Multithreading and Synchronisation

Introductory seminar: Advanced programming concepts

Boris Kosmynin
boris.kosmynin@rwth-aachen.de

Lennart Meyer
lennart.friedrich.meyer@rwth-aachen.de

May 18, 2017

1 Introduction

Every user should be familiar with the concurrency of applications in one or another way. The most apparent way is during the day to day usage of a computer: while copying multiple files or simply by listening to music while writing a document. But it does not mean that all the Threads of which any process are one composed are run simultaneously. Depending on the properties of the hardware used, Threads can be run in a pseudo-parallel way if only one processing unit is available or actually simultaneously on a multi-core or even multi-processor machine[1, 1.1]. In the case of Java, which runs in a virtual environment called the Java Virtual Machine or for short JVM, it is decided at run-time whether the operating system and underlying hardware are able to run and schedule multiple Threads. Then the scheduling task is given (if possible) to the operating system for a better utilisation of the hardware and a better performance or kept by the JVM which would then emulate a suitable scheduler, even on a device which is not able to handle multiple, parallel processes[2, 12.1.1]. The fact that the JVM will emulate the necessary environment adds an important compatibility feature which allows for mainstream deployment of multithreaded applications.

2 Scheduling

Scheduling is the process of assigning resources to special tasks or in our context to Threads. This very important task is carried out by a component of the operating system, the scheduler if available or emulated by the JVM if not. There are many methods by which the scheduler can assign resources. A very well known one is the round robin, which assigns every task a time slot and cycles through them[3, p.178].
As scheduling is the basis of multithreaded computing a well-developed scheduler is essential for performance and user satisfaction. As already mentioned in the introduction there are two basic schedulers: a native one, handled by the operating system, and the one of the JVM. We are going to take a look at the native types of schedulers.
These can be divided into three categories: Batch, Interactive and Real-Time. Batch scheduling is most often used in the business and industry worlds for all kinds of computations for which

people usually do not wait. Interactive scheduling is used in systems with multiple, independent users, who all want compute-time and do not want to wait for it. Also, it has to keep processes from hogging the CPU and blocking out other processes and users. The last kind is Real-Time scheduling, but it does not need a normal scheduler because processes running on such systems are aware of their timing or block quickly[4, pp.153]. In the chapter Scheduling Goals we are going to take a closer look at the most important goals and most common algorithms to accommodate this task.

2.1 Scheduling Goals

For a running system it is important that the scheduler handles all processes in a fair way. That means that every process has a fair share of the compute resources. Also policy enforcement is essential for a stable and secure operation. High-priority processes like safety control should not be killed under any circumstances, even if a process gets delayed. Another important goal is to keep the system balanced and busy. For example processes which utilize CPU power can run simultaneously with processes that utilize the I/O, without compromising the performance.
Depending on the scheduling system, other important goals are in place: For Batch-systems it is important to maximise the systems' throughput and to minimise the turnaround time. That means that the number of jobs per hour should be maximised and the time needed for the completion(from submission to termination) of tasks should be minimised.
On the other hand interactive-systems give greater value to minimising response time and to meet users' expectations. Minimising the response time is a mostly trivial concept but the banaler it gets the more important is a well-conceived implementation. No user wants to wait for an application to schedule when he clicks on its icon. This directly ties into meeting an users' expectations: While a file transaction like uploading or copying is expected to take time, other tasks, like changing the volume, should be processed as fast as possible.
Real-Time systems have other properties, thus different goals. This type of system is used for example in video and audio playback, thus data consistency has a high priority when scheduling processes[4, pp.154-156].

2.2 Scheduling in Batch Systems

Now we are going to have a look at some scheduling algorithms for batch systems.

First-Come, First-Served The most basic algorithm there is. It assigns computing resources as tasks come in and queues all further tasks. Its strength is its simplicity and fairness, as every process gets the time it needs. The disadvantage is the lack of resource distribution. As an example consider a compute-bound process that takes one second to complete followed by an I/O-bound process that does 100 reads or writes. Instead of splitting the compute-bound process and inserting one read or write at each split, barely slowing the process down, it first completes one process and then starts another[4, pp.156].

Shortest Job First This also very basic algorithm optimises the turnaround time of a system by running the short jobs first. At this point it is implied that one needs to know the run times in advance. As example lets' assume we have four jobs A, B, C, D with run times of 8, 4, 4, 4 minutes respectively. Now we can see in Figure 1 that the overall time for the queue does not change, but instead of taking 8 minutes for A, 12 for B, 16 for C and 20 for D, which is 14 minutes on average, the turnaround time is now 4, 8, 12 and 20 minutes, which is only 11 minutes on average. At this point it is worth pointing out that an optimal result is only achieved if all jobs are available simultaneously, and no wait time needs to be taken into account[4, pp.157].

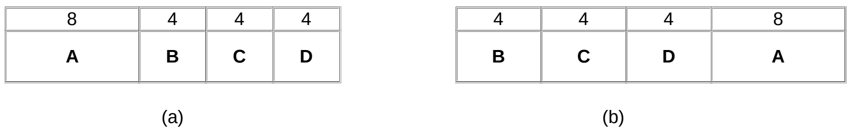

Figure 1: An example of Shortest-Job-First scheduling[4, p.157].

Shortest Remaining Time Next This algorithm is a variant of the Shortest-Job-First algorithm. It additionally needs to know the expected run times and compares the run time of each new job with the remaining time of the current one. If the new job needs less time to finish then the current, the current job gets suspended and the new one started[4, p.157]. It should be noticed that jobs may be suspended for longer periods if the incoming jobs have smaller run times than the remaining time of the current job.

2.3 Scheduling in Interactive Systems

The following algorithms can be used to schedule an interactive system.

Round-Robin Scheduling Round-Robin is one of the oldest, simplest, fairest and most widely used algorithms. Each job is assigned a time slot, called a quantum. If a job exceeds its quantum it gets suspended and is moved to the end of the queue. Then the next job gets started. If a job finishes before its quantum has elapsed the next one starts directly. The only really interesting issue with Round-Robin is the length of a quantum. After each quantum the system requires some time to do administrative work, like saving and loading registers or reloading the memory cache. So depending on the quantums length quite some time is wasted. But longer quanta while improving the CPU efficiency may also worsen the turnaround time, if many jobs are added to the queue at once. A reasonable quantum length is often around 20-50 msec[4, pp.158].

Priority Scheduling Round-Robin assumes that all processes have the same priority, but not all operators share this opinion. This leads to priority scheduling. The process with the highest priority is allowed to run and the rest is queued up behind it. To keep processes from running indefinitely, each process either gets an maximum time quantum and is pushed back in the queue when the quantum expires or the priority gets reduced at each clock until another process has the highest, which then gets run and so on. Priorities can be assigned statically or dynamically. Statically means that each group[1] has a fixed value, which gets applied as the priority of the

[1]For example grouped by the users rank or the price of an option in a commercial data center.

processes it is running. Dynamically means that the system assigns priorities depending on its' goals. An often used variation is the combination of priority scheduling and Round-Robin scheduling, where processes are grouped into priority classes, which use priority scheduling, but use Round-Robin inside of the classes[4, pp.159].

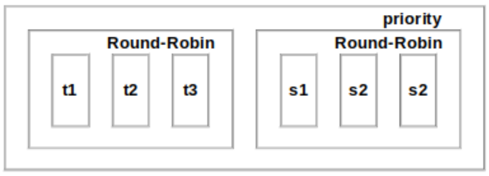

Figure 2: An example of the combination of Round-Robin and Priority scheduling with the tasks $t1, t2, t3, s1, s2, s3$.

3 Synchronisation

Synchronisation is an omnipresent term in the common cooperation and communication of Threads. When several Threads access the same variables, data and commands, it might be possible that unintentional concurrency occur between them. As the scheduler coordinates the priorities of the different Threads it becomes possible that one Thread is influenced to a greater extent than another. So one Thread has more influence to a variable than the other ones. Moreover, if a Thread executes a non-atomic operation[2] and the execution is interrupted by the scheduler, the operation will not be continued until the Thread is given permission to go on working. This can lead to inconsistent results, because the evaluation of the command won't give any sense to the user[5, pp.479].

3.1 Concepts of synchronisation

There are different ways to realize synchronisation in Java. Now we are going to explain the concepts of Monitoring and Locks.

3.1.1 Monitoring

In Java, several commands and methods represent a critical part of Threads. A keyword capsules these commands or methods, so they are locked for other Threads as soon as one Thread wants to use them. The keyword `synchronized` defines the beginning of the critical areas. For example: When one Thread uses the synchronized command or method, a control barrier will be set to the common used object-variable or to the this.-Pointer of the `synchronized` method, so other Threads have to wait until the current Thread has finished its task and deallocates the `synchronized` block[5, p.481-485].

Wait and Notify Next to the principle of monitoring, the methods `void wait()` and `void notify()` are equally important to synchronisation. They provide a trouble-free control in the chronological sequence of events between several Threads. Besides the barrier, the object variable, which is marked with the keyword `synchronized`, contains a waitlist for Threads. Until a Thread enters

[2]Atomic operations are not interruptible by other operations.

a monitored area and claims for its work, the method `wait()` is able to remove the barrier and the Thread will be set to the wait-modus by scheduling. It is guaranteed that the Thread does not disturb other processes and consume too much computing power while waiting. When in another place of the program a Thread works on the object variable inside of a monitored area and uses the `notify()` method, the Thread that is in a wait mode, is allowed to go on working. Furthermore, the methods are only allowed to implement within the critical areas. In literature for Computerterms the scenario is described through an example of the interaction between producer and consumer: The consumer waits until the producer is finished so that the consumer is able to use the product [5, p.485-488].

Interrupt() Next to `notify()` the method `void interrupt()` is able to reactivate waiting Threads. Every Thread is equipped with an Interrupt-Flag. It can be influenced by the non-static method `t.interrupt()` to `True` or `False`. When an Interrupt-Flag of a Thread `t` is set to `True` and then executes the method `wait()` afterwards, the interpreter throws instantly the `InterruptedException`. This implicates that the Thread is henceforth interrupted. Therefore the method `interrupt()` is able to interrupt other waiting Threads instead of its own.
To check the Interrupt-Flag the method `boolean isInterrupted()` is used to establish the Status of a Thread[6, pp. 350].

3.1.2 Locks

In contrast to monitors, locks do not have any keywords like `synchronized`, so the programmer is forced to set up and to remove Locks manually using the methods `lock()` and `unlock()`. One advantage in contrast to monitors is that Locks are not set in blocks. It is possible that the lock is set in a different method than the unlock of the barrier. Another advantage is that locks are differentiable, whether they are used for writing or reading[1, pp.141].

ReentrantLock Locks are implemented by the interface `ReentrantLock`. It provides other useful methods to organize the communication between several Threads. One example is that the programmer is able to call up the waiting Threads[1, pp.142].

ReadWriteLock The interface `ReadWriteLock` is used to specialize and characterize existing Locks. So the programmer is able to differ between writing and reading by using the methods `readLock()` and `writeLock()`[1, p.143].

Conditions For solving general problems of synchronisation, Locks are equipped with the interface `Conditions` which can be compared to the methods `wait()` and `notify()` under the alias term `await()` and `signal()`[1, pp.143].

4 Possible Issues and Errors

A Thread is limited in the way it communicates with other Threads. Issues and error conditions sneak into the process without any knowledge of the programmer.

4.1 Deadlocks

Deadlocks occur when one Thread is locked and a second Thread has not the opportunity to unlock it, because it also has to wait. Both Threads are useless during the whole program-cycle because they stay under mutual dependency to each other. This phenomenon is known as

Deadlock. One example: The first client goes to an ATM and wants to withdraw money, but the ATM is empty and a Bank-Officer can not refill the ATM because the bank has a short run of money. The second Client is not able to deposit his money, because the first Client blocks the entrance. Without communication it is not possible to remove the disagreement and to solve the problem. A Deadlock is difficult to detect, because it does not appear by compiling. The only way to prevent Deadlocks is to adjust the code[7, p.61-64].

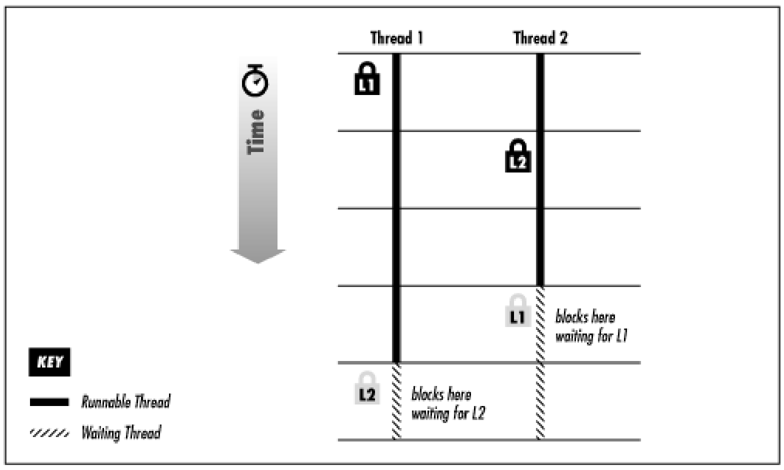

Figure 3: Deadlock[7, p.63]

4.2 Exceptions

Multithreading does not depend on special hardware or is not involved in a complex interaction structure. It only needs one or more processors and additional memory to run Threads correctly. Memory and processor errors are caused by the whole virtual machine and not by a single Thread instead. Generally speaking there should not occur any errors. The only serious errors are generated by the programmer himself when he coded incorrectly or fits the synchronisation in a wrong place in the program. To lean forward these issues, Java is able to `throw` exceptions. In the next part, we are going to introduce the common Exceptions which occur when the programmer wants to work on Threads[7, pp.217].

InterruptedException `InterruptedException` is the most emerged Exception by Thread implementations. When the programmer uses the `sleep()` method and the exception is thrown by the interpreter it implies that the Thread does not sleep for the stated amount of time. In addition, when the `wait()` method is implemented in the program and the `InterruptedException` occurs, it is possible that the `wait()` method does not gain the notification to set the Thread into the line[7, pp.218].

IllegalThreadStateException The `IllegalThreadStateException` and `IllegalArguementException` are specialized as `run-time exceptions`. The `start()` method which is implemented in the Thread class too, throws this type of Exception when a Thread was started in the past but is activated again by this method[7, pp.220].

IllegalArgumentException When the user wants to allocate a priority manually to a Thread and the priority is not remitted, the `IllegalArguementException` is thrown by the interpreter[7, p.221].

5 Practical Implementation in Java 8

In the following we are going to take a look at how to implement multithreading in Java.

5.1 The Thread Class and the Runnable Interface

The first thing to know is that one has the option to use the Runnable interface or to extend the Thread class. It is considered the better practice to extend the Thread class if possible, but because Java does not allow for multiple inheritances there is the option to implement the Runnable interface. This option has only minor disadvantages compared to the Thread class because that implements the interface itself. In both cases, one has to overwrite the `run()` method. The method contains the code which should run in a separate Thread. To start a Thread, the classes `start()` method has to be excecuted[1, pp.19-22]. This is a possible source of error because it sometimes seems logical to execute the `run()` method instead. The effect might not be noticed because the code is run in both cases but only by using the `start()` method will a new Thread be created[2, 12.2.3].

Code 1: Thread creation

```java
public class MyThread extends Thread{
    public void run(){
        // my Code
    }
    public static void main (String [] args){
        MyThread t = new MyThread();
        t.start();
    }
}
```

5.2 Synchronisation of Java Threads

The interaction between Threads, while being synchronised, is interpreted as simple as the example below. We present different methods which are equipped with the special keyword `synchronized`. Also, the keyword can be used to clarify single object variables, which are accessed by the Threads. The keyword represents the beginning of a critical area, where only one single Thread is able to enter simultaneously. This measure prevents race conditions between the several Threads and permits that they work together instead of harming each other. This was already clarified in chapter 3.

5.2.1 Wait and Notify

Most important, the user has to pay attention that `wait()` and `notify()` are declared within the `synchronized` block. The method `void notifyAll()` is used to reactivate all waiting Threads. In addition it has to be implemented like `wait()` and `notify()` in the program code. Next, the

method `wait()` can be specialized by the parameters `long millisecond` and `int nanosecond` to `wait(milli, nano)`. This declares the maximal wait-time of a Thread until it starts again. When the parameters are both zero it is familiar to the regular `wait()` method without parameters. Furthermore, there is no maximum the Threads have to wait.
When `wait()` and `notify()` occur recursively in the program this can results in a Deadlock. Another disadvantage is that `wait()` and `notify()` are not static methods so the are not able to implement in non-static `synchronized` blocks easily[6, pp. 348].
To improve `synchronized` we embedded the methods `wait()` and `notify()` in our example.

Code 2: Synchronised Code[6, p.218]

```java
class Synchronization{
  synchronized void waiterMethod() {
   // Do some Stuff

   //Now we need to wait for notifier to do something
   wait();
   //Continue where we left off
  }
  synchronized void notifierMethod() {
    //Do some Stuff

    //Notify waiter that we've done it
    notify();
    //Do more things
  }
}
```

5.3 Scheduling

Similar to the priorities assigned to a task by the operating system a developer has the possibility to micromanage the priorities of the Threads he implements. To do this one has to extend the Thread class. It offers the methods `setPriority(int newPriority)` and `getPriority()` to set and get a Threads' priority. Also the constants `MIN_PRIORITY, MAX_PRIORITY` and `NORM_PRIORITY`, which are set to 1, 10 and 5 can be used[1, pp.75].

5.4 Example on the Implementation of Multithreading in Java 8

As an example for the implementation of multithreading we are going to write a program which consists of two classes. The `ThreadExampleMain` class manages the GUI[3] and starts the `ThreadExampleCounter` class as a new Thread. Furthermore, we use the `SimpleIO`[4] class to prompt the user for a command using a graphical window.
We start by implementing the `ThreadExampleCounter` class. We override the `run` method of the `Thread` class and use a while-loop to constantly increase an integer and output the value. Furthermore, we embed the `sleep` command to slow down the execution of the loop. Because

[3]Graphical User Interface

[4]SimpleIO is known from the 'Programmirung' lecture held by Prof. Dr. J. Giesl at RWTH Aachen University in 2016

of that we have to nest the while-loop in a try-block to catch the `InterruptedException` mentioned earlier, which is thrown when we try to interrupt the `sleep` method.
We start the `main` method of the `ThreadExampleMain` class by creating a new object from the `ThreadExampleCounter` class we implemented earlier and starting the new Thread. We are using a while-loop to constantly prompt the user for a command. If the command is the letter 'q' the application quits by sending an `interrupt` to the running Thread. In any other case the command is translated to an integer and a for-loop counting down from the inputted integer to zero is started. These operations again have to be nested in a try-block to catch a possible `NumberFormatException` which is thrown if the input does not consist of only numbers but also has letters in it.

Code 3: ThreadExampleMain.java

```java
public class ThreadExampleMain{
    public static void main(String[] args){
        boolean state = true;
// new Tread from the ThreadExampleCounter class
        ThreadExampleCounter tc = new ThreadExampleCounter();
// start the Thread
        tc.start();
// notice .start() not .run()
        while(state){
            int j;
// generating a SimpleIO prompt
            String s = SimpleIO.getString("Enter a command:");
            if(s.equals("q")){
// disable the while loop and send interrupt to the Thread
                state = false;
                tc.interrupt();
            }else{
// catch errors in case input does not consists of numbers
                try{
                    j= Integer.parseInt(s);
                    for(int i=j; i>0; i--){
                        System.out.println(i);
                    }
                }catch(NumberFormatException e){
                    SimpleIO.output("Wrong Number Format!","Error");
                }
            }
        }
    }
}
```

Code 4: ThreadExampleCounter.java

```java
public class ThreadExampleCounter extends Thread{
// overwritten run()-methode from the Thread class
    public void run(){
        int i = 0;
// try to catch the InterruptedException from the sleep methode
        try{
            while(true){
                System.out.println(i);
                i++;
                Thread.sleep(200);
            }
        } catch(InterruptedException e){} } }
```

6 Conclusion

To sum up, one could say that the concept of multithreading is one of, if not the most valuable, advanced programming concepts. In future, every programmer will be needing multithreading because of its diversity. It allows for better hardware utilisation and thus better performance. Furthermore, it allows better separation of application components like GUI and logic.
All in all, we are now able to produce professional code which concludes parts of multithreading and synchronisation. Also now we know how to handle different exceptions, errors, deadlocks and race conditions which can occur within multithreading.
Hopefully, the reader would have become more interested in the topic and in the world of multithreading and synchronisation. It is also wished that this term paper enriches the reader's programming skills and their understanding of the theory of multithreading in general.

References

[1] R. Oechsle, *Parallele und verteilte Anwendungen in Java: mit 156 Programmen.* Lehrbücher zur Informatik, München: Hanser, 3., erw. aufl ed., 2011. OCLC: 700067788.

[2] C. Ullenboom, *Java ist auch eine Insel: das umfassende Handbuch.* Galileo computing, Bonn: Galileo Press, 10., aktualisierte und überarb. aufl., 1. nachdr ed., 2012. OCLC: 844927873.

[3] P. Niemeyer and J. Peck, *Exploring Java.* The Java series, Bonn ; Sebastopol [Calif]: O'Reilly & Associates, 1st ed ed., 1996.

[4] A. S. Tanenbaum and H. Bos, *Modern operating systems.* Always learning, Boston: Pearson, global ed., 4. ed ed., 2015. OCLC: 892574803.

[5] G. Krüger, *Handbuch der Java-Programmierung.* München: Addison-Wesley, 3. aufl., [studentenausgabe]., [nachdr.] ed., 2004. OCLC: 249133452.

[6] T. Rauber and G. Rünger, *Parallele Programmierung.* eXamen.press, Berlin: Springer, 3. aufl ed., 2012. OCLC: 816197646.

[7] S. Oaks and H. Wong, *Java threads.* The Java series, Sebastopol, CA: O'Reilly & Associates, 1st ed ed., 1997.