# Process Synchronization

**1)What is critical Section Problem ? Explain the requirements of the same(IA/SEE).**

Ans)

Critical Section is the part of a program which tries to access shared resources.

The critical section problem is used to design a set of protocols which can ensure that the Race condition among the processes will never arise.

In order to synchronize the cooperative processes, our main task is to solve the critical section problem. We need to provide a solution in such a way that the following conditions can be satisfied.

**Requirements**

- **Mutual Exclusion**
  If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
- **Progress**
  If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will enter in the critical section next, and the selection cannot be postponed indefinitely.
- **Bounded Waiting**
  A bound must exist on the number 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.

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

**2) Explain Peterson's Solution for critical section Problem with algorithm(IA/SEE)**

Ans)

**Peterson's Solution**
Peterson's Solution is a classical software based solution to the critical section problem.

In Peterson's solution, we have two shared variables:

- boolean flag[i] :Initialized to FALSE, initially no one is interested in entering the critical section
- int turn : The process whose turn is to enter the critical section.

```
do {

    flag[i] = TRUE ;
    turn = j ;
    while (flag[j]  &&  turn == j) ;

        critial section

    flag[i] = FALSE ;

        remainder section

} while (TRUE) ;
```

Peterson's Solution preserves all three conditions :

- Mutual Exclusion is assured as only one process can access the critical section at any time.
- Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.
- Bounded Waiting is preserved as every process gets a fair chance.

Disadvantages of Peterson's Solution

- It involves Busy waiting
- It is limited to 2 processes

**3) Explain Semaphore implementation to solve critical section problem(IS/SEE)**

Ans)

Must guarantee that no two processes can execute the wait() and

signal() on the same semaphore at the same time

Thus, the implementation becomes the critical section problem where

the wait and signal code are placed in the critical section

Could now have busy waiting in critical section implementation

But implementation code is short

Little busy waiting if critical section rarely occupied

Note that applications may spend lots of time in critical sections and

therefore this is not a good solution

To overcome the need for busy waiting, we can modify the definition of the wait ()

and signal () semaphore operations. When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait.

However, rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

With each semaphore there is an associated waiting queue

Each entry in a waiting queue has two data items:

- value (of type integer)
- pointer to next record in the list

Two operations:

- block – place the process invoking the operation on the appropriate waiting queue
- wakeup – remove one of processes in the waiting queue and place it in the ready queue

```
typedef struct {
        int value;
        struct process *list;
} semaphore;
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P); } }
```

## 4) Classical Problems of Synchronization:

a) Bounded Buffer Problem(Generally asked in SEE)

b) Reader-Writer Problem(Generally asked in SEE)

c) Dining philosophers Problem with structure(Most Important for IA/SEE)

Ans)

**a) Bounded Buffer Problem(Generally asked in SEE)**

n buffers, each can hold one item

Semaphore mutex initialized to the value 1

Semaphore full initialized to the value 0

Semaphore empty initialized to the value n

The structure of the producer process

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

The structure of the consumer process

```
Do {
    wait(full);
    wait(mutex);
    ...
```

```
        /* remove an item from buffer to next_consumed */
        ...
        signal(mutex);
        signal(empty);
        ...
        /* consume the item in next consumed */
        ...
    } while (true);
```

**b) Reader-Writer Problem(Generally asked in SEE)**

Consider a situation where we have a file shared between many people.

- If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
- However if some person is reading the file, then others may read it at the same time.

Precisely in OS we call this situation as the **readers-writers problem**

Shared Data

Data set

Semaphore rw_mutex initialized to 1

Semaphore mutex initialized to 1

Integer read_count initialized to 0

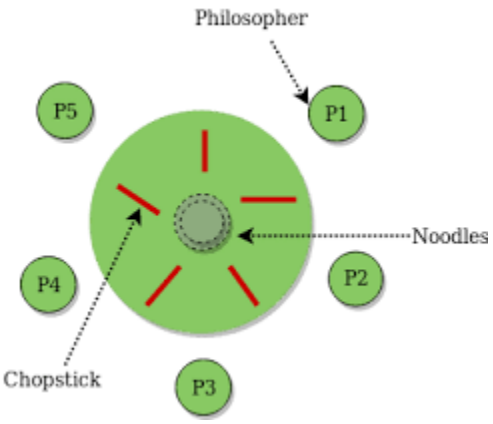
The structure of a writer process

```
do {
    wait(rw_mutex);
        ...
        /* writing is performed */
        ...
    signal(rw_mutex);
} while (true);
```

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

**c) Dining philosophers Problem with structure (Most Important for IA/SEE)**




Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the centre of the table is a bowl of rice, and the table is laid in the case of 5 philosophers with five single chopsticks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbours).

A philosopher may pick up only one chopstick at a time. Obviously, she cannot

pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

One simple solution is to represent each chopstick with a semaphore.

A philosopher tries to grab a chopstick by executing a wait () operation on that semaphore; she releases her chopsticks by executing the signal() operation on the appropriate semaphores.

Thus, the shared data are

semaphore chopstick[5];

where all the elements of chopstick are initialized to 1.

The structure of philosopher / is shown in below.

The structure of Philosopher i:

```
do {
        wait (chopstick[i] );
        wait (chopStick[ (i + 1) % 5] );
        // eat
                signal (chopstick[i] );
                signal (chopstick[ (i + 1) % 5] );
        // think
} while (TRUE);
```

# Deadlocks

## 1)Deadlock Characterisation(Generally asked in SEE)

Ans)

Deadlock can arise if four ( Necessary) conditions hold simultaneously.

- Mutual exclusion: only one process at a time can use a resource
- Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes
- No preemption: a resource can be released only voluntarily by the process

holding it, after that process has completed its task

- Circular wait: there exists a set {P0, P1, …, Pn} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn–1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P0.

2) Resource allocation Graph(Generally asked in SEE)

Ans)

# Main Memory

## 1)Explain Process Of swapping (Compulsory Question for IA/SEE)

Ans)

Swapping is a memory management scheme in which any process can be temporarily swapped from main memory to secondary memory so that the main memory can be made available for other processes. It is used to improve main memory utilization. In secondary memory, the place where the swapped-out process is stored is called swap space.

A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution

Total physical memory space of processes can exceed physical memory

**Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images

**Roll out, roll in** – swapping variant used for priority-based scheduling algorithms;

lower-priority process is swapped out so higher-priority process can be loaded and executed

Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped

System maintains a ready queue of ready-to-run processes which have memory images on disk

Does the swapped out process need to swap back in to same physical addresses?

Depends on address binding method.

Normally, a process that is swapped out will be swapped back into the same memory space it occupied previously.

If binding is done at assembly or load time, then the process cannot be easily moved to a different location.

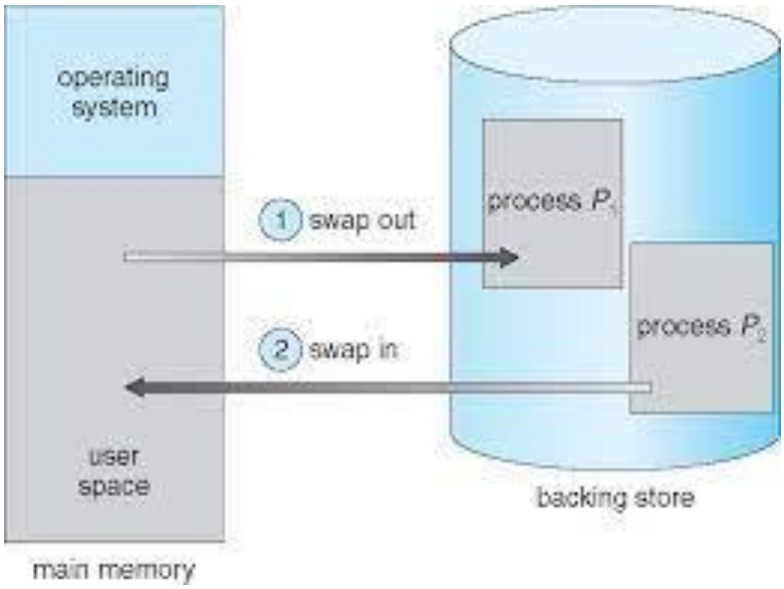
If execution-time binding is being used, however, then a process can be swapped into a different memory space, because the physical addresses are computed during execution time.

Swapping is constrained by other factors as well. If we want to swap a process, we must be sure that it is completely idle. Never swap a process with pending I/O.

The purpose of the swapping in operating system is to access the data present in the hard disk and bring it to RAM so that the application programs can use it. The thing to remember is that swapping is used only when data is not present in RAM.

Although the process of swapping affects the performance of the system, it helps to run larger and more than one process. This is the reason why swapping is also referred to as memory compaction.

- Swap-out is a method of removing a process from RAM and adding it to the hard disk.
- Swap-in is a method of removing a program from a hard disk and putting it back into the main memory or RAM.




### 3) What is Paging? Explain Paging Hardware with TLB with diagram

Ans)

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. This scheme permits the physical address space of a process to be non – contiguous.

- Logical Address or Virtual Address (represented in bits): An address generated by the CPU
- Logical Address Space or Virtual Address Space( represented in words or bytes): The set of all logical addresses generated by a program
- Physical Address (represented in bits): An address actually available on memory unit
- Physical Address Space (represented in words or bytes): The set of all physical addresses corresponding to the logical addresses

# Virtual Memory

## 1)Page Replacement Algorithms(IMP for IA/SEE)

### Types of Page Replacement Algorithms

There are various page replacement algorithms. Each algorithm has a different method by which the pages can be replaced.

1. **Optimal Page Replacement algorithm** → this algorithms replaces the page which will not be referred for so long in future. Although it can not be practically implementable but it can be used as a benchmark. Other algorithms are compared to this in terms of optimality.

*Advantages*

- Easy to Implement.
- Simple data structures are used.
- Highly efficient.

*Disadvantages*

- Requires future knowledge of the program.
- Time-consuming.

2. **Least recent used (LRU) page replacement algorithm** → this algorithm replaces the page which has not been referred for a long time. This algorithm is just opposite to the optimal page replacement algorithm. In this, we look at the past instead of staring at future.

*Advantages*

- Efficient.
- Doesn't suffer from Belady's Anomaly.

*Disadvantages*

- Complex Implementation.

- Expensive.
- Requires hardware support.

3. **FIFO** → in this algorithm, a queue is maintained. The page which is assigned the frame first will be replaced first. In other words, the page which resides at the rare end of the queue will be replaced on the every page fault.

*Advantages*

- Simple and easy to implement.
- Low overhead.

*Disadvantages*

- Poor performance.
- Doesn't consider the frequency of use or last used time, simply replaces the oldest page.
- Suffers from Belady's Anomaly(i.e. more page faults when we increase the number of page frames).

## 3)What is Page Fault?What are steps to handle Page Fault.Explain with diagram

Ans)

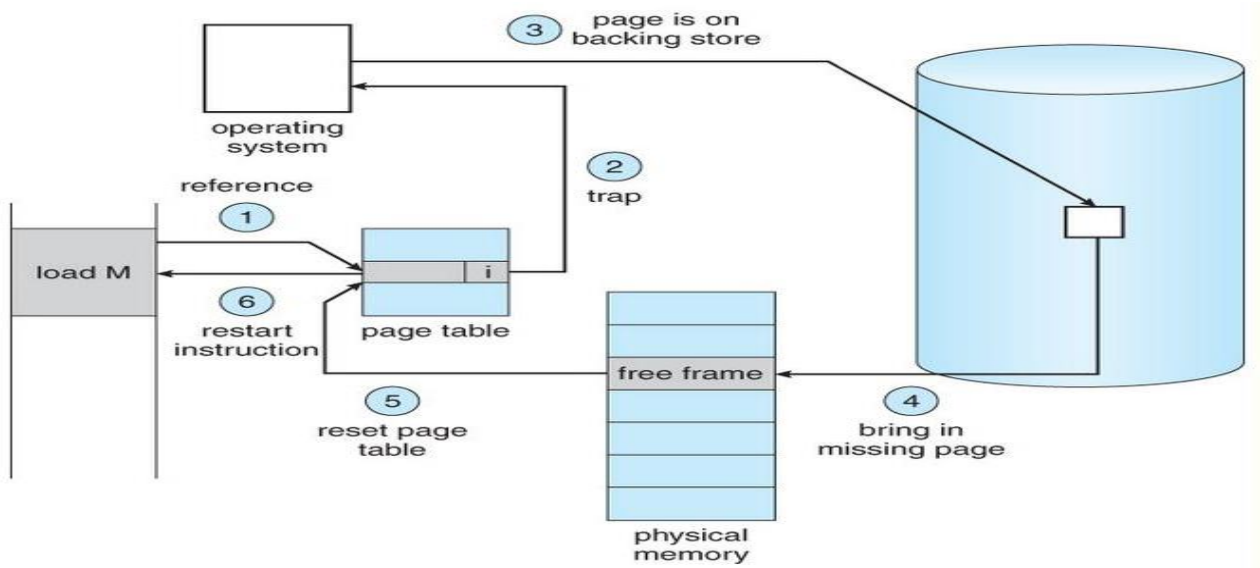
A page fault is a type of exception raised by computer hardware when a running program accesses a memory page that is not currently mapped by the memory management unit (MMU) into the virtual address space of a process.

If there is a reference to a page, first reference to that page will trap to operating system:

This trap is the result of the operating system's failure to bring the desired page into memory.

This leads to ,page fault

STEPS




1. Check the location of the referenced page in the PMT
2. If a page fault occurred, call on the operating system to fix it
3. Using the frame replacement algorithm, find the frame location
4. Read the data from disk to memory
5. Update the page map table for the process
6. The instruction that caused the page fault is restarted when the process resumes execution.