Multithreading and Synchronization

Lucas Sita | **Christopher Ritz**

Proseminar: Advanced Programming Concepts

1 Introduction

In our world many things happen concurrently, so it is no surprise that we expect the same from our computers. It is our task as programmers to thrive for efficiency. However, problems such as concurrent problems are hard to tackle, in terms of efficiency. A concurrent problem refers to a situation where parts of a program should be executed simultaneously without affecting the final outcome. One approach in order to approach this problem is *multithreading*.
However, how *parallelism* is implemented depends on the architecture of the computer. For instance, computers with one processor can never achieve real parallelism because only one task can be assigned at a time. Nevertheless, they can still accomplish quasi-parallelism by switching from one task to another rapidly, a practice called *multiprogramming* where the user has the illusion of parallelism. Moreover, it depends on the resources if parallelism is well implemented. At best, tasks running in parallel address different resources so that the effort is balanced. The resources on which the use should be balanced are the following three: the processor, the hard disk and the server/network connection.

2 Threads

Concurrent Programs are implemented using threads in Java. The aim of this chapter is thus to explain some threading principles and give the reader an outlook on how they could be implemented. In this chapter all examples will be implemented in Java.
In order to explain the motivation behind threads, we should first introduce the idea of a process:

A process is an instance of a computer program that is being executed. A process consists of the program code (called text section), the value of the *program counter*, the content of the *processor registers*, a *data section* (containing global variables) and data structures such as the *stack* and the *heap*. The stack contains temporary data whereas the heap is memory which is dynamically allocated. Each process has its own address space to protect its data from being corrupted by other processes. This task is accomplished by the virtual memory management of an operating system. Moreover, processes can share data through interprocess communication which will not be discussed further in this chapter.

2.1 What are threads? And why should we use them?

Threads within a process allow the process to execute multiple threads of execution simultaneously, hence the name. All processes consist of at least one thread. Threads share the memory and resources of the process which they belong to. They exist in the same *address space*. The question that arises is whether threads are more advantageous than creating child processes.

There are 4 main benefits associated with multithreading:

1. *Resource sharing.* Processes can share data through interprocess communication techniques such as shared memory or message passing. In terms of efficiency, those methods are far more costly than exchanging data within a process. This leads to a benefit of threads. Since they share the same address space, memory and resources, resource sharing is considerably faster than the above mentioned methods.
2. *Responsiveness.* Multithreading is a necessity in order to create an interactive application. If such an application is single-threaded and has a time-consuming operation, the application will be blocked, and the responsiveness is lost, while multithreaded applications avoid this problem by using another thread to remain responsive.
3. *Economy.* In order to create a process, the operating system allocates memory and resources, which is a costly operation. It is far more economical to create a thread since threads share memory and resources. Furthermore, the context-switch between threads is more efficient than between processes.
4. *Scalability.* In multiprocessor architectures, processes with multiple threads can run simultaneously on different processing cores, whereas a single-threaded process can only run on one processor regardless of how many cores there is at disposal.

The *Java Library* provides classes and interfaces to tackle concurrency. Here are some noteworthy mentions:

`Thread` *(class):*	Every running thread is an instance of this class.[1]
`Runnable` *(interface):*	Program code which is executed by the Java Virtual Machine (*JVM*).
`Lock`*:*	Marks a critical section that only one thread can enter.
`Condition`*:*	Threads can wait for notifications from other threads.

Furthermore, locks and conditions will be discussed in chapter 3.

The Java Runtime Environment offers besides the basic thread class other threads. For example:

Main-Thread: In Java all program code runs in a thread. As soon as the JVM starts up, it creates a thread called Main-Thread. This thread is responsible for the execution of the main-method.

GUI-Thread: As soon as you open a window in Java, there is a thread called GUI-Thread which handles events on a window.

Who am I?: The Java Library provides the method `currentThread`, which returns a reference to the currently executing thread object:

```
public static void main(String[] args) {
   System.out.println(Thread.currentThread()); //Thread[main,5,main]
}
```

2.2 Creating Threads

There are basically two ways for creating a thread in Java:

1. Implement the interface `Runnable`.
2. extend the class `Thread`.

2.2.1 Implementing the Interface `Runnable`

Before a thread can execute code, some instructions are needed. These are the code sequences in the `run` method of a class that implements the `Runnable` interface. The created `Runnable` object is then passed as the target to initialize a `Thread` instance. The `run` method is then executed as soon as the thread is started using the method `start`.
This is what an implementation could look like:

```
public class Proseminar implements Runnable {
    @Override public void run() {
        while(true) {
            System.out.println("Proseminar");
        }
    }
}
```

2.2.2 Extending Thread

However, there is another way to create a thread by simply extending the `Thread` class provided by the Java Library. The class `Thread` also implements the `Runnable` interface. Therefore, the only thing that is left to do is to override the run method, whose body is empty by default. The `run` method is then executed as soon as the subclass calls the method `start` inherited from the superclass `Thread`.
Moreover, a desirable outcome would be that the subclass starts itself when created. To do so we must call the method `start` in the constructer of the subclass so it can start automatically when created:

```
public class Proseminar extends Thread {
    public Proseminar {
        start(); //starts itself when created
    }
    @Override public void run() {
        while(true) {
            System.out.println("Proseminar");
        }
    }
}
```

2.2.3 The Procedures in Comparison

Let us have a closer look at the advantages of both procedures. One recommendation is to use the `Runnable` interface if we are only intend to override the `run` method and no other method from the class `Thread`. A feature in favor of the first procedure is that an object that implements the `Runnable` interface is easy to pass on and therefore flexible.

2.3 Properties and States of Threads

A thread has its own set of properties and states. Some of them are not modifiable once the method start is called. In the following part, we will mention some key properties of the class `Thread`.

Every thread has a name. The name can be set with the method `setName(...)`. The method `getName()` returns this thread's name.

Furthermore, a thread is always in one of the following *states*: *NEW* (the thread has not been started by the method `start()` yet), *RUNNABLE* (it is currently executing), *BLOCKED* (is blocked waiting for an action to take place is in this state), *WAITING* (it is waiting for another thread to call the method `notify()`), *TIMED_WAITING* (waits up to a specified time limit) *or TERMINATED* (the thread has exited). The method `getState()` returns the state of this thread (should not be used for synchronization control). The method returns a constant of the enum type `Thread.state`. Please be carful as these states reflect virtual machine states and not operating system thread states.

The method `isAlive()` tests if this thread is alive. A thread is alive if it has been started and has not died yet.

Each thread has a priority which tells how much runtime is assigned to a thread compared to other threads. A priority is a value between `Thead.MIN_PRIORITY` (1) and `Thread.MAX_PIORITY` (10). A thread gets usually the priority `Thread.NORM_PRIORITY` (5) when initialized. Nevertheless the priority of a thread can be set by the method `setPriority(int newPriority)`. Likewise the current value of the priority can be returned by the getter-method `getPriority()`.

The operating system (or the JVM) prioritizes waiting threads in the waiting-queue according to their priorities. A thread with a higher priority is always favored over those with lower priorities at the next draft. However, it is not guaranteed that priorities are considered (non-determinism) and it always depends on how the operating system is implemented.

In certain circumstances it is necessary to halt a thread for a given time. This can be accomplished with the static method `Thread.sleep(...)`. The reason why this method is static is to avoid that external threads can put each other to sleep. Therefore, it is only the currently executing thread which is put to sleep. The sleep can be interrupted by an `InterruptedException`.

Sometimes a thread must wait for actions (e.g. an input) until it can continue working. One approach would be busy waiting, which means to continuously check whether a condition is fulfilled, e.g. in a while-loop, which is a loss of resources. Another approach would be to wait an arbitrary time using the method `sleep`. This is not efficient since we do not know exactly how much time we must wait for the input. This is why the class `Thread` offers two methods to tackle this problem:

1. *`yield()`: H*ints the scheduler that the current thread is willing to yield its use of the processor. Consequently, a thread with inferior priority gets a chance to execute.
2. *`onSpinWait()`:* Indicates that the current thread is temporarily unable to progress, until the occurrence of one or more actions depending on other actors. Depending on the architecture the JVM can then issue the processor instructions to deal more efficiently with the problem.

Yet the best way is to be acknowledged by an event that some action is completed.

A thread has finished its execution when one of the following is true:

- The `run` method is completely executed (we reached the end of it).
- A `RuntimeException` occurs which only affects the thread itself.
- The JVM has ended prematurely.
- The thread is stopped from outside using the method `stop`.

So, it is only possible to willingly terminate a thread from the outside using the method `stop`. This method however is outdated (DEPRECATED) and it is highly recommended to only use it in emergencies. Instead, `Thread` provides three methods: `isInterrupted`, `interrupt`, `interrupted`. These methods are designed to ask another thread to terminate itself. The following example shows a typical implementation:

The following program prints a message in the console every ½ seconds until another external program sequence (could be another thread) calls the method `interrupt()`. At that point our program gets notified that someone wishes it to stop and does so.

```
Runnable doSomething = () -> {
   System.out.println("Beginning");
   while(!Thread.currentThread().isInterrupted()) {
      System.out.println("Running");
      try {
         Thread.sleep(500);
      } catch (InterruptedException e) {
         Thread.currentThread.interrupt();
         System.out.println("Interrupted while sleeping");
      }
   }
   System.out.println("The End");
   Thread t = new Thread(doSomething); t.start();
   Thread.sleep(2000);
   t.interrupt();
}
```

Note: It is important to put the thread to sleep every 500 milliseconds since the method `sleep` throws an `InterruptedException` and therefore can be caught in the `catch` block. Note that we call the method `interrupt` again in the catch-block because the `sleep` method deletes the flag for the interrupt. Therefore, we must set it again to leave the `while` loop.

3 Synchronization

3.1 Race Conditions and Critical Regions

Since threads share the same address space, it often occurs that they need to operate on the same data. This could be, for example, a database system in which every thread handles one request from the user.
Let us consider an airline reservation system and two users trying to reserve the last seat available on a specific flight. The code which handles such a request on the server side may look like this:

```
public void reserve(User costumer, Seat seat) {
   if(seat.isAvailable()){
      costumer.signal("Seat"+seat+" is available!");
      seat.reserveFor(costumer);
   } else {
      costumer.signal("Seat is not available.");
   }
}
```

fig. 3.1-1

In this abstract code, the server checks whether the seat is available. If so, it lets the costumer know and reserves the seat. This code would work just fine if every function or method would be *atomic*, that is, executed without interruptions. When executed atomically, after we signal the costumer that his or her seat is available, we can be sure that the next instruction that the server will run is the actual reservation for the seat. In case we get multiple requests for the same seat at the same time, we could either a) choose the request that came in first, even just milliseconds before the other ones, or b) choose randomly which request gets processed first.

This system would work. Unfortunately for us, modern computer systems use the concept of *multiprogramming*, which is the rapid switching between multiple programs and/or threads. For us as programmers this means that the execution of our thread can be interrupted at any time by the JVM or the operating system while a different thread gets its turn to start or continue its execution.
With this in mind it is easy to pin down the problem with our code example: We consider two threads running the same code, as shown in fig. 3.1-1. Thread 1 starts first, checking whether the seat is available, which it is. It then signals the costumer that his or her seat is available, which leads to a happy costumer at the other end. At this time, the JVM decides that thread 1 occupied the CPU for too long. Thread 1 gets interrupted and it is thread 2's turn to run. Thread 2 also checks the seat availability, sends a signal to the costumer, and reserves the seat before it has done its job and terminates. Now, thread 1 gets to run again and wants to reserve the seat for its happy costumer, but it is too late since thread 2 already reserved the seat for another costumer. Although both costumers got a message saying their seat is available, only one of them actually got a reservation.

What we just witnessed was a so-called *race condition*, which occurs whenever two or more threads (or processes) are reading or writing some shared data and the final result depends on who runs precisely when. In other words, the threads compete (race) for the shared data, in our example they compete for the last airplane seat.
It is to be mentioned that the time when a thread gets interrupted and switched for another thread can not be known in advance, from our perspective it is random. So running the same program twice can result in two different outcomes, depending on when the JVM or the operating system interrupts our threads.
The key to preventing problems is to achieve *mutual exclusion*, which is restricting the access to the shared data so that only one thread at a time can access it.
For this matter, we distinguish between the times when a thread is doing internal computations, not accessing the shared space, and the times when it does. The part of the program that accesses the shared space is called the *critical region*.
So in other words, what we want to accomplish is to prevent two threads to be in their critical region at the same time and thus preventing a thread to access the data before another thread is done working on it.

In order to have threads working correctly in parallel, avoiding race conditions by accomplishing mutual exclusion is a good start, but not sufficient. In general we need four conditions to hold to have a good solution (A.S. *Tanenbaum, H. Bos, Modern Operating Systems)*:

1. No two processes may be inside their critical regions simultaneously.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running inside its critical region may block any process.
4. No process should have to wait forever to enter its critical region.

Although referring to processes, those four rules apply equally well to threads.
We will now discuss solutions to race conditions, while keeping the four rules in mind.

3.2 Solving Race Conditions

3.2.1 Lock Variables: A Negative Example

A solution that comes intuitively to mind is that we just use one lock variable as an entrance control into each critical region. Like a traffic light that controls when each lane can pass the intersection, we use one global variable to control which thread can enter its critical region.
We assume our lock variable is a boolean variable declared under `LockVar.lock` with `false` meaning no thread is in its critical region right now and `true` meaning there is one thread in its critical region.
Before entering its critical region, each thread would now observe the lock variable, that is checking continuously for it to become `false` (busy waiting). Upon entering its critical region, the thread would set the lock to `true`, and back to `false` once it leaves its critical region. The thread would wait in the while-loop until the data becomes available.

The code frame would look like this:

```
while(LockVar.lock == true) {}
LockVar.lock = true;
//Critical Region
LockVar.lock = false;
```

fig. 3.2.1-1

Of course, continuously checking the lock using a while-loop is very resource-intensive. But there is a bigger problem here: This approach contains the exact same flaw that we have seen in fig. 3.1-1. Suppose a thread waits in the loop until the lock becomes `false`. It then jumps out of the loop but then gets interrupted, before it can set the lock to `true`. At this point another thread gets to run, sees that the lock is `false`, sets it to `true` and enters its critical region. During the time the second thread is in its critical region it gets interrupted and the first thread runs again, now setting the lock to `true`, which is unnecessary since the second thread already did that. In this moment, both threads are in their critical region, a state that violates rule number 1.

3.2.2 Locking Threads

In part 2.1 we introduced the idea of an atomic method. If we could execute a method in a single, atomic action we would not have to worry about the JVM or the operating system interrupting our thread at any given moment. Since the operating system itself can always interrupt any kind of process or thread, we can not guarantee full atomicity in Java. But we can guarantee atomicity concerning mutual exclusion: Java provides ways to lock threads so they operate mutually exclusive on the same data object.
There are two ways of accomplishing that. We will look at both of them:

Locking with ReentrantLock

First, we take a look at the code frame:

```
myLock.lock();
try {
   //Critical Region
} finally {
   myLock.unlock();
}
```

fig. 3.2.2-1

We see that, similar to using our lock variable in 3.2.1, we lock before entering our critical region and unlock after exiting it. In contrast to our simple lock variable, the JVM knows what we are trying to accomplish here and supports us:
When a thread calls the method lock(), it either gets granted access into its critical region, meaning it can just move on, or it has to wait until another thread already inside that critical region unlocks. A thread waits by remaining inside the `lock()` method. Once it's granted access, it can move on.
While a thread is inside its critical region and has not unlocked yet, no other thread can get beyond its own `lock()` call. That is why it is very important for a thread to unlock the critical region once it is done. If it does not, no other thread would be able to enter the critical segment, leaving us in a deadlocked state in which every thread is waiting to enter the critical region stands still, unable to continue. This is the reason why in fig. 3.2.2-1 we use a `try-finally` block to unblock even if anything inside our critical region throws an error.

The object `myLock` is an instance of the class `ReentrantLock`, which implements the `Lock` interface. Important for us is that, before being able to use the `lock()` method, we have to create a new instance of `ReentrantLock` as a private attribute in the corresponding class:

```
private Lock myLock = new ReentrantLock();
```

It is also important to note that a lock securing a critical region is only valid for one instance of the class it is in. To better understand this, let us consider our flight reservation example from 3.1:

```
class Airplane {
   private Lock planeLock = new ReentrantLock();
   //...
   public void reserve(User costumer, Seat seat) {
      planeLock.lock();
      try{
         if (seat.isAvailable()) {
            costumer.signal("Seat "+seat+"is available!");
            seat.reserveFor(costumer);
         } else {
            costumer.signal("Seat is not available");
         }
      } finally { planeLock.unlock(); }
   }
}
```

fig. 3.2.2-2

Here we have wrapped our method `reserve(…)` in a class called `Airplane`. If we want to reserve a seat, we ask the object of the aircraft this seat is in (how we find out in which plane our seat is does not concern us here). Furthermore, we secure the critical region in which we actually reserve the seat against race conditions, meaning our initial problem is solved because we achieved mutual exclusion.

Now, if we have two instances of the type `Airplane`, we also have two separate locks, one inside each `Airplane` object. While a thread is locked out of the critical region of one `Airplane` object, it can very well access the critical region of a different `Airplane` object, if another thread is not already in that one.

Locking with "synchronized"
Although using dedicated lock instances gives us a good amount of control over our critical regions, there is a more concise approach to accomplish mutual exclusion on one method: the keyword `synchronized`, which is used as follows:

```
public synchronized void reserve(User costumer, Seat seat){...}
```

The method body is the same as in fig. 3.1-1.
This code is roughly equivalent to the one we saw in fig. 3.2.2-2, meaning it accomplishes the same thing: mutual exclusion on this method. In a sense, the keyword `synchronized` is more strict than using a `ReentrantLock`: A thread can't even get into the method when it is synchronized, while when using a `ReentrantLock`, multiple threads can be in the same method of the same object, but never between the `lock()` and `unlock()` calls. We could modify our example so that the first thing a thread does upon calling the method `reserve(…)` is to check whether the seat is available before calling `lock()`, entering the critical region and actually reserving the seat. Multiple threads could then call and be inside the `reserve(…)` method, but only one thread can be inside the critical region, actually reserving the flight. First checking whether it is even necessary to enter the critical region would ensure that no thread locks out other threads for no reason, which is the concept behind lock conditions.

3.2.3 Lock Conditions

A lock instance can have so-called conditions. A thread has to meet these conditions in order to enter the critical region. We could imagine a thread waiting for an already reserved seat until this seat is made available, for example because another costumer withdrew his reservation. We create a new condition for our `planeLock` object as an attribute of our `Airplane` class:

```
seatAvailable = planeLock.newCondition();
```

A thread now waits for a seat to become available by calling the `await()` method:

```
while(!seat.isAvailable()) {
          seatAvailable.await();
}
```

fig. 3.2.3-1

A thread calling the method `await()` of our new condition object waits inside this call until another thread gives a signal that the conditions may have changed. This means in our class `Airplane` there could be another method `cancelReservation(…)`. If a thread calls this method and cancels a reservation, it signals this to our Condition object `seatAvailable`, and thus to all other threads waiting:

```
planeLock.lock();    //Using the same lock object as in other methods
try {  seat.cancelReservationFor(costumer);
       seatAvailable.signalAll(); //Notifying waiting threads: a new
                                  //seat is available!
} finally { planeLock.unlock(); }
```

fig. 3.2.3-2

Once a thread calls `.signalAll()` it lets other threads know that the condition, for which other threads are waiting to change, actually changed. In our case it is a new seat that became available. Our waiting thread now jumps out of the `await()` call and checks the loop condition if the seat, which just became available, is the seat it has been waiting for. If so, it jumps out of the loop and reserves the seat inside the critical region. If it's not the seat the thread has been waiting for it goes back into the `await()` method to wait longer.
The `while` loop is not required by Java, but it makes sense to implement it this way, otherwise a thread would just 'give up' after the first seat that became available is not the seat it has been waiting for.
It is important to mention that only a thread actually holding the lock can call its method `signalAll()`. This is the case in our example: Inside our `reserve(…)` and `cancelReservation(…)` methods we use the same `Lock` instance `planeLock` of our class `Airplane`. Also, we can only use this approach of lock conditions when using a lock as an instance of `ReentrantLock`. When using the keyword `synchronized` the API is a little different, but accomplishes the same after all. An overview about both versions of the API is given below.

Synchronizing API

```
java.util.concurrent.locks.Lock
      • void lock()
      • void unlock()
      • Condition newCondition()

java.util.concurrent.locks.Condition
      • void await()
      • void signalAll()

java.util.concurrent.locks.ReentrantLock
      • ReentrantLock()
```

When using synchronized:

```
java.lang.Object
      • void notifyAll()
      • void wait()
      • void wait(long millis)
```

3.3 Deadlocks and Avoidance

Now we can let threads wait until other threads explicitly give a signal, but this brings us to another problem, which we will very briefly discuss: deadlocks.
Imagine a generic scenario in which thread A waits for a condition X to change. Thread B wants to change condition X, but first it has to wait for a different condition Y to change. Unfortunately, the only thread able to change condition Y is thread A. The two threads are waiting for each other to signal them to continue, the threads stand still. This state is called a *deadlock*.
One (very general) way of avoiding a deadlocked state is to only let threads run if they hold all the locks and objects they need to accomplish their task. Using this approach, a thread would never have to wait for an object to become available, thus eliminating the issue.
Unfortunately, Java does not provide a way to avoid or detect deadlocks, although there are some theoretical approaches to the issue. A good overview is provided in Tanenbaum's and Bos' book "Modern Operating Systems".

4 Summary

Parallel computing is crucial for modern systems and in Java we can achieve it with threads. We have seen how threads are created and how we can stay on top of all the threads we create by using their properties and provided methods. The performance gain we get from using them is often greatly beneficial, but when working on the same data problems can arise, for example race conditions, which can be solved by locking the critical regions of the program, thus achieving mutual exclusion. To make this approach a little more efficient, we introduced lock conditions, before ultimately we briefly touched deadlocks. Having these tools in mind during development can greatly improve a system's efficiency, which is always desirable.

Literature:

Andrew S. Tanenbaum, Herbert Bos: *Modern Operating Systems.* Pearson, 2015.

Cay S. Horstmann, Gary Cornell: *Core Java Band 2 - Expertenwissen*. Sun Microsystems Press, 2005.

Christian Ullenboom: *Java ist auch eine Insel*. Galileo Computing, 2012.