# Synchronization lab report
# Amo Samuel

In my exploration of multithreaded programming, I came across the need to synchronize access to shared resources. Without proper synchronization, threads can interfere with each other, leading to race conditions or inconsistent data. In this document, I'll explain synchronization concepts and best practices, using a simple **Print Queue System** as an example.

## 2. Using Locks to Synchronize Access

In Java, one of the most effective ways to manage synchronization is through the **ReentrantLock** class. I chose this approach because it provides more flexibility than synchronized blocks. For example:

- I can manually lock and unlock critical sections of the code using lock() and unlock().
- I can also use **condition variables** (Condition objects) to make threads wait until certain conditions are met, such as waiting for space in the queue or waiting for jobs to be added.

Here's how I applied it in the PrintQueue class:

```
public void submitJob(String job) throws InterruptedException{ 1 usage
    lock.lock();
    try {
        while (queue.size() == capacity){
            System.out.println("User waiting..printer queue is full");
            notFull.await();
        }
        queue.add(job);
        System.out.println("User Submitted job: " + job);
        notEmpty.signal();
    }finally {
        lock.unlock();
    }
}
```

**Managing Job Processing with Locks:** I also had to synchronize the way the printer processes jobs from the queue. The printer should wait until a job is available and should process only one job at a time. Here's how I implemented that:

```
public String processJob() throws InterruptedException {  1 usage    new *
    lock.lock();
    try {
        while (queue.isEmpty()) {
            System.out.println("Printer waiting... No jobs in the queue.")
            notEmpty.await();
        }
        String job = queue.poll();
        System.out.println("Printer processing job: " + job);

        notFull.signal();
        return job;
    } finally {
        lock.unlock();
```

This code guarantees that the printer processes jobs one by one and only when there are jobs available. I found that using await() and signal() methods with condition variables helped me manage the interaction between users and the printer effectively.

### 4. Best Practices  for Synchronization

While working on this, I identified several best practices that helped me avoid common pitfalls:

- **Always unlock in a finally block**: By locking and unlocking within a try-finally block, I ensured that the lock was always released, even if something went wrong.
- **Use condition variables to manage thread communication**: In situations where threads need to wait for a certain condition (like waiting for a job in the queue), using await() and signal() was crucial. It allowed me to avoid busy waiting and let threads wait efficiently.
- **Minimize the time spent holding a lock**: I made sure that the locks were held only while adding or removing jobs from the queue. This improved performance by allowing other threads to run as much as possible.
- **Prevent deadlocks**: I was careful to avoid deadlocks by always ensuring that locks were released properly and that no thread was indefinitely waiting for a resource that would never be available.

## Conclusion

In this document, I explored the fundamental concepts of synchronization in multithreaded programming and demonstrated their practical application through a simple print queue system. By utilizing tools like ReentrantLock and Condition, I was able to ensure that multiple threads could safely interact with shared resources without causing conflicts or inconsistencies.