OS5 - Synchronisation Issues and Solutions - Mutex and Semaphores

Tuesday, 11 July 2023 1:53 AM

SYNCHRONISATION

In operating systems, synchronization refers to the coordination and control of concurrent processes or threads to ensure orderly and consistent execution. It involves managing access to shared resources or critical sections to prevent conflicts, race conditions, and data inconsistencies.

Scenarios where there is a problem of Sync. issue

1) Critical Section

```
add() {
    ______
    ______
    count += 1
    ______
}
```

count += 1 → Critical section of the code → deals with data that is shared among multiple threads

2) Race Condition

A scenario where multiple threads compete to get to the Critical Section.

Race conditions typically occur when multiple threads or processes are performing read and write operations on shared data concurrently. If the operations are not properly synchronized or coordinated, conflicts can arise. For example, if two threads attempt to update a shared variable simultaneously without proper synchronization, the final value of the variable may not be as expected because the threads might overwrite each other's changes.

3) Preemption

Synchronisation issues only arise when we have a Critical Section in our code and Race Conditions occur, and also Preemption of the code execution occurs.

Solutions to Synchronization Issues

1. Mutual Exclusion

Two threads should not be allowed to access the Critical Section at the same time.

2. Progress

Your process should always progress.

3. Bounded Wait

No thread should wait for a long indefinite amount of time.

4. No Busy Waiting

In busy waiting, a program or thread keeps checking a condition repeatedly in a tight loop, consuming CPU cycles while waiting for the desired condition to become true. This approach is often used when the waiting time is expected to be very short, and it is more efficient to actively wait rather than block the process or thread.

However, busy waiting can be inefficient and wasteful in scenarios where the waiting time is longer. It consumes CPU resources even when no progress is being made, leading to unnecessary CPU utilization and potentially impacting the performance of other tasks or threads running on the system.

How to deal with Synchronisation Issues

1 Mutex

$\boxed{\text{Mut}}$ual $\boxed{\text{Ex}}$clusion

↳ Locking mechanism

When a Thread executes a Critical Section, it first locks it up for itself, and then executes it until reaching the very end of it. Once it has executed the whole CS, it releases the lock for other threads to access it.

★ Using boolean flag

boolean lock = false

```
1  if lock == false {
```

Let's say before line 4, the thread T1 got preempted. Now

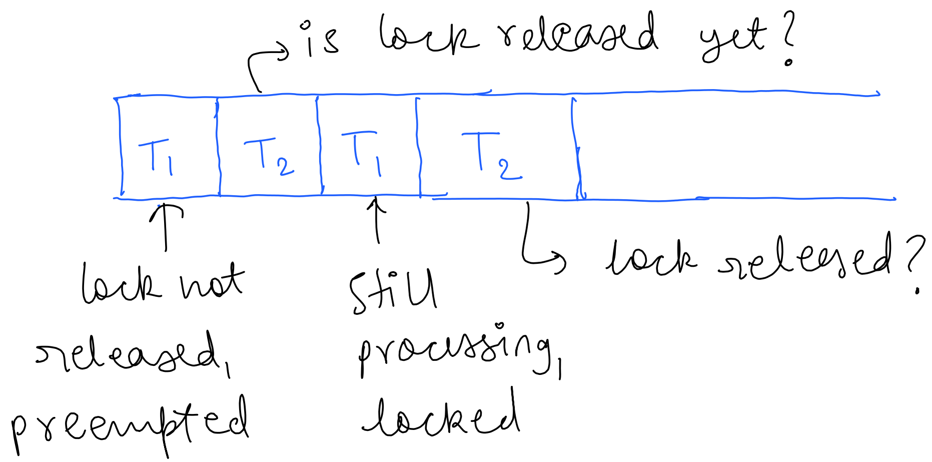
```
lock = true
count += 1
lock = false
}
```

a new thread T2 is executing, and since lock is still = true, it will not be able to execute the code block.

So if both the threads T1 and T2 were to increment count each to make it 2 ideally, it would end up to be 1 because T2 did not even execute the code block.

```
while lock == true {
    continue
}
lock = true
count += 1
lock = false
```

In this case, the thread T2 will be checking again and again if the lock is released by the other thread or not. This is the Busy waiting condition and it will waste unnecessary CPU cycles.



Also check https://github.com/angrave/SystemProgramming/wiki

✯ How to add lock

```java
package addersubtractor;

public class Adder implements
Runnable {
    private Count count;
    Lock lock;

    // Lock lock = new ReentrantLock()

    public Adder(Count count, Lock lock)
{
        this.count = count;
      this.lock = lock;
    }

    @Override
    public void run() {
      lock.lock()
        for(int i = 1; i <= 10000; ++i) {
            count.value += i;
        }
        lock.unlock()
    }
}
```

This is not multithreaded bcos the loop completes all at once

```java
@Override
public void run() {

    for(int i = 1; i <= 10000; ++i) {
        lock.lock()
        count.value += i;
        lock.unlock()
    }
}
```

This is multithreaded & well synchronised code

Or we can avoid the lock variable mechanism by using synchronised ()

```java
@Override
public void run() {

    for(int i = 1; i <= 10000; ++i) {
      synchronized(sharedCount) {
          sharedCount.count += i;
```

Every object has it's own implicit / internal lock. only works with one shared

```
        }
      }
    }
```

variable.

this is an object

SEMAPHORE

Semaphores are helpful when we want only N number of threads to access the Critical Section and lock it for any more threads.

A semaphore is essentially a counter that is used to control access to a shared resource. It can have an initial value greater than or equal to zero. Threads or processes can acquire or release the semaphore, and the value of the semaphore is adjusted accordingly.

There are two types of semaphores: binary semaphores and counting semaphores.

1. Binary Semaphore: A binary semaphore is a semaphore with an initial value of either 0 or 1. It is commonly used to represent a mutex (mutual exclusion) or a lock. Only one thread or process can acquire the binary semaphore at a time. If the semaphore is already acquired, subsequent attempts to acquire it will be blocked until it is released.
2. Counting Semaphore: A counting semaphore is a semaphore with an initial value greater than 1. It allows multiple threads or processes to acquire the semaphore concurrently up to the specified limit. Each acquisition decreases the semaphore value, and each release increases it. If the semaphore value reaches zero, subsequent attempts to acquire it will be blocked until it is released by another thread or process.

```java
import java.util.concurrent.Semaphore;

public class SemaphoreExample {
    public static void main(String[] args) {
        Semaphore semaphore = new Semaphore(3); // Initialize semaphore with 3 permits

        // Create and start 5 threads
        for (int i = 1; i <= 5; i++) {
            Thread thread = new Thread(new Worker(semaphore, i));
            thread.start();
        }
    }
```

```java
    }

    static class Worker implements Runnable {
        private final Semaphore semaphore;
        private final int id;

        public Worker(Semaphore semaphore, int id) {
            this.semaphore = semaphore;
            this.id = id;
        }

        @Override
        public void run() {
            try {
                System.out.println("Worker " + id + " is trying to acquire the
semaphore.");   semaphore.acquire(); // Acquire the semaphore

                System.out.println("Worker " + id + " has acquired the semaphore.");
                // Simulate some work
                Thread.sleep(2000);

                System.out.println("Worker " + id + " is releasing the semaphore.");
                semaphore.release(); // Release the semaphore
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Simplified Implementation of Semaphore

```java
public class Semaphore {
    private int permits;

    public Semaphore(int permits) {
        this.permits = permits;
    }

    public synchronized void acquire() throws InterruptedException {
        while (permits == 0) {
            wait(); // Wait until a permit becomes available
        }
        permits--;
    }

    public synchronized void release() {
        permits++;
        notify(); // Notify waiting threads that a permit is available
    }
}
```