Process Synchronization

Cooperating Processes

- Processes can be **independent** or **cooperating**.
- Cooperating processes can **affect** or be affected by other processes.
- Cooperating processes **share data**:
  - **Inter-Process Communication (IPC)** in heavyweight processes.
  - **Shared address space** in case of threads.
  - Use of **message passing** for communication.

Why Synchronization?

- **Concurrent processes/threads** need to be protected from one another to avoid conflicts.
- Example: Protect one process's memory from being accessed by another.
- In case of cooperation, processes/threads must be synchronized to ensure they work together correctly.

Example: One thread handles input (mouse/keyboard), another handles display, and another runs programs.

Synchronization Problem

- **Concurrent access** to shared data can result in **inconsistency**.
- **Data consistency** requires mechanisms to ensure **orderly execution** of cooperating processes/threads.
- Example: If process **A** produces data and process **B** prints it, B must wait until A finishes producing the data.

Lack of Synchronization

- If processes/threads aren't synchronized, critical activities can interfere with each other.
- Proper **execution order** is crucial, especially when processes are dependent on one another.

Example: Producer-Consumer Problem

Modified Solution (Bounded Buffer)

- **Producer** and **consumer** share a buffer.
- A variable `counter` is added, initialized to 0.
  - `counter++` when an item is added.
  - `counter--` when an item is removed.

Producer (Code):

```
while (true) {
    /* produce an item in next_produced */
    while (counter == BUFFER_SIZE);
    /* add item to buffer */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Consumer (Code):

```
while (true) {
    while (counter == 0);
    /* consume the item */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```

Synchronization Issue

- The producer and consumer routines are correct **separately**, but problems arise when executed **concurrently**.
- Suppose the `counter` is 5, and both producer and consumer execute `counter++` and `counter--` at the same time. The value of `counter` can become incorrect due to race conditions.

Example of Synchronization Problem (Race Condition)

- **Concurrent execution** of `counter++` and `counter--` can lead to unexpected outcomes due to instruction interleaving.
- Consider an execution where:
    - **T0**: Producer loads `counter = 5` into `register1`.
    - **T2**: Consumer loads `counter = 5` into `register2`.
    - **T4**: Producer sets `counter = 6` (after increment).
    - **T5**: Consumer sets `counter = 4` (after decrement).
- Result: Incorrect state where `counter = 4` even though 5 items are in the buffer.

This example highlights the need for **synchronization mechanisms** to ensure the correct execution of shared data operations.

Race Condition

Definition:

- A **race condition** occurs when multiple processes **access and manipulate the same data** concurrently, and the outcome depends on the specific order of access.
- **Challenges in Debugging**:
  - Most test runs might execute fine, making race conditions hard to detect.
- **Prevention**:
  - To avoid race conditions, **concurrent processes must be synchronized**.

Reason Behind Race Condition:

- **Shared Variable Conflict**: Process **B** accesses a shared variable **before process A finishes** with it.
- Processes can either:
  - Perform **internal computations** (no race conditions).
  - Access **shared data**, leading to potential race conditions.
- The portion of the program where **shared memory** is accessed is called the **Critical Region**.
- **Race Avoidance**:
  - No two processes should be in the **critical region** at the same time.

Critical Section

Mutual Exclusion:

- **At any time, only one process** can be in the **critical section**.
- **Illustration**:
  - Process **A** enters its critical section, and process **B** is blocked until A finishes.

Critical Section Problem:

- The problem is to ensure that **when one process** is executing in its critical section (CS), **no other process** is allowed to enter their CS.
- **General Process Structure**:

  ```
  do {
      entry section
      critical section (CS)
      exit section
      reminder section
  } while (1);
  ```

- **Only Two Processes**: For simplicity, the problem often focuses on **two processes**, P0 and P1, which need to synchronize their actions when entering and leaving the critical section.

Solution to Critical-Section Problem

To prevent problems like **race conditions**, processes must carefully coordinate their access to **critical sections** where shared data is accessed or modified.

Requirements for a Solution

1. **Mutual Exclusion**:
    - Only one process should be in its critical section at a time.
2. **Progress**:
    - If no process is in its critical section, and some processes want to enter, one of them must be allowed to proceed.
3. **Bounded Waiting**:
    - There should be a bound on how long a process waits to enter its critical section, preventing **starvation**.

1. Mutual Exclusion

Mutual exclusion means that only one process can be in its critical section at any given time. This can be achieved using several methods.

Example 1: Using a Flag (Lock Variable)

This approach uses a **shared variable** (flag) to indicate whether a process is in the critical section. If the flag is `TRUE`, it means another process is in the critical section, so the current process must wait.

```
bool FLAG = TRUE;  // No process is in the critical section (initial state)

void process() {
    while (true) {
        // Entry Section: Wait until FLAG is TRUE
        while (FLAG == FALSE);  // Busy waiting

        // Critical Section
        FLAG = FALSE;           // Lock the critical section
        // Access shared resources here
        FLAG = TRUE;            // Unlock the critical section

        // Non-Critical Section
        // Do some other work
    }
}
```

Problem: This solution involves **busy waiting**, which is inefficient because the process is wasting CPU cycles while waiting for the flag.

2. Progress

The **progress** requirement ensures that if no process is in its critical section, and one or more processes want to enter, one of them should eventually enter. **No process should wait indefinitely** if it is the only one wanting access.

Example 2: Strict Alternation

In this approach, we keep track of whose turn it is to enter the critical section.

```
int turn = 1;  // Initialize turn variable

void process1() {
    while (true) {
        // Entry Section: Wait for the turn
        while (turn != 1);  // Process 1 waits if it's not its turn

        // Critical Section
        // Access shared resources
        turn = 2;           // Pass the turn to Process 2

        // Non-Critical Section
        // Do some other work
    }
}

void process2() {
    while (true) {
        // Entry Section: Wait for the turn
        while (turn != 2);  // Process 2 waits if it's not its turn

        // Critical Section
        // Access shared resources
        turn = 1;           // Pass the turn to Process 1

        // Non-Critical Section
        // Do some other work
    }
}
```

Problem: If one process is much faster than the other, the faster process will **unnecessarily wait** when the slower process is not interested in entering the critical section. This violates **progress**.

Algorithm 2

In this approach, we replace the `turn` variable with a **boolean array** `Interested[]`, where:

- `Interested[0]` indicates whether **Process 0** is interested in entering the critical section.
- `Interested[1]` indicates whether **Process 1** is interested in entering the critical section.

Key Idea:

- A process will express its interest by setting its respective `Interested[]` flag to `TRUE`.
- Before entering the critical section, a process checks whether the other process is interested (`Interested[1]` for **Process 0** and `Interested[0]` for **Process 1**).
- If the other process is interested, it waits. If not, it proceeds to the critical section

Process 0:

```
bool Interested[2] = {FALSE, FALSE};  // Initialize both processes as not int
erested

void process0() {
    while (TRUE) {
        Interested[0] = TRUE;  // Process 0 expresses its interest

        // Wait until Process 1 is not interested
        while (Interested[1] == TRUE);  // Busy waiting

        // Critical Section
        critical_section();  // Access shared resources here

        Interested[0] = FALSE;  // Process 0 is no longer interested

        // Non-Critical Section
        noncritical_section();  // Do some other work
    }
}
```

Process 1:

```
bool Interested[2] = {FALSE, FALSE};  // Initialize both processes as not int
erested

void process1() {
    while (TRUE) {
        Interested[1] = TRUE;  // Process 1 expresses its interest

        // Wait until Process 0 is not interested
        while (Interested[0] == TRUE);  // Busy waiting
```

```
        // Critical Section
        critical_section();  // Access shared resources here

        Interested[1] = FALSE;  // Process 1 is no longer interested

        // Non-Critical Section
        noncritical_section();  // Do some other work
    }
}
```

How It Works:

1. **Interest Expression**:
    - Each process sets its respective `Interested[]` flag to `TRUE` to signal its intent to enter the critical section.
2. **Mutual Exclusion**:
    - Before entering the critical section, a process checks if the other process is interested. If the other process is interested (i.e., its `Interested[]` flag is `TRUE`), it waits.
    - Once the other process has finished (i.e., its `Interested[]` flag is `FALSE`), the current process enters the critical section.
3. **Exit from Critical Section**:
    - After completing the critical section, the process resets its `Interested[]` flag to `FALSE`, signaling that it is no longer interested.
4. **Non-Critical Section**:
    - The process can now perform any other non-critical work before it loops back and potentially re-enters the critical section.

Potential Issue: Deadlock

While this algorithm resolves some issues from strict alternation, it introduces the potential for **deadlock**. If both processes set their `Interested[]` flags to `TRUE` at the same time, they would both be stuck in their **while loops**, waiting for the other to become uninterested.

3. Bounded Waiting

Bounded waiting ensures that a process is not **starved** and will eventually enter its critical section. We can achieve this by keeping track of whether each process is interested in entering the critical section.

Example 3: Peterson's Algorithm

Peterson's Solution is a well-known algorithm for solving the **Critical Section Problem** for two processes in a **software-based** manner. This solution is a combination of the two previous approaches: it uses both a `turn` variable and an `interested` (or `flag`) array to manage process access to the critical section. It ensures **mutual exclusion, bounded waiting,** and **progress**, which are the essential requirements for solving the critical section problem.

In Peterson's Solution:

- The `turn` variable is used to keep track of whose turn it is to enter the critical section.
- The `interested[]` array (or `flag[]`) signals whether each process is interested in entering the critical section. If `interested[i]` is `TRUE`, it means that **Process i** is ready to enter.

Steps:

1. **Set Interest**: Each process indicates its interest by setting its respective `interested[]` flag to `TRUE`.
2. **Set Turn**: Each process then sets the `turn` variable to give the other process a chance to enter the critical section if it is also interested.
3. **Wait**: A process will only enter the critical section if either:
    - The other process is **not interested**.
    - The other process is **willing to yield** by setting `turn` to the current process.
4. **Critical Section**: Once the condition is met, the process enters the critical section.
5. **Reset Interest**: After leaving the critical section, the process sets its `interested[]` flag to `FALSE`, indicating it no longer needs access.

Code Example of Peterson's Solution

Let's break down the code for both **Process 0** and **Process 1**:

```
#include <stdbool.h>

int turn;                    // Keeps track of whose turn it is
bool interested[2] = {false, false};  // Flags indicating interest of each pr
ocess

void process0() {
    while (true) {
        interested[0] = true;    // Process 0 expresses interest
        turn = 1;                // Gives turn to Process 1

        // Wait while Process 1 is interested and it's their turn
        while (interested[1] && turn == 1);

        // Critical Section
        critical_section();      // Process 0 accessing shared resources

        interested[0] = false;   // Process 0 no longer interested

        // Non-Critical Section
```

```
        noncritical_section();   // Process 0 doing other work
    }
}

void process1() {
    while (true) {
        interested[1] = true;    // Process 1 expresses interest
        turn = 0;                // Gives turn to Process 0

        // Wait while Process 0 is interested and it's their turn
        while (interested[0] && turn == 0);

        // Critical Section
        critical_section();      // Process 1 accessing shared resources

        interested[1] = false;   // Process 1 no longer interested

        // Non-Critical Section
        noncritical_section();   // Process 1 doing other work
    }
}
```

Explanation of Key Properties

1. Mutual Exclusion:

- Only one process can enter the critical section at a time because each process waits until either:
  - The other process is not interested.
  - The other process has yielded the turn.
- This ensures that the critical section is not accessed by both processes simultaneously.

2. Bounded Waiting:

- Peterson's Solution provides a bound on how long a process has to wait before it can enter the critical section.
- If a process is interested, it will eventually get a chance to enter the critical section because the `turn` variable alternates between the two processes, giving each a fair chance.

3. Progress:

- If one process is not interested in the critical section, the other can enter without waiting.
- This avoids deadlock, as processes do not hold each other in an indefinite wait.

Example Run-through

1. **Process 0 wants to enter**:
   - Sets `interested[0] = TRUE` and `turn = 1`, allowing **Process 1** the chance to enter if it's interested.
   - If `interested[1]` is `FALSE` (Process 1 is not interested) or `turn != 1`, **Process 0** enters the critical section.
2. **Process 1 wants to enter at the same time**:
   - Sets `interested[1] = TRUE` and `turn = 0`, giving **Process 0** the chance to enter.
   - Since both processes are interested, only the process with the turn (determined by `turn` variable) can enter. This allows only one process at a time to proceed to the critical section.

Limitations

1. **Two-Process Limitation**:

   Peterson's Solution works only for two processes, as the `turn` variable is binary. Expanding this to more than two processes is complex and typically not feasible with this approach.
2. **Practicality**:

   Due to modern hardware and compiler optimizations, Peterson's Solution may not always work reliably on multiprocessor systems. It's mainly used as a theoretical foundation to understand how synchronization can be managed in two-process systems.

Semaphores in Operating Systems

A **semaphore** is a synchronization tool used to control access to a common resource by multiple processes in a concurrent system. Semaphores are particularly useful when dealing with critical sections in multi-threaded or multi-process applications.

Basic Concepts of Semaphores

1. **Semaphore (S)**: It is an integer variable that, besides initialization, can only be modified by two atomic operations: `wait()` and `signal()`.
2. **Working Mechanism**:
   - When a process performs `wait(S)`, it checks the value of S. If S is positive, it decrements it and proceeds. If S is zero or negative, the process waits (blocks) until S becomes positive.
   - When a process performs `signal(S)`, it increments the value of S, which may allow a waiting process to proceed.

Semaphore Operations

`wait(S)` Operation

```
void wait(S) {
    while (S <= 0);  // Busy waiting if S is not positive
    S--;             // Decrement S if it's positive, allowing the process to
```

```
enter
}
```

`signal(S)` Operation

```
void signal(S) {
    S++;  // Increment S, signaling that the resource is available
}
```

> Note: Busy waiting is a drawback of this simple implementation since the process continuously checks if S is positive, consuming CPU resources without performing any useful work.

Using Semaphores to Synchronize Processes

Consider two processes, P_1 and P_2, where a mutual exclusion semaphore, **mutex**, is initialized to 1. This ensures that only one process can enter the **critical section** at a time.

Process Code Using Semaphore

For **Process P_i**:

```
do {
    wait(mutex);       // Enter critical section
    // Critical section
    signal(mutex);     // Exit critical section
    // Remainder section
} while (true);
```

For **Process P_j**:

```
do {
    wait(mutex);       // Enter critical section
    // Critical section
    signal(mutex);     // Exit critical section
    // Remainder section
} while (true);
```

In this setup, only one process can enter the critical section because the other process will be blocked by `wait(mutex)` if **mutex** is 0.

Example: Synchronizing Processes with Semaphores

Assume we have **n processes** P_1, P_2, ..., P_n that use a semaphore S initialized to 1. Each process does the following:

```
do {
    wait(S);            // Request access to critical section
    // Critical section code
    signal(S);          // Release access to critical section
} while (true);
```

Here, only one process can access the critical section at a time because the semaphore `S` ensures mutual exclusion.

Avoiding Busy Waiting with Blocking and Wakeup

In systems with **busy waiting**, a process continuously checks if a semaphore allows entry, leading to wasted CPU cycles. To avoid this:

1. **block()**: The process is placed into a waiting queue if it cannot enter.
2. **wakeup()**: A process is removed from the waiting queue and moved to the ready state when it can proceed.

Semaphore with No Busy Waiting Implementation

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

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

void signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process from S->list;
        wakeup(P);
    }
}
```

Issues with Semaphores: Deadlock and Starvation

1. **Deadlock**: Occurs when two or more processes wait indefinitely for resources held by each other.
   - Example: Two semaphores **S** and **Q** initialized to 1.

```
P0:
wait(S);
wait(Q);
// Critical section
signal(Q);
signal(S);

P1:
wait(Q);
wait(S);
// Critical section
signal(S);
signal(Q);
```

Here, if P_0 holds **S** and P_1 holds **Q**, both will wait indefinitely for each other, causing a deadlock.

2. **Starvation**: A process may never get access to a critical section if others keep preempting it, particularly in a **Last-In-First-Out (LIFO)** queue implementation.

Classical Synchronization Problems in Operating Systems

In operating systems, synchronization is essential when multiple processes or threads interact with shared resources. Using **semaphores**, we can manage access to critical sections, ensuring threads don't interfere with each other, leading to issues like race conditions, deadlock, or starvation.

1. Signaling

In the **Signaling** problem, one thread needs to complete a task before another thread can start its task. This is achieved by having a semaphore initialized to `0`, so the second thread waits until the first thread signals completion.

Example:
Thread A reads a line (
`a1`), and Thread B displays the line (`b1`), ensuring `a1` completes before `b1` begins.

```
Semaphore sem = 0; // Initial value 0

// Thread A
void threadA() {
    a1();              // Read a line from file
    signal(sem);       // Signal completion
}

// Thread B
void threadB() {
    wait(sem);         // Wait for Thread A to complete
```

```
    b1();             // Display the line
}
```

2. Rendezvous Problem

The **Rendezvous** problem involves two threads waiting for each other at specific points, ensuring a particular execution order.

Goal: Ensure `a1` happens before `b2` and `b1` happens before `a2`, while the order of `a1` and `b1` is flexible.

Solution:

1. Define two semaphores (`aArrived` and `bArrived`) initialized to `0`.
2. Thread A signals `aArrived` after `a1`, allowing `b2` in Thread B to proceed.
3. Thread B signals `bArrived` after `b1`, allowing `a2` in Thread A to proceed.

```
Semaphore aArrived = 0;
Semaphore bArrived = 0;

// Thread A
void threadA() {
    a1();                  // Execute a1
    signal(aArrived);      // Signal aArrived for Thread B
    wait(bArrived);        // Wait for Thread B to signal bArrived
    a2();                  // Execute a2
}

// Thread B
void threadB() {
    b1();                  // Execute b1
    signal(bArrived);      // Signal bArrived for Thread A
    wait(aArrived);        // Wait for Thread A to signal aArrived
    b2();                  // Execute b2
}
```

3. Mutex Problem

The **Mutex Problem** ensures that only one thread accesses a shared variable at any given time. This mutual exclusion is achieved by initializing a semaphore `mutex` to `1`.

Example:
Suppose we have a shared variable
`count` that we want to protect from concurrent access by multiple threads.

```
Semaphore mutex = 1; // Initial value 1 for mutual exclusion
int count = 0;       // Shared variable
```

```
void threadFunction() {
    while (true) {
        wait(mutex);        // Enter critical section
        count++;            // Critical section (update shared variable)
        signal(mutex);      // Exit critical section
        // Other non-critical operations
    }
}
```

4. Multiplex

The **Multiplex** problem is a generalization of the mutex problem, where a limited number of threads are allowed in the critical section simultaneously.

Goal: Allow `n` threads to enter the critical section at a time.

Solution: Initialize a semaphore `multiplex` to `n` to limit the maximum concurrent access to the critical section.

```
const int n = 3;            // Maximum number of concurrent threads
Semaphore multiplex = n;    // Initial value of semaphore set to `n`

void threadFunction() {
    while (true) {
        wait(multiplex);    // Enter if less than `n` threads in critical sect
ion
        // Critical section
        signal(multiplex); // Exit, allowing another thread in
        // Non-critical operations
    }
}
```

Applications of Semaphores

Semaphores are widely used in operating systems to manage concurrent processes and prevent issues in multi-threading environments. Key types include:

1. **Binary Semaphores**: Used for mutual exclusion (mutex).
2. **Counting Semaphores**: Used for managing multiple identical resources.

Applications of Semaphores

1. **Critical Section Problem**: Ensure only one thread accesses a shared resource at a time.
2. **Deciding Order of Execution**: Coordinate between threads to enforce a specific order.
3. **Resource Management**: Manage access to limited resources, e.g., multiple printers.

Classical Problems of Synchronization

1. Bounded-Buffer (Producer-Consumer) Problem

In the **Bounded-Buffer Problem**, we have a shared buffer with a finite number of slots. A producer thread adds items to the buffer, while a consumer thread removes them. We use semaphores to synchronize access to the buffer.

Shared Data:

- `full` semaphore: Counts filled slots, initially `0`.
- `empty` semaphore: Counts empty slots, initially `n`.
- `mutex` semaphore: Ensures mutual exclusion, initially `1`.

```
Semaphore full = 0;
Semaphore empty = n;
Semaphore mutex = 1;

void producer() {
    while (true) {
        // Produce an item in nextp
        wait(empty);         // Wait for an empty slot
        wait(mutex);         // Lock buffer access
        // Add item to buffer
        signal(mutex);       // Release buffer
        signal(full);        // Signal a filled slot
    }
}

void consumer() {
    while (true) {
        wait(full);          // Wait for a filled slot
        wait(mutex);         // Lock buffer access
        // Remove item from buffer
        signal(mutex);       // Release buffer
        signal(empty);       // Signal an empty slot
        // Consume the item
    }
}
```

2. Readers-Writers Problem

The **Readers-Writers Problem** deals with synchronization between multiple readers and one writer accessing a shared resource.

- **Readers**: Can read concurrently.

- **Writers**: Require exclusive access.

There are two versions:

1. **First Reader-Writers Problem (Reader's Precedence)**: Allows readers to access the resource before writers, which can lead to **writer starvation**.

```
Semaphore mutex = 1;
Semaphore wrt = 1;
int readCount = 0;

void reader() {
    while (true) {
        wait(mutex);
        readCount++;
        if (readCount == 1) wait(wrt);  // First reader locks the writer
        signal(mutex);

        // Read the resource

        wait(mutex);
        readCount--;
        if (readCount == 0) signal(wrt);  // Last reader unlocks the write
r
        signal(mutex);
    }
}

void writer() {
    while (true) {
        wait(wrt);  // Wait until no readers
        // Write to the resource
        signal(wrt);
    }
}
```

2. **Second Readers-Writers Problem (Writer's Precedence)**: Prioritizes writers over readers, reducing the risk of writer starvation.

```
Semaphore mutex1 = 1, mutex2 = 1;
Semaphore rd = 1, wrt = 1;
int readCount = 0, writeCount = 0;

void reader() {
    while (true) {
        wait(rd);
```

```
        wait(mutex1);
        readCount++;
        if (readCount == 1) wait(wrt);  // First reader locks the writer
        signal(mutex1);
        signal(rd);

        // Read the resource

        wait(mutex1);
        readCount--;
        if (readCount == 0) signal(wrt);  // Last reader unlocks the write
r
        signal(mutex1);
    }
}

void writer() {
    while (true) {
        wait(mutex2);
        writeCount++;
        if (writeCount == 1) wait(rd);  // First writer locks readers
        signal(mutex2);

        wait(wrt);  // Exclusive access to the resource
        // Write to the resource
        signal(wrt);

        wait(mutex2);
        writeCount--;
        if (writeCount == 0) signal(rd);  // Last writer unlocks readers
        signal(mutex2);
    }
}
```

3. Dining Philosophers Problem

The Dining Philosophers Problem is a classical synchronization challenge that illustrates the difficulty of allocating resources (forks) among processes (philosophers) without causing deadlocks or starvation.

Problem Description

- **Scenario**:
  - Five philosophers sit around a table alternating between eating and thinking.
  - Each philosopher needs **two forks** to eat, but only **five forks** are available.
- **Goal**: Ensure no deadlock (where philosophers wait indefinitely) or starvation (where some never eat).

Key Concepts

- Each philosopher is represented as a **process**.
- **Forks** are modeled using an array of semaphores (`fork[i]`), initialized to 1 (indicating availability).

Code for Dining Philosophers (Basic Approach)

```
Pi() {
   while (TRUE) {
      think;
      wait(fork[i]);
      wait(fork[(i+1) % 5]);
      eat;
      signal(fork[(i+1) % 5]);
      signal(fork[i]);
   }
}
```

- **Issue**: Deadlock occurs if each philosopher picks their left fork first.

Solution to Avoid Deadlock

- Introduce an additional semaphore **T** to limit the number of philosophers at the table to **4** (ensuring at least one fork remains free).

- **Initialization**:

```
T.count = 4;
```

Improved Code

```
Pi() {
   while (TRUE) {
      think;
      wait(T);
      wait(fork[i]);
      wait(fork[(i+1) % 5]);
      eat;
      signal(fork[(i+1) % 5]);
      signal(fork[i]);
      signal(T);
   }
}
```

Additional Strategies

- A philosopher can only pick up both forks **simultaneously**.
- Philosophers in **even positions** pick the right fork first, then the left, while those in **odd positions** pick the left fork first, then the right.

Bakery Algorithm

The Bakery Algorithm is a mutual exclusion algorithm designed to ensure that multiple processes can safely access a **critical section** (CS) in a distributed system. It is particularly useful for systems with **n processes** where each process must wait its turn, just like customers in a bakery take numbered tickets and wait for their number to be called.

Key Concepts

- **Critical Section (CS)**: A section of code that can be executed by only one process at a time.
- **Ticket System**: Each process gets a **unique ticket number**, similar to a bakery counter. The process with the smallest ticket number enters the CS.
- **Monotonic Ordering**: Ticket numbers increase but are not reused, ensuring fairness.

Steps of the Bakery Algorithm

1. **Taking a Ticket**:
    - When a process wants to enter the CS, it selects a ticket number larger than any currently held by other processes.

- If two processes get the same ticket, they resolve the tie by comparing their process IDs (lower ID wins).

2. **Checking Eligibility**:
   - A process checks if any other process with a smaller ticket number (or same number but lower ID) is waiting.
   - If no such process exists, the current process enters the CS.
3. **Releasing the Ticket**:
   - Once the process finishes its task in the CS, it releases its ticket by resetting it to 0, allowing other processes to enter.

Algorithm Implementation

Let `n` be the number of processes, `choosing[]` be a flag array, and `ticket[]` be the ticket array.

```
choosing[i] = true;
ticket[i] = max(ticket[0], ticket[1], ..., ticket[n-1]) + 1;
choosing[i] = false;

for (j = 0; j < n; j++) {
   while (choosing[j]); // Wait if the process is choosing a ticket
   while (ticket[j] != 0 &&
          (ticket[j] < ticket[i] ||
          (ticket[j] == ticket[i] && j < i))); // Wait if another process has a smaller ticket or same ticket with lower ID
}

// Critical Section

ticket[i] = 0; // Release ticket after exiting CS
```

Example

1. **Process 1**: Takes ticket 3.
2. **Process 2**: Takes ticket 5.
3. **Process 3**: Takes ticket 4.
4. **Order of Execution**: Process 1 → Process 3 → Process 2.

Advantages

- **Fairness**: No process is starved; all processes get a chance based on ticket order.
- **Simplicity**: Easy to understand and implement for small systems.

Key Properties

- **Mutual Exclusion**: Ensures only one process is in the CS at any time.
- **Bounded Waiting**: A process will not wait indefinitely to enter the CS.
- **Progress**: If no process is in the CS, one of the waiting processes will eventually enter.