Shared Memory

⇒ Interprocess communication is a capability supported by operating systems that allows one process to communicate with another process.

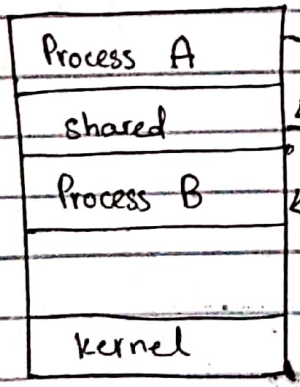
⇒ It is used by the processes to exchange data and information.

⇒ IPC are divided into two types ① Independent processes and ② Cooperating process.

⇒ A process that can affect or be affected by any other process executing in the system is called independent process.

⇒ A process that can affect or be affected by any other proces executing in the system is called cooperating process.

⇒ The shared memory model establishes a region of memory that is shared by cooperating processes.



the region where memory is shared b/w the processes.

⇒ Typically, a shared-memory region resides in the address space of the process creating the shared memory segment. (In the previous example, it lies in the Process A region as A created the shared memory).

⇒ Other processes that wish to communicate using this shared-memory segment must attach it to their address space.

⇒ Normally, the OS tries to prevent one process from accessing another process's memory.

⇒ Shared memory requires that two or more processes agree to remove this restriction.

Producer Consumer Problem :-

⇒ A producer process produces information that is consumed by a consumer process. For example, a compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader.

⇒ We need to make the producer and consumer run concurrently. If the consumer has nothing to consume, then it should wait.

⇒ One solution to the producer-consumer problem uses shared memory.

⇒ To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.

⇒ A buffer is a place where the producer can produce the information and it can be used by the consumer.

⇒ This buffer will reside in a region of memory that is shared by the producer and consumer processes.

⇒ A producer can produce one item and keep it to the buffer while the consumer is consuming another item from the same buffer.

⇒ The producer and consumer must be synchronized so that the consumer does not try to consume an item that has not yet been produced.

Two kinds of buffers

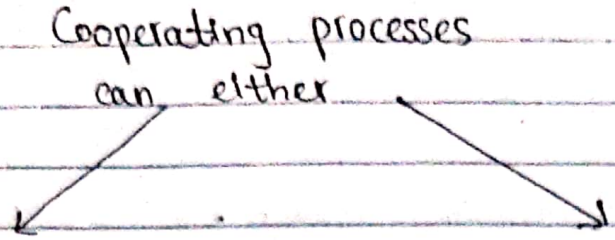
- unbounded buffer
  - places no practical limit on the size of buffer. The consumer may have to wait for new items, but the producer can always produce new items.
- bounded buffer
  - Assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty and the producer must wait if the buffer is full.

Bounded buffer — Shared memory solution :-

```
// shared data
int n = 5 , item , in , out

int buffer [n]

in = 0
out = 0

// Producer Process
while (true)
{
    while (in + 1 % n == out)
    {
        // do nothing
    }
    buffer[in] = nextProduction
    in = in + 1 % n
}

// Consumer process
while (true)
{
    while ( in == out)
    {
        // do nothing
    }
    nextConsumed = buffer[out]
    out = out + 1 % n
}
```

Process Synchronization

⇒ Process synchronization is a cooperating process.



⇒ Concurrent access to shared data may result in data inconsistency.

(The producer Consumer Example)

⇒ We use a counter variable initialized to zero for the producer consumer problem.

⇒ The counter is incremented every time we add a new item to the buffer. (counter++)

⇒ The counter is decremented every time we remove one item from the buffer (counter--).

Example :-

⇒ Suppose that the value of the variable is currently 5.

⇒ The producer and consumer processes execute the statements "counter++" and "counter--" concurrently.

⇒ Following the execution of these two statements, the value of the variable counter may be 4, 5 and 6.

⇒ The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute seperately.

⇒ Counter++ may be implemented in machine language (on a typical machine) as :

$$\begin{aligned} register_1 &= counter \\ register_1 &= register_1 + 1 \\ counter &= register_1 \end{aligned}$$

⇒ Counter-- may be implemented in machine language as :

$$\begin{aligned} register_2 &= counter \\ register_2 &= register_2 - 1 \\ counter &= register_2 \end{aligned}$$

⇒ A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the action takes place is called a race condition.

⇒ Process synchronization is used to eliminate race conditions.

⇒ Process synchronization deals with various mechanisms to ensure orderly execution of cooperating processes that share a logical address space.

The critical Section Problem :-

⇒ Consider a system consisting of n processes $\{P_0, P_1 \ldots P_n\}$.

⇒ Each process has a segment of code, called a critical section.

⇒ In critical section, the process may be changing common variables, updating a table, writing a file and so on.

⇒ When one process is executing in its critical section no other process is to be allowed to execute in its critical section.

⇒ That is, no two processes are executing in their critical sections at the same time.

⇒ The critical-section problem is to design a protocol that the processes can use to cooperate (synchronized with each other properly).

Rules when processes are operating in critical section:

⇒ Each process must request permission to enter its critical section.

⇒ The section of code implementing this request is the entry section.

⇒ The critical section may be followed by an exit section.

⇒ The remaining code is the remainder section.

```
do
{
    entry section
    critical section
    exit section
    remainder section
}
while (true);
```

fig. of a process

Solutions to critical section problem:

① **Mutual exclusion:** If process P_i, is executing in its critical section, then no other processes can be executing in their critical sections.

(2) Progress: If no process is executing in its critical section and some processes wish to enter their critical sections, then only processes that are not executing in their remainder sections can participate in the decision on which it will enter its critical section next, and this selection cannot be postponed indefinitely.

(3) Bounded waiting: There exists a bound or limit on the no. of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted. (IF no limit is put then a process would wait for an indefinite time).

Peterson's Solution:

=> Peterson's solution is restricted to two processes that alternate execution b/w their critical sections and remainder sections. Let's call the processes (P_i) and (P_j).

=> Peterson's solution requires two data items to be shared b/w the two processes.

`int turn` → indicates whose turn it is to enter its critical section.

`boolean flag[2]` → used to indicate if a process is ready to enter its critical section.

Hardware Synchronization:

=> Many systems provide hardware support for critical section code.

- Uniprocessors

How it works in Uniprocesses?

=> could disable interrupt.

=> current running code would execute without preemption.

=> Generally too inefficient on multiprocessor system.

=> These are the some hardware instructions used for synchronization: Test, Set and Swap.

Example:-

```
boolean TestANDSet (boolean* target)
{
    boolean rv = *target;
    *target = true;
    return rv;
}

void swap (boolean* a, boolean* b)
{
    boolean temp = *a
    *a = *b
    *b = temp
}
```

Swap example:

```
do
{
    key = true
    while (key == true)
        swap (& lock , & key)
    Critical Section
    lock = false
    Remainder Section
}
while (true)
```

Test and Set example:

```
do
{
    while (TestANDSet (&lock))
        // do nothing

    lock = false
}
while (true)
```

Virtual Memory

⇒ Virtual memory is a technique that is used to execute processes that are not in the main memory completely.

⇒ It allows to execute larger processes than main memory. The operating system places the currently required parts of a process in main memory.

⇒ The remaining parts are placed on backing storage. These parts are swapped into main memory as and when required.

⇒ If flow of execution moves to a part that is not in the memory, the operating system loads the required part from secondary storage into the main memory.

Demand Paging:

⇒ Virtual memory can be implemented by a technique called demand paging.

⇒ Demand paging is a technique in which a page is brought into memory when it is actually needed.

Life cycle of a process:

(1) When a process is intiated, the operating system must at least load one page in real memory. It is the page containing the execution part of the process.

(2) Execution of the process commences and proceeds through subsequent instructions beyond the starting point.

(3) This execution continues as long as memory references generated by this page are also within the same page. The virtual address created may reference a page that is not in real memory. This is called a page fault. It generates an interrupt that asks for the referenced page to be loaded. This is called demanding page.

(4) The operating system will try to load the referenced page into a free real memory frame. When this is achieved the execution can continue.

(5) Finally when the process terminates, the operating system releases all the pages belonging to the process. The pages become available to other processes.

Page faults:

⇒ Page faults occur when a process references an address on a page that is not in memory.

⇒ If there are no free frames in memory then a page of some active process from memory must be evicted so that the page that was faulted for can be brought in.

⇒ It is important that the operating system chooses a good page replacement policy.

⇒ If a page is evicted just before it will be needed again, it will also cause a page fault.

Page Replacement:

⇒ Page replacement is another possibility in this situation for eliminating page faults.

⇒ Page replacement writes the contents of the frame to the disk and changes the page table to indicate this page is no longer in memory.

Page Replacement Algorithims:

⇒ The algorithims used to select the page to be replaced are called page replacement algorithims.

(1) First-In First Out (FIFO):

⇒ The first-in first-out policy removes the page that has been resident in memory for the longest time.

(2) Page Optimal Replacement:

⇒ This algorithim produces lowest page fault rate. It replaces the page that will not be used for the longer period of time.

③

Least Recently Used :

⇒ Least Recently Used policy replaces the page whose time since last reference is greater.

④ Not Recently Used (NRU) :

⇒ Not recently used algorithim associates a page reference bit with each page frame.

Refer to book for all the examples of algorithims.

Allocation of frames :

⇒ When the user program starts its execution, it will generate a sequence of page faults.

⇒ The user program would get all free frames from the free frame list. As soon as this list was exhausted and more free frames are required, the page replacement algorithim can be used to select one of the in-used pages to be replaced with the next required page and so on.

⇒ The simplest way is to divide m available frames among n processes to give everyone an equall share m/n shares. This is called equal allocation.

⇒ Another scheme is to give available memory to each process according to its size. This is called propotional allocation.