OS Module 3

Inter Process Communication

CONCURRENCY IN OS

Concurrency in an operating system refers to the ability to execute multiple processes or threads simultaneously, improving resource utilization and system efficiency. It allows several tasks to be in progress at the same time, either by running on separate processors or through context switching on a single processor. Concurrency is essential in modern OS design to handle multitasking, increase system responsiveness, and optimize performance for users and applications.

There are several motivations for allowing concurrent execution:

- **Physical resource Sharing:** Multiuser environment since hardware resources are limited
- **Logical resource Sharing:** Shared file (same piece of information)
- **Computation Speedup:** Parallel execution
- **Modularity:** Divide system functions into separation processes

Relationship Between Processes of Operating System

The Processes executing in the operating system is one of the following two types:

- Independent Processes
- Cooperating Processes

Independent Processes

Its state is not shared with any other process.

- The result of execution depends only on the input state.
- The result of the execution will always be the same for the same input.
- The termination of the independent process will not terminate any other.

Cooperating Processes

Its state is shared along other processes.

- The result of the execution depends on relative execution sequence and cannot be predicted in advanced(Non-deterministic).
- The result of the execution will not always be the same for the same input.
- The termination of the cooperating process may affect other process.

Principles of Concurrency

Both interleaved and overlapped processes can be viewed as examples of concurrent processes, they both present the same problems.
The relative speed of execution cannot be predicted. It depends on the following:

- The activities of other processes

- The way operating system handles interrupts
- The scheduling policies of the operating system

Problems in Concurrency

- **Sharing global resources:** Sharing of global resources safely is difficult. If two processes both make use of a global variable and both perform read and write on that variable, then the order in which various read and write are executed is critical.
- **Optimal allocation of resources:** It is difficult for the operating system to manage the allocation of resources optimally.
- **Locating programming errors:** It is very difficult to locate a programming error because reports are usually not reproducible.
- **Locking the channel:** It may be inefficient for the operating system to simply lock the channel and prevents its use by other processes.

CRITICAL SECTION

When more than one processes try to access the same code segment that segment is known as the critical section. The critical section contains shared variables or resources that need to be synchronized to maintain the consistency of data variables.

In simple terms, a critical section is a group of instructions/statements or regions of code that need to be executed atomically such as accessing a resource (file, input or output port, global data, etc..) In concurrent programming, if one thread tries to change the value of shared data at the same time as another thread tries to read the value (i.e., data race across threads), the result is unpredictable. The access to such shared variables (shared memory, shared files, shared port, etc.) is to be synchronized.

Few programming languages have built-in support for synchronization. It is critical to understand the importance of race conditions while writing kernel-mode programming (a device driver, kernel thread, etc.) since the programmer can directly access and modify kernel data structures

Although there are some properties that should be followed if any code in the critical section

1. **Mutual Exclusion:** If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded Waiting:** There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Two general approaches are used to handle critical sections:

1. **Preemptive kernels**: A preemptive kernel allows a process to be preempted while it is running in kernel mode.

2. **Non preemptive kernels**: A non preemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exists in kernel mode, blocks, or voluntarily yields control of the CPU. A non preemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time.

Critical Section Problem

The use of critical sections in a program can cause a number of issues, including:

Deadlock: When two or more threads or processes wait for each other to release a critical section, it can result in a deadlock situation in which none of the threads or processes can move. Deadlocks can be difficult to detect and resolve, and they can have a significant impact on a program's performance and reliability.

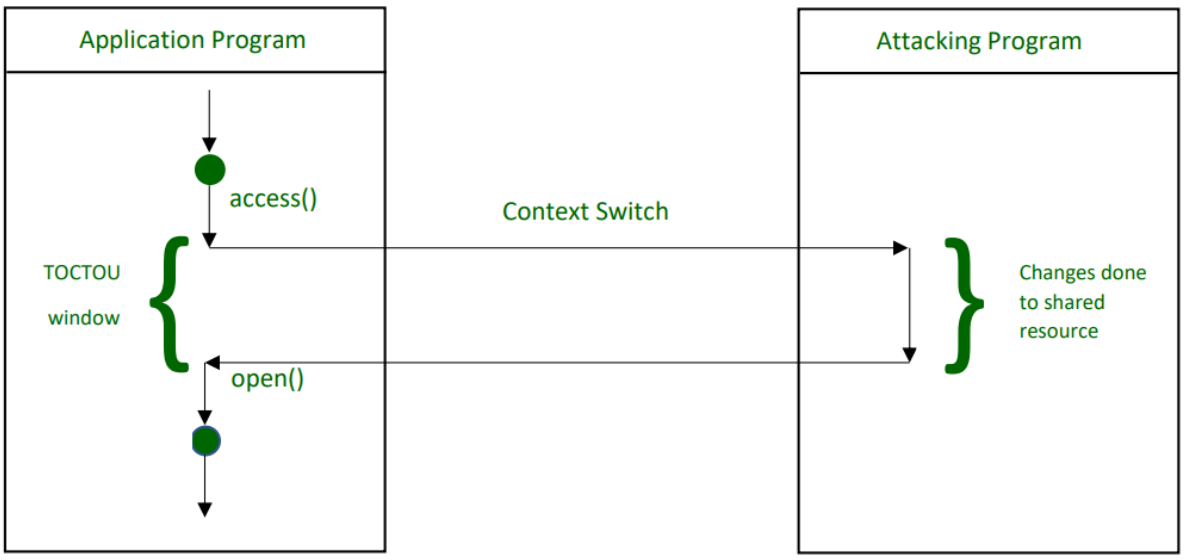
Starvation: When a thread or process is repeatedly prevented from entering a critical section, it can result in starvation, in which the thread or process is unable to progress. This can happen if the critical section is held for an unusually long period of time, or if a high-priority thread or process is always given priority when entering the critical section.

Overhead: When using critical sections, threads or processes must acquire and release locks or semaphores, which can take time and resources. This may reduce the program's overall performance.

RACE CONDITION

A race condition is a situation that may occur inside a critical section. This happens when the result of multiple thread execution in a critical section differs according to the order in which the threads execute. Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. Also, proper thread synchronization using locks or atomic variables can prevent race conditions.

This is a major security vulnerability [CWE-362], and by manipulating the timing of actions anomalous results might appear. This vulnerability arises during a TOCTOU (time-of-check, time-of-use) window.



Flow of File Access during it's TOCTOU Window

Flow of File Access during its TOCTOU Window So, what if we lock the file during this TOCTOU window itself?

1. **General Misconception –** A trivial cure to this vulnerability could be locking the file itself during this check-and-use window because then no other process can use the file during the time window. Seems easy, then why isn't this practical? Why can't we use this approach to solve the race condition problem? The answer is simply that such a vulnerability could not be prevented by just locking the file.

2. **Problems while locking the file –** A file is locked out for other processes only if it is already in the open state. This process is called the check-and-open process and during this time it is impossible to lock a file. Any locks created can be ignored by the attacking or the malicious process. What happens is that the call to Open() does not block an attack on a locked file. When the file is available for a check-and-open process, the file is open to any access/ change. So it's impossible to lock a file at this point in time. This makes any kind of lock virtually non-existent to the malicious processes. Internally it is using the sleep_time which doubles at every attempt. More commonly this is referred to as a spinlock or the busy form of waiting. Also, there is always a possibility of the file getting locked indefinitely i.e. danger of getting stuck in a deadlock.

3. **What would happen even if we were somehow able to lock the file?** Let's try to lock the file and see what could be the possible drawbacks. The most common locking mechanism that is available is atomic file locking. It is done using a lock file to create a unique file on the same filesystem. We make use of link() to make a link to the lock file for any kind of access to the file.

   - If link() returns 0, the lock is successful.

The most common fix available is to store the PID of the application in the lock file, which is checked against the active PID at that time. Then again a flaw with this fix is that PID may have been reused.

4. **Actual Solutions –** A better solution is to rather than creating locks on the file as a whole, lock the parts of the file to different processes. **Example –** When a process wants to write into a file, it first asks the kernel to lock that file or a part of it. As long as the process keeps the lock, no other process can ask to lock the same part of the file. Hence you could see that the issue with concurrency is getting resolved like this. In the same way, a process asks for locking before reading the content of a file, which ensures no changes will be made as long as the lock is kept. Differentiating this different kind of locks is done by the system itself. The system has the capability to distinguish between the locks required for file reading and those required for file writing. This kind of locking system is achieved by the flock() system call. Flock() call can have different values :

   - LOCK_SH (lock for reading)
   - LOCK_EX (for writing)
   - LOCK_UN (release of the lock)

Pipes in OS

A **pipe** is a communication channel that allows the output of one process to be used as input for another process. Pipes are often used in UNIX/Linux-based systems for redirecting output from one command or process to another. Pipes follow the **producer-consumer model**, where one process produces data and another consumes it.

Types of Pipes

1. **Anonymous Pipes (Unidirectional Pipes)**
    - These are the simplest form of pipes and are commonly used to connect related processes, such as a parent and child process.
    - They are **unidirectional**, meaning data flows in one direction (from one process to another).
2. **Named Pipes (FIFO)**

- These pipes are more advanced and allow communication between unrelated processes (not parent-child).
- A **named pipe** is a special file that exists on the filesystem and can be used for bidirectional communication.
- Data in these pipes is handled **First In, First Out (FIFO)**.
- Unlike anonymous pipes, the pipe has a name and persists beyond the process lifetime, making it suitable for processes that start at different times.
- Named pipes are created using the mkfifo command or the mknod function.

3. **Half-Duplex Pipes**
   A form of unidirectional communication where data can only flow in one direction at a time. Typically associated with anonymous pipes.
4. **Full-Duplex Pipes**
   These pipes allow bidirectional communication, meaning data can flow both ways (in and out simultaneously).
   Often implemented using **sockets** or **message queues** in modern IPC mechanisms.

Pipe Operations

- **Reading from a pipe:** The receiving process reads the data from the pipe, typically done using the read() system call in C, Python, or other programming languages.
- **Writing to a pipe:** The producer process writes data to the pipe using the write() system call.

Use Cases of Pipes

- **Command chaining in shell scripts**: As shown in the example with ls | grep.
- **Data streaming between processes**: One process generates data, while another consumes it.
- **IPC (Inter-Process Communication)**: In complex applications where different processes need to exchange information efficiently.

MUTUAL EXCLUSION

Mutual Exclusion is a property of process synchronization that states that "**no two processes can exist in the critical section at any given point of time**". The term was first coined by **Dijkstra**. Any process synchronization technique being used must satisfy the property of mutual exclusion, without which it would not be possible to get rid of a race condition.
The need for mutual exclusion comes with concurrency. There are several kinds of concurrent execution:

- Interrupt handlers
- Interleaved, preemptively scheduled processes/threads
- Multiprocessor clusters, with shared memory
- Distributed systems

Mutual exclusion methods are used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical sections.
The requirement of mutual exclusion is that when process P1 is accessing a shared resource R1, another process should not be able to access resource R1 until process P1 has finished its operation with resource R1.
Examples of such resources include files, I/O devices such as printers, and shared data structures.

Conditions Required for Mutual Exclusion
According to the following four criteria, mutual exclusion is applicable:

- When using shared resources, it is important to ensure mutual exclusion between various processes. There cannot be two processes running simultaneously in either of their critical sections.
- It is not advisable to make assumptions about the relative speeds of the unstable processes.
- For access to the critical section, a process that is outside of it must not obstruct another process.
- Its critical section must be accessible by multiple processes in a finite amount of time; multiple processes should never be kept waiting in an infinite loop.

Approaches To Implementing Mutual Exclusion

- **Software Method:** Leave the responsibility to the processes themselves. These methods are usually highly error-prone and carry high overheads.
- **Hardware Method:** Special-purpose machine instructions are used for accessing shared resources. This method is faster but cannot provide a complete solution. Hardware solutions cannot give guarantee the absence of deadlock and starvation.
- **Programming Language Method:** Provide support through the operating system or through the programming language.

Requirements of Mutual Exclusion

- At any time, only one process is allowed to enter its critical section.
- The solution is implemented purely in software on a machine.
- A process remains inside its critical section for a bounded time only.
- No assumption can be made about the relative speeds of asynchronous concurrent processes.
- A process cannot prevent any other process from entering into a critical section.
- A process must not be indefinitely postponed from entering its critical section.

SEMAPHORES

Semaphores are just normal variables used to coordinate the activities of multiple processes in a computer system. They are used to enforce mutual exclusion, avoid race conditions, and implement synchronization between processes.

The process of using Semaphores provides two operations: wait (P) and signal (V). The wait operation decrements the value of the semaphore, and the signal operation increments the value of the semaphore. When the value of the semaphore is zero, any process that performs a wait operation will be blocked until another process performs a signal operation.

When a process performs a wait operation on a semaphore, the operation checks whether the value of the semaphore is >0. If so, it decrements the value of the semaphore and lets the process continue its execution; otherwise, it blocks the process on the semaphore. A signal operation on a semaphore activates a process blocked on the semaphore if any, or increments the value of the semaphore by 1. Due to these semantics, semaphores are also called counting semaphores. The initial value of a semaphore determines how many processes can get past the wait operation.

Types of Semaphore

- **Binary Semaphore** – This is similar to mutex lock but not the same thing. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.
- **Counting Semaphore** – Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

MUTEX

Mutex is a specific kind of binary semaphore that is used to provide a locking mechanism. It stands for Mutual Exclusion Object. Mutex is mainly used to provide mutual exclusion to a specific portion of the code so that the process can execute and work with a particular section of the code at a particular time.

Mutex uses a priority inheritance mechanism to avoid priority inversion issues. The priority inheritance mechanism keeps higher-priority processes in the blocked state for the minimum possible time. However, this cannot avoid the priority inversion problem, but it can reduce its effect up to an extent.

Advantages of Mutex

1. No race condition arises, as only one process is in the critical section at a time.
2. Data remains consistent and it helps in maintaining integrity.
3. It's a simple locking mechanism that can be obtained by a process before entering into a critical section and released while leaving the critical section.

Disadvantages of Mutex

1. If after entering into the critical section, the thread sleeps or gets preempted by a high-priority process, no other thread can enter into the critical section. This can lead to starvation.
2. When the previous thread leaves the critical section, then only other processes can enter into it, there is no other mechanism to lock or unlock the critical section.
3. Implementation of mutex can lead to busy waiting that leads to the wastage of the CPU cycle.

Mutex	**Semaphore**
A mutex is an object.	A semaphore is an integer.
Mutex works upon the locking mechanism.	Semaphore uses signaling mechanism
Operations on mutex: • Lock • Unlock	Operation on semaphore: • Wait • Signal
Mutex doesn't have any subtypes.	Semaphore is of two types: • Counting Semaphore • Binary Semaphore
A mutex can only be modified by the process that is requesting or releasing a resource.	Semaphore work with two atomic operations (Wait, signal) which can modify it.
If the mutex is locked then the process needs to wait in the process queue, and mutex can only be accessed once the lock is released.	If the process needs a resource, and no resource is free. So, the process needs to perform a wait operation until the semaphore value is greater than zero.

MONITORS

Monitors are a higher-level synchronization construct that simplifies process synchronization by providing a high-level abstraction for data access and synchronization. Monitors are implemented as programming language constructs, typically in object-oriented languages, and provide mutual exclusion, condition variables, and data encapsulation in a single construct.

1. A monitor is essentially a module that encapsulates a shared resource and provides access to that resource through a set of procedures. The procedures provided by a monitor ensure that only one process can access the shared resource at any given time, and that processes waiting for the resource are suspended until it becomes available.

2. Monitors are used to simplify the implementation of concurrent programs by providing a higher-level abstraction that hides the details of synchronization. Monitors provide a structured way of sharing data and synchronization information, and eliminate the need for complex synchronization primitives such as semaphores and locks.

3. The key advantage of using monitors for process synchronization is that they provide a simple, high-level abstraction that can be used to implement complex concurrent systems. Monitors also ensure that synchronization is encapsulated within the module, making it easier to reason about the correctness of the system.

However, monitors have some limitations. For example, they can be less efficient than lower-level synchronization primitives such as semaphores and locks, as they may involve additional overhead due to their higher-level abstraction. Additionally, monitors may not be suitable for all types of synchronization problems, and in some cases, lower-level primitives may be required for optimal performance.

The monitor is one of the ways to achieve Process synchronization. The monitor is supported by programming languages to achieve mutual exclusion between processes. For example Java Synchronized methods. Java provides wait() and notify() constructs.

1. It is the collection of condition variables and procedures combined together in a special kind of module or a package.

2. The processes running outside the monitor can't access the internal variable of the monitor but can call procedures of the monitor.

3. Only one process at a time can execute code inside monitors.

READERS-WRITERS PROBLEM

The readers-writer problem in operating systems is about managing access to shared data. It allows multiple readers to read data at the same time without issues but ensures that only one writer can write at a time, and no one can read while writing is happening. This helps prevent data corruption and ensures smooth operation in multi-user systems. Probably the most fundamental problem in concurrent programming is to provide safe access to shared resources. A classic problem used to illustrate this issue is Readers-Writers. It is a significant problem for showing how data structures might be synchronized such that consistency is guaranteed and efficiency is ensured. The Readers-Writers problem refers specifically to situations where a number of processes or threads may possibly have access to some common resource, like a database or a file. It also gives rise to the need for good synchronization mechanisms so as not to bias the requirement of readers against that of writers in access to the resource, considering the integrity of data.

What is The Readers-Writers Problem?

The Readers-Writers Problem is a classic synchronization issue in operating systems that involves managing access to shared data by multiple threads or processes. The problem addresses the scenario where:

- **Readers**: Multiple readers can access the shared data simultaneously without causing any issues because they are only reading and not modifying the data.
- **Writers**: Only one writer can access the shared data at a time to ensure data integrity, as writers modify the data, and concurrent modifications could lead to data corruption or inconsistencies.

Challenges of the Reader-Writer Problem

The challenge now becomes how to create a synchronization scheme such that the following is supported:

- **Multiple Readers**: A number of readers may access simultaneously if no writer is presently writing.
- **Exclusion for Writers**: If one writer is writing, no other reader or writer may access the common resource.

Solution of the Reader-Writer Problem

There are two fundamental solutions to the Readers-Writers problem:

- **Readers Preference:** In this solution, readers are given preference over writers. That means that till readers are reading, writers will have to wait. The Writers can access the resource only when no reader is accessing it.
- **Writer's Preference:** Preference is given to the writers. It simply means that, after arrival, the writers can go ahead with their operations; though perhaps there are readers currently accessing the resource.

Focus on the solution Readers Preference in this paper. The purpose of the Readers Preference solution is to give a higher priority to the readers to decrease the waiting time of the readers and to make the access of resource more effective for readers.

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

Solution When Reader Has The Priority Over Writer

Here priority means, no reader should wait if the share is currently open for reading. There are four types of cases that could happen here.

Case	**Process 1**	**Process 2**	**Allowed/Not Allowed**
Case 1	Writing	Writing	Not Allowed
Case 2	Writing	Reading	Not Allowed
Case 3	Reading	Writing	Not Allowed
Case 4	Reading	Reading	Allowed

Three variables are used: **mutex, wrt, readcnt** to implement a solution.

1. **semaphore** mutex, wrt; // semaphore **mutex** is used to ensure mutual exclusion when **readcnt** is updated i.e. when any reader enters or exits from the critical section, and semaphore **wrt** is used by both readers and writers
2. **int** readcnt; //**readcnt** tells the number of processes performing read in the critical section, initially 0

Functions for Semaphore

Semaphores are synchronization tools used in operating systems to manage access to shared resources by multiple threads or processes. They use simple integer values and two main operations to control access:

- **wait() :** decrements the semaphore value.
- **signal() :** increments the semaphore value.

Writer Process

- Writer requests the entry to critical section.
- If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
- It exits the critical section.

Reader Process

- Reader requests the entry to critical section.
- If allowed:

    - it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **wrt** semaphore to restrict the entry of writers if any reader is inside.
    - It then, signals mutex as any other reader is allowed to enter while others are already reading.
    - After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore “wrt” as now, writer can enter the critical section.
- If not allowed, it keeps on waiting.

PRODUCER CONSUMER PROBLEM

The Producer-Consumer problem is a classic synchronization issue in operating systems. It involves two types of processes: producers, which generate data, and consumers, which process that data. Both share a common buffer. The challenge is to ensure that the producer doesn’t add data to a full buffer and the consumer doesn’t remove data from an empty buffer while avoiding conflicts when accessing the buffer. In this article, we are going to solve this problem by using semaphores.

Producer Consumer Problem Statement

We have a buffer of fixed size. A producer can produce an item and can place it in the buffer. A consumer can pick items and consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, the buffer is the critical section.

To solve this problem, we need two counting semaphores – Full and Empty. “Full” keeps track of some items in the buffer at any given time and “Empty” keeps track of many unoccupied slots.

Solution for Producer

```
do{

//produce an item

wait(empty);
wait(mutex);

//place in buffer

signal(mutex);
signal(full);

}while(true)
```

When producer produces an item then the value of “empty” is reduced by 1 because one slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item and thus the value of “full” is increased by 1. The value of mutex is also increased by 1 because the task of producer has been completed and consumer can access the buffer.

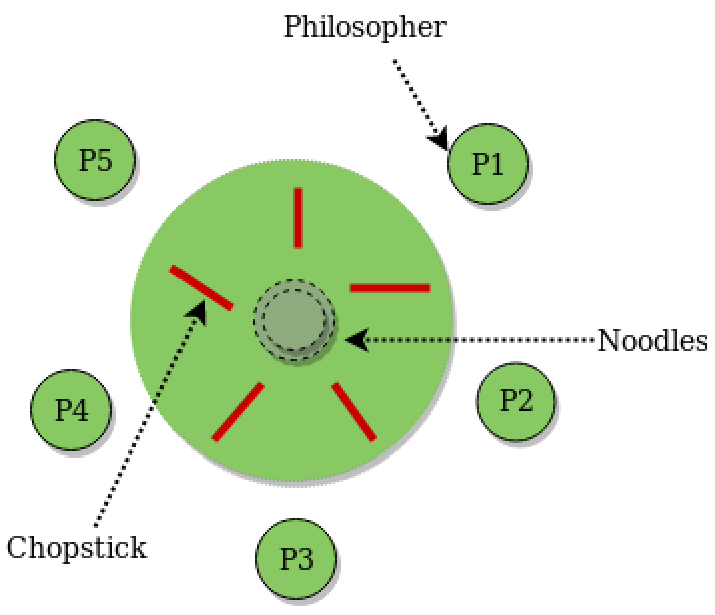
Solution for Consumer

```
do{

wait(full);
wait(mutex);

// consume item from buffer

signal(mutex);
signal(empty);


}while(true)
```

As the consumer is removing an item from buffer, therefore the value of “full” is reduced by 1 and the value is mutex is also reduced so that the producer cannot access the buffer at this moment. Now, the consumer has consumed the item, thus increasing the value of “empty” by 1. The value of mutex is also increased so that producer can access the buffer now.

<u>DINING PHILOSOPHER PROBLEM</u>

The Dining Philosopher Problem states that K philosophers are 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 pick up the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.



Dining Philosopher

Semaphore Solution to Dining Philosopher

There are three states of the philosopher: **THINKING, HUNGRY, and EATING**. Here there are two semaphores: Mutex and a semaphore array for the philosophers. Mutex is used such that no two philosophers may access the pickup or put it down at the same time. The array is used to control the behavior of each philosopher. But, semaphores can result in deadlock due to programming errors.

The steps for the Dining Philosopher Problem solution using semaphores are as follows

1. Initialize the semaphores for each fork to 1 (indicating that they are available).

2. Initialize a binary semaphore (mutex) to 1 to ensure that only one philosopher can attempt to pick up a fork at a time.

3. For each philosopher process, create a separate thread that executes the following code:

- While true:
    - Think for a random amount of time.
    - Acquire the mutex semaphore to ensure that only one philosopher can attempt to pick up a fork at a time.
    - Attempt to acquire the semaphore for the fork to the left.
- If successful, attempt to acquire the semaphore for the fork to the right.
- If both forks are acquired successfully, eat for a random amount of time and then release both semaphores.
- If not successful in acquiring both forks, release the semaphore for the fork to the left (if acquired) and then release the mutex semaphore and go back to thinking.

4. Run the philosopher threads concurrently.

By using semaphores to control access to the forks, the Dining Philosopher Problem can be solved in a way that avoids deadlock and starvation. The use of the mutex semaphore ensures that only one philosopher can attempt to pick up a fork at a time, while the use of the fork semaphores ensures that a philosopher can only eat if both forks are available.

Overall, the Dining Philosopher Problem solution using semaphores is a classic example of how synchronization mechanisms can be used to solve complex synchronization problems in concurrent programming.

DEADLOCK

Deadlock is a situation in computing where two or more processes are unable to proceed because each is waiting for the other to release resources. Key concepts include mutual exclusion, resource holding, circular wait, and no preemption.

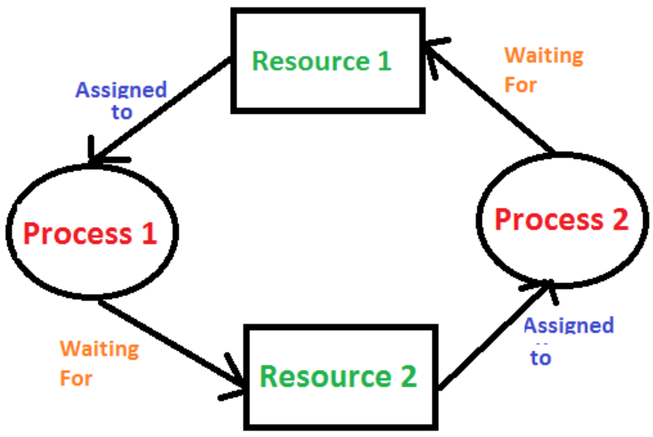
Consider an example when two trains are coming toward each other on the same track and there is only one track, none of the trains can move once they are in front of each other. This is a practical example of deadlock.

How Does Deadlock occur in the Operating System?

Before going into detail about how deadlock occurs in the Operating System, let's first discuss how the Operating System uses the resources present. A process in an operating system uses resources in the following way.

- Requests a resource
- Use the resource
- Releases the resource

A situation occurs in operating systems when there are two or more processes that hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.



Examples of Deadlock

There are several examples of deadlock. Someof them are mentioned below.

1. The system has 2 tape drives. P0 and P1 each hold one tape drive and each needs another one.

2. Semaphores A and B, initialized to 1, P0, and P1 are in deadlock as follows:

- P0 executes wait(A) and preempts.
- P1 executes wait(B).
- Now P0 and P1 enter in deadlock.

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

3. Assume the space is available for allocation of 200K bytes, and the following sequence of events occurs.

P0	P1
Request 80KB;	Request 70KB;
Request 60KB;	Request 80KB;

Deadlock occurs if both processes progress to their second request.

Necessary Conditions for Deadlock in OS

Deadlock can arise if the following four conditions hold simultaneously (Necessary Conditions)

- **Mutual Exclusion:** Two or more resources are non-shareable (Only one process can use at a time).
- **Hold and Wait:** A process is holding at least one resource and waiting for resources.
- **No Preemption:** A resource cannot be taken from a process unless the process releases the resource.
- **Circular Wait:** A set of processes waiting for each other in circular form.

What is Deadlock Detection?

Deadlock detection is a process in computing where the system checks if there are any sets of processes that are stuck waiting for each other indefinitely, preventing them from moving forward. In simple words, deadlock detection is the process of finding out whether any process are stuck in loop or not. There are several algorithms like

- Resource Allocation Graph
- Banker's Algorithm

These algorithms helps in detection of deadlock in Operating System.

What are the Methods For Handling Deadlock?

There are three ways to handle deadlock

- Deadlock Prevention or Avoidance
- Deadlock Recovery
- Deadlock Ignorance

Deadlock Prevention or Avoidance

Deadlock Prevention and Avoidance is the one of the methods for handling deadlock. First, we will discuss Deadlock Prevention, then Deadlock Avoidance.

What is Deadlock Prevention?

In deadlock prevention the aim is to not let full-fill one of the required condition of the deadlock. This can be done by this method:

(i) Mutual Exclusion

We only use the Lock for the non-share-able resources and if the resource is share- able (like read only file) then we not use the locks here. That ensure that in case of share -able resource , multiple process can access it at same time. Problem- Here the problem is that we can only do it in case of share-able resources but in case of no-share-able resources like printer , we have to use Mutual exclusion.

(ii) Hold and Wait

To ensure that Hold and wait never occurs in the system, we must guarantee that whenever process request for resource , it does not hold any other resources.

- we can provide the all resources to the process that is required for it's execution before starting it's execution . **problem –** for example if there are three resource that is required by a process and we have given all that resource before starting execution of process then there might be a situation that initially we required only two resource and after one hour we want third resources and this will cause starvation for the another process that wants this resources and in that waiting time that resource can allocated to other process and complete their execution.
- We can ensure that when a process request for any resources that time the process does not hold any other resources. Ex- Let there are three resources DVD, File and Printer . First the process request for DVD and File for the copying data into the file and let suppose it is going to take 1 hour and after it the process free all resources then again request for File and Printer to print that file.

(iii) No Preemption

If a process is holding some resource and requestion other resources that are acquired and these resource are not available immediately then the resources that current process is holding are preempted. After some time process again request for the old resources and other required resources to re-start.

For example – Process p1 have resource r1 and requesting for r2 that is hold by process p2. then process p1 preempt r1 and after some time it try to restart by requesting both r1 and r2 resources.

Problem – This can cause the Live Lock Problem .

Live Lock **:** Live lock is the situation where two or more processes continuously changing their state in response to each other without making any real progress.

Example:

- suppose there are two processes p1 and p2 and two resources r1 and r2.
- Now, p1 acquired r1 and need r2 & p2 acquired r2 and need r1.
- so according to above method- Both p1 and p2 detect that they can't acquire second resource, so they release resource that they are holding and then try again.
- continuous cycle- p1 again acquired r1 and requesting to r2 p2 again acquired r2 and requesting to r1 so there is no overall progress still process are changing there state as they preempt resources and then again holding them. This the situation of Live Lock.

(iv) Circular Wait:

To remove the circular wait in system we can give the ordering of resources in which a process needs to acquire.

Ex: If there are process p1 and p2 and resources r1 and r2 then we can fix the resource acquiring order like the process first need to acquire resource r1 and then resource r2. so the process that acquired r1 will be allowed to acquire r2 , other process needs to wait until r1 is free.

This is the Deadlock prevention methods but practically only fourth method is used as all other three condition removal method have some disadvantages with them .

What is Deadlock Avoidance?

Avoidance is kind of futuristic. By using the strategy of "Avoidance", we have to make an assumption. We need to ensure that all information about resources that the process will need is known to us before the execution of the process. We use Banker's algorithm (Which is in turn a gift from Dijkstra) to avoid deadlock.

In prevention and avoidance, we get the correctness of data but performance decreases.

What is Deadlock Recovery?

If Deadlock prevention or avoidance is not applied to the software then we can handle this by deadlock detection and recovery. which consist of two phases:

1. In the first phase, we examine the state of the process and check whether there is a deadlock or not in the system.
2. If found deadlock in the first phase then we apply the algorithm for recovery of the deadlock.

In Deadlock detection and recovery, we get the correctness of data but performance decreases.

Methods of Deadlock Recovery

There are several Deadlock Recovery Techniques:

- Manual Intervention
- Automatic Recovery
- Process Termination
- Resource Preemption

1. Manual Intervention

When a deadlock is detected, one option is to inform the operator and let them handle the situation manually. While this approach allows for human judgment and decision-making, it can be time-consuming and may not be feasible in large-scale systems.

2. Automatic Recovery

An alternative approach is to enable the system to recover from deadlock automatically. This method involves breaking the deadlock cycle by either aborting processes or preempting resources. Let's delve into these strategies in more detail.

3. Process Termination

- **Abort all Deadlocked Processes**
    - This approach breaks the deadlock cycle, but it comes at a significant cost. The processes that were aborted may have executed for a considerable amount of time, resulting in the loss of partial computations. These computations may need to be recomputed later.
- **Abort one process at a time**
    - Instead of aborting all deadlocked processes simultaneously, this strategy involves selectively aborting one process at a time until the deadlock cycle is eliminated. However, this incurs overhead as a deadlock-detection algorithm must be invoked after each process termination to determine if any processes are still deadlocked.
    - **Factors for choosing the termination order:**
        - The process's priority
        - Completion time and the progress made so far
        - Resources consumed by the process
        - Resources required to complete the process
        - Number of processes to be terminated
        - Process type (interactive or batch)

4. Resource Preemption

- **Selecting a Victim**
    - Resource preemption involves choosing which resources and processes should be preempted to break the deadlock. The selection order aims to minimize the overall cost of recovery. Factors considered for victim selection may include the number of resources held by a deadlocked process and the amount of time the process has consumed.
- **Rollback**
    - If a resource is preempted from a process, the process cannot continue its normal execution as it lacks the required resource. Rolling back the process to a safe state and restarting it is a common approach. Determining a safe state can be challenging, leading to the use of total rollback, where the process is aborted and restarted from scratch.
- **Starvation Prevention**
    - To prevent resource starvation, it is essential to ensure that the same process is not always chosen as a victim. If victim selection is solely based on cost factors, one process might repeatedly lose its resources and never complete its designated task. To address this, it is advisable to limit the number of times a process can be chosen as a victim, including the number of rollbacks in the cost factor.

What is Deadlock Ignorance?

If a deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take. we use the ostrich algorithm for deadlock ignorance.

In Deadlock, ignorance performance is better than the above two methods but the correctness of data is not there.

Safe State

A safe state can be defined as a state in which there is no deadlock. It is achievable if:

- If a process needs an unavailable resource, it may wait until the same has been released by a process to which it has already been allocated. if such a sequence does not exist, it is an unsafe state.
- All the requested resources are allocated to the process.

Aspect	**Deadlock**	**Starvation**
Definition	A condition where two or more processes are blocked forever, each waiting for a resource held by another.	A condition where a process is perpetually denied necessary resources, despite resources being available.
Resource Availability	Resources are held by processes involved in the deadlock.	Resources are available but are continuously allocated to other processes.
Cause	Circular dependency between processes, where each process is waiting for a resource from another.	Continuous preference or priority given to other processes, causing a process to wait indefinitely.
Resolution	Requires intervention, such as aborting processes or preempting resources to break the cycle.	Can be mitigated by adjusting scheduling policies to ensure fair resource allocation.