DEPARTMENT OF AIML

23CST202- OPERATING SYSTEMS

II YEAR IV SEM AIML-B

UNIT 2-PROCESS SCHEDULING AND SYNCHRONIZATION

TOPIC –CRITICAL SECTION PROBLEM AND SYNCHRONIZATION HARDWARE

CRITICAL SECTION PROBLEM

A **critical section** is a part of a program where shared resources like memory or files are accessed by multiple processes or threads. To avoid issues like data inconsistency or race conditions, synchronization techniques ensure that only one process or thread uses the critical section at a time.

- The critical section contains shared variables or resources that need to be synchronized to maintain the consistency of data variables.
- In simple terms, a critical section is a group of instructions/statements or regions of code that need to be executed atomically, such as accessing a resource (file, input or output port, global data, etc.) In concurrent programming, if one process tries to change the value of shared data at the same time as another thread tries to read the value (i.e., data race across threads), the result is unpredictable. The access to such shared variables (shared memory, shared files, shared port, etc.) is to be synchronized.

Few programming languages have built-in support for synchronization. It is critical to understand the importance of race conditions while writing kernel-mode programming (a device driver, kernel thread, etc.) since the programmer can directly access and modify kernel data structures

Although there are some **properties that should be followed if any code in the critical section**

1. **Mutual Exclusion:** If process Pi 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 those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded Waiting:** There exists a bound, or limit, 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.

Two general approaches are used to handle critical sections:

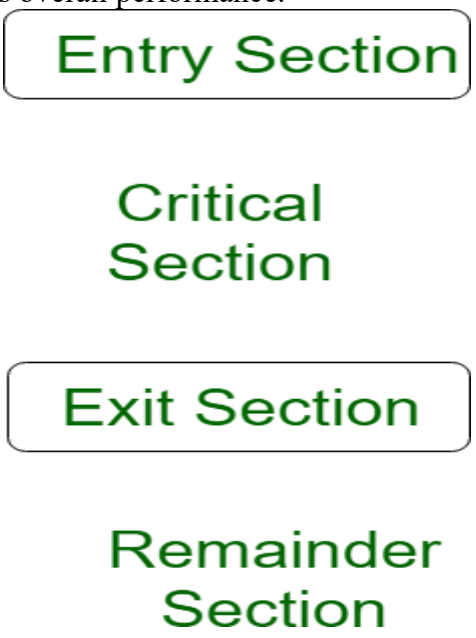
1. **Preemptive kernels**: A preemptive kernel allows a process to be preempted while it is running in kernel mode.
2. **Non preemptive kernels**: A non-preemptive kernel does not allow a process running in kernel mode to be preempted. A kernel-mode process will run until it exists in kernel mode, blocks, or voluntarily yields control of the CPU. A non-

preemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time.

Critical Section Problem

The use of critical sections in a program can cause a number of issues, including:

- **Deadlock:** When two or more threads or processes wait for each other to release a critical section, it can result in a deadlock situation in which none of the threads or processes can move. Deadlocks can be difficult to detect and resolve, and they can have a significant impact on a program's performance and reliability.
- **Starvation:** When a thread or process is repeatedly prevented from entering a critical section, it can result in starvation, in which the thread or process is unable to progress. This can happen if the critical section is held for an unusually long period of time, or if a high-priority thread or process is always given priority when entering the critical section.
- **Overhead:** When using critical sections, threads or processes must acquire and release locks or semaphores, which can take time and resources. This may reduce the program's overall performance.



It could be visualized using the pseudo-code below –

```
do{
   flag=1;
   while(flag); // (entry section)
      // critical section
   if (!flag)
      // remainder section
} while(true);
```

Solution to Critical Section Problem

A simple solution to the critical section can be thought of as shown below,

```
acquireLock();
Process Critical Section
releaseLock();
```

A thread must acquire a lock prior to executing a critical section. The lock can be acquired by only one thread. There are various ways to implement locks in the above pseudo-code. Let us discuss them in future articles.

To read about detailed solution to Critical Section Problem read – Solution to Critical Section Problem

Strategies for avoiding problems: While deadlocks, starvation, and overhead are mentioned as potential issues, there are more specific strategies for avoiding or mitigating these problems. For example, using timeouts to prevent deadlocks, implementing priority inheritance to prevent priority inversion and starvation, or optimizing lock implementation to reduce overhead.

Examples of critical sections in real-world applications: While the article describes critical sections in a general sense, it could be useful to provide examples of how critical sections are used in specific real-world applications, such as database management systems or web servers.

Impact on scalability: The use of critical sections can impact the scalability of a program, particularly in distributed systems where multiple nodes are accessing shared resources.

In process synchronization, a critical section is a section of code that accesses shared resources such as variables or data structures, and which must be executed by only one process at a time to avoid race conditions and other synchronization-related issues.

A critical section can be any section of code where shared resources are accessed, and it typically consists of two parts: the entry section and the exit section. The entry section is where a process requests access to the critical section, and the exit section is where it releases the resources and exits the critical section.

To ensure that only one process can execute the critical section at a time, process synchronization mechanisms such as semaphores and mutexes are used. A semaphore is a variable that is used to indicate whether a resource is available or not, while a mutex is a semaphore that provides mutual exclusion to shared resources.

When a process enters a critical section, it must first request access to the semaphore or mutex associated with the critical section. If the resource is available, the process can proceed to execute the critical section. If the resource is not available, the process must wait until it is released by the process currently executing the critical section.

Once the process has finished executing the critical section, it releases the semaphore or mutex, allowing another process to enter the critical section if necessary.

Proper use of critical sections and process synchronization mechanisms is essential in concurrent programming to ensure proper synchronization of shared resources and avoid race conditions, deadlocks, and other synchronization-related issues.

Advantages of Critical Section in Process Synchronization

1. **Prevents race conditions:** By ensuring that only one process can execute the critical section at a time, race conditions are prevented, ensuring consistency of shared data.
2. **Provides mutual exclusion:** Critical sections provide mutual exclusion to shared resources, preventing multiple processes from accessing the same resource simultaneously and causing synchronization-related issues.
3. **Reduces CPU utilization:** By allowing processes to wait without wasting CPU cycles, critical sections can reduce CPU utilization, improving overall system efficiency.
4. **Simplifies synchronization:** Critical sections simplify the synchronization of shared resources, as only one process can access the resource at a time, eliminating the need for more complex synchronization mechanisms.

Disadvantages of Critical Section in Process Synchronization

1. **Overhead:** Implementing critical sections using synchronization mechanisms like semaphores and mutexes can introduce additional overhead, slowing down program execution.
2. **Deadlocks:** Poorly implemented critical sections can lead to deadlocks, where multiple processes are waiting indefinitely for each other to release resources.
3. **Can limit parallelism:** If critical sections are too large or are executed frequently, they can limit the degree of parallelism in a program, reducing its overall performance.
4. **Can cause contention:** If multiple processes frequently access the same critical section, contention for the critical section can occur, reducing performance.

Important Points Related to Critical Section in Process Synchronization

1. Understanding the concept of critical section and why it's important for synchronization.
2. Familiarity with the different synchronization mechanisms used to implement critical sections, such as semaphores, mutexes, and monitors.
3. Knowledge of common synchronization problems that can arise in critical sections, such as race conditions, deadlocks, and live locks.
4. Understanding how to design and implement critical sections to ensure proper synchronization of shared resources and prevent synchronization-related issues.
5. Familiarity with best practices for using critical sections in concurrent programming.

SYNCHRONIZATION HARDWARE

Hardware Synchronization Algorithms : Unlock and Lock, Test and Set, Swap

Process Synchronization problems occur when two processes running concurrently share the same data or same variable. The value of that variable may not be updated correctly before its being used by a second process. Such a condition is known as Race Around Condition. There are a software as well as hardware solutions to this problem. In this article, we will talk about the most efficient hardware solution to process synchronization problems and its implementation.

There are three algorithms in the hardware approach of solving Process Synchronization problem:

1. Test and Set
2. Swap
3. Unlock and Lock

Hardware instructions in many operating systems help in the effective solution of critical section problems.

1. Test and Set:

Here, the shared variable is lock which is initialized to false. TestAndSet(lock) algorithm works in this way – it always returns whatever value is sent to it and sets lock to true. The first process will enter the critical section at once as TestAndSet(lock) will return false and it'll break out of the while loop. The other processes cannot enter now as lock is set to true and so the while loop continues to be true. Mutual exclusion is ensured. Once the first process gets out of the critical section, lock is changed to false. So, now the other processes can enter one by one. Progress is also ensured. However, after the first process, any process can go in. There is no queue maintained, so any new process that finds the lock to be false again can enter. So bounded waiting is not ensured.

Test and Set Pseudocode –

```
//Shared variable lock initialized to false
boolean lock;

boolean TestAndSet (boolean &target){
    boolean rv = target;
    target = true;
    return rv;
}

while(1){
    while (TestAndSet(lock));
 critical section
     lock = false;
remainder section
}
```

2. Swap:

Swap algorithm is a lot like the TestAndSet algorithm. Instead of directly setting lock to true in the swap function, key is set to true and then swapped with lock. First process will be executed, and in while(key), since key=true , swap will take place and hence lock=true and key=false. Again next iteration takes place while(key) but key=false , so while loop breaks and first process will enter in critical section. Now another process will try to enter in Critical section, so again key=true and hence while(key) loop will run and swap takes place so, lock=true and key=true (since lock=true in first process). Again on next iteration while(key) is true so this will keep on executing and another process will not be able to enter in critical section.

Therefore Mutual exclusion is ensured. Again, out of the critical section, lock is changed to false, so any process finding it gets t enter the critical section. Progress is ensured. However, again bounded waiting is not ensured for the very same reason.

Swap Pseudocode –

```
// Shared variable lock initialized to false
// and individual key initialized to false;

boolean lock;
Individual key;

void swap(boolean &a, boolean &b){
    boolean temp = a;
    a = b;
    b = temp;
}

while (1){
    key = true;
    while(key)
        swap(lock,key);
 critical section
    lock = false;
remainder section
}
```

3. Unlock and Lock :

Unlock and Lock Algorithm uses TestAndSet to regulate the value of lock but it adds another value, waiting[i], for each process which checks whether or not a process has been waiting. A ready queue is maintained with respect to the process in the critical section. All the processes coming in next are added to the ready queue with respect to their process number, not necessarily sequentially. Once the ith process gets out of the critical section, it does not turn lock to false so that any process can avail the critical section now, which was the problem with the previous algorithms. Instead, it checks if there is any process waiting in the queue. The queue is taken to be a circular queue. j is considered to be the next process in line and the while loop checks from jth process to the last process and again from 0 to (i-1)th process if there is any process waiting to access the critical section. If there is no process waiting then the lock value is changed to false and any process which comes next can enter the critical section. If there is, then that process' waiting value is turned to false, so that the first while loop becomes false and it can enter the critical section. This ensures bounded waiting. So the problem of process synchronization can be solved through this algorithm.

Unlock and Lock Pseudocode –

```
// Shared variable lock initialized to false
// and individual key initialized to false

boolean lock;
```

```
Individual key;
Individual waiting[i];

while(1){
    waiting[i] = true;
    key = true;
    while(waiting[i] && key)
        key = TestAndSet(lock);
    waiting[i] = false;
 critical section
    j = (i+1) % n;
    while(j != i && !waiting[j])
        j = (j+1) % n;
    if(j == i)
        lock = false;
    else
        waiting[j] = false;
remainder section
}
```