OS6 - Semaphores and Concurrent DS

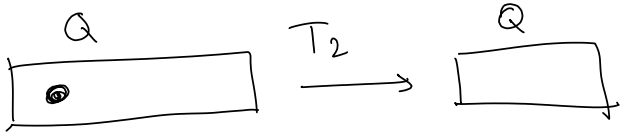
Friday, 14 July 2023 10:32 PM

Producer – Consumer Problem

We have two threads T_1 & T_2

```
// Consumer
1 if (queue.size > 0) {
2     queue.pop()
3 }
```

T1 gets preempted right after line 1.
T2 executes fully and empties the Queue.
T1 starts again from line 2, but the Queue is already empty.




```
// Producer
1 if (queue.size > 0) {
2     queue.push(1)
3 }
```

T1 gets preempted right after line 1
T2 executes fully and fills up the Queue with max size 3
T1 starts again from line 2, but since max size is already reached, it overflows the Queue

overflow

So even though on line 1, we are making checks using if condition, but still those checks are failing. This is due to lack of proper synchronization handling.

This lock only allows only 1 thread at a time.

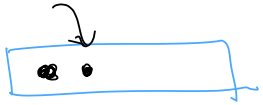
```
class ProducerConsumer {
    private final Lock lock = new ReentrantLock();
    private final Queue<Integer> burgerQueue = new LinkedList<>();

    public void produceBurger() throws InterruptedException {
        lock.lock();
        if (burgerQueue.size() >= 10) {
            System.out.println("Queue is full. Producer is waiting...");
            lock.unlock();
            Thread.sleep(1000);
            produceBurger();
            return;
        }

        // Produce a burger
        burgerQueue.add(1);
        lock.unlock();
    }

    public void consumeBurger() throws InterruptedException {
        lock.lock();
        if (burgerQueue.isEmpty()) {
            System.out.println("Queue is empty. Consumer is
waiting...");lock.unlock();
            Thread.sleep(2000);
            consumeBurger();
            return;
        }

        // Consume a burger
        int consumedBurger = burgerQueue.poll();
        lock.unlock();
    }
}
```

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Main {
    public static void main(String[] args) {
        ProducerConsumer producerConsumer = new ProducerConsumer();

        // Create producer thread
        Thread producerThread = new Thread(() -> {
            try {
                while (true) {
                    producerConsumer.produceBurger();
                    Thread.sleep(1000);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        // Create consumer thread
        Thread consumerThread = new Thread(() -> {
            try {
                while (true) {
                    producerConsumer.consumeBurger();
                    Thread.sleep(2000);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        // Start the threads
        producerThread.start();
        consumerThread.start();
    }
}
```

Using Semaphores to allow multiple threads

Producer

P.acquire() P--=1

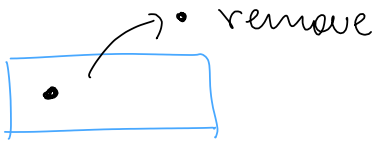


C.release()

C++=1

Consumer

C.acquire() C--=1



P.release()

P++=1

P = 3

C = 0

maxSize = 3

P_1 ⟶ P-- = 2, C++ = 1

P_2 ⟶ P-- = 1, C++ = 2

C_1 ⟶ C-- = 1, P++ = 2

Kartik Bhatia11:09 PM

https://leetcode.com/problems/the-dining-philosophers/
Kartik Bhatia11:09 PM

https://leetcode.com/problems/building-h2o/
Kartik Bhatia11:08 PM

https://leetcode.com/problems/building-h2o/
Kartik Bhatia11:08 PM

https://leetcode.com/problems/fizz-buzz-multithreaded/description/
Kartik Bhatia10:56 PM

https://leetcode.com/problems/print-in-order/

Atomic Data Types (ThreadSafe)

The ideal way to use variables/data types in production code is not just to use locking mechanisms, but a better alternative is to use Atomic Data Types.

int → AtomicInteger

AtomicInteger count = new AtomicInteger(0); //initial value

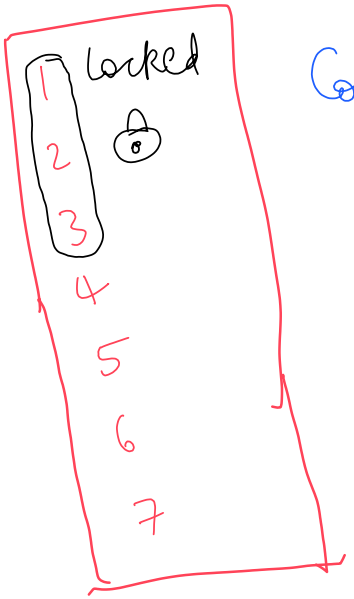
count += 1 → count.getAndAdd(1)
count -= 1 → count.getAddSubtract(1)

Atomic Data Types eliminate the need to implement Locks/Semaphores manually as they have all those mechanism built in and make sure that they are well synchronised in a multithreaded environment.

Concurrent Data Structures

Eg.
In a **normal Hashmap**, when one thread is working on the hashmap, even for any particular

key, the whole hashmap is locked for that thread and no other thread can have access to it unless the lock is released.
Whereas, in a **Concurrent Hashmap**, for a thread accessing the HM, only the bucket associated with the particular key is locked.



Concurrent Hashmap