# Assigment 1

Bjørn Christian Weinbach  Thameez Bodhanya

September 7, 2021

## 1 Concurrency and Non-Determinism

The code provided on studium is shown below

```
#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();

```

```
39   return 0;
40 }
```

Listing 1: non-determinism.cpp

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

```
1  Task 1 is running.
2  Task 3 is running.
3  Task 3 is terminating.
4  Task 1 is terminating.
5  Task 4 is running.
6  Task 5 is running.
7  Task 6 is running.
8  Task 7 is running.
9  Task 7 is terminating.
10 Task 5 is terminating.
11 Task 4 is terminating.
12 Task 6 is terminating.
13 Task 2 is running.
14 Task 2 is terminating.
15 Task 8 is running.
16 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

```
1  void inc()
2  {
3    bool r = true;
4    while (r)
5      {
6        mutex.lock();
7        ++x;
8        r = run;
9        mutex.unlock();
10     }
```

```
}
```

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();
```

```
10    run = false;
11    mutex.unlock();
12
13    t1.join();
14    t2.join();
15    t3.join();
16
17    return 0;
18 }
```

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.

# 3 Race Conditions and Data Races

A race condition occurs when a program depends on the timing of one or more processes to function correctly. A data race is a special case of race condition where a shared variable is being operated on by several threads. The first program *non-determinism.cpp* could potentially produce a race condition and the second program *shared-variable.cpp* could produce a data race.

This depends on the desired operation of the above mentioned programs. The first thread reports which task is running and terminating in what order. Since the print statements are in mutual exclusive sections a error in the print statement will not happen. Because of that i do not consider *non-determinism.cpp* to produce a race condition. If these mutual exlusive sections were not implemented, another thread could take over before in the time that the print is finished and a race condition is produced.

The second program, *shared-variable.cpp* does produce different outputs dependent on the sequence of execution on the threads but i consider that to be a part of the operation of the program to demonstrate how the access of the threads is random for each time the program is ran. The operations on the shared variable are atomic and will not produce errors and invalid output that does not reflect the behaviour of the program. I therefore do not consider *shared-variable.cpp* to produce a data race.

# 4 Multicore Architectures

## 4.1 Explore the CPU

```
-bash-4.1$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                32
On-line CPU(s) list:   0-31
Thread(s) per core:    2
Core(s) per socket:    8
Socket(s):             2
NUMA node(s):          4
Vendor ID:             AuthenticAMD
CPU family:            21
Model:                 1
Model name:            AMD Opteron(TM) Processor 6274
Stepping:              2
CPU MHz:               1400.000
BogoMIPS:              4399.38
Virtualization:        AMD-V
L1d cache:             16K
L1i cache:             64K
L2 cache:              2048K
L3 cache:              6144K
NUMA node0 CPU(s):     0-7
NUMA node1 CPU(s):     8-15
NUMA node2 CPU(s):     16-23
NUMA node3 CPU(s):     24-31
```

Listing 6: Output from lscpu command

We see that 32 logical CPU cores and this is shared across 2 physical CPUs with 8 cores each which are dual threaded. Resulting in $2 \times 8 \times 2 = 32$.

We see that this processor has a lot more physical cores. In summary we have:

- There are in total 2 physical CPUs. Each with 8 physical cores.
- Each physical core has two logical CPUs.
- Each physical core has an L1d, L1i and L2 cache of sizes 16, 64, AND 2048KB respectively.
- Four physical cores share one L3 cache of size 6144KB.

We see that the caches are somewhat similar. The L2 cache fo each physical core have more memory compared to the Intel I5-450M processors. The L3 cache is also larger but share 4 physical cores instead of 2. The main difference is that there are a lot more physical and logical cores, which will allow for concurrent multithreaded programming.

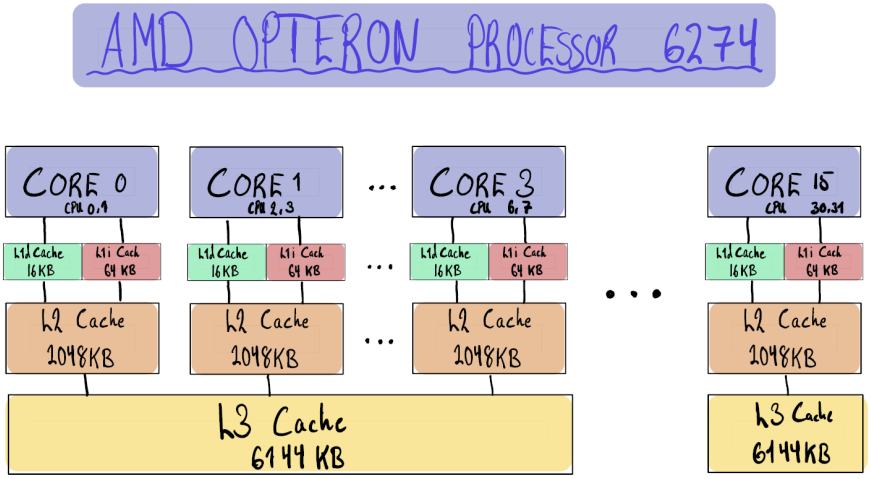


Figure 1: Cache Hierarchy of AMD Opteron Processor 6274 om gullviva.it.uu.se