Thread Control and Deadlocks Lab Report

Amo Samuel

Thread Interruption

Thread interruption is a way in Java to signal a running thread to stop what it's doing. This is useful when you want a thread to stop processing, perhaps because the work it was doing is no longer needed, or you want to shut down an application.

When a thread is interrupted, it doesn't immediately stop. Instead, it sets an "interrupted" flag, which the thread can check to decide whether to stop itself.

How I Implemented It:

In my ArrayPrinterTask, I used this concept to make sure that the thread checks if it has been interrupted while it's printing elements of an array. I used Thread.currentThread().isInterrupted() to check the interrupted status of the thread.

If the thread is interrupted, I make it throw an InterruptedException to exit the loop and stop further processing. This approach allows the task to stop safely if needed, which is important for preventing the thread from continuing to work on something that might no longer be necessary.

Fork/Join Framework

The Fork/Join framework in Java is designed to take advantage of multiple processors by breaking tasks into smaller pieces that can be executed in parallel. It works well for tasks that can be divided into independent subtasks, which can be processed simultaneously to improve performance.

The basic idea is to "fork" (or split) a big task into smaller tasks, and once these smaller tasks are done, "join" them together to get the final result.

How I Implemented It:

In my ***MaxValueTask***, I used the ***Fork/Join framework*** to find the maximum value in an array. I split the array into smaller parts and processed each part in parallel. If a part was small enough, I handled it directly; otherwise, I split it further.

I used fork() to run one part of the task asynchronously and join() to wait for it to complete before combining the results. This approach allowed me to efficiently find the maximum value even in large arrays by using multiple threads to share the workload.

Deadlock Prevention

Deadlock happens when two or more threads are waiting for each other to release resources, and as a result, they all get stuck. It’s like a situation where two people are holding keys that the other needs, and neither can move forward.

Deadlock is a serious issue in multithreaded applications because it can cause your program to freeze completely.

How I Implemented Deadlock Prevention:

To prevent deadlock in my ***DeadLockArrayList class***, I used a method called tryLock() instead of the regular lock(). The tryLock() method tries to acquire a lock, but if it can't get it within a certain time, it gives up. This prevents the thread from getting stuck waiting forever.

I also made sure to always acquire the locks in the same order across all threads. This way, I avoided the situation where two threads try to lock the same resources in different orders, which can lead to deadlock.

By using these strategies, I ensured that my code runs smoothly without the risk of getting stuck in a deadlock.

Conclusion

Through working on these concepts—thread interruption, the Fork/Join framework, and deadlock prevention—I’ve learned how to manage multithreading effectively in Java. These tools are essential for building applications that are both efficient and reliable. As I continue to develop more complex software, understanding and applying these principles will help me write code that is robust and safe from common concurrency issues.