THREAD INTERUPTION, FORK/JOIN AND DEADLOCK PREVENTION

Thread Interruption

Interrupts

An interrupt is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate. This is the usage emphasized in this lesson.

A thread sends an interrupt by invoking interrupt on the Thread object for the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.

Supporting Interruption

How does a thread support its own interruption? This depends on what it's currently doing. If the thread is frequently invoking methods that throw InterruptedException, it simply returns from the run method after it catches that exception.

Many methods that throw InterruptedException, such as sleep, are designed to cancel their current operation and return immediately when an interrupt is received.

The Interrupt Status Flag

The interrupt mechanism is implemented using an internal flag known as the interrupt status. Invoking Thread.interrupt sets this flag. When a thread checks for an interrupt by invoking the static method Thread.interrupted, interrupt status is cleared. The non-static isInterrupted method, which is used by one thread to query the interrupt status of another, does not change the interrupt status flag.

By convention, any method that exits by throwing an InterruptedException clears interrupt status when it does so. However, it's always possible that interrupt status will immediately be set again, by another thread invoking interrupt.

Fork/Join

The fork/join framework is an implementation of the ExecutorService interface that helps you take advantage of multiple processors. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of your application.

As with any ExecutorService implementation, the fork/join framework distributes tasks to worker threads in a thread pool. The fork/join framework is distinct because it uses a *work-stealing* algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy.

Understanding the framework classes

The fork/join framework has two main classes, **ForkJoinPool** and **ForkJoinTask**.

ForkJoinPool is an implementation of the interface **ExecutorService**. In general, executors provide an easier way to manage concurrent tasks than plain old threads. The main feature of this implementation is the work-stealing algorithm.

There's a common **ForkJoinPool** instance available to all applications that you can get with the static method **commonPool()**:

The common pool is used by any task that is not explicitly submitted to a specific pool, like the ones used by parallel streams. According to the class documentation, using the common pool normally reduces resource usage because its threads are slowly reclaimed during periods of non-use, and reinstated upon subsequent use.

You can also create your own ForkJoinPool instance using one of these constructors:

```
ForkJoinPool()

ForkJoinPool(int parallelism)

ForkJoinPool(int parallelism,

    ForkJoinPool.ForkJoinWorkerThreadFactory factory,

    Thread.UncaughtExceptionHandler handler,

    boolean asyncMode

)
```

Just like an **ExecutorService** executes an implementation of either the Runnable or the Callable, the **ForkJoinPool** class invokes a task of type **ForkJoinTask**, which you have to implement by extending one of its two subclasses:

- RecursiveAction, which represents tasks that *do not* yield a return value, like a **Runnable**.
- RecursiveTask, which represents tasks that yield return values, like a **Callable**.

These classes contain the **compute()** method, which will be responsible for solving the problem directly or by executing the task in parallel. Most of the time, this method is implemented according to the following pseudo-code:

```
if (problem is small)
  directly solve problem
else {
  split problem into independent parts
  fork new subtasks to solve each part
  join all subtasks
  compose result from subresults
}
```

ForkJoinTask subclasses also contain the following methods:

- fork(), which allows a **ForkJoinTask** to be scheduled for asynchronous execution (launching a new subtask from an existing one).
- join(), which returns the result of the computation when it is done, allowing a task to wait for the completion of another one.

First, you have to decide when the problem is small enough to solve directly. This acts as the the base case. A big task is divided into smaller tasks recursively until the base case is reached.

Each time a task is divided, you call the **fork()** method to place the first subtask in the current thread's deque, and then you call the **compute()** method on the second subtask to recursively process it.

Finally, to get the result of the first subtask you call the **join()** method on this first subtask. This should be the last step because **join()** will block the next program from being processed until until the result is returned.

Thus, the order in which you call the methods is important. If you don't call **fork()** before **join()**, there won't be any result to retrieve. If you call **join()** before **compute()**, the program will perform like if it was executed in one thread and you'll be wasting time.

If you follow the right order, while the second subtask is recursively calculating the value, the first one can be stolen by another thread to process it. This way, when **join()** is finally called, either the result is ready or you don't have to wait a long time to get it.

You can also call the method invokeAll(ForkJoinTask<?>... tasks) to fork and join the task in the right order.

Using the fork/join framework can speed up processing of large tasks, but to achieve this outcome, we should follow some guidelines:

- **Use as few thread pools as possible.** In most cases, the best decision is to use one thread pool per application or system.
- **Use the default common thread pool** if no specific tuning is needed.
- **Use a reasonable threshold** for splitting *ForkJoinTask* into subtasks.
- **Avoid any blocking in *ForkJoinTasks*.**

Deadlock Prevention

Lock Ordering

Deadlock occurs when multiple threads need the same locks but obtain them in different order.

If you make sure that all locks are always taken in the same order by any thread, deadlocks cannot occur. Look at this example:

Thread 1:

    lock A

     lock B

Thread 2:

    wait for A

     lock C (when A locked)

Thread 3:

```
    wait for A
    wait for B
    wait for C
```

If a thread, like Thread 3, needs several locks, it must take them in the decided order. It cannot take a lock later in the sequence until it has obtained the earlier locks.

For instance, neither Thread 2 or Thread 3 can lock C until they have locked A first. Since Thread 1 holds lock A, Thread 2 and 3 must first wait until lock A is unlocked. Then they must succeed in locking A, before they can attempt to lock B or C.

Lock ordering is a simple yet effective deadlock prevention mechanism. However, it can only be used if you know about all locks needed ahead of taking any of the locks. This is not always the case.

Lock Timeout

Another deadlock prevention mechanism is to put a timeout on lock attempts meaning a thread trying to obtain a lock will only try for so long before giving up. If a thread does not succeed in taking all necessary locks within the given timeout, it will backup, free all locks taken, wait for a random amount of time and then retry. The random amount of time waited serves to give other threads trying to take the same locks a chance to take all locks, and thus let the application continue running without locking.

Here is an example of two threads trying to take the same two locks in different order, where the threads back up and retry:

Thread 1 locks A

Thread 2 locks B

Thread 1 attempts to lock B but is blocked

Thread 2 attempts to lock A but is blocked

Thread 1's lock attempt on B times out

Thread 1 backs up and releases A as well

Thread 1 waits randomly (e.g. 257 millis) before retrying.

Thread 2's lock attempt on A times out

Thread 2 backs up and releases B as well

Thread 2 waits randomly (e.g. 43 millis) before retrying.

In the above example Thread 2 will retry taking the locks about 200 millis before Thread 1 and will therefore likely succeed at taking both locks. Thread 1 will then wait already trying to take lock A. When Thread 2 finishes, Thread 1 will be able to take both locks too (unless Thread 2 or another thread takes the locks in between).

An issue to keep in mind is, that just because a lock times out it does not necessarily mean that the threads had deadlocked. It could also just mean that the thread holding the lock (causing the other thread to time out) takes a long time to complete its task.

Additionally, if enough threads compete for the same resources they still risk trying to take the threads at the same time again and again, even if timing out and backing up. This may not occur with 2 threads each waiting between 0 and 500 millis before retrying, but with 10 or 20 threads the situation is different. Then the likeliness of two threads waiting the same time before retrying (or close enough to cause problems) is a lot higher.

A problem with the lock timeout mechanism is that it is not possible to set a timeout for entering a synchronized block in Java. You will have to create a custom lock class or use one of the Java 5 concurrency constructs in the java.util.concurrency package. Writing custom locks isn't difficult but it is outside the scope of this text. Later texts in the Java concurrency trails will cover custom locks.

Deadlock Detection

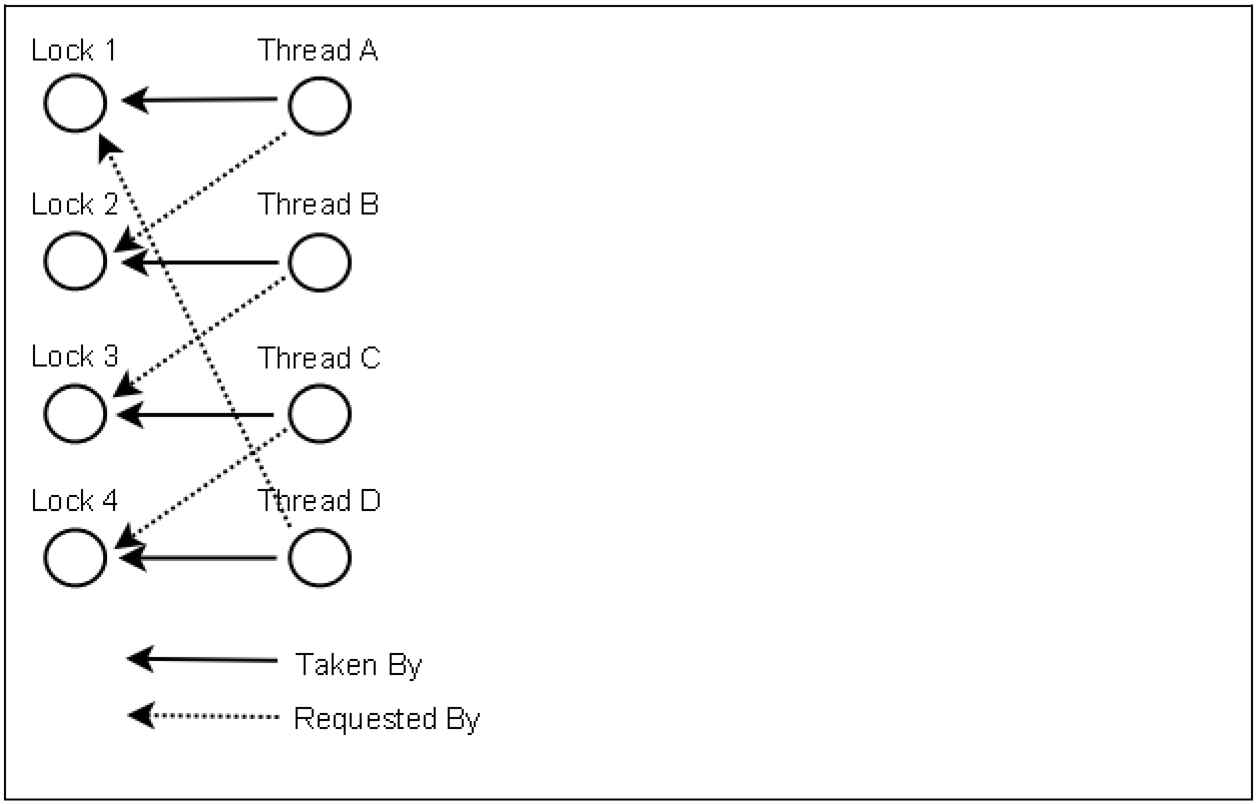
Deadlock detection is a heavier deadlock prevention mechanism aimed at cases in which lock ordering isn't possible, and lock timeout isn't feasible.

Every time a thread **takes** a lock it is noted in a data structure (map, graph etc.) of threads and locks. Additionally, whenever a thread **requests** a lock this is also noted in this data structure.

When a thread requests a lock but the request is denied, the thread can traverse the lock graph to check for deadlocks. For instance, if a Thread A requests lock 7, but lock 7 is held by Thread B, then Thread A can check if Thread B has requested any of the locks Thread A holds (if any). If Thread B has requested so, a deadlock has occurred (Thread A having taken lock 1, requesting lock 7, Thread B having taken lock 7, requesting lock 1).

Of course a deadlock scenario may be a lot more complicated than two threads holding each others locks. Thread A may wait for Thread B, Thread B waits for Thread C, Thread C waits for Thread D, and Thread D waits for Thread A. In order for Thread A to detect a deadlock it must transitively examine all requested locks by Thread B. From Thread B's requested locks Thread A will get to Thread C, and then to Thread D, from which it finds one of the locks Thread A itself is holding. Then it knows a deadlock has occurred.

Below is a graph of locks taken and requested by 4 threads (A, B, C and D). A data structure like this that can be used to detect deadlocks.



So what do the threads do if a deadlock is detected?

One possible action is to release all locks, backup, wait a random amount of time and then retry. This is similar to the simpler lock timeout mechanism except threads only backup when a deadlock has actually occurred. Not just because their lock requests timed out. However, if a lot of threads are competing for the same locks they may repeatedly end up in a deadlock even if they back up and wait.

A better option is to determine or assign a priority of the threads so that only one (or a few) thread backs up. The rest of the threads continue taking the locks they need as if no deadlock had occurred. If the priority assigned to the threads is fixed, the same threads will always be given higher priority. To avoid this you may assign the priority randomly whenever a deadlock is detected.

References

Baeldung

Eugen, B. (n.d.). **Java Fork/Join Framework**. Baeldung. Retrieved from https://www.baeldung.com/java-fork-join

Medium

Schapira, J. (2020, July 6). **Today I Learned: Deadlock and Deadlock Prevention**. Medium. Retrieved from https://medium.com/@jschapir/today-i-learned-deadlock-and-deadlock-prevention-ff5041316ad0

Jenkov

Jenkov, J. (n.d.). **Deadlock Prevention in Java**. Jenkov. Retrieved from https://jenkov.com/tutorials/java-concurrency/deadlock-prevention.html

Oracle

Oracle. (n.d.). **Fork/Join**. Oracle. Retrieved from https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html

Pluralsight

Chadwick, D. (2018, May 17). **Introduction to the Fork/Join Framework**. Pluralsight. Retrieved from https://www.pluralsight.com/resources/blog/guides/introduction-to-the-fork-join-framework