Practical Session #08 - More Concurrency Issues

Tom Mahler
In This Recitation We’ll Cover

- Atomic Instructions
- Producers-Consumers Pattern And Blocking Queues
- Callables
- Thread Cancellation
- Advanced Threads Synchronization Tools
Atomic Instructions
Atomic Instructions

- Most CPU operations (like add, mov etc.) are atomic
  - I.e., the instruction result can be seen as if it "happened at once"

- We only need to use synchronization if we want several such instructions to be atomic

- Most CPUs today offers a set of atomic instructions that are designed for multi-threading - an important one is compare-and-set or CAS
Examples

- CAS is an instruction that behaves as the following example

- See code... (See Even)
The cas instruction is atomic which means that threads sees its result as if it happens at once.

- In other words, the scheduler cannot stop a thread in the middle of this operation - only before or after it.

Although it might not look like one, the cas instruction is extremely useful and powerful.

Java has several classes (all beginning with Atomic* that allow us to use cas).
Examples

- To understand its power, let's rewrite the Even counter class using AtomicInteger:

- See code... (See Even)
Atomic Instructions

- Let's revise the code in the example:
  - AtomicInteger is a class that holds integer that we can call cas on
    - I.e., it will be the target of the cas
  - There is no usage of synchronized anywhere in the class
Atomic Instructions

- Let's revise the code in the example:
  - If there are \( n \) threads \( t_1, ..., t_n \) that attempt to invoke \( \text{add} \) one time all at once then, without the loss of generality:
    - \( t_1 \) will enter the while loop once
    - \( t_2 \) at most twice
    - \( t_n \) at most \( n \) times
Atomic Instructions

- Lets revise the code in the example:
  - This code runs significantly faster than the synchronized one as we do not send a thread to sleep (i.e., invokes a system call) whenever a concurrent access is happen.
Atomic Instructions

- This kind of implementation is called lock-free

- Basically, if threads cannot block each other inside the data-structure implementation, the implementation is called **lock-free**
Atomic Instructions

- Can we create a lock-free implementation for data structures that require operations that are more complex than adding two numbers?
  - YES, in many cases the basic idea stays the same
    - Copy the cas target to a local variable
    - Change this variable locally
    - Try to commit your change using cas
      - If failed, retry
  - This is it!
Examples

- Lets see another example and implement our linked list lock-free:

- See code...
  (See LinkedList)
Atomic Instructions

- Again this implementation uses no locks, it is faster and more efficient.

- **An important note:** while lock-free data structures seem better than lock-based structures, they also have their limitations:
  - Since there are no locks, lock-free data structures cannot block threads, which may be a required property of the data structure.
  - For example, blocking queues.
An important note: while lock free data structures seems better than lock based structures they also has their limitations:

- Lock-Free data structures are notoriously known of being much harder to write correctly
- There are some cases where the state of the data-structure may be complex and may require a lock-free implementation that need to copy (to local variables) large amount of data
  - which may not always be possible or may lead to poor performance
Atomic Instructions

- In other words, in programming - there is no silver bullet
  - you need to master many different techniques and choose the one that is most suitable to your task

- In addition, like most other techniques we learn about in this lecture, java has some pre-made implementation for us to use
Some useful examples of such are:

- ConcurrentHashMap
- ConcurrentLinkedQueue

which you can read more about via their APIs javadoc
Producers-Consumers Pattern And Blocking Queues
A **Blocking Queue** is a queue that blocks a caller if it tries to put an element into a full queue, or take an element from an empty queue, until the operation can be performed.

- When the queue becomes non-full or non-empty, respectively
Blocking Queue

- The Java API contains an interface for Blocking Queue in `java.util.concurrent`, called `BlockingQueue`.

- This interface contains the following methods (among others):
  - `void put(E o)` - Adds the specified element to this queue, waiting if necessary for space to become available.
  - `E take()` - Retrieves and removes the head of this queue, waiting if no elements are present on this queue.
Implementing Blocking Queue With `wait` And `notifyAll`

- We can implement the blocking queue functionality using the `wait()` and `notifyAll` primitives
Examples

- Lets see another example and implement our linked list lock-free:

- See code...
  (See LinkedList)
ArrayBlockingQueue

- An implementation of Blocking Queue is present in java.util.concurrent

- It is called ArrayBlockingQueue
The producer-consumer problem deals with the situation in which several producers generate instances of some product.

Once a producer produces a product, it puts it into a shared bounded capacity queue.
Producer-Consumer With Blocking Queue

- In addition there are several consumers, each may try to consume a product at any time, that is take a product from the shared queue

- A producer should block if the queue is full, and a consumer should block if the queue is empty

- A solution to this problem is illustrated using ArrayBlockingQueue
Examples

▶ See code...
(See ProdCons)
Callables
Callables

- In case you expect your threads to return a computed result you can use `java.util.concurrent.Callable`

- Callables allow to return values after completion

- Callable uses generic to define the type of object which is returned
Since you wish to receive the returned result, you need to keep a connection to the Callable thread.

In order to do so, you may use one of these tools:
- Futures
- ExecutorCompletionService
If you submit a callable to an executor the framework returns a `java.util.concurrent.Future`. This future can be used to check the status of a callable and to retrieve the result from the callable.
Futures

- On the executor you can use the method submit to submit a Callable and to get a future

- To retrieve the result of the future use the get() method
Examples

See code...
(See Futures, Worker)
ExecutorCompletionService

- A CompletionService that uses a supplied Executor to execute tasks

- This class arranges that submitted tasks are, upon completion, placed on a queue accessible using `take`
The class is lightweight enough to be suitable for transient use when processing groups of tasks.

Upon Callable completion, the returned results are added to the ExecutorCompletionService's queue, where you can fetch the results in a convenient manner.
Examples

- See code...
  (See ExecutorCompletionServiceExample)
Thread Cancellation
Thread Cancellation

- Each thread has a predefined task it should be working on.
- When the task ends, the thread finishes working and exits.
- However, sometimes we need to tell a thread to stop working.
  - For example, in assignment 3, threads should stop when the user enters a certain command in the observer.
Thread Cancellation

How do we tell a thread to stop working?

One way is calling the thread's `stop()` command
  This is always unsafe, and you should never do that

Another way is telling the thread it should stop, by changing a variable the thread sees
Examples

▶ See code...
(See Worker)
Thread Cancellation

- Here, another thread can call the `stop()` method of a task, and inform the thread that currently runs the task that it should stop.

- The thread does not stop immediately, and there is no guarantee it will stop at all.
  - In our case, it will stop while it gets to the `while(!this.shouldStop())` line and see that `shouldStop()` returns true.
Thread Cancellation

- The while loops checks `this.shouldStop()` and not `this.shouldStop_`

- Because the call to `stop()` happens from another thread, and we want to guarantee that the current thread sees the change to the `this.shouldStop_` variable, so we must access it under synchronization
Thread Cancellation

- This approach may fail

- Consider the case in which the method `doSomeWork()` has a `wait()` inside it

- It is possible for the thread to never come out of the `notify`, and thus never getting to the `this.shouldStop()` check, and thus never stop
The interrupt() Mechanism

- The above problem is handled with the interrupt mechanism

- Each thread has a flag specifying if it is "interrupted" or not
The interrupt() Mechanism

- Being in an interrupted state means that another thread signals that it should stop
  - Very similar to the shouldStop() method we implemented above

- However, if a thread is inside wait() or sleep() when it is being interrupted, an InterruptedException will be thrown
The interrupt() Mechanism - Summary

- `t.isInterrupted()` is used in code to check if `t` is in interrupted state

- `t.interrupt()` will:
  - If `t` is blocked (i.e. in `wait()` or `sleep()`), then `InterruptedException` is thrown
  - If `t` is not blocked, `isInterrupted()` will return true
The interrupt() Mechanism

- **Note that**
  - If InterruptedException is thrown, then isInterrupted() will return false unless you call interrupt() again
Examples

- See code...
  (See Worker)
interrupt() When Using An Executor

- If using an Executor `e` to run your threads, calling `e.shutdownNow()` will call `interrupt()` on all of the threads in the Executor.
Advanced Threads Synchronization Tools
Advanced Threads Synchronization Tools

- So far we have covered some basic techniques for thread coordination and timing.

- Next we present an example for an advanced model implemented by Java classes as part of the java.util.concurrent package.
CountDownLatch

- A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes

- A CountDownLatch is initialized with a given count
The await methods block until the current count reaches zero due to invocations of the countDown() method, after which all waiting threads are released and any subsequent invocations of await return immediately.

This is a one-shot phenomenon - the count cannot be reset.
CountDownLatch

- If you need a version that resets the count, consider using a CyclicBarrier

- This is a good practice for initialization purposes

- When we need to use some waiting point only once, the latch is best to do the job
For instance, in multilayer games you don’t want any player to start until all players have joined, and this procedure should work only once at the start of the game.