Primer to Concurrent Programming

An Introductory Overview

Eric Guldbrand

August 2018

1 Introduction

When I took a course in introductory concurrent programming I found that the core concepts were hidden behind theory and it took long before I could see the complete picture. When I could however, it was much easier to both understand and value the theory. This document aims to give you that overview before you venture into more theoretical aspects.

2 Why is concurrency needed?

Modern processors has reached a limit in clock frequency and it is difficult to build a faster one without causing it to overheat. Instead we use multiple processors. That way, if we have 10 tasks and 2 processors, each can run 5 of the tasks and we will be done in the same time as if we had a processor twice as fast. In practice we don't even need multiple processors since modern CPU's usually have multiple cores and each core can have multiple threads. Each thread is capable of running tasks separately from one another without each requiring its own set of memory etc.

Unfortunately, tasks can't generally be distributed between threads at random since the result of one task may depend on the result of another, making it necessary for certain tasks to be run before others. Concurrent programming is all about resolving this issue.

A note on cores and threads: Multiple cores are quite separate from each other and how many you have depends on your CPU. Threads can be entirely virtual and may not always actually run in parallel.

3 Interleaving and atomic statements

To understand the problems that arise, you first need to understand atomic statements and interleaving.

An **atomic statement** is a piece of code that can never be halfway done. It is always "not done" until it is "done". It might for instance be a variable assignment. If a register in memory is given a new value there is no point at which it contains half the new value. Another atomic statement might be a simple addition, the calculation can never be interrupted half-way.

If (as above) assignment is atomic and addition is atomic, what about the following code?

```
a = 1 + 2
```

Indeed it is not atomic but consists of one addition, A, and one assignment to memory, B. It is conceivable that the processor performs this addition A, and stores the result in a register. It then performs some action C before it executes B. This means that action C might not have access to the result of A in memory, or reversely, that B might change the value stored in the register before it is written to memory. Doing so changes the apparent result of A.

When some statement is executed between other statements like above, we say that the statements are **interleaved**.

When multiple processes run concurrently, non-atomic statements are likely to sometimes be interleaved with other statements. If the processes are dependent on each other, which they can be if they in any way share memory, the way statements are interleaved may change the result of the program. Let's see an example of this.

In figure 1 is the code for two concurrently run processes with shared variables. In this example, every line is atomic. Note that this is different from the previous example. What an atomic statement is in reality is not pre-defined and depends on which programming language or model you are using.

```
1  a = 1
2  a = a + a
```

Process A

```
1  b = 3
2  a = b
```

Process B

Figure 1: *Two processes running concurrently with the same variables. The final value of* `a` *depends on circumstances outside of our control.*

What will the final value of variable `a` be? The answer is 2, 3 or 6, but we don't know which one. The result depends entirely on in which order the statements happen to be executed. If the order is [A1, A2, B1, B2] we get 3, but if it is [A1, B1, B2, A2] the result will be 6. Which ordering of statements we get is outside our control like processor scheduling or electromagnetic fluctuations. Because of this we will see different results if we run the program multiple times.

4 The critical section problem

The **critical section problem** is the problem where multiple processes require access to the same resource but the resource only allows one at a time. Imagine two trains running on parallel tracks like in figure 2. Just before a bridge the two tracks merge, and split again on the other side. Here the bridge is the critical section; one train must wait for the other to pass first.

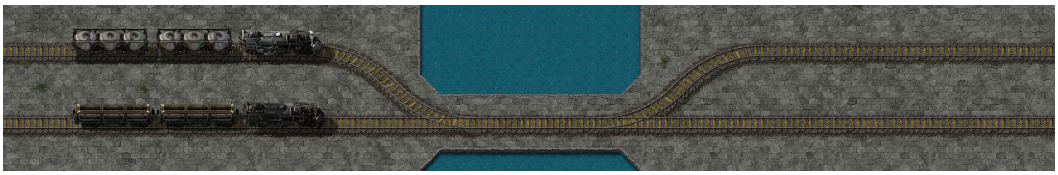

Figure 2: *Two trains approaching a critical section of the railway. One train has to wait, or they will crash.*

In reality, this problem often occur when processes have to both read from and write to variables.

Example:

> Process A wants to perform $a = a+1$. Process B wants to perform $a = a+2$. To do this, both processes first need to read the current value of a, perform an addition, and then write back the new value to memory.
>
> Due to interleaving, these operations can happen in multiple ways, resulting in the final value of a being increased by either 1, 2 or 3.

5 How to solve the critical section problem

Simply put, the critical section problem is solved by making sure only one process is in it at any one time, and that every process trying to enter it is able to do so at some point. In more formal terms the problem is solved if there is:

1. **Mutual Exclusion**: Critical sections are never entered by multiple processes simultaneously.
2. **Freedom from deadlock**: If *some* processes tries to enter the critical section, one of them will eventually succeed. Read about deadlocks in section 6.1.
3. **Freedom from starvation** If *one* process tries to enter its critical section, it will eventually succeed. Read about starvation in section 6.2.

Mutual exclusion can be ensured through the use of locks, semaphores and other control structures. Read more about those in this section. Unfortunately, the use of locks and semaphores may cause deadlock and starvation, which you can read more about in section 6.

5.1 Locks and blocks

A **lock** is a structure that limits the use of a resource to a single process at a time. When a process requires access the resource, it will request to lock the resource. If the resource is available, the process can use it. If the resource is already locked by another process, the first process will **block** on that resource. This means that the process will pause execution until it has acquired the lock on the resource. As soon as a process is done with a resource, it should unlock it to make it available for others again.

Be aware there is much theory concerning blocking and what to do when multiple processes block on the same thread.

In Java, locks are represented by the `Lock` interface with methods `lock()` and `unlock()` for acquiring and releasing the lock. One implementation of this interface is `ReentrantLock`.

5.2 Semaphores

A **semaphore** is very similar to a lock but can grant multiple accesses (permits) to one resource. If a lock is a table with one chair (only one process can sit at a time) a 2-semaphore is a table with two chairs, a 3-semaphore a table with three chairs etc. Thus the semaphore can lock several times before a process will have to block. A lock is the same as a 1-semaphore.

In Java, semaphores are represented by the `Semaphore` object with methods `acquire(int permits)` and `release(int permits)` for acquiring and releasing a certain number of permits.

5.3 Monitors

A **monitor** is an advanced control structure that enforces mutual exclusion on an entire object or group of objects. It does this by implementing methods that automatically lock and unlock resources as they are requested. This means that a process can never forget to unlock a resource after it is done with it, which it might do when using regular locks.

In Java, you either need to implement the monitor yourself, or use the `synchronized` keyword. In figure 3 are two implementations of a simple monitor. These are functionally the same and showcases what `synchronized` does. As can be seen, the monitor is just an object with private fields representing the resources that should be synchronized (that is, be under mutual exclusion). Finally, getters and setters provide access to the resources while operating the locks or semaphores.

```
1  public class MyMonitor {
2    private int a;
3    private Lock l;
4
5    public MyMonitor(){
6      a = 0;
7      l = new ReentrantLock();
8    }
9
10   public int getA(){
11     try {
12       l.lock();
13       return a;
14     } finally {
15       l.unlock();
16     }
17   }
18
19   public void setA(int a){
20     try {
21       l.lock();
22       this.a = a;
23     } finally {
24       l.unlock();
25     }
26   }
27 }
```

```
1  public class MyMonitor {
2    private int a;
3
4    public MyMonitor(){
5      a = 0;
6    }
7
8    public synchronized int getA(){
9      return a;
10   }
11
12   public synchronized void setA(int a){
13     this.a = a;
14   }
15 }
```

Figure 3: *Two implementations of a simple monitor in Java. These implementations have the same function and gives a good idea of what the* `synchronized` *keyword does.*

5.4 Channels

A channel is a high-level structure that allows processes to send messages to each other. Channels use other control structures beneath the surface to make sure that these messages play nice with concurrency, but using them is often quite simple. They are mentioned here just to let you know they exist.

6 Problems created by concurrency

Unfortunately, locks, semaphores and other blocking constructs create new problems. Your program may deadlock, or a part of it might starve. This section should give you a basic idea of what these problems are.

6.1 Deadlock

A deadlock is when two processes are waiting for each other indefinitely, each requiring a resource the other has before it can release the resource the other needs. One way to avoid deadlock situations is to lock all your resources at once. However, doing that may instead lead to starvation.

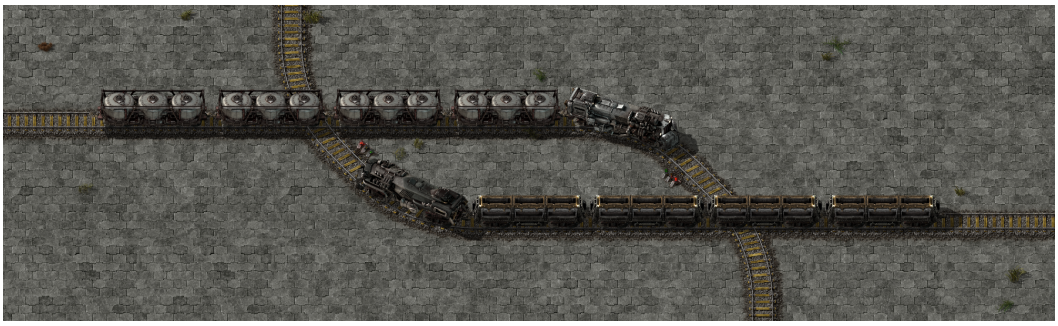

Figure 4: *Two trains in deadlock. Train signals acting as locks on each crossing tell the trains to wait because the crossing is full. A better approach would be to treat the entire intersection as a single critical section.*

In figure 4 two trains are deadlocked. Train signals guard the crossings to stop trains from running into each other, acting much like locks would. The problem is that a train need to pass both crossings before it can let another train pass. The right thing to do would be to consider the entire intersection as the critical section and lock that instead. Thus only one train is allowed in the intersection at any one time. See figure 5.

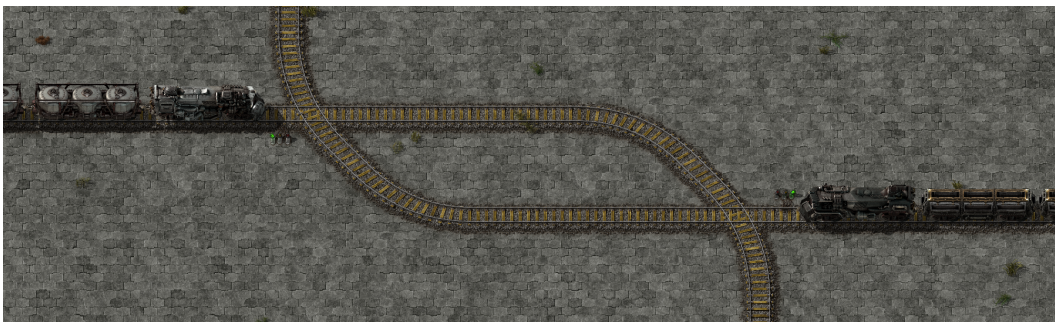

Figure 5: *The same intersection as in figure 4 but with the entire intersection treated as the critical section. In this scenario, either train can go first.*

In general, deadlocks can be avoided by thoroughly considering the system at hand, but small mistakes can still lead to deadlock. What is really needed is experience and an understanding of further theory not presented in this document.

6.2 Starvation

Starvation is when some process is never able to acquire the resources it needs. This may be more likely to happen when one ”heavy” process shares resource with a ”light” one.

Example:

> Process A needs to acquire locks on resource 1, 2 and 3 before it can run. Process B need to acquire resource 2. Process C need to acquire resource 3. As it happens, B and C are run quite often which results in that resource 2 or 3 is always locked. Because A needs to acquire all its resources at once (or it might risk deadlock), A can never begin to acquire resources and is never run. In this case, process A is starved.

In figure 6 we see a starved train system. The black train will never be able to reserve passage because it is waiting to acquire the lock on all three crossings simultaneously, but at least one crossing is always busy. The black train could try to reserve one crossing at a time as they become available, but as we saw in section 6.1, such behaviour might lead to deadlock.

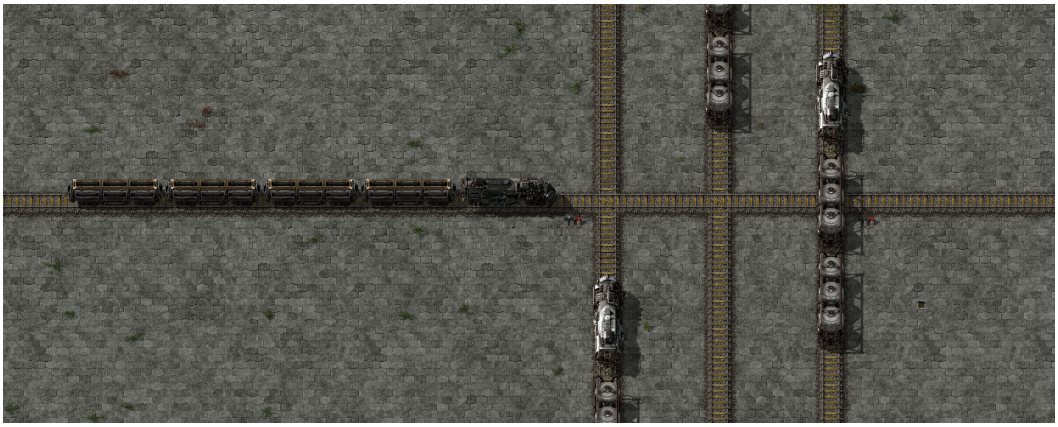

Figure 6: *A starved train system. The white trains only need one resource each and can easily acquire them. The black train need to acquire all three resources at the same time, but there may be no time when that is possible.*

Avoiding starvation can be difficult. How to avoid it will be the subject of much theory you will encounter in a full course on concurrent programming.

7 Code example in Java

In Java, every process is called a "thread". You can start a thread using a `Thread` object. To program each thread you can either subclass `Thread` and override the `run()` method, or instantiate the thread with an object implementing the `Runnable` interface. Which way you choose doesn't really matter. In figure 7 is a simple program that prints a line of text from a new thread using the runnable approach. If you search online for *how to start a thread in java* you can find more examples.

```
1  public class Main {
2
3    public static void main(String[] args) {
4      Runnable r = new Runnable() {
5        @Override
6        public void run() {
7          System.out.println("I'm not the main thread!");
8        }
9      };
10
11     Thread t = new Thread(r);
12     t.start();
13     System.out.println("I am the main thread :D");
14   }
15 }
```

Figure 7: *A program that starts a new thread using a runnable. One line of text is printed from the thread, and another line is printed from the main thread.*