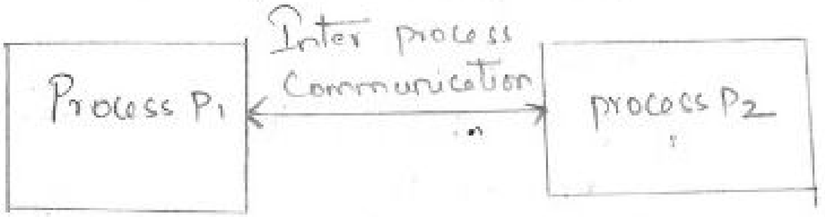
What is Interprocess Communication?

Interprocess Communication is the mechanism provided by the Operating system that allows process to communicate with each other. This communication could involve a process letting another process know that some event has occurred or the transferring of data from one person to another.



Synchronization in Interprocess Communication.

synchronization is a necessary part of interprocess communication. It is either provided by the interprocess control mechanism or handled by the communicating process. some of the methods to provide synchronization are as follows-

Semaphore- A semaphore is a variable that controls the access to a common resource by multiple processes. The two types of semaphore are binary semaphore and counting semaphore.

Mutual Exclusion: Mutual Exclusion requires that only one process thread can enter the critical section at a time This is useful for synchronization and also prevents race condition.

Barrier: A barrier does not allow individual processes to produced until all the processes reach it. Many parallel languages and collective routines impose barriers.

Spinlock: This is a type of lock. The processes trying t acquire this lock wait in a loop while checking if the lo is available or not this is known as busy waiting becau the process is not doing any useful operations even though it is active.

Approache's to Interprocess communication.

The different approaches to implement interprocess communication are given as follows -

Pipe -- A pipe is a data channel that is unidirectional. Two pipes can be used to create a two-way data channel between two processes.

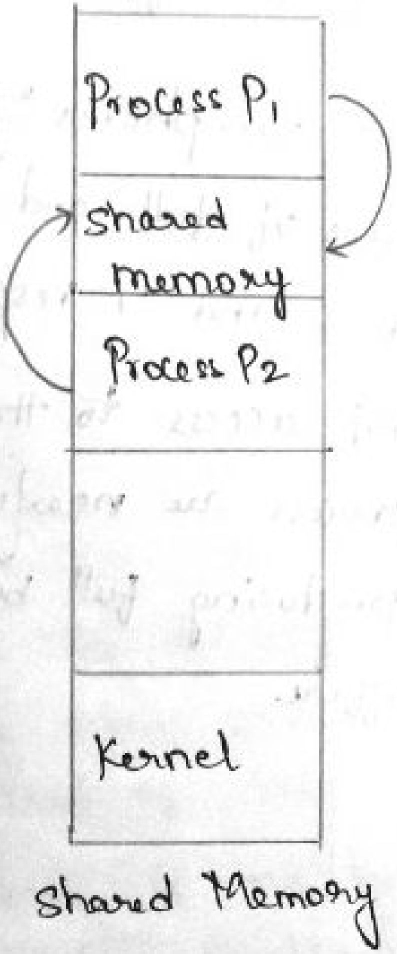
Socket: The socket is the endpoint for sending or recei data in a network. This is true for data sent between processes on the same computer or data sent between different computers on the same network.

File: A file is a data record that may be stored on a disk or acquired on demand by a file server. Multiple processes can access a file as required. All operating systems use files for data storage.

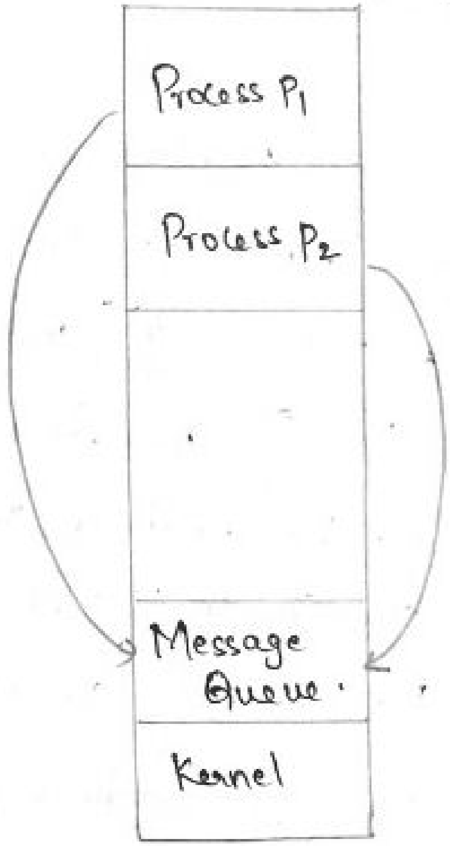
Signal:- Signals are useful in interprocess communication in a limited way. They are system messages that are sent from one process to another. Normally, signals are not used to transfer data but are used for remote commands between processes.

Shared memory: shared memory is the memory that can be simultaneously accessed by multiple processes. This is done so that the processes can communicate with each other

Message Queue: Multiprocesses can read and write data to the message queue without being connected to each other. Messages are stored in the queue until recipient retrieves them. Message Queues are quite useful for interprocess communication and are used by most operating system.



Shared Memory



Message Queue.

What are the different problems of process synchroniz

The following classic problems are used to test Virtu every new proposed synchronization algorithm.

<u>Bounded - Buffer problem</u>:-

* It is a generalization of producer-consumer problem wherein acces is controlled to shared group of buffers of limit size

* In this solution, the two counting semaphores "full" an "empty" Keep track of current numbers of full and empty buffers respectively [and initialized to 0 and N respectively

* The binary semaphore mutex control access to the critic section. The producer and consumer process are nearly identical One can think of the producer as producing full buffers, and the consumer producing empty buffers.

<u>Structure</u> of producer process:-

```
do {
  ......
  // produce an item in next p
  ......
   wait (empty );
   wait (mutex);
   .....
  // add nextp to buffer.
   ....
  signal (mutex);
  signal (full);
} while (true);
```

structure of consumer problem:-

```
do {
    wait (full);
    wait (mutex);
    .....
  // remove an item from buffer to next c
  .......
    signal (mutex);
    signal (empty);
  ........
  // consume the item  in next c
  ......
  } while (TRUE);
```

Readers - Writers Problem:-

* In this problem, there are some process who only read the shared data, and never change it and there are only other process who may change data in addition to or instead of reading it.

* There is no limit to how many readers can access the data, simultaneously but when it comes to writer accessin a data need to be exclusive access.

Variations:

First readers - writers problem.

* It gives priority to readers. In this problem, if a reader wants to access to data, and there is not already writer accessing it, then access is granted to reader

* Solution to this problem can lead to starvation of writers, as there could always be more readers coming

across data.

The second readers writer problem:-

* It gives priority to writers In this problem, when a writer wants to access to data it jumps to the head of the queue.

* All waiting readers are blocked and writer gets acc to the data as soon as it becomes available

* In this solution, readers may be starved by a stead stream of writers.

Example of a first readers-writers problem:-

sturcture of a writer process:-

```
do {
      wait (rw-mutex);
      .....
      /* writing is performed */
      .....
      signal (rw-mutex);
      } while (TRUE);
```

Structure of a reader process:-

```
do {
     wait (mutex);
     read_count++;
     if (read_count == 1)
         wait (rw_mutex);
     signal (mutex);
     .....
    /* reading is performed */
wait (mutex);
read_ count - - ;
if (read_ count = = 0)
    signal (rw_ mutex);
} while (true);
```

- Read count - used by read processes, to count number of readers currently accessing data.
- Mutex - Semaphore used (to block or) only by readers for controlled access to read count.
- rw-mutex: Semaphore used to block or release writers the first reader to access the data will set this lock and last reader exit will release it. The remaining readers do not touch rw-mutex.

* Some hardware implementation provide specific reader-writer locks, which are accessed using an argument specifying which access is requested for reading or writing.

* The use of reader-writer lock is beneficial for situations in which.

i, Process can be easily identified as either reader or writer.

ii, There are significantly more readers than writers, making additional overhead of the reader-writer lock pay off in terms of increased concurrency of readers.

Explain the dining philospher problem:-

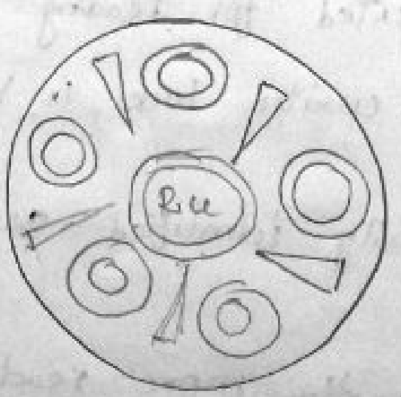
The dining philospher's problem is the classical problem of synchronization which says that five philospher are sitting around a circular table and their job is to think and eat alternatively.

* There is a bowl of rice for each other philosphers a 5 chopsticks.

* A philospher needs both their right and left chopstick to e

* A hungry philospher may only eat if there are both chop sticks available.

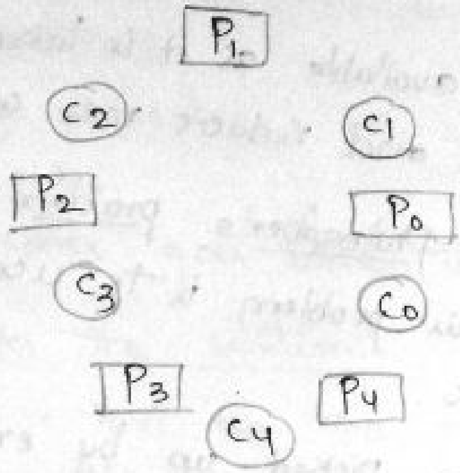
* Otherwise, a philospher puts down their chopstick and begin thinking again.



<u>Dinning</u> <u>Philospher's</u> <u>problem</u>:-

The Dinning philospher's problem can be understood by below code.

* Let P_0, P_1, P_2, P_3, P_4 be five philosphers.
* C_0, C_1, C_2, C_3, and C_4 be chopsticks.



P = philospher
C = chopstick

```
Void philospher
{
  while (1)
   {
     take_chopstick [i];
     take_chopstick [(i+1)%5];
     ......
      EATING THE RICE
      put_chopstick [i];
      Put_chopstick [(i+1)%5];
     .....
      THINKING
     .....
   }
}
```

Suppose philospher p0 wants to eat, it will enter in philospher() function, and execute take_chopstick [i]; by doing this it holds c0 chopsticks after that it executes take_chopstick [(i+1)%5]; by doing this it holds c1 chopsticks (∵ i=0, ∴ (0+1)%5 = 1)

ly.

Suppose new philospher p1 wants to eat it, it will enter in philospher() function, and execute take_chopstick [i]; by doing this it holds c1 chopsticks after that it execute take_chopstick [(i+1)%5]; by doing this it holds c2 chopstick (∵ i=1, ∵ (i+1)%5 = 2)

But,
practically C1 is not available as it is taken by Po, hence
Code generates problems and reduces race condition.

Solution of Dining Philospher's problem:-

* A Solution for this problem is to use a semaphore to represent a chopstick.

* A chopstick can be picked up by executing a wait operation on semaphore and released by executing single semaphore.

structure of chopsticks:-

```
    semaphore chopsticks [5];
```

Initially the elements of chopstick are initialized to 1, as chopsticks are on table and not picked by philospher.

structure of random philospher:-

```
  do {
       wait (chopstick [i]);
       wait (chopstick [(i+1)%5]);
       .....
       EATING RICE
       signal (chopstick [i]);
       signal (chopstick [(i+1)%5]);
       .....
       THINKING
     } while (1);
```

* In this, the first wait operation is performed on chopsticks [i] and chopsticks [(i+1)%5] means philospher i has picked up chopsticks on his sides. Then, The eating function is performed.

* After that, signal operation is performed on chopsticks[i] and chopsticks [(i+1)%5]

This means, the philospher i has eaten and put down chopsticks on his sides.

Then, the philospher goes back to thinking.

<u>Difficulty</u> <u>with</u> <u>the</u> <u>solution</u>:-

* It makes, sure that no two neighbosing philosphers can eat at the same time

* But, this solution can lead to a deadlock.

* This may happen if all the philosphers picks their left chopsticks simultaneously. Then none of them can eat and deadlock occurs.

Some of the ways to avoid deadlock are:-

□ There should be atmost four philospher on table.

□ -An even philospher should pick the right chopsticks and then the left chopstick.

□ -An odd philospher should pick first left chopstick and then the right chopstick.

□ -A philospher should only be allowed to pick their chopsticks if both are available at same time.

Discuss about semaphores.

Semaphores:

Semaphores are integer variables that are used to s the critical section problem by using two atomic operat wait and signal that are used for process synchronization

□ Wait:

The wait operation decrements the value of its argum S, if it is positive. If S is -ve or '0' then no operation performed.

```
wait (s)
{
 while (s<=0);
  s--;
}
```

□ signal:

The signal operation increments the value of its arguments S.

```
signal (s)
{
 s++;
}
```

Types Of Semaphore:-

There are two main types of semaphores.

□ Counting Semaphore:-

* These are integer value semaphore and have an unrestricted value domain

* These Semaphores are used to coordinate the resource access, where the semaphore count is the number of

available resources.

* If resources are added, semaphore count automatically incremented and if resources are removed, count is decremented.

Binary Semaphores.

* The Binary semaphores are like counting semaphores but their value is restricted to 0 and 1.
* The wait Operation only works when the semaphore is 1 and the signal operation succeds when semaphore is 0.
* It is sometimes easier to implement binary semaphores than counting semaphores.

Advantages:-

* It allows only one process into critical section they follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization
* There is no resource wastage because of busy waiting in semaphore as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
* Semaphores are implemented in the machine independent code of microkernel so they are independent.

Dis advantages:-

* These are complicated to the wait and signal operations must be implemented in the correct order to prevent deadlock.
* These are impratical for last scale use as their use lead to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.