Comparison between various CPU Scheduling algorithms

Here is a brief comparison between different CPU scheduling algorithms:

Algorithm	Allocation	Complexity	Average waiting time (AWT)	Preemption	Starvation	Performance
FCFS	According to the arrival time of the processes, the CPU is allocated.	Simple and easy to implement	Large.	No	No	Slow performance
SJF	Based on the lowest CPU burst time (BT).	More complex than FCFS	Smaller than FCFS	No	Yes	Minimum Average Waiting Time
RR	According to the order of the process arrives with fixed time quantum (TQ)	The complexity depends on Time Quantum size	Large as compared to SJF and Priority scheduling.	Yes	No	Each process has given a fairly fixed time

Algorithm	Allocation	Complexity	Average waiting time (AWT)	Preemption	Starvation	Performance
Pre-emptive	According to the priority. The bigger priority task executes first	This type is less complex	Smaller than FCFS	Yes	Yes	Well performance but contain a starvation problem

Multiple-Processor Scheduling in Operating System

In multiple-processor scheduling **multiple CPU's** are available and hence **Load Sharing** becomes possible. However multiple processor scheduling is more **complex** as compared to single processor scheduling. In multiple processor scheduling there are cases when the processors are identical i.e. HOMOGENEOUS, in terms of their functionality, we can use any processor available to run any process in the queue.

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 means a processes has an **affinity** for the processor on which it is currently running. When a process runs on a specific processor there are certain effects on the cache memory. The data most recently accessed by the process populate the cache for the processor and as a result successive memory access by the process are often satisfied in the cache memory. Now if the process migrates to another processor, the contents of the cache memory must be invalidated for the first processor and the cache for the second processor must be repopulated. Because of the high cost of invalidating and repopulating caches, most of the SMP(symmetric multiprocessing) systems try to avoid migration of processes from one processor to another and try to keep a process running on the same processor. This is known as **PROCESSOR AFFINITY**.
There are two types of processor affinity:

1. **Soft Affinity –** When an operating system has a policy of attempting to keep a process running on the same processor but not guaranteeing it will do so, this situation is called soft affinity.
2. **Hard Affinity –** Hard Affinity allows a process to specify a subset of processors on which it may run. Some systems such as Linux implements soft affinity but also provide some system calls like *sched_setaffinity()* that supports hard affinity.

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 :

1. **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.
2. **Pull Migration –** Pull Migration occurs when an idle processor pulls a waiting task from a busy processor for its execution.

Virtualization and Threading –

In this type of **multiple-processor** scheduling even a single CPU system acts like a multiple-processor system. In a system with Virtualization, the virtualization presents one or more virtual CPU to each of virtual machines running on the system and then schedules the use of physical CPU among the virtual machines. Most virtualized environments have one host operating system and many guest operating systems. The host operating system creates and manages the virtual machines. Each virtual machine has a guest operating system installed and applications run within that guest.Each guest operating system may be assigned for specific use cases,applications or users including time sharing or even real-time operation. Any guest operating-system scheduling algorithm that assumes a certain amount of progress in a given amount of time will be negatively impacted by the virtualization. A time sharing operating system tries to allot 100 milliseconds to each time slice to give users a reasonable response time. A given 100 millisecond time slice may take much more than 100 milliseconds of virtual CPU time. Depending on how busy the system is, the time slice may take a second or more which results in a very poor response time for users logged into that virtual machine. The net effect of such scheduling layering is that individual virtualized operating systems receive only a portion of the available CPU cycles, even though they believe they are receiving all cycles and that they are scheduling all of those cycles.Commonly, the time-of-day clocks in virtual machines are incorrect because timers take no longer to trigger than they would on dedicated CPU's.
Virtualizations can thus undo the good scheduling-algorithm efforts of the operating systems within virtual machines.

Scheduling in Real Time Systems

Real-time systems are systems that carry real-time tasks. These tasks need to be performed immediately with a certain degree of urgency. In particular, these tasks are related to control of certain events (or) reacting to them. Real-time tasks can be classified as hard real-time tasks and soft real-time tasks.
A hard real-time task must be performed at a specified time which could otherwise lead to huge losses. In soft real-time tasks, a specified deadline can be missed. This is because the task can be rescheduled (or) can be completed after the specified time,

In real-time systems, the scheduler is considered as the most important component which is typically a short-term task scheduler. The main focus of this scheduler is to reduce the response time associated with each of the associated processes instead of handling the deadline.

If a preemptive scheduler is used, the real-time task needs to wait until its corresponding tasks time slice completes. In the case of a non-preemptive scheduler, even if the highest priority is allocated to the task, it needs to wait until the completion of the current task. This task can be slow (or) of the lower priority and can lead to a longer wait.

A better approach is designed by combining both preemptive and non-preemptive scheduling. This can be done by introducing time-based interrupts in priority based systems which means the currently running process is interrupted on a time-based interval and if a higher priority process is present in a ready queue, it is executed by preempting the current process.

Process co-ordination

Process co-ordination deals with mutual exclusion and process synchronization.When a process is executed in a critical section, then no other process can be executed in their critical section. It consists of a single process temporarily excluding all others from using a shared resource in a way that guarantees the integrity of the system. It assures that when one thread is functioning

Mutual Exclusion

There are four conditions applied on mutual exclusion. These are the following:

- Mutual exclusion must be guaranteed between the different processes when accessing the shared resource. There cannot be two processes within their respective critical sections at any time.
- No assumptions should be made as to the relative speed of the conflicting processes.
- No process that is outside its critical section should interrupt another for access to the critical section.
- When more than one process wants to enter its critical section, it must be granted entry in a finite time, that is, it will never be kept waiting in a loop that has no end.

Process Synchronization

Process Synchronization means coordinating the execution of processes such that no two processes access the same shared resources and data. It is required in a multi-process system where multiple processes run together, and more than one process tries to gain access to the same shared resource or data at the same time.

Changes made in one process aren't reflected when another process accesses the same shared data. It is necessary that processes are synchronized with each other as it helps avoid the inconsistency of shared data.

For example: A process P1 tries changing data in a particular memory location. At the same time another process P2 tries reading data from the same memory location. Thus, there is a high probability that the data being read by the second process is incorrect.

Sections of a Program in OS

Following are the four essential sections of a program:

1. Entry Section: This decides the entry of any process.
2. Critical Section: This allows a process to enter and modify the shared variable.
3. Exit Section: This allows the process waiting in the Entry Section, to enter into the Critical Sections and makes sure that the process is removed through this section once it's done executing.
4. Remainder Section: Parts of the Code, not present in the above three sections are collectively called Remainder Section.

Semaphores in Operating System

Types of process in Operating System

On the basis of synchronization, the following are the two types of processes:

1. Independent Processes: The execution of one process doesn't affect the execution of another.
2. Cooperative Processes: Execution of one process affects the execution of the other. Thus, it is necessary that these processes are synchronized in order to guarantee the order of execution.

Critical Section Problem in OS

A segment of code that a signal process can access at a particular point of time is known as the critical section. It contains the shared data resources that can be accessed by other processes.

Wait() function (represented as P()) handles the entry of a process to the critical section. Whereas signal() function (represented as V()) handles the exit of a process from the critical section. A single process executes in a critical section at a time. Other processes execute after the current process is done executing.

Race Condition in OS

When more than one processes execute the same code or access the same memory/shared variable, it is possible that the output or value of the shared variable is wrong. In this condition, all processes race ahead in order to prove that their output is correct. This situation is known as race condition.

When multiple processes access and manipulate the same data concurrently the outcome depends on the order in which these processes accessed the shared data. When the output of multiple thread execution differs according to the order in which the threads execute, a race condition occurs.

We can avoid it if we treat the critical section as an atomic instruction and maintain proper thread synchronization using locks or atomic variables.

Rules for Critical Section

There are three rules that need to be enforced in the critical section. They are:

1. Mutual Exclusion: A special type of binary semaphore used to control access to shared resources. It has a priority inheritance mechanism that helps avoid extended priority inversion problems. Only one process can execute at a time in the critical section.

2. Progress: We use it when the critical section is empty, and a process wants to enter it. The processes that are not present in their reminder section decide who should go in, within a finite time.

3. Bound Waiting: Only a specific number of processes are allowed into their critical section. Thus, a process needs to make a request when it wants to enter the critical section and when the critical section reaches its limit, the system allows the process' request and allows it into its critical section.

Solutions To The Critical Section

Following are some common solutions to the critical section problem:

1. Peterson Solution: If a process executes in a critical state, the other process can only execute the rest of the code and vise-versa. This method developed by Peterson is widely used and helps make sure that only one process runs at a specific time in the critical section

Example: Let there be N processes (P1, P2, … PN) such that at some point of time every process needs to enter the Critical Section. There is a FLAG[] array of size N that is false by default. Therefore, the flag of a process needs to be set true, whenever it wants to enter the critical section.

A variable TURN tells the process number waiting to enter the Critical Section. When a process enters the critical section it changes the TURN to another number from the list of ready processes when it is exiting.

Program:

```
PROCESS Pi
FLAG[i] = true
while( (turn != i) AND (CS is !free) ){ wait;
```

```
}
CRITICAL SECTION FLAG[i] = false
turn = j; //choose another process
```

2. Synchronization Hardware: Hardware can also help resolve the problems of critical sections sometimes. Some OS offer lock functionality. This gives a process a lock when it enters the critical section and releases the lock after it leaves a critical section. Due to this other processes can't enter a critical section when a process is already inside.

3. Mutex Locks: Mutex Locks is a strict software method in which a LOCK over critical resources is given to a process in the entry section of code. The process can use this LOCK inside the critical section and can get rid of it in the exit section.

4. Semaphore Solution: Semaphore is a non-negative variable that is shared between threads. It is a signaling mechanism that uses a thread waiting on a semaphore, to signal another thread. It makes use of wait and signal for process synchronization.

Example:

```
WAIT ( S );
while ( S <= 0 );
S = S - 1;
SIGNAL ( S ):
S = S + 1;
```

Semaphore

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

The definitions of wait and signal are as follows −

- **Wait**
  The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

```
wait(S)
```

```
{
  while (S<=0);

  S--;
}
```

- **Signal**
  The signal operation increments the value of its argument S.

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

Types of Semaphores

There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows −

- **Counting Semaphores**
  These are integer value semaphores 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 the resources are added, semaphore count automatically incremented and if the resources are removed, the 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 succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.

Advantages of Semaphores

Some of the advantages of semaphores are as follows −

- Semaphores allow only one process into the 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 semaphores 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 the microkernel. So they are machine independent.

Disadvantages of Semaphores

Some of the disadvantages of semaphores are as follows −

- Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
- Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
- Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.