IE2032 – Secure Operating System Assignment

An Evaluation of Adaptive Partitioning of Real-Time Workload on Linux

Group ID- sos_16

Group members' details.

Name	IT Numbers	Work on section
T.N. Uduwanage	IT22293312	• Abstract • Installation Process • Deadlock management • CPU Scheduling • Strengths and weaknesses of the operating system
Perera A.P. j	IT22280992	• Abstract • Secondary disk scheduling management • Standard support • Protection and security • Strengths and weaknesses of the operating system
D. N. R. Jayawardana	IT22104908	• Introduction • Process Control • Memory Management • Mass Storage System • Strengths and weaknesses of the operating system
VIDARA S V	IT22299048	• Introduction • System Requirement • User Interface • Synchronization • Strengths and weaknesses of the operating system

Table of Content

An Evaluation of Adaptive Partitioning of Real-Time Workload on Linux

T.N. Uduwanage
IT22293312
it22293312@my.sliit.lk

Perera A.P. j
IT22280992
it22280992@my.sliit.lk

D. N. R. Jayawardana
IT22104908
it22104908@my.sliit.lk

VIDARA S V
IT22299048
it22299048@my.sliit.lk

***Abstract*- This paper investigates the idea of adaptive partitioning for Linux operating system real-time workloads. It looks at the specialized evaluation and adaptation of certain operating system features, including memory management, deadlock handling, and process control, for real-time operations. The paper explores the development of these features, comparing and contrasting program dependencies and system requirements (CPU, RAM, storage) from the past with those from the present. Analysis is also done on user interface interactions and installation procedures. This paper also covers synchronization, mass storage systems, protection, scheduling, and their benefits, as well as CPU scheduling and the various ways it might be compromised. The article ends with a discussion of standard support for real-time systems and available secondary disc scheduling strategies. The goal of this integrated knowledge is to enhance the comprehension and use of adaptive partitioning for Linux workload management in real time.**

***Index term* – Real-time Operating system, Process control, Linux Kernel**

I. INTRODUCTION

Linux is an open-source operating system. Therefore, everyone can modify, improve, and distribute the code. Linux was created by Linus Torvalds in 1991. If anyone having program knowledge, can customize this operating system. The Linux runs on desktops, notebooks, tablets, and smartphones. This is one of the most flexible and powerful, operating systems. However it is not completely safe, and it is less vulnerable than other operating systems.

Especially in this operating system, does not require any safeguard program. Mainly this operating system has 3 components. There is user mode, system call, and kernel mode. This language was specially developed for creating the UNIX system. Using this new technique, it was much vendors were quick to adapt since they could sell ten times more software almost effortlessly. Weird new situations came into existence: imagine for instance computers from different vendors communicating in the same network, or users working on different systems without the need for extra education to use another computer.

UNIX did a great deal to help users become compatible with different systems. Throughout the next couple of decades, the development of UNIX continued. More things became possible to do and more hardware and software vendors added support for UNIX to their products. UNIX was initially found only in very large environments with mainframes and minicomputers (note that a PC is a "micro" computer). You had to work at a university, for the government, or for large financial corporations to get your hands on a UNIX system. But smaller computers were being developed, and by the end of the 80's, many people had home computers. By that time, there were several versions of UNIX available for the PC architecture, but none of them were truly free, and more importantly: they were all slow, so most people ran MS-DOS or Windows 3.1 on their home PCs. 1.1.2. Linus and Linux.

II. SYSTEM REQUIREMENTS

System requirements can divided mainly into two parts. They are hardware requirements and software requirements. According to hardware

requirements, it needs to Minimum RAM, central processing unit and hard disk (HDD) requirement for the desktop edition of the Debian-based operating system and the RedHat based operating system is listed here. To run the Linux operating system smoothly, a minimum 1-2GB RAM is required. The actual minimum memory requirements for the Linux operating system only (without any additional software), are less than these numbers.

Linux can give optimum performance on Intel Pentium or high processor. Linux also supports non-intel processors such as the Cyrix 6X86 and AMD K5 and K6. A4 to 5GB hard disk is enough to store the system and other important files of a Linux distribution. However, to store user data it's recommended to have at least 25GB hard disk size.

Here are some details of hardware requirements which need to when working with Linux operating system.

System Requirement/ DX NetOps Spectrum Component	OneClick Client	OneCLICK SERVER	SpectroSERVER
Operating System (OS)	Red Hat Enterprise Linux (RHEL) 8.x (64-bit) RHEL 7.x (64-bit) X-based desktop environment (such as KDE or GNOME)	RHEL 8.x (64-bit) RHEL 7.x (64-bit)	RHEL 8.x (64-bit) RHEL 7.x (64-bit)
Memory (RAM) [1]	Dependent on the configuration and number of managed devices	Dependent on the configuration and number of managed devices	Dependent on the configuration and number of managed devices
Processor	Dependent on the configuration and number of managed devices	Dependent on the configuration and number of managed devices	Dependent on the configuration and number of managed devices
Disk Space Dependent on the configuration and number of managed devices	Dependent on the configuration and number of managed devices	HDD: 100 GB (Minimum 50 GB)	HDD: 100 GB (Minimum 50 GB)
Graphical User Interface	X11 system that the JRE supports	Motif	Motif

III. INSTALLATION PROCESS

- INTRODUCTION

The Linux operating system installation procedure consists of multiple important phases that are all necessary for the system to be successfully configured. The purpose of this study is to describe the essential steps of installing Linux on a computer, emphasizing the standard process that applies to all Linux distributions.

- PRE-INSTALLATION PREPARATION

Before starting the installation, there are a few things you must do. These include getting the installation media a USB drive, DVD, or ISO file—making sure your hardware is compatible with the Linux distribution you have selected, and backing up your important data.

- BOOTING INTO THE INSTALLATION ENVIRONMENT

The machine must be booted from the installation media as soon as it is ready. By doing this, you can force the installation media to boot from the hard drive or SSD instead of other storage devices by going into the BIOS or UEFI firmware settings.

- SELECTING INSTALLATION OPTIONS

Users are asked to choose from a list of settings to personalize the installation process when they boot into the installation environment. Choosing the preferred language and keyboard layout for the installation wizard to use is usually part of this process.

- DISK PARTITIONING

During the installation process, disc partitioning is a crucial stage when users specify the Linux system's storage configuration. Users have the choice of manually configuring partitions and selecting mount points for partitions such as root (/), swap, and /home, or they can choose automatic partitioning.

- INSTALLATION

After the disc partitioning process is finished, the installer copies the required files from the installation media to the target drive's partitions. Installing the Linux kernel, system libraries, and certain software packages in accordance with the user's choices are included in this.

- CONFIGURING SYSTEM SETTINGS

After the installation of the main components, users are asked to set up the necessary system configurations. In order to do this, you must set the root password, create a user account with the necessary permissions, specify the hostname, and configure the network settings so that you can connect to the internet.

- BOOT LOADER INSTALLATION

Installing the boot loader—such as GRUB or LILO—that makes it easier for the Linux operating system to boot up is another step in the installation process. To ensure correct boot management, users must install the boot loader to the EFI or Master Boot Record (MBR) partition.

- FINALIZING THE INSTALLATION

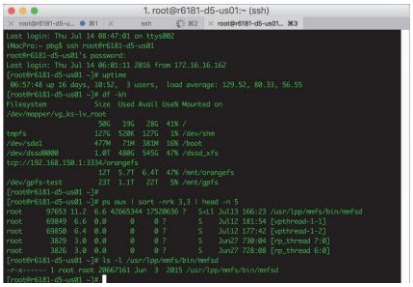
Before starting the installation process, users check the installation summary to make sure all the configuration choices are set. To complete the installation, they take out the installation disc and restart the computer.

- POST-INSTALLATION STEPS

Users utilize the login credentials they created during installation to log in after rebooting. To further personalize the Linux environment to their tastes, they install extra applications as needed, update system packages, and adjust system settings.

IV. USER INTERFACE

There are several ways for users to interface with the operating system. But mainly we focus on 2 types. They are the command line interface, and graphical user interface. Another way is the touch screen interface. Those allow users to communicate with computers or communication devices. In the command line interface (CLI), the user interacts with operating systems by using commands. If the user types commands, those commands are caught by the command interpreter. These interpreters are known as shells. The main duty of the command interpreter is to get and execute the next instruction which provide by user. When user types commands represented by short keywords or press the special keys on the keyboard to enter data and instructions.



HERE ARE SOME ADVANTAGES OF CLI.

- CLI can be a lot faster and efficient than any other type of interface. It can also handle repetitive tasks easily.
- A CLI requires less memory to use in comparison to other interfaces. It also does not use as much CPU processing time as other interfaces.
- A CLI doesn't require Windows and a low-resolution monitor can be used. Meaning, it needs fewer resources, yet is highly precise.

DISADVANTAGES OF CLI

- If user is new user or have never used a CLI, this approach can be confusing.
- Accuracy is of the utmost importance. If there is a spelling error, a command will fail.

Graphical user interface is the other UI. This one interacts with user through a user-friendly graphical way. This one is more different rather than command line interface. Because if user wants to interact with operating system, it uses mouse to access windows, icons, and menus. This GUI first appeared due in the early 1970s at XEROX PARC research faculty.

ADVANTAGES OF GUI

- It requires just a click on the simple picture or image in order to use its functionalities.
- It is very easy to use by novice as it is user friendly.
- Programmer or user need not have to understand working of the computer system.
- It looks very attractive and multi-colored.
- It is much better than command driven interface which has many drawbacks.
- User can switch quickly between tasks on the GUI interface.
- Full screen interaction is also possible with quick and wholesome access to anywhere on the screen.

DISADVANTAGES OF GUI

- It uses more computer memory as the aim is to make it for user friendly and not resource optimized. As a result, it can be slow on older machines.
- GUI becomes more complex if user needs to communicate with the computer directly.
- Certain tasks may take long due to many menus to select the desired choice.
- Hidden commands need to be searched using Help file.
- GUI based applications require more RAM to run.
- It uses more processing power compare to other interface types.

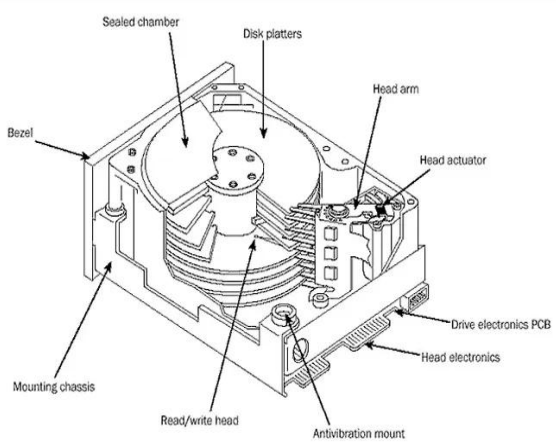
Touch screen interface is the same for GUI. But this one has not mouse. Touching screen

interface use to interact with user by making on the touch screen. That can press and swiping fingers across the screen. This method normally we can see smartphones, tablets, and laptops. Earlier smartphones include a physical keyboard. But now it changed. This Linux operating system has CLI and GUI interface.

V. SECONDARY DISK SCHEDULING MANAGEMENT

Linux secondary disk scheduling management is the complex coordination of disk access to maximize system efficiency, especially in high-throughput and real-time applications. The order in which the disk responds to read and write requests is determined by several scheduling algorithms, including CFQ, deadline, and snoop. Partitioned scheduling techniques are frequently used in real-time systems when a single missed deadline might have disastrous consequences. These techniques use the best outcomes from the literature on single-processor scheduling to assign jobs to certain CPU cores. However, allocating work in advance among CPUs is a computationally challenging problem that is usually solved with integer linear programming methods. Global scheduling techniques, on the other hand, dynamically divide work among cores, making adoption simpler but limiting the capacity to fully utilize system resources. Global scheduling techniques such as Earliest Deadline First (EDF) are frequently used in soft real-time systems, which are tolerant of rare deadline misses. For real-time multimedia workloads, Linux, for instance, has the SCHED_DEADLINE policy, a global EDF implementation that is becoming more and more popular.

Through experimental assessments, researchers have investigated a variety of scheduling algorithms and methodologies, frequently utilizing Linux as the evaluation platform. For example, a lot of research has been done on the efficacy of adaptive partitioning algorithms such as apEDF and a2pEDF. The objective of these methods is to optimize system performance by distributing workloads evenly among cores and reducing migrations. Synthetic task sets with different properties, such as overall utilization, and number of tasks per core, are created for experimental assessments. Researchers have seen how various scheduling algorithms behave in terms of migrations, deadline misses, and overall system performance via rigorous testing.

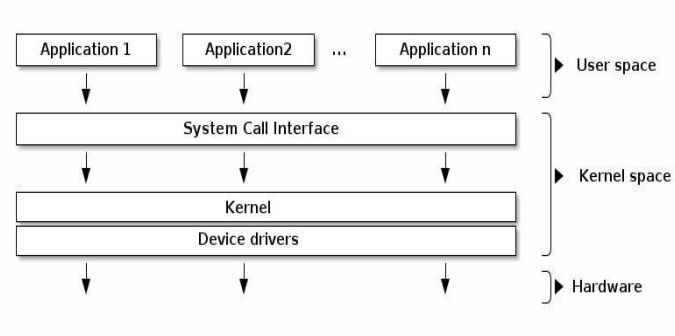


VI. STANDARD SUPPORT

With a variety of kernel-implemented scheduling principles and procedures, Linux offers standard support for real-time scheduling. The following are some essential elements of the Linux real-time scheduling standard support.

1. Real-time Scheduling Policies:
   - SCHED_FIFO: First out scheduling policy
   - SCHED_RR: Round Robin scheduling policy
   - SCHED_DEADLINE: Deadline-based scheduling policy.
2. Kernel Configuration Options:
   - Real-time scheduling capability can be configured into the Linux kernel during compilation. This enables developers to set particular scheduling policies and functionalities to be enabled or disabled based on their needs.
3. System Calls:
   - Linux offers system functions for adjusting a process's scheduling policy and parameters, such as sched_setscheduler() and sched_getscheduler(). Applications can ask the kernel for real-time scheduling behavior through these system functions.
4. User-space Libraries and Tools:
   - User-space libraries and tools to help developers manage and analyze real-time jobs are frequently included in Linux distributions. These libraries might include real-time programming APIs as well as capabilities for analysis and performance tracking.
5. Documentation and community support:
   - Linux distributions usually offer community resources and documentation to assist developers in comprehending and making use of real-time scheduling features. This comprises official records, discussion boards, email groups, and

community-driven initiatives with an emphasis on real-time development.



VII. PROCESS CONTROL

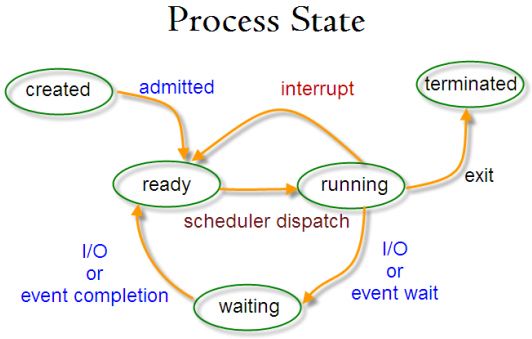
Process is a program in execution. Process execution must progress a sequential execution.

Program is passive entity stored on disk (executable file), process is active. Program becomes process when an executable file is loaded into memory.

One program can be several processes.

PROCESS STATE

- New : The process is being created
- Running : Instructions are being executed
- Waiting : The process is waiting for some event to occur
- Ready : The process is waiting to be assigned to a processor
- Terminated : The process has finished execution



PROCESS CONTROL BLOCK

- Process State: The state of the process is stored in the PCB which helps to manage the processes and schedule them. There are different states for a process which are "running," "waiting," "ready," or "terminated."
- Process ID: The OS assigns a unique identifier to every process as soon as it is created which is known as Process ID, this helps to distinguish between processes.
- Program Counter: While running processes when the context switch occurs the last instruction to be executed is stored in the program counter which helps in resuming the execution of the process from where it left off.
- CPU Registers: The CPU registers of the process helps to restore the state of the process so the PCB stores a copy of them.
- Memory Information: The information like the base address or total memory allocated to a process is stored in PCB which helps in efficient memory allocation to the processes.

Process Scheduling Information: The priority of the processes or the algorithm of scheduling is stored in the PCB to help in making scheduling decisions of the OS.

Accounting Information: The information such as CPU time, memory usage, etc helps the OS to monitor the performance of the process.

There can be more information about a process that can be stored in the PCB.

There are two conventional ways used for creating a new process in Linux:

- Using The System() Function – this method is relatively simple, however, it's inefficient and has significantly certain security risks.
- Using fork() and exec() Function – this technique is a little advanced but offers greater flexibility, speed, together with security.

How Does Linux Identify Processes?

Because Linux is a multi-user system, meaning different users can be running various programs on the system, each running instance of a program must be identified uniquely by the kernel.

And a program is identified by its process ID (PID) as well as it's parent processes ID (PPID), therefore processes can further be categorized into:

- Parent processes – these are processes that create other processes during run-time.
- Child processes – these processes are created by other processes during run-time.

OPERATIONS THAT ARE CARRIED OUT WITH THE HELP OF PCB

- Process Scheduling: The different information like Process priority, process state, and resources used can be used by the OS to schedule the process on the execution stack. The scheduler checks the priority and other information to set when the process will be executed.
- Multitasking: Resource allocation, process scheduling, and process synchronization altogether helps the OS to multitask and run different processes simultaneously.
- Context Switching: When context switching happens in the OS the process state is saved in the CPU register and a copy of it is stored in the PCB. When the CPU switches to another process and then switches back to that process the CPU fetches that value from the PCB and restores the previous state of the process.
- Resources Sharing: The PCB stores information like the resources that a process is using, such as files open and memory allocated. This information helps the OS to let a new process use the resources which are being used by any other process to execute sharing of the resources.

ADVANTAGES OF USING PROCESS CONTROL BLOCK

As, PCB stores all the information about the process so it lets the operating system execute different tasks like process scheduling, context switching, etc.

Using PCB helps in scheduling the processes and it ensures that the CPU resources are allocated efficiently.

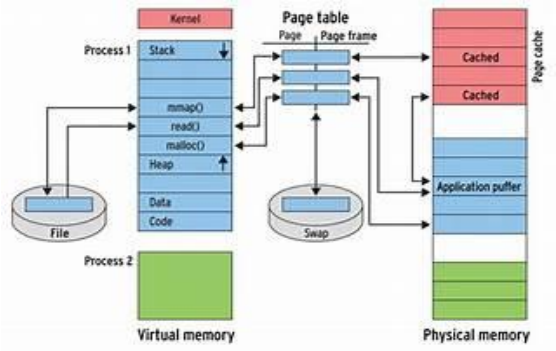
When the different resource utilization information about a process are used from the PCB they help in efficient resource utilization and resource sharing.

The CPU registers and stack pointers information helps the OS to save the process state which helps in Context switching.

DISADVANTAGES OF USING PROCESS CONTROL BLOCK

To store the PCB for each and every process there is a significant usage of the memory in there can be a large number of processes available simultaneously in the OS. So using PCB adds extra memory usage.

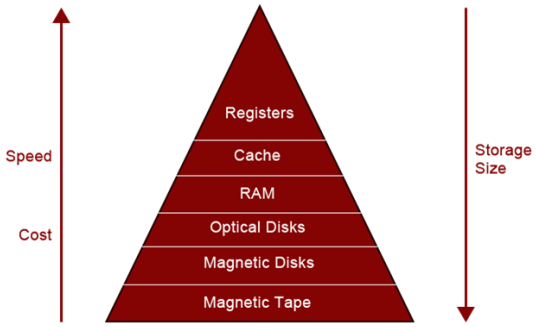
Using PCB reduces the scalability of the process in the OS as the whole process of using the PCB adds some complexity to the user so it makes it tougher to scale the system further.



VIII. MEMORY MANAGEMENT

Memory management is a vital and complex operating system task. It enables running multiple processes simultaneously without interruptions.

The main components in memory management are a processor and a memory unit. The efficiency of a system depends on how these two key components interact.



Efficient memory management depends on two factors:

1. Memory unit organization. Several different memory types make up the memory unit. A computer's memory hierarchy and organization affect data access speeds and storage size. Faster and smaller caches are closer to the CPU, while larger and slower memory is further away.

Knowing how memory management in operating systems works is crucial for system stability and improving system performance.

2. Memory access. The CPU regularly accesses data stored in memory. Efficient memory access

influences how fast a CPU completes tasks and becomes available for new tasks. Memory access involves working with addresses and defining access rules across memory levels.

Memory management balances trade-offs between speed, size, and power use in a computer. Primary memory allows fast access but no permanent storage. On the other hand, secondary memory is slower but offers permanent storage.

WHY IS MEMORY MANAGEMENT NECESSARY?

Main memory is an essential part of an operating system. It allows the CPU to access the data it needs to run processes. However, frequent read-and-write operations slow down the system.

Therefore, to improve CPU usage and computer speed, several processes reside in memory simultaneously. Memory management is necessary to divide memory between processes in the most efficient way possible.

As a result, memory management affects the following factors:

- Resource usage. Memory management is a crucial aspect of computer resource allocation. RAM is the central component, and processes use memory to run. An operating system decides how to divide memory between processes. Proper allocation ensures every process receives the necessary memory to run in parallel.
- Performance optimization. Various memory management mechanisms have a significant impact on system speed and stability. The mechanisms aim to reduce memory access operations, which are CPU-heavy tasks.
- Security. Memory management ensures data and process security. Isolation ensures processes only use the memory they were given. Memory management also enforces access permissions to prevent entry to restricted memory spaces.

Operating systems utilize memory addresses to keep track of allocated memory across different processes.

- Memory Addresses

Memory addresses are vital to memory management in operating systems. A memory address is a unique identifier for a specific memory or storage location. Addresses help find and access information stored in memory.Memory management tracks every memory location, maps addresses, and manages the memory address space. Different contexts require different ways to refer to memory address locations.

The two main memory address types are explained in the sections below. Each type has a different role in memory management and serves a different purpose.

- Physical address

A physical address is a numerical identifier pointing to a physical memory location. The address represents the actual location of data in hardware, and they are crucial for low-level memory management. Hardware components like the CPU or memory controller use physical addresses. The addresses are unique and have fixed locations, allowing hardware to locate any data quickly. Physical addresses are not available to user programs.

- Virtual Addresses

A virtual address is a program-generated address representing an abstraction of physical memory. Every process uses the virtual memory address space as dedicated memory.Virtual addresses do not match physical memory locations. Programs read and create virtual addresses, unaware of the physical address space. The main memory unit (MMU) maps virtual to physical addresses to ensure valid memory access.

The virtual address space is split into segments or pages for efficient memory use.

STATIC VS. DYNAMIC LOADING

Static and dynamic loading are two ways to allocate memory for executable programs. The two approaches differ in memory usage and resource consumption. The choice between the two depends on available memory, performance results, and resource usage.

Static loading allocates memory and addresses during program launch. It has predictable but inefficient resource usage where a program loads into memory along with all the necessary resources in advance. System utilities and applications use static loading to simplify program distribution. Executable files require compilation and are typically larger files. Real-time operating systems, bootloaders, and legacy systems utilize static loading.

Dynamic loading allocates memory and address resolutions during program execution, and a

program requests resources as needed. Dynamic loading reduces memory consumption and enables a multi-process environment. Executable files are smaller but have added complexity due to memory leaks, overhead, and runtime errors. Modern operating systems (Linux, macOS, Windows), mobile operating systems (Android, iOS), and web browsers use dynamic loading.

Static and dynamic linking are two different ways to handle libraries and dependencies for programs. The memory management approaches are similar to static and dynamic loading: Static linking allocates memory for libraries and dependencies before compilation during program launch. Programs are complete, and do not seek external libraries at compile time.

Dynamic linking allocates memory for libraries and dependencies as needed after program launch. Programs search for external libraries as the requirement appears after compilation. Static loading and linking typically unify into a memory management approach where all program resources are predetermined. Likewise, dynamic loading and linking create a strategy where programs allocate and seek resources when necessary. Combining different loading and linking strategies is possible to a certain extent. The mixed approach is complex to manage but also brings the benefits of both methods.

SWAPPING

Swapping is a memory management mechanism operating systems use to free up RAM space. The mechanism moves inactive processes or data between RAM and secondary storage (such as HDD or SSD).

The swapping process utilizes virtual memory to address RAM space size limits, making it a crucial memory management technique in operating systems. The technique uses a section from a computer's secondary storage to create swap memory as a partition or file.

Swap space enables exceeding RAM space by dividing data into fixed-size blocks called pages. The paging mechanism tracks which pages are in RAM and which are swapped out through page faults.Excessive swapping leads to performance degradation due to secondary memory being slower. Different swapping strategies and swappiness values minimize page faults while ensuring that only essential data is in RAM.

FRAGMENTATION

Fragmentation is a consequence that appears when attempting to divide memory into partitions. An operating system takes a part of the main memory, leaving the rest available for processes which divides further into smaller partitions. Partitioning does not utilize virtual memory.

There are two approaches to partitioning remaining memory: into fixed or dynamic partitions. Both approaches result in different fragmentation types:

- Internal. When remaining memory is divided into equal-sized partitions, programs larger than the partition size require overlaying, while smaller programs take up more space than needed. The unallocated space creates internal fragmentation.
- External. Dividing the remaining memory dynamically results in partitions with variable sizes and lengths. A process receives only the memory it requests. The space is freed when it is completed. Over time, unused memory gaps appear, resulting in external fragmentation.

INTERNAL AND EXTERNAL FRAGMENTATION

Internal fragmentation requires design changes. The typical resolution is through the paging and segmentation mechanism.

External fragmentation requires the operating system to periodically defragment and free up unused space.

MEMORY MANAGEMENT TECHNIQUES

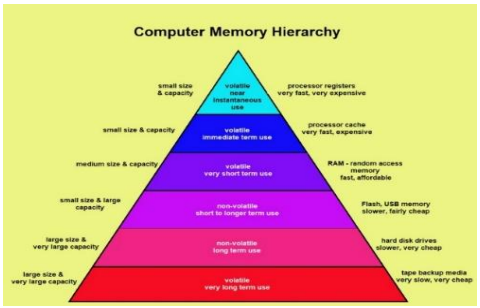
Different memory management techniques address the various issues that appear because of memory organization. One of the main goals is to improve resource usage on a system.

There are two main approaches to memory allocation and management: contiguous and non-contiguous. Both approaches have advantages, and the choice depends on system requirements and hardware architecture.

CONTIGUOUS MEMORY MANAGEMENT SCHEMES

Contiguous memory management schemes allocate continuous memory blocks to processes. Memory addresses and processes are linear, which makes this system simple to implement.

There are different ways to implement contiguous memory management schemes. The sections below provide a brief explanation of notable schemes.

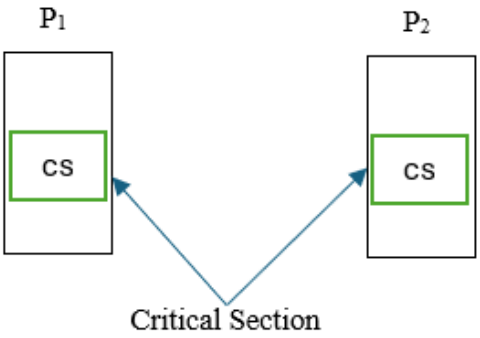


IX. SYNCHRONIZATION

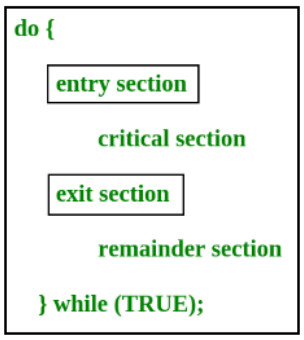
Before introducing synchronization, should identify how is the background of its. Sometimes process can be executed concurrently, sometimes one process executes, and others execute after it. If processes execute concurrently there will be an issue regarding the shared variables. Therefore, we use the "Race Condition". According to that outcome is dependent on the order of execution. To achieve this, we need synchronization. It makes sure that only one process at a time can manipulate this shared variable.

CRITICAL SECTION PROBLEM

If take n processes each process has a segment code name "Critical Section".



Critical Section problems are to design a protocol to achieve synchronization. That means, solving this to solution for synchronization to design a protocol. That process can be called a "Critical Section Problem".



In the critical section, there should be one process. Entry section there will be many processes. But by ordering two processes, select who is next executed by the selection process.

There are three pre-requirements for solutions to critical-section problems. There are,

- Mutual Exclusion – it means only one process at a time in the critical section.
- Progress – if there is no process executing in a critical section, and other processes are waiting outside the critical section, (it means they wait on the entry section), then only those processes that are not executing in their reminder section can participate in deciding which will enter the critical section next.
- Bounded Waiting - it means if a process has made a request to enter its critical section and before that request is granted, a bound must exit on the number of times that other processes are allowed to enter their critical sections.

SOLUTIONS FOR CRITICAL SELECTION PROBLEMS

There are several solutions for this one.

1. Interrupt – based solution
2. Software Solution1
3. Peterson's Solution
4. Synchronization Hardware

1. Interrupt–based solution

This solution is based on the Interrupt. Here the entry section disables the interrupts. The reason is, that if not disable the interrupt while the critical section is executing, there can be an interrupt happening, to stop that disable that entry section and enable the interrupt and the exit section.

Entry Section(disable the interrupts)

Critical Section

Exit Section (enable the interrupts)

Interrupts-based solution does not follow the above requirements. If the code runs for an hour in the

critical section, can some processes never enter their critical section. Therefore, this solution is not good.

2. Software Solution1

This solution is a process solution. In here machine language instructions are atomics. That means it cannot be interrupted. There are no breakpoints. "turn" is the variable used to indicate whose turn it is to enter the critical section. Therefore, the two processes share one variable.

int turn;

if the turn = 0; that means P_0 has to chance to enter the critical section.

This Software Solution1 only preserved Mutual Exclusion. Reason is,

If *turn = i* ;

Then p_i only can enter the critical section.

Specially turn can not be 0 and 1 at the same time.

It does not preserve Progress requirements and Bounded waiting requirements.

3. Peterson's Solution

This one has two process solution same as Software Solution1. But here two processes share two variables. They are,

int turn; and *boolean flag* [2]

the "turn" indicates who can enter the critical section and the "boolean flag" indicates if a process is ready to enter the critical section.

```
do {
    flag[i] = TRUE ;
    turn = j ;
    while (flag[j]  &&  turn == j) ;
        critial section
    flag[i] = FALSE ;
        remainder section
} while (TRUE) ;
```

This Peterson's Solution is preserved all three critical sections requirements.

However, Perterson's solution architecture is not working for modern architecture. Because modern architecture has several processes, several CPU cycles, and multiple thread processers.

4. Synchronization Hardware

Here hardware support is to implement the critical section code. Therefore there are mainly two types support this process.

- Hardware instructions
- Atomic variables.

Hardware instructions

That allows it to execute automatically. It means without any interruptions. It has two parts.

- Test-and-Set instruction
- Compare-and-Swap instruction

In Test-and-Set instruction is only preserved Mutual Exclusion". But the Compare-and-Swap instruction is preserved in all three requirements.

Atomic variables

This one is the same as the Compare-and-Swap instruction. Atomic variables mean that update without any interruption. They update on basic data types such as integers and booleans.

These solutions are complicated and generally inaccessible. Therefore Operating system designers introduce software tools for this critical selection problem. There are,

- Mutex Locks
- Semaphore
- Condition Variables
- Monitor

Mutex Locks

This is the simplest version. Here, boolean variables indicate if the lock is available or not.

The process of mutex lock is,

Get the lock,
Do the work on the lock,
Release the lock.

There are mainly two types of functions used to protect a critical section.

acquire ()
release ()

these two must be atomic and it is implemented via hardware atomic instructions. If get the lock at that time we should have to wait.

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

That means it needs to be busy waiting. Therefore, that lock can be called a "spinlock".

Semaphore

This is the more sophisticated way of synchronization tool. In Semaphore we indicate "S" as an integer variable. It can only accessed via these two atomic operations.

- wait() ⟶ P()
- signal() ⟶ V()

wait(S) is used to decrease the values and signal(S) is used to increase the value.

- Definition of the `wait()` operation

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

- Definition of the `signal()` operation

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

There are two types of Semaphores.

- Counting Semaphore – this one is used when we have a finite number of instances from a resource. It initializes the value which is equal to the number of available resources.
- Binary Semaphore – this one is the same as Mutex Locks. The integer value range depends on only between 0 and 1. Here one Semaphore will shared among several processes in Binary Semaphore.

If the Semaphore becomes 0, that means all resources have been used and if the process wishes to use these resources will have to wait until the value becomes the positive value.

Semaphore Implementation

In the semaphore implementation there are no two processes that can execute the wait() and signal() at the same time. If the wait() and signal() codes are placed in the critical section at that time semaphore becomes the critical section problem.

Problems of Semaphore

The main problem is the incorrect syntax use of operations.

Ex –

signal(mutex) … wait(mutex)
wait(mutex) … wait(mutex)
omitting of *wait(mutex)* and / or *signal(mutex)*

Condition Variables

This way allows processes to wait until a certain condition is met.

If we take *conditions x, y;*
x.wait() – the process is suspended until ***x.signal***
x.signal() – resumes one of a process that invoked ***x.wait()***

Monitors

This mechanism provides convenient and effective Synchronization. In here Only one process may be within the monitor at a time.

```
monitor monitor-name
{
        shared variable declarations
        procedure body P1 (...) {
                . . .
        }
        procedure body P2 (...) {
                . . .
        }
        procedure body Pn (...) {
                . . .
        }
        {
                initialization code
        }
}
```

CLASSICAL PROBLEMS OF SYNCHRONIZATION.

This one is used to test newly – proposed synchronization schemes. There are mainly three problems that are considered classical problems.

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining- Philosophers Problem

Bounded-Buffer Problem

This type of problem is also called the "Producer-Consumer Problem". There can be *n* buffers and each buffers can hold one item.

According to that,

Semaphore "mutex" initialized to the value 1
Semaphore "full" initialized to the value 0
Semaphore "empty" initialized to the value n

In this Bounded-Buffer Problem, there are 2 structures

1. structure of the producer process

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

2. structure of the consumer process

```
Do {
    wait(full);
    wait(mutex);
       ...
    /* remove an item from buffer to next_consumed */
    /* see slide 7 */
       ...
    signal(mutex);
    signal(empty);
       ...
    /* consume the item in next consumed */
       ...
} while (true);
```

Readers – Writers Problems

This problem depends on two parts.

1. Readers – only read the data set
2. Writers – can both read and write

If there are two readers reader or multiple readers reading the data set, there are no issues. But when one is reading and another one is writing then there can be an issue.

According to that, we should consider three parts.

1. Shared Data

   Semaphore *rw_mutex* initialized to 1
   Semaphore *mutex* initialized to 1
   Integer *read_count* initialized to 0

2. Structure of a writer process

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

3. Structure of reader process

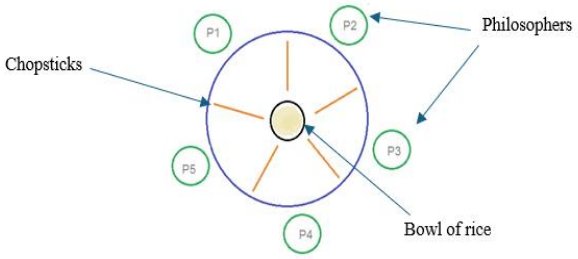
```
while (true){
        wait(mutex);
        read_count++;
        if (read_count == 1)
        wait(rw_mutex);
        signal(mutex);
        ...
        /* reading is performed */
        ...
        wait(mutex);
        read count--;
        if (read_count == 0)
                signal(rw_mutex);
        signal(mutex);
}
```

There are mainly 5 parameters considered in this situation.

- One data is shared among several processes.
- If once the writer is ready, it performs its write. That means only one writer may write at a time.
- If a process is writing, no other processers can read it.
- If the reader is reading, there is no other process can write.
- Readers may not write and only read.

Dining-Philosophers Problem

This problem is used to evaluate situations where there is a need to allocate multiple resources to multiple processes.



Consider 5 philosophers sitting around a circular table and in the middle, there is a bowl of rice. There are 5 chopsticks scattered as shown in the diagram. In any instance, a philosopher is either thinking or eating.

When a philosopher wants to think, he keeps

down both chopsticks. But in case they do not interact with their neighbors. They try to a chopstick to eat from the bowl. Therefore, they take one chopstick at a time. Another chopstick they released.

According to that, a bowl of rice is the same as shard data, and chopsticks same as a semaphore. There are 5 semaphore chopsticks and they are initialized to 1. (Semaphore chopsticks[5] initialized 1).

Problem Algorithm

```
while (true){
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );

     /* eat for awhile */

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

     /* think for awhile */

}
```

Solution to Dining Philosophers

According to the problem, a philosopher can think for an indefinite amount of time and when a philosopher starts eating, he has to stay at some point in time. Therefore, philosophers are in an endless cycle of thinking and eating.

```
    void test (int i) {
            if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {
                 state[i] = EATING ;
             self[i].signal () ;
            }
    }

        initialization_code() {
           for (int i = 0; i < 5; i++)
           state[i] = THINKING;
         }
}
```

Advantages of Process Synchronization

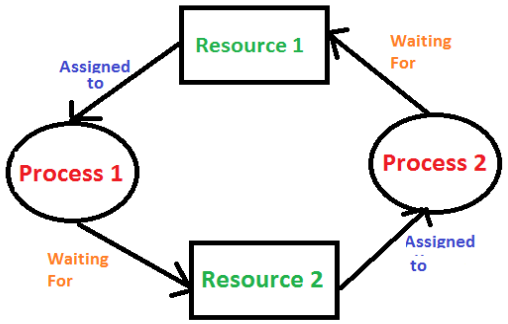
- Ensures data integrity and consistency.
- Avoids race conditions.
- Supports efficient and effective use of shared resources.
- Prevents inconsistent data due to concurrent access.

Disadvantages of Process Synchronization

- Adds overhead to the system and increases the complexity of the system.
- Leads to performance degradation.
- Can cause deadlock if it is not implemented properly.

IX. DEADLOCK MANAGEMENT

In operating systems, a deadlock happens when two or more processes can't move forward because they're waiting on each other to release a resource. Stated differently, this refers to a state in which all processes are unable to advance because of their circular dependencies.



Normally, a process has to follow these steps to request a resource before using it and release it when finished:

1. Request - Should the request not be approved right away, the procedure must wait for the necessary resource or resources to become available. For example, the system calls **open(), new(), and request()**.
2. Use - The process makes use of the resource, reading from a file or printing to a printer, for example.
3. Release - The resource is released by the procedure. in order for it to be accessible for other procedures. For example, **close(), free(), delete(), and release().**

The kernel maintains track of all resources under its control, including those that are allocated to specific processes, those that are free, and a queue of processes that are waiting for a resource to become available. Applications that manage resources can be managed using binary or counting semaphores, mutexes, or wait() and signal() calls.

When all of the processes in a set are waiting for a resource that is being assigned to another process (and can only be released when that other waiting process advances), the set is said to be deadlocked.

DEADLOCK CHARACTERIZATION

DEADLOCK WITH MUTEX LOCKS

Let's see how deadlock can occur in a multithreaded Pthread program using mutex locks. The `pthread_mutex_init()` function initializes an unlocked mutex. Mutex locks are acquired and released using `pthread_mutex_lock()` and `pthread_mutex_unlock()`, respectively. If a thread attempts to acquire a locked mutex, the call to `pthread_mutex_lock()` blocks the thread until the owner of the mutex lock invokes `pthread_mutex_unlock()`.

Two mutex locks are created in the following code example:

```
/* Create and initialize the mutex locks */
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);
```

Next, two threads – thread_one and thread_two – are created, and both these threads have access to both mutex locks. thread_one and thread_two run in the functions do_work_one() and do_work_two(), respectively, as shown below:

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

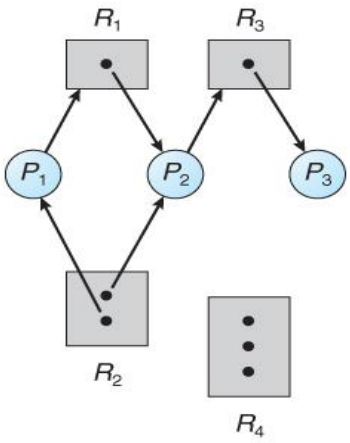
In this example, thread_one attempts to acquire the mutex locks in the order (1) first_mutex, (2)second_mutex, while thread_two attempts to acquire the mutex locks in the order (1) second_mutex, (2) first_mutex. Deadlock is possible if thread_one acquires first _mutex while thread_two acquires second mutex.

- There are four conditions that are necessary to achieve deadlock:
  1. **Mutual Exclusion -** One or more resources must be kept in a non-sharable mode; if another process asks for this resource, it has to wait for its release.
  2. **Hold and Wait** - A process needs to be holding at least one resource and waiting on at least one resource that another process is holding at the same time.
  3. **No preemption** - A process cannot lose control over a resource unless it releases it willingly once the resource is in its possession (that is, after its request has been approved).
  4. **Circular wait** - For every P[i], there must be a set of processes { P0, P1, P2,..., PN } that wait for P[(i + 1) % (N + 1)]. Although the hold-and-wait condition is implied by this condition, handling the conditions is simpler if the four are taken into consideration independently.

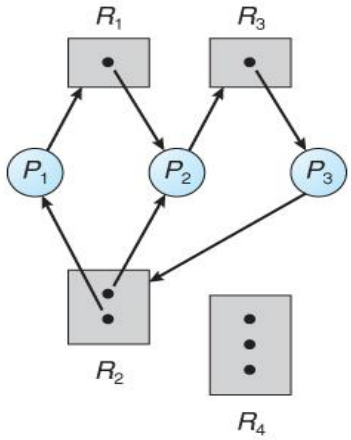
- **Deadlocks can sometimes be better understood by using Resource-Allocation Graphs, which include the following characteristics:**

  1. A group of resource categories that show up on the graph as square nodes, { R1, R2, R3,..., RN }. Specific instances of the resource are indicated by dots inside the resource nodes. (For instance, two dots could stand for two laser printers.)
  2. A set of processes {P1, P2, P3, P4, ...Pn}
  3. Request Edges – A collection of directed arcs connecting Pi to Rj, signifying that Pi has requested Rj and is awaiting its availability.
  4. Assignment Edges - a collection of directed arcs from Rj to Pi that show Pi is now in possession of resource Rj and that Rj has been assigned to process Pi.
  5. It should be noted that after a request is approved, the arc's orientation can be reversed to turn the request edge into an assignment edge. (But also take note of the fact that assignment edges originate from a specific instance dot inside the box, whereas request edges point to the category box.)

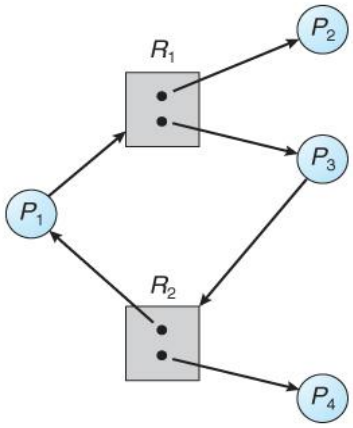
As an illustration:



The system is not stalled if there are no cycles in the resource allocation graph. (Keep in mind that these are directed graphs when searching for cycles.) Refer to the Diagram below for an example.



A stalemate occurs when there are cycles in a resource-allocation graph and there is only one instance of each resource category.



A cycle in the resource-allocation graph suggests the likelihood of a deadlock but does not ensure one if a resource category has several instances. Take a look at the Figures here, for instance.

In general, deadlocks can be resolved in three ways:

1. Deadlock prevention and avoidance – Keep the system from becoming stuck in a deadlock.
2. Deadlock detection and recovery – When deadlocks are found, terminate the process or restrict specific assets.
3. Ignore the problem altogether – If deadlocks only happen once a year or so, it might be preferable to just let them happen and restart the system when needed rather than paying the ongoing costs and performance penalties linked to deadlock detection or prevention. This is the stance that UNIX and Windows adopt.

Deadlock Prevention

By preventing at least one of the four necessary circumstances, deadlocks can be avoided:

1. **Mutual Exclusion** - Read-only files and other shared resources don't cause deadlocks. Sadly, certain resources like tape drives and printers need to be accessed only by one process at a time.
2. **Hold and Wait** - We can limit how processes obtain resources to prevent them from becoming stuck while others wait for resources (deadlock). However there are compromises associated with these limitations. Forcing processes to request all resources upfront is one way, but it can be inefficient if not all are needed right once. An alternative would be to mandate that processes relinquish all resources they have held before requesting new ones. However, this would put them at risk of losing the released resources if another process claims them first. Both approaches have the potential to starve processes that require popular resources.
3. **No Preemption** - Deadlocks may be avoided by forcing a waiting process to yield its resources so that they can be reallocated through preemption. There are two primary methods:
   - As with upfront requests, the process releases all of its resources while it waits for a new one.
   - In order to fulfill the new request, the system may pilfer resources from another waiting process. These techniques are effective for resources that can be readily saved, such as memory, but they are ineffective for printers and other devices where stopping a process could result in data loss.
4. **Circular Wait** - Resources can be numbered, and processes can only request them in a specific order (increasing or decreasing) to avoid circular waits in deadlocks. This makes sure that a process won't wait for a "lower" resource to hold a "higher" resource. But establishing this order is a big problem. Determining which resources have "higher" priority for a process to collect first can be challenging because different resources may not naturally follow a hierarchy.

Deadlock Avoidance

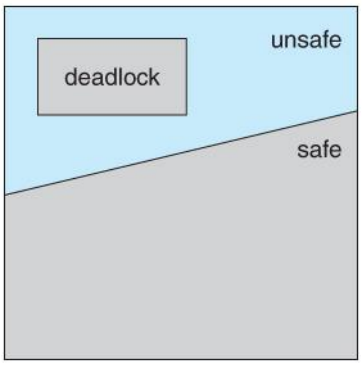
While deadlock avoidance seeks to completely eliminate deadlocks, it necessitates more advance knowledge about processes than deadlock detection. This may be too cautious and result in less efficient use of resources. While certain algorithms can only utilise the greatest number of resources needed for each activity, others might benefit from the precise

sequence in which resources are needed. In essence, the scheduler foresees whether approving a request might result in a future impasse and prevents it by rejecting the request or postponing the action. The resource allocation state, which takes into account allotted resources, available resources, and the greatest potential needs of all operations, is crucial in making these decisions.

Safe State

When all processes may use the resources they require from the system (up to their specified maximums) without experiencing a stalemate, the condition is considered safe. Formally speaking, a state is secure if there is a safe sequence of processes { P0, P1, P2,..., PN } such that all procedures Pj where j < i and all resource requests for Pi can be fulfilled with the resources currently allotted to Pi. (In other words, Pi will be able to finish using the resources that have been freed up if all the processes that came before it finished free up their resources.)

The system is in an unsafe state that MAY result in deadlock if a safe sequence is absent.



As an illustration, let's look at a system that has 12 tape drives distributed as follows. Is this condition safe? Which order is safe?

	Maximum Needs	Current Allocation
P0	10	5
P1	4	2
P2	9	2

The safety of the system depends on the specifics of the table (resource availability, allocations, and maximum demands) if process P2 asks and is granted another tape drive. The allocation would only be accepted in accordance with the safe state method if the request is granted while maintaining a safe state. This implies that in order to prevent deadlocks, the operating system would have to anticipate future resource allocations based on process needs and make sure a sequence exists wherein all processes can eventually get the resources they require.

Deadlock Detection

Unlike avoidance, which aims to eliminate deadlocks completely, deadlock detection detects deadlocks that have already occurred in the system. By looking for cycles in a resource-allocation graph, it functions on systems with solitary resource instances. Processes, resources, and their distribution are displayed in this graph. Incorporating "claim edges" enhances detection.

These dotted lines connect a procedure to possible requests for resources in the future. The system is able to detect circular wait scenarios, or deadlocks, in which processes are stalled while waiting for resources that are jointly held by other processes, by examining the graph with claim edges.

Recovery from Deadlock

There are three fundamental methods for resolving a deadlock:

1. Inform the system operator, and allow him/ her to take manual intervention.
2. Terminate one or more processes involved in the deadlock.
3. Preempt resources.

Process Termination

- Deadlocks can be recovered from in two basic ways, both of which recover resources from ended processes. The first, more drastic method ends all connected processes. While it can be inefficient, this ensures the resolution of deadlocks. The second, more cautious strategy kills each process individually until the impasse is broken. Although it requires repeated deadlock detection after each termination, this is less disruptive.
- In the latter scenario, a variety of criteria may be taken into consideration when determining which operations to end next:
  - Set priorities for processes.
  - How long has the procedure been ongoing and how near its completion it is?
  - The quantity and kind of resources that the process is retaining. (Is it simple to anticipate and recover them?)
  - How many more resources are required for the procedure to be finished?
  - How many procedures will have to be stopped?
  - whether batch or interactive processing is used.

- (Regardless of whether the process has altered any resource in a way that cannot be reversed.)

Resource Preemption

Three key concerns need to be taken into consideration when preempting resources to break a deadlock:

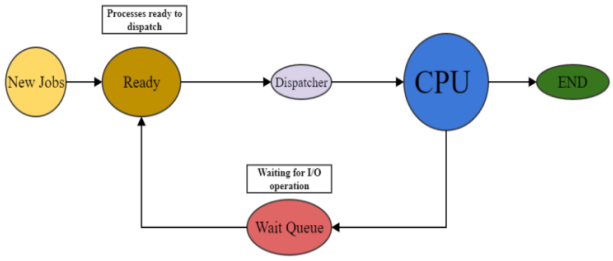
1. **Selecting a victim** - Many of the same principles for decision-making apply when deciding which resources to forego throughout which procedures.
2. **Rollback** - The ideal scenario would be to restore a preempted process to a safe state before the resource was first assigned to the process. Since identifying such a safe condition can often be challenging or impossible, the only safe rewind option is to go back to the starting point.
3. **Starvation** - How can you ensure that a process that is continuously preempted from resources won't starve? Using a priority system and raising a process's priority each time one of its resources is preempted is one way to go about things. It should eventually gain sufficient importance to stop being preempted.

XI. CPU SCHEDULING

CPU scheduling is an essential operating system concept that deals with dividing up a computer system's central processing unit (CPU) time across several processes.

Consider having many programs open at once. Which program uses the CPU's processing capacity at any given time is determined by CPU scheduling. The goal of this allocation is optimal performance, taking responsiveness, efficiency, and justice into account. CPU scheduling guarantees that all programs have an opportunity to proceed and maintains your computer feeling responsive by carefully selecting which process to run next.

To better comprehend this, let us consider a scenario involving billing waits at a grocery. Assume that people in the queue are instances of processes that are in the "Ready Queue" state. The CPU and the cashier are comparable. As a process in the "running" stage, the cashier attends to one client at a time. If a new customer enters (an interruption), the cashier may transition to servicing that customer (preemption).



General flow of the CPU Scheduling

The process that will be using up the CPU is New Jobs, as can be seen in the diagram above. The process will go to "Ready Queue" once it is prepared. The CPU scheduler will choose another process and the dispatcher will move it from the ready queue to the CPU for execution if there isn't already a process executing on the CPU. The CPU processes that need to wait for an I/O operation will be shifted to the "wait queue" while the other processes continue to operate. The process will advance from the wait queue to the ready queue after the I/O operation is finished. CPU scheduling will indicate that a process is finished if it has finished using the CPU.

CPU Scheduler

For multitasking, the CPU scheduler functions as a conductor. To keep things moving along smoothly, the scheduler selects the next process from the "ready queue" after the CPU completes a job. In contrast to a simple line (FIFO), the selected procedure is determined by the particular scheduling method that is used. This algorithm establishes the order of precedence for each running program, guaranteeing equity, effectiveness, and responsiveness for all of them.

Dispatcher

The module known as the dispatcher is responsible for granting the scheduler-selected process CPU control. This role entails:

1. Context switching
2. Switching to user mode
3. Jumping to the proper location in the newly loaded program

Since the dispatcher is executed at every context switch, it must operate as quickly as possible. **Dispatch latency** is the amount of time the dispatcher takes.

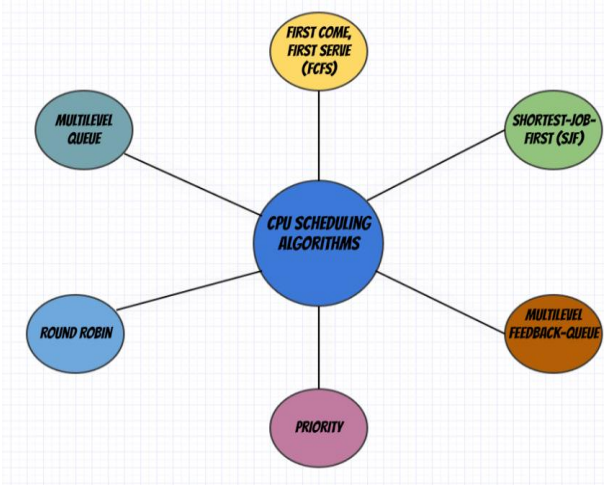
Scheduling Criteria

The concepts and elements that operating systems take into account when determining how much CPU

time to allot to various processes are known as scheduling criteria.

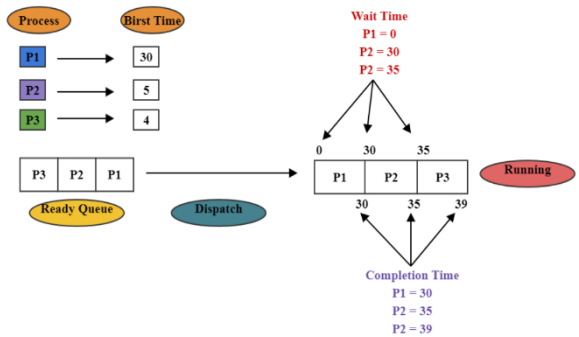
- **CPU Utilization**: Making the most of CPU utilization guarantees that the processor is working on tasks.
- **Throughput**: Total number of processes that finish running in a specified amount of time
- **Turnaround Time**: The amount of time needed for a specific process to finish, from the point of submission until its conclusion.
- **Waiting Time**: The amount of time that processes wait in the ready queue for a CPU opportunity.
- **Response Time**: The time taken for a system to respond to an input or request.
- **Context Switching Overhead**: Maintaining system efficiency requires minimizing the overhead related to context switching across processes.

CPU SCHEDULING ALGORITHMS IN OS



First come first serve scheduling, FCFS

Similar to a FIFO queue, this scheduling is quite straightforward, yet it results in some extremely long average wait times.



The real flow of FCFS scheduling, assuming 0 arrival time, is depicted in the above image. P2 and P3 must wait in the ready queue for P1 to finish its 30-second burst duration once it begins operating. After P1 is over, P2 and P3 will get a turn running.

FCFS works well in situations when processes have comparable burst periods and where waiting time optimization is less important than simplicity. However, because of the possibility of lengthy wait times, it might not be the ideal option for many applications in real life.

Advantages of FCFS

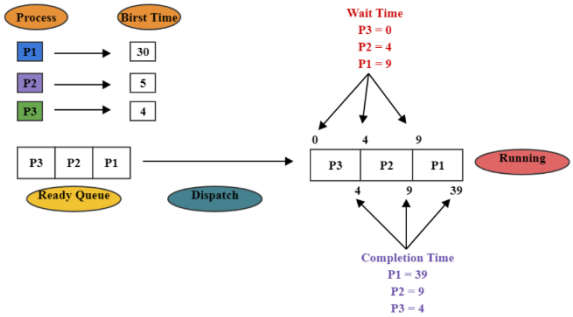
- Easy to implement
- First come, first serve method

Disadvantages of FCFS

- FCFS experiences the Convoy effect.
- Compared to other algorithms, the average waiting time is substantially higher.
- FCFS is not very efficient because it is fairly straightforward and simple to use.

Shortest Job First Scheduling, SJF

The process with the least burst time is chosen by the SJF algorithm to run first. It seeks to cut down on waiting times and the overall processing time. Shortest Remaining Time First, or SJF, can be non-preemptive or preemptive.



The image above makes it quite evident that the process with the smallest burst time will be given the opportunity first and will then proceed in the same order, assuming a 0 arrival time.

This reduces turnover and waiting times and is effective in terms of overall processing time.

Its implementation in a non-preemptive setting is challenging since it depends on knowing when subsequent bursts will occur. If lengthy jobs follow short jobs, it might also result in famine.

Advantages of SJF Scheduling

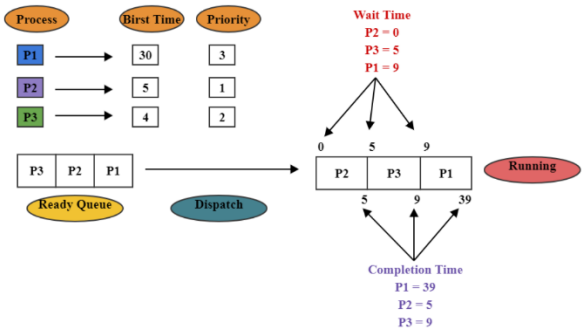
- SJF scheduling algorithm is superior to the first come, first served scheduling algorithm since it lowers the average waiting time.
- SJF is typically employed in long-term planning.

Disadvantages of SJF Scheduling

- Starvation is among the demerits of SJF.
- Predicting how long a CPU request would take may often become challenging.

PRIORITY SCHEDULING

Using this approach, the process that has the highest priority is chosen to be executed first, and each process is given a number. Similar priority processes are arranged according to other parameters, including arrival time or different scheduling methods.



It makes it possible to rank the importance of certain procedures. Real-time applications where some tasks require immediate attention can be managed with this.

If higher-priority processes keep coming in, lower-priority processes will starve to death. It might be necessary to use additional scheduling criteria (like FCFS) in the event of equal priority.

Advantages of Priority Scheduling

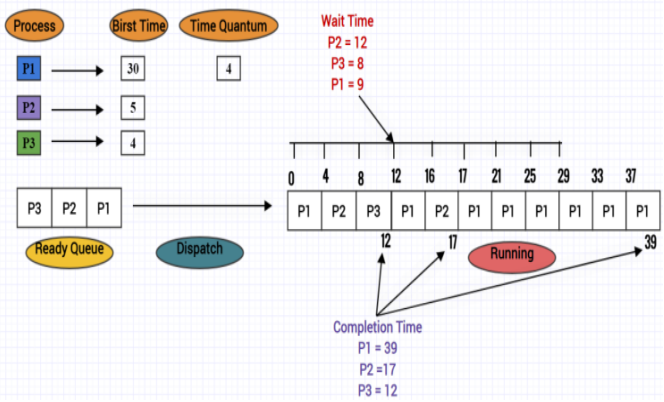
- The average waiting time is less than FCFS.
- Less complex.

Disadvantages of Priority Scheduling

- The Starvation Problem is one of the most often mentioned drawbacks of the Preemptive Priority CPU scheduling technique. This refers to the issue when a process needs to wait longer to be scheduled into the CPU. The hunger problem is the name given to this situation.

Round Robin Scheduling

With the Round Robin CPU scheduling algorithm, a set time slot is cyclically assigned to each process. It is the First Come First Serve CPU Scheduling algorithm in a preemptive manner. The time-sharing method is often the main focus of the Round Robin CPU Algorithm.



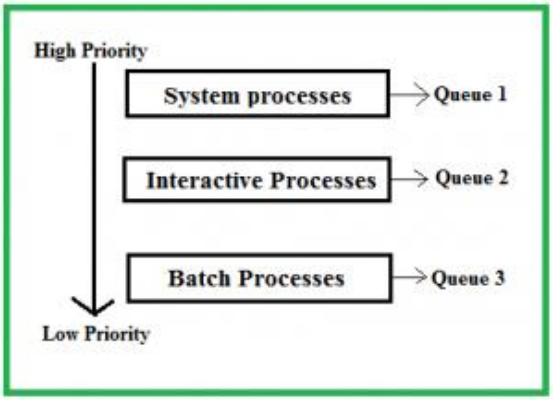
For CPU allocation, round-robin (RR) scheduling is similar to a roundtable discussion. Each process receives a set time slice (time quantum) and rotates in a fair circle (ready queue). This makes it perfect for multitasking systems by guaranteeing that every process has a chance to use the CPU. But RR isn't flawless. It may lag short processes (having to wait behind longer ones for their next turn) and struggle with long processes (waiting a long time for their next turn). Even with these disadvantages, RR is still a popular option for systems like time-sharing and interactive environments where everyone needs an opportunity to be heard because of its fairness.

Advantages of Round Robin Scheduling

- Since each process receives an equal portion of the CPU, round robin appears to be fair.
- The newly formed process gets moved to the back of the queue once it is ready.

Multilevel Queue Scheduling

The ready queue is divided into several queues using this scheduling strategy, each with a different priority level. Each queue has its scheduling mechanism, and processes are assigned to it depending on particular criteria. Processes are moved between queues according to their requirements and behavior; the queues are arranged according to priority.



The description of the processes in the above diagram is as follows:

System Processes: The CPU has a process, known as a system process, that it must execute.

Interactive Processes: A process that calls for the same kind of engagement is known as an interactive process.

Batch Processes: Generally speaking, batch processing is an operating system technique where programs and data are gathered in one batch before processing begins.

Advantages of Multilevel Queue Scheduling

- The multilevel queue's primary advantage is its little scheduling overhead.

Disadvantages of Multilevel Queue Scheduling

- Starvation problem
- It is inflexible in nature

MULTILEVEL FEEDBACK QUEUE SCHEDULING

Scheduling Multilevel Feedback Queues (MLFQ) Similar to multilevel queue scheduling, CPU scheduling allows for queue switching. and so far more effective than queue scheduling with multiple levels.

Characteristics of MLFQ

- Processes entering the system are assigned to a queue permanently under a multilayer queue-scheduling method, and they are not permitted to switch across queues.
- This configuration has the benefit of minimal scheduling overhead because the processes are always assigned to the queue, but it also has the drawback of being rigid.

Advantages of MLFQ

- It is more flexible.
- It permits various processes to switch between various queues.

Disadvantages of MLFQ

- It also produces CPU overheads.
- It is the most complex algorithm.

XII. PROTECTION AND SECURITY

Why the Linux operating system need protection and security,

Maintaining the availability, confidentiality, and integrity of data and resources requires protecting and securing a Linux operating system. Like any other operating system, Linux is susceptible to various threats such as denial-of-service attacks, viruses, and unwanted access and data breaches. Sensitive data, including user credentials, financial records, and proprietary material, held on Linux systems may be compromised if insufficient security precautions are taken.

This might result in monetary losses, harm to one's reputation, and legal repercussions. Furthermore, bad actors may use unprotected Linux computers to conduct attacks against other systems or interfere with essential services, potentially resulting in extensive chaos and disruption. Organizations may assure the ongoing dependability and credibility of their Linux systems by putting strong security measures in place, such as user authentication, access limits, encryption, intrusion detection, and frequent security upgrades. Proactive security methods also help to create a safer computing environment for both users and companies by deterring potential attackers in addition to providing protection against known risks.

Program Threats**:**

- Trojan Horse - Code segment that misuses its environment mechanisms to allow programs written by some user to be executed by other users
- Trap Door - A hole in the software purposely left by the software's designer that can be used only by the designer.
- Logic bomb -A security hole that is created only when a predefined condition is met
- Stack and Buffer Overflow -The most common way for an attacker outside of the system.

Linux is a widely-used and popular operating system known for its stability, flexibility, and security. However, even with its built-in security features, Linux systems can still be vulnerable to security breaches.

LINUX SECURITY AND VULNERABILITIES: STATS,

There are fewer vulnerabilities in Linux than in some other operating systems such as Windows and macOS. However, it does not mean that all types of cyberattacks are impossible. The most common vulnerabilities for Linux are those that imply system information stealing: privilege escalation, memory corruption, and information disclosure .

These security gaps are the most frequently exploited by cyber criminals aiming at gaining unauthorized access to a Linux-developed system. The reports of such security vulnerabilities as The National Vulnerability Database and Crowdstrike demonstrate an annual increase in the numbers of Linux-developed systems. For instance, 1,958 Linux vulnerabilities had been reported in 2020 ; there was a 35% increase in malware targeting Linux within 2021 and up to 1.7 million new Linux malwares in 2022 .

LINUX SECURITY TIPS AND PRACTICES

1. Use Strong Passwords

(Basic security mechanism)

Use strong passwords and change them regularly as a basic step to securing your Linux system. Strong passwords prevent unauthorized access to the system and reduce the risk of identity theft, data loss, and other security incidents.A strong password is at least 12 characters long and includes a mixture of upper and lowercase letters, numbers, and special characters.

That makes brute-force attacks extremely more difficult.Regularly changing passwords also improves security. The process reduces the risk of password reuse and exposure, giving a potential attacker a limited time frame to exploit the password if it becomes compromised.

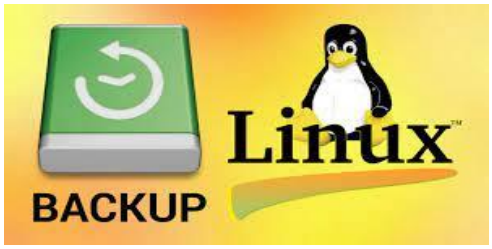
2. Ensure OpenSSH Server Security
(Intermediate security mechanism)

OpenSSH is a widely used and secure implementation of SSH for Linux systems. It provides encryption for data in transit, robust authentication methods, and a secure way to administer systems and transfer files remotely. To ensure the security of OpenSSH, minimize the tool's vulnerabilities.

Secure the OpenSSH server by following these tips:

- Use non-standard SSH ports.
- Limit user access and disable root login.
- Use SSH key pairs for authentication.
- Disable root login and password-based logins on the server.
- Keep OpenSSH updated regularly.
- Use strong authentication methods.
- Limit the number of authentication attempts.
- Disable unused protocols and features.
- Implement a firewall.
- Monitor logs regularly.

3. Backup Linux System
(Intermediate security mechanism)



Because it enables users to recover from a system breach or data loss, backing up a Linux system is essential for security. In the event of a hardware malfunction, unexpected calamity, or security breach, a data backup copy aids in restoring the system to a safe condition.

Use a cloud-based service like AWS or conventional UNIX backup tools like dump and restore to create a backup of a Linux system. Encrypt the backups and store them somewhere safe, such a NAS server or external storage device, to ensure their security.
Backup utilities are included into the majority of Linux distributions. Go to the system menu and look for backup tools to begin using backup on Linux.

4. Automatic Online Update

Moreover, YaST has the option to configure an automated update. Choose Software › Online Update via Automatic. Set up a weekly or daily update. Certain patches—like kernel updates—need human input, which would halt the automated update process. If you would want the update process to run automatically, choose Skip Interactive Patches. In this scenario, periodically do a manual Online Update to apply interaction-required fixes. The patches are downloaded at the designated time but are not installed if Only Download Patches is selected. Rpmor Zypper must be used to manually install them.

5. Encryption

Linux offers a number of encryption methods to protect data while it's in transit and at rest. Full disk encryption is offered by programs like LUKS (Linux Unified Key Setup), and encryption is available for network communication through protocols like SSH and SSL/TLS. Combining these security measures can significantly improve a Linux system's security posture. Maintaining a safe environment also requires routinely examining and upgrading security configurations, as well as remaining up to date on security best practices.

:

XIII. MASS STORAGE SYSTEM

1 Magnetic Disks

Traditional magnetic disks have the following structure:

- One or more platters in the form of disks covered with magnetic media. Hard disk platters are made of rigid metal, while "floppy" disks are made of more flexible plastic.
- Each platter has two working surfaces. Older hard disk drives would sometimes not use the very top or bottom surface of a stack of platters, as these surfaces were more susceptible to potential damage.
- Each working surface is divided into a number of concentric rings called tracks. The collection of all tracks that are the same distance from the edge of the platter, (i.e. all tracks immediately above one another in the following diagram) is called a cylinder.
- Each track is further divided into sectors, traditionally containing 512 bytes of data each, although some modern disks occasionally use larger sector sizes. (Sectors also include a header and a trailer, including checksum information among other things. Larger sector sizes reduce the fraction of the disk consumed by headers and trailers, but increase internal fragmentation and the amount of disk that must be marked bad in the case of errors.)
- The data on a hard drive is read by read-write heads. The standard configuration (shown below) uses one head per surface, each on a separate arm, and controlled by a common arm assembly which moves all heads simultaneously from one cylinder to another. (Other configurations, including independent read-write heads, may speed

up disk access, but involve serious technical difficulties.)

- The storage capacity of a traditional disk drive is equal to the number of heads (i.e. the number of working surfaces), times the number of tracks per surface, times the number of sectors per track, times the number of bytes per sector. A particular physical block of data is specified by providing the head-sector-cylinder number at which it is located.

- In operation the disk rotates at high speed, such as 7200 rpm (120 revolutions per second.) The rate at which data can be transferred from the disk to the computer is composed of several steps:

The positioning time, a.k.a. the seek time or random access time is the time required to move the heads from one cylinder to another, and for the heads to settle down after the move. This is typically the slowest step in the process and the predominant bottleneck to overall transfer rates.

The rotational latency is the amount of time required for the desired sector to rotate around and come under the read-write head.This can range anywhere from zero to one full revolution, and on the average will equal one-half revolution. This is another physical step and is usually the second slowest step behind seek time. (For a disk rotating at 7200 rpm, the average rotational latency would be 1/2 revolution / 120 revolutions per second, or just over 4 milliseconds, a long time by computer standards.

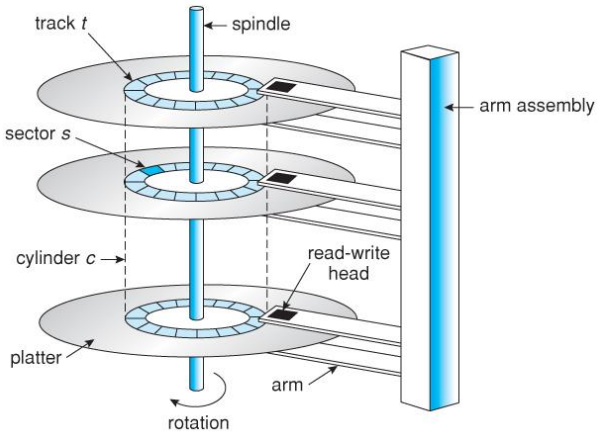
The transfer rate, which is the time required to move the data electronically from the disk to the computer. (Some authors may also use the term transfer rate to refer to the overall transfer rate, including seek time and rotational latency as well as the electronic data transfer rate.)

- Disk heads "fly" over the surface on a very thin cushion of air. If they should accidentally contact the disk, then a head crash occurs, which may or may not permanently damage the disk or even destroy it completely. For this reason it is normal to park the disk heads when turning a computer off, which means to move the heads off the disk or to an area of the disk where there is no data stored.

- Floppy disks are normally removable. Hard drives can also be removable, and some are even hot-swappable, meaning they can be removed while the computer is running, and a new hard drive inserted in their place.

- Disk drives are connected to the computer via a cable known as the I/O Bus. Some of the common interface formats include Enhanced Integrated Drive Electronics, EIDE; Advanced Technology Attachment, ATA; Serial ATA, SATA, Universal Serial Bus, USB; Fiber Channel, FC, and Small Computer Systems Interface, SCSI.

- The host controller is at the computer end of the I/O bus, and the disk controller is built into the disk itself. The CPU issues commands to the host controller via I/O ports. Data is transferred between the magnetic surface and onboard cache by the disk controller, and then the data is transferred from that cache to the host controller and the motherboard memory at electronic speeds.



Solid-State Disks - New

As technologies improve and economics change, old technologies are often used in different ways. One example of this is the increasing used of solid state disks, or SSDs.

SSDs use memory technology as a small fast hard disk. Specific implementations may use either flash memory or DRAM chips protected by a battery to sustain the information through power cycles.

Because SSDs have no moving parts they are much faster than traditional hard drives, and certain problems such as the scheduling of disk accesses simply do not apply.

However SSDs also have their weaknesses: They are more expensive than hard drives, generally not as large, and may have shorter life spans.
SSDs are especially useful as a high-speed cache of hard-disk information that must be accessed quickly. One example is to store filesystem meta-data, e.g. directory and inode information, that must be accessed quickly and often. Another variation is a boot disk containing the OS and some application executables, but no vital user data. SSDs are also used in laptops to make them smaller, faster, and lighter.

Because SSDs are so much faster than traditional hard disks, the throughput of the bus can become a limiting factor, causing some SSDs to be connected directly to the system PCI bus for example.

Magnetic Tapes

Magnetic tapes were once used for common secondary storage before the days of hard disk drives, but today are used primarily for backups.

Accessing a particular spot on a magnetic tape can be slow, but once reading or writing commences, access speeds are comparable to disk drives.
Capacities of tape drives can range from 20 to 200 GB, and compression can double that capacity

Disk Structure

The traditional head-sector-cylinder, HSC numbers are mapped to linear block addresses by numbering the first sector on the first head on the outermost track as sector 0. Numbering proceeds with the rest of the sectors on that same track, and then the rest of the tracks on the same cylinder before proceeding through the rest of the cylinders to the center of the disk. In modern practice these linear block addresses are used in place of the HSC numbers for a variety of reasons:

1. The linear length of tracks near the outer edge of the disk is much longer than for those tracks located near the center, and therefore it is possible to squeeze many more sectors onto outer tracks than onto inner ones.
2. All disks have some bad sectors, and therefore disks maintain a few spare sectors that can be used in place of the bad ones. The mapping of spare sectors to bad sectors in managed internally to the disk controller.
3. Modern hard drives can have thousands of cylinders, and hundreds of sectors per track on their outermost tracks. These numbers exceed the range of HSC numbers for many (older) operating systems, and therefore disks can be configured for any convenient combination of HSC values that falls within the total number of sectors physically on the drive.

There is a limit to how closely packed individual bits can be placed on a physical media, but that limit is growing increasingly more packed as technological advances are made.

Modern disks pack many more sectors into outer cylinders than inner ones, using one of two approaches:

- With Constant Linear Velocity, CLV, the density of bits is uniform from cylinder to cylinder. Because there are more sectors in outer cylinders, the disk spins slower when reading those cylinders, causing the rate of bits passing under the read-write head to remain constant. This is the approach used by modern CDs and DVDs.
- With Constant Angular Velocity, CAV, the disk rotates at a constant angular speed, with the bit density decreasing on outer cylinders. (These disks would have a constant number of sectors per track on all cylinders.)

Disk Attachment

Disk drives can be attached either directly to a particular host (a local disk) or to a network.

Host-Attached Storage

- Local disks are accessed through I/O Ports as described earlier.
- The most common interfaces are IDE or ATA, each of which allow up to two drives per host controller.
- SATA is similar with simpler cabling.
- High end workstations or other systems in need of larger number of disks typically use SCSI disks:

The SCSI standard supports up to 16 targets on each SCSI bus, one of which is generally the host adapter and the other 15 of which can be disk or tape drives.

A SCSI target is usually a single drive, but the standard also supports up to 8 units within each target. These would generally be used for accessing individual disks within a RAID array. (See below.)

The SCSI standard also supports multiple host adapters in a single computer, i.e. multiple SCSI busses.

Modern advancements in SCSI include "fast" and "wide" versions, as well as SCSI-2.
SCSI cables may be either 50 or 68 conductors. SCSI devices may be external as well as internal.
See wikipedia for more information on the SCSI interface.

- FC is a high-speed serial architecture that can operate over optical fiber or four-conductor copper wires, and has two variants:

A large switched fabric having a 24-bit address space. This variant allows for multiple devices and multiple hosts to interconnect, forming the basis for the storage-area networks, SANs, to be discussed in a future section.

The arbitrated loop, FC-AL, that can address up to 126 devices (drives and controllers.)

Disk Scheduling

As mentioned earlier, disk transfer speeds are limited primarily by seek times and rotational latency. When multiple requests are to be processed there is also some inherent delay in waiting for other requests to be processed.

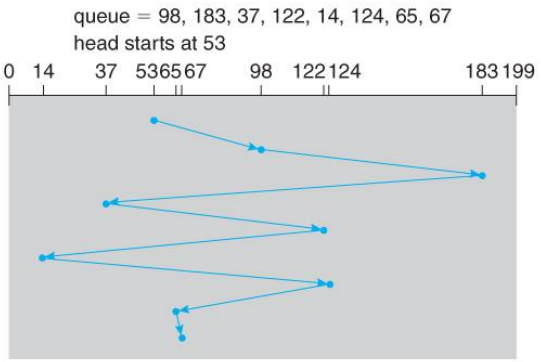
- Bandwidth is measured by the amount of data transferred divided by the total amount of time from the first request being made to the last transfer being completed, (for a series of disk requests.)

- Both bandwidth and access time can be improved by processing requests in a good order.

- Disk requests include the disk address, memory address, number of sectors to transfer, and whether the request is for reading or writing

FCFS Scheduling

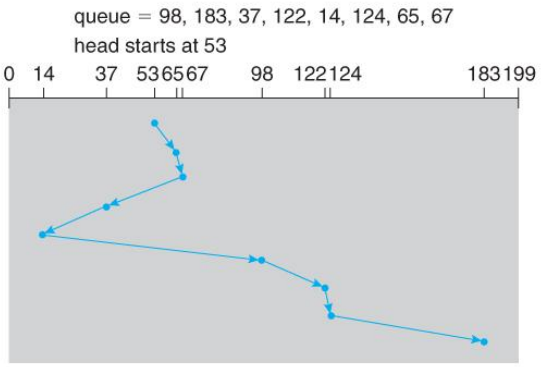
First-Come First-Serve is simple and intrinsically fair, but not very efficient. Consider in the following sequence the wild swing from cylinder 122 to 14 and then back to 124:



SSTF Scheduling

Shortest Seek Time First scheduling is more efficient, but may lead to starvation if a constant stream of requests arrives for the same general area of the disk.

SSTF reduces the total head movement to 236 cylinders, down from 640 required for the same set of requests under FCFS. Note, however that the distance could be reduced still further to 208 by starting with 37 and then 14 first before processing the rest of the requests.



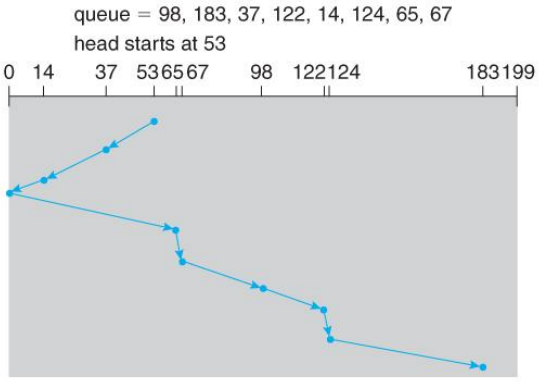
SCAN Scheduling

- The SCAN algorithm, a.k.a. the elevator algorithm moves back and forth from one end of the disk to the other, similarly to an elevator processing requests in a tall building.

- Under the SCAN algorithm, If a request arrives just ahead of the moving head then it will be processed right away, but if it

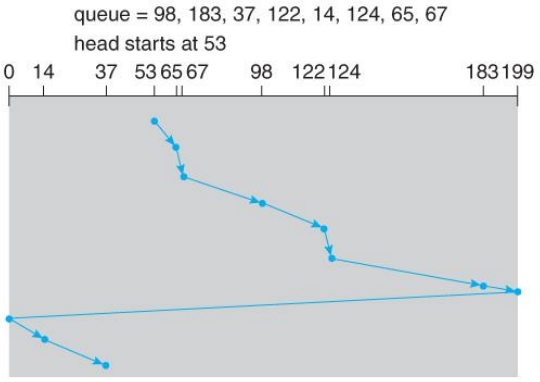
arrives just after the head has passed, then it will have to wait for the head to pass going the other way on the return trip. This leads to a fairly wide variation in access times which can be improved upon.

- Consider, for example, when the head reaches the high end of the disk: Requests with high cylinder numbers just missed the passing head, which means they are all fairly recent requests, whereas requests with low numbers may have been waiting for a much longer time. Making the return scan from high to low then ends up accessing recent requests first and making older requests wait that much longer.



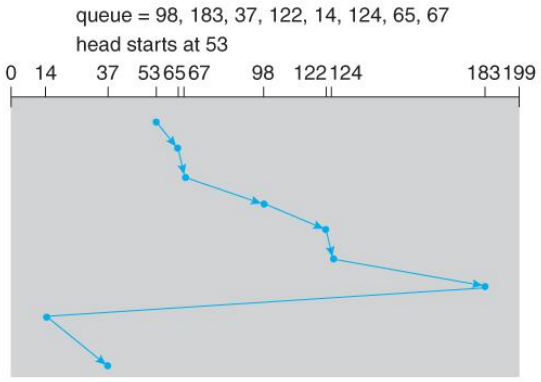
C-SCAN Scheduling

The Circular-SCAN algorithm improves upon SCAN by treating all requests in a circular queue fashion - Once the head reaches the end of the disk, it returns to the other end without processing any requests, and then starts again from the beginning of the disk



LOOK Scheduling

LOOK scheduling improves upon SCAN by looking ahead at the queue of pending requests, and not moving the heads any farther towards the end of the disk than is necessary. The following diagram illustrates the circular form of LOOK:



Network-Attached Storage

- Network attached storage connects storage devices to computers using a remote procedure call, RPC, interface, typically with something like NFS filesystem mounts. This is convenient for allowing several computers in a group common access and naming conventions for shared storage.

- NAS can be implemented using SCSI cabling, or ISCSI uses Internet protocols and standard network connections, allowing long-distance remote access to shared files.

- NAS allows computers to easily share data storage, but tends to be less efficient than standard host-attached storage.

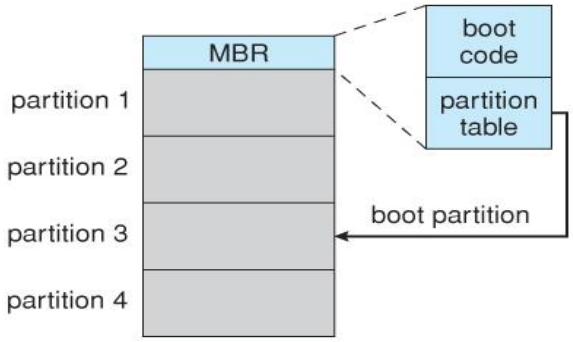
Boot Block

- Computer ROM contains a bootstrap program (OS independent) with just enough code to find the first sector on the first hard drive on the first controller, load that sector into memory, and transfer control over to it. (The ROM bootstrap program may look in floppy and/or CD drives before accessing the hard drive, and is smart enough to recognize whether it has found valid boot code or not.)

- The first sector on the hard drive is known as the Master Boot Record, MBR, and contains a very small amount of code in addition to the partition table. The partition

table documents how the disk is partitioned into logical disks, and indicates specifically which partition is the active or boot partition.

- The boot program then looks to the active partition to find an operating system, possibly loading up a slightly larger / more advanced boot program along the way.
- In a dual-boot (or larger multi-boot) system, the user may be given a choice of which operating system to boot, with a default action to be taken in the event of no response within some time frame.
- Once the kernel is found by the boot program, it is loaded into memory and then control is transferred over to the OS. The kernel will normally continue the boot process by initializing all important kernel data structures, launching important system services (e.g. network daemons, sched, init, etc.), and finally providing one or more login prompts. Boot options at this stage may include single-user a.k.a. maintenance or safe modes, in which very few system services are started - These modes are designed for system administrators to repair problems or otherwise maintain the system.



STRENGTHS AND WEAKNESSES OF THE OPERATING SYSTEM

STRENGTHS:

1. Open Source:
   - This implies that anyone can see, alter, and share its source code without restriction. This encourages teamwork and makes quick progress and innovation possible.
2. Customization:
   - You can customize Linux a lot. Users can choose from a wide range of distributions (distros) according to their interests and use cases. Moreover, users may modify nearly all the system's features, allowing them to customize it to suit their own needs.
3. Stability:
   - Linux is renowned for being dependable and stable, particularly in server contexts. Its durability and uptime are enhanced by its strong memory management, process separation, and error recovery features.
4. Security:
   - Because of its architecture, which is based on Unix, Linux is intrinsically more secure than some other operating systems. Features like user privilege separation, file permissions, and SELinux (Security-Enhanced Linux) are all built-in. Being open source also makes it possible for the community to swiftly find and fix security flaws.
5. Performance:
   - Performance under Linux is usually very good, especially on server systems. Because of its effective resource management, reduced overhead, and superior networking capabilities, it is a

good choice for server workloads and high-demand applications.

6. Wide Hardware Support:

- Linux is compatible with a wide range of hardware architectures, therefore it may be used with embedded systems and supercomputers. Furthermore, a lot of hardware vendors ensure compatibility with current hardware by offering drivers and support for Linux.

WEAKNESSES

1. Complexity for Beginners:

- The learning curve for Linux can be quite high, particularly for people coming from Windows or macOS. For novices, learning how to use the system setup and command-line interface (CLI) might be daunting and take some time.

2. Software Compatibility:

- Despite the abundance of open-source software available for Linux, some proprietary applications that are often used by individuals or industries may not work with Linux. While there are workarounds and alternatives, switching to Linux may provide challenges for certain individuals.

3. Gaming support:

- Even while Linux gaming has advanced considerably in recent years, performance and game compatibility remain inferior to Windows. Not all games are completely supported on Linux, even though programs like Wine and Proton allow you to run many popular Windows games on the operating system.

4. Driver Support for Certain Hardware:

- Although a large variety of hardware is supported by Linux, some specialized or proprietary hardware might not have official Linux drivers or might only function partially in comparison to Windows. Those who need certain hardware may find this concerning.

5. Enterprise Support and Software:

- Although Red Hat corporate Linux (RHEL) and Windows Server are commercial operating systems with extensive support, certifications, and corporate software compatibility, some firms may prefer them despite the widespread use of Linux in enterprise settings.

REFERENCES

1. *Andrea Stevanato, Daniel Bristot De Olivei, "An Evaluation of Adaptive Partitioning of Real-Time Workloads on Linux" 09 July 2021, IEEE references.*

1. *W. Stallings, "Operating Systems: Internals and Design Principles".*

2. *"Classical Problems of Synchronization with Semaphore Solution",* GeeksforGeeks,

https://www.geeksforgeeks.org/classical-problems-of-synchronization-with-semaphore-solution/.

3. *Andrea Stevanato, Daniel Bristot De Olivei, "An Evaluation of Adaptive Partitioning of Real-Time Workloads on Linux" 09 July 2021, IEEE references.*

https://www.geeksforgeeks.org/introduction-of-deadlock-in-operating-system/

Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 7

2. *S. Zivanov, "phoenixnap," 16 03 2023. [Online]. Available: https://phoenixnap.com/.*

3. *Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 7*

[1] S. Zivanov, "phoenixnap," 16 03 2023. [Online]. Available: https://phoenixnap.com/.

[2] S. Zivanov, "phoenixNAP," [Online]. Available: https://phoenixnap.com/.

[3] S. Zivanov, "phoenixNAP," [Online]. Available: https://phoenixnap.com/.