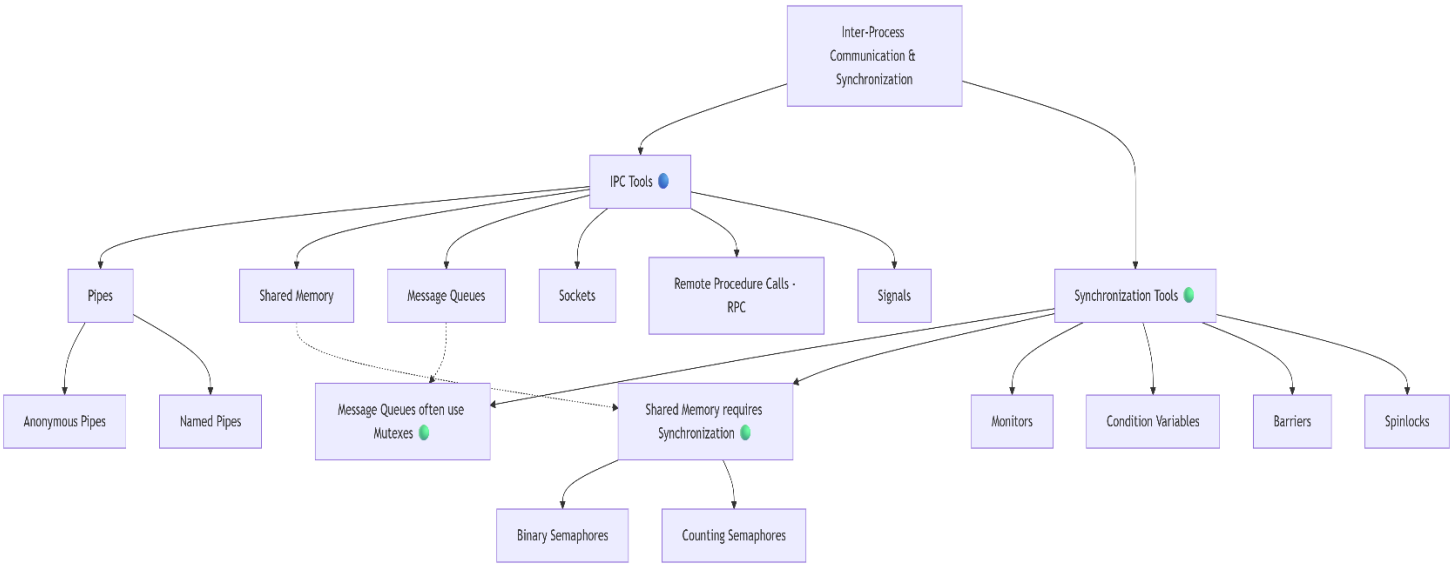
1.Slow Learner Activity:--



Question 1: IPC Focus

Q: ***Two processes need to exchange data on the same machine. One is a parent process, the other is its child. Which IPC tool should they use? Explain why.***

Solution:

- **Tool: Anonymous Pipes (🔵).**
- **Why:**
  - **Anonymous pipes are designed for one-way communication between parent-child processes.**
  - **They act like a temporary "data tunnel" (e.g., parent → child).**
  - **Example: In Linux, created via pipe() syscall before fork().**

Question 2: Synchronization Focus

Q: ***A critical resource (e.g., a printer) can only serve one process at a time. Which synchronization tool ensures exclusive access? How does it work?***

Solution:

- **Tool: Mutex (🟢).**
- **How it works:**
  1. **A process locks the mutex before using the resource.**
  2. **If another process tries to lock it, it waits (blocks).**
  3. **When done, the process unlocks the mutex.**
- ***Analogy:*** **Like a single key to a bathroom – only the key holder can enter.**

Question 3: Combined IPC + Synchronization

Q: ***Two processes use Shared Memory (🔵) to edit the same file. Why might they corrupt data? Which synchronization tool (🟢) fixes this?***

Solution:

- **Problem:**
  Both processes might write to the same memory region simultaneously, causing a race condition (e.g., overwriting each other's data).
- **Synchronization Tool: Mutex or Semaphore (🟢).**
- **How it helps:**
  - **Processes lock the mutex/semaphore before editing.**
  - **Only one process accesses the shared memory at a time.**
  - **Example:**

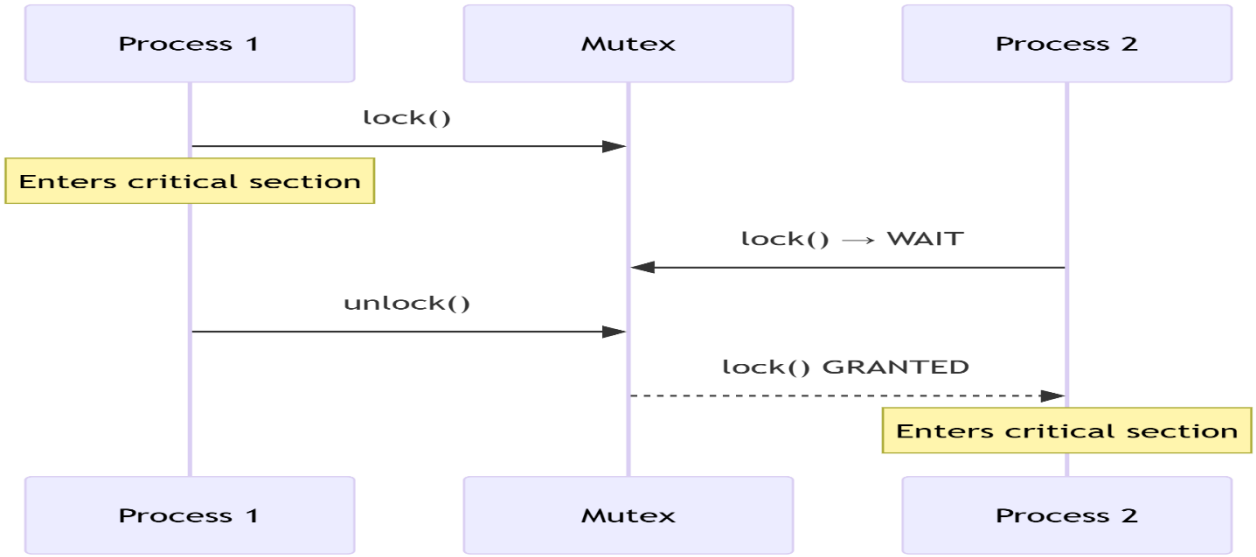
Code:

```
lock(mutex);      // Enter critical section
edit_shared_file();
unlock(mutex);    // Exit
```

Summary for Learners:

1. **IPC Tools (🔵) = *Communication* (e.g., Pipes, Shared Memory).**
2. **Sync Tools (🟢) = *Coordination* (e.g., Mutex for exclusive access).**
3. **Shared Memory requires Sync – Always pair 🔵 with 🟢 to prevent chaos!**

2.



Question 1: Mechanism

Q: *Process A locks a mutex to access a shared printer. What happens when Process B attempts to lock the same mutex?*
Solution:

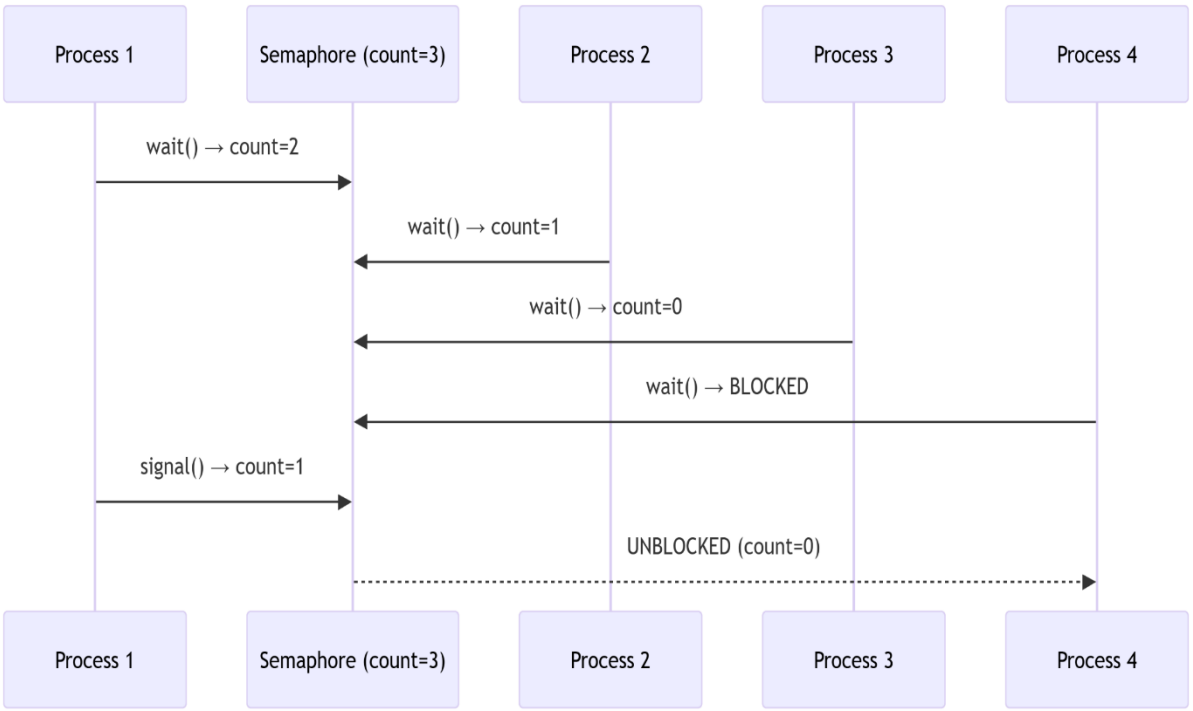
- Process B **blocks (waits)** until Process A unlocks the mutex.
- *Why?* Mutexes enforce strict mutual exclusion – only one process can hold the lock at a time.

Question 2: Printer Use Case

Q: *Why must a printer use a mutex instead of allowing concurrent access? What happens without it?*
Solution:

- **Why?** Printers cannot handle mixed data from multiple processes (e.g., half Page 1 from Process A + half Page 1 from Process B = garbage).
- **Without Mutex:**
    - Processes print simultaneously → **corrupted output**.
    - Example: A letter and a report print blended into unreadable gibberish.

Moderate Learner Learner:--



3.

Question 1: Resource Allocation

Q: *The semaphore starts with 3 resources. After Processes 1, 2, and 3 each acquire a resource via wait(), how many resources remain? What happens when Process 4 tries to acquire one?*
Solution:

- **Resources left: 0** (3 - 3 = 0).
- **Process 4: Blocks immediately** because no resources are available.
- *Key concept:* wait() is a **resource request** – it fails if count=0.

Question 2: Signal Operation

Q: *After Process 4 is blocked, Process 1 releases its resource with signal(). Why does the semaphore count temporarily become 1 but then drop to 0? What happens to Process 4?*
Solution:

1. **Process 1's** signal()**:**
    - Releases 1 resource → count increases from **0 → 1**.

2. **Process 4:**
    - Is **unblocked automatically** and immediately consumes the resource.
    - Performs wait() → count decreases from **1 → 0**.

- *Why?* signal() releases a resource AND wakes a waiting process.

3.

Which of the following are not shared by the threads of the same process?
(a) Stack
(b) Registers
(c) Address Space
(d) Message Queue

1. (a) and (d)
2. (b) and (c)
3. (a) and (b)
4. More than one of the above
5. None of the above

→ **Solution:--**

Option 3 : (a) and (b)

Operating Systems Question 3 Detailed Solution

- A thread is a lightweight process that can be managed independently by a scheduler.
- A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier.
- Threads share the code and data segments and the heap, but they don't share the stack and registers. Message queue comes under data and therefore it is shared by thread.
- Threads of the same process don't share stack and register, hence option 3 is correct

1.In the shared memory diagram, if Process A writes data and Process B reads *at the same time* without synchronization, what problem occurs?

Answer:
Data corruption – Process B might read partially updated/wrong values.

Why? (Simple Explanation)

1. **Write Operation isn't instant**:
    - Process A takes time to write all data (e.g., 1 second).
2. **Reads can happen mid-write**:
    - Process B might read **half-old + half-new data**.

Fix: Use a **mutex** so Process B waits until Process A finishes writing!

Fast Learner Activity:--

1.

Consider the following threads, T_1, T_2, and T_3 executing on a single processor, synchronized using three binary semaphore variables, S_1, S_2, and S_3, operated upon using standard $wait()$ and $signal()$. The threads can be context switched in any order and at any time.

T_1	T_2	T_3

while(true){
wait(S_3);
print("C");
signal(S_2); }
	while(true){
	wait(S_1);
	print("B");
	signal(S_3); }
		while(true){
		wait(S_2);
		print("A");
		signal(S_1); }

Which initialization of the semaphores would print the sequence BCABCABCA ...?

→

Initialization:

$S_1 = 1$, $S_2 = 0$, $S_3 = 0$

Execution Order:

1. T_2 runs first (waits on $S_1=1$ → prints **B** → signals S_3).
2. T_1 runs next (waits on $S_3=1$ → prints **C** → signals S_2).
3. T_3 runs last (waits on $S_2=1$ → prints **A** → signals S_1).
4. Repeat: **B → C → A → B → C → A...**

Why?

- $S_1=1$ allows T_2 (prints **B**) to start.
- $S_2=0$ and $S_3=0$ block T_3 and T_1 until signaled.
- Cycle: $T_2 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2$... produces **BCA** repeatedly.

2.

Consider the following two threads T1 and T2 that update two shared variables a and b. Assume that initially a = b = 1. Though context switching between threads can happen at any time, each statement of T1 or T2 is executed atomically without interruption.

T1	T2
a = a + 1;	b = 2 * b;
b = b + 1;	a = 2 * a;

Which one of the following options lists all the possible combinations of values of a and b after both T1 and T2 finish execution?

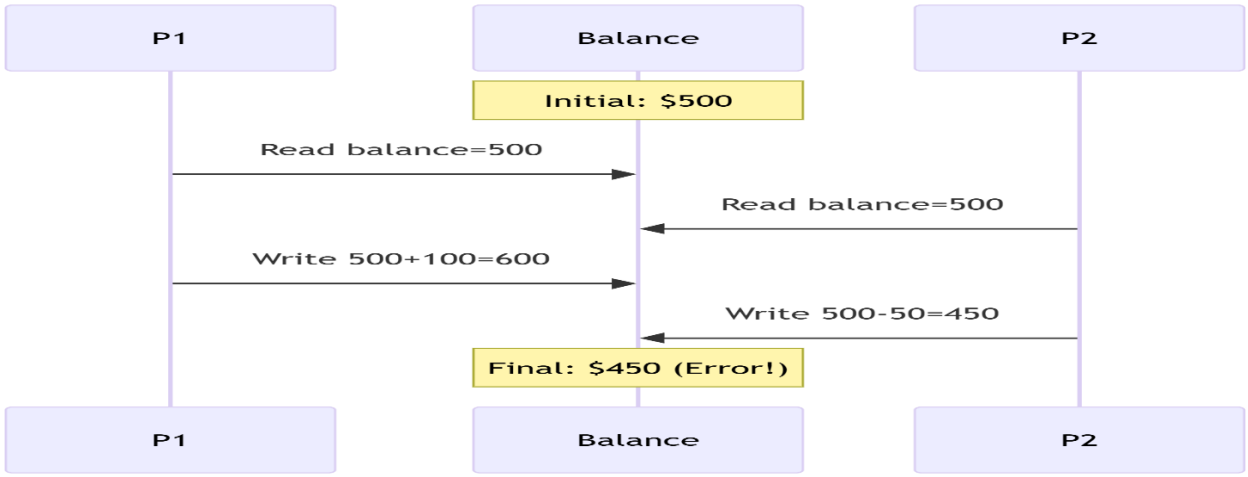
Possible final values of (a, b):

1. **(3, 3)**
   - Occurs when T2 executes fully before T1:
     `b = 2*1 = 2` → `a = 2*1 = 2` → `a = 2+1 = 3` → `b = 2+1 = 3`
2. **(4, 3)**
   - Occurs in 4 interleavings (e.g., partial overlap):
     `b = 2*1 = 2` → `a = 1+1 = 2` → `a = 2*2 = 4` → `b = 2+1 = 3`
3. **(4, 4)**
   - Occurs when T1 executes fully before T2:
     `a = 1+1 = 2` → `b = 1+1 = 2` → `b = 2*2 = 4` → `a = 2*2 = 4`

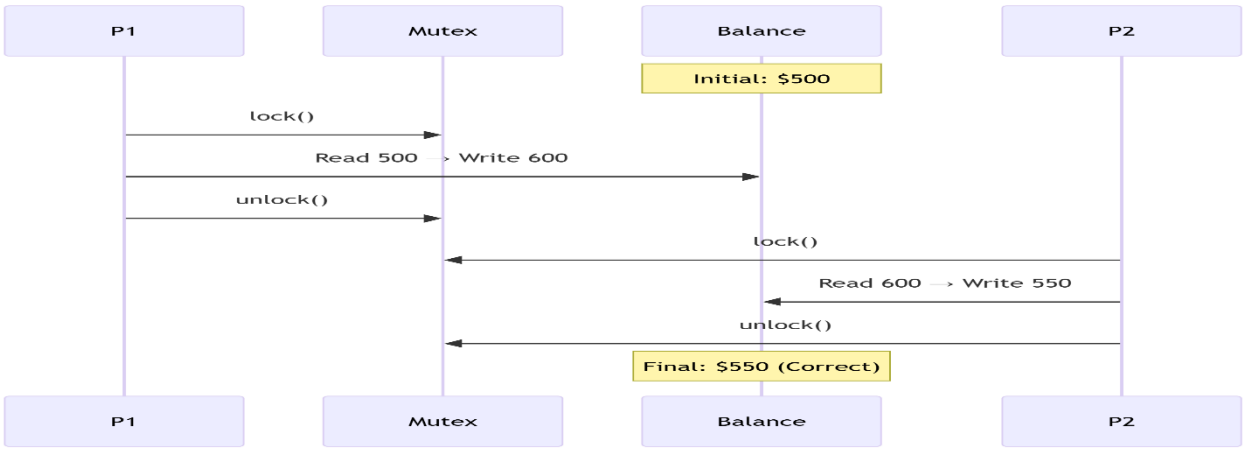
Impossible outcome: (3, 4)

- No execution order produces this combination.

3.**No Synchronization**:--



With Mutex:--



Based on the diagrams:

1. **Why does the first diagram result in ticket=-1 and double booking?**
2. **How does the mutex in the second diagram prevent this?**
3. **What synchronization principle does this scenario demonstrate?**

→ **Answer Summary:**

1. **Race Condition:** Both users see ticket_available simultaneously before either reserves.
2. **Mutex Serialization:**
    - User 1 locks → completes entire transaction → unlocks.
    - User 2 sees ticket=0 after lock.
3. **Principle: Mutual Exclusion** - Only one process accesses critical section (ticket reserve) at a time.

Without synchronization: Corrupt data (negative tickets).
With synchronization: Consistent state (no overbooking).