Unit III

CAOS

In computer architecture and operating systems, a critical region denotes a section of code or a set of instructions that accesses shared resources or variables. It requires exclusive access to maintain data integrity and prevent race conditions. Critical regions are typically guarded by synchronization mechanisms such as locks, semaphores, or mutexes to ensure mutual exclusion.

Key characteristics of critical regions:

1. **Atomic Execution:** The instructions within a critical region must execute atomically, without interruption from other threads or processes.
2. **Mutual Exclusion:** Only one thread or process can access the critical region at any given time. Concurrent access can lead to data corruption or inconsistency.
3. **Synchronization:** Synchronization primitives like locks or semaphores are used to control access to critical regions, ensuring that only one thread or process can enter at a time.
4. **Data Integrity:** Critical regions are vital for maintaining data integrity in shared memory environments, preventing issues such as race conditions or deadlock.

Effective management of critical regions is essential for the correct functioning of multi-threaded and multi-process systems, ensuring safe and consistent access to shared resources.

Synchronization:

The synchronization of chaos refers to the phenomenon where chaotic systems, which exhibit seemingly random and unpredictable behavior, can exhibit a certain degree of order or correlation when coupled or driven by external forces. This synchronization has been observed in various fields, including physics, biology, and engineering, and it has implications for understanding complex systems and their behavior.

In chaotic systems, small changes in initial conditions can lead to drastically different outcomes over time, making them inherently unpredictable. However, when two or more chaotic systems interact, they can sometimes synchronize their chaotic behavior, leading to a correlated motion or pattern.

There are different types of synchronization phenomena observed in chaos, including:

1. **Complete Synchronization**: In this case, the states of two or more chaotic systems become identical or mirror each other over time.
2. **Phase Synchronization**: Here, the phases of the chaotic systems align with each other, even though their amplitudes may differ.
3. **Lag Synchronization**: This occurs when there's a constant time delay between the synchronized chaotic systems.
4. **Generalized Synchronization**: This is a more relaxed form of synchronization where the dynamics of two systems are related, but not necessarily identical.

Semaphore:

Semaphore chaos refers to the phenomenon observed in systems governed by semaphore rules where seemingly chaotic behavior emerges from simple rules and interactions. Semaphores are signaling mechanisms used in computer science for process synchronization and communication between multiple threads or processes.

In semaphore chaos, the interactions between different processes or threads, each governed by semaphore rules, can lead to complex and unpredictable behavior. Despite the deterministic nature of semaphore rules, the collective interactions of multiple processes can result in chaotic patterns, making the system difficult to predict or control.

Synchronization Problems

These problems are used for testing nearly every newly proposed synchronization scheme. The following problems of synchronization are considered as classical problems:

```
1. Bounded-buffer (or Producer-Consumer) Problem,
2. Dining-Philosophers Problem,
3. Readers and Writers Problem,
```

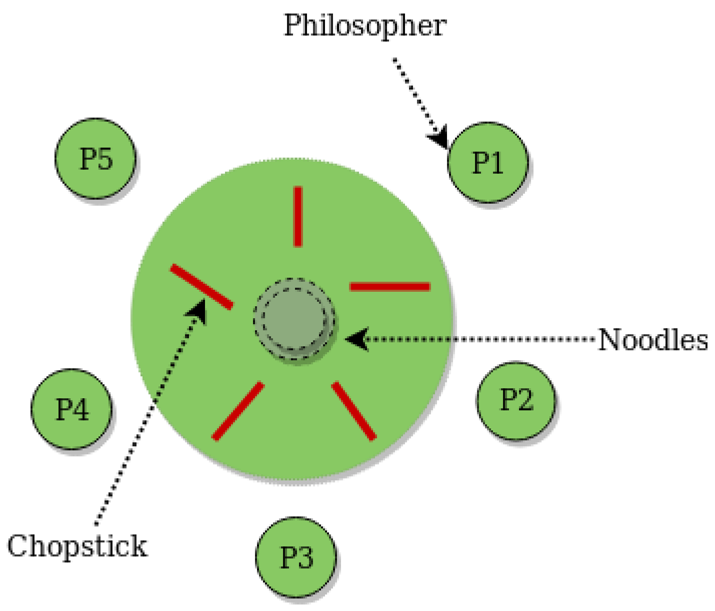
classical problems of synchronization as examples of a large class of concurrency-control problems. In our solutions to the problems, use semaphores for synchronization

Bounded-Buffer (or Producer-Consumer) Problem

The Bounded Buffer problem is also called the producer-consumer problem. This problem is generalized in terms of the Producer-Consumer problem. The solution to this problem is, to create two counting semaphores “full” and “empty” to keep track of the current number of full and empty buffers respectively. Producers produce a product .

Dining-Philosophers Problem

The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pickup the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both. This problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.



Readers and Writers Problem

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as readers and to the latter as writers. Precisely in OS we call this situation as the readers-writers problem. Problem parameters:

- One set of data is shared among a number of processes.
- Once a writer is ready, it performs its write. Only one writer may write at a time.
- If a process is writing, no other process can read it.
- If at least one reader is reading, no other process can write.
- Readers may not write and only read.

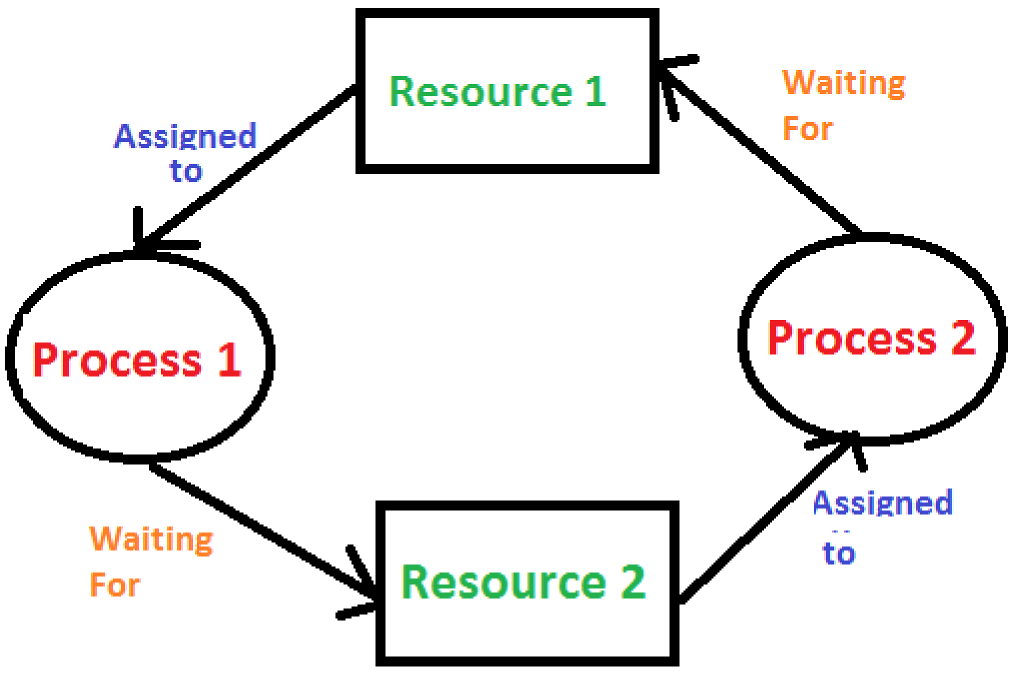
Dead Lock preventation,handling and recovery

Deadlock detection and recovery is the process of detecting and resolving deadlocks in an operating system

There are two main approaches to deadlock detection and recovery:

1. **Prevention**: The operating system takes steps to prevent deadlocks from occurring by ensuring that the system is always in a safe state, where deadlocks cannot occur. This is achieved through resource allocation algorithms such as the Banker's Algorithm.
2. **Detection and Recovery:** If deadlocks do occur, the operating system must detect and resolve them. Deadlock detection algorithms, such as the Wait-For Graph, are used to identify deadlocks, and recovery algorithms, such as the Rollback and Abort algorithm, are used to resolve them. The recovery algorithm releases the resources held by one or more processes, allowing the system to continue to make progress.

3. **If resources have a single instance –**
In this case for Deadlock detection, we can run an algorithm to check for the cycle in the Resource Allocation Graph. The presence of a cycle in the graph is a sufficient condition for deadlock.



Advantages of Deadlock Detection and Recovery in Operating Systems:

1. **Improved System Stability:** Deadlocks can cause system-wide stalls, and detecting and resolving deadlocks can help to improve the stability of the system.
2. **Better Resource Utilization:** By detecting and resolving deadlocks, the operating system can ensure that resources are efficiently utilized and that the system remains responsive to user requests.
3. **Better System Design**: Deadlock detection and recovery algorithms can provide insight into the behavior of the system and the relationships between processes and resources, helping to inform and improve the design of the system.
4. **Performance Overhead:** Deadlock detection and recovery algorithms can introduce a significant overhead in terms of performance, as the system must regularly check for deadlocks and take appropriate action to resolve them.
5. **Complexity:** Deadlock detection and recovery algorithms can be complex to implement, especially if they use advanced techniques such as the Resource Allocation Graph or Timestamping.
6. **False Positives and Negatives:** Deadlock detection algorithms are not perfect and may produce false positives or negatives, indicating the presence of deadlocks when they do not exist or failing to detect deadlocks that do exist.
7. **Risk of Data Loss**: In some cases, recovery algorithms may require rolling back the state of one or more processes, leading to data loss or corruption.

Direct Memory Access (DMA):

What is a DMA Controller?

The term DMA stands for direct memory access. The hardware device used for direct memory access is called the DMA controller.

DMA Controller Diagram in Computer Architecture

DMA controller provides an interface between the bus and the input-output devices. Although it transfers data without intervention of processor, it is controlled by the processor.

a) **Burst Mode**: In this mode DMA handover the buses to CPU only after completion of whole data transfer. Meanwhile, if the CPU requires the bus it has to stay ideal and wait for data transfer.
b) **Cycle Stealing Mode**: In this mode, DMA gives control of buses to CPU after transfer of every byte. It continuously issues a request for bus control, makes the transfer of one byte and returns the bus. By this CPU doesn't have to wait for a long time if it needs a bus for higher priority task.
c) **Transparent Mode:** Here, DMA transfers data only when CPU is executing the instruction which does not require the use of buses.

UNIT-4

Free and Open-Source

Linux is completely free of cost, and expenses are never a hindrance to using it as an operating system.

Linux is open-source. This means that modification of code, analysis of codes, redistribution of codes

Extremely Flexible

Linux has incorporated itself into embedded products like watches, digital equipment and supercomputing servers.

Lightweight Infrastructure

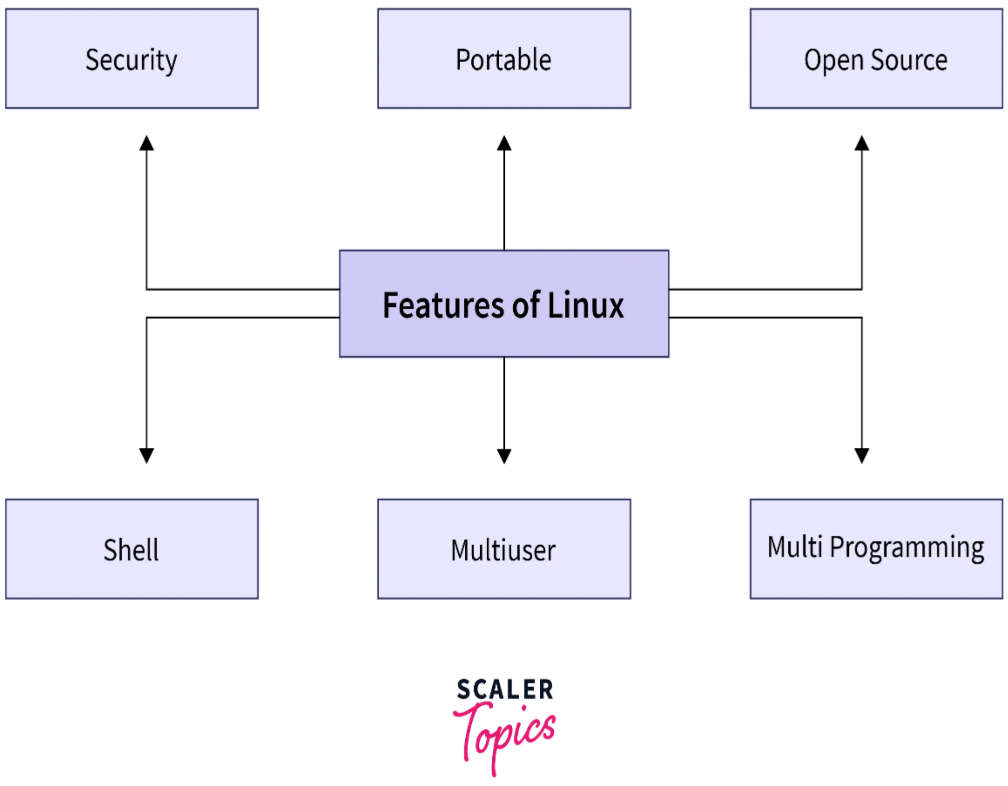
Linux consumes lesser storage space, and its installation requires around 4GB to 8GB of disk space.

Graphical User Interface (GUI)

Even though *Linux* works on using the command line interface but it can be converted to be used like windows having a *Graphical user interface*.

End-to-end encryption

Linux allows end-to-end encryption while accessing data thus storing public keys in the server. All data is password protected

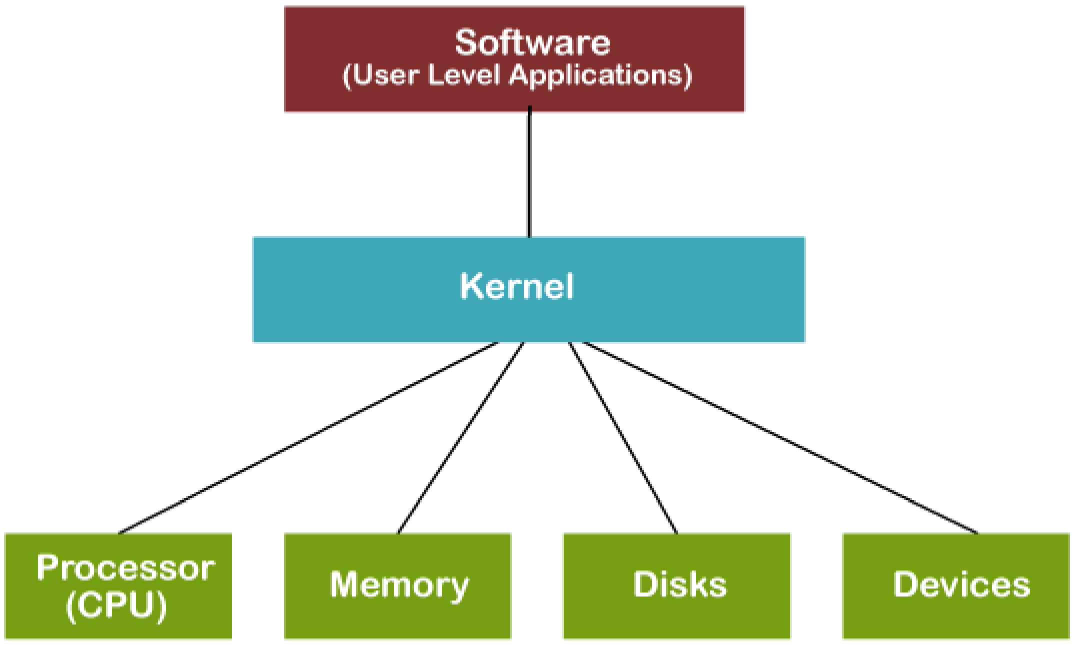


Kernel Info:

Functions of a Kernel

Functions of a Kernel

Functions of a Kernel

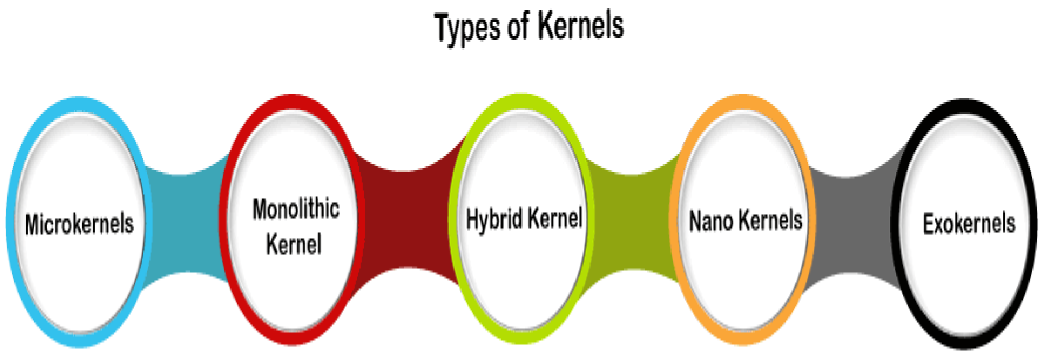


Functions of a Kernel

Memory Management

Resource Management

Accessing Computer Resources



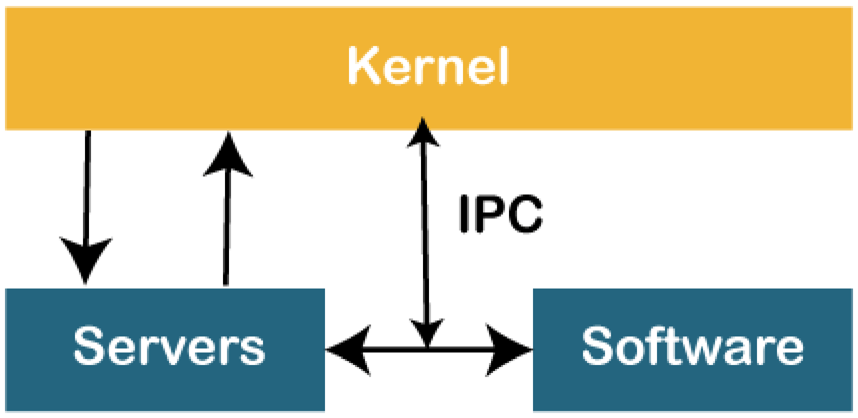
The execution of processes is also faster than other kernel types as it does not use separate user and kernel space.

Examples of Monolithic Kernels are **Unix, Linux, Open VMS, XTS-400, etc**

- As it is a single piece of software hence, it's both sources and compiled forms are smaller.

Disadvantages:

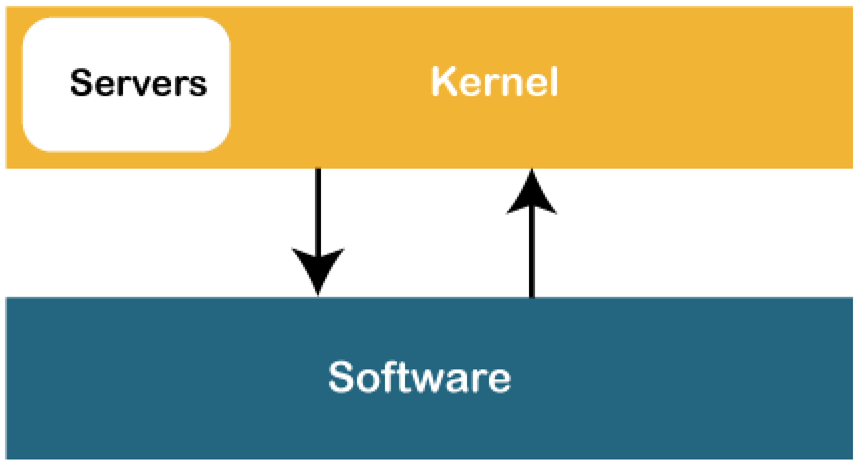
- If any service generates any error, it may crash down the whole system.
- These kernels are not portable, which means for each new architecture, they must be rewritten.
- Large in size and hence become difficult to manage.



Examples of Microkernel are **L4, AmigaOS, Minix, K42**, etc.

Advantages

- Microkernels can be managed easily.
- A new service can be easily added without modifying the whole OS.



Advantages:

- There is no requirement for a reboot for testing.
- Third-party technology can be integrated rapidly.

Disadvantages:

- There is a possibility of more bugs with more interfaces to pass through.

Nanokernel

As the name suggests, ***in Nanokernel, the complete code of the kernel is very small, which means the code executing in the privileged mode of the hardware is very small***.

Examples of Nanokernel **are EROS etc.**

Advantages

- It provides hardware abstractions even with a very small size.

5. Exokernel

Exokernel is still developing and is the experimental approach for designing OS.

Advantages:

- The exokernel-based system can incorporate multiple library operating systems. Each library exports a different API, such as one can be used for high-level UI development, and the other can be used for real-time control**.**

Disadvantages:

- The design of the exokernel is very complex.

There are **5** basic operators in bash/shell scripting:

- Arithmetic Operators
- Relational Operators
- Boolean Operators
- Bitwise Operators
- File Test Operators

1. Arithmetic Operators: These operators are used to perform normal arithmetics/mathematical operations. There are 7 arithmetic operators:

- **Addition (+)**: Binary operation used to add two operands.
- **Subtraction (-)**: Binary operation used to subtract two operands.
- **Multiplication (*)**: Binary operation used to multiply two operands.

- **Division (/)**: Binary operation used to divide two operands.
- **Modulus (%)**: Binary operation used to find remainder of two operands.
- **Increment Operator (++)**: Unary operator used to increase the value of operand by one.
- **'==' Operator**: Double equal to operator compares the two operands. Its returns true is they are equal otherwise returns false.
- **'!=' Operator**: Not Equal to operator return true if the two operands are not equal otherwise it returns false.
- **'<' Operator**: Less than operator returns true if first operand is less than second operand otherwise returns false.
- **'<=' Operator**: Less than or equal to operator returns true if first operand is less than or equal to second operand otherwise returns false
- **'>' Operator**: Greater than operator return true if the first operand is greater than the second operand otherwise return false.
- **'>=' Operator**: Greater than or equal to operator returns true if first operand is greater than or equal to second operand otherwise returns false

Logical Operators : They are also known as boolean operators. These are used to perform logical operations. They are of 3 types:

- **Logical AND (&&)**: This is a binary operator, which returns true if both the operands are true otherwise returns false.
- **Logical OR (||)**: This is a binary operator, which returns true if either of the operands is true or if both the operands are true. It returns false only if both operands are false.
- **Not Equal to (!)**: This is a unary operator which returns true if the operand is false and returns false if the operand is true.

Bitwise Operators: A bitwise operator is an operator used to perform bitwise operations on bit patterns. They are of 6 types:

- **Bitwise And (&)**: Bitwise & operator performs binary AND operation bit by bit on the operands.
- **Bitwise OR (|)**: Bitwise | operator performs binary OR operation bit by bit on the operands.

- **Bitwise XOR (^)**: Bitwise ^ operator performs binary XOR operation bit by bit on the operands.
- **Bitwise complement (~)**: Bitwise ~ operator performs binary NOT operation bit by bit on the operand.
- **Left Shift (<<)**: This operator shifts the bits of the left operand to left by number of times specified by right operand.
- **Right Shift (>>)**: This operator shifts the bits of the left operand to right by number of times specified by right operand.