Thread synchronisation

Agenda

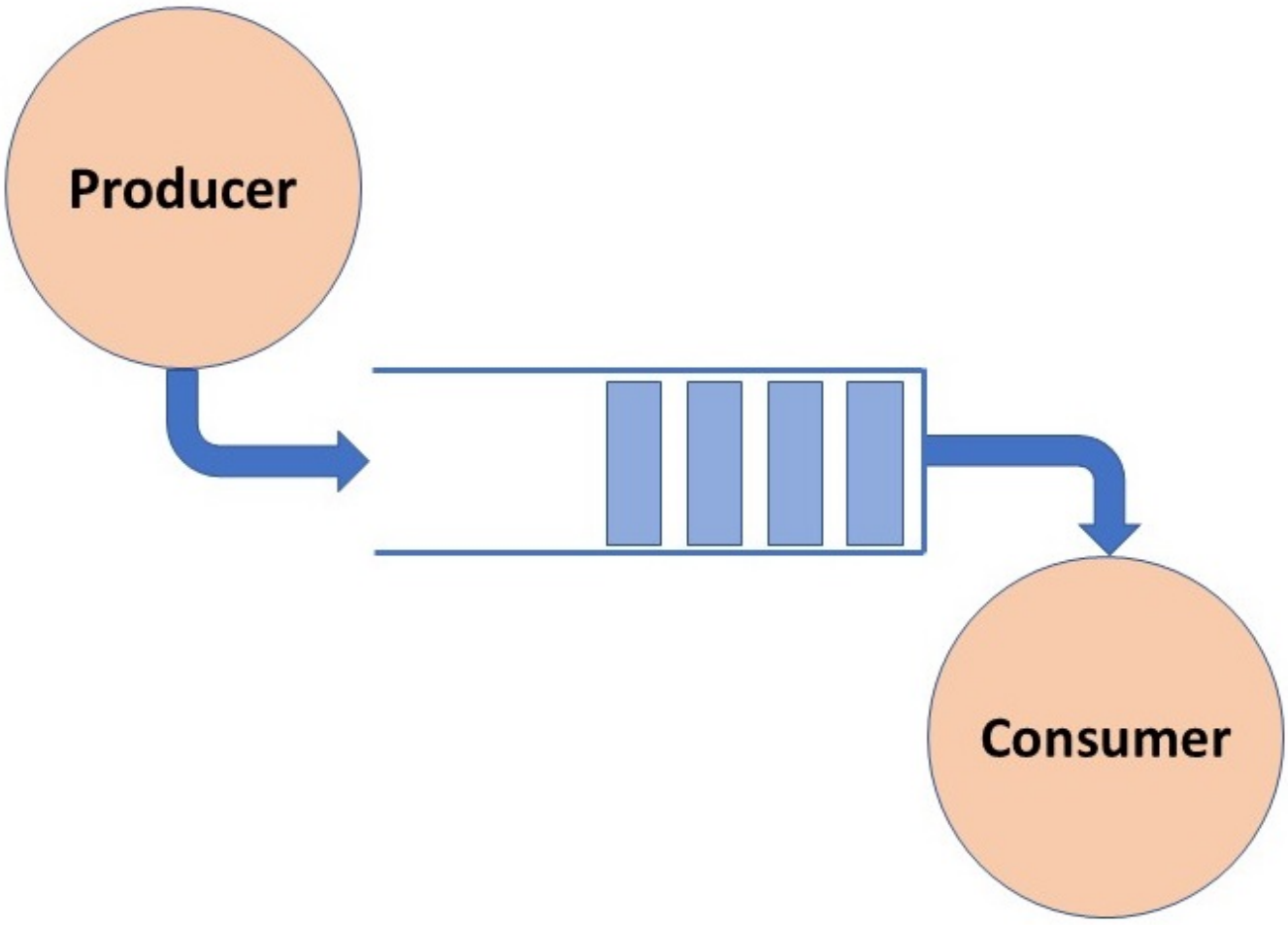
- Producer-consumer problem
- Semaphores
- Concurrent data structures
    - Atomic Integer
    - Concurrent Hash Map

Producer-consumer problem

The Producer-Consumer problem is a classic synchronization problem in operating systems.

The problem is defined as follows: there is a fixed-size buffer and a Producer process, and a Consumer process.

The Producer process creates an item and adds it to the shared buffer. The Consumer process takes items out of the shared buffer and “consumes” them.



Certain conditions must be met by the Producer and the Consumer processes to have consistent data synchronisation:

- The Producer process must not produce an item if the shared buffer is full.
- The Consumer process must not consume an item if the shared buffer is empty.

Producer

The task of the producer is to create a unit of work and add it to the store. The consumer will pick that up when it is available. The producer cannot add exceed the number of max units of the store.

```
    public void run() {
        while (true) {
            if (store.size() < maxSizeOfStore) {
                store.add(new UnitOfWork());
            }
        }
    }
```

Consumer

The role of the consumer is to pick up unit of works from the queue or the store once they have been added by the consumer. The consumer can only pick up units if there are any available.

```
    public void run() {
        while (true) {
            if (store.size() > 0) {
                store.remove();
            }
        }
    }
```

The above code will lead to concurrency issues since multiple thread can access the store at one time. What happens if there is only one unit present, but two consumers try to acquire it at the same time. Since the size of the store will be 1, both of them will be allowed to execute, but one will error out.

Base Solution - Mutex

In order to solver the concurrency issue, we can use mutexes or locks. In Java this can be achieved by simply wrapping the critical section in a synchronised block.

```
    public void run() {
        while (true) {
            synchronized (store) {
                if (store.size() < maxSizeOfStore) {
                    store.add(new Shirt());
                }
            }
```

```
        }
    }
```

Now only one thread will be able to access the store at one time. This will solve the concurrency issue, but our execution does not happen in parallel now. Due to the mutex, only one thread can access the store at a time.

Parallel solution - Semaphore

A semaphore is a variable or abstract data type used to control access to a common resource by multiple threads and avoid critical section problems in a concurrent system such as a multitasking operating system. The main attribute of a semaphore is how many thread does it control. If a semaphore handles just one thread, it is effectively similar to a mutex.

To solve our producer and consumer problem, we can use two semaphores:

1. For Producer - This semaphore will control the maximum number of producers and is initialised with the max stor size.
2. For Consumer - This semaphore controls the maximum number of consumers. This starts with 0 active threads.

```
Semaphore forProducer = new Semaphore(maxSize);
Semaphore forConsumer = new Semaphore(0);
```

The ideal situation for us is that:

- There are parallel threads that are able to produce until all the store is filled.
- There are parallel threads that are able to consume until all the store is empty.

Thus, to achieve this we use each producer thread to signal to the consumer that it has added a new unit. Similarly, once the consumer has reduced the unit of works, it signals it to the producer to start making more.

```
// Producer

while (true) {
    forProducer.acquire();
    store.add(new UnitOfWork());
    forConsumer.release();
}
```

```
// Consumer

while (true) {
```

```
        forConsumer.acquire();
        store.add(new UnitOfWork());
        forProducer.release();
    }
```

Concurrent Data structures

A concurrent data structure is a particular way of storing and organizing data for access by multiple computing threads (or processes) on a computer. A shared mutable state very easily leads to problems when concurrency is involved. If access to shared mutable objects is not managed properly, applications can quickly become prone to some hard-to-detect concurrency errors.

Some common concurrent data structures:

1. Atomic Integer

   > The AtomicInteger class protects an underlying int value by providing methods that perform atomic operations on the value

2. Concurrent hash maps