i) Available: A vector of length m indicates the number of available resources of each type.

(ii) Allocation: An n × m matrix defines the number of resources of each type currently allocated to each process.

(iii) Request: An n×m matrix indicates the current request of each process if Request[i][j] Equals K then Process P_i is requesting K more instances of resource type R.

steps **Algorithm: (Deadlock Detection):**

1. Let work and finish be vectors of length m and n, respectively.

Initialise $\boxed{work = Available}$

for $i = 0, 1, \text{-----} n-1$

- if $Allocation_i \neq 0$, then

$finish[i] = false;$

otherwise, $finish[i] = true$.

2. Find an index i such that both

a. $finish[i] == false$

b. $Request_i \leq work$

If no such i exists, go to step-4.

3. $work = work + Allocation_i$

$finish[i] = true$.

Go to step-2.

4. If $finish[i] == false$ for some i, $0 \leq i < n$, then the system is in a deadlock state. if $finish[i] == false$, then Process P_i is deadlocked.

This algorithm requires an order of $m \times n^2$ Operations to detect whether the system is in a deadlocked state

Deadlock Avoidance:

Banker's Algorithm:

Example: Problem: Consider a system with five Processes P_0 through P_4 resource types A, B, C.

" " A with 10 Instances
" " B has 5 "
" " C " 7 " .

At time To, The system has ~~taken~~ the State is

	Allocation A B C	MAX A B C	Available A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

To Find the Need Matrix. (vector)

$$\boxed{\text{Need} = \text{MAX} - \text{Allocation.}}$$

Need = Max − Allocation   Need

P_0 — 7 5 3 − 0 1 0 = 7 4 3

P_1 — 3 2 2 − 2 0 0 = 1 2 2

P_2 — 9 0 2 − 3 0 2 = 6 0 0

P_3 — 2 2 2 − 2 1 1 = 0 1 1

P_4 4 3 3 − 0 0 2 = 4 3 1

We claim that the system is currently in a safe state.
& the safe sequence is

then $< P_1, P_3, P_4, P_2, P_0 >$.

⇒ Suppose, the process P_1 requests
1 instance of A,
2 " " C. A B C

so $Request_1 = (1, 0, 2)$.

⇒ To decide, whether this request can be immediately granted, check that

$Request_1 \leq Available$

$(1,0,2) \leq (3,3,2)$ ✓

which is true.

⇒ Suppose, this request has been fullfilled. then new state of the System is.

	Allocation	Need	Available
	A B C	A B C	A B C
P_0 —	0 1 0	7 4 3	2 3 0
P_1 —	3 0 2	0 2 0	
P_2 —	3 0 2	6 0 0	
P_3 —	2 1 1	0 1 1	
P_4 —	0 0 2	4 3 1	

we must determine whether this new system state is safe.

step-1 Execute Safety Algorithm.

P_1 —

work = Available
= (2,3,0)

$Need_1 \leq work$

$(0,2,0) \leq (2,3,0)$ ✓

then

work = work + Allocation.

= (2,3,0) + (3,0,2)

work = (5,3,2)

finish[1] = true.

P_3

$Need_2 \leq work$

~~(2,1,1)~~

$(0,1,1) \leq (5,3,2)$ ✓

then

work = (5,3,2) + (2,1,1)

work = (7,4,3) ✓

finish[3] = true.

P_4 $\quad Need_4 \leq work$

$(4,3,1) \leq (7,4,3)$

$work = (7,4,3) + (0,0,2)$

$= (7,4,5).$

finish[4] = true.

P_0. $\quad Need_0 \leq work$

$(7,4,3) \leq (7,4,5)$

$work = (7,4,5) + (0,1,0)$

$= (7,5,5).$

Finish[0] = true.

P_2 $\quad Need_2 \leq work.$

$(6,0,0) \leq (7,5,5)$

$work = (7,5,5) + (3,0,2)$

$= (10,5,7).$

Finish[2] = true.

So, the safe sequence is:

$< P_1, P_3, P_4, P_0, P_2 >.$

Satisfies the safety requirement. Hence we can immediately grant the request of Process P_1.

Deadlock Detection Algorithm:

(Applicable for Multiple Instances of each resource type)

Example: Problem: a System with five processes P_0 through P_4 and

⇒ three resource types, A, B and C.

⇒ Resource type A – has seven Instances.

Resource type	A –	has	seven	Instances.
"	"	B –	" 2	"
"	"	C –	6	"

at time T_0, System state.

	Allocation A B C	Request A B C	Available A B C
P_0 —	0 1 0	0 0 0	0 0 0
P_1 —	2 0 0	2 0 2	
P_2 —	3 0 3	0 0 0	
P_3 —	2 1 1	1 0 0	
P_4 —	0 0 2	0 0 2	

⇒ We claim that the System is not in a deadlocked state. Indeed, If we Execute algorithm, we will find that the sequence $< P_0, P_2, P_3, P_1, P_4 >$ results in Finish[i] == true for all i:

⇒ Suppose, now that process P_2 makes one additional request for an instance of type C.

⇒ The Request matrix is modified as follows:

	Request A	B	C
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

- work = Available = (0,0,0)
- $Request_2 \le work$
  $(0,0,1) \le (0,0,0)$
  → which is false
- So, finish[2] = false
- deadlock occurs

⇒ we claim that the system is now deadlocked. Although we can reclaim the resources held by process P_0, the number of available resources is not sufficient to fullfile the requests of the other processes.

Thus, a deadlock exists, consisting of processes P_1, P_2, P_3 & P_4.

Deadlock is detected

* Recovery * from Deadlock

when a detection algorithm determines that a deadlock Exists, Several alternatives are available -

① Simply to abort one or more processes to break the circular wait.

② To preempt some resources from one or more of the deadlocked Processes.

I. Process * Termination:

To Eliminate deadlocks by aborting a Process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated Processes.

- **Abort all deadlocked Processes:**

→ This method will break the deadlock cycle, but at great Expense.

⇒ The deadlocked processes may have computed for a long-time, and the results of these partial Computations must be discarded and will have to be recomputed later.

- **Abort one process at a time until the deadlock cycle is Eliminated:**

⇒ This method incurs a overhead, since after Each process is aborted, a deadlock - detection algorithm must be invoked to determine whether any processes are still deadlocked.

⇒ Aborting a process may not be easy. If the process was in the middle of updating a file, terminating it will leave that file in an incorrect state.

⇒ Similarly, if the process was in the middle of printing data on a printer, the system must reset the printer to a correct state before printing the next job.

⇒ If the partial termination method is used, then we must determine which deadlocked process should be terminated.

⇒ The question is basically an Economic one, we should abort those processes whose termination will incur the minimum cost.

⇒ Many factors may affect which process is chosen including:

1. what the priority of the process is.
2. How long the process has computed and how much longer the process will compute before completing its designated task.
3. How many and what types of resources the process has used.
4. How many more resources the process needs in order to complete.
5. How many processes will need to be terminated
6. whether the process is interactive or batch.

II Resource Preemption:

⇒ To Eliminate deadlocks using resource Preemption, we preempt some resources from processes and give these resources to other Processes until the deadlock cycle is broken.

⇒ If preemption is required to deal with deadlocks then three issues need to be addressed:

1. Selecting a victim:

⇒ which resources and which Processes are to be preempted?

⇒ we must determine the order of Preemption to minimize cost.

Cost factors may include:

→ the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed during its Execution.

2. Rollback: If we preempt a resource from a Process, what should be done with that process?

⇒ It cannot Continue with its normal Execution

⇒ it is missing Some needed resource.

⇒ we must roll back the Process to Some safe state and restart it from that state.

3. Starvation: How do we Ensure that starvation will not occur? i.e. how can we guarantee that resources will not always be preempted from the Same process? So we must Ensure that a Process can be picked as a victim only a (small) finite number of times. The most common solution is to include the no. of rollbacks in the cost factor.

1. Do... and keys:

Inter Process Communication Mechanisms.

* Pipes:

→ A pipe acts as a conduit (tube through which something such as water passes), allowing two process to communicate.

⇒ Pipes were one of the first IPC mechanisms in early UNIX systems and typically provide one of the simpler ways for processes to communicate with one another.

⇒ In implementing a pipe, four issues must be considered:

1. Does the pipe allow unidirectional communication or bidirectional communication?
2. If two-way communication is allowed, is it half duplex (data can travel only one way at a time). or full duplex (data can travel in both directions at the same time). ?
3. Must a relationship (such as parent-child) exist between the communicating processes?
4. can the pipes communicate over a network, or must the communicating processes reside on the same machine?

⇒ Two common types of pipes used on both UNIX and windows systems.

1. ordinary pipes (unnamed pipes).
2. Named pipes (FIFO)

① Ordinary Pipes:

→ Ordinary pipes are unidirectional, allowing only one-way Communication. If two-way Communication is required, two pipes must be used, with each pipe sending data in a different direction.

```
#include <unistd.h>
 int pipe(int filedes[2]);
                         Returns: 0 if ok,
                                  -1 on Error.
```

→ fd[0] is the read-End of the pipe.

→ fd[1] is the write End "

→ Pipes can be accessed using ordinary read( ) & write( ) System-calls

An ordinary Pipe cannot be accessed from outside the process that Creates it. Thus a parent process creates a pipe and uses it to Communicate with a child process it Creates via fork( )

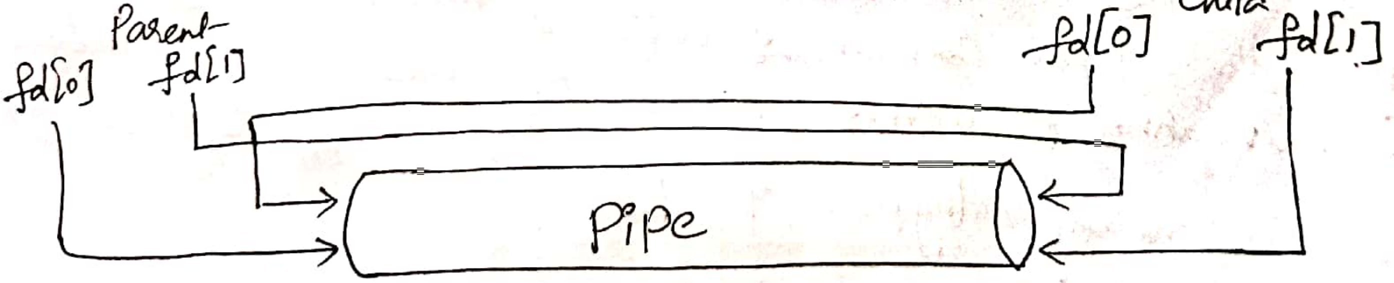


Fig: File descriptors for an ordinary pipe.

* Named pipes: (fIfo).

Ordinary pipes provide a simple communication mechanism between a pair of processes. & these exist only while the processes are are communicating with one other., once the processes have finished communication & terminated, the ordinary pipe ceases to exist.

with FIFO

→ Communication can be bidirectional,

→ no parent-child relationship is required.

→ once a named pipe is established, several processes can use it for communication.

↳ Named pipes provide a much more powerful communication tool.

⇒ A FIFO is created with the mkfifo() system call and manipulated with the ordinary open(), read(), write(). and ,lose() system calls.

```
#include < Sys/stat.h>
int mkfifo(Const char * pathname,
              mode_t mode);
                         Returns: 0 if ok,
                                 -1 on Error.
```

⇒ It will continue to exist until it is explicitly deleted from the file system.

⇒ Although FIFO's allow bidirectional communication, only half-duplex transmission is permitted.

⇒ If data must travel in both directions, two FIFOs are typically used.

⇒ Additionally, the communicating processes must reside on the same machine.

note: ⇒ Sockets must be used if intermachine communication is required (different computers).

IPC

* Identifiers and keys:

```
#include <sys/ipc.h>
key_t ftok(const char *path, int id);
                Returns: key if ok,
                     -1 on Error.
```

⇒ The path argument must refer to an Existing file. Only the lower 8 bits of id are used when generating the key.

⇒ The client and the server can agree on a pathname and Project ID (the project ID is a character value between 0 and 255) and call the function ftok to Convert these two values into a key.

* Message Queues:

→ A message queue is a linked list of messages stored within the Kernel and identified by a message queue identifier.

→ A new queue is created or an Existing queue opened by msgget. New messages are added to the End of a queue by msgsnd. Every message has a positive long integer type field,

→ a non-negative length, and the actual data types all of which are specified to msgsnd when the message is added to a queue.

⇒ Messages are fetched from a queue by msgrcv. we don't have to fetch the messages in a 'first-in, first-out order. Instead, we can fetch messages based on their type field.

⇒ The first function normally called is msgget to Either open an Existing queue or Create a new queue.

①
```
#include <sys/msg.h>
int msgget(key_t key, int flag);
                    Returns: message queue ID
                             if OK
                          -1 on Error.
```

② The msgctl function performs various on a queue.

```
  #include <sys/msg.h>.
int msgctl(int msqid, int cmd,
                struct msqid_ds *buf);
              Returns : 0 If OK,
                       -1 on Error.
```

The cmd argument specifies the command to be performed on the queue specified by msqid

→ Data is placed on-to a message queue by calling msgsnd.

(3)

```
#include < Sys/msg.h >
int msgsnd( int msqid, Const void *ptr,
        Size_t nbytes, int flag);

            Returns : 0 if OK,
                     -1 on Error.
```

⇒ Each message is composed of a positive long integer type field, a non-negative length (nbytes), and the actual data bytes (Corresponding to the length). Messages are always placed at the End of the queue.

The ptr argument points to a long integer that Contains the positive integer message type, and it is immediately followed by the message data.

⇒ Messages are retrieved from a queue by msgrcv.

4

```
#include < sys/msg.h>
ssize_t msgrcv( int msqid, void *ptr,
        size_t nbytes, long type, int flag);
Returns: size of data portion of message
         if ok,   -1 on Error.
```

The ptr - argument points to a long integer followed by a data buffer for the actual message data.

nbytes — the size of the data buffer.

The type argument lets us specify which message we want.

type == 0   The first message on the queue is returned.

type > 0   The first message on the queue whose message type equals type is returned.

type < 0   The first message on the queue whose message type is the lowest value less than or equal to the absolute value of type is returned.

* Shared Memory:

Shared memory allows two or more processes to share a given region of memory. This is the fastest form of IPC, because the data does not need to be copied between the client and the server.

The only trick in using shared memory is synchronizing access to a given region among multiple processes.

⇒ If the server is placing data into a shared memory region, the client should n't try to access the data until the server is done., semaphores are used to synchronize shared memory access.

1. The first function called is usually shmget, to obtain shared memory identifier.

```
#include <sys/shm.h>
int shmget( key_t key, size_t size, int flag)
```

Returns: shared memory ID if OK,
-1 on Error.

2. The shmctl function is the catchall for various shared memory operations.

```
#include <sys/shm.h>
int shmctl (int shmid, int cmd,
    struct shmid_ds *buf );
```

Returns: 0 if ok,
-1 on Error.

③ #include <Sys/shm.h>

void *shmat(int shmid, Const void *addr,
int flag);

Returns: pointer to shared memory segment
if ok, -1 on Error.

the segment is attached depends on the addr argument and whether the SHM_RND bit is specified in flag.
(round)

if addr =0, The segment is attached at the first available address selected by the kernel.
if addr is nonzero, and SHM_RND is not specified, the segment is attached at the address given by addr.

④ #include <Sys/shm.h>

int shmdt(void *addr);

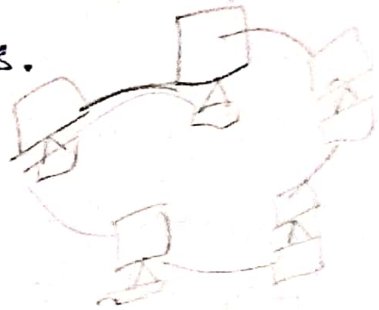
Returns: 0 if OK,
-1 on Error.

when we're done with a shared memory segment we call shmdt to detach it.

Message - Passing Systems:

Message passing provides a mechanism to allow processes to Communicate and to Synchronize their actions without Sharing the same address Space and is particularly useful in a distributed Environment, where the Communicating processes may reside on different Computers Connected by a network.

For Example, a chat program used on the WWW could be designed so that chat participants Communicate with one another by Exchanging messages.



It provides two operations:

→ send(message)

→ receive(message).

16 MB

5 Members

⇒ Messages sent by a process can be of Either fixed or variable size.

⇒ If only fixed-sized messages can be sent, the system-level implementation is straightforward. however, this restriction makes the task of programming more difficult.

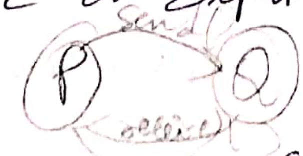
⇒ variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler.

⇒ If processes P and Q want to communicate, they must send messages to and receive messages from each other.

⇒ A communication link must exist between them. This link can be implemented in a variety of ways.

There are several methods for logically implementing a link and the send( ) / receive(·) operations:

- Direct & indirect communication (Naming)
- Synchronous & asynchronous communication (Synchronization)
- Automatic & explicit buffering. (Buffering)



① Naming:

Processes that want to communicate must have a way to refer to each other.

SMS

They can use either direct & indirect communication.

Shooting

→ **Direct communication:**

unity

Each process that wants to communicate must explicitly name the recipient & sender of the communication.

india

- send (P, message) — Send a message to process P.
- receive(Q, message) — Receive a message from process Q.

⇒ A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.

- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.

⇒ This scheme exhibits symmetry in addressing. i.e., both the sender process and the receiver process must name the other to communicate.

⇒ A variant of this scheme employs asymmetry in addressing. Here, only the sender names the recipient, the recipient is not required to name the sender.

- send(P, message) — send a message to process P.
- receive (id, message):— Receive a message from any process, the variable id is set to the name of the process with which communication has taken place.

⇒ The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions.

Indirect Communication:

The messages are sent to and received from mailboxes, or ports.

A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification.

The send( ) and receive( ) primitives are defined as follows:

- send(A, message) — Send a message to mailbox A.
- receive(A, message) — Receive message from mailbox A.

a Communication link has the following Properties:

- A link is Established between a pair of Processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between Each pair of Communicating processes, there may be a number of different links, with Each link corresponding to one mailbox.

- A mailbox may be owned either by a Process or by the operating System.

If the mailbox is owned by a process (i.e, the mailbox is part of the address space of the process), then we distinguish between the owner (which can only receive messages through this mailbox) and the user (which can only Send messages to the mailbox). Since Each mailbox has a unique owner, there can be no confusion about which process should receive a message sent to this mailbox.

When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.

In contrast, a mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process.
The operating system then must provide a mechanism that allows a process to do the following:

- Create a new mailbox.
- Send and receive messages through the mailbox
- Delete a mailbox.

Synchronization:

Communication between processes takes place through calls to send( ) and receive( ) primitives.
There are different design options for implementing each primitive.
Message passing may be either **blocking** or **nonblocking** — also known as **synchronous** and **asynchronous**.

- **Blocking send**: The sending process is blocked until the message is received by the receiving process or by the mailbox.

- Nonblocking send: The sending process sends the message and resumes operation.
- Blocking receie: The receiver blocks until a message is available.
- Nonblocking receive: The receiver retrieves either a valid message or a null.

Different combinations of send( ) and receive( ) are possible. when both send( ) and receive( ) are blocking, we have a rendezvous between the sender and the receiver.

Buffering:

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways.

no buffering

→ Zero Capacity: The queue has a maximum length of zero. Thus, the link cannot have any messages waiting in it. In this case, the sender must block until the receipient receives the message.

automatic buffering

→ Bounded Capacity: The queue has finite length n; thus at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue, and the sender can continue execution without waiting.

Unbounded capacity: The queue's length is infinite thus any number of messages can wait in it. The sender never blocks.

* Peterson's solution:

→ It is restricted to two Processes that alternate Execution between their critical sections and remainder sections.

The processes are P_0 and P_1

Peterson's solution requires the two Processes to share two data items:

```
int turn;
boolean flag[2];
```

⇒ if $\boxed{turn == i}$, then Process P_i is allowed to Execute in its critical section.

⇒ The flag array is used to indicate if a process is ready to Enter its critical section.

Eg: if flag[i] is true, this value indicates that P_i is ready to Enter its critical section

To Enter the critical section, Process P_i first sets flag[i] to be true and then sets turn to the value j, asserting that if the other process which to Enter the critical section, it can do so.

```
do
{
    flag[i] = TRUE;
    turn = j;
    while ( flag[j] && turn==j);

        Critical Section;

    flag[i] = FALSE;

        remainder Section.
} while(TRUE);
```

Fig: The Structure of Process P_i in Peterson's Solution.

⇒ To prove that this solution is correct. we need to show that:

1. Mutual Exclusion is Preserved.
2. The Progress requirement is satisfied.
3. The bounded-waiting requirement is met.

To Prove Property 1,

P_i Enters its critical section only if Either flag[j]== false or turn == i.

Any solution to the critical-section problem requires a simple tool — a lock.

Race Conditions are prevented by requiring that critical regions be protected by locks.

i.e., a Process must acquire a lock before Entering a critical section. it releases the lock when it Exits the critical section.

```
do {
        acquire lock
            critical section
        release lock
            remainder section
} while (TRUE).
```

Fig: Solution to the critical-section problem using locks.

we start by Presenting Some simple hardware instructions that are available on many Systems and Showing how they can be used Effectively in solving the critical-section problem. Hardware features can make any Programming task Easier and improve System Efficiency.

→ The critical-section problem could be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified.

Unfortunately, this solution is not feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, as the message is passed to all the processors. this message passing delays entry into each critical section, and system efficiency decreases. Also consider the effect on a system's clock if the clock is kept updated by interrupts.

→ Many modern computer systems therefore provide special hardware instructions that allow us either to 1) test and modify the content of a word or 2) to swap the contents of two words atomically — that is, as one uninterruptible unit. we can use these special instructions to solve the critical-section problem.

The TestAndSet( ) instruction can be defined as

```
boolean TestAndSet( boolean *target)
    {
        boolean rv = *target;
        *target = TRUE;
        return rv;
    }.
```

→ The important characteristic of this instruction is that it is <u>Executed atomically</u> thus,

→ if two TestAndSet() instructions are Executed Simultaneously (Each on a different CPU), they will be Executed sequentially in some arbitrary order.

→ If the machine supports the <u>TestAndSet()</u> instruction, then we can implement mutual Exclusion by declaring a <u>Boolean variable lock</u>, initialised <u>to false.</u>

```
do{
      while(TestAndSet(&lock))
            ; // do nothing

      // Critical Section      lock - TRUE

      lock = FALSE;

      // remainder section
   }while(TRUE);
```

Fig: Mutual-Exclusion implementation with TestAndSet().

⇒ In contrast to the TestAndSet( ) instruction, the Swap( ) instruction, operates on the contents of two words, and is executed atomically.

⇒ If the machine supports the Swap( ) instruction, then mutual exclusion can be provided as follows.

⇒ A global Boolean variable lock is declared and is initialized to false. In addition, each process has a local Boolean variable key.

⇒ Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement.

global Boolean lock = false   key = false

```
void swap(boolean *a, boolean *b)
{
    boolean temp = *a;
        *a = *b;
        *b = temp;
}
```

Fig: The definition of the Swap( ) instruction

```
do {
    ~~while ( TestAndSet( &lock))~~
      ~~; // do nothing~~
                                     key = TRUE
                                     while(key == TRU
                                     Swap(&lock, &k
      // critical section
    lock = FALSE;
      // remainder section
} while (TRUE);
```

Fig: Mutual-Exclusion

⇒ We Present another algorithm using the TestAndSet( ) Enstruction that satisfies all the Critical-section requirements.

The common data structures are:

boolean waiting[n];
boolean lock;

these data structures are initialized to false.

⇒ To prove that the mutual Exclusion requirement is met, that Process P_i can Enter its critical section only if either

waiting[i] == false & key == false

⇒ The value of key can become false only if the TestAndSet( ) is Executed.

⇒ The first Process to Execute the TestAndSet( ) will find key == false, all others must wait.

⇒ The variable waiting[i] can become false only if another process leaves its critical section. only one waiting[i] is set to false, maintaining the mutual-Exclusion requirement.

→ ~~Cooling and power Availability:~~

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while( waiting[i] && key)
            key = TestAndSet( &lock);
        waiting[i] = FALSE;

        // Critical section.

    j = ( i + 1) % n;
    while(( j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;

        // remainder section
} while ( TRUE);
```

Fig: Bounded-waiting mutual Exclusion with TestAndSet

→ To prove that the progress requirement is met, we note that the arguments presented for mutual Exclusion also apply here, since a process exiting the critical section either sets lock ~~t~~o false or sets waiting[j] to false. Both allow a process that is waiting to enter its critical section to proceed.

⇒ To prove that the bounded-waiting requirement is met, we note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i+1, i+2, ..., n-1, 0, ..., i-1).

⇒ It designates the first process in this ordering that is in the entry section (waiting[j] == true) as the next one to enter the critical section.

⇒ Any process waiting to enter its critical section will thus do so within n-1 turns.

* Semaphores:

The hardware-based solutions to the critical-section problem are complicated for application programmers to use.

To overcome this difficulty, we can use a synchronization tool called a semaphore.

A $\boxed{\text{semaphore } S}$ is an integer variable that, is accessed only through two standard atomic operations: wait( ) and signal( ).

The $\boxed{\text{wait( )}}$ operation was originally termed $\boxed{P}$ (from the Dutch 'Proberen, "to test").

Signal( ) was originally called V (from Verhogen, "to increment").

The definition of wait ( ) is as follows:

```
wait ( S ) {
        while ( S<=0)
            ; // no-op

        S--;
}
```

The definition of signal( ) is as follows:

```
Signal( S) {
    S++;
}
```

All modifications to the integer value of the semaphore in the wait( ) and signal( ) operations must be executed indivisibly. (atomic)

⇒ that is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

→ In addition, in the case of wait(S), the testing of the integer value of S ($S \leq 0$), as well as its possible modifications ($S--$), must be executed without interruption.

Usage:

Two types of Semaphores: The value of a

① Counting semaphore ⇒ can range over an Unrestricted domain.

② Binary semaphore ⇒ can range only between 0 and 1.

Binary Semaphores:

On some systems, binary semaphores are known as mutex locks. as they are, locks that provide mutual Exclusion.

⇒ we can use binary semaphores to deal with the critical-section problem for multiple processes.

The n processes share a semaphore, mutex, initialized to 1. Each process. P_i is organised as:

```
do
  {  wait(mutex);
          // critical section
       signal(mutex);
        // remainder section
  } while( TRUE);
```

Fig: Mutual-Exclusion implementation with semaphores.

Counting semaphores:

⇒ Counting Semaphores can be used to Control access to a given resource Consisting of a finite number of instances. The Semaphore is initialized to the number of resources available.

⇒ Each Process that wish to use a resource performs a wait() operation on the Semaphore (thereby decrementing the count). 3 Print
count = 3 ⇒

⇒ when a process releases a resource, it performs a signal() operation (incrementing the count).

⇒ when the count for the semaphore goes to 0, all resources are being used.

⇒ After that, Processes that wish to use a resource will block until the count becomes greater than 0.

⇒ The main disadvantage of the Semaphore definition given here is that it requires busy waiting.

⇒ while a Process is in its Critical Section any other Process that tries to enter its c.s must loop Continuously in the entry code. This Continual looping is clearly a problem in a real multiprogramming System, where a single CPU is shared among many Processes.

⇒ Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a Spinlock because the process "spins" while waiting for the lock.

⇒ To overcome the need for busy waiting, we can modify the definition of the wait( ) and signal( ) semaphore operations.

→ when a process executes the wait( ) operation and finds that the semaphore value is not +ve, it must wait.
However, rather engaging in busy waiting, the process can block itself.

⇒ The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

⇒ A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal( ) operation.

⇒ The Process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state.

⇒ The process is then placed in the ready queue.

To implement Semaphores under this definition, we define a semaphore as a "C" struct:

```
typedef    struct {
             int  value;
             struct  Process  *list;
        } semaphore;
```

⇒ Each semaphore has an integer value and a list of Processes list.

⇒ when a process must wait on a Semaphore, it is added to the list of processes.

⇒ A signal() operation removes one process from the list of waiting processes and awakens that Process.

The wait() semaphore operation can now be defined as.

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

The signal() semaphore operation can now be defined as:

```
Signal( Semaphore *S)
{
        S -> value++;
    if( S-> Value <=0)
    {
        remove a process P from
                        S -> list;
        wakeup(P);
    }
}.
```

⇒ The block() operation suspends the process that invokes it.

⇒ The wakeup(P) operation resumes the Execution of a blocked process P.

⇒ These two operations are provided by the operating System as basic System calls.

The list of waiting Processes can be Easily implemented by a link field in Each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs.

One way to add and remove processes from the list so as to ensure bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue.

It is critical that semaphores be executed atomically.

⇒ We must guarantee that no two processes can execute wait() and signal() operations on the same semaphore at the same time. This is a critical-section problem.

→ In a single-processor environment, we can solve it by simply preventing interrupts during the time the wait() and signal() operations are executing. This scheme works in a single-processor environment because, once interrupts are prevented, instructions from different processes cannot be interleaved. Only the currently running process executes until interrupts are reenabled and the scheduler can regain control.

⇒ In a multiprocessor environment, interrupts must be disabled on every processor, otherwise, instructions from different processes (running on different processors) may be interleaved in some arbitary way.

⇒ Disabling interrupts on every processor can be a difficult task and furthermore can seriously decrease performance. Therefore, SMP systems must provide alternative locking techniques — such as spinlocks — to ensure that wait() and signal() are performed atomically.

Deadlocks and Starvation:

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal() operation. when such a state is reached, these processes are said to be deadlocked.

Eg: Consider a system consisting of two processes, P_0 and P_1, each accessing two semaphores, S and Q, set to the value 1:

S=1 P_0	P_1 Q=1
wait(S);	wait(Q);
wait(Q); Q=0	wait(S); S=0
⋮	⋮
signal(S);	signal(Q);
signal(Q);	signal(S);

Priority inversion:

A Scheduling challenge arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process — or a chain of lower-priority processes.

Since kernel data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource.

The situation becomes more complicated if the lower-priority process is preempted infavor of another process with a higher priority.

Eg: we have three processes, L, M, and H, whose priorities follow the order L<M<H.

Assume that process H requires resource R, which is currently being accessed by process L. Ordinarily, Process H would wait for L to finish using resource R. However now Suppose that process M becomes runnable, thereby preempting process L.

Indirectly, a process with a lower priority - process M - has affected how long process H must wait for L to relinquish resource R.

This problem is known as priority inversion. It occurs only in system with more than two priorities. This problem can be solved by implementing a priority-inheritance protocol.

According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources that in question. (required)

→ when they are finished, their priorities revert to their original values.

⇒ In the above example, a priority-inheritance protocol would allow process L to temporarily inherit the priority of process H, thereby preventing process M from preempting its execution. when process L had finished using resource R, it would relinquish its inherited priority from H and assume its original priority. Because resource R would now be available, to process H - not M - H would run next.

*Classic Problems of Synchronization:

① The Bounded-Buffer Problem:

we assume that the pool consists of 'n' buffers, Each capable of holding One Item.

The mutex semaphore provides mutual Exclusion for accesses to the buffer pool and is initialised to the value 1. mutex = 1

The Empty and full semaphores count the number of Empty and full buffers.

The semaphore Empty is initialised to the value n. Empty = n. The Smaphore full is initialized to the value 0: full = 0

```
do
{ ....
  //Produce an item in nextp
  -----
  wait(Empty);
  wait(mutex)
  -----
  //add nexp to buffer
  -----
  signal(mutex);
  Signal(full);
} while (TRUE);
```

fig: The structure of the Producer Process.

```
do {
  wait(full);
  wait(mutex);
  ------
  //remove an item from buffer to nextc
  ------
  Signal(mutex);
  Signal(Empty);
  //consume the item in nextc
} while(TRUE);
```

fig: The structure of the Consumer Process.

(2) The Readers-writers problem:

Suppose that a database is to be shared among several Concurrent Processes.

Some of these processes may want only to read the database, (reffered as readers) whereas others may want to update (that is, to read and write) the database. (reffered as writers.).

If two readers access the shared data simultaneously, no harmful Effects will result.

However, if a writer and some other process (Either a reader or a writer) access the database simultaneously, confusion may Ensure.

To Ensure that these difficulties do not arise, we require that the writers have Exclusive access to the shared database while writing to the database. This synchronization problem is refferred to as the readers - writers problem.

The readers – writers problem has several Variations:

(1) No reader be kept waiting unless a writer has already obtained permission to use the shared object. (i.e) no reader should wait for other readers to finish simply because a writer is waiting.

② once a writer is ready, that writer performs its write as soon as possible.

(i.e.) if a writer is waiting to access the object, no new readers may start reading.

- A solution to either problem may result in starvation.

i.e. In the first case, writers may starve,

In the second case, readers may starve.

① A solution to the first readers-writers problem:

The reader processes share the following data structures:

```
semaphore mutex , wrt;
int    readcount;
```

→ The semaphores mutex and wrt are initialised to 1. readcount is initialised to 0.

→ The semaphore wrt is common to both reader and writer processes.

→ The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated.

⇒ The readcount variable keeps track of how many processes are currently reading the object.

⇒ The semaphore wrt functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section.

* It is not used by readers who Enter or Exit while other readers are in their critical sections.

```
do
{
    wait(wrt);
    - - - - - - -
    // writing is performed
    - - - - - - -
    Signal (wrt);
} while (TRUE);
```

Fig: The structure of a writer Process.

```
do
{
    wait(mutex);
    read count ++;
    if (read count == 1)
        wait(wrt);
    Signal(mutex);
    - - - - - -
    // reading is performed
    - - - - -
    wait(mutex);
    read count --;
    If (read count == 0)
        Signal (wrt);
    Signal (mutex);
} while (TRUE);
```

Fig: The structure of a reader Process.

⇒ If a writer is in the critical section and 'n' readers are waiting, then one reader is queued on wrt, and n-1 readers are queued on mutex.

⇒ When a writer executes signal(wrt), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

The readers-writers problem and its solutions have been generalized to provide reader-writer locks, one some systems.

→ Acquiring a reader-writer lock requires specifying the mode of the lock: either read or write access.

→ when a process whishes only to read shared data, it requests the reader-writer lock in read mode.

→ a process whishing to modify the shared data must request the lock in write mode.

→ Multiple processes are permitted to concurrently acquire a reader-writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.

Reader-writer locks are most useful in the following situations:

→ In applications where it is easy to identify which processes only read shared data and which processes only write shared data.

⇒ In applications that have more readers than writers.

The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader-writer lock.

③ The Dining-philosophers problem:

Consider five philosophers who spend their lives thinking and Eating. The philosophers share a circular table surrounded by five chairs, Each belonging to one philosopher.

In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.

When a philosopher thinks, she does not interact with her colleagues.

From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her.

⇒ A philosopher may pick up only one chopstick at a time.

She cannot pick up a chopstick that is already in the hand of a neighbor.

When a hungry philosopher has both her chopsticks at the same time, she Eats without releasing her chopsticks. When she is finished Eating, She puts down both of her chopsticks and starts thinking again.

One simple solution is to represent Each chopstick with a semaphore. A philosopher tries to grab a chopstick by Executing a wait() operation on that semaphore. She releases her chopsticks by Executing the signal() operation on the appropriate semaphores.

The shared data are:

```
Semaphore chopstick[5];
```

where all the elements of chopstick are initialized to 1.

```
do{
    wait(chopstick[i]);
    wait(chopstick[(i+1)%5]);
    ......

    //Eat
    .....

    signal(chopstick[i]);
    signal(chopstick[(i+1)%5]);

    .....
    //think
    .....
}while(TRUE);
```

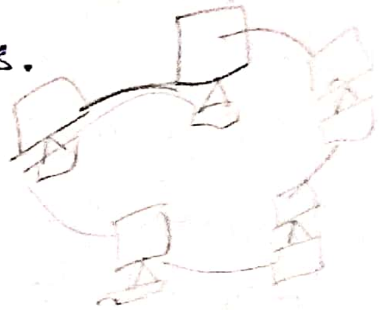
fig: The Structure of philosopher i.

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. when each philosopher tries to grab her right chopstick, she will be delayed forever.

Message - Passing Systems:

Message passing provides a mechanism to allow processes to Communicate and to Synchronize their actions without Sharing the same address Space and is particularly useful in a distributed Environment, where the Communicating processes may reside on different Computers Connected by a network.

For Example, a chat program used on the WWW could be designed so that chat participants Communicate with one another by Exchanging messages.



It provides two operations:

→ send(message)

→ receive(message).

16MB
5 Members

⇒ Messages sent by a process can be of Either fixed or Variable size.

⇒ If only fixed-sized messages can be sent, the system-level implementation is straightforward. however, this restriction makes the task of programming more difficult.

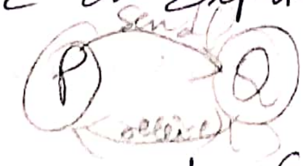
⇒ Variable-sized messages require a more Complex system-level implementation, but the programming task becomes simpler.

⇒ If processes P and Q want to Communicate, they must send messages to and receive messages from each other.

⇒ A Communication link must Exist between them. This link can be implemented in a variety of ways.

there are several methods for logically implementing a link and the send( ) / receive(·) operations:

- Direct & indirect Communication (Naming)
- Synchronous & asynchronous Communication (Synchronization)
- Automatic & Explicit buffering. (Buffering)



① Naming:

Processes that want to communicate must have a way to refer to each other.

SMS

they can use either direct & indirect Communication.

→ Direct Communication:

Each process that wants to Communicate must Explicitly name the recipient & sender of the Communication.

India

- send (P, message) — Send a message to process P.
- receive(Q, message) — Receive a message from Process Q.

⇒ A Communication link in this scheme has the following Properties:

- A link is Established automatically between Every pair of processes that want to Communicate. The processes need to know only each other's identity to Communicate.

- A link is associated with Exactly two processes.
- Between Each pair of Processes, there Exists Exactly one link.

⇒ This scheme Exhibits symmetry in addressing. i.e., both the sender process and the receiver process must name the other to communicate.

⇒ A variant of this scheme employs asymmetry in addressing. Here, only the sender names the recipient, the recipient is not required to name the sender.

- send(P, message) :- send a message to process P.
- receive (id, message) :- Receive a message from any process, the variable id is set to the name of the process with which communication has taken place.

⇒ The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions.

Indirect Communication:

the messages are sent to and received from mailboxes, or ports.

A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification.

The send( ) and receive( ) primitives are defined as follows:

- send(A, message) — Send a message to mailbox A.
- receive(A, message) — Receive message from mailbox A.

a Communication link has the following Properties:

- A link is Established between a pair of Processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between Each pair of Communicating processes, there may be a number of different links, with Each link corresponding to one mailbox.

- A mailbox may be owned either by a Process or by the operating system.

If the mailbox is owned by a process (i.e, the mailbox is part of the address space of the process), then we distinguish between the owner (which can only receive messages through this mailbox) and the user (which can only send messages to the mailbox). Since Each mailbox has a unique owner, there can be no confusion about which process should receive a message sent to this mailbox.

When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.

In contrast, a mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process.
The operating system then must provide a mechanism that allows a process to do the following:

- Create a new mailbox.
- Send and receive messages through the mailbox
- Delete a mailbox.

Synchronization:

Communication between processes takes place through calls to send() and receive() primitives.
There are different design options for implementing each primitive.
Message passing may be either blocking or nonblocking - also known as synchronous and asynchronous.

- Blocking send: The sending process is blocked until the message is received by the receiving process or by the mailbox.

- Nonblocking send: The sending process sends the message and resumes operation.
- Blocking receive: The receiver blocks until a message is available.
- Nonblocking receive: The receiver retrieves either a valid message or a null.

Different combinations of send( ) and receive( ) are possible. when both send( ) and receive( ) are blocking, we have a rendezvous between the sender and the receiver.

Buffering:

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways.

no buffering

→ Zero Capacity: The queue has a maximum length of zero. Thus, the link cannot have any messages waiting in it. In this case, the sender must block until the receipient receives the message.

automatic buffering

→ Bounded Capacity: The queue has finite length n; thus at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue, and the sender can continue execution without waiting.

Unbounded capacity: The queue's length is infinite thus any number of messages can wait in it. The sender never blocks.

* Peterson's solution:

→ It is restricted to two Processes that alternate Execution between their critical sections and remainder sections.

The processes are P_0 and P_1

Peterson's solution requires the two Processes to share two data items:

```
int turn;
boolean flag[2];
```

⇒ if $\boxed{turn == i}$, then Process P_i is allowed to Execute in its Critical section.

⇒ The flag array is used to indicate if a process is ready to Enter its critical section.

Eg: if flag[i] is true, this value indicates that P_i is ready to Enter its Critical Section

To Enter the critical section, Process P_i first sets flag[i] to be true and then sets turn to the value j, asserting that if the other process which to Enter the Critical section, it can do so.

```
do
{
    flag[i] = TRUE;
    turn = j;
    while ( flag[j] && turn == j);

        Critical Section;

    flag[i] = FALSE;

        remainder Section.
} while(TRUE);
```

Fig: The Structure of Process P_i in Peterson's Solution.

⇒ To proove that this Solution is Correct.
we need to show that:

1. Mutual Exclusion is Preserved.
2. The Progress requirement is satisfied.
3. The bounded-waiting requirement is met.

To Prove Property 1,

P_i Enters its critical section only if either flag[j] == false or turn == i.

Any solution to the critical-section problem requires a simple tool — a lock.

Race Conditions are prevented by requiring that critical regions be protected by locks.

i.e., a Process must acquire a lock before Entering a critical section. it releases the lock when it Exits the critical section.

```
do {
        acquire lock
              critical section
        release lock
              remainder section
} while (TRUE);
```

Fig: Solution to the critical-section problem using locks.

we start by Presenting Some simple hardware instructions that are available on many Systems and Showing how they can be used Effectively in solving the critical-section problem. Hardware features can make any Programming task Easier and improve System Efficieny.

→ The critical-section problem could be solved simply in a uniprocessor environment if we could prevent interrupts from occuring while a shared variable was being modified.

Unfortunately, this solution is not feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, as the message is passed to all the processors. this message passing delays entry into each critical section, and system efficiency decreases. Also consider the effect on a system's clock if the clock is kept updated by interrupts.

→ Many modern computer systems therefore provide special hardware instructions that allow us either to 1) test and modify the content of a word or 2) to swap the contents of two words atomically — that is, as one uninterruptible unit. we can use these special instructions to solve the critical-section problem.

The TestAndSet( ) instruction can be defined as

```
boolean TestAndSet( boolean *target)
  {  boolean rv = *target;
     *target = TRUE;
     return rv;
  }.
```

→ The important characteristic of this instruction is that it is Executed atomically thus,

→ if two TestAndSet() instructions are Executed Simultaneously (Each on a different CPU), they will be Executed sequentially in some arbitrary order.

→ If the machine supports the TestAndSet() instruction, then we can implement mutual Exclusion by declaring a Boolean variable lock, initialised to false.

```
do{
      while(TestAndSet(&lock))
            ; // do nothing

      // Critical Section   lock = TRUE

   lock = FALSE;

   // remainder Section
} while (TRUE);
```

Fig: Mutual-Exclusion implementation with TestAndSet().

⇒ In contrast to the TestAndSet() instruction, The Swap() instruction, operates on the contents of two words, and is executed atomically.

⇒ If the machine supports the Swap() instruction, then mutual exclusion can be provided as follows.

⇒ A global Boolean variable lock is declared and is initialized to false. In addition, each process has a local Boolean variable key.

⇒ Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement.

global

```
Boolean lock = false
```

key = false

```
void swap(boolean *a, boolean *b)
{
    boolean temp = *a;
      *a = *b;
      *b = temp;
}
```

Fig: The definition of the Swap() instruction

```
do {
    ~~while (TestAndSet(&lock))~~
      ~~; // do nothing~~
      // critical section
    lock = FALSE;
      // remainder section
} while (TRUE);
```

```
key = TRUE
while(key == TRU
Swap(&lock, &k
```

Fig: Mutual-Exclusion implementation with [illegible]

⇒ We present another algorithm using the TestAndSet( ) instruction that satisfies all the critical-section requirements.

The common data structures are:

```
boolean  waiting[n];
boolean  lock;
```

these data structures are initialized to false.

⇒ To prove that the mutual exclusion requirement is met, that process P_i can enter its critical section only if either :

```
waiting[i] == false  or  key == false
```

⇒ The value of key can become false only if the TestAndSet( ) is executed.

⇒ The first process to execute the TestAndSet( ) will find key == false, all others must wait.

⇒ The variable waiting[i] can become false only if another process leaves its critical section. Only one waiting[i] is set to false, maintaining the mutual-exclusion requirement.

→ ~~[illegible] and power - [illegible]:~~

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while( waiting[i] && key)
        key = TestAndSet( &lock);
    waiting[i] = FALSE;

    // Critical section.

    j = ( i + 1) % n;
    while(( j != i) && ! waiting[j])
        j = (j + 1)% n;
    if (j == i)
        lock = FALSE;
    else
        waiting[j]= FALSE;

    // remainder section
} while ( TRUE);
```

Fig: Bounded-waiting mutual Exclusion with TestAndSet

→ To prove that the progress requirement is met, we note that the arguments presented for mutual Exclusion also apply here, since a process Exiting the critical section Either sets lock ~~t~~o false or sets waiting[j] to false. Both allow a process that is waiting to Enter its critical section to proceed.

⇒ To prove that the bounded-waiting requirement is met, we note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i+1, i+2, ..., n-1, 0, ... i-1).

⇒ It designates the first process in this ordering that is in the entry section (waiting[j] == true) as the next one to enter the critical section.

⇒ Any process waiting to enter its critical section will thus do so within n-1 turns.

* Semaphores:

The hardware-based solutions to the critical-section problem are complicated for application programmers to use.

To overcome this difficulty, we can use a synchronization tool called a semaphore.

A Semaphore S is an integer variable that, is accessed only through two standard atomic operations: wait( ) and signal( ).

The wait( ) operation was originally termed P (from the Dutch "Proberen, "to test").

Signal( ) was originally called V (from Verhogen, "to increment").

The definition of wait( ) is as follows:

```
wait ( S ) {
        while ( S<=0)
            ; // no-op

        S--;
  }
```

The definition of signal( ) is as follows:

```
Signal( S) {
      S++;
 }
```

All modifications to the integer value of the semaphore in the wait( ) and signal( ) operations must be executed indivisibly. (atomic)

⇒ that is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

→ In addition, in the case of wait(S), the testing of the integer value of S ($S \leq 0$), as well as its possible modifications (S--), must be executed without interruption.

Usage:

Two types of Semaphores: The value of a

① Counting semaphore ⇒ can range over an Unrestricted domain.

② Binary semaphore ⇒ can range only between 0 and 1.

Binary semaphores:

On some systems, binary semaphores are known as mutex locks. as they are, locks that provide. mutual Exclusion.

⇒ we can use binary semaphores to deal with the critical-section problem for multiple processes.

The n processes share a semaphore, mutex, initialized to 1. Each process. P_i is organised as:

```
do
  {   wait(mutex);
           // critical section
        signal(mutex);
        // remainder section
   } while( TRUE);
```

Fig: Mutual-Exclusion implementation with semaphores.

Counting semaphores:

⇒ Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available.

⇒ Each process that wish to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). 3 Print

count = 3 ⇒

⇒ when a process releases a resource, it performs a signal() operation (incrementing the count).

⇒ when the count for the semaphore goes to 0, all resources are being used.

⇒ After that, processes that wish to use a resource will block until the count becomes greater than 0.

⇒ The main disadvantage of the semaphore definition given here is that it requires busy waiting.

⇒ while a process is in its critical section any other process that tries to enter its C.S must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes.

⇒ Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a Spinlock because the process "spins" while waiting for the lock.

⇒ To overcome the need for busy waiting, we can modify the definition of the wait() and signal() semaphore operations.

→ when a process executes the wait() operation and finds that the semaphore value is not +ve, it must wait. However, rather engaging in busy waiting the process can block itself.

⇒ The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

⇒ A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation.

⇒The Process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state.

⇒The process is then placed in the ready queue.

To implement semaphores under this definition, we define a semaphore as a "C" struct:

```
typedef    struct {
              int  value;
              struct  Process  *list;
        } semaphore;
```

⇒Each semaphore has an integer value and a list of Processes list.

⇒ when a process must wait on a semaphore, it is added to the list of processes.

⇒A signal() operation removes one process from the list of waiting processes and awakens that process.

The wait() semaphore operation can now be defined as:

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

The signal() semaphore operation can now be defined as:

```
Signal (Semaphore *S)
{
        S -> value++;
    if ( S-> value <=0)
    {
        remove a process P from
                        S -> list;
        wakeup(P);
    }
}.
```

⇒ The block() operation suspends the process that invokes it.

⇒ The wakeup(P) operation resumes the execution of a blocked process P.

⇒ These two operations are provided by the operating system as basic system calls.

The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs. So one way to add and remove processes from the list so as to ensure bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue.

It is critical that semaphores be executed atomically.

⇒ We must guarantee that no two processes can execute wait() and signal() operations on the same semaphore at the same time. This is a critical-section problem.

→ In a single-processor environment, we can solve it by simply preventing interrupts during the time the wait() and signal() operations are executing. This scheme works in a single-processor environment because, once interrupts are prevented, instructions from different processes cannot be interleaved. Only the currently running process executes until interrupts are reenabled and the scheduler can regain control.

⇒ In a multiprocessor environment, interrupts must be disabled on every processor, otherwise, instructions from different processes (running on different processors) may be interleaved in some arbitary way.

⇒ Disabling interrupts on every processor can be a difficult task and furthermore can seriously decrease performance. Therefore, SMP systems must provide alternative locking techniques — such as spinlocks — to ensure that wait() and signal() are performed atomically.

Deadlocks and Starvation:

The implementation of a Semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an Event that Can be caused only by one of the waiting processes. The Event in question is the Execution of a signal() operation. when such a state is reached, these processes are said to be deadlocked.

Eg: Consider a System Consisting of two processes, P_0 and P_1, Each accessing two Semaphores, S and Q, set to the value 1:

```
S=1        P0          |        P1      Q=1
      ---------------  |  ---------------
      wait(S);         |  wait(Q);
      wait(Q);   Q=0   |  wait(S);        S=0
         .             |     .
         .             |     .
         .             |     .
      signal(S);       |  signal(Q);
      signal(Q);       |  signal(S);
```

Priority inversion:

A Scheduling challenge arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process — or a chain of lower-priority processes.

Since kernel data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource.

The situation becomes more complicated if the lower-priority process is preempted infavor of another process with a higher priority.

Eg: we have three processes, L, M, and H, whose priorities follow the order $L<M<H$.

Assume that process H requires resource R, which is currently being accessed by process L. Ordinarily, process H would wait for L to finish using resource R. However now suppose that process M becomes runnable, thereby preempting process L.

Indirectly, a process with a lower priority — Process M — has affected how long process H must wait for L to relinquish resource R.

This problem is known as priority inversion. It occurs only in system with more than two priorities. This problem can be solved by implementing a priority-inheritance protocol.

According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources that in question (required)

→ when they are finished, their priorities revert to their original values.

⇒ In the above example, a priority-inheritance protocol would allow process L to temporarily inherit the priority of process H, thereby preventing process M from preempting its execution. When process L had finished using resource R, it would relinquish its inherited priority from H and assume its original priority. Because resource R would now be available, to process H — not M — H would run next.

*Classic Problems of Synchronization:

(1) The Bounded-Buffer Problem:

we assume that the pool consists of 'n' buffers, Each capable of holding One Item.

The mutex semaphore provides mutual Exclusion for accesses to the buffer pool and is initialised to the value 1. $\boxed{\text{mutex} = 1}$

The Empty and full semaphores count the number of Empty and full buffers.

The semaphore Empty is initialised to the value n. $\boxed{\text{Empty} = n}$. The smaphore full is initialized to the value 0: $\boxed{\text{full} = 0}$

```
do
{ ....
  //Produce an item in nextp
  -----
  wait(Empty);
  wait(mutex)
  -----
  //add nexp to buffer
  -----
  signal(mutex);
  Signal(full);
}while (TRUE);
```

fig: The structure of the Producer Process.

```
do{
    wait(full);
    wait(mutex);
    -------
  //remove an item from
  buffer to nextc
    -------
    Signal(mutex);
    Signal(Empty);
  //consume the item
    in nextc
}while(TRUE);
```

fig: The structure of the Consumer Process.

② The Readers-writers problem:

Suppose that a database is to be shared among several concurrent processes.

Some of these processes may want only to read the database, (reffered as readers)

whereas others may want to update (that is, to read and write) the database. (reffered as writers.).

If two readers access the shared data simultaneously, no harmful Effects will result.

However, if a writer and some other process (Either a reader or a writer) access the database simultaneously, confusion may Ensure.

To Ensure that these difficulties do not arise, we require that the writers have Exclusive access to the shared database while writing to the database. This synchronization problem is refferred to as the readers - writers problem.

The readers-writers problem has several variations:

(1.) No reader be kept waiting unless a writer has already obtained permission to use the shared object. (i.e) no reader should wait for other readers to finish simply because a writer is waiting.

② once a writer is ready, that writer performs its write as soon as possible.

(i.e) if a writer is waiting to access the object, no new readers may start reading.

— A solution to Either problem may result in starvation.

i.e. In the first case, writers may starve,

In the second case, readers may starve.

① A solution to the first readers-writers problem:

The reader processes share the following data structures:

```
semaphore   mutex , wrt ;
int         readcount ;
```

→ The semaphores mutex and wrt are initialised to 1. readcount is initialised to 0.

→ The semaphore wrt is common to both reader and writer processes.

→ The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated.

→ The readcount variable keeps track of how many processes are currently reading the object.

→ The semaphore wrt functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section.

* It is not used by readers who enter or exit while other readers are in their critical sections.

```
do
{
    wait(wrt);
    - - - - - - -
    // writing is performed
    - - - - - - -
    signal(wrt);
} while(TRUE);
```

Fig: The structure of a writer process.

```
do
{
    wait(mutex);
    readcount++;
    if(readcount == 1)
        wait(wrt);
    signal(mutex);
    - - - - - -
    // reading is performed
    - - - - -
    wait(mutex);
    readcount--;
    if(readcount == 0)
        signal(wrt);
    signal(mutex);
} while(TRUE);
```

fig: The structure of a reader process.

⇒ If a writer is in the critical section and 'n' readers are waiting, then one reader is queued on wrt, and n-1 readers are queued on mutex.

⇒ When a writer Executes Signal(wrt), we may resume the Execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

The readers-writers problem and its solutions have been generalized to provide reader-writer locks, one Same Systems.

→ Acquiring a reader-writer lock requires Specifying the mode of the lock: Either read or write access.

→ When a process whishes only to read shared data, It requests the reader-write lock in read mode.

→ a process whishing to modify the shared data must request the lock in write mode.

→ Multiple processes are permitted to concurrently acquire a reader-writer lock in read mode, but only one process may acquire the lock for writing, as Exclusive access is required for writers.

Reader-writer locks are most useful in the following situations:

→ In applications where it is Easy to identify which processes only read shared data and which processes only write shared data.

⇒ In applications that have more readers than writers.

The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader-writer lock.

③ The Dining-philosophers problem:

Consider five philosophers who spend their lives thinking and Eating. The philosophers share a circular table surrounded by five chairs, Each belonging to one philosopher.

In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.

when a philosopher thinks, she does not interact with her colleagues.

From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her.

⇒ A philosopher may pick up only one chopstick at a time.

→ She cannot pick up a chopstick that is already in the hand of a neighbor.

when a hungry philosopher has both her chopsticks at the same time, she Eats without releasing her chopsticks. when she is finished Eating, she puts down both of her chopsticks and starts thinking again.

One simple solution is to represent Each chopstick with a semaphore. A philosopher tries to grab a chopstick by Executing a wait() operation on that semaphore. she releases her chopsticks by Executing the signal() operation on the appropriate semaphores.

The shared data are:

```
Semaphore chopstick[5];
```

where all the elements of chopstick are initialized to 1.

```
do{
    wait(chopstick[i]);
    wait(chopstick[(i+1)%5]);
    ......
    //Eat
    .....
    signal(chopstick[i]);
    signal(chopstick[(i+1)%5]);
    .....
    //think
    .....
}while(TRUE);
```

fig: The Stoucture of philosopher i.

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. when each philosopher tries to grab her right chopstick, she will be delayed forever.

FCFS:

Consider the performance of FCFS Scheduling in a dynamic situation:

Example:

one CPU -Bound process and

many I/O Bound processes

1) CPU -Bound process will get and hold the CPU.

2) I/O Bound processes will finish their I/O and

move into the ready queue., waiting for CPU.

3) I/O devices are idle.

4) CPU -Bound process finishes its CPU burst and moves to an I/O device.

5) All the I/O Bound processes will finish their short CPU burst and move back to the I/O queues.

6) at this point CPU sits idle.

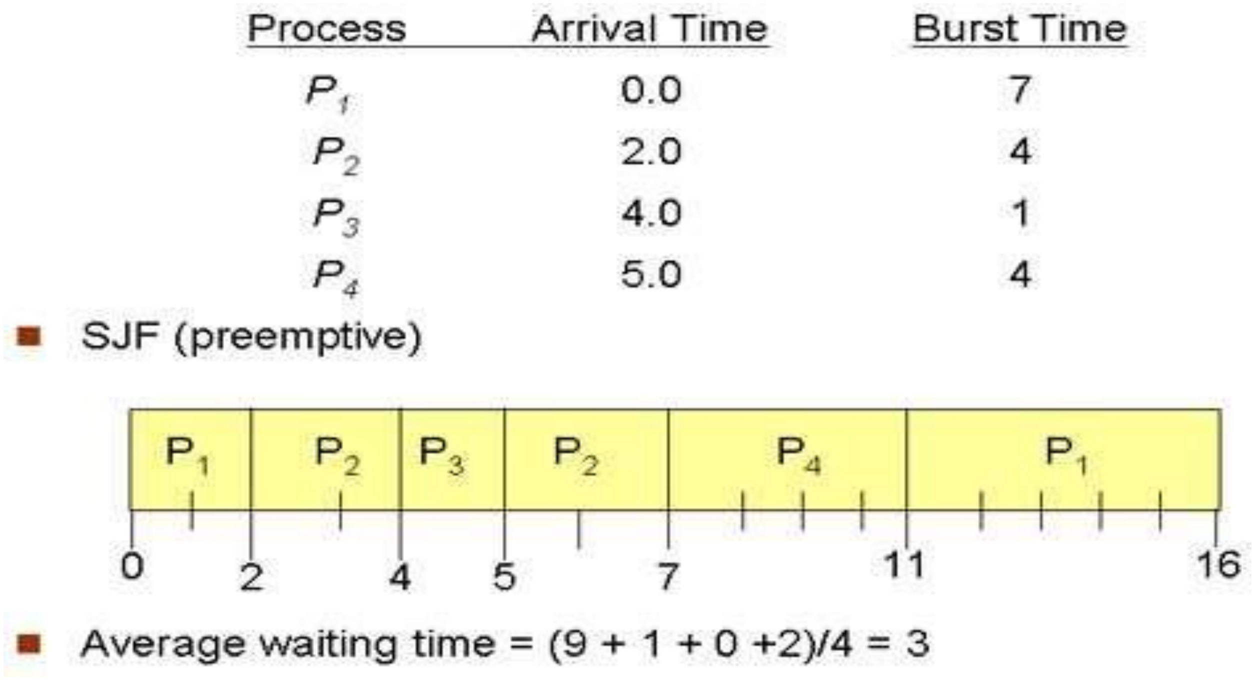
7) CPU -Bound process will then move back to the ready queue and be allocated the CPU.

8) again, All the I/O Bound processes end up waiting in the ready queue until the CPU -Bound process is done.

Convoy effect : as all the other processes wait for the one big process to get off the CPU.

Preemptive SJF scheduling : is sometimes called shortest-remaining- time-first scheduling (SRTF)

Example:1



Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)

- Average waiting time = (9 + 1 + 0 +2)/4 = 3

p1 remain time 7-2=5

p2 =4 so p1>p2 allow p2 to execute

at 4th unit time p3=1 has arrived

p2 remain time 4-2=2

compare p2 >p3 i.e p1=5 , p2=2 , p3=1

so allow p3 for execution then

at 5th unit time p3 completed and p4 =4 has arrived in ready queue we have

p1=5, p2=2, p4=4 --------> so p2 is small then p2 gets the chance to execute then p4 and p1.

waiting time= service time - arrival time

waiting time of p1 = (11-2)-0 =9

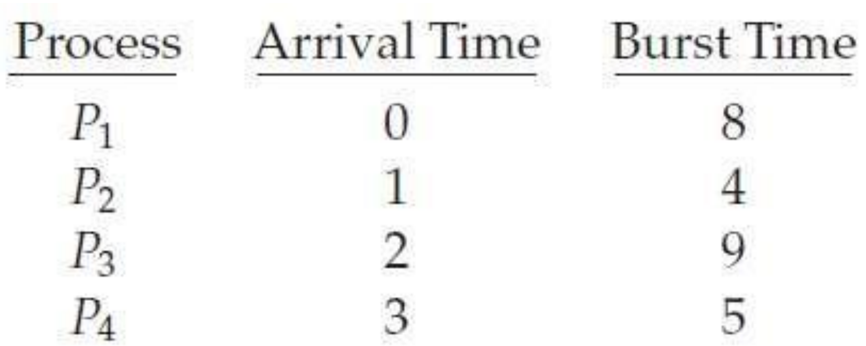
waiting time of p2 = (5-2) -2= 1

waiting time of p3= (4-4)=0

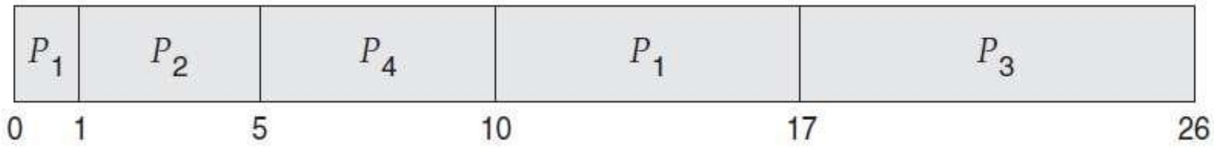
waiting time of p4= 7-5=2

Average waiting time = (9+1+0+2)/4=3ms

Example:2



Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



p1=8-1=7 remaining time

p2 = 4   p1>p2 ..

at 2 unit time p3 =9

p1=7, p3=9, remaining time p2=3

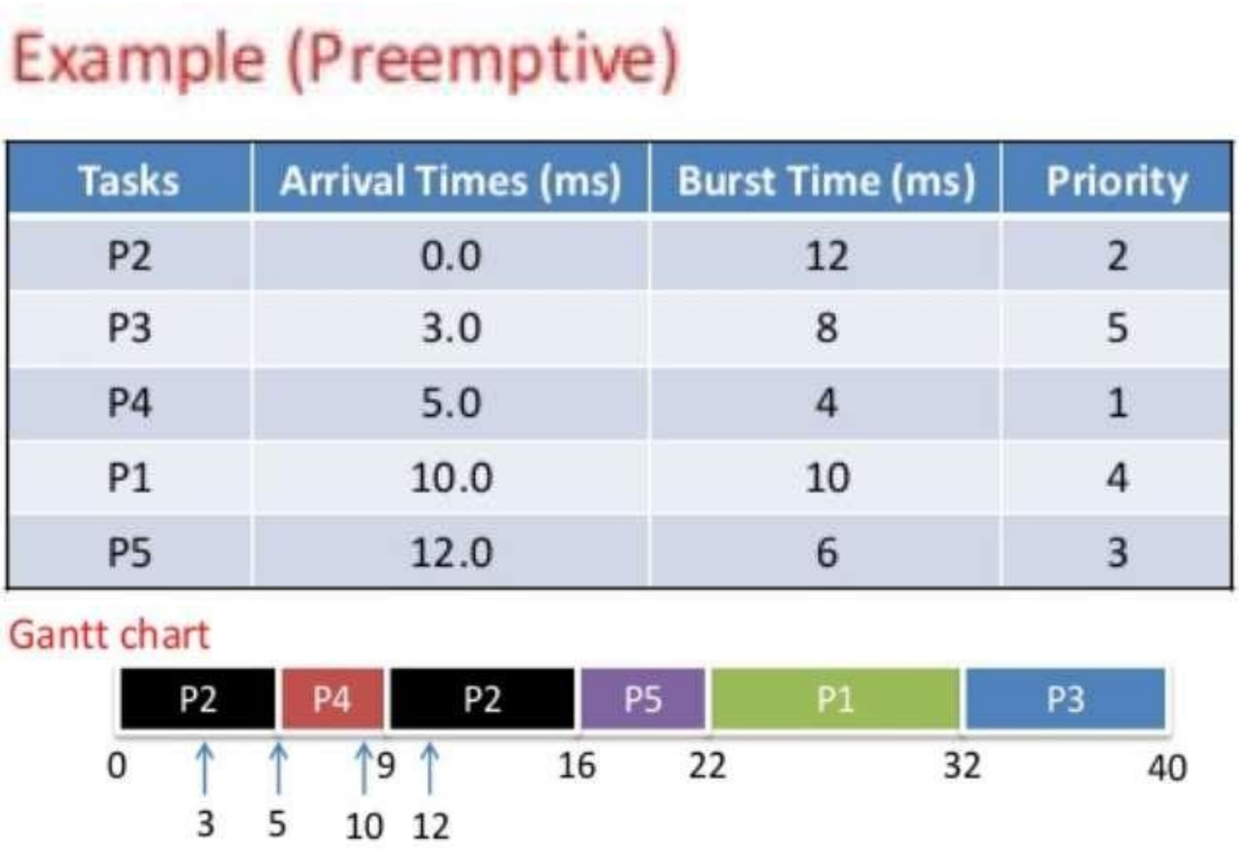
at 3 unit time p4=5 has arrived

p1=7, p3=9, p4=5 ,, remaining time p2=2

p4=5 smaller burst time

p1=7 , p3=9 in ready queue

p1 then p3



Example (Preemptive)

Tasks	Arrival Times (ms)	Burst Time (ms)	Priority
P2	0.0	12	2
P3	3.0	8	5
P4	5.0	4	1
P1	10.0	10	4
P5	12.0	6	3

Gantt chart

at 3 unit time p3 pri #5 > p2 pri#2 (less is high priority)   p3 in ready queue

at 5 unit time p4 pri# 1

p2 ---2

p3--- 5

p4----1

p4 is allocated with CPU=== completed

9 unit time ---- p2 allocated with CPU

at 10 unit time p1 with pri#4

p2 is running pri#2

p3 in ready queue with pri#5

p1 ------------pri#4

p2 conti....

at 12 unit p5 with Pri#3

p2 is running, p3 and p1 ....new p5

p2 to continu...

p3 pri 5, p1 pri-4, p5 pri#3

Consider set of n tasks with known runtimes R1,R2, Rn to be run uniprocessor machine . which of the following scheduling algorithm will result in the maximum throughput?

Ans: SJF

----

starvation problem:

high pri# 0   ---- low pri# 127

suppose a process low pri#127

Aging technique

increase pri by 1 for every 15 minutes

come to pri#0 not more than 32 hours

Round Robin:

Time quantum/time slice is defined.

2 cases:

1)TQ > CPU burst time

tq=4ms

cpu burst =3ms=== process voluntarily releases CPU and scheduler selects the next process

2) tq < cpu burst time

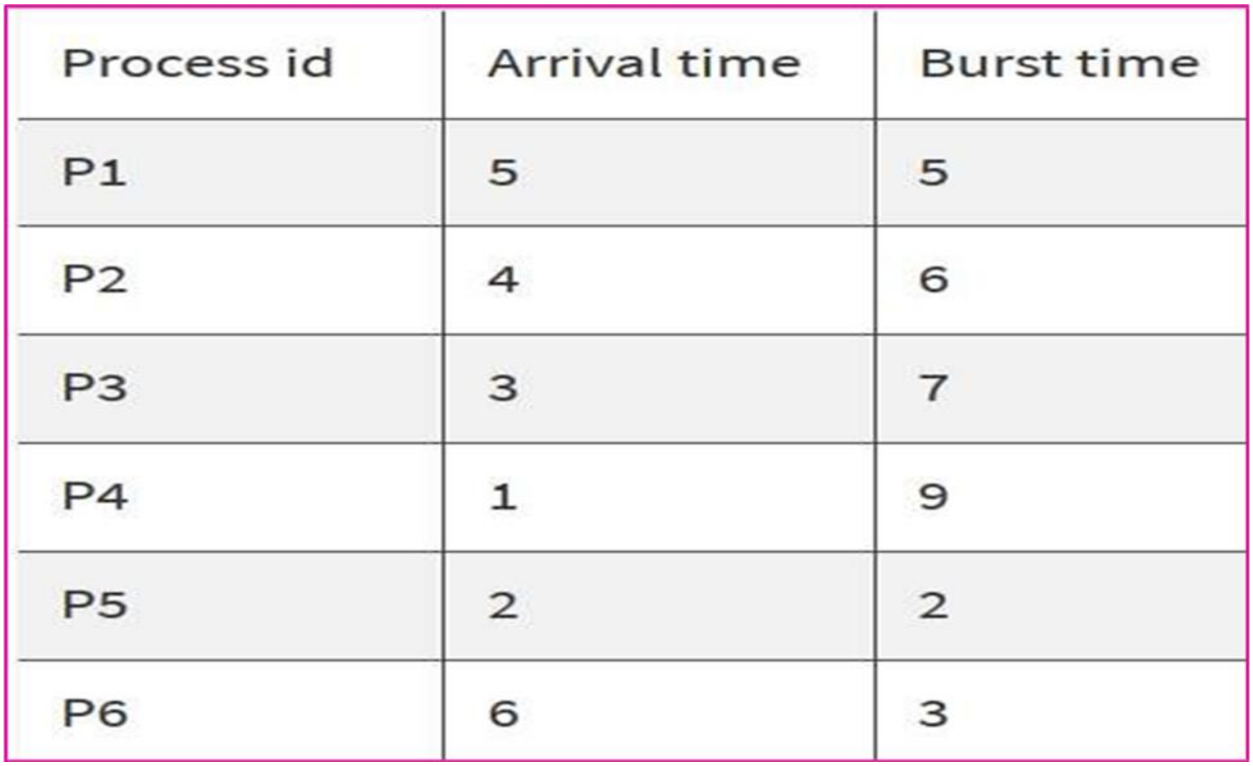
once the timer expires , process is preempted and added at the back end of the ready queue.

If the time quantum size is 2 units of an there is only one job of 14 units time unit in ready queue, Round Robin scheduling algorithm will cause________ context switches.

Ans: 6

Example:

Consider the following processes with arrival time and burst time. Calculate average turnaround time, average waiting time and average response time using round robin with time quantum 3?



Process id	Arrival time	Burst time
P1	5	5
P2	4	6
P3	3	7
P4	1	9
P5	2	2
P6	6	3

1) At 0(Zero) unit time --- CPU is idle.

2) At 1 unit time --- P4 =9 BT has arrived...and Schedule on CPU for 1 time quantum (3ms)

3) during p4 execution time --- p5,p3 and p2 processes are arrived and placed in ready queue.

p4 is preempted after 1tq ,timer expires. moved to back end of ready queue. i.e.

P5=2	P3=7	P2=6	P4=6

4) p5 is scheduled for next... 1 tq but it required only 2 ms , swap out then p3 is scheduled 1tq

ready queue: p2, p4,p1,p6 --- p3=4ms

turnaround time= completion time- arrival time

(or)

Turnaround time= Burst time + waiting time

Waiting time= service time - arrival time

Response time =first service time -arrival time

CPU Scheduling Algorithms:

1) FCFS (Non-Preemptive)

2) SJF   (Preemptive (or) Non-Preemptive)

3) Priority (Preemptive (or) Non-Preemptive)

4) Round Robin (purely preemptive)

Another Class of Scheduling Algorithms are:

1) Multilevel Queue Scheduling

2) Multilevel Feedback-Queue Scheduling

Multiple -Processor Scheduling:

Asymmetric multiprocessing

Symmetric multiprocessing(SMP)

Processor Affinity

Load Balancing