UNIT-5

TASK COMMUNICATION

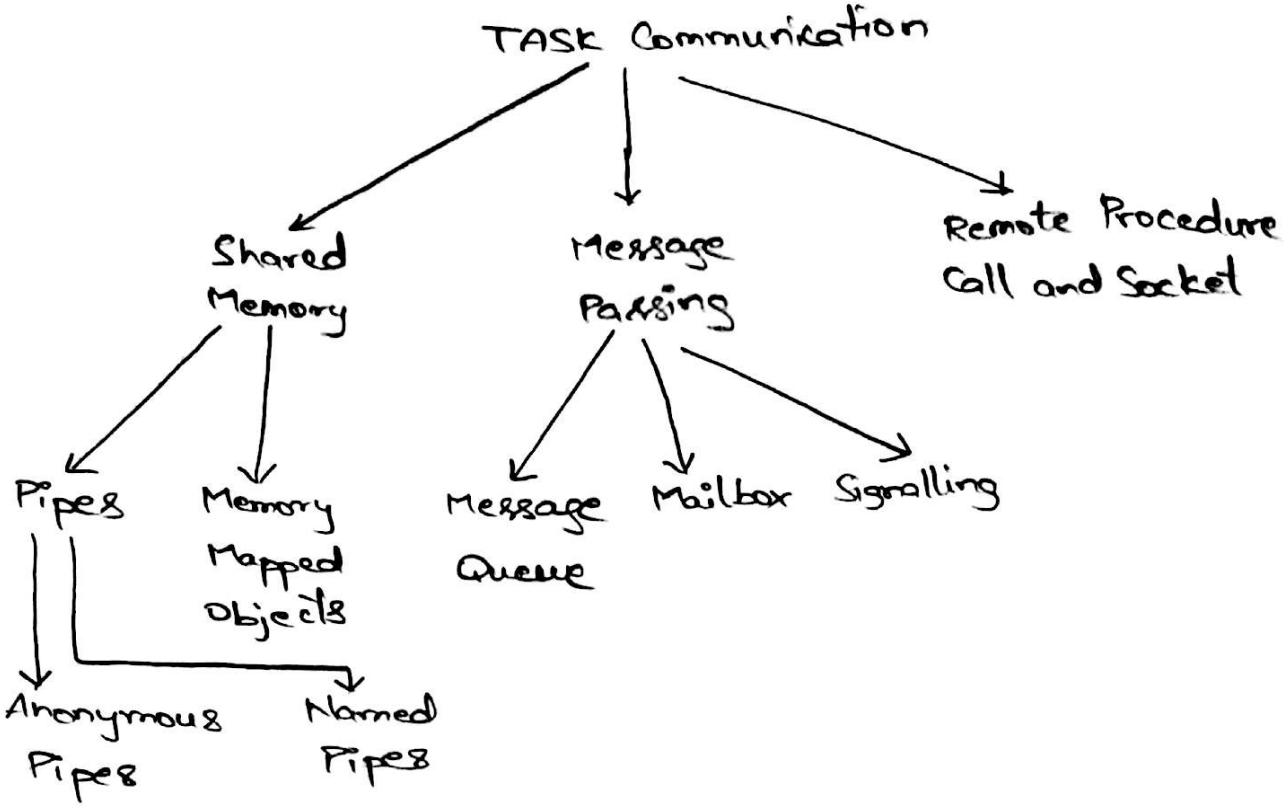
→ **Task Communication :-**

In a multitasking sys, multiple task runs concurrently and each in a process may or may not interact between. Based on the degree of interaction the processes running on an OS are classified as.

* **Co-Operating Processes** :- Here one process needs its o/p from other processes to complete its execution.
* **Competing Processes** :- This process do not share any thing among themselves but they share system resources.
* **Co-Operation Through Sharing** :- This process exchanges data through some shared resources.
* **Co-Operation Through Communication** :- No data is shared between processes. But they communicate for synchronization.

The process or

The mechanism through which process communicate with each other is known as Inter Process Communication (IPC). The various types of IPC mechanisms adopted by a process are kernal (O.S) dependent.



① Shared Memory :—

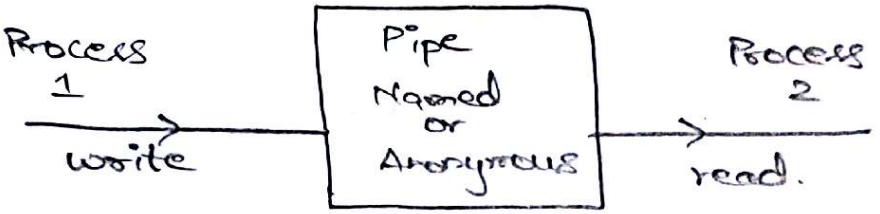
- * Processes shares some area of the memory to communicate among them.
- * Information to be communicated by the process is written to the shared memory area. Other processes which require this information can read the same from the shared memory area.

Process 1	Shared Memory Area	Process 2

- * The implementation of shared memory concept is kernal dependent.
- * Different mechanisms are adopted by different kernels for implementing this.

(1.1) Pipes :-

* It is a section of shared memory used by process for communicating.
* Pipes follow client-server architecture.
* A process which creates a pipe is called as pipe server and a process which connects to a pipe is called as pipe client.

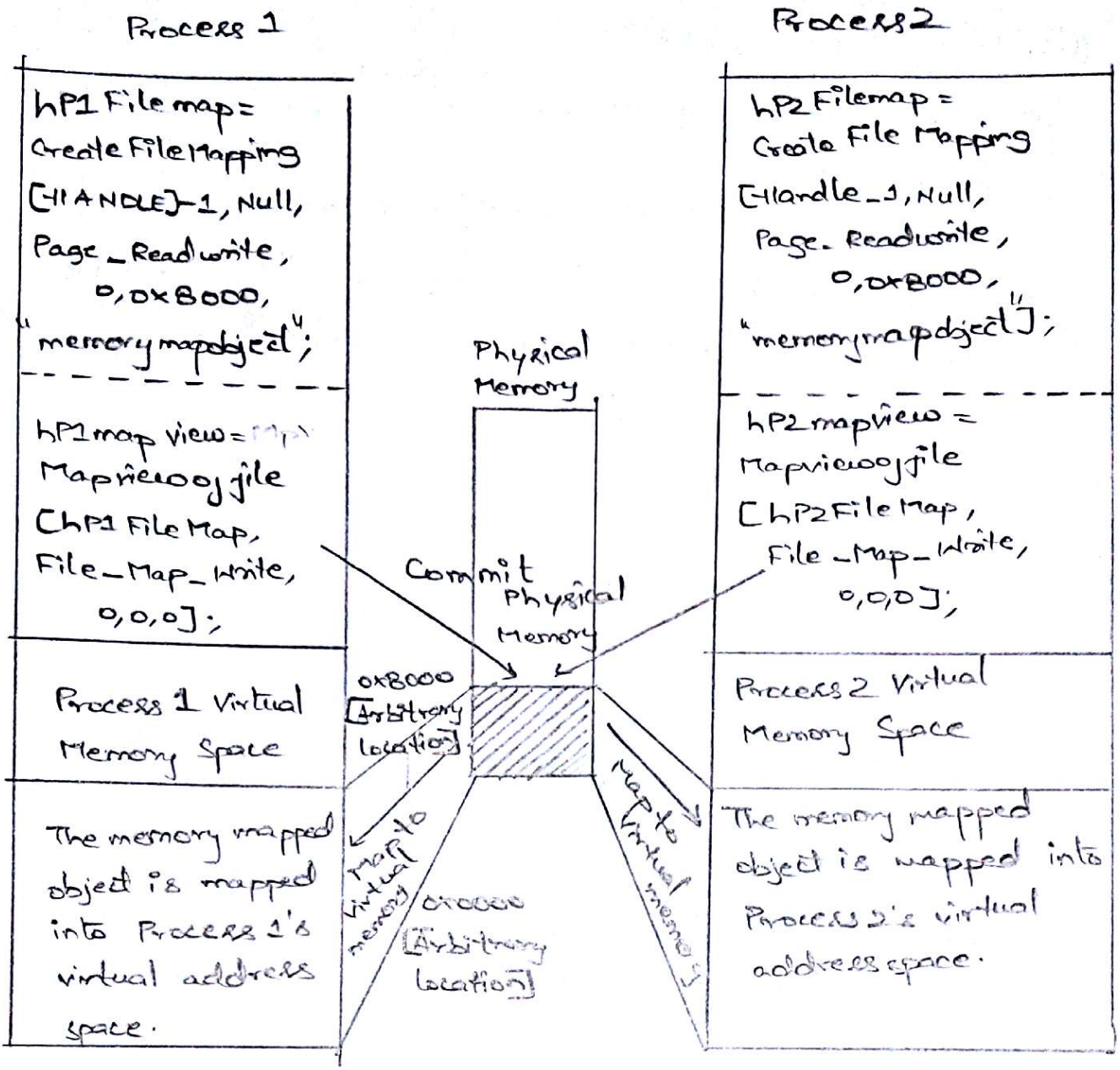


Anonymous Pipe :- The anonymous pipes are unnamed, undirectional pipes used for data transfer between two processes.

Named Pipe :- The named pipe are named, uni-directional or bi-directional for data exchange between two processes.

(1.2) Memory Mapped Objects :-

* Memory mapped object is a shared memory technique adopted by some RTOS for allocating a shared block of memory which can be accessed by multiple process simultaneously.
* In this approach a mapping object is created and physical storage for it is reserved and committed. A process can mappe the entire committed physical area or a block of it to its virtual address space.



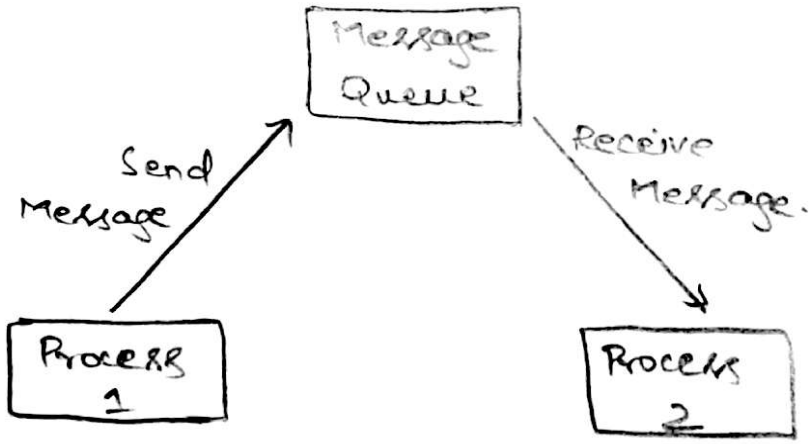
For using memory mapped object across multiple threads of a process, it is not required for all the threads of the process to create/open the memory mapped object and map it to the threads virtual address space. Since the thread's address space is part of the virtual address space, which contains the thread, only one thread, preferably the parent thread is required to create the memory mapped object and map it to the process's virtual address space.

(2) Message Passing :-

* Message passing is an
  - (a) Synchronous information exchange mechanism used for Inter Process Communication.
* It is relatively fast and free from the synchronization overheads compared to shared memory.
* Based on the message passing operation between the processes, message passing is classified as following.

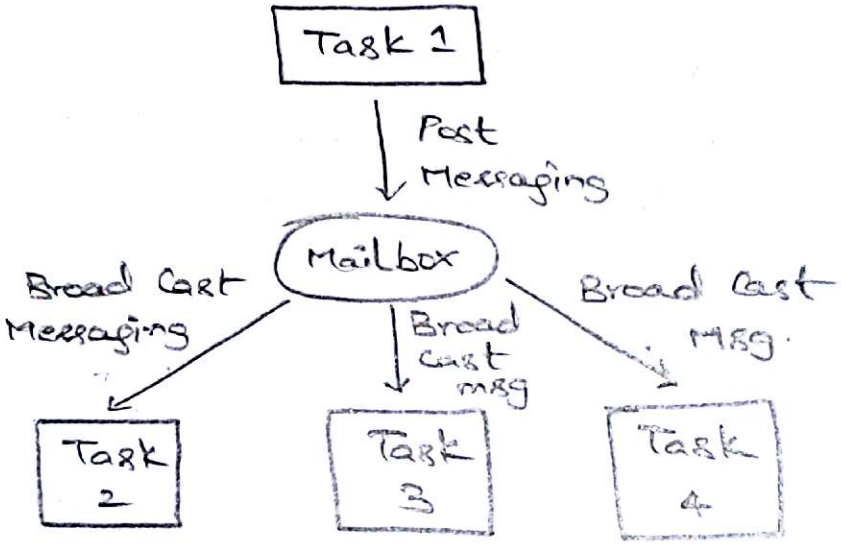
(2.1) Message Queues :-

* Usually the process which wants to talk to other process posts the message to First In First Out queue called Message Queue.
* This stores messages temporarily in a system defined memory object, to pass it to the desired process.
* Messages are sent and received through send [Name of the process to which the message is to be sent, message] and receive [Name of the process from which the message is to be received, message] methods.



(2.2) Mailbox :-

* It is an alternative form of message queues and it is used in certain RTOS for IPC.
* Mailbox technique for IPC in RTOS is usually used for one way messaging.
* The task/thread which wants to send a message to other task/thread creats a mailbox for porting the messages.

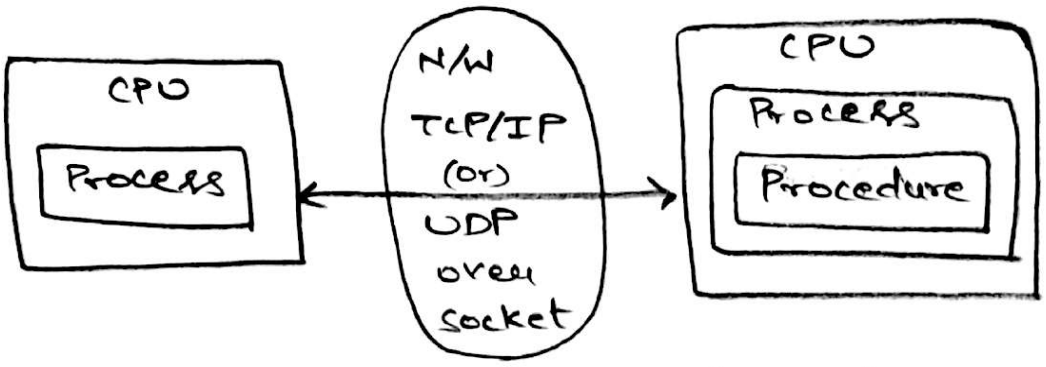


(2.3) Signalling :-

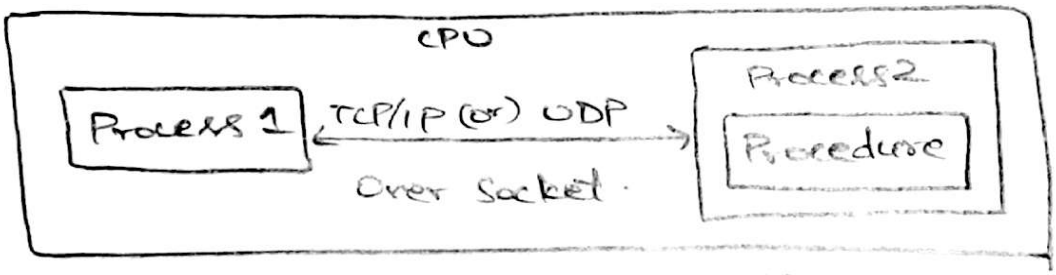
* It is a primitive way of communication between process and threads.
* Signals are used for asynchronous notifications where one process fires a signal, indicating the occurence of a scenario which the processes are waiting.
* Signals are not queued and they do not carry any data.

③ Remote Procedure Call and Sockets :-

* RPC is the interprocess communication (IPC) mechanism used by a process to call a procedure of another process running on the same CPU or on a different CPU which is interconnected in a N/W.



Process running on different CPU's which are networked.



Process Running On Same CPU.

* In the object oriented language terminology RPC is also known as Remote ~~Invocation~~ Invocation (or) Remote Method ~~Invocation~~ Invocation (RMI).

* RPC is mainly used for distributed applications like client server applications.

* With RPC it is possible to communicate over a heterogeneous N/W [ie; N/W where client and server applications are running on different O.S].

* Sockets are used for RPC communication. Socket is a logical endpoint in a 2-way communication link between 2 applications running on a N/W.
* Sockets are of different types, namely, Internet Socket (INET), UNIX socket, etc. INET works on internet communication protocol. They are classified as
  ① Stream socket.
  ② Datagram socket.

→ Task Synchronization :-

* The act of making a processes aware of the access of shared resources by each process to avoid conflicts is known as Task Synchronization.
* Various synchronization issues may arise in a multitasking environment if processes are not synchronised properly.

Task Synchronization Issues :-

① Racing :- Race condition is the situation in which multiple process compete with each other to access and manipulate shared data concurrently. Let us have a look at the following piece of code.

```
#include <windows.h>
#include <stdio.h>
// counter is an integer variable and buffer is a byte
// array shared between 2 processes A and B
```

```
char buffer [10] = {1,2,3,4,5,6,7,8,9,10};
short int counter = 0;

//Process A
void process_A (void) {
int i;
for (i=0; i<5; i++)
   {
      if (Buffer [i] > 0]
       counter++;
      }
   }.

//Process B
void process_B (void) {
int j;
for (j=5; j<10; j++)
 {
   if (Buffer [j] > 0]
     counter++;
   }
}

//main thread.
int main () {
    DWORD id;
Create thread (Null, 0,
          (LPTHREAD_START_ROUTINE) Process_A,
          (LPVOID) 0, 0, &id);
```

```
max eax, dword ptr [ebp-4]
add eax, 1
mov dword ptr [ebp-4], eax

mov eax, dword ptr [ebp-4]
add eax, 1
mov dword ptr [ebp-4], eax.
```

```
Create thread (NULL, 0,
                (LPTHREAD_START_ROUTINE) Process_B,
                (LPVOID) 0, 0, &id);

Sleep (100000);
return 0;
```

② Deadlock :-

A race condition produces incorrect results whereas a deadlock generates a situation where none of the processes are able to make any progress in their execution, resulting in a set of deadlocked processes. The diff cond favouring deadlock situation are

Mutual Exclusion :- The criteria that only one process can hold resource at time. Typical example is the accessing of display H/W in an embedded devices.

Hold And Wait :- The condition under which the process holds the shared resources by aquiring the lock controlling the shared access and waiting for additional resources held by other process.

No Resource Preemption :- The criteria that operating system cannot take back the resource from the process which is currently holding it and the resource can only be released voluntarily by the process holding it.

Circular Wait :- A process is waiting for a resource which is currently held by another process which in turn is held by the resource held by the first process.

Deadlock Handling :- A smart OS may foresee the deadlock precondition and act proactively to avoid such situations. The OS may adopt any of the following techniques to detect and prevent dead lock occurence.

(a) Ignore deadlocks
(b) Detect and recovers
(c) Avoid deadlock
(d) Prevent deadlock

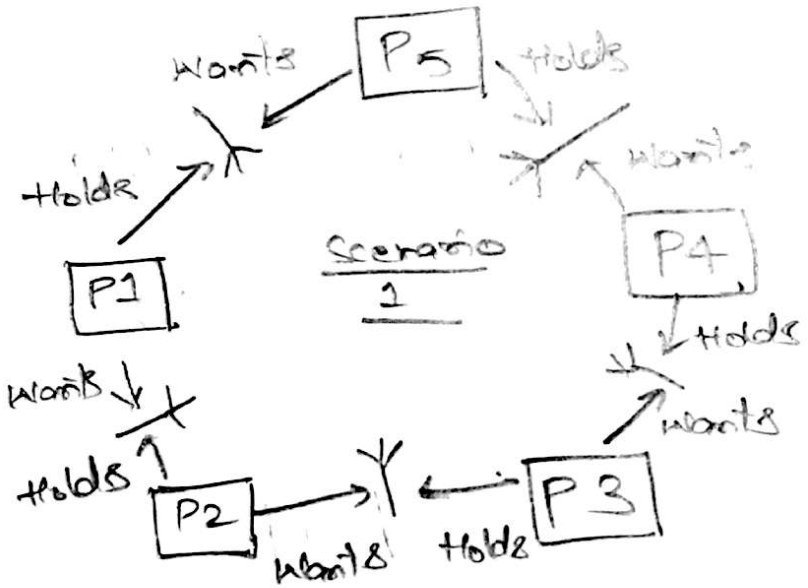
(3) Dinning Philosopher's Problem :-

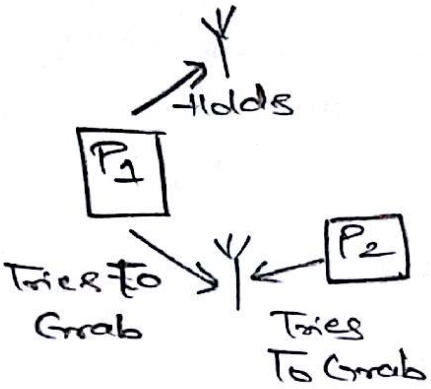
This is an synchronization problem in resource utilization. Lets analyse the various scenarios that may occur in this situation.

Scenario 1 :- All the philosophers are involved in brainstorming together and try to eat together. Each philosopher picks up the left fork and unable to proceed as 2 forks are required for eating. Philosopher 1 thinks that philosphere 2 sitting to his right will put the fork down and wait for it. and so on for remaining philosopheers. This condition results in starvation and deadlock.

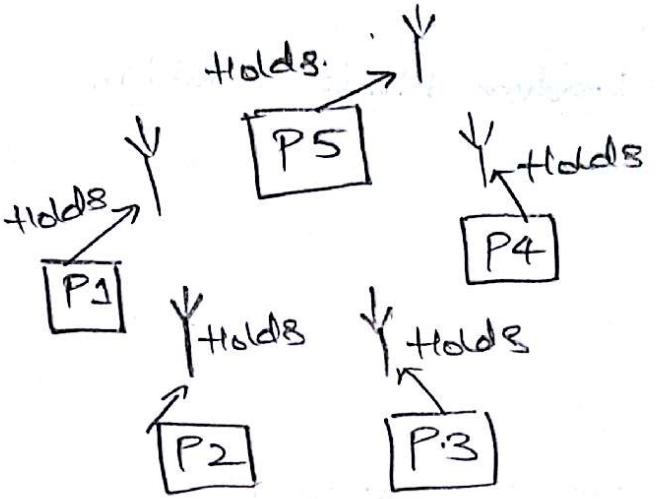
Scenario 2:- All the philosophers start brainstorming together. One of the philosophers is hungry and he starts early by picking up the left fork. When the philosophers tries to pickup the right fork, the philosopher sitting to his right also feels hungry and tries to grab the left side fork which is right fork to his neighbouring philosopher who is trying to lift it, resulting in a 'race condition'.

Scenario 3:- All the philosophers involve in a brain storming together and try to eat together. Each philosopher picks up the left fork and is unable to proceed, since 2 forks are required for eating. Each of them anticipates that the adjacently sitting philosphers will put his fork down and waiting for a fixed duration. This condition leads to livelock and starvation.





Scenario-2



Scenario-3

④ Producer – Consumer / Bounded Buffer Problem:-

It is a common data sharing problem when 2 process concurrently access a shared buffer with fixed size. A process which produces data is called "Producer Process" and process which consumes data is called Consumer Process. The situation under which the producer process more ~~data~~ rate and consumer process consumes at slower data rate, then this will lead to buffer overrun. Similarly if the consumer consumes the data at a faster rate, ~~then the~~ and the data is produced at lesser rate then this will lead to buffer ~~or~~ underrun.

The different situation which may arise based on scheduling of the producer threads and consumer threads are listed as below.

ⓐ Producer threads are scheduled more frequently then consumer threads.

ⓑ Consumer threads are scheduled more frequently then producer threads.

⑤ Readers-Writers Problem :-

This is a common issue observed in processes competing for limited shared resources. They are characterised by multiple shared processes trying to read and write shared data concurrently.

⑥ Priority Inversion :-

It is the byproduct of the combination of blocking based process synchronisation and pre-emptive priority scheduling. Priority Inversion is the condition in which a high priority task and low priority task and a medium priority task which doesn't require the shared resource continue its execution by preempting the low priority task.

Priority Inheritance :- A low priority task that is concurrently accessing a shared resources requested by high priority task temporarily "inherits" the priority of that high-priority task, from the moment that the high priority task arises the request.

Priority Ceiling :- In priority ceiling a priority is associated with each shared resource. The priority associated to each resource is the priority of highest priority task which uses this shared resources. This priority is called ceiling priority. Whenever a task access a shared memory resources, the scheduler elevates the priority of the task to that of the priority of the resource.

→ Task Synchronization Techniques :-

* Process / task synchronization is essential for.

1) Avoiding conflicts in resource access. [racing, deadlock, starvation, livelock, etc] in a multitasking environment.

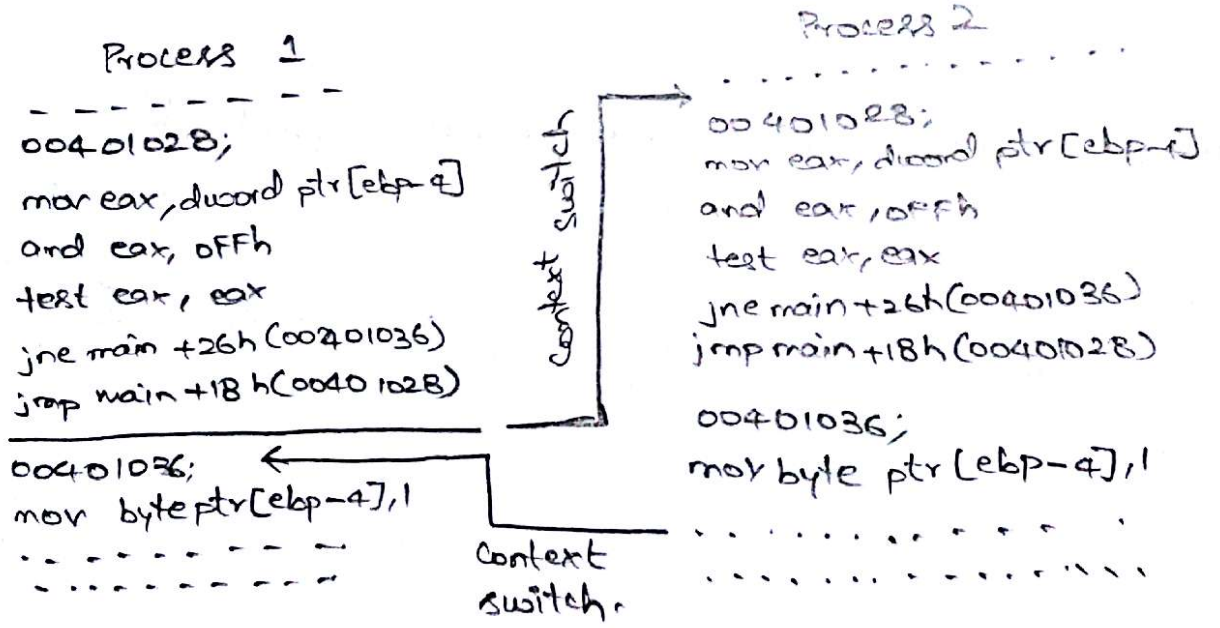
2) Ensuring proper sequence of operations across processes. The producer consumer problem is a typical example of processes requiring proper sequence of operation. In producer consumer problem, accessing the shared buffer by different processes is not the issue, the issue is the writing process should write to the shared buffer only if the buffer is not full and the consumer thread should not read from the buffer if it is empty.

3) Communication btw processes. The code memory area which holds the program instructions for accessing a shared resources, is known as critical section. The exclusive access to critical section of code is provided through mutual exclusion mechanism. Mutual exclusion blocks a process. Based on the behaviour of the blocked process, mutual exclusion methods can be classified into 2 categories.

Mutual Exclusion Through Busy Waiting / Spin lock :-

* Busy waiting is the ~~str~~ simplest method for enforcing mutual exclusion. ~~I~~
* The busy waiting technique uses a lock variable for implementing mutual exclusion.

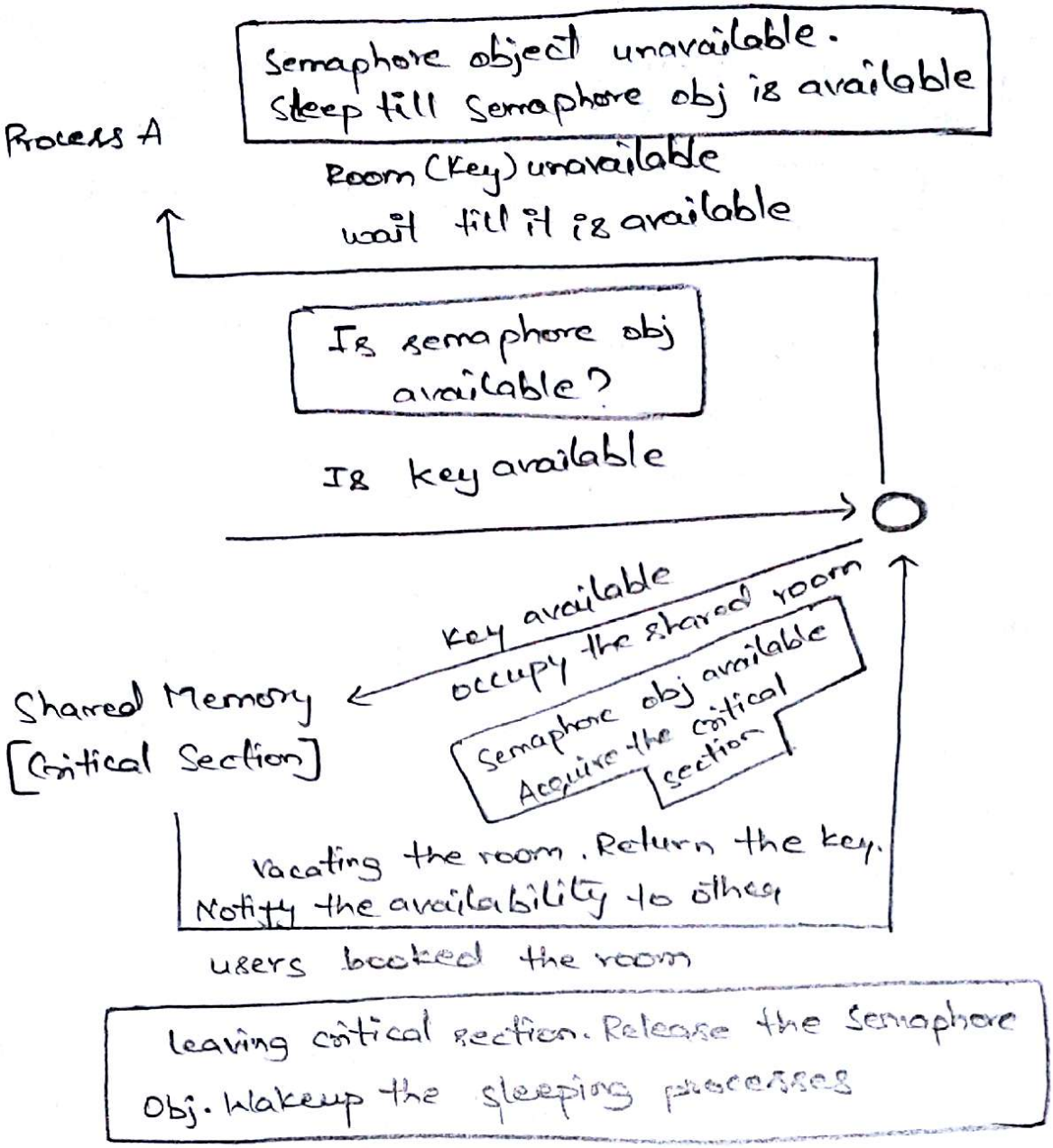
* Each process/thread checks this lock variable before entering the critical section.
* The lock is set to 1 by a process if process is already in its critical section, otherwise the lock is set to 0.
* The major challenge in implementing the lock variable based synchronization is the non-volatile/non-availability of a single atomic instruction, which combines the reading, comparing and setting of the lock variable.
* Most of the 3 operations related to the locks, viz. the operation of reading the lock variable, checking its present value and setting it are achieved with multiple low level language/instructions.
* The low level implementation of these operations are dependent on the underlying processor instruction set and the compiler in use.
* The low level implementation of the busy waiting code snippet, which we discussed earlier, under windows XP operating system running on an Intel Centrino Duo processor is given below:



Mutual Exclusion Through Sleep and Wakeup :—

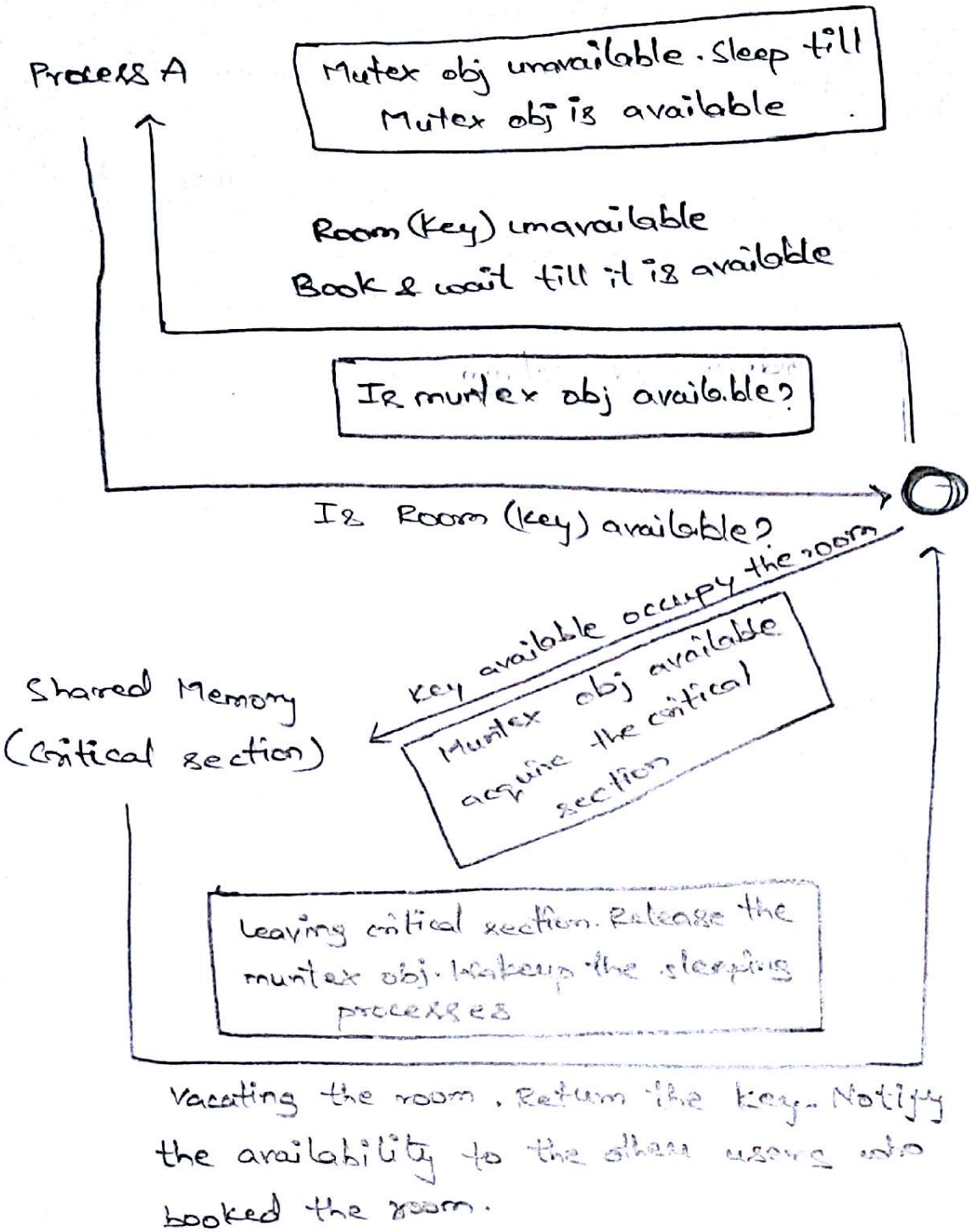
* The busy waiting mutual exclusion enforcement mechanism used by processes makes the CPU always busy by checking the lock to see whether they can proceed.
* This results in the wastage in the CPU time and leads to high power consumption.
* An alternative to busy waiting is the sleep and wakeup mechanism.
* When a process is not allowed to access a critical section, which is currently being locked by another process, the process undergoes sleep and enters the blocked state.
* The process which is blocked on waiting for access to the critical section is awakened by process which currently owns the critical section.
* The process which owns the critical section sends a wakeup message to the process, which is sleeping as a ~~process~~ result of waiting for the access to the critical section, when the process leaves the critical section.

* **Semaphore** :— Semaphore is a sleep and wakeup based mutual exclusion implementation for shared resource access. Semophore is a system resource and the process which wants to access the shared resource can first acquire their system object to indicate the other processes which wants the shared resource that the shared resource is currently acquired by it.



* Binary Semaphore [Mutex]:-

* Binary Semaphore is a synchronisation object provided by OS for process/thread synchonisation.
* Any process/thread can create a "mutex object" and other processes/threads of the system can use this "mutex object" at a time.
* The state of a mutex object is set to signalled when it is not owned by the any process/thread and set to non-signalled when it is owned by any process/thread.



* Critical Section Objects :-

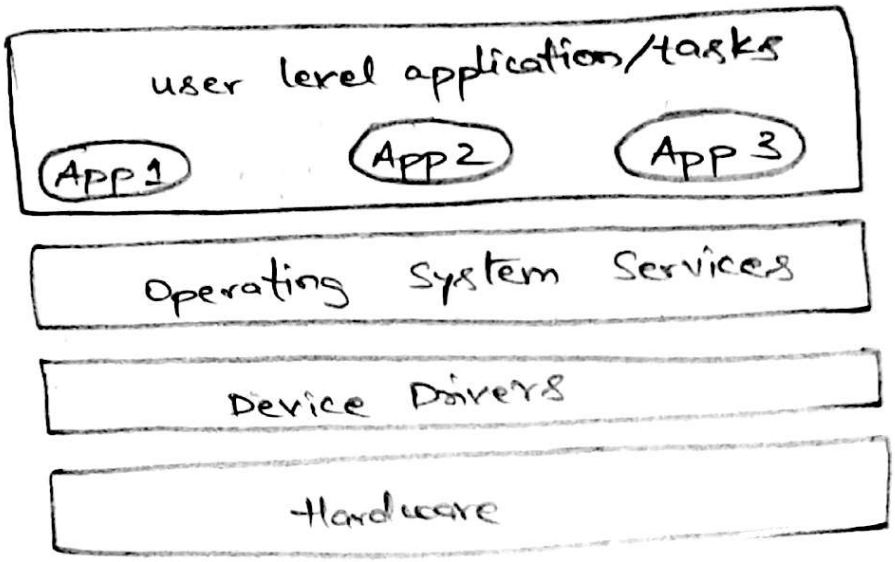
* In windows CE, the critical section obj is same as mutex obj except that critical section obj can only used by the threads of a single process.

* The piece of code which needs to be made as critical section is placed as the critical section area by the process.
* The memory area which is to be used as the critical section is allocated by the process.
* The process creates a critical section area by a creating a variable of type critical_section.
* The critical section must be initialized before the threads of a process can use it for getting exclusive access.
* The initialize critical section API initialises the critical section pointed by the pointer IP critical section to the critical section.

* Events:-
* Event obj is a synchronization technique which uses the notification mechanism for synchronisation.
* In concurrent execution we may come across situations which demand the processes to wait for a particular sequence for its operations.
* A typical example of this is the producer consumer threads, where the consumer thread should wait for the consumer thread to produce the data and produce thread should wait for the consumer thread to consume the data before producing fresh data.

→ Device Drivers :—

* Device drivers is a piece of software, that acts as a bridge between the OS and the hardware.
* In an OS based product architecture, the user application talk to the OS and kernal all necessary information exchange including communication with the H/W peripherals.



* The architecture of the OS kernal, will not allow direct device access from the user applications.
* All the device related access should flow through the OS kernal and the OS kernal routes it to the concerned hardware peripheral.
* OS provides interfaces in the form of application programming interfaces for accessing the hardware.
* The device driver abstracts the hardware from user applications.
* Device drivers are responsible for initiating and managing the communication with the hardware peripherals.

* They are responsible for establishing the connectivity, initialising the H/w and transferring data.
* An embedded product may contain different types of hardware components like wifi - module, file systems, storage device interface, etc.
* The initialisation of these devices and the protocols required for communicating with these devices may be different.
* All their requirements are implemented in drivers and a single driver wont be sufficient to satisfy all these. Hence each driver requires a unique driver component.
* Certain drivers come as part of the OS kernal and certain drivers need to be installed on the fly.
* A device driver implements the following
  → Device initialisation and interrupt configuration.
  → Interrupt handling and processing.
  → Client interfacing.
* The basic interrupt configuration involves the following.
  → Set the interrupt type (or) level triggered, enabled the interrupts and set the interrupt priorities.
  → Bind the interrupt with an interrupt request. The processor identifies an interrupt through IRQ.

→ How To Choose An RTOS :-

* The decision for choosing an RTOS for an embedded design is very crucial.
* A lot of factors need to be analysed carefully before making a decision on the selection of an RTOS.
* These functions can be either functional or non-functional.

Functional Requirements :-

* Processor Support :- It is not necessary that all RTOS's support all kinds of processor architecture. It is essential to ensure that the processor support by the RTOS.
* Memory Requirements :- The OS requires ROM memory for holding the OS files and it is normally stored in a non-volatile memory like Flash.
* Real-Time Capabilities :- It is not mandatory that the OS for all embedded system need to be real-time and all embedded operating system are real time behaviour.
* Kernel and interrupt latency :- The kernal of the OS may disable interrupts while executing certain services and it may lead to interrupt latency.
* Modularisation Support :- Most of the OS provide a bunch of features. At times it may not be necessary for an embedded product for its functioning

* Inter Process Communication And Task Synchronization :-

* The implementation of inter process communication and synchronisation is OS kernal dependent. Certain kernals may provide a bunch of options whereas others provide very limited options.

* Support For Networking And Communication :- The OS kernal may provide stack implementation and drivers support for a bunch of communication interfaces and networking.

* Development Language Support :- Certain OS include the run time libraries required for running applications written in languages like JAVA and C++.

Non-Functional Requirements :-

* Custom Developed or Off the Shelf :- Depending on the OS requirement, if it is possible to go for the complete development of an OS suiting the embedded system needs or use an off the shelf, readily available operating system, which is either a commercial product or an open source product, which is in close match with the system requirements.

* Cost :- The total cost for developing or buying the OS and maintaining it in terms of commercial product and custom build needs to be evaluated before taking a decision on the selection of OS.
* Development and Debugging Tools Availability :- The availability of development and debugging tools is a critical decision making factor in the selecton of an OS for embedded design.
* Ease Of Use :- How easy it is to use a commercial RTOS is another important feature that needs to be considered in the RTOS selection.
* After Sales :- For a commercial embedded RTOS, after sales in the form of e-mail, on-call services etc. for bux fixes, critical patch updates and support for production issues etc, should be analysed thoroughly.