Unit 3

Process Synchronization:

A **cooperating process** is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages.

1. The Critical section problem

Consider a system consisting of *n* processes {*P*0, *P*1, ..., *Pn*−1}. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The ***critical-section problem*** is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**. The general structure of a typical process *Pi* is shown in below Figure. The entry section and exit

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

General structure of a typical process P_i.

A solution to the critical-section problem must satisfy the following three requirements:

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.

Each process has its **own critical section**, but these critical sections often refer to **shared resources** or data that multiple processes need to access. The critical section is the part of a process's code where it accesses shared resources (like variables, files, or hardware). The issue arises because **multiple processes should not access shared resources simultaneously** to prevent data corruption or inconsistency.

To summarize:

- **Each process has its own critical section** in its code.
- These critical sections are problematic when they access the **same shared resource**, leading to the need for synchronization mechanisms (e.g., mutexes, semaphores) to ensure only one process at a time can enter its critical section.

Two general approaches are used to handle critical sections in operating systems: **preemptive kernels** and **nonpreemptive kernels**. A preemptive kernel allows a process to be preempted while it is running in kernel mode. A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks,or voluntarily yields control of the CPU.

Obviously, a nonpreemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time.

A **race condition** in an operating system occurs when two or more processes or threads access shared resources (like memory or data) simultaneously, and the final outcome depends on the specific order of execution. This creates unpredictable results because the processes "race" to access and modify the resource, leading to inconsistencies or errors if proper synchronization mechanisms are not used.

Example of Race Condition:

Consider two threads, **Thread A** and **Thread B**, both trying to increment the same global counter variable. If both threads try to read, increment, and write the value of the counter at the same time, a race condition can occur:

- **Thread A** reads the current value of the counter (say 5).
- **Thread B** also reads the value of the counter (also 5).

- **Thread A** increments the value (5 + 1 = 6).
- **Thread B** increments the same original value (5 + 1 = 6).
- Both threads write the value back as **6**, but the correct result should be **7**.

Here, due to the simultaneous access, the final value is incorrect.

Preventing Race Conditions:

To avoid race conditions, operating systems use synchronization mechanisms, such as:

- **Locks/Mutexes**: Ensure that only one thread can access the shared resource at a time.
- **Semaphores**: Limit access to shared resources by controlling the number of threads that can use the resource simultaneously.
- **Monitors**: High-level constructs that manage access to shared resources in a controlled manner.

The Critical Section Problem

The **critical section problem** occurs in concurrent programming when multiple processes or threads need to access a shared resource (such as a variable, file, or memory location) simultaneously. The goal is to ensure that only one process or thread can access the shared resource at any given time to prevent inconsistencies or race conditions.

Key Concepts of the Critical Section Problem:

1. **Critical Section**: A section of the code where the shared resource is accessed or modified. Only one process should be in the critical section at a time to ensure correctness.
2. **Entry Section**: The part of the program that requests permission to enter the critical section. This is where a process checks if it can proceed into the critical section.
3. **Exit Section**: After a process completes its task in the critical section, it executes the exit section, which allows other processes to enter the critical section.
4. **Remainder Section**: The portion of the code that does not involve accessing the shared resource and can be executed without any restrictions.

Conditions to Solve the Critical Section Problem:

To correctly manage the critical section problem, the solution must satisfy the following three requirements:

1. **Mutual Exclusion**: Only one process can enter the critical section at a time. If one process is executing in its critical section, no other process can enter their critical section.
2. **Progress**: If no process is in the critical section and multiple processes wish to enter, one of the processes must be allowed to enter the critical section eventually. This ensures that processes don't get stuck waiting indefinitely.
3. **Bounded Waiting**: There must be a limit on how long a process can wait to enter the critical section. This prevents indefinite postponement or starvation of any process.

Solutions to the Critical Section Problem:

Several solutions have been proposed to address the critical section problem, such as:

1. **Peterson's Algorithm**: A software-based solution that uses two variables (flags) to achieve mutual exclusion and progress.
2. **Semaphores**: A synchronization primitive that uses counters to manage access to the critical section.
3. **Mutex Locks**: Locks that ensure only one process can enter the critical section by acquiring the lock, and it releases the lock once the process exits the critical section.

Importance:

Handling the critical section properly is crucial for ensuring the consistency of shared resources, avoiding race conditions, and maintaining the correct functioning of concurrent systems.

2. Peterson's Solution

Peterson's Solution is a classic software-based algorithm for solving the **critical section problem** in concurrent programming. It ensures **mutual exclusion** and prevents race conditions when two processes need to access a shared resource. This algorithm is known for being simple and efficient but works only for two processes.

Peterson's Solution uses two shared variables:

1. **flag[]**: An array of two Boolean flags (flag[0] and flag[1]), where each process sets its flag to indicate its intention to enter the critical section.
2. **turn**: A variable that indicates whose turn it is to enter the critical section. If turn == i, process i has priority to enter the critical section.

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

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

The structure of process P_i in Peterson's solution.

Explanation

1. `flag[i] = true;`
   - This line sets the flag for the current process `i` to `true`, indicating that process `i` wants to enter the critical section.
2. `turn = j;`
   - This line gives the turn to process `j`, allowing the other process (process `j`) the chance to enter the critical section if it also wants to.
3. `while (flag[j] && turn == j);`
   - This line is a **busy-waiting loop**. It keeps process `i` waiting if:
     - `flag[j]` is `true` (meaning the other process `j` wants to enter the critical section), and
     - `turn == j` (meaning it's process `j`'s turn to enter the critical section).

- Process `i` will proceed to the critical section only when either process `j` doesn't want to enter (i.e., `flag[j] == false`) or it is process `i`'s turn (`turn != j`).

```
4. critical section
```

- This is the part of the code where process `i` enters the **critical section** and performs its operation on the shared resource.

```
5. flag[i] = false;
```

- After exiting the critical section, process `i` sets its `flag` to `false`, indicating that it no longer wants to enter the critical section.

```
6. remainder section
```

- This is the section of code that process `i` executes outside of the critical section (non-critical operations).

```
7. } while (true);
```

- The `do-while` loop ensures that this process repeats indefinitely, meaning the process can repeatedly request access to the critical section.

Steps in Peterson's Solution:

Let's assume there are two processes: **P0** and **P1**.

1. **Before entering the critical section**, each process sets its own flag to true to show its intention to enter the critical section.
2. The process then gives the other process a chance to proceed by setting the turn variable to the other process.
3. The process will only enter the critical section if:
   - The other process does not wish to enter the critical section (its flag is false), or
   - It is the process's turn to enter the critical section.
4. **After leaving the critical section**, the process resets its own flag to false to indicate it has left the critical section.

Properties:

- **Mutual Exclusion**: Only one process can enter the critical section at any time.
- **Progress**: If no process is in the critical section, the one that wants to enter is allowed to do so.
- **Bounded Waiting**: No process waits indefinitely to enter the critical section; the turn variable ensures that each process gets a fair chance.

Limitations:

- Peterson's solution is limited to **two processes** and may not work in modern systems due to optimizations such as instruction reordering by compilers and hardware. For more than two processes, more complex algorithms or synchronization primitives (like semaphores or mutexes) are needed.

Process Synchronization:

1. Synchronization hardware

Software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures. There are other solutions to **critical section problem** which are based on the premise of **locking** —that is, protecting critical regions through the use of locks.

In simple terms, hardware instructions available in many systems can help solve the **critical-section problem**—which occurs when multiple processes try to access shared resources at the same time. These hardware features can make programming easier and improve system efficiency.

One straightforward solution in a **single-processor environment** is to temporarily stop interrupts (signals that can pause the current task). This ensures that no other process or instruction can interrupt the current sequence of steps, especially when a shared variable is being updated. By preventing interrupts, we ensure that the task finishes completely before any other process gets a chance to modify the shared variable. This approach is often used in **nonpreemptive kernels**, where tasks aren't interrupted once they start, ensuring safe and predictable access to shared resources.

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

The definition of the `test_and_set()` instruction.

Breakdown of `test_and_set()`:

1. **Input**:
   - The function takes a pointer to a boolean variable `target`. This variable acts as a **lock** that controls access to the critical section. Initially, this `target` (or lock) is set to `false`, meaning the critical section is not occupied.
2. **Save the Value**:
   - It stores the current value of `target` in a local variable `rv` (`rv = *target;`). If `target` is `false`, this means the critical section is available.
3. **Set the Lock**:
   - It then sets `target` to `true` (`*target = true;`), meaning the lock is now acquired, and no other process can enter the critical section.
4. **Return the Original Value**:
   - The function returns the original value of `target` (`return rv;`). If `rv` is `false`, this indicates that the critical section was previously available and the current process can proceed. If `rv` is `true`, it indicates the section is already locked by another process.

Saving the original value lets the function check the lock status in an atomic operation. Without this step, there's no way to tell if the process was already locked by another process before it attempted to acquire the lock.

Example

- **When `P1` calls `test_and_set`:**
  - `P1` reads the current value of `target`, storing it in `rv`. Suppose `target` is `false` initially.
  - `rv` is set to `false`, and `P1` then sets `target` to `true`, indicating that `P1` has entered the critical section.
  - `P1` returns the value of `rv` (which is `false`), indicating it successfully acquired the lock.
- **When `P2` calls `test_and_set` after `P1`:**
  - `P2` calls `test_and_set`, and within this call, `rv` stores the current value of `target`, which is now `true` (since `P1` has set it).
  - `rv` in `P2`'s call is therefore `true`, and `P2` sets `target` to `true` again (though it's already `true`), indicating the lock is still taken.
  - `P2` returns `rv` (which is `true`), meaning it did **not** acquire the lock since it was already held by `P1`.

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Mutual-exclusion implementation with `test_and_set()`.

Step-by-step Explanation:

1. **Busy-Wait Loop (`while (test_and_set(&lock));`)**:
   - The `test_and_set()` function is repeatedly called in a loop (busy-wait) until it returns `false`. This indicates that the lock was previously released, and the process can now enter the critical section.
   - If `test_and_set(&lock)` returns `true`, the process will keep waiting because another process has acquired the lock.
2. **Critical Section**:
   - Once `test_and_set(&lock)` returns `false`, the process can enter the critical section and execute its critical operations.
3. **Release the Lock (`lock = false;`)**:
   - After finishing its critical section, the process sets `lock = false`, releasing the lock so that other processes can enter the critical section.
4. **Remainder Section**:
   - The process then proceeds to the remainder section, where it can perform non-critical operations.
5. **Repetition**:
   - The process repeats this cycle indefinitely (`do-while(true)`), constantly requesting access to the critical section when needed.

The code uses the **test_and_set()** instruction to implement **mutual exclusion** and ensure only one process can enter the **critical section** at a time.

- **test_and_set()** checks the current value of the lock and sets it to true (locked) at the same time. It returns the original value of the lock before it was changed.
- In the loop, the process keeps checking the lock. If test_and_set() returns true (meaning the lock was already set), the process waits. If it returns false, the process has acquired the lock and can enter the **critical section**.
- After the critical section, the lock is reset to false (unlocked), allowing other processes to access the critical section.

This ensures that only one process can access the critical section at a time, preventing conflicts and ensuring **mutual exclusion**.

In a **multiprocessor environment**, the solution of disabling interrupts doesn't work as well. This is because turning off interrupts on all processors takes time, as the system needs to send a message to each processor to stop them. This delay makes entering the **critical section** slower and reduces the system's overall efficiency.

Additionally, if the system's clock is updated using interrupts, turning off interrupts can affect the accuracy of the clock, causing problems with timekeeping in the system. So, while this method works for single processors, it's less practical for systems with multiple processors.

Modern computer systems offer special hardware instructions that let us perform certain tasks in one uninterrupted step (atomically). These instructions can either:

1. **Test and modify** the content of a memory word.
2. **Swap** the contents of two memory words.

Since these tasks are done as a single, uninterruptible operation, they help solve the **critical-section problem** more easily. By using these special instructions, we can ensure that shared data is safely accessed and modified by different processes without interference.

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

The definition of the `compare_and_swap()` instruction.

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

Mutual-exclusion implementation with the compare_and_swap() instruction.

`int *value`:

- This is a pointer to an integer variable in memory that is shared among multiple threads. The `*value` represents the current value held by that variable.

`while (compare_and_swap(&lock, 0, 1) != 0)`: This line implements a busy-wait (spinlock). Here's what happens:

- `compare_and_swap(&lock, 0, 1)` checks if `lock` is `0` (indicating that the critical section is currently free).
- If `lock` is `0`, it sets `lock` to `1` (locking it) and allows the thread to enter the critical section.
- If `lock` is already `1`, the `compare_and_swap` call returns `1`, so the loop spins (continues to `do nothing`) until the lock becomes available.

The given code implements **mutual exclusion** using the compare_and_swap() instruction to ensure only one process can enter the **critical section** at a time. Here's a breakdown:

- **compare_and_swap()** checks if the value of a variable (lock) matches an expected value (in this case, 0). If so, it changes the value to a new one (1). If not, it leaves the value unchanged and returns the original.
- The loop repeatedly checks until compare_and_swap() returns 0, meaning the lock is free (unlocked). Once the lock is acquired (set to 1), the process enters the **critical section**.
- After completing the critical section, the lock is reset to 0 (unlocked), allowing other processes to enter.

This ensures **mutual exclusion**—only one process can enter the critical section at a time, preventing race conditions.

Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement. In the below Figure , we present another algorithm using the test and set() instruction that satisfies all the critical-section requirements. The common data structures are

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

      /* critical section */

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

   if (j == i)
      lock = false;
   else
      waiting[j] = false;

      /* remainder section */
} while (true);
```

Bounded-waiting mutual exclusion with `test_and_set()`.

This code implements **bounded-waiting mutual exclusion** using the test_and_set() instruction, ensuring each process gets a fair chance to enter the **critical section** without waiting indefinitely.

1. **Initial setup**:
   - The current process sets waiting[i] = true, indicating it wants to enter the critical section.
   - The key is set to true, and the process tries to acquire the lock using test_and_set(&lock).
2. **Acquiring the lock**:
   - The loop continues until the lock is free (test_and_set() returns false). When the process successfully sets the lock, it exits the loop and sets waiting[i] = false.
3. **Critical section**:
   - Once inside the critical section, the process performs its task.

4. **Releasing the lock**:
    - After leaving the critical section, the process checks if any other process is waiting to acquire the lock.
    - It looks for the next process (j) that has set waiting[j] = true. If no other process is waiting, it releases the lock (lock = false); otherwise, it allows the next waiting process to enter by setting waiting[j] = false.

This method ensures **bounded waiting**—meaning every process will eventually get its turn to access the critical section without indefinite delays—and maintains **mutual exclusion** so that only one process is in the critical section at any time.

https://youtu.be/q7yksMlPlJU?si=v3U-pCPQqxBPD1lI

2. Mutex locks

A **mutex lock** is a software tool used to solve the **critical section problem**, which helps prevent **race conditions** when multiple processes access shared resources.

- **Mutex** stands for **mutual exclusion**, meaning only one process can access the critical section at a time.
- Before entering a **critical section**, a process needs to call the acquire() function to **lock** the section.
- After finishing, it calls the release() function to **unlock** the section so other processes can use it.

The mutex uses a simple **boolean variable** (true/false) to track if the lock is available. If the lock is available, a process can successfully acquire it. If another process tries to acquire the lock while it's in use, it will be **blocked** (wait) until the lock is released by the current process.

The definition of acquire() is as follows:

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;;
}
```

The definition of release() is as follows:

```
release() {
   available = true;
}
```

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

Solution to the critical-section problem using mutex locks.

The main **disadvantage** of using a **mutex lock** (as implemented) is the issue of **busy waiting**, also known as **spinlocks**.

- When a process is inside its **critical section**, other processes that want to enter will continuously check the lock by repeatedly calling acquire(). This makes them **"spin"** in a loop, wasting valuable **CPU cycles** instead of doing useful work.
- In a **multiprogramming system**, where many processes share the CPU, this results in poor performance since the CPU is consumed by processes just waiting for the lock instead of doing productive tasks.

However, **spinlocks** can be useful in certain cases:

- They **don't require a context switch**, which saves time because a context switch (switching from one process to another) can take a significant amount of time.
- Spinlocks are ideal when the **lock is expected to be held for short periods**, especially in **multiprocessor systems**, where one thread can wait (or spin) on one processor while another thread uses the lock on a different processor.

3. Semaphores

A **semaphore** is a synchronization tool used in operating systems to manage access to shared resources by multiple processes, helping to avoid issues like race conditions or deadlocks.

There are two types of semaphores:

1. **Counting Semaphore**: It allows multiple processes to access a resource up to a certain limit. It has a counter that indicates the number of available resources. When the counter is positive, processes can access the resource, and when it's zero, they must wait.
2. **Binary Semaphore**: This functions like a mutex, allowing only one process to access a critical section at a time. It can take the value of 0 or 1, indicating whether the resource is locked or unlocked.

Semaphores work with two primary operations:

- **Wait (P)**: Decreases the semaphore value. If the value is less than or equal to 0, the process waits until it's positive again.
- **Signal (V)**: Increases the semaphore value, allowing waiting processes to proceed.

Semaphores help coordinate process execution and prevent conflicts in accessing shared resources.

definition of `wait()` is as follows:

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

The definition of `signal()` is as follows:

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

In semaphore operations, modifications to the semaphore's integer value during the **wait()** and **signal()** operations must occur indivisibly. This means that no other process can change the semaphore value while one process is currently modifying it, ensuring synchronization.

For instance:

- During a **wait(S)** operation, the system checks if the semaphore value (S) is less than or equal to zero ($S \leq 0$), and if so, it blocks the process from proceeding. If the value is positive, it decrements the value (S--) and allows the process to proceed.
- During a **signal(S)** operation, the semaphore value is incremented, allowing waiting processes to continue.

The critical point is that both the checking and modifying of the semaphore value (S) must be done atomically—without interruption from other processes. This guarantees that race conditions are avoided and processes interact with shared resources in a safe and controlled manner.

Key Differences Between Semaphore and Mutex

Feature	Semaphore	Mutex
Purpose	Used to manage multiple instances of a resource.	Used to manage single access to a resource.
Value Range	Can have a value greater than 1 (counting semaphores).	Binary value (locked or unlocked) - essentially 0 or 1.
Types	Counting semaphore (value > 1) and binary semaphore (value = 1).	Only one type, functioning as a binary lock.
Ownership	No concept of ownership – any process can signal or wait.	The process that locks (owns) it must be the one to unlock.
Use Case	Useful for limiting access to resources with multiple instances (e.g., limiting threads accessing a pool of connections).	Best for mutual exclusion, where only one thread accesses a critical section at a time.
Blocking Mechanism	If a semaphore's value is zero, threads trying to access it will wait until another thread releases (signals) it.	If locked, a thread trying to lock it will block until the mutex is released.
Deadlock Prevention	Less strict, allowing different threads to wait or signal, which can reduce deadlock risk but may require more careful handling.	Deadlocks can occur if a thread fails to unlock after locking.
Performance Overhead	Generally more complex with slightly higher overhead.	Simple binary lock; generally faster with minimal overhead.

i) Semaphore Usage

Operating systems distinguish between two types of semaphores:

1. **Counting Semaphores**: These have values that range over an unrestricted domain. They are often used to control access to a pool of resources (e.g., multiple instances of a resource). The semaphore value represents the number of available resources. When a process needs a resource, it performs a **wait()** operation, which decrements the semaphore's value. When a process releases a resource, it performs a **signal()** operation, incrementing the value. When the semaphore reaches 0, it indicates that all resources are in use, and further processes must wait until resources are freed.
2. **Binary Semaphores**: These are similar to **mutex locks** and can take values only between 0 and 1, controlling access to a single resource. Binary semaphores are often used for **mutual exclusion**, ensuring that only one process can access a critical section at a time. If a system does not have mutex locks, binary semaphores can be used instead.

Example of Synchronization Using Semaphores:

Consider two processes, **P1** and **P2**, where we need to ensure that **P2** executes statement **S2** only after **P1** has executed statement **S1**.

P2→ S2 iff P1→S1

- Semaphore **synch** is initialized to 0.
- In **P1**, we insert the statements

```
S1;
signal(synch);
```

In process P_2, we insert the statements

```
wait(synch);
S2;
```

- Process P1 executes statement S1.
- After completing S1, it calls signal(synch) to increment the semaphore's value by 1.
- This signals that S1 has completed, allowing P2 to proceed if it is waiting on the semaphore.

- Process `P2` starts by calling `wait(synch)`. Since `synch` is initially `0`, `P2` is blocked and cannot proceed until the semaphore value becomes greater than `0`.
- When `P1` completes `S1` and calls `signal(synch)`, the semaphore value is incremented to `1`, unblocking `P2`.
- `P2` then continues to execute `S2`.

This setup guarantees that **P2** will execute **S2** only after **P1** has completed **S1**, solving the synchronization problem.

ii) Semaphore Implementation

In operating systems, semaphore implementations can be improved to avoid **busy waiting**. Instead of having a process loop continuously while waiting for a semaphore to become available, the process can **block** itself. This approach uses a waiting queue to manage processes that are waiting for a semaphore.

1. **Semaphore Structure**:
   - A semaphore consists of an integer value and a list of processes that are waiting on it.

```
typedef struct
{
   int value;          // Semaphore value
   struct process *list; // List of waiting processes
} semaphore;
```

- **`int value`**:
  - Represents the current value of the semaphore.
  - **Positive values**: Indicate the number of resources available (or that no processes are waiting).
  - **Zero or negative values**: Reflect contention for the resource.
    - A value of 0 means the resource is currently being accessed, but no processes are waiting.
    - A negative value indicates how many processes are waiting for the resource (e.g., `value = -3` means three processes are waiting).

- **`struct process *list`**:
  - This is a pointer to a list of processes that are currently waiting for the semaphore.
  - When a process executes a `wait` operation (`P()`), and the semaphore's value is less than or equal to 0, the process is added to this waiting list.

- When the semaphore is incremented using `signal` (`V()`), a process from the waiting list is removed and allowed to proceed.

2. **Wait Operation**:
    - When a process calls wait (semaphore *S), the semaphore value is decremented. If the value is less than 0, the process is added to the waiting list and blocked (stops executing).

```
wait(semaphore *S)
{
   S->value--;
   if (S->value < 0)
   {
      add this process to S->list; // Block the process
      block(); // Switch to waiting state
   }
}
```

3. **Signal Operation**:
    - When a process calls signal (semaphore *S), the semaphore value is incremented. If the value is less than or equal to 0, it means there are waiting processes, and one of them is removed from the list and awakened.

```
signal(semaphore *S)
{
   S->value++;
   if (S->value <= 0)
   {
      remove a process P from S->list; // Wake up a waiting process
      wakeup(P); // Change the process to ready state
   }
}
```

4. **Blocking and Waking Up**:
    - Blocking a process when it waits on a semaphore prevents busy waiting, which wastes CPU cycles. The blocked process is switched out for another one, improving overall efficiency.
    - When a signal is issued, a waiting process is transitioned back to the ready state.
5. **Atomic Operations**:
    - It's essential that wait() and signal() operations are executed atomically, meaning that no two processes should execute these operations on the same semaphore at the same time. This prevents race conditions.
6. **Handling Busy Waiting**:
    - Although this approach minimizes busy waiting, it does not completely eliminate it, especially if the critical sections are long. In those cases, busy waiting can still be inefficient.

iii) Deadlocks and Starvation

Deadlocks and starvation are two critical issues that can arise in systems that use semaphores for synchronization.

Deadlocks

- **Definition**: A deadlock occurs when two or more processes are waiting indefinitely for events that can only be triggered by one of those processes. In this state, none of the involved processes can continue executing.
- **Example**: Consider two processes, P0 and P1, both trying to access two semaphores, S and Q, each initialized to 1:
    - P0 executes wait(S) and then tries to execute wait(Q).
    - P1 executes wait(Q) and then tries to execute wait(S).

The sequence of operations might look like this:

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

- At this point, P0 is waiting for P1 to release Q, and P1 is waiting for P0 to release S, creating a deadlock situation where neither process can proceed.
- **Deadlocked State**: We say a set of processes is in a deadlocked state when every process in the set is waiting for an event that can only be caused by another process in the same set.

Starvation

- **Definition**: Starvation (or indefinite blocking) occurs when a process waits indefinitely to acquire a semaphore. This can happen if processes are removed from the waiting queue in a way that consistently favours certain processes over others, such as using a **last-in, first-out (LIFO)** order.
- **Example**: If a semaphore's waiting list follows a LIFO order, a newly waiting process might always get preference over an older waiting process. This situation can lead to the older process being starved indefinitely if new processes keep entering the queue.

Deadlocks are situations where processes are waiting on each other indefinitely, while **starvation** occurs when a process is perpetually denied the resources it needs to proceed. Both problems highlight the importance of careful resource management and scheduling in operating systems to ensure fairness and avoid situations where processes cannot make progress. Mechanisms to detect and handle deadlocks, as well as to prevent starvation, are essential for maintaining system stability.

Deadlock is a situation where two or more processes are unable to proceed because each is waiting for the other to release a resource. It is a state of indefinite waiting that occurs due to improper resource allocation in a concurrent system.

All the below four conditions must be present simultaneously for a deadlock to occur. By ensuring that at least one of these conditions is broken, a system can avoid or prevent deadlocks.

The four **key characteristics of deadlock** are as follows:

1. **Mutual Exclusion**:
   - At least one resource involved in the deadlock must be held in a **non-shareable mode**. This means only one process can use the resource at any given time. If another process requests it, it must wait until the resource is released.
2. **Hold and Wait**:
   - A process is holding one or more resources and is simultaneously waiting to acquire additional resources that are currently held by other processes. This creates dependency chains.
3. **No Preemption**:
   - Resources cannot be forcibly taken from a process holding them. The resource can only be released voluntarily by the process once it has completed its task.
4. **Circular Wait**:
   - A set of processes are involved in a circular chain of resource requests, where each process in the chain is waiting for a resource held by the next process in the chain. For example:
     - Process A is waiting for a resource held by Process B.
     - Process B is waiting for a resource held by Process C.
     - Process C is waiting for a resource held by Process A.

Example:

- Process A holds Resource 1 and waits for Resource 2.
- Process B holds Resource 2 and waits for Resource 1.
- Neither process can proceed, resulting in a deadlock.

Dealing with Deadlocks:

1. **Prevention**: Avoid one or more of the four conditions necessary for deadlock.
2. **Avoidance**: Use algorithms (like the Banker's algorithm) to ensure the system remains in a safe state.
3. **Detection and Recovery**: Detect deadlocks and recover by terminating or preempting processes.
4. **Ignoring**: Sometimes, deadlocks are rare and considered acceptable in certain systems.

iv) Priority Inversion

Priority inversion is a scheduling problem that occurs in multitasking operating systems when a higher-priority process is forced to wait for a lower-priority process to release a resource. This situation can be worsened when the lower-priority process is preempted by an intermediate-priority process.

Example Scenario

1. **Processes and Priorities**:
    - Let's assume we have three processes:
        - Process L (low priority)
        - Process M (medium priority)
        - Process H (high priority)
    - Their priority order is: L < M < H.
2. **Resource Usage**:
    - Process H requires access to a resource, say **resource R**, that is currently being used by Process L.
    - Normally, Process H would wait for Process L to finish using resource R.
3. **Interruption**:
    - If Process M becomes runnable (e.g., it is ready to run and gets CPU time), it preempts Process L, which is still holding resource R.
    - Now, Process H must wait for Process L to finish, but Process L cannot run because it has been preempted by the higher-priority Process M.

Consequence of Priority Inversion

In this scenario, Process H, which has the highest priority, ends up waiting longer than necessary because its execution is indirectly blocked by Process M, which has a lower priority than H but higher than L. This results in **priority inversion** where the higher-priority process's execution is delayed due to a chain of lower-priority processes.

Solution: Priority Inheritance Protocol

To resolve the issue of priority inversion, operating systems often implement a **priority-inheritance protocol**:

- When a lower-priority process (like L) holds a resource needed by a higher-priority process (like H), it temporarily inherits the higher priority until it releases the resource.
- This prevents any medium-priority process (like M) from preempting the lower-priority process (L), allowing it to complete its task quickly.

How It Works:

1. Process L, while holding resource R, inherits the priority of Process H.
2. As a result, Process L can run without being preempted by Process M.
3. Once Process L finishes its execution and releases resource R, it reverts back to its original priority.
4. Now, Process H can run immediately, as it can access resource R.

Priority inversion can lead to inefficiencies and unpredictable behavior in multitasking systems. The priority-inheritance protocol helps mitigate this issue by allowing lower-priority processes to temporarily assume the priority of higher-priority processes when they hold resources needed by them. This way, the system can ensure that higher-priority tasks are completed in a timely manner, improving overall performance and responsiveness.

4. Classic problems of synchronization

i) The Bounded-Buffer Problem / Producer-Consumer Problem

The **Bounded-Buffer Problem**, also known as the **Producer-Consumer Problem**, is a classic synchronization challenge. It involves coordinating multiple processes that produce and consume items in a fixed-size buffer. Here's a breakdown:

Problem Overview

In this problem:

- A **producer** generates items and places them in a shared buffer.
- A **consumer** retrieves items from the buffer.
- The buffer has a limited capacity, so it can hold only a certain number of items at any time.

Constraints

1. The **producer** must wait if the buffer is full (i.e., no space to add more items).
2. The **consumer** must wait if the buffer is empty (i.e., no items to consume).
3. Only one process (producer or consumer) should modify the buffer at a time to avoid race conditions.

Solution with Synchronization

To implement this solution, we use:

- **Semaphores** to control access and ensure synchronization.
- A **mutex (binary semaphore)** to manage exclusive access to the buffer.
- Two **counting semaphores**:
  - `full` to track the number of items in the buffer.
  - `empty` to track the available space in the buffer.

Shared Data Structures: The producer and consumer processes share the following data structures:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

- **Integer n**: Represents the number of available buffer slots.
- **Semaphores**:
  - mutex: Ensures mutual exclusion when accessing the buffer. Initialized to 1.
  - empty: Counts the number of empty buffers. Initialized to n (all buffers are empty initially).
  - full: Counts the number of full buffers. Initialized to 0 (no buffers are full initially).

Overview of the Problem

- **Producer Process**: The producer generates data and places it into the buffer. It must ensure that it does not add data to a full buffer.
- **Consumer Process**: The consumer retrieves data from the buffer. It must ensure that it does not attempt to consume data from an empty buffer.

Producer:

- The producer generates an item and waits until there's an empty buffer available.
- It locks the buffer using the mutex semaphore to ensure exclusive access while adding the item.

- After adding the item, it signals the full semaphore to indicate that there is now one more full buffer.

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

The structure of the producer process.

Consumer:

- The consumer waits until there is a full buffer to consume.
- It locks the buffer using the mutex semaphore to ensure exclusive access while removing the item.
- After removing the item, it signals the empty semaphore to indicate that there is now one more empty buffer.

```
do {
   wait(full);
   wait(mutex);
      . . .
   /* remove an item from buffer to next_consumed */
      . . .
   signal(mutex);
   signal(empty);
      . . .
   /* consume the item in next_consumed */
      . . .
} while (true);
```

The structure of the consumer process.

1. **`wait(full);`**
   - The consumer waits (decrements) the `full` semaphore, which represents the number of filled slots in the buffer.
   - If `full` is greater than zero, it means there's at least one item in the buffer, and the consumer can proceed to consume it.
   - If `full` is zero, it means the buffer is empty, and the consumer will wait until a producer adds an item.
2. **`wait(mutex);`**
   - The consumer then waits (decrements) the `mutex` semaphore to enter the critical section.
   - `mutex` is a binary semaphore (also known as a mutex lock) that ensures **mutual exclusion** in accessing the shared buffer. Only one process (either a producer or a consumer) can access the buffer at a time.
   - This prevents race conditions, ensuring that no other process can modify the buffer while this consumer is removing an item.
3. **`/* remove an item from buffer to next consumed */`**
   - Inside the critical section, the consumer removes an item from the buffer.
   - This item is stored in a temporary variable, `next consumed`, where it will later be processed by the consumer.
   - This part of the code removes one item, effectively freeing up a space in the buffer.
4. **`signal(mutex);`**
   - The consumer signals (increments) the `mutex` semaphore to release the lock on the critical section.
   - Other processes (like another consumer or producer) can now access the buffer.
5. **`signal(empty);`**
   - The consumer signals (increments) the `empty` semaphore, which represents the number of empty slots in the buffer.
   - By incrementing `empty`, the consumer indicates that there is now an additional space in the buffer where a producer can add an item.
6. **`/* consume the item in next consumed */`**
   - Outside the critical section, the consumer processes or "consumes" the item that it previously removed from the buffer (stored in `next consumed`).
   - This step represents the actual use of the item, which could be anything from processing data to displaying output.
7. **`while (true);`**
   - The loop continues infinitely, meaning the consumer will repeatedly attempt to remove and consume items from the buffer as long as the program runs.

Purpose of Each Semaphore

- **`full`**: Ensures the consumer only tries to remove an item when there's something in the buffer. It prevents the consumer from consuming from an empty buffer.
- **`mutex`**: Provides mutual exclusion, ensuring that only one process (consumer or producer) can access the buffer at any given time.
- **`empty`**: Informs the producer that there is an empty slot in the buffer, allowing it to add new items when space becomes available.

ii) The Readers-Writers Problem

The Readers-Writers Problem is a classic synchronization issue that arises in database management systems when multiple concurrent processes need to access shared data. This problem distinguishes between two types of processes:

1. **Readers**: Processes that only read data from the database.
2. **Writers**: Processes that can read and modify the data.

Key Requirements

- **Readers** can access the database simultaneously without issues.
- **Writers**, however, must have exclusive access to the database when writing. This means that if a writer is writing to the database, no other readers or writers should access it.

Variants of the Problem: There are two common variations of the Readers-Writers Problem:

1. First Readers-Writers Problem (Readers Preference)

- Readers are prioritized over writers.
- If a reader is reading, a writer has to wait until all readers finish reading.
- **Drawback**: Writers can suffer **starvation** if there is a continuous stream of readers, as they never get a chance to write.

2. Second Readers-Writers Problem (Writers Preference)

- Writers are prioritized over readers.
- When a writer requests access, it gets priority and will wait until all readers currently reading are done, but no new readers can start reading.
- **Drawback**: Readers may suffer starvation if there are frequent writing requests.

Potential Issues

- **Starvation**: This can occur in both variants. In the first case, writers may starve if there are always readers present. In the second case, readers may starve if a writer is waiting.

Solution for the First Readers-Writers Problem

The solution involves managing access to the shared data using semaphores and a counter to track the number of active readers.

Data Structures- In the solution to the first readers–writers problem, the reader processes share the following data structures:

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

- **Semaphores**:
  - semaphore rw_mutex = 1;: Used to control access for writers.
  - semaphore mutex = 1;: Used to ensure mutual exclusion when updating the read_count.
- **Integer**:
  - int read_count = 0;: Tracks the number of active readers.

Reader Process:

- The reader first acquires a lock on mutex to safely increment the read_count.
- If it is the first reader (i.e., read_count was 0), it then locks out writers by calling wait(rw_mutex).
- After reading, it decrements read_count. If it was the last reader, it signals the writers that they can proceed by calling signal(rw_mutex).

- **mutex** synchronizes access to read_count, not the shared resource.
- **rw_mutex** allows concurrent reads as long as read_count is non-zero and prevents writer access during that time

```
do {
   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);
} while (true);
```

The structure of a reader process.

- **Initialization and Setup**
  - Assume that `mutex` and `rw_mutex` are binary semaphores initialized to 1.
  - The variable `read_count` is initially set to 0, representing the number of active readers.

- **Reader Process Logic**

The `do...while` loop represents the continuous behavior of a reader process that may repeatedly read data. Each reader process goes through the following steps:

- **Step 1:** `wait(mutex);`
  The reader acquires the `mutex` semaphore before updating the `read_count` variable. This ensures that changes to `read_count` are synchronized, so only one reader can modify `read_count` at a time.
- **Step 2:** `read_count++;`
  The reader increments `read_count` to reflect that a new reader is accessing the shared resource.
- **Step 3:** `if (read_count == 1) wait(rw_mutex);`
  If this is the first reader (i.e., `read_count` becomes 1), it acquires the `rw_mutex` semaphore to block writers. This ensures that writers cannot access the resource while readers are reading.
- **Step 4:** `signal(mutex);`
  The reader releases the `mutex` semaphore after updating `read_count`. This allows other readers to enter the critical section and read concurrently, as long as no writer is waiting.

- **Reading Phase**
  - The reader performs the actual reading of the shared resource in this phase. Multiple readers can read simultaneously, as they do not interfere with each other.

- **Completion of Reading**

After reading, the reader decrements `read_count` to indicate it is done:

- **Step 5:** `wait(mutex);`
  The reader acquires `mutex` again before modifying `read_count`, ensuring exclusive access to the variable.
- **Step 6:** `read_count--;`
  The reader decrements `read_count` to reflect that it has finished reading.
- **Step 7:** `if (read_count == 0) signal(rw_mutex);`
  If this reader is the last one (i.e., `read_count` becomes 0), it releases the `rw_mutex` semaphore, allowing waiting writers to proceed. This ensures that writers can access the shared resource once all readers have finished.
- **Step 8:** `signal(mutex);`
  Finally, the reader releases `mutex`, allowing other readers to access or modify `read_count` as needed.

- **Loop Continuation**

The `do...while(true);` loop allows the reader process to repeat this procedure, simulating a reader that repeatedly accesses the shared resource.

Writer Process:

- The writer process acquires the rw_mutex semaphore for exclusive access. Once it finishes writing, it releases the lock.

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

The structure of a writer process.

1. **`wait(rw_mutex);`**
   - The writer first calls `wait(rw_mutex);` to request access to the shared resource.
   - If `rw_mutex` is available (i.e., its value is 1), the writer can proceed to the writing section, and `rw_mutex` is decremented to 0. This locks access, preventing any other writer or reader from accessing the shared resource.
   - If `rw_mutex` is already locked (i.e., its value is 0), the writer will be blocked until `rw_mutex` is released by another writer.
2. **`/* writing is performed */`**
   - Once the writer has successfully acquired `rw_mutex`, it can safely perform the writing operation on the shared resource.
   - During this time, no other process (reader or writer) can access the shared resource, as `rw_mutex` remains locked.
3. **`signal(rw_mutex);`**
   - After completing the write operation, the writer releases `rw_mutex` by calling `signal(rw_mutex);`, which increments `rw_mutex` to 1.

- This unlocks the resource, allowing either a waiting reader or another writer to access the shared data.

4. `while (true);`
   - The `while (true);` loop means that the writer will continuously attempt to access and write to the shared resource, making this a repeated writing process.

Purpose of `rw_mutex`

- **Exclusive Access for Writers**: The semaphore `rw_mutex` ensures that only one writer can access the shared data at any given time, enforcing **mutual exclusion**.
- **Preventing Reader Interference**: While `rw_mutex` is locked by a writer, readers are also prevented from accessing the shared resource, ensuring that no data inconsistencies occur while writing.

In the **Readers-Writers Problem**, the rw_mutex semaphore typically functions as a **binary semaphore** (or mutex), which means it can only hold values **0** or **1**.

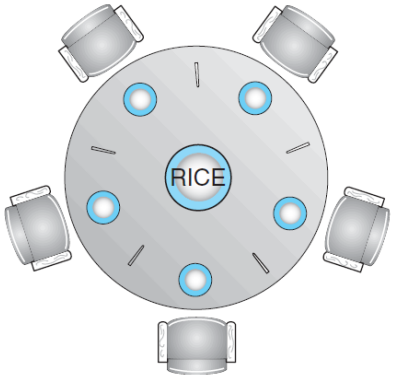
rw_mutex Values:

1. **rw_mutex = 1**
   - A value of **1** indicates that the shared resource is currently **available**.
   - When rw_mutex is 1, it means no writer is currently writing, and the resource is free for a writer to lock for exclusive access.
   - Readers or writers can attempt to access the shared resource, depending on the implementation.
2. **rw_mutex = 0**
   - A value of **0** indicates that the shared resource is **locked** by a writer.
   - When a writer calls wait(rw_mutex), it decrements rw_mutex from 1 to 0, signaling that it has exclusive access to the resource.
   - As long as rw_mutex remains 0, other writers and readers will be blocked from accessing the resource.

The Readers-Writers Problem is an important example in synchronization, illustrating how semaphores can be effectively used to manage access to shared resources. Solutions to this problem can help prevent race conditions and ensure data consistency, especially in systems where read operations significantly outnumber write operations.

iii) The Dining-Philosophers Problem

The Dining-Philosophers Problem is a classic synchronization problem that illustrates challenges in resource allocation, particularly in a concurrent environment. It involves a scenario with five philosophers who alternate between thinking and eating, sharing a limited number of resources (chopsticks) that can lead to issues like deadlock and starvation.



The situation of the dining philosophers.

Problem Description

In this problem:

- **Five Philosophers** are seated around a circular dining table.
- **Five Forks** (or chopsticks) are placed between each pair of philosophers.
- Each philosopher has two possible states:
    1. **Thinking**: The philosopher does not need any forks.
    2. **Eating**: The philosopher needs both forks (one on their left and one on their right) to eat.

To eat, each philosopher needs both the fork to their left and the fork to their right. After eating, they put the forks back on the table and go back to thinking.

Problem Constraints and Challenges

1. **Mutual Exclusion**:
    - Each fork can only be used by one philosopher at a time.
2. **Avoiding Deadlock**:
    - If all philosophers pick up the fork to their left simultaneously, none can access the fork to their right, leading to a **deadlock** situation where each philosopher waits indefinitely.
3. **Avoiding Starvation**:
    - Some philosophers might eat more frequently than others, leading to **starvation** (where some philosophers never get a chance to eat).

4. **Concurrency and Synchronization**:
    - The challenge is to design a solution where philosophers can eat and think concurrently without conflict.

Possible Solutions to Avoid Deadlock

1. **Limit the Number of Philosophers**: Allow at most four philosophers to sit at the table at any one time, ensuring at least one chopstick is available.
2. **Critical Section for Picking Up Chopsticks**: A philosopher may pick up both chopsticks only in a critical section, ensuring that the operation is atomic. This requires using semaphores to represent each chopstick:
   semaphore chopstick[5] = {1, 1, 1, 1, 1}; // Each chopstick initialized to 1 (available)
3. **Asymmetric Solution**: Odd-numbered philosophers first pick up their left chopstick, and even-numbered philosophers first pick up their right chopstick. This approach prevents a circular wait condition.

The code for a philosopher trying to eat is shown below. Each philosopher will loop indefinitely, alternating between thinking and eating:

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

The structure of philosopher *i*.

`wait(chopstick[i]);`

- The philosopher attempts to pick up the chopstick to their **left** (referred to as `chopstick[i]`).
- The `wait` operation locks the chopstick, indicating that it is in use and cannot be picked up by the neighboring philosopher on that side.
- If `chopstick[i]` is unavailable (locked), the philosopher will wait until it becomes free.

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

- After picking up the left chopstick, the philosopher attempts to pick up the **right** chopstick, referred to as `chopstick[(i+1) % 5]`.

- This `% 5` calculation ensures a circular arrangement, where Philosopher 4 (the last philosopher) wraps around to Philosopher 0, maintaining the structure of a circular table.
- With both chopsticks held, the philosopher is ready to eat.

```
/* eat for awhile */
```

- Now that both chopsticks are held, the philosopher enters the **eating** phase.
- During this phase, the philosopher consumes food for a certain period.
- As both chopsticks are occupied, no neighboring philosophers can eat at the same time, ensuring **mutual exclusion**.

```
signal(chopstick[i]);
```

- After eating, the philosopher puts down the **left** chopstick by calling `signal(chopstick[i]).`
- This operation unlocks the left chopstick, making it available for the neighboring philosopher on the left to pick up.

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

- The philosopher then puts down the **right** chopstick by calling `signal(chopstick[(i+1) % 5]).`
- This unlocks the right chopstick, allowing the neighboring philosopher on the right to access it.

```
/* think for awhile */
```

- After putting down both chopsticks, the philosopher enters the **thinking** phase, where they do not need either chopstick.
- This phase allows other philosophers to attempt to pick up the chopsticks and eat.

```
 while (true);
```

- The philosopher continuously repeats this cycle of thinking, picking up chopsticks, eating, and putting down chopsticks.

While deadlock-free solutions can be implemented, it's crucial to ensure that starvation does not occur. A starving philosopher may wait indefinitely if the scheduling policy favours other philosophers. The Dining-Philosophers Problem serves as a model for understanding synchronization issues in concurrent systems. It exemplifies the complexities of resource allocation and highlights the need for careful design to avoid deadlock and starvation while ensuring efficient use of shared resources. Solutions to this problem have implications for many real-world applications, especially in operating systems and distributed computing environments.

https://youtu.be/HHoB2t_B6MI?si=IcK-DiZ_ClHDOfvl