

OPERATING SYSTEMS

UNIT 3 — Detailed Notes

Inter-Process Communication (IPC)

Topics covered: Critical Section • Race Conditions • Mutual Exclusion • Hardware Solution • Strict Alternation • Peterson's Solution • Producer-Consumer Problem • Semaphores • Event Counters • Monitors • Message Passing • Classical IPC Problems (Readers-Writers, Dining Philosophers)

 Green box = Sticky Note |  Red box = VVI Hotpoint |  Pink box = Numerical/Trace Practice Space |  Yellow box = Diagram Suggestion

1. Introduction to Inter-Process Communication (IPC)

1.1 What is Inter-Process Communication?

Inter-Process Communication (IPC) is the mechanism that allows multiple processes (or threads) to exchange data and coordinate their actions, especially when they are working on a shared task. When two or more processes work together to achieve a common goal, they are called Cooperating Processes, and they need a safe way to talk to each other and share data — this is exactly what IPC provides.

1.2 Independent vs Cooperating Processes

Aspect	Independent Process	Cooperating Process
Definition	Cannot affect or be affected by other processes; runs in complete isolation	Can affect or be affected by other processes; shares data with other processes
Data Sharing	No shared data with other processes	Shares data through shared memory or message passing
Effect of other processes	Execution result does not depend on order of execution of others	Execution result MAY depend on order of execution of others
Example	Two unrelated apps: a calculator and a music player	A producer process filling a buffer and a consumer process emptying it

1.3 Race Condition

A Race Condition occurs when two or more processes (or threads) access and manipulate the same shared data concurrently (at the same time), and the final result of that shared data depends on the exact order/timing in which the processes run. Since this timing can change every time the program runs, the output becomes unpredictable and incorrect.

Example of a Race Condition

Suppose two processes P1 and P2 both want to increase a shared variable 'count' by 1, and currently count = 5. The statement `count = count + 1` is actually executed in 3 low-level steps: (1) Read count into a register, (2) Add 1 to the register, (3) Write the register value back to count.

Shared variable: count = 5

Process P1	Process P2
-----	-----
R1 = count (R1 = 5)	
R1 = R1 + 1 (R1 = 6)	
back!	R2 = count (R2 = 5) <-- reads BEFORE P1 writes
	R2 = R2 + 1 (R2 = 6)
count = R1 (count = 6)	count = R2 (count = 6)
Expected final count = 7 (since both P1 and P2 increment it once)	
Actual final count = 6 (one increment got LOST!)	

★ **VVI Hotpoint (Exam Important):** This exact 'count = count + 1' example (with the 3 steps: Read, Add, Write) is the MOST commonly asked race condition example in exams. Always show that due to bad timing/interleaving, one of the updates gets lost.

● **Sticky Note:** Easy definition to remember: 'Race Condition = when the final result depends on WHO WINS THE RACE to access shared data first.' If the answer changes depending on timing, it's a race condition.

2. Critical Section Problem

2.1 What is a Critical Section?

A Critical Section is a part of a program (a piece of code) where a process accesses shared resources (like a shared variable, shared file, or shared data structure) that are also being used by other processes. To avoid race conditions, the OS must make sure that when one process is executing inside its critical section, NO other process is allowed to execute inside its own critical section for the same shared resource at the same time.

2.2 General Structure of a Process with a Critical Section

```
do {  
    // ----- Entry Section -----  
    // Request permission to enter the critical section  
  
    // ----- Critical Section -----  
    // The actual code that accesses shared data  
  
    // ----- Exit Section -----  
    // Announce that the process is leaving, allow others to enter  
  
    // ----- Remainder Section -----  
    // Rest of the code, does not touch shared data  
}  
while (true);
```

- **Entry Section:** Code that requests and checks whether it is safe to enter the critical section.
- **Critical Section:** The actual sensitive code that touches the shared resource.
- **Exit Section:** Code that signals that the process has finished using the shared resource.
- **Remainder Section:** All the other code of the process, which does not involve the shared resource.

2.3 Three Requirements of a Correct Solution (Critical Section Problem)

Requirement	Meaning
Mutual Exclusion	If a process is executing in its critical section, no other process can be allowed to execute in its own critical section at the same time.
Progress	If no process is currently in its critical section, and some processes want to enter, only those processes that are NOT in their remainder section can take part in deciding who enters next — and this decision cannot be postponed forever.
Bounded Waiting	There must be a limit on how many times other processes can enter their critical section after a process has requested entry and before that request is granted — so no process waits forever (no starvation).

★ **VVI Hotpoint (Exam Important):** These 3 requirements — Mutual Exclusion, Progress, Bounded Waiting — are asked almost EVERY exam, sometimes as a direct question, sometimes while checking if a given solution is 'correct'. Memorize all three with their exact meaning.

📌 **Diagram Suggestion (search on Google Images):** Draw a simple flow box diagram: a rectangle 'Entry Section' with an arrow going into a highlighted/shaded rectangle 'Critical Section', then an arrow to 'Exit Section', then an arrow to 'Remainder Section', and a final arrow looping back to 'Entry Section' (since it's inside a do-while loop). Search Google Images: "critical section entry exit remainder section diagram".

3. Mutual Exclusion & Solutions

3.1 What is Mutual Exclusion?

Mutual Exclusion is the requirement/property that ensures that if one process is executing in its critical section, then no other process is allowed to execute in its critical section at the same time — only ONE process can access the shared resource at any given moment. It is the most important of the three critical section requirements.

3.2 Strict Alternation (A Simple Software Solution)

Strict Alternation is one of the earliest and simplest attempts to achieve mutual exclusion between TWO processes using a shared variable called 'turn'. The value of 'turn' decides which process is allowed to enter the critical section next, and the processes must strictly take turns, one after another.

```
int turn = 0;    // shared variable

// Process P0
while (true) {
    while (turn != 0)
        ; // busy wait (do nothing)

    // critical section

    turn = 1;

    // remainder section
}

// Process P1
while (true) {
    while (turn != 1)
        ; // busy wait

    // critical section

    turn = 0;

    // remainder section
}
```

- **Problem with Strict Alternation:** It forces processes to enter the critical section in a strict turn-by-turn order, even if one process does not need to enter at all. If P0 finishes and sets turn=1, but P1 is busy doing something else in its remainder section (not interested in critical section), P0 CANNOT enter again until P1 takes its turn — this VIOLATES the Progress requirement.

 **Sticky Note:** Strict Alternation satisfies Mutual Exclusion, but FAILS Progress — this exact point ('violates progress condition') is the most common exam answer expected when asked 'what is wrong with strict alternation'.

3.3 Peterson's Solution (Software Solution for 2 Processes)

Peterson's Solution is a classic software-only algorithm that correctly solves the critical section problem for exactly TWO processes, satisfying all three requirements: Mutual Exclusion, Progress, and Bounded Waiting. It uses two shared variables: an array flag[2] (to show interest in entering critical section) and a variable turn (to break ties when both want to enter).

```
boolean flag[2] = {false, false}; // shared: does process i want to enter?
int turn;                          // shared: whose turn is it if both want in

// Process Pi
flag[i] = true; // I am interested
turn = j;       // but let the other go first

while (flag[j] && turn == j)
    ; // busy wait

// Process Pj (the other process)
flag[j] = true;
turn = i;

while (flag[i] && turn == i)
    ; // busy wait
```

```
// ----- CRITICAL SECTION ----- // ----- CRITICAL SECTION -----
flag[i] = false; // done, I am leaving flag[j] = false;
// ----- remainder section ----- // ----- remainder section -----
```

- **How it guarantees Mutual Exclusion:** Both processes cannot be in the critical section together, because if both set `flag[i]=true` and `flag[j]=true`, only ONE value of 'turn' can be true at a time (turn is either i or j, not both) — so one process's while-loop condition stays true and it keeps waiting.
- **How it guarantees Progress:** If a process is not interested (`flag=false`), the other can enter freely without waiting for a 'turn'.
- **How it guarantees Bounded Waiting:** After a process exits the critical section (`flag[i]=false`), the waiting process immediately gets permission, so waiting is bounded to just one entry by the other process.

★ **VVI Hotpoint (Exam Important):** Peterson's Solution code (with `flag[]` array and `turn` variable) is a VERY frequently asked 'write and explain' question. Practice writing this code from memory, and be ready to explain how each of the 3 requirements is satisfied.

3.4 Hardware Solutions for Mutual Exclusion

Software solutions like Peterson's work only for 2 processes and can be complex for more processes. Hardware solutions use special CPU instructions that complete as a single, indivisible (atomic) step — meaning no other process can interrupt them midway.

3.4.1 Test-and-Set Instruction

`TestAndSet(lock)` is a hardware instruction that reads the old value of 'lock', sets 'lock' to true, and returns the old value — and crucially, all of this happens as ONE atomic (uninterruptible) operation.

```
// Hardware guarantees this whole function runs atomically (as one step):
boolean TestAndSet(boolean *lock) {
    boolean old = *lock;
    *lock = true;
    return old;
}

// Using it for mutual exclusion (shared variable: lock = false initially):
while (TestAndSet(&lock))
    ; // busy wait (keep spinning) while lock was already true

// ----- CRITICAL SECTION -----

lock = false; // exit section: release the lock
```

- **Why it works:** Only ONE process can find `lock=false` and flip it to true at the exact same atomic instant; all others will keep finding `lock=true` and keep spinning (this is called Busy Waiting or a Spinlock).

3.4.2 Swap (Exchange) Instruction

The Swap instruction atomically exchanges the values of two variables. It can also be used for mutual exclusion in a very similar pattern:

```
// Hardware guarantees this swap happens atomically:
void Swap(boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
```

```
    *b = temp;
}

// Shared: lock = false initially. Each process has a local variable 'key'.
key = true;
while (key == true)
    Swap(&lock, &key);    // keep swapping until key becomes false

// ----- CRITICAL SECTION -----

lock = false;    // exit section
```

Both hardware solutions share the same key drawback:

- **Busy Waiting:** A waiting process keeps the CPU busy running an empty loop just to check a condition repeatedly, instead of being put to sleep — this wastes CPU time that could be given to another useful process. This is exactly the problem that Semaphores (next topic) solve more efficiently.

4. Semaphores

4.1 What is a Semaphore?

A Semaphore is a special integer variable, invented by Edsger Dijkstra, that is used to solve the critical section problem and synchronize processes, WITHOUT the busy-waiting problem of hardware solutions. A semaphore can only be accessed through two special, atomic operations: wait() (also called P, from Dutch 'proberen' = to test) and signal() (also called V, from Dutch 'verhogen' = to increment).

4.2 wait() and signal() Operations

```
wait(S) {                               // also written as P(S) or down(S)
    while (S <= 0)                       // busy wait (in the simplest definition)
        ;
    S = S - 1;
}

signal(S) {                             // also written as V(S) or up(S)
    S = S + 1;
}

// Both operations are performed ATOMICALLY (as one indivisible step)
```

 **Sticky Note:** Easy way to remember: wait() DECREASES the semaphore (a process is 'asking/taking permission'), signal() INCREASES the semaphore (a process is 'giving back/releasing permission').

4.3 Types of Semaphores

Type	Description	Used for
Binary Semaphore	Can only hold the value 0 or 1. Works exactly like a lock/mutex — 1 means resource is free, 0 means resource is busy.	Mutual exclusion for a single shared resource (like a lock on one shared variable)
Counting Semaphore	Can hold any non-negative integer value. The value represents the number of units of a resource currently available.	Managing a resource that has multiple identical instances (like a pool of 5 printers)

4.4 Using Semaphore for Mutual Exclusion

```
semaphore mutex = 1;    // binary semaphore, shared, initialized to 1

do {
    wait(mutex);         // Entry Section — decreases mutex to 0, enters

    // ----- CRITICAL SECTION -----

    signal(mutex);       // Exit Section — increases mutex back to 1

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

How it works: Initially mutex=1. When a process calls wait(mutex), it becomes 0, and the process enters the critical section. If another process now calls wait(mutex), it sees mutex=0 (<=0) and must wait (sleep), since only one process can hold the resource. When the first process finishes and calls signal(mutex), mutex becomes 1 again, and the waiting process is woken up and allowed to enter.

4.5 Key Advantage over Hardware Solutions

In a proper OS implementation, when a process calls `wait()` and finds $S \leq 0$, instead of busy-waiting in a loop (wasting CPU), the process is moved to a Waiting State and added to a queue associated with that semaphore — the CPU is immediately given to some other useful process. When `signal()` is later called, one waiting process is woken up and moved back to the Ready queue. This avoids wasting CPU cycles, unlike `TestAndSet/Swap`.

★ **VVI Hotpoint (Exam Important):** Classic exam question: 'How does a semaphore avoid busy waiting?'
Answer: By blocking the process (moving it to a waiting queue tied to the semaphore) instead of looping, so CPU can be used by other processes meanwhile.

4.6 Event Counters

An Event Counter is another synchronization tool, simpler than a semaphore in concept — it is a monotonically increasing integer variable (it only ever goes up, never down) associated with an event. It counts how many times a certain event has occurred.

- **Key operations:** `Advance(E)` increases the event counter `E` by 1 whenever the event occurs. `Read(E)` returns the current value of the counter. `Await(E, v)` causes a process to wait (sleep) until the event counter `E` reaches at least the value `v`.
- **Difference from Semaphore:** A semaphore's value can both increase (signal) and decrease (wait), but an event counter's value only ever increases — it simply counts occurrences and lets processes wait for a specific count to be reached, which makes it useful for producer-consumer style synchronization without the risk of a semaphore going negative or being misused.

- **Why wait(empty) before wait(mutex)?** The ORDER of wait() calls is very important. If mutex is locked first and then the process waits for empty/full, it could lead to a situation called Deadlock, where the Producer holds mutex while waiting for empty space, and the Consumer can never get mutex to free up a slot. So resource-counting semaphores (empty/full) must always be waited on BEFORE the mutex.

★ **VVI Hotpoint (Exam Important):** A VERY common trick question: 'What happens if we swap the order of wait(mutex) and wait(empty) in the Producer?' Answer: It can cause a DEADLOCK — always explain this using the mutex-held-while-waiting logic above.

✎ **Numerical / Trace Practice Space:** Practice Trace: Take buffer size $N = 3$ (so empty starts at 3, full starts at 0, mutex = 1). Manually trace through this sequence of events step-by-step, writing the value of mutex/empty/full after each step: Producer adds item, Producer adds item, Consumer removes item, Producer adds item (buffer now full), Consumer removes item, Consumer removes item. This kind of semaphore-value trace table is a common numerical-style question.

6. Monitors

6.1 What is a Monitor?

A Monitor is a high-level synchronization construct (provided directly by the programming language, not just the OS) that bundles together shared data AND the procedures that operate on that shared data, into a single unit. The key guarantee of a monitor is that only ONE process can be active (executing any procedure) inside the monitor at any given time — this automatic mutual exclusion is built-in, so the programmer does not need to manually place wait()/signal() calls everywhere like with semaphores.

● **Sticky Note:** Think of a Monitor as a 'smart room' — shared data and the only functions allowed to touch that data are locked together inside one room, and the room's door only lets ONE person in at a time automatically. You don't need to manually put a lock on the door every time — it's built into the room itself.

6.2 Structure of a Monitor

```
monitor MonitorName {  
    // shared variables (data) – only accessible from inside this monitor  
    data variables;  
  
    // condition variables, used for waiting inside the monitor  
    condition x, y;  
  
    procedure P1(...) {  
        ...  
    }  
  
    procedure P2(...) {  
        ...  
    }  
  
    initialization code {  
        ...  
    }  
}
```

6.3 Condition Variables

Even though a Monitor gives automatic mutual exclusion, sometimes a process inside the monitor needs to wait for some specific condition to become true (e.g., 'wait until the buffer is not empty'). For this, Monitors provide Condition Variables, which support exactly two operations:

- **x.wait():** The process calling this is suspended (paused) and placed in a waiting queue for condition x, and importantly, it temporarily gives up the monitor so other processes can enter.
- **x.signal():** Wakes up (resumes) exactly one process that is waiting on condition x. If no process is waiting, signal() has no effect (unlike a semaphore's signal, which always increments the value even if nobody is waiting).

★ **VVI Hotpoint (Exam Important):** Key difference to remember for exams: Semaphore's signal() ALWAYS changes the semaphore value (even if no process is waiting, the count still increases, which can be 'remembered' for a future wait()). But Condition Variable's signal() has NO EFFECT and is simply LOST if no process is currently waiting on that condition.

6.4 Monitor vs Semaphore

Aspect	Monitor	Semaphore
Level	High-level, language-supported construct	Low-level, OS/hardware-supported construct
Mutual Exclusion	Automatic/built-in (enforced by the compiler/language)	Must be manually coded by the programmer using wait()/signal()
Ease of use	Easier, less error-prone (fewer chances of bugs)	More error-prone — forgetting a wait() or signal() can break everything
Waiting mechanism	Condition variables (wait/signal on specific conditions)	The semaphore integer value itself controls waiting

7. Message Passing

7.1 What is Message Passing?

Message Passing is a method of Inter-Process Communication where processes communicate and synchronize their actions WITHOUT sharing any common memory/address space — they simply send and receive messages to/from each other. This is especially useful in Distributed Systems, where processes may be running on completely different physical machines that have no shared memory at all.

Two basic operations are used:

- **send(message):** Used by a process to send a message (to another process or to a mailbox/port).
- **receive(message):** Used by a process to receive a message (from another process or from a mailbox/port).

7.2 Direct vs Indirect Communication

Aspect	Direct Communication	Indirect Communication
How processes are addressed	Each process must explicitly name the other process it wants to communicate with: send(P, message) and receive(Q, message)	Processes send/receive through a shared mailbox or port, not directly to each other: send(mailbox, message) and receive(mailbox, message)
Link existence	A link exists between exactly TWO specific processes only	A link can be shared/associated with many processes (many can send/receive via the same mailbox)
Flexibility	Less flexible — processes must know each other's identity in advance	More flexible — a mailbox can be shared, and senders/receivers can change over time

7.3 Synchronous vs Asynchronous Message Passing

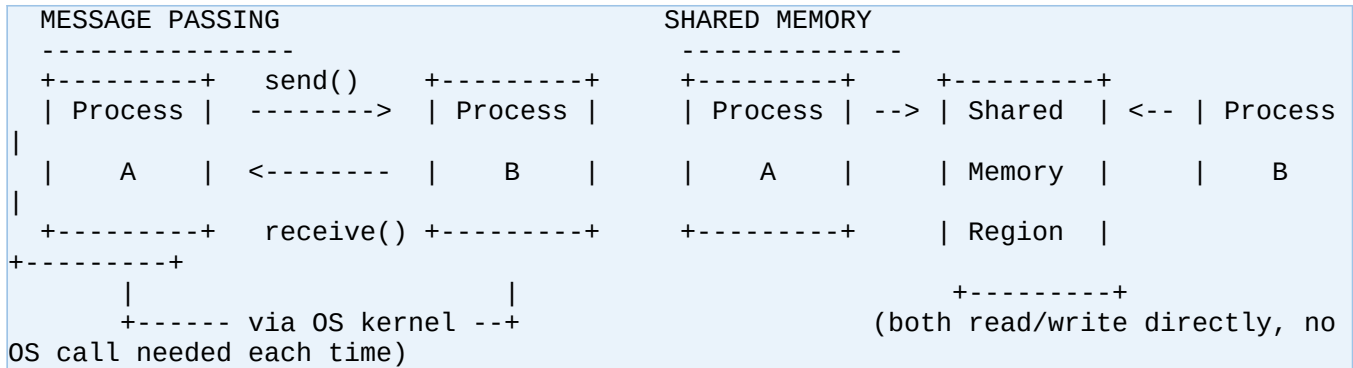
Type	Behavior
Blocking send	The sending process is BLOCKED (paused) until the message is actually received by the receiver.
Blocking receive	The receiving process is BLOCKED (paused) until a message becomes actually available to receive.
Non-blocking send	The sending process sends the message and CONTINUES its execution immediately, without waiting.
Non-blocking receive	The receiving process either receives a valid message immediately or receives a null/empty result right away, without waiting.

- **Rendezvous:** When BOTH send and receive are blocking, it is called a Rendezvous — both the sender and receiver pause and 'meet' at the exact moment the message is exchanged.

7.4 Message Passing vs Shared Memory

Aspect	Message Passing	Shared Memory
Communication method	Processes exchange data directly via send()/receive() system calls	Processes read/write to a common region of memory set up for sharing
Speed	Slower — every message passes through the kernel/OS (system call overhead)	Faster — once set up, no OS involvement needed for each

Aspect	Message Passing	Shared Memory
		read/write, direct memory access
Best suited for	Distributed systems, or when processes need to exchange small amounts of data occasionally	Systems on the same machine needing to exchange large amounts of data frequently
Synchronization needed?	Built into the send/receive mechanism itself	Must be handled separately by the programmer (e.g., using semaphores)



8. Classical IPC Problems

8.1 Readers-Writers Problem

Problem Statement: A shared data item (like a database record or file) is accessed by two types of processes — Readers, who only read the data (never modify it), and Writers, who modify (write to) the data. The rules required are:

- Multiple Readers can read the data AT THE SAME TIME — this is safe since nobody is changing the data.
- Only ONE Writer can write at a time, and while a Writer is writing, NO other process (reader or writer) can access the data at all.

Solution (First Readers-Writers Problem — Readers given priority)

```
semaphore mutex = 1;    // protects the 'read_count' variable
semaphore wrt = 1;      // controls access to the shared data (used by writers,
                        // and by first/last reader)
int read_count = 0;     // number of readers currently reading

Writer process:
do {
    wait(wrt);          // request exclusive
                        // access to the data

    // ----- WRITING ----- //
    locks out writers

    signal(wrt);        // release access
} while (true);

lets writers in

Reader process:
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(wrt);    // first reader

    signal(mutex);

    // ----- READING ----- //

    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(wrt);  // last reader

    signal(mutex);
} while (true);
```

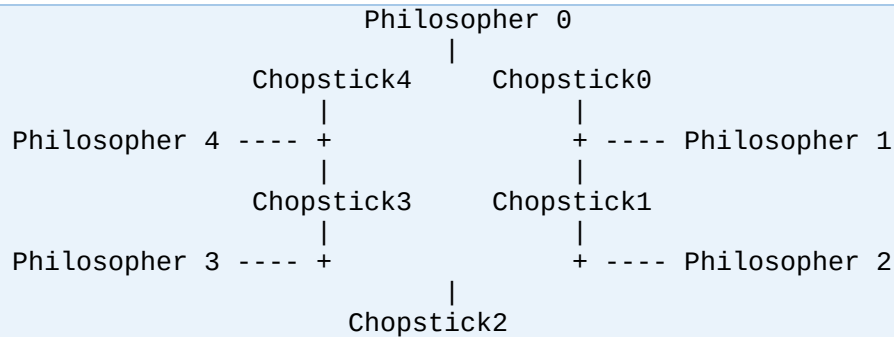
- **Why read_count matters:** wrt (write semaphore) is locked by only the FIRST reader that starts reading (blocking writers), and unlocked only by the LAST reader that finishes (letting a writer in). In between, many readers can read freely without touching wrt at all, since read_count safely tracks how many readers are active.

★ **VVI Hotpoint (Exam Important):** Classic exam question: 'In the Readers-Writers problem, why do we need the read_count variable along with a separate mutex for it?' Answer: Because read_count itself is shared data being modified by multiple readers, so it needs its own protection (mutex) to avoid a race condition on read_count.

● **Sticky Note:** A possible drawback of this 'readers-priority' solution: if readers keep arriving continuously, a writer may wait forever — this is called Writer Starvation. Variations of this problem (Second/Third Readers-Writers problem) try to fix this fairness issue.

8.2 Dining Philosophers Problem

Problem Statement: Five philosophers sit around a circular table. Between each pair of adjacent philosophers, there is exactly ONE chopstick (fork) — so there are 5 chopsticks total for 5 philosophers. Each philosopher alternates between THINKING (not using any chopstick) and EATING (needs BOTH the chopstick on their left AND the chopstick on their right to eat).



(5 philosophers sit in a circle, 5 chopsticks between them — each philosopher needs BOTH neighbouring chopsticks to eat)

The Core Problem: Deadlock

If EVERY philosopher picks up their LEFT chopstick at exactly the same time, then every single chopstick gets picked up, and every philosopher is now stuck waiting forever for their RIGHT chopstick (which their neighbour is already holding) — nobody can proceed. This is a classic example of Deadlock.

```
// NAIVE (BUGGY) solution — causes deadlock if all start together:
do {
    wait(chopstick[i]);           // pick up left chopstick
    wait(chopstick[(i+1) % 5]);   // pick up right chopstick

    // ----- EAT ----- //

    signal(chopstick[i]);         // put down left chopstick
    signal(chopstick[(i+1) % 5]); // put down right chopstick

    // ----- THINK ----- //
} while (true);
```

Solutions to Avoid Deadlock

- **Limit the number of philosophers at the table:** Allow at most 4 philosophers (one less than the number of chopsticks) to sit at the table at the same time — this guarantees at least one philosopher can always get both chopsticks.
- **Pick up chopsticks only if both are available:** A philosopher only picks up a chopstick if BOTH the left and right chopsticks are free at that moment (pick up both together, atomically, usually done by protecting the pickup check with a mutex).
- **Asymmetric solution (break the symmetry):** Make odd-numbered philosophers pick up their LEFT chopstick first then RIGHT, but even-numbered philosophers pick up their RIGHT chopstick first then LEFT. This breaks the 'everyone grabs left at once' circular pattern that caused the deadlock.

★ **VVI Hotpoint (Exam Important):** Dining Philosophers is mainly used to demonstrate DEADLOCK and how to AVOID it — exams commonly ask 'how does deadlock occur in this problem' and 'suggest any one solution to prevent it'. The 3 solutions above are the standard expected answers.

 **Numerical / Trace Practice Space:** Practice Trace: With 5 philosophers (P0-P4) and the 'limit to 4 philosophers at the table' solution, manually trace a scenario where P0, P1, P2, P3 are seated (P4 waits outside) and show step-by-step how at least one philosopher is always able to pick up both chopsticks and eat, proving no deadlock occurs.

Quick Revision Summary

Topic	One-Line Recap
Race Condition	Occurs when multiple processes access shared data concurrently and the final result depends on the timing/order of execution.
Critical Section	The part of a process's code that accesses shared resources; must be protected so only one process executes it at a time.
3 CS Requirements	Mutual Exclusion, Progress, and Bounded Waiting — every correct solution must satisfy all three.
Strict Alternation	Simple turn-based software solution; satisfies Mutual Exclusion but violates Progress.
Peterson's Solution	Software solution for 2 processes using flag[] and turn; satisfies all 3 requirements correctly.
Hardware Solutions	TestAndSet and Swap instructions give atomic operations for mutual exclusion but rely on Busy Waiting (CPU wastage).
Semaphore	An integer variable with atomic wait()/signal() operations; binary (0/1) or counting; avoids busy-waiting by blocking processes.
Event Counter	A monotonically increasing counter used to track how many times an event has occurred; processes can await a target count.
Producer-Consumer Problem	Solved using 3 semaphores: mutex (mutual exclusion), empty (free slots), full (filled slots); wrong order of wait() can cause deadlock.
Monitor	High-level language construct bundling shared data + procedures with automatic mutual exclusion; uses condition variables (wait/signal) for waiting.
Message Passing	IPC without shared memory, using send()/receive(); can be direct/indirect and synchronous(blocking)/asynchronous(non-blocking).
Readers-Writers Problem	Multiple readers can read together, but only one writer can write exclusively; uses read_count + mutex + wrt semaphores.
Dining Philosophers Problem	Classic deadlock-demonstration problem; solved by limiting diners, atomic pickup of both chopsticks, or asymmetric pickup order.

15 Exam-Style Questions — Unit 3

- Q1.** What is a Race Condition? Explain with a suitable example (e.g., $\text{count} = \text{count} + 1$).
- Q2.** What is a Critical Section? Explain the structure of a process containing Entry, Critical, Exit, and Remainder sections.
- Q3.** State and explain the three requirements that any correct solution to the Critical Section Problem must satisfy.
- Q4.** Explain Strict Alternation as a solution for mutual exclusion. Why does it fail to satisfy the Progress requirement?
- Q5.** Explain Peterson's Solution for two processes with code, and show how it satisfies Mutual Exclusion, Progress, and Bounded Waiting.
- Q6.** Explain the Test-and-Set hardware instruction and how it is used to achieve mutual exclusion. What is its main drawback?
- Q7.** What is a Semaphore? Explain `wait()` and `signal()` operations with their implementation.
- Q8.** Differentiate between Binary Semaphore and Counting Semaphore.
- Q9.** Explain how semaphores avoid the problem of busy waiting, unlike hardware solutions.
- Q10.** Explain the Producer-Consumer (Bounded Buffer) Problem and its solution using semaphores (`mutex`, `empty`, `full`).
- Q11.** What is a Monitor? Explain its structure and the role of condition variables (`wait` and `signal`).
- Q12.** Differentiate between a Monitor and a Semaphore.
- Q13.** What is Message Passing? Differentiate between Direct and Indirect Communication, and between Synchronous and Asynchronous communication.
- Q14.** Explain the Readers-Writers Problem and its semaphore-based solution. Why is a separate `mutex` needed for `read_count`?
- Q15.** Explain the Dining Philosophers Problem. How does deadlock occur, and what are the possible solutions to prevent it?