# PCPP Assignment 1

Group name: Thread Heresy
Real names : Nedas Surkus, Niclas Abelsen,
Github username: nesu, niab

September 2022

## Contents

# Exercise 1

## 1

We get inconsistent values below 20 million but above 19 million when executing the code several times. Examples of 3 results is:

1. `Count is 19910614 and should be 20000000`
2. `Count is 19492775 and should be 20000000`
3. `Count is 19889471 and should be 20000000`

## 2

Code is more likely to be 200. But after several reruns. The answer is around 190 and goes back to 200. This is not stable answer. Thus we cannot guarantee that the output will be 200.

## 3

Refactoring the `count = count + 1` to either `count += 1` or `count++` will have the same outcome as IT is a similar operation but put under abstraction. Thus count++ still operates on : Take a temp value. Assign current count to temp. And then do the addition and return result. Same for count +=1
This can be proven by showing it is translated to the same intermediate code. We create an `increment` for each count method respectively and exports the java bytecode which can be seen in figure 1. The result shows that every `increment` function increments the counter the same way. Thus, the result of the *increment* is not affected by the suggested changes. Note that the additional 'alod_0' instruction for `increment()` happens because of `count = count + 1`. This can be proven by looking at the java bytecode.

```
 public void increment();
   Code:
      0: aload_0
      1: aload_0
      2: getfield      #2                  // Field count:J
      5: lconst_1
      6: ladd
      7: putfield      #2                  // Field count:J
     10: return

  public void increment2();
   Code:
      0: aload_0
      1: dup
      2: getfield      #2                  // Field count:J
      5: lconst_1
      6: ladd
      7: putfield      #2                  // Field count:J
     10: return

  public void increment3();
   Code:
      0: aload_0
      1: dup
      2: getfield      #2                  // Field count:J
      5: lconst_1
      6: ladd
      7: putfield      #2                  // Field count:J
     10: return

```

Figure 1: Bytecode for the three versions of the method

## 4

We identified that the critical section is the addition and retrieving the `count` that is being called in `LongCounter`. As the variable of count is a mutable variable. Thus we must make the object thread-safe so no two threads can access the variable at the same time in order to avoid thread overriding count that is being increased and retrieving stale data.

Despite the requirement to use ReetranLock. We will use ReetranReadWriteLock. Which is ReetranLock with some more functionality, line 2. For making the `increment()` thread safe we use `writeLock`, line 9 - 14, which makes the `coun++` mutual exclusive. For making the `get()` method thread safe we use `readLock`, line 18. Which allows the code to be read by multiple threads only if there is no writelock currently active. And thus after implementing the locks we make the object thread-safe and now only 1 thread at a time can perform a write (increment) operation and any other thread that wants to get data must wait until all write operations have executed before having access to get method.

We also define 1 lock for 1 variable, line 3 - 4. Which is good code practice.

```
 1  class LongCounter {
 2    ReadWriteLock lock = new ReentrantReadWriteLock();
 3    Lock readLock = lock.readLock();
 4    Lock writeLock = lock.writeLock();
 5
 6    private long count = 0;
 7
 8    public void increment() {
 9      writeLock.lock();
10      try {
11        count++;
12      } finally {
13        writeLock.unlock();
14      }
15    }
16
17    public long get() {
18      readLock.lock();
19      try {
20        return count;
21      } finally {
22        readLock.unlock();
23      }
24    }
```

Figure 2:

## 5

Yes we only include the part the critical section which contains 1 line of code.

## Exercise 1.2

### 1

The code for the program can be found in `week01exercicses/app/src/main/PrinterExercise.java` and seen in the code below.

```java
class PrinterExercise {
    Printer printer;

    public PrinterExercise(){
        printer = new Printer();

        Thread t1 = new Thread(() -> {
                while (true){
                    printer.print();
                }
        });

        Thread t2 = new Thread(() -> {
                while (true){
                    printer.print();
                }
        });

        t1.start(); t2.start();
        try { t1.join(); t2.join(); }
        catch (InterruptedException exn) {
            System.out.println("Some thread was interrupted");
        }
    }


    public static void main(String[] args) {
  new PrinterExercise();
    }

    class Printer {

        public void print() {
      writeLock.lock();
      try {
        System.out.print("-");
        try {
          Thread.sleep(50);
        } catch (InterruptedException exn) {
          System.out.println(exn);
        }
        System.out.print("|");
      } finally {
        writeLock.unlock();
      }
    }
    }
}
```

### 2

When we execute threads we do actions in concurrency. Meaning each statement is executed at a similar time in each thread. When statement executes there are possible that each thread will execute each operation at different time or at the same time which can cause errors when doing operations. Its not unique for threads but any concurrent program.

There are many types of interleaving that might happen in this program but we will show a single example that can be seen in figure. `t1`(Thread 1) and `t2` (Thread 2) will execute the print method. `t1` calls "System.out.print("-")" then it calls "Thread.sleep(50)". While `t1` is waiting `t2` starts at a slightly delayed time and calls "System.out.print("-")" and then continues to execute the method.

Thus we can see that in the first iteration there is a mistake happening. The iteration will print out two "–" thus ruining the "-|" flow. In that sense we can conclude that interleaving is happening. However it should be noted that its not the only place it can happen. It can happen in 3 places. 3.

```
Iteration 1 (interleaving happens) :
    t1(System.out.print("-"),
    t2(System.out.print("-"),

    t1(... Thread.sleep(50)...)
    t2(... Thread.sleep(50)...)

    t1(System.out.print("|");)
    t2(System.out.print("|");)
Iteration 2 (The correct iteration):
    t1(System.out.print("-"),
    t1(... Thread.sleep(50)...)
    t1(System.out.print("|");)

    t2(System.out.print("-"),
    t2(... Thread.sleep(50)...)
    t2(System.out.print("|");)

```

## 3

The critical part of the `Printer` class the section where the strings are printed out as only one thread is allowed to access this section to avoid duplicate string values. The thread safe version of printer can be viewed in figure 3.
In terms of happens-before, we see that a lock is created before entering the critical section, line 4 and we have the unlock is called before the lock, line 9, for the second thread.

```
1   class Printer {
2        private ReentrantLock lock;
3        public void print() {
4            lock.lock();
5            try {
6                System.out.print("–");
7                System.out.print("|");
8            } finally {
9                lock.unlock();
10           }
11       }
12   }
```

## Exercise 1.3

### 1 & 2

The Turnstile object has been modified to include an if-command that checks whether the counter has reached the MAX_PEOPLE_COVID.

A lock is surrounding the entire inner part of the for-loop as it is the critical section. We do not wish the counter to be written to as we increase the value nor do we want threads to increase the value at the same time in case the counter is one value below the MAX_PEOPLE_COVID. The changes can be seen in figure 3

```
[...]
public void run() {
    for (int i = 0; i < PEOPLE; i++) {
        lock.lock();
        try {
            if (counter == MAX_PEOPLE_COVID){
                break;
            }

            counter++;
        } finally {
            lock.unlock();
        }
    }
}
```

Figure 3:

## Exercise 1.4

### 1

It is impossible to find systems that fit Goetz model but do not fit the concurrency model. The are several reasons but the main one that the Goetz definition was made before concurrent systems were prevalent as it is now. Many systems if not all of them are currently fitting into the concurrency categories rather than Goetz as concurrency is very prevalent in the modern age. However, the same cannot be said for vice-versa.

When discussing convenience and resource utilization we can make an argument that some systems will not utilize these definitions. Computer hardware resources are not as limited today as it was before and it can be beneficial to wait for an input and output. Especially when we take something specialized as servers or super computers into account. Who solo job is to run a single program that waits for a call or input. But yet can still have a kind of interface. They can exploit the hardware to run calculations at the same time and are hidden from each other.

Another one is convenience. Modern day programs each can perform multiple tasks such as loading an animation and sending out an HTTP call to the server and waiting for a response. Or simply loading the resource in the background while the user scrolls down on a website. Thus it is now not easier to write multiple programs to do a singular task. If a separate program would open that will be responsible for playing a loading screen and another one downloaded data. The user experience will degrade and will break the illusion of a 'seamlessly working application'. Thus most modern applications no longer have the desire to perform only a single task but rather perform a suite of tasks that can be executed at the same time utilizing different threads.

### 2

Online multiplayer games. There game controls (inputs/outputs). It executes multiple streams of statements and communicates with a server without slowing down the interface (Exploitation). And is hidden. It allows you to have a browser open and play at the same time.

Watching a YouTube video. YouTube has an user interfaces (comments / search bars). It coordinates video download from a server to display on your screen and loads the video at the same time as playing it. The hidden aspect is that the video playing has 2 processes running at the same time that download 1 resource and each acts independently of another. It loads the video and plays it.

Communicating through chat software (such as Discord). It has user interfaces where you type messages. Exploitation is capable of sending data to the server and displaying a loading message while its being transmitted. The hidden part is that each program has a resource from the server but act as individual software.

## Exercise 2.1

### 1

The implementation of the monitor can be seen in figure 4

```
[..]
public class ReadWriteMonitor {
    private int readers;
    private boolean hasWriter;
    private Object lock;

    public ReadWriteMonitor(){
        readers = 0;
        hasWriter = false;
        lock = new Object();
    }

    public void readLock(){
        synchronized(lock) {
            try {
                while(hasWriter){
                    lock.wait();
                }
                readers++;
            } catch(InterruptedException e) {}
        }
    }

    public void readUnlock(){
        synchronized(lock) {
            readers--;
            if (readers == 0){
                lock.notifyAll();
            }
        }
    }

    public void writeLock(){
        synchronized(lock){
            while(hasWriter)
                try {
                    lock.wait();
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            hasWriter=true;
            while(readers != 0)
                try {
                    lock.wait();
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
            }
        }
    }

    public void writeUnlock(){
        synchronized(lock) {
            hasWriter = false;
            lock.notifyAll();
        }
    }
}
```

Figure 4:

### 2

The solution is fair to towards the writer threads. As soon as a thread tries to write, the boolean `hasWriter` is sat to true, line 41 in figure4 and lock, line 42 - 46. When a thread tries to read as another thread writes, it will wait until there is no writer, line 15 - 18. As there is no priority queue, it will ensure that the readers will not starve making this implementation have a strong fairness property .

## Exercise 2.2

### 1

Yes we can observe that the main threads write will remain invisible thus it can loop forever.

It can loop forever as explained in Brian Goetz book (Chapter 3. Page 34). The modified value might never become visible to the thread. However in this specific scenario there is very little risk it will print out an incorrect value due to reordering. As the print value operation relies on the objects internal methods.

### 2

The initial implementation of `MutableInteger` has a risk of having stale data. This happens when the reader will attempt to use the get method to access data and will access out of date value. This happens because integer `value` is not thread safe as both the main thread and the other thread can access and modify information at the same time. In order to avoid this, we must ensure that the "value" is thread safe.

The thread safe implementation can be found in figure 7. By using `GuardedBy(‘‘this’’)` We ensure that only a single lock will be used to guard the value and utilizing the `syncronise` keyword we can ensure that the `value` is accessed by only one thread at a time.

`synchronized` keyword functions as a monitor in the internal java code and utilizes locking to ensure that the object /keyword / method can only be accessed by a single thread at a time.

Due to the synchronized keyword, we can ensure that the thread will have access to the updated data and prevent the thread from running infinitely.

```
class MutableInteger {
    @GuardedBy("this") private int value = 0;
    public synchronized void set(int value) { this.value = value; }
    public synchronized int get() {return value;}
}
```

Figure 5:

### 3

There is a chance that threat `t` will not terminate if get is not defined as `synchronized` as thread `t` can still be able to see stale values. In this case the stale value can be `0` which would prevent threat `t` from exiting and thus running forever.

### 4

```java
class MutableInteger {
    private volatile int value = 0;
    public void set(int value) { this.value = value; }
    public int get() {return value;}
}
```

Figure 6:

Yes. Utilizing volatile keyword ensures that value is visible to the program.

This is due to the fact that when a field is declared as a volatile, the compiler and runtime are put on notice and that this variable is shared and that operations related to this variable should not be reorder with other memory operations (Book. Chapter 3.14. Page 38). They are not cached in registers or caches and thus are not hidden from other processors. However, utilizing volatile is a weaker form of synchronization as they cannot make certain operations atomic unless the programmer can guarantee that variable is only written from a single thread. Thus it is preferred to use synchronized or locking to ensure both visibility and atomicity.

## Exercise 2.3

### 1

We get inconsistent values below 2 million when executing the code several times. This implies that race conditions happens in the code.

Examples of 3 results is:

1. `Sum is 1936427.000000 and should be 2000000.000000`
2. `Sum is 1878734.000000 and should be 2000000.000000`
3. `Sum is 1896157.000000 and should be 2000000.000000`

### 2

Race conditions appear due to the different kind of locks and permissions utilized for static synchronization and normal synchronization. When utilizing normal synchronization, it utilizes object-level lock. However, when using static synchronization , java implements a class wide lock which also counts as a different type of lock when compared to the object − level lock. This means that `t1` and `t2` can access the object "Sum" parallel and cause race conditions by utilizing stale data. Example: `t2` requests "AddStatic" method. Which locks the whole class using a class level lock. If some kind of thread 3 would want to utilize "AddStatic" or "sum" method. It would notice the static keyword and would have to wait until Thread 2 has completed its execution of AddStatic method before it can attempt to use the methods due to class- level lock. But `t1` requests "AddInstance" method. This method only has object level lock. Thus, when `t1` Requests access it gets permission to modify "Sum" keyword as the class is locked behind a class-level lock and not an object-lock. Thus `t1` and `t2` have access to stale data which causes a race condition and does modify the correct result.

## 3

This new implementation implements a write lock and a read lock.

When modifying the critical section (sum+=x) we firstly lock it using a write lock. This specific lock will allow only a single writer to write into the critical section. Thus by implementing the same lock for both addStatic and addInstance we prevent multiple threads from modifying sum in parallel and thus avoiding stale data. By implementing read block we also prevent the application from getting stale data and reading while writers are still active. Thus only 1 thread at a time can access the critical sections and this prevents any race conditions from occurring.

```java
class Mystery {
    private static double sum = 0;
  private static ReadWriteLock lock;
  private static Lock writeLock;
  private static Lock readLock;

    public Mystery(){
        lock = new ReentrantReadWriteLock();
        writeLock = lock.writeLock();
        readLock = lock.readLock();
    }

    public static synchronized void addStatic(double x) {
        writeLock.lock();
        try{
            sum += x;
        } finally {
            writeLock.unlock();
        }
    }

    public synchronized void addInstance(double x) {
        writeLock.lock();
        try {
            sum += x;
        } finally {
            writeLock.unlock();
        }
    }

    public static synchronized double sum() {
        readLock.lock();
        try{
            return sum;
        } finally {
            readLock.unlock();
        }
    }
}
```

Figure 7: