Practical Session #07 - Liveness

Tom Mahler
In This Recitation We’ll Cover

- Few Design Approaches Towards Avoiding Safety Problems
- Java Concurrency
- Thread Safe Singleton
- Concurrency - States
- Guarded Methods - Introduction
Few Design Approaches Towards Avoiding Safety Problems
Sometimes we want to allow several threads to access a resource which is not read-only.

Locking is a means of achieving consistent data. I.e., all threads see the most up-to-date values of shared variables if they synchronize on a common lock.

Locking is achieved using the synchronized mechanism in Java.
Java's synchronization mechanism can be compared to a “room with one key only“.

Entering a synchronized section stands for “taking the key”, and leaving the section “returns the key“.

The “key” is associated with an object instance.

If no implicit pointer is specified, “this” is assumed.
Synchronization Syntax: Synchronized Method

- You can synchronize a method, in which case the object on which it is invoked acts as the synchronization “key”

- The key is acquired when entering the method, and is released on leaving it
Examples

See code...
(See SynchronizedCounter)
Synchronization Syntax: Synchronized Method

- It is not possible for two invocations of synchronized methods on the same object to collide:
  - When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.

- Note that constructors cannot be synchronized — using the synchronized keyword with a constructor is a syntax error.
  - Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed...
Synchronization Syntax: Synchronized Block

- Another option is to synchronize a specific code block

- In this case, you have to explicitly specify the synchronization object
Examples

▶ See code...
  (See SynchronizedCounter)
Java Concurrency
Computer users take it for granted that their systems can do more than one thing at a time.

They assume that they can continue to work in a word processor, while other applications download files, manage the print queue, and stream audio.

Even a single application is often expected to do more than one thing at a time.

For example, streaming audio application must simultaneously read the digital audio off the network, decompress it, manage playback, and update its display.
Java Concurrency

- Even the word processor should always be ready to respond to keyboard and mouse events, no matter how busy it is reformatting text or updating the display.

- Software that can do such things is known as concurrent software.

- The Java platform is designed from the ground up to support concurrent programming, with basic concurrency support in the Java programming language and the Java class libraries.
  - Since version 5.0, the Java platform has also included high-level concurrency APIs.
Java Concurrency

- A better fix for the Counter from the previous practical session is to use an increment method that is thread safe.

- This means that reading the value, incrementing it and assigning the new value will happen within a single clock tick (no other thread can interfere in between).

- This can be achieved using the Java's Atomic library.
  - In the following example, we will use AtomicInteger.
Examples

- See code...
  (See Counter)

- This solution doesn't require blocking other threads while a thread is in the problematic scope
The following is the implementation of `incrementAndGet` in Java 7:

```java
public final int incrementAndGet() {
    for (;;) {
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next))
            return next;
    }
}
```
Java Concurrency

- The `compareAndSet(expected, update)` action is atomic in Java (happens within a single clock tick).

- This action checks if the current value is equal to the expected value.
  - If it is not, it updates the value to the given update.
Thread Safe Singleton
Examples

- Recall the implementation of the Singleton we saw in the previous practical session

- See code...
  (See ClassicSingleton)
Thread Safe Singleton

We mentioned that this implementation is not thread safe because two instances of the Singleton can be created if a thread is preempted after the "if" statement in the getInstance() method.

The following will work:
Examples

- Recall the implementation of the Singleton we saw in the previous practical session

- See code... (See ClassicSingletonSafe)
Thread Safe Singleton

- This class is thread safe because only one thread is responsible for loading classes in java, this implementation will be thread safe and promise only one instance

- **Problem:** whenever the ClassicSingletonSafe class will be imported, the instance will be created
  - This is called **eager initialization** - creating the instance before we actually need it
  - This is a problem because most classes that are designed as a Singleton will be large and their initialization won't be trivial

- We want **lazy initialization** - creating the instance only when it is needed
Thread Safe Singleton

- We need another solution to make the Singleton thread safe

- The natural solution will be to synchronize the getInstance() method
Examples

See code... (See ClassicSingletonSynchronized)
Thread Safe Singleton

- **Problem:** this creates a bottleneck if a lot of threads call the method

- Using synchronized means blocking every thread that wants to get the singleton if another thread is currently getting it

- We want a solution that allows every thread to call this method without blocking it

- How about the following solution?
Examples

See code...
(See ClassicSingletonSynchronized2)
Thread Safe Singleton

- At first sight, it looks good

- We use double checking because when a thread waits for the lock, another thread that already has the lock can create the instance
Thread Safe Singleton

**Problem:** new is not an atomic action
- It breaks down into two actions:
  1. Allocating memory for the object
  2. Calling its constructor

Let's say that a thread was preempted after the first action
- This means that the memory needed for the object was already allocated (the object isn't null) but the constructor wasn't called yet
- If another thread calls getInstance(), the condition won't hold and a reference to an uninitialized object will be returned
Examples

- How about this?

- See code...
  (See ClassicSingeltonSynchronized3)
Thread Safe Singleton

**Problem:** the java compiler uses optimizations and one of them is to get rid of unnecessary variables

So this code, when compiled, will be the same code in the previous solution
Examples

The easiest (working) solution:

See code...
(See ClassicSingeltonSynchronized4)
The SingletonHolder class won't be loaded until we reference it

This means that when we import the ClassicSingletonSynchronized class, the instance won't be created until the moment the getInstance() method is called (the SingletonHolder is referenced)

Because only a single thread loads classes, only one instance will be created and we still achieve lazy initialization
Concurrency - States
A concurrent application's ability to execute in a timely manner is known as its liveness.

This section describes the most common kind of liveness problems, deadlock, and goes on to briefly describe two other liveness problems, starvation and livelock.
Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.
Examples

- See code... (See Deadlock)
Deadlock

- One way of dealing with such problems, is using resource ordering
- All threads should always acquire their locks in the same order
- Another known example related to problems in concurrency is the dining philosophers problem
  - For more info see [here](#)
Starvation and livelock are much less common problem than deadlock, but are still problems that every designer of concurrent software is likely to encounter.
Starvation

- Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress.

- This happens when shared resources are made unavailable for long periods by "greedy" threads.

- For example, suppose an object provides a synchronized method that often takes a long time to return.
  - If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.
Starvation

- Another example, think that you have a pool of threads, and you are in charge of the scheduling
  - Imagine that every thread has a certain priority, so you group threads with similar priority and run them in a round robin manner
    - (i.e. every thread from the same priority group receives a similar amount of time to run)
  - Furthermore, a process from a lower priority group runs only if there is no higher priority process waiting
  - Now, it is clear that threads with lower priority will not run (unless there are no threads with higher priorities)
Starvation

A way to attack this problem, although not in our control, is the usage of **semaphores**, which guarantees some kind of fairness in the threads scheduling, in contrast to the synchronize mechanism and the wait/notify mechanism.
Livelock

- A thread often acts in response to the action of another thread.

- If the other thread's action is also a response to the action of another thread, then livelock may result.

- As with deadlock, livelocked threads are unable to make further progress.
  - However, the threads are not blocked — they are simply too busy responding to each other to resume work.
This is comparable to two people attempting to pass each other in a corridor:

- Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass
- Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left
- They're still blocking each other, so...
Another example, a husband and wife are eating at a restaurant, and they are sharing a fork.
Since each one of them is so in love with the other, they won't eat unless they are sure the other one has eaten first.
This results in a state of livelock:
The wife takes the fork, checks if the husband has eaten, returns the fork.
The husband takes the fork, checks if the wife has eaten, returns the fork.
This situation may go on indefinitely causing a state of livelock.
Examples

- See code...
  (See Livelock)
Guarded Methods - Introduction
Guarded Methods - Introduction

- All the problems presented above originated from our desire to keep our programs running in a timely manner.

- Adding too much safety elements is complicated and hard to predict.

- We would like to be able to synchronize lock and block as less code as possible.

- The guarded methods model gives us this ability.
It is not enough to have thread safe object, we also need to ensure that every thread will not eventually lock up.

This can happen due to locking, waiting, input, CPU starvation, failures.

Sometimes threads depend on each other to continue with execution, and wait for some condition to be met.

The guarded method model delays the execution of a thread until a condition is satisfied.
Guarded Methods - Introduction

- The simplest way to understand the dependency between 2 threads is this case:
  - When T1 sends a message m to a shared object O and the method pre-condition for m is not met, T1 cannot proceed
  - In the sequential case, the only thing we can do that makes sense is to throw an exception
  - In the concurrent case, T1 can wait until another active object T2 changes the state of O so that the precondition holds
Guarded Methods - Introduction

- This means that a new way to deal with precondition failure is for a thread to simply wait
  - With the understanding that another thread will cooperate with us and make the condition we are waiting for become true in the future

- In this practical session we present various methods for the guarded method model

- We begin with basic methods and then we introduce Java classes for advanced timing
Basic Threads Timing

- In this section we talk about basic methods of the guarded method model

- The user is responsible for the timing and setting the appropriate variables

- Notice that the two first examples are usually a wrong solution and the last example is the right solution
Basic Threads Timing

- Before you is the model of the Objects in the following example.

- Notice that only main is calling the change method, and only the created threads are calling the check method.
Busy Wait (a.k.a Spinning)

- Each thread constantly checks whether the condition is met
- This results in heavy CPU usage, and is considered to be a very bad solution
Busy Wait (a.k.a Spinning)

- Imagine a situation where a customer wants to buy a TV, which is not in stock, persistently asks the salesman whether the TV has arrived.

- It would waste a lot of that salesman time to answer this repeating question.

- Spinning is appropriate in very specific cases:
  - When we know that the condition we are waiting for will occur very shortly, and it is critical to react to it with no delay.
  - These cases are extremely rare.
Examples

- See code...
  (See Threads01)
Sleep & Check

- This method is very similar to busy waiting in 2.3.1 but this time the thread is ‘nice’ enough to go to sleep each time the condition fails.

- By doing this we do not waste much of the CPU time.
Sleep & Check

- This time the annoying customer comes every day, asks the question about the TV and then leaves the store

- This time not so much of the salesman's time is wasted

- The disadvantage is that we introduce a delay in how fast the waiting thread will react after the condition is met

- In our example, the customer may miss his TV by 23:55 hours
  - If he asks the salesman at 10:00 on Monday whether the TV is ready, and the TV arrives at 10:05 on Monday, the customer will only get his TV on Tuesday at 10:00
Examples

See code...
(See Threads02)
The wait & notify mechanism supports communication between threads, as follows:

- A thread voluntarily waits until some condition (a prerequisite for continued execution) occurs.
- Some other thread can then notify the waiting thread, to continue its execution.
Wait & Notify

- The most advanced (and most practiced method in reality) is that once the customer has learn that there is no TV in stock, he would leave a phone number.

- Once the TV is delivered, the salesman will call the customer to come and take it.
Wait & Notify

- Once the thread is notified, it must validate the condition again

- This is because multiple threads might be waiting for notification
In our example, once the customers are notified about the TV, only the first one to reach the store would get it.

The others would have to go home and wait again.
Examples

See code...
(See Threads03)
Tips When Using The Wait/Notify Mechanism

▶ Use a `while` loop (and not an `if` condition) to check the precondition
  ▶ This is important because another thread could have changed something so that the condition won't hold before the thread that woke up continued
  ▶ In addition, threads can spontaneously wake up without being notified (doesn't happen very often)

▶ A thread that releases an object's lock, will NOT release other locks it has
Tips When Using The Wait/Notify Mechanism

- `notify()` vs. `notifyAll()` - which of them should you use?
  - When calling `wait()`, `notifyAll()` or `notify()` on an object, make sure the calling thread holds the object's lock (since it changes the object's state by adding/removing to/from its queue
    - Notice that if you do not hold the object's lock you will receive an illegal monitor runtime exception
  - After performing a `wait` call on the object, the thread releases the object's lock and that lock alone, so if the thread acquired more than one lock, it releases only the lock of the waiting object
    - Furthermore, before exiting the `wait` set of the object, the thread must re-lock the object