Concurrency

Module 4

Dr. Naveenkumar J

Associate Professor,

PRP- 217 - 4

Inter-Process Communication (IPC)

- A mechanism provided by operating systems that allows processes to communicate with each other.
- This is essential for processes to **exchange data**, **synchronize their actions**, and **coordinate activities** while running in parallel.
- IPC ensures **efficient operation and resource sharing** in multi-processing environments.

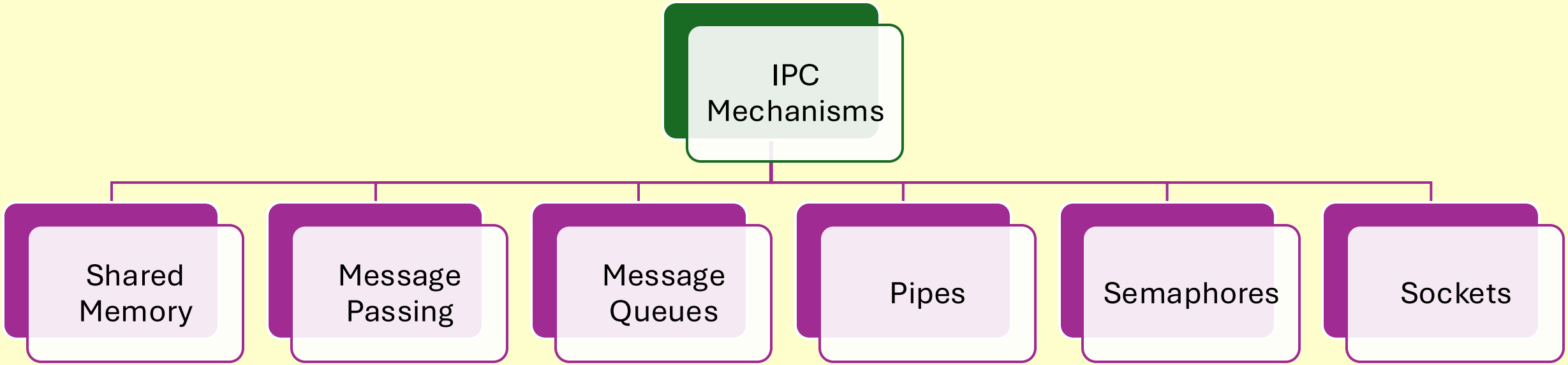
Inter-Process Communication (IPC)

- **Why is IPC Needed?**
  - Sharing data between processes that have related tasks.
  - Coordinating activities so processes can work together.
  - Managing resources among multiple concurrent programs.
  - Achieving modularity by separating functionalities into different processes.

Inter-Process Communication (IPC)

- Example
  - Copy and Paste (Using the Clipboard) - When you copy text from your web browser (Process 1) and paste it into a word processor like Microsoft Word (Process 2), you are using an IPC mechanism called the clipboard.

Inter-Process Communication (IPC)

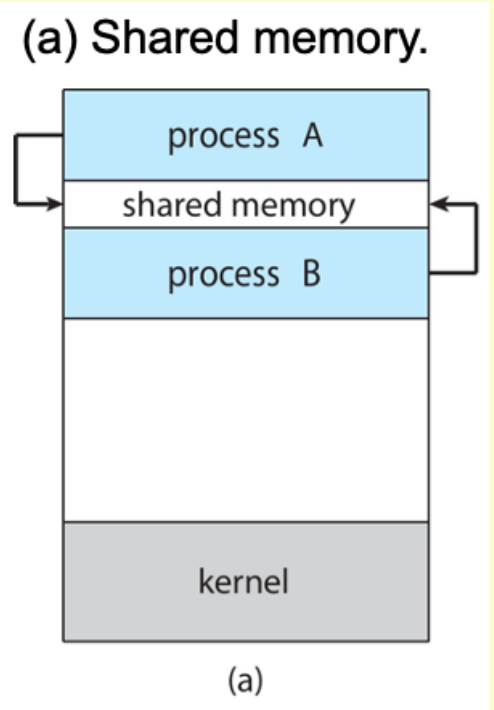


IPC Mechanisms – Shared Memory

- It allows **multiple independent processes** to **access** the same **block of physical memory**, enabling them to **exchange large amounts of data** with minimal overhead.
- **Working**
  - Shared memory works by creating a special segment in the computer's RAM that the operating system maps into the virtual address space of two or more processes.
  - This means that although each process "sees" the memory at a potentially different address, they are all reading from and writing to the same physical location.

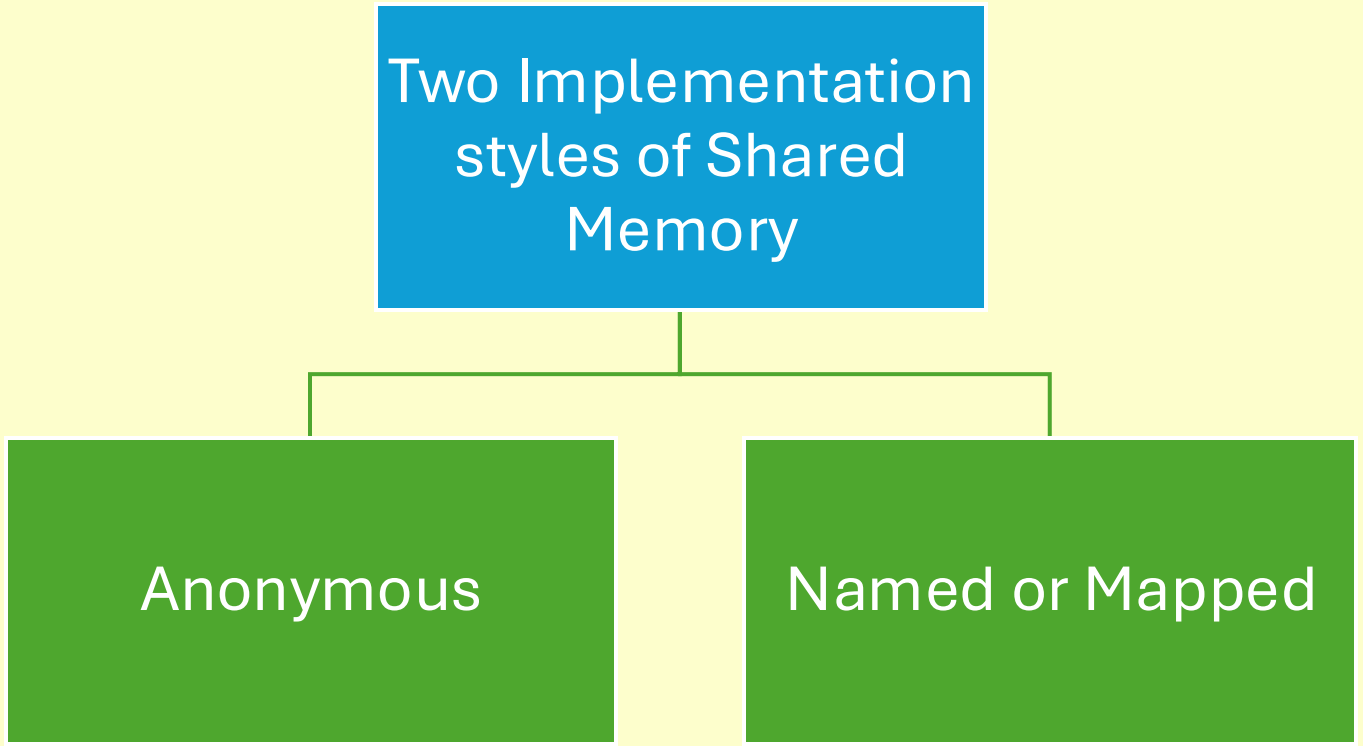
IPC Mechanisms – Shared Memory

- It allows **multiple independent processes** to **access** the same **block of physical memory**, enabling them to **exchange large amounts of data** with minimal overhead.
- **Working**
  - Shared memory works by creating a special segment in the computer's RAM that the operating system maps into the virtual address space of two or more processes.
  - This means that although each process "sees" the memory at a potentially different address, they are all reading from and writing to the same physical location.



IPC Mechanisms – Shared Memory

- **How is the shared memory segment identified and whether it is associated with a file on the file system?**
- Two Implementation styles of Shared Memory
  - Anonymous
  - Named or Mapped



IPC Mechanisms – Shared Memory

Anonymous	Mapped or Named
Anonymous shared memory is a region of memory that is **not associated with any file** in the filesystem. It is "anonymous" because it doesn't have a name or path that other, unrelated processes can use to find it.	Mapped shared memory, also known as memory-mapped files, is a region of memory that is directly associated with a specific file on the file system. Processes communicate by mapping this same file into their respective address spaces
This type of shared memory is typically used between a parent process and its child processes. The parent creates the anonymous segment, and when it forks a child, the child inherits the file descriptor or handle to that memory.	Processes access the shared memory by using a common name—the path to the file. This allows completely unrelated processes to communicate, if they **both know the file path and have the necessary permissions**
Since there is no public name, it is more secure. Unrelated processes cannot easily discover and attach to the memory segment, preventing unauthorized access	It can be used to share data between processes that are not part of the same family
It is ideal for tightly coupled processes, such as a main application and its worker processes, where the parent can directly pass the memory handle to its children.	It is perfect for applications that need to share data between unrelated processes, or when the shared data needs to be saved and persist beyond the lifetime of the processes.

IPC Mechanisms – Shared Memory

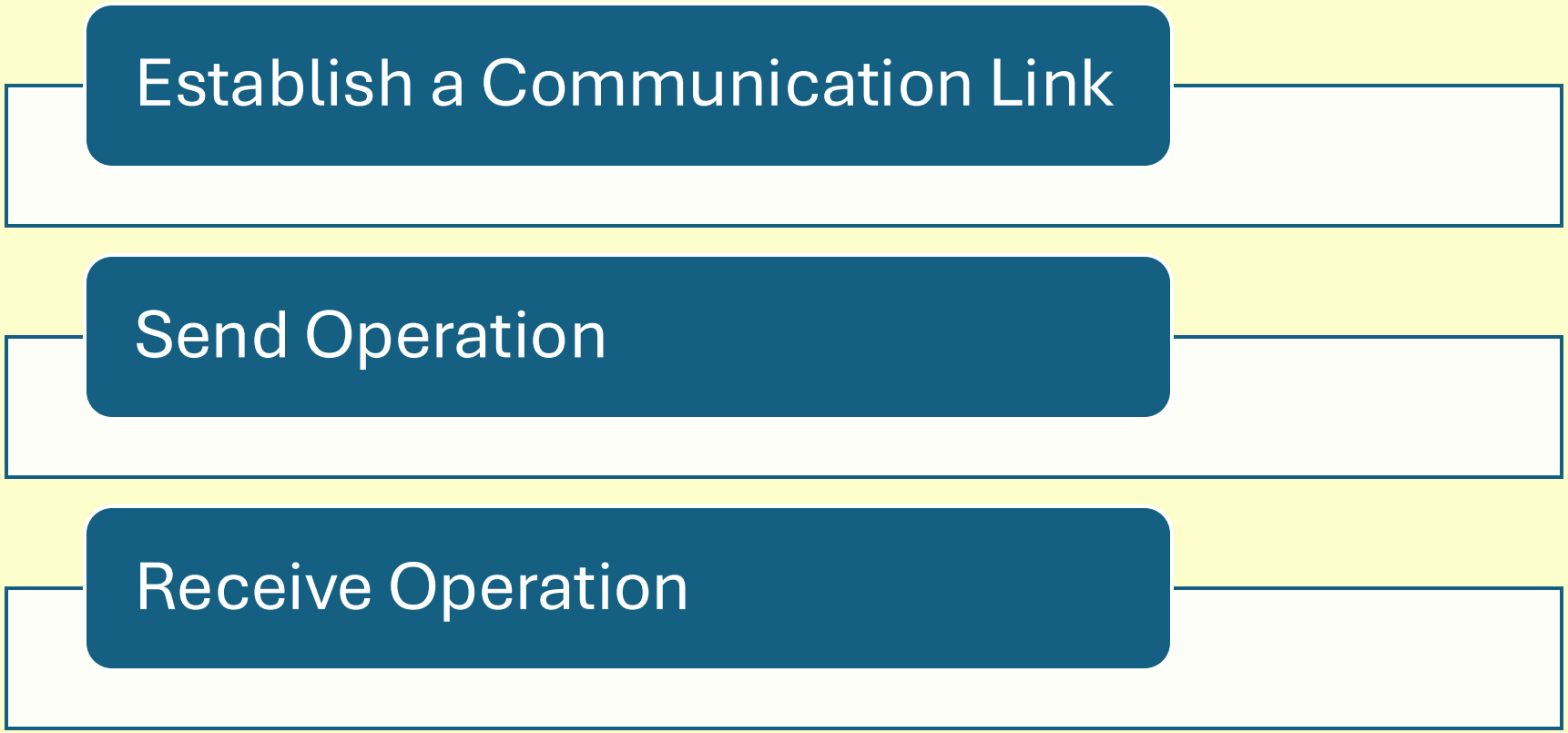
Merits	Demerits
Speed: Since processes are accessing the same physical memory, there is no need for the kernel to mediate or copy data between them. **This avoids system call overhead**.	**Synchronization Complexity**: Processes must manually manage access to the shared memory to prevent race conditions and data corruption. This requires implementing complex synchronization primitives like semaphores or mutexes, which can be difficult to get right.
Efficiency: Data is not duplicated for each process; a single copy is shared among all, reducing overall memory consumption.	**Security Risks**: If not properly secured, a shared memory segment could be accessed by unauthorized processes, leading **to data leaks or corruption**.
Large Data Volumes: It is ideal for transferring large amounts of data, such as video frames, large datasets, or scientific computing results, where other methods would be too slow.	**Potential for Deadlocks**: Improperly implemented synchronization can lead to deadlocks, where two or more processes are **stuck waiting for each other** to release a lock on the shared memory, bringing the system to a halt.

IPC Mechanisms – Message Passing

- Processes communicate without sharing the same address space. Instead, they exchange discrete messages managed by the operating system.
- Think of it as processes talking to each other by sending letters through a postal service (the OS) rather than writing on a shared whiteboard
- The implementation relies on the operating system's kernel to act as an intermediary.

IPC Mechanisms – Message Passing

- How Message Passing is Implemented in an OS?



IPC Mechanisms – Message Passing

- How Message Passing is Implemented in an OS?
  - **Establish a Communication Link**: Before communication can begin, a link must be established between the processes. This can be:
    - **Direct Communication**: The sender and receiver explicitly name each other (e.g., **send(Process_B, message))**. This creates a **one-to-one link**.
    - **Indirect Communication**: Messages are sent to and received from a **central "mailbox" or "port**." Multiple processes can use the **same mailbox**, allowing for more flexible, **many-to-many communication**.

IPC Mechanisms – Message Passing

- The **send Operation**:
  - A process packages its data into a message and executes a **send system call**. The kernel takes control, copies the message from the sender's private memory into a secure **kernel buffer**, and **queues it for delivery**.
- The **receive Operation**:
  - The receiving process executes a **receive system call**. The **kernel then copies the message from its buffer into the receiver's private memory space**.

IPC Mechanisms – Message Passing

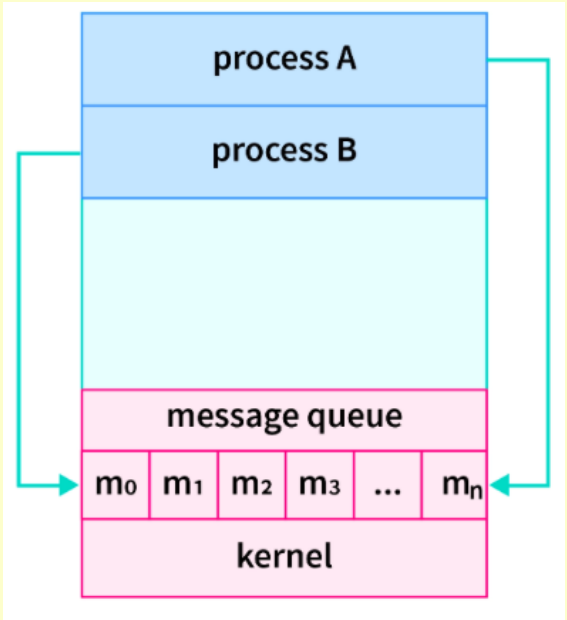
- This process can be either synchronous (blocking) or asynchronous (non-blocking)
  - **Synchronous (Blocking)**: The sender is blocked until the receiver has successfully received the message. This ensures the message was delivered but can make the sender wait.
  - **Asynchronous (Non-Blocking)**: The sender hands the message to the OS and continues its own execution immediately, without waiting for the receiver. This is more efficient for the sender but provides no guarantee of when (or if) the message is read.

IPC Mechanisms – Message Passing

Merits	Demerits
Simpler to Implement for Programmers: It avoids the complexities of synchronization (like mutexes and semaphores) because the kernel handles the message transfer. This makes it easier to write correct concurrent code.	**Slower Performance**: The involvement of the kernel in every send and receive operation adds significant overhead. Data must be copied from the sender's memory to the kernel, and then from the kernel to the receiver's memory. **This is much slower than directly accessing shared memory**.
Enhanced Safety and Isolation: Since processes don't share memory, there is no risk of one process accidentally corrupting another's data. Each process operates in its own protected address space.	**Overhead and Latency:** The packaging, sending, and unboxing of messages introduces latency, which can be a problem for performance-critical applications.
Ideal for Distributed Systems: Because it doesn't require a shared physical memory, message passing is the natural choice for communication between processes running on different computers across a network.	**Not Ideal for Large Data**: The copying process makes it inefficient for transferring very large amounts of data compared to shared memory.

IPC Mechanisms – Message Queues

- It allows processes to **communicate asynchronously** by exchanging messages through a **shared queue structure managed** by the **operating system's kernel**
- The sender doesn't have to wait for the receiver to be ready, and the receiver doesn't need to know anything about the sender, only where the mailbox is.



IPC Mechanisms – Message Queues

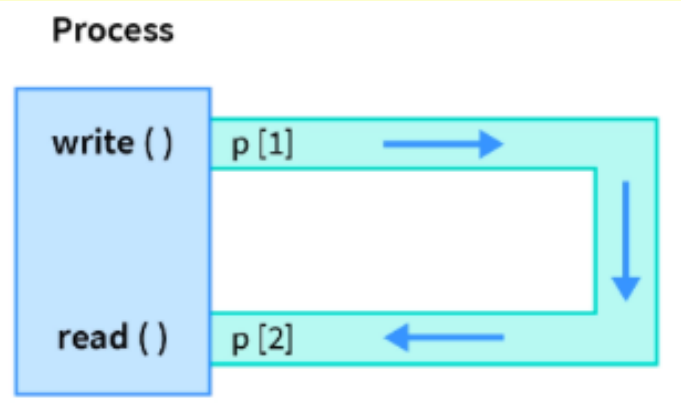
- The queue acts as an intermediary, allowing Sender and Receiver processes to operate independently.
- A process can send a message and continue with its work without waiting for a response. The message will be stored securely in the queue until a receiver is ready to process it.
- The system/kernel preserves message boundaries, ensuring that what one process sends is exactly what another receives.
- A single queue can have multiple senders and multiple receivers, making it a flexible tool for complex application architectures.

IPC Mechanisms – Message Queues

- **POSIX Message Queues:** POSIX queues are **identified by names** (like file paths) and offer additional features like **message prioritization and asynchronous notifications** when a new message arrives.

IPC Mechanisms – PIPES

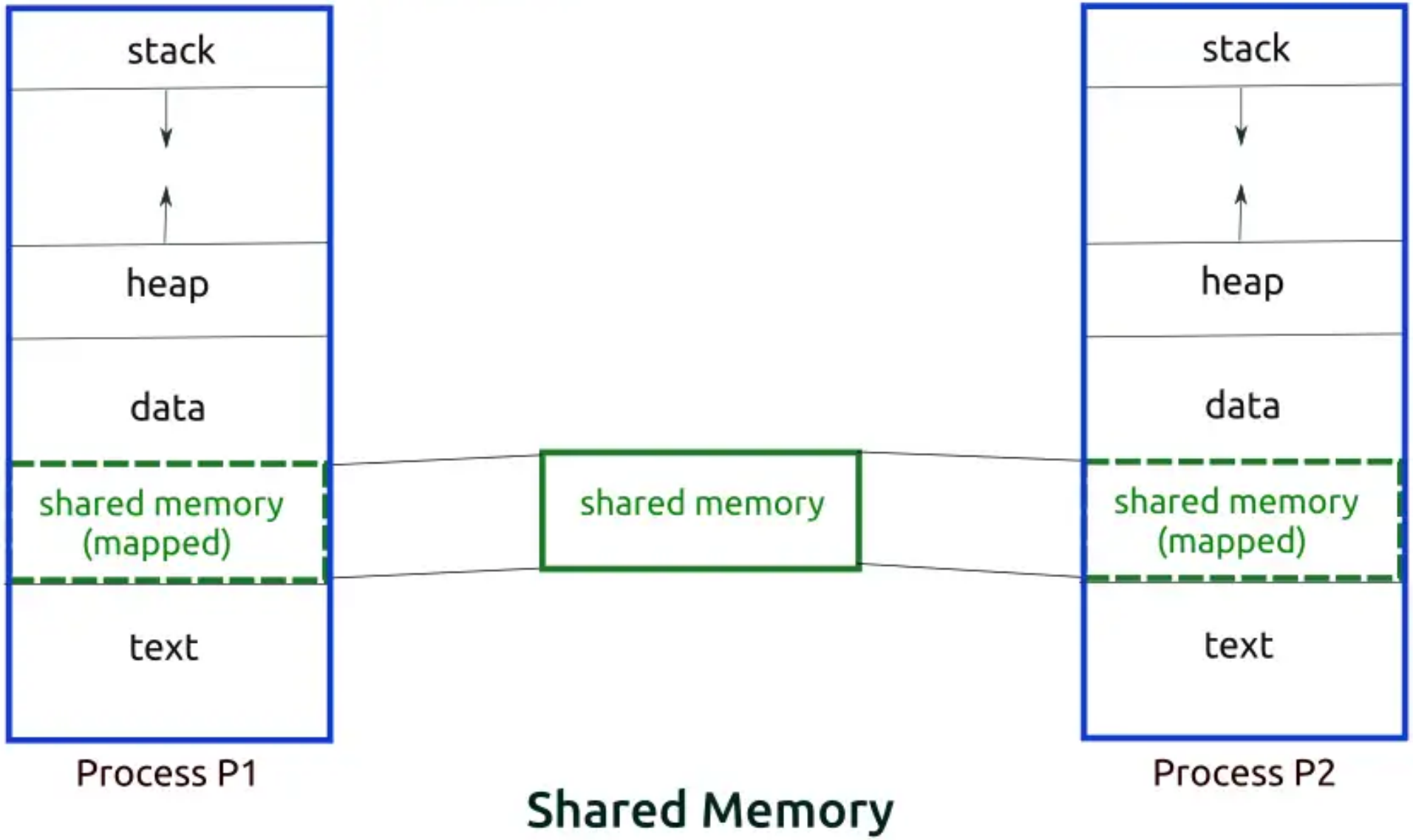
- They create a **unidirectional communication channel** that allows the **output of one process** to be **fed directly as the input to another process**.
- Pipes are implemented and managed by the operating system's kernel.
- A single process (the parent) makes a pipe() system call. The OS doesn't create a file on disk; instead, it creates a small, in-memory buffer and returns two file descriptors to the process:
  - A file descriptor for the read end of the pipe.
  - A file descriptor for the write end of the pipe.



IPC Mechanisms – PIPES

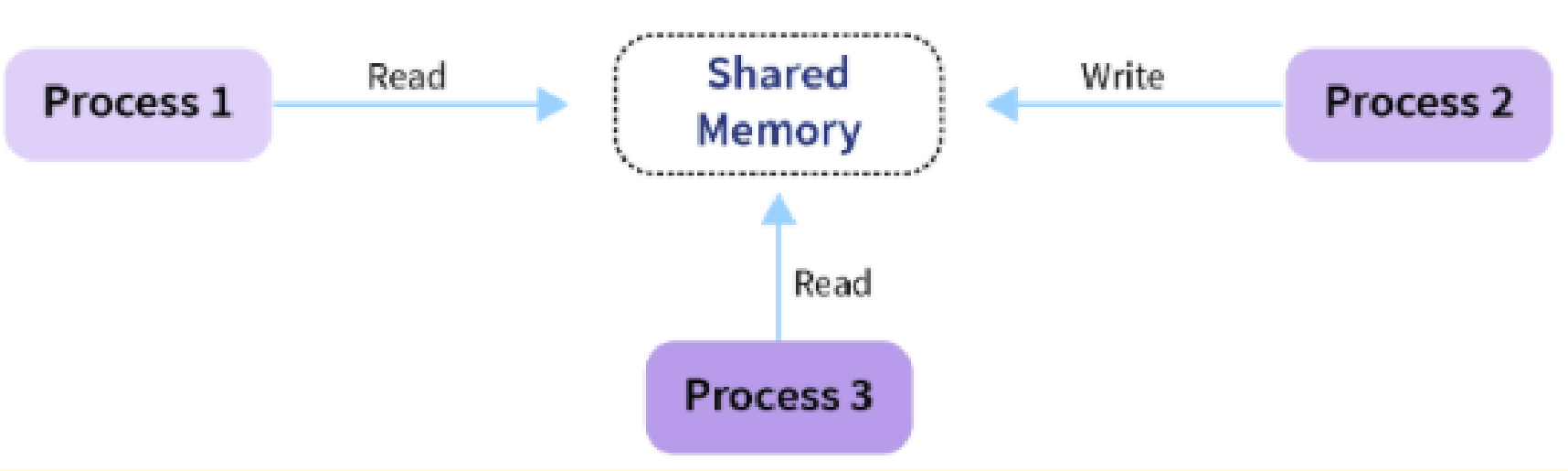
- The sending process uses a standard write() system call on its file descriptor, and the receiving process uses a read() system call on its file descriptor.
- The OS kernel handles the data transfer through the in-memory buffer, ensuring that the reader process waits if the pipe is empty and the writer process waits if the pipe's buffer is full.

Summary – Shared Memory



Synchronization

- Synchronization is the coordination of multiple processes or threads so that they can work together without interfering with each other.
- It ensures that when multiple processes access shared resources (like variables, files, or devices), they do so in an orderly manner to prevent conflicts or errors.



Synchronization – Critical Section

- The "**critical section**" is the **part of the program** where the **shared resource is accessed**.
- **Critical Section Problem**
  - The critical section problem occurs when multiple processes need to access and modify shared data simultaneously
  - The problem is to design a way so that only one process can be inside its critical section at a time, preventing inconsistent or corrupted data.
  - If two or more processes enter their critical sections simultaneously, it can cause race conditions, where the outcome depends on the unpredictable timing of processes.

Critical Section – Race Condition

- If two or more processes or threads run increment() simultaneously, the following can happen:
  - Thread A reads shared_counter as 0.
  - Before Thread A writes back, Thread B also reads shared_counter as 0.
  - Both increment to 1 independently.
  - Both write 1 back to shared_counter.
  - Instead of two increments making shared_counter 2, it stays 1 due to simultaneous access.
- This unexpected behavior is a **race condition** because threads are "racing" to access and modify the shared data.

```
int shared_counter = 0;  // shared resource

void increment()
{
    // Critical Section starts
    int temp = shared_counter; // read shared value
    temp = temp + 1;           // modify value
    shared_counter = temp;     // write back to shared value
    // Critical Section ends
}
```

- The block of code:

```
int temp = shared_counter;
temp = temp + 1;
shared_counter = temp;
```

is the critical section because **it accesses and modifies the shared variable**.

The problem is that multiple threads can enter this critical section at the same time.

This leads to incorrect or inconsistent values like the race condition above.

Critical Section – Race Condition

```
int shared_var = 0; // Shared variable
```

Thread 1 / Process 1	Thread 2 / Process 2	Main Program

void* process1(void* arg) {
// Critical Section starts
int temp = shared_var; // Read shared variable
temp = temp + 1; // Modify
shared_var = temp; // Write back
printf("Process 1 updated shared_var to %d\n", shared_var);
// Critical Section ends
return NULL;
	void* process2(void* arg) {
	// Critical Section starts
	int temp = shared_var; // Read shared variable
	temp = temp + 1; // Modify
	shared_var = temp; // Write back
	printf("Process 2 updated shared_var to %d\n", shared_var);
	// Critical Section ends
	return NULL;
		pthread_t t1, t2;
		pthread_create(&t1, NULL, process1, NULL);
		pthread_create(&t2, NULL, process2, NULL);
		
		pthread_join(t1, NULL);
		pthread_join(t2, NULL);
		
		printf("Final value of shared_var: %d\n", shared_var);

- shared_var is accessed by both processes, representing critical shared data.
- The lines reading, modifying, and writing shared_var form the critical section that must be accessed atomically.
- Without synchronization, both threads could read the same original value before writing, causing one increment to be lost.
- The final value of shared_var may incorrectly be 1 instead of 2, due to overlapping critical sections and race condition.

Critical Section – Race Condition - Analogy

- Imagine you and a family member share a joint bank account with a current balance of $1,000. On the same day, at almost the exact same time, you both decide to withdraw $100 from different ATMs.

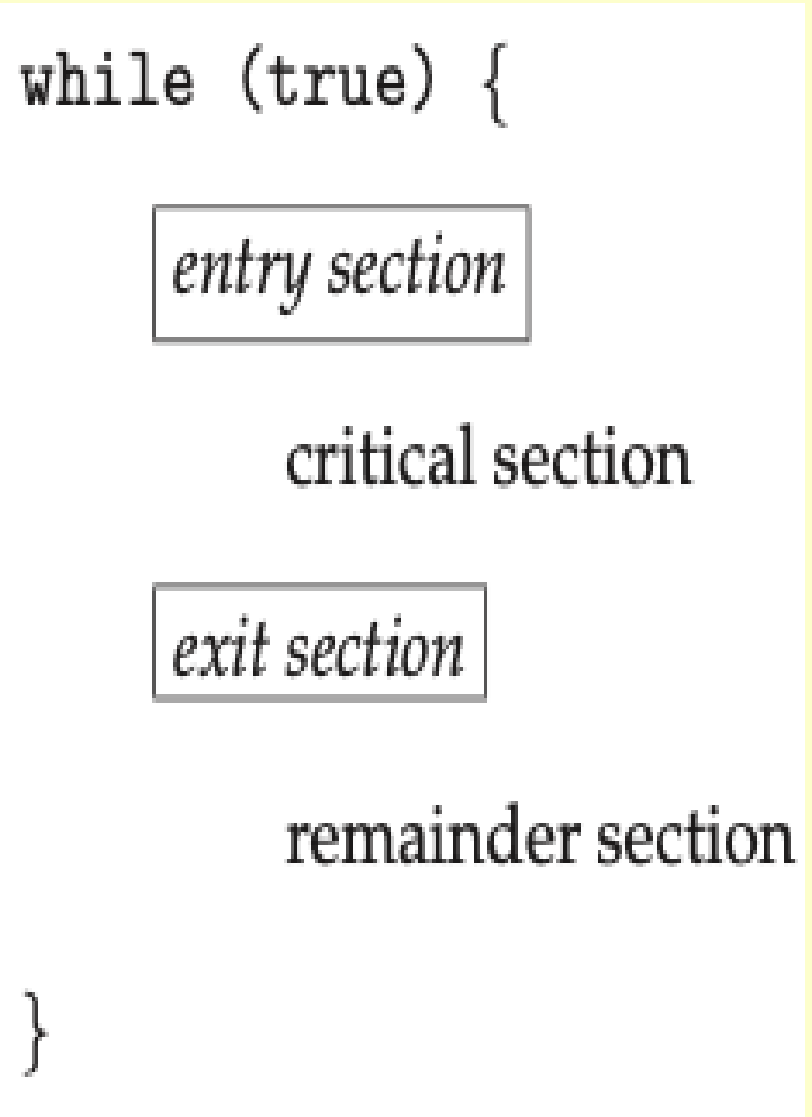
ATM A (CHENNAI)	ATM B (DELHI)
Reads the account balance. It sees $1,000.	At the exact same moment, before ATM A can update the balance, ATM B also reads the account balance. It also sees $1,000.
Calculates the new balance: $1,000 - $100 = $900. It updates the account balance to $900.	It doesn't know about ATM A's update. It performs its own calculation based on the balance it originally read: $1,000 - $100 = $900. It updates the account balance to $900.

- Even though a total of $200 was withdrawn, the final account balance is incorrectly recorded as $900 instead of the correct $800. The bank has lost $100 because the second transaction overwrote the result of the first one.

Critical Section Primitives

- Entry Section: The part of the program where a process requests permission to enter its critical section.
- Critical Section: The code section where the process accesses shared resources exclusively.
- Exit Section: The part where the process signals it has finished its critical section, allowing others to enter.
- Remainder Section: The rest of the code outside the critical section, where no shared resource access occurs.

Every process repeatedly executes these sections, and the challenge is to ensure only one process enters its critical section at a time, preventing race conditions.



Synchronization Working

- Synchronization works by using special tools or mechanisms like **locks, semaphores, or monitors** to **control access**:
  - When a process wants to enter its critical section, it requests permission from the synchronization tool.
    - If no other process is in the critical section, permission is granted.
  - If the critical section is occupied, other processes wait until the resource is free.
  - After finishing, the process releases the lock or signals the synchronization tool, allowing others to enter.
- These mechanisms ensure that processes take turns safely accessing shared resources without conflicts, preserving data accuracy and system stability.

Requirements of Synchronization

- To solve the critical section problem, synchronization mechanisms must satisfy these three main requirements:

  - **Mutual Exclusion**: Only one process can be inside the critical section at any time.
  - **Progress**: If no process is in the critical section, and some processes want to enter, only those not in their remainder section (non-critical part) can decide who enters next immediately, ensuring no indefinite waiting.
  - **Bounded Waiting**: A process that wants to enter its critical section must have a limit on how many times others can enter before it is allowed, preventing starvation (waiting forever).

Solution 1 – Peterson's Solution

- Peterson's solution is a **classic software-based method** to solve the critical section problem **for two processes**.
- It uses two **shared variables**:
  - An array flag to indicate if a process wants to enter the critical section.
  - A variable turn to indicate which process's turn it is to enter.
- **Concept**:
  - Each process announces its intention to enter by setting its flag to true.
  - Then it gives turn to the other process to give it a chance.
  - Next, each process waits if the other wants to enter and it's the other's turn.
  - When the other process leaves the critical section, the waiting process proceeds.

Solution 1 – Peterson's Solution

Process 0	Process 1	Action Explanation
flag[0] = true;	flag[1] = true;	Each process signals its intention to enter the critical section by setting its own flag to true.
turn = 1;	turn = 0;	Each process gives preference to the other by setting turn to the opponent's ID. This means it is willing to wait if the other process wants to enter.
while (flag[1] == true && turn == 1) Busy wait	while (flag[0] == true && turn == 0) Busy wait	▪ Each process checks if the other process wants to enter (flag[other] == true) and if it is the other's turn (turn == other). ▪ If both are true, the process waits (busy-waits) before entering the critical section.
// Critical Section starts // access shared resources here // Critical Section ends	*// Critical Section starts // access shared resources here // Critical Section ends*	▪ When the while condition is false, the process enters the critical section to access shared resources exclusively. ▪ No other process can enter because flag[other] or turn conditions block them.
flag[0] = false;	flag[1] = false;	▪ After exiting the critical section, the process resets its flag to false, signaling it no longer needs exclusive access. ▪ This allows the other process to enter if it is waiting.

Solution 2 – Bakery Algorithm

- A classic solution to the critical section problem for **N processes**
- It **ensures mutual exclusion**, meaning only one process can enter its critical section at a time; while also **ensuring fairness** by serving the processes in the order they request access.
- **Concept**:
  - **Each process** wanting to enter its critical section **picks a number**.
  - The **process with the smallest number gets to enter the critical section first**.
  - If **two processes have the same number**, the one with the **smaller process ID** is given priority.
  - Numbers are assigned by taking **one more than the maximum number currently held** by any process.
  - When a process exits the critical section, it **resets its number to zero** to indicate it no longer needs access.

Solution 2 – Bakery Algorithm

- **Data Structures Used:**
  - **choosing[i]**: Boolean array indicating whether process i is choosing a number.
  - **number[i]**: Integer array holding the ticket number for process i.

Solution 2 – Bakery Algorithm

Algorithm	Explanation

choosing[i] = true;
number[i] = 1 + max(number[0…N-1]);
choosing[i] = false;
	▪ A process sets choosing[i] to true to indicate it is picking a number.
	▪ It sets its number to one more than the maximum number taken by any process, ensuring unique increasing numbers.
	▪ Sets choosing[i] to false after picking the number.

for (j = 0; j < N; j++) { while (choosing[j]) { /* busy wait */ }	▪ Waits for other processes: ▪ It waits if another process is currently choosing a number.
while (number[j] != 0 && (number[j] < number[i] || (number[j] == number[i] && j < i))) { /* busy wait */ }	▪ It waits if another process has a smaller number, or the same number but a lower process ID (to break ties).
// Critical Section // (Access shared resource here)	▪ When no other process with higher priority is waiting, it enters the critical section.

// After critical section
number[i] = 0;
	▪ After execution, it resets its number to 0, signaling it has left the critical section.

Solution 2 – Bakery Algorithm

- The algorithm ensures mutual exclusion as no two processes can have the smallest number simultaneously.
- It provides progress and bounded waiting, so no process has to wait indefinitely.
- Suitable for any number of processes.
- It uses only shared memory and no special hardware instructions.
- However, it involves busy waiting (spinning) while waiting.

Solution 3 – H/w Based – Test & Set

- Test and Set is a hardware atomic instruction that reads a memory location and sets it to 1 simultaneously.
- It returns the original value before setting it.
- Used to implement spinlocks or simple mutexes.
- Working
  - A lock variable initialized to 0 means unlocked.
  - A process executes Test_and_Set(&lock):
    - If the returned value is 0, the process acquires the lock.
    - If it returns 1, the lock is already held, so the process keeps trying (busy waits).
  - This atomicity avoids race conditions on the lock variable.

Solution 3 – H/w Based – Test & Set

```
int TestAndSet(int *lock) {
    int old = *lock;    // Read old value
    *lock = 1;          // Set lock to 1
    return old;         // Return old value
}
```

- When a process calls acquire_lock(), it repeatedly calls TestAndSet(&lock).
- If another process holds the lock (lock == 1), it keeps spinning.
- When the lock becomes free (lock == 0), the process successfully sets it to 1 and enters the critical section.
- After finishing, it calls release_lock() to set lock back to 0.

```
int lock = 0;  // 0 means unlocked, 1 means locked

void acquire_lock() {
    while (TestAndSet(&lock) == 1) {
        // Busy wait (spin) until lock becomes available
    }
    // Lock acquired
}

void release_lock() {
    lock = 0;  // Release the lock
}
```

Solution 4 – H/w Based – Compare & Swap

- It Is an atomic hardware instruction widely used in multithreading and multiprocessing environments to perform synchronization without locks.
- CAS takes three parameters:
  - A memory location- p
  - An expected old value- old
  - A new value- new

Solution 4 – H/w Based – Compare & Swap

- It compares the current value at memory location p with old.
- If the current value is equal to old, it swaps (updates) the value at p with new.
- If the current value is not equal to old (meaning another thread/process changed it), it does nothing.
- The operation executes atomically, guaranteeing no other thread can interrupt it during execution.
- CAS returns a boolean or the original value indicating whether the swap was successful.

Solution 4 – H/w Based – Compare & Swap

- The CAS (Compare and Swap) hardware mechanism is an atomic instruction used to protect critical sections in concurrent programming.
- It works by comparing the contents of a memory location with an expected old value, and if they are the same, it atomically swaps the memory location with a new value.
- This operation is done as a single atomic step, ensuring no other thread can interfere during the comparison and update.

Solution 4 – H/w Based – Compare & Swap

- Suppose there is a shared memory location called lock which is initially 0 (meaning the critical section is free).
- A thread wants to enter the critical section, so it expects the value in lock to be 0.
- The thread executes CAS with these parameters:
  - expected old value = 0,
  - new value = 1 (meaning the thread wants to acquire the lock).
- CAS checks if the current value of lock is indeed 0.
  - If yes, CAS atomically sets lock to 1 and returns success.
  - If no (another thread already changed it), CAS returns failure, and the thread must try again.

Solution 4 – H/w Based – Compare & Swap

```
lock = 0  // initial state

function tryEnterCriticalSection() {
   expected = 0
   newValue = 1

   // Atomic CAS operation:
   // if (lock == expected) then lock = newValue else do nothing
   success = CAS(&lock, expected, newValue)

   if success {
       // Enter critical section
       …
       // Release lock after critical section
       lock = 0
   } else {
       // Failed to acquire lock, retry later
   }
}
```

Solution 4 – H/w Based – Compare & Swap

atomic_int lock = 0; // *shared lock variable*

Thread 1/ Process 1	Thread 1 / Process 1
Wants to enter critical section	Wants to enter critical section
Calls CAS: Compare if lock == 0	Calls CAS: Compare if lock == 0
Reads lock, which is currently 0	Reads lock, which is currently 0
Compares it to expected old value 0 (match)	Compares it to expected old value 0 (match)
Since match, atomically swaps lock to 1	CAS operation is queued by CPU but not yet executed because CPU handles one at a time
Returns success (true), Thread 1 acquired lock	Reads updated lock, now 1 from Thread 1's successful CAS
Enters critical section	CAS fails because lock ≠ 0 (it's 1)
Works inside critical section; other thread blocked	Thread 2 retries CAS in a loop (busy waiting)
Finishes critical section	Thread 2 continuously retries CAS
Sets lock back to 0 to release lock	Eventually reads lock as 0

Solution 6 : Monitors

- A monitor is a high-level synchronization tool designed to prevent conflicts when multiple processes or threads try to access a shared resource at the same time.
- Monitors simplify synchronization by bundling shared data and the procedures that operate on that data into a single unit, similar to a class in object-oriented programming.
- It data is not accessed directly from outside the monitor; it can only be manipulated through the monitor's own procedures

Solution 6 : Monitors

- The **core principle** of a monitor is **mutual exclusion**, which is enforced automatically.
- This means a monitor allows only one thread or process to be active within it at any point in time.
- If a second thread tries to enter the monitor while it's already occupied, it will be blocked and placed in an "entry queue" until the first thread exits.

Solution 6 : Monitors - Key Components

Shared Data	• These are the variables or resources that need to be protected from simultaneous access. This data is private to the monitor.
Procedures	• These are the functions that a process can call to interact with the shared data. A process outside the monitor can't access the data directly but can call these procedures.
Initialization Code	• This is a block of code that runs only once when the monitor is first created. It's used to set up the initial state of the shared data.
Condition Variables	• These are special variables used within the monitor to manage the synchronization of processes. They allow a process to wait for a specific condition to become true before proceeding

Solution 6 : Monitors - Key Components

- While mutual exclusion prevents multiple threads from executing in the monitor simultaneously, condition variables handle more complex synchronization scenarios. They support two main operations:

wait()

- When a process inside the monitor calls wait() on a condition variable, it is suspended and moved out of the monitor, allowing another process to enter. The suspended process waits until another process signals that the condition it was waiting for has been met.

signal()

- When a process calls signal() on a condition variable, it wakes up one of the processes that was suspended by a wait() call on the same condition variable. The awakened process can then re-enter the monitor to continue its execution when the monitor is free.

Solution 6: Monitors – Dining-Philosopher

```
// N is the number of philosophers
#define N 5
#define LEFT (i + N - 1) % N
#define RIGHT (i + 1) % N

// Define philosopher states
enum { THINKING, HUNGRY, EATING }
state[N];

// Monitor to manage the dining philosophers
monitor DiningSolution {
// Condition variable for each philosopher
    condition self[N];

// Initialization: all philosophers start by thinking
    initialization_code() {
        for (int i = 0; i < N; i++) {
            state[i] = THINKING;
        }
    }
```

```
// Private helper procedure to check if a philosopher can eat
    procedure test(int i) {
        if ((state[LEFT] != EATING) && (state[i]
== HUNGRY) && (state[RIGHT] != EATING)) {
            state[i] = EATING;

// Signal the philosopher that they can now proceed
            self[i].signal();
        }
    }
```

```
// Called by a philosopher to request forks
    procedure pickup(int i) {
        state[i] = HUNGRY;

// See if forks are available
        test(i);

// If not able to eat, wait
        if (state[i] != EATING) {
            self[i].wait();
        }
    }
```

```
// Called by a philosopher to release forks
    procedure putdown(int i) {
        state[i] = THINKING;
        // Check if neighbors can now
eat
        test(LEFT);
        test(RIGHT);
    }
}

// The code for each philosopher process
procedure philosopher(int i) {
    while (true) {
        think();
// Philosopher is thinking
        DiningSolution.pickup(i);
// Request forks
        eat();
// Philosopher is eating
        DiningSolution.putdown(i);
// Release forks
    }
}
```

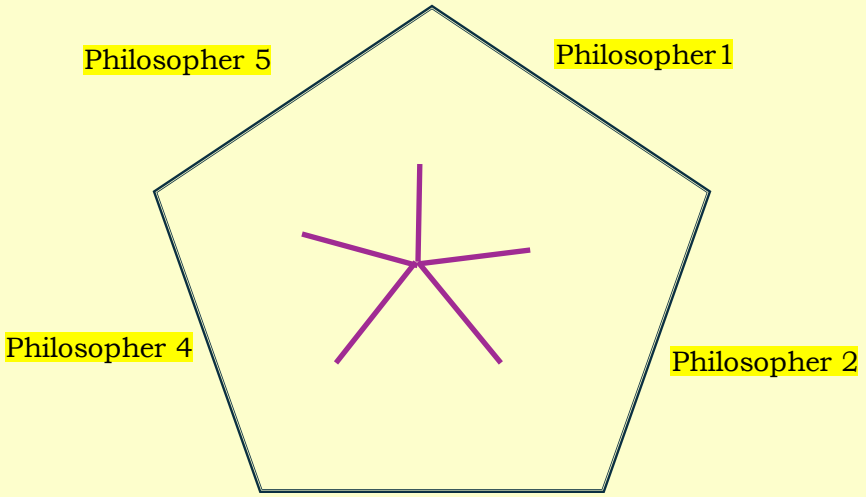
Solution 6: Monitors – Dining-Philosopher

```
// N is the number of philosophers
#define N 5
#define LEFT (i + N - 1) % N
#define RIGHT (i + 1) % N

// Define philosopher states
enum { THINKING, HUNGRY, EATING } state[N];

// Monitor to manage the dining philosophers
monitor DiningSolution {
// Condition variable for each philosopher
    condition self[N];

// Initialization: all philosophers start by thinking
    initialization_code() {
        for (int i = 0; i < N; i++) {
            state[i] = THINKING;
        }
    }
```



Thinking	Thinking	Thinking	Thinking	Thinking

An array tracks the state of each philosopher: THINKING, HUNGRY, or EATING. This shared data is protected by the monitor.

A high-level structure that encapsulates the shared state array and the procedures that modify it, ensuring mutual exclusion.

An array of condition variables, one for each philosopher. A philosopher who is hungry but cannot get forks will wait on their own condition variable.

Solution 6: Monitors – Dining-Philosopher

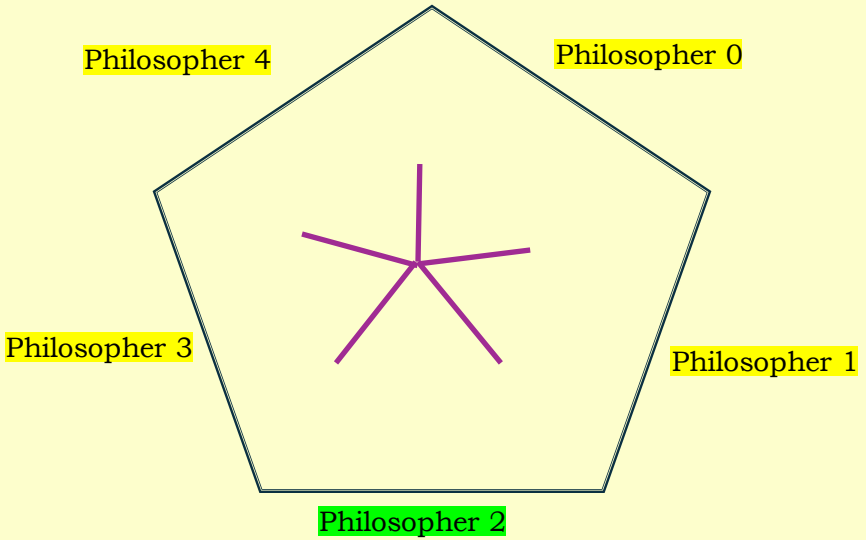
```
// Private helper procedure to check if a philosopher can eat
    procedure test(int i) {
        if ((state[LEFT] != EATING) && (state[i] == HUNGRY) && (state[RIGHT] != EATING)) {
            state[i] = EATING;

// Signal the philosopher that they can now proceed
            self[i].signal();
        }
    }

// Called by a philosopher to request forks
    procedure pickup(int i) {
        state[i] = HUNGRY;

// See if forks are available
        test(i);

// If not able to eat, wait
        if (state[i] != EATING) {
            self[i].wait();
        }
    }
```



Thinking	Thinking	Hungry	Thinking	Thinking

P2 becomes hungry:
P2 calls pickup(2).
Inside the monitor, state becomes HUNGRY.

Solution 6: Monitors – Dining-Philosopher

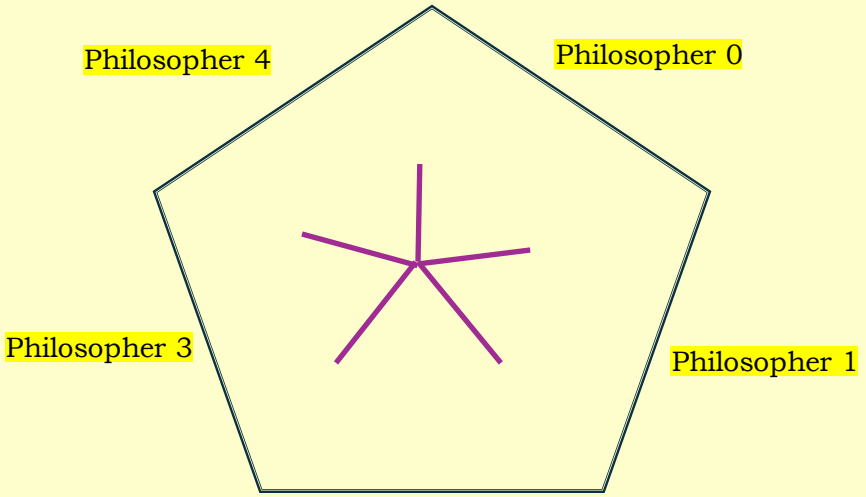
```
// Private helper procedure to check if a philosopher can eat
    procedure test(int i) {
        if ((state[LEFT] != EATING) && (state[i] == HUNGRY) && (state[RIGHT] != EATING)) {
            state[i] = EATING;

// Signal the philosopher that they can now proceed
            self[i].signal();
        }
    }

// Called by a philosopher to request forks
    procedure pickup(int i) {
        state[i] = HUNGRY;

// See if forks are available
        test(i);

// If not able to eat, wait
        if (state[i] != EATING) {
            self[i].wait();
        }
    }
```



Thinking	Thinking	Hungry	Thinking	Thinking

P2 becomes hungry:
P2 calls pickup(2).
Inside the monitor, state becomes HUNGRY.

test(2) is called.
It checks P1 and P3, who are THINKING

Solution 6: Monitors – Dining-Philosopher

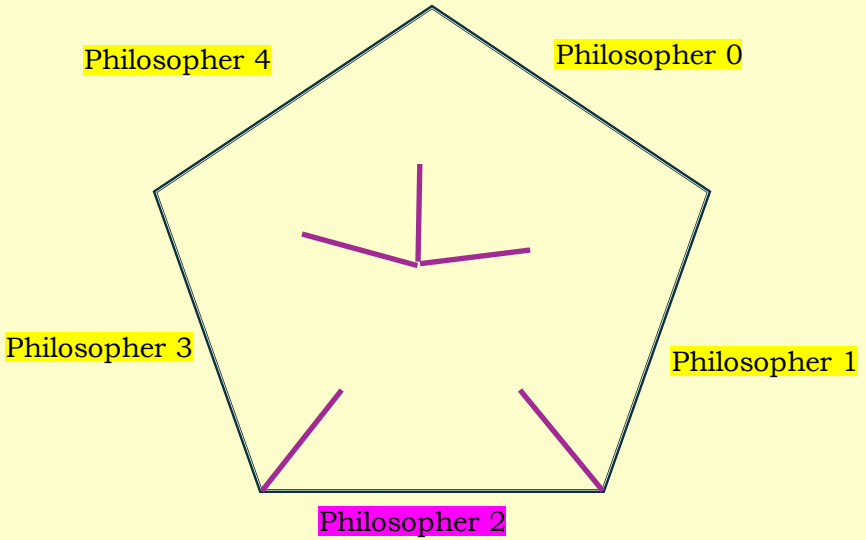
```
// Private helper procedure to check if a philosopher can eat
    procedure test(int i) {
        if ((state[LEFT] != EATING) && (state[i] == HUNGRY) && (state[RIGHT] != EATING)) {
            state[i] = EATING;

// Signal the philosopher that they can now proceed
            self[i].signal();
        }
    }

// Called by a philosopher to request forks
    procedure pickup(int i) {
        state[i] = HUNGRY;

// See if forks are available
        test(i);

// If not able to eat, wait
        if (state[i] != EATING) {
            self[i].wait();
        }
    }
```



Thinking	Thinking	Eating	Thinking	Thinking

P2 becomes hungry:
P2 calls pickup(2).
Inside the monitor, state becomes HUNGRY.

test(2) is called.
It checks P1 and P3, who are THINKING

The condition is true, so state is set to EATING, and
self.signal() is called (which has no effect as P2 isn't waiting).

P2 exits the monitor and starts EATING.

Solution 6: Monitors – Dining-Philosopher

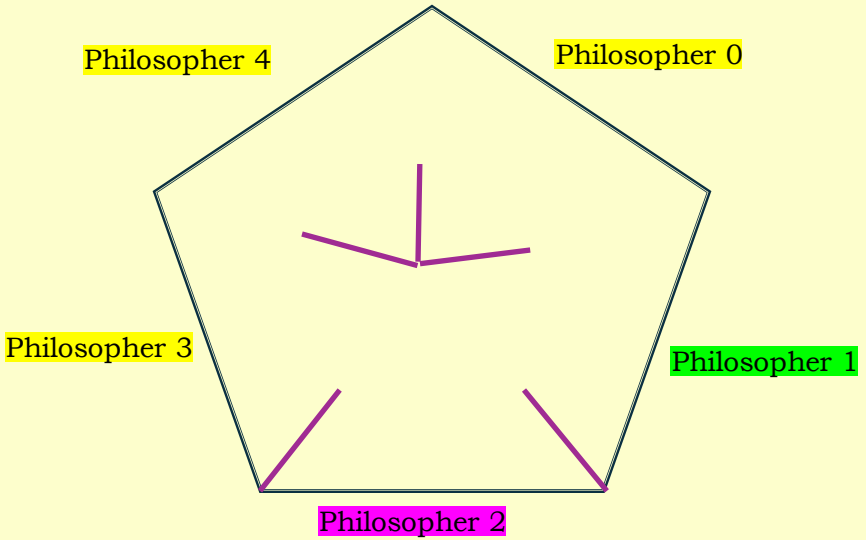
```
// Private helper procedure to check if a philosopher can eat
    procedure test(int i) {
        if ((state[LEFT] != EATING) && (state[i] == HUNGRY) && (state[RIGHT] != EATING)) {
            state[i] = EATING;

// Signal the philosopher that they can now proceed
            self[i].signal();
        }
    }

// Called by a philosopher to request forks
    procedure pickup(int i) {
        state[i] = HUNGRY;

// See if forks are available
        test(i);

// If not able to eat, wait
        if (state[i] != EATING) {
            self[i].wait();
        }
    }
```



Thinking	Hungry	Eating	Thinking	Thinking

P1 becomes hungry:
P1 calls pickup(1).
Inside the monitor, state becomes HUNGRY.

Solution 6: Monitors – Dining-Philosopher

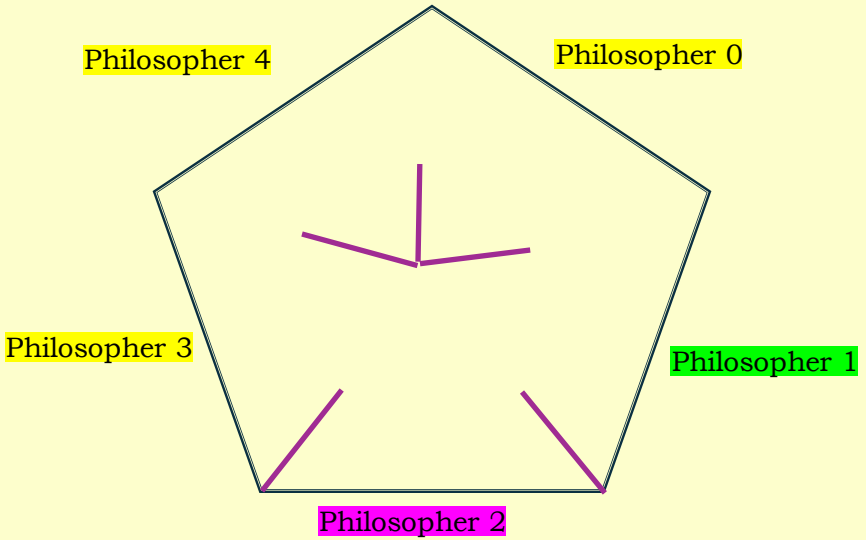
```
// Private helper procedure to check if a philosopher can eat
    procedure test(int i) {
        if ((state[LEFT] != EATING) && (state[i] == HUNGRY) && (state[RIGHT] != EATING)) {
            state[i] = EATING;

// Signal the philosopher that they can now proceed
            self[i].signal();
        }
    }

// Called by a philosopher to request forks
    procedure pickup(int i) {
        state[i] = HUNGRY;

// See if forks are available
        test(i);

// If not able to eat, wait
        if (state[i] != EATING) {
            self[i].wait();
        }
    }
```



Thinking	Hungry	Eating	Thinking	Thinking

P1 becomes hungry:
P1 calls pickup(1).
Inside the monitor, state becomes HUNGRY.

test(1) is called.
It checks P0 (THINKING) and P2 (EATING).
The condition state != EATING is false.
Nothing happens.

Solution 6: Monitors – Dining-Philosopher

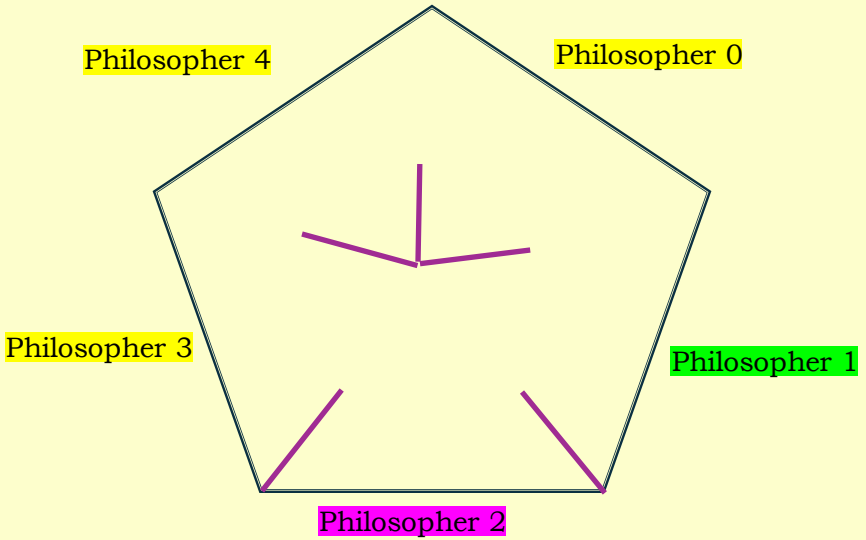
```
// Private helper procedure to check if a philosopher can eat
    procedure test(int i) {
        if ((state[LEFT] != EATING) && (state[i] == HUNGRY) && (state[RIGHT] != EATING)) {
            state[i] = EATING;

// Signal the philosopher that they can now proceed
            self[i].signal();
        }
    }

// Called by a philosopher to request forks
    procedure pickup(int i) {
        state[i] = HUNGRY;

// See if forks are available
        test(i);

// If not able to eat, wait
        if (state[i] != EATING) {
            self[i].wait();
        }
    }
```



Thinking	Hungry	Eating	Thinking	Thinking

P1 becomes hungry:
P1 calls pickup(1).
Inside the monitor, state becomes HUNGRY.

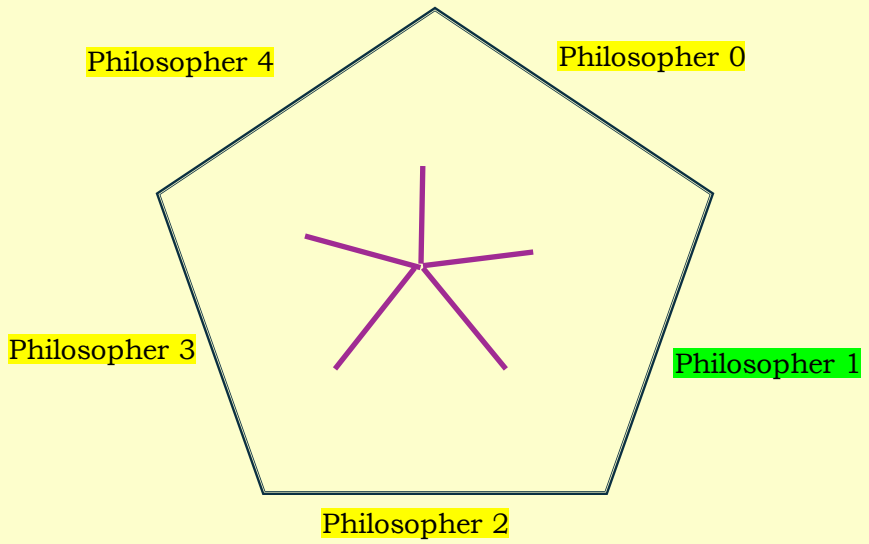
test(1) is called.
It checks P0 (THINKING) and P2 (EATING).
The condition state != EATING is false.
Nothing happens.

Back in pickup(1),
state is still HUNGRY,
so P1 calls self.wait().
P1 is now blocked and waiting.

Solution 6: Monitors – Dining-Philosopher

```
// Called by a philosopher to release forks
    procedure putdown(int i) {
        state[i] = THINKING;
        // Check if neighbors can now eat
        test(LEFT);
        test(RIGHT);
    }
}

// The code for each philosopher process
procedure philosopher(int i) {
    while (true) {
        think();
// Philosopher is thinking
        DiningSolution.pickup(i);
// Request forks
        eat();
// Philosopher is eating
        DiningSolution.putdown(i);
// Release forks
    }
}
```



Thinking	Hungry	Thinking	Thinking	Thinking

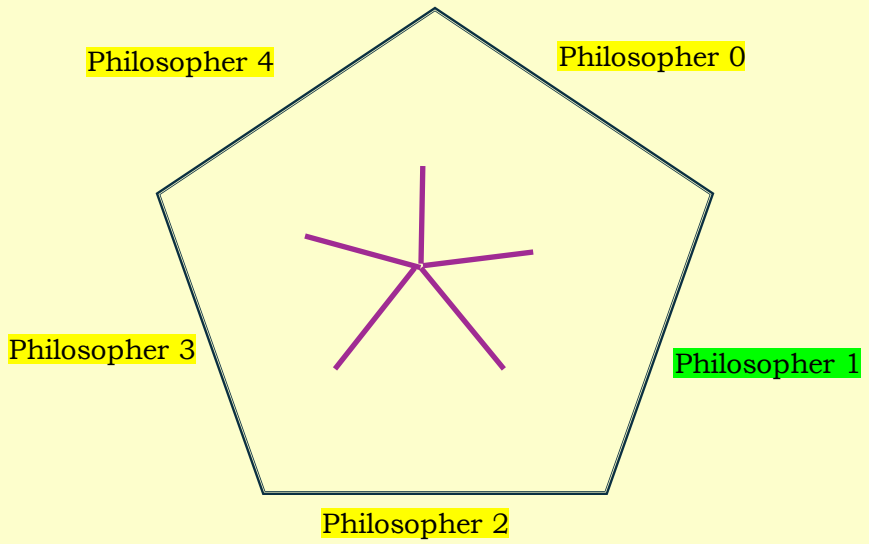
P2 finishes eating:
P2 calls putdown(2)
Inside the monitor, state becomes THINKING

putdown(2) calls test(LEFT),
which is test(1)
The condition is now true. state is set to EATING
self.signal() is called, which wakes up the waiting P1.

Solution 6: Monitors – Dining-Philosopher

```
// Called by a philosopher to release forks
    procedure putdown(int i) {
        state[i] = THINKING;
        // Check if neighbors can now eat
        test(LEFT);
        test(RIGHT);
    }
}

// The code for each philosopher process
procedure philosopher(int i) {
    while (true) {
        think();
// Philosopher is thinking
        DiningSolution.pickup(i);
// Request forks
        eat();
// Philosopher is eating
        DiningSolution.putdown(i);
// Release forks
    }
}
```



Thinking	Hungry	Thinking	Thinking	Thinking

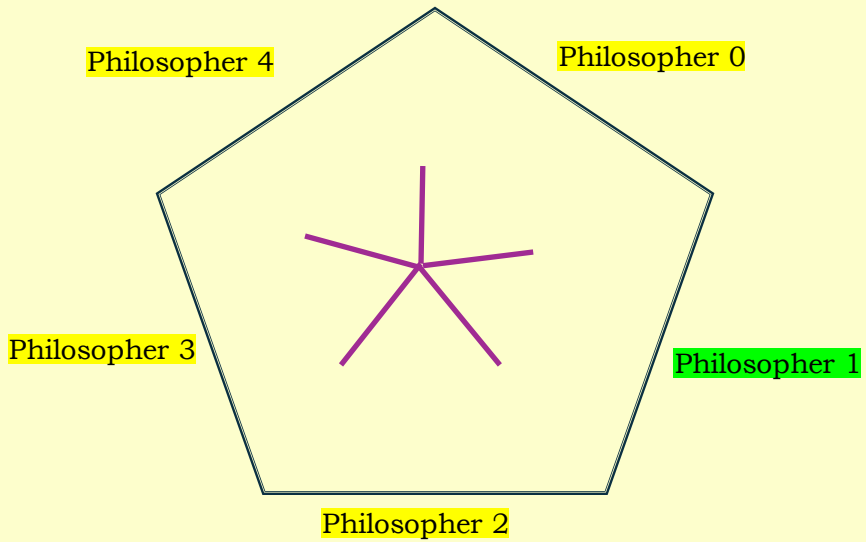
P2 finishes eating:
P2 calls putdown(2)
Inside the monitor, state becomes THINKING

putdown(2) calls test(LEFT),
which is test(1)
test(1) runs.
P1 is HUNGRY, and its neighbors P0 and P2 are THINKING.

Solution 6: Monitors – Dining-Philosopher

```
// Called by a philosopher to release forks
    procedure putdown(int i) {
        state[i] = THINKING;
        // Check if neighbors can now eat
        test(LEFT);
        test(RIGHT);
    }
}

// The code for each philosopher process
procedure philosopher(int i) {
    while (true) {
        think();
// Philosopher is thinking
        DiningSolution.pickup(i);
// Request forks
        eat();
// Philosopher is eating
        DiningSolution.putdown(i);
// Release forks
    }
}
```



Thinking	Hungry	Thinking	Thinking	Thinking

P2 finishes eating:
P2 calls putdown(2)
Inside the monitor, state becomes THINKING

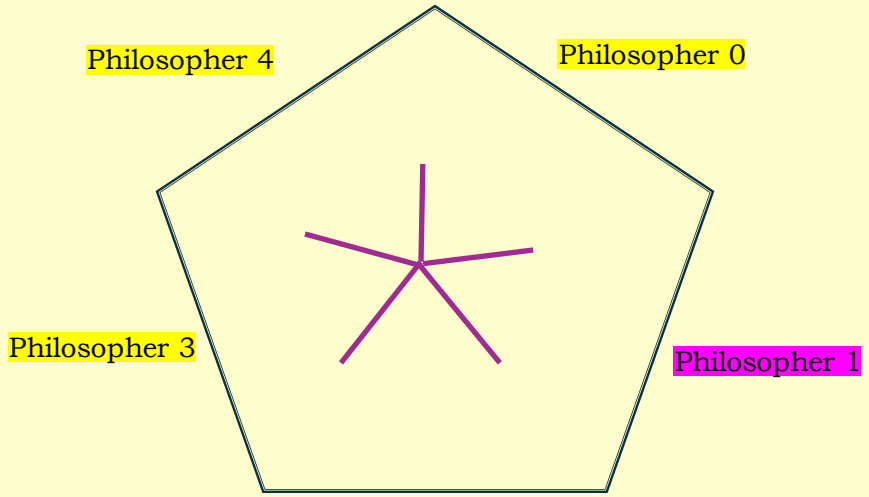
putdown(2) calls test(LEFT),
which is test(1)
test(1) runs.
P1 is HUNGRY, and its neighbors P0 and P2 are THINKING.

putdown(2) then calls test(RIGHT),
which is test(3).
Assuming P3 is not hungry, nothing happens.
P2 exits the monitor and returns to THINKING

Solution 6: Monitors – Dining-Philosopher

```
// Called by a philosopher to release forks
    procedure putdown(int i) {
        state[i] = THINKING;
        // Check if neighbors can now eat
        test(LEFT);
        test(RIGHT);
    }
}

// The code for each philosopher process
procedure philosopher(int i) {
    while (true) {
        think();
// Philosopher is thinking
        DiningSolution.pickup(i);
// Request forks
        eat();
// Philosopher is eating
        DiningSolution.putdown(i);
// Release forks
    }
}
```



Thinking	Eating	Thinking	Thinking	Thinking

P1 resumes:
P1 wakes up from its wait call,
re-acquires the monitor lock,
and its pickup(1) procedure completes.
P1 now starts EATING.

Solution 5 – Monitors – Dining Philosopher

- To use a monitor to solve the Dining Philosophers problem, you **encapsulate the shared resources (the forks) and the operations on them within a monitor**.
- This provides automatic mutual exclusion, ensuring only one philosopher can access the monitor's code at a time.
- The monitor also uses condition variables to **manage the state of each philosopher**, preventing deadlock by only allowing a philosopher to pick up both forks simultaneously when they become available.

Solution 5 – Monitors – Dining Philosopher

- **Monitor structure**
  - **States**: An array **state[5]** tracks each philosopher's state (0-4), which can be **THINKING**, **HUNGRY**, or **EATING**.
  - **Condition variables**: An array of condition variables **self[5]** blocks a philosopher if they are HUNGRY but cannot eat.
  - **Monitor functions**: The monitor includes three main functions:
    - **pickup(i)**: Called by philosopher i when they want to eat.
    - **putdown(i)**: Called by philosopher i when they finish eating.
    - **test(i)**: An internal helper function that checks if philosopher i can start eating.

Solution 5 – Monitors – Dining Philosopher

- **Initial State**
  - All five philosophers (P0-P4) are THINKING.
  - All five forks (F0-F4) are available.
  - The state array is all THINKING.
  - The self condition variables are all empty.

Solution 5 – Monitors – Dining Philosopher

```
monitor DiningPhilosophers
    enum state {THINKING, HUNGRY, EATING};
    state philosopher_state[5];
    condition self[5];
```

Solution 5 – Monitors – Dining Philosopher

```
// Helper function to check and allow eating
    private procedure test(i)
        // Check if philosopher `i` is hungry and both neighbors are not eating
        if (philosopher_state[i] == HUNGRY AND
            philosopher_state[(i + 4) % 5] != EATING AND
            philosopher_state[(i + 1) % 5] != EATING)
        then
            philosopher_state[i] = EATING;
            self[i].signal(); // Wake up philosopher `i` if they are waiting
        end if
    end procedure
```

Solution 5 – Monitors – Dining Philosopher

```
// Called by a philosopher `i` when they want to eat
    public procedure pickup(i)
        philosopher_state[i] = HUNGRY;
        test(i); // Attempt to start eating
        if (philosopher_state[i] != EATING)
        then
            self[i].wait(); // Wait if unable to eat
        end if
    end procedure
```

Solution 5 – Monitors – Dining Philosopher

```
// Called by a philosopher `i` when they finish eating
    public procedure putdown(i)
        philosopher_state[i] = THINKING;
        test((i + 4) % 5); // Check if the left neighbor can now eat
        test((i + 1) % 5); // Check if the right neighbor can now eat
    end procedure
```

Solution 5 – Monitors – Dining Philosopher

```
// Initialization block
    initialization_code()
        for i = 0 to 4
            philosopher_state[i] = THINKING;
        end for
    end initialization_code


end monitor
```

Solution 5 – Monitors – Dining Philosopher

Philosopher process

The pseudocode for each philosopher's individual process shows how it interacts with the monitor to coordinate its eating and thinking cycle.

```pseudocode
procedure philosopher(i)
    while (true)
        // The philosopher is thinking
        think();

        // The philosopher is hungry and wants to eat
        DiningPhilosophers.pickup(i);

        // The philosopher is eating
        eat();

        // The philosopher has finished eating
        DiningPhilosophers.putdown(i);
    end while
end procedure
```

Solution 5 – Monitors – Dining Philosopher

- **monitor DiningPhilosophers**: This declares the beginning of a monitor named DiningPhilosophers. The monitor is a high-level synchronization construct that provides automatic mutual exclusion, meaning only one process or thread can be active inside the monitor's code at any given time.

- **enum state {THINKING, HUNGRY, EATING};:** This line defines an enumeration to represent the three possible states of each philosopher.
  - THINKING: The philosopher is not trying to eat.
  - HUNGRY: The philosopher wants to eat and is trying to acquire the forks.
  - EATING: The philosopher has acquired both forks and is eating.

- **state philosopher_state[5];**: An array philosopher_state is declared to store the state of each of the five philosophers.

- **condition self[5];**: An array of condition variables self is declared, one for each philosopher. A condition variable is a queue for threads that are waiting for a specific condition to become true. In this case, a philosopher waits on self[i] when they are hungry but cannot eat

Solution 5 – Monitors – Dining Philosopher

- **private procedure test(i)**: This helper procedure is an internal monitor function. It is not called directly by the philosophers outside the monitor. The function checks if a philosopher i can transition from the HUNGRY to the EATING state.
- if (philosopher_state[i] == HUNGRY AND ...): This is the safety check that prevents deadlock. It tests three conditions:
  - philosopher_state[i] == HUNGRY: Checks if the philosopher i is trying to eat.
  - philosopher_state[(i + 4) % 5] != EATING: Checks if the philosopher's left neighbor is not eating. The modulo operator % 5 handles the circular table. For philosopher 0, the left neighbor is philosopher 4.
  - philosopher_state[(i + 1) % 5] != EATING: Checks if the philosopher's right neighbor is not eating. For philosopher 4, the right neighbor is philosopher 0.
    - then philosopher_state[i] = EATING;: If all conditions are met, the philosopher's state is updated to EATING. This signals that the philosopher can now acquire both forks.
  - self[i].signal();: This operation wakes up a single thread waiting on the self[i] condition variable. If the philosopher i was waiting to eat, this signal allows them to proceed. If they weren't waiting, the signal has no effect.

Solution 5 – Monitors – Dining Philosopher

- public procedure pickup(i): This procedure is called by philosopher i when they become hungry and wish to eat.
- philosopher_state[i] = HUNGRY;: The philosopher's state is immediately updated to HUNGRY.
- test(i);: The test procedure is called to see if the philosopher can begin eating.
- if (philosopher_state[i] != EATING) then: If the test(i) call failed (because a neighbor was eating), this condition will be true.
  - self[i].wait();: The philosopher is blocked and put in a waiting queue on the condition variable self[i]. This also releases the monitor's lock, allowing another philosopher to enter the monitor.

Solution 5 – Monitors – Dining Philosopher

- public procedure putdown(i): This procedure is called by philosopher i when they finish eating.
- philosopher_state[i] = THINKING;: The philosopher's state is updated to THINKING.
- test((i + 4) % 5);: The test procedure is called for the left neighbor. If that neighbor was HUNGRY and now can eat, they will be signaled.
- test((i + 1) % 5);: The test procedure is called for the right neighbor, potentially allowing them to eat.

Solution 5 – Monitors – Dining Philosopher

- procedure philosopher(i): This outlines the behavior of an individual philosopher.
- think();: Represents the time a philosopher spends thinking.
- DiningPhilosophers.pickup(i);: The philosopher calls the monitor procedure to acquire the forks. This call will block if the forks are not available.
- eat();: Once the pickup(i) call completes, the philosopher has both forks and can eat.
- DiningPhilosophers.putdown(i);: After eating, the philosopher releases the forks by calling the putdown procedure inside the monitor.

Solution 5 – Monitors – Dining Philosopher

Scenario: All philosophers become hungry

- P0 calls pickup(0):
  - P0 enters the monitor and acquires the lock.
  - state[0] is set to HUNGRY.
  - The test(0) function is called.
  - test(0) checks if P0's neighbors are eating.
    - Since P4 and P1 are THINKING, the condition is met.
  - state[0] is set to EATING, and the self[0].signal() operation is called. Since P0 is not waiting, the signal has no effect.
  - P0 now has both forks (F0 and F1) and exits the monitor, releasing the lock.

Solution 5 – Monitors – Dining Philosopher

Scenario: All philosophers become hungry

- P1 calls pickup(1):
  - P1 enters the monitor.
  - state[1] is set to HUNGRY.
  - test(1) is called.
    - test(1) checks if P0 and P2 are eating. Since P0 is EATING, the condition is not met.
  - P1 calls self[1].wait() and is blocked.
  - It releases the monitor lock and is added to the waiting queue for self[1].

Solution 5 – Monitors – Dining Philosopher

Scenario: All philosophers become hungry

- P2 calls pickup(2):
  - P2 enters the monitor.
  - state[2] is set to HUNGRY.
  - test(2) is called.
    - test(2) checks if P1 and P3 are eating. Since P1 is HUNGRY, the condition is not met.
  - P2 calls self[2].wait(), releases the lock, and is blocked on self[2].

Solution 5 – Monitors – Dining Philosopher

Scenario: All philosophers become hungry

- P3 calls pickup(3):
  - P3 enters the monitor.
  - state[3] is set to HUNGRY.
  - test(3) is called.
    - test(3) checks if P2 and P4 are eating. Since P2 is HUNGRY, the condition is not met.
  - P3 calls self[3].wait(), releases the lock, and is blocked on self[3].

Solution 5 – Monitors – Dining Philosopher

Scenario: All philosophers become hungry

- P4 calls pickup(4):
  - P4 enters the monitor.
  - state[4] is set to HUNGRY.
  - test(4) is called.
    - test(4) checks if P3 and P0 are eating. Since P0 is EATING, the condition is not met.
  - P4 calls self[4].wait(), releases the lock, and is blocked on self[4].

Solution 5 – Monitors – Dining Philosopher

Scenario: P0 finishes eating

P0 calls putdown(0):

P0 enters the monitor and acquires the lock.

state[0] is set to THINKING.

test() is called for P0's neighbors:

test(4): P4 is HUNGRY and its neighbors (P3 and P0) are now THINKING and THINKING, respectively. P4's condition is met.

state[4] is set to EATING.

self[4].signal() is called, which wakes up P4 from its waiting queue.

test(1): P1 is HUNGRY and its neighbors (P0 and P2) are THINKING and HUNGRY, respectively.

P1's condition is not met yet, so no signal is sent.

P0 exits the monitor, releasing the lock.

P4 now has both forks (F4 and F0) and begins eating.

Solution 5 – Monitors – Dining Philosopher

Scenario: P4 finishes eating

P4 calls putdown(4):

P4 enters the monitor.

state[4] is set to THINKING.

test() is called for P4's neighbors:

test(3): P3 is HUNGRY and its neighbors (P2 and P4) are HUNGRY and THINKING, respectively.

The condition is not yet met.

test(0): P0 is THINKING and is not hungry.

No action needed.

P4 exits the monitor.