Multiple-Processor Scheduling

Introduction

- In multiple-processor scheduling multiple CPU's are available and hence Load Sharing becomes possible.
- It is more complex as compared to single processor scheduling.

Approaches to Multiple-Processor Scheduling –

- One approach is when all the scheduling decisions and I/O processing are handled by a single processor which is called the **Master Server** and the other processors executes only the **user code**.
- This is simple and reduces the need of data sharing. This entire scenario is called **Asymmetric Multiprocessing**.

- A second approach uses **Symmetric Multiprocessing** where each processor is **self scheduling**.
- All processes may be in a common ready queue, or each processor may have its own private queue for ready processes.
- The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute.

Processor Affinity –

- Processor Affinity is a concept that allows an operating system to control the way applications use a computer's processor resources.
- Knowing how to use this feature, can help you maximize your processor's efficiency and reduce the amount of time it takes for applications to complete tasks.
- Processor affinity can effectively decrease cache issues.

- Affinity enables assigning processes and threads to a specific CPU or subset of CPUs.
- It can also improve application performance by reducing resource contention.
- It is possible to dedicate CPU resources to particular tasks using processor affinity while running other tasks on separate CPUs.

- There are two types of processor affinity: **(a) Hard Affinity (b) soft Affinity**
- **Hard Affinity –** In this type, the system enforces the affinity and ensures that the process or thread always run on the designated subset of processors.
- This can provide better performance predictability for real time applications that require guaranteed CPU resources.

- **Soft Affinity –** In this type, the system only tries to assign the process or thread to the designated processor or subset of processors.
- It also allows it to run on other available CPUs if the assigned CPU is busy or unavailable.
- Soft affinity is helpful for applications that do not require strict CPU resource guarantees and may benefit from load balancing across multiple CPUs.

Load Balancing –

- Load Balancing is the **phenomena** which keeps the **workload** evenly **distributed** across all processors in an SMP system.
- Load balancing is necessary only on systems where each processor has its own private queue of process which are eligible to execute.
- Load balancing is unnecessary because once a processor becomes idle it immediately extracts a runnable process from the common run queue.

- On SMP(symmetric multiprocessing), it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor else one or more processor will sit idle while other processors have high workloads along with lists of processors awaiting the CPU.

- There are two general approaches to load balancing :
- **Push Migration –** In push migration a task routinely checks the load on each processor and if it finds an imbalance then it evenly distributes load on each processors by moving the processes from overloaded to idle or less busy processors.
- **Pull Migration –** Pull Migration occurs when an idle processor pulls a waiting task from a busy processor for its execution.

Process Coordination

Introduction

- Process coordination comprises mechanisms for processes carrying out cooperative work, usually by means of communication (pipes, sockets, shared memory, etc).
- Process coordination is necessary in a concurrent system
- It avoids conflicts when accessing shared items

- It allows processes to cooperate
- It Can be used when
  - Process waits for I/O
  - Process waits for another process
- Example of cooperation among processes: UNIX pipes

Process Synchronization

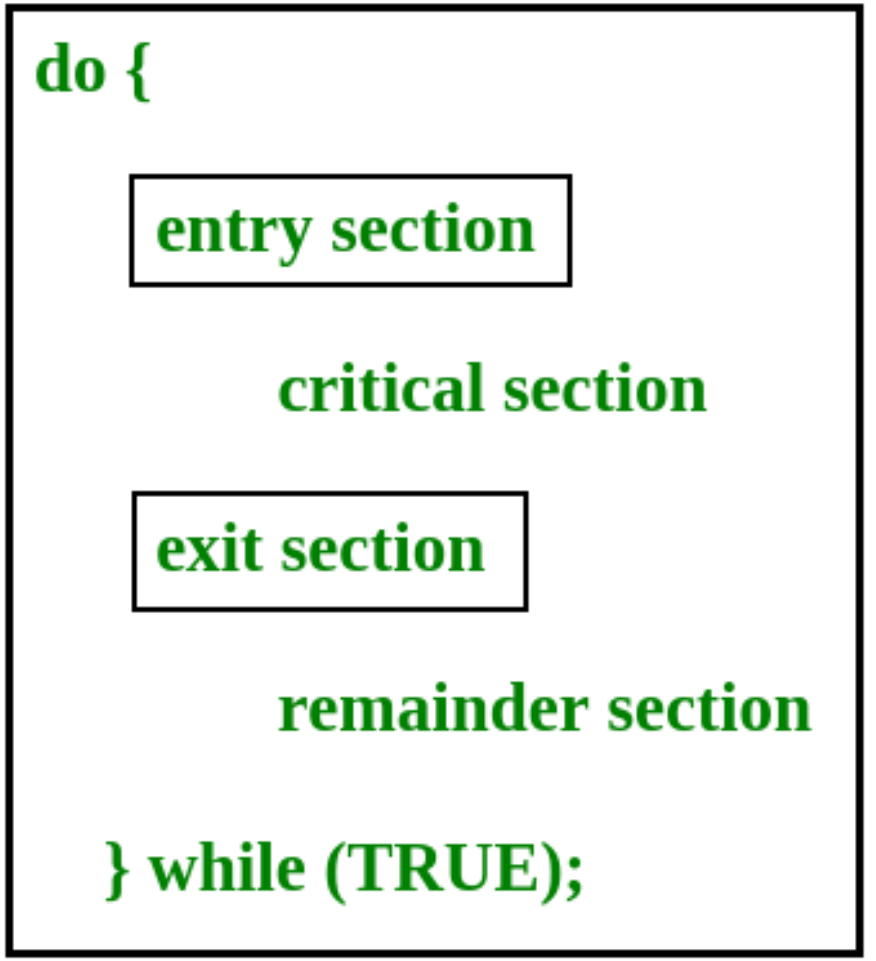
- **Process Synchronization** is the task of coordinating the execution of processes in a way that no two processes can have access to the same shared data and resources.
- It is specially needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or data at the same time.

- This can lead to the inconsistency of shared data.
- So the change made by one process not necessarily reflected when other processes accessed the same shared data.
- To avoid this type of inconsistency of data, the processes need to be synchronized with each other.

- On the basis of synchronization, processes are categorized as one of the following two types:
- **Independent Process** : Execution of one process does not affects the execution of other processes.
- **Cooperative Process** : Execution of one process affects the execution of other processes.

Critical Section Problem

- Critical section is a code segment that can be accessed by only one process at a time.
- Critical section contains shared variables which need to be synchronized to maintain consistency of data variables.



- In the entry section, the process requests for entry in the **Critical Section.**
- Any solution to the critical section problem must satisfy three 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 can not 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.

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.

Synchronization Hardware

- Many systems provide hardware support for critical section code.
- The critical section problem could be solved easily in a single-processor environment if we could disallow interrupts to occur while a shared variable or resource is being modified.
- In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption.
- Unfortunately, this solution is not feasible in a multiprocessor environment.

- Disabling interrupt on a multiprocessor environment can be time-consuming as the message is passed to all the processors.
- This message transmission lag delays the entry of threads into the critical section, and the system efficiency decreases.

- **Mutex Locks**
- As the synchronization hardware solution is not easy to implement for everyone, a strict software approach called Mutex Locks was introduced. In this approach, in the entry section of code, a LOCK is acquired over the critical resources modified and used inside the critical section, and in the exit section that LOCK is released.
- As the resource is locked while a process executes its critical section hence no other process can access it.

Semaphores

- A semaphore is a signaling mechanism and a thread that is waiting on a semaphore can be signaled by another thread. This is different than a mutex as the mutex can be signaled only by the thread that called the wait function.
- A semaphore uses two atomic operations, wait and signal for process synchronization.

  A Semaphore is an integer variable, which can be accessed only through two operations *wait()* and *signal()*.

  There are two types of semaphores: **Binary Semaphores** and **Counting Semaphores**

- Binary Semaphores: They can only be either 0 or 1.
- They are also known as mutex locks, as the locks can provide mutual exclusion.
- All the processes can share the same mutex semaphore that is initialized to 1.

- Then, a process has to wait until the lock becomes 0.
- Then, the process can make the mutex semaphore 1 and start its critical section.
- When it completes its critical section, it can reset the value of mutex semaphore to 0 and some other process can enter its critical section.

- Counting Semaphores: They can have any value and are not restricted over a certain domain.
- They can be used to control access to a resource that has a limitation on the number of simultaneous accesses.
- The semaphore can be initialized to the number of instances of the resource.

- Whenever a process wants to use that resource, it checks if the number of remaining instances is more than zero, i.e., the process has an instance available.
- Then, the process can enter its critical section thereby decreasing the value of the counting semaphore by 1.
- After the process is over with the use of the instance of the resource, it can leave the critical section thereby adding 1 to the number of available instances of the resource.

Classic Problems of Synchronization

- We will discuss the following three problems:
- Bounded Buffer (Producer-Consumer) Problem
- Dining Philosophers Problem
- The Readers Writers Problem

- **Bounded Buffer Problem**
- Because the buffer pool has a maximum size, this problem is often called the **Bounded buffer problem**.
- This problem is generalised in terms of the **Producer Consumer problem**, where a **finite** buffer pool is used to exchange messages between producer and consumer processes.

- Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively.
- In this Producers mainly produces a product and consumers consume the product, but both can use of one of the containers each time.
- The main complexity of this problem is that we must have to maintain the count for both empty and full containers that are available.

- **Dining Philosophers Problem**
- The dining philosopher's problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.
- There are five philosophers sitting around a table, in which there are five chopsticks/forks kept beside them and a bowl of rice in the centre, When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right.
- When a philosopher wants to think, he keeps down both chopsticks at their original place.

- **The Readers Writers Problem**
- In this problem there are some processes(called **readers**) that only read the shared data, and never change it, and there are other processes(called **writers**) who may change the data in addition to reading, or instead of reading it.
- There are various type of readers-writers problem, most centred on relative priorities of readers and writers.
- The main complexity with this problem occurs from allowing more than one reader to access the data at the same time.


Thank You