Assigment 1

Bjørn Christian Weinbach

September 7, 2021

1 Concurrency and Non-Determinism

The code provided on studium is shown below

```cpp
#include <iostream>
#include <mutex>
#include <thread>


std::mutex mutex;

void loop(int n)
{
  mutex.lock();
  std::cout << "Task " << n << " is running." << std::endl;
  mutex.unlock();

  mutex.lock();
  std::cout << "Task " << n << " is terminating." << std::endl;
  mutex.unlock();
}

int main()
{
  std::thread t1(loop, 1);
  std::thread t2(loop, 2);
  std::thread t3(loop, 3);
  std::thread t4(loop, 4);
  std::thread t5(loop, 5);
  std::thread t6(loop, 6);
  std::thread t7(loop, 7);
  std::thread t8(loop, 8);

  t1.join();
  t2.join();
  t3.join();
  t4.join();
  t5.join();
  t6.join();
  t7.join();
  t8.join();

  return 0;
}
```

Listing 1: non-determinism.cpp

And when we compile this program as told in the exercise, we get an output similar to this:

```
Task 1 is running.
Task 3 is running.
Task 3 is terminating.
Task 1 is terminating.
Task 4 is running.
Task 5 is running.
Task 6 is running.
Task 7 is running.
Task 7 is terminating.
Task 5 is terminating.
Task 4 is terminating.
Task 6 is terminating.
Task 2 is running.
Task 2 is terminating.
Task 8 is running.
Task 8 is terminating.
```

But exactly what tasks starts runnning and terminating in what order is observed to be stochastic. This is known as **non-determinism** that is known to arise when we have concurrent programs. In the observed output which task that will start to run and in what order is random. The non-determinism of concurrent programs is important to take into account as this could lead to the final output also becoming non-deterministic. Therefore we need mutual exclusion flags to ensure that the concurrent program outputs the correct solution for the problem that it is solving. In the program above this mutual exlustion flag is raised when outputting that a task is running and when it is terminating. Mutual exlusion flags solves some problems but will also lead to other problems which we will see later.

2 Shared-Memory Concurrency

We have, as stated in the assignment desctiption, three threads that operate on a shared integer variable named **x**

The first thread **t1** operates the following code

```
void inc()
{
  bool r = true;
  while (r)
    {
      mutex.lock();
      ++x;
      r = run;
      mutex.unlock();
    }
}
```

Listing 2: Thread 1 source code

This thread runs an 'infinite' loop on the boolean variable r which is true until the boolean value r is equal to *run*, which we will see later in the main function. Thread 1 raises it mutual exclusion flag before it increments x and assigns *run* to r before it lowers its flag.

The second thread **t2** operates the following code

```
void dec()
{
  bool r = true;
  while (r)
    {
      mutex.lock();
      --x;
      r = run;
      mutex.unlock();
    }
}
```

Listing 3: Thread 2 source code

This thread does the same thing as *t1* but opposite. It loops infinitely and raises it flag when decrementing x and assigning *run* to r.

The last thread **t3** operates the following code

```
void print()
{
  bool r = true;
  while (r)
    {
      mutex.lock();
      std::cout << x << std::endl;
      r = run;
      mutex.unlock();
    }
}
```

Listing 4: Thread 3 source code

This thread also loops infinitely on r until that gets assigned false by *run*. It raises its mutual exclusion flag before outputting the value of x and when assigning *run* to r.

Lastly, lets see the main method, how it uses these threads and what we would expect as output.

```
int main()
{
  std::thread t1(inc);
  std::thread t2(dec);
  std::thread t3(print);

  std::this_thread::sleep_for(std::chrono::seconds(1));

  mutex.lock();
  run = false;
  mutex.unlock();

  t1.join();
  t2.join();
  t3.join();

  return 0;
}
```

Listing 5: Main method

We see that the threads are started and the program sleeps for 1 second. In that time, all three threads are operating on the same shared variable. Given Given how the mutual exclusion flags are handled. When one thread loops another thread may be given access to the variable. Since the order of what threads will get access and for what time during the second of sleep is non-deterministic, the variable x should also have a non-deterministic value when outputted.

After five trial runs, the last outputted value of x was: 57031, -44762, -45637, 220150 and 11428. So we see that the final value of x can be considered stochastic and non-deterministic.