Assignment 1

Bjørn Christian Weinbach Thameez Bodhanya

September 9, 2021

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. 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. Due to the problems discussed above, the exact output is undefined and we could see other orders printed out.

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();
```

```
  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.

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* does have a race condition which gives different outputs based on the order of the threads. No data race exists though as there is no shared variable that the threads operate on.

The second program, *shared-variable.cpp* does produce different outputs dependent on the sequence of execution on the threads and we therefore have a race condition. The operations on the shared variable are atomic and therefore the program does not have a data race on the shared variable. This has been verified by compling the program with 'fsanitize=thread' and running it.

In summary: Both programs are examples of race conditions, none of them are examples of data races due to atomic operations on shared data.

Multicore Architectures

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

Based on the output above it seems that the CPU has dualthreaded cores and 8 cores per cpu. With our two physical cpus this would make 16 cores and 32 virtual CPUs. But this is an AMD processor and the architecture is a bit different than it first seems. According to cpu-world.com the CPU actually has 16 physical cores. It has L1 data caches for each core, and shared instruction caches for every two cores. The L2 cache is shared between two cores as well. L3 Cache is shared among 8 cores. See figure 1 for a visual illustration of a single AMD Opetron 6274.

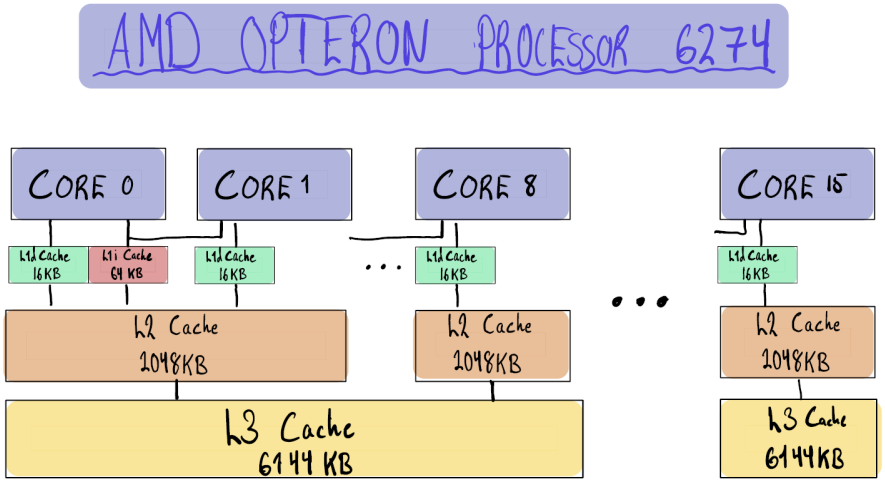

Figure 1: Cache Hierarchy of AMD Opteron Processor 6274 on gullviva.it.uu.se

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

- There are in total 2 physical CPUs. Each with 16 physical cores.
- Each physical core has an L1d of 16KB
- There is an L1i Cache of 64KB for every two cores.
- There is a L2 Cache of 2048KB for every two cores.
- Eight physical cores share one L3 cache of size 6144KB.

Compared with the Intel cpu the cache levels are the same. The memory split is what is the main different between the two cpus. There are more physical cores in the AMD cpu but the instruction memory is shared between the cores. The CPU we are using also has a lot more cores than the intel CPU.

Performance Measurements

Script used to run program is shown in listing 7.

```
#!/bin/bash

set +e
rm results.txt

set -eo pipefail

for t in {1,2,4,8,16,32}; do
    for n in {1,2,4,8,16,32}; do
    echo "($t,$n)" >> results.text
    ./performance $t $n >> results.txt
    done
done
```

Listing 7: Bash script to try all values of T and N

Runtimes

```
(1,1)
Finished in 1.21908 seconds (wall clock).
(1,2)
Finished in 2.43608 seconds (wall clock).
(1,4)
Finished in 4.90978 seconds (wall clock).
(1,8)
Finished in 9.84641 seconds (wall clock).
(1,16)
Finished in 19.692 seconds (wall clock).
(1,32)
Finished in 39.3932 seconds (wall clock).
(2,1)
```

```
Finished in 0.616555 seconds (wall clock).
(2,2)
Finished in 1.2252 seconds (wall clock).
(2,4)
Finished in 2.45917 seconds (wall clock).
(2,8)
Finished in 4.94952 seconds (wall clock).
(2,16)
Finished in 9.88132 seconds (wall clock).
(2,32)
Finished in 19.7601 seconds (wall clock).
(4,1)
Finished in 0.309365 seconds (wall clock).
(4,2)
Finished in 0.616561 seconds (wall clock).
(4,4)
Finished in 1.22603 seconds (wall clock).
(4,8)
Finished in 2.46182 seconds (wall clock).
(4,16)
Finished in 4.97059 seconds (wall clock).
(4,32)
Finished in 9.9639 seconds (wall clock).
(8,1)
Finished in 0.161366 seconds (wall clock).
(8,2)
Finished in 0.316883 seconds (wall clock).
(8,4)
Finished in 0.63344 seconds (wall clock).
(8,8)
Finished in 1.24348 seconds (wall clock).
(8,16)
Finished in 2.49317 seconds (wall clock).
(8,32)
Finished in 5.01298 seconds (wall clock).
(16,1)
Finished in 0.083777 seconds (wall clock).
(16,2)
Finished in 0.161929 seconds (wall clock).
(16,4)
Finished in 0.319613 seconds (wall clock).
(16,8)
Finished in 0.635709 seconds (wall clock).
(16,16)
Finished in 1.25654 seconds (wall clock).
(16,32)
Finished in 2.52042 seconds (wall clock).
(32,1)
Finished in 0.040681 seconds (wall clock).
(32,2)
Finished in 0.075294 seconds (wall clock).
(32,4)
Finished in 0.151482 seconds (wall clock).
(32,8)
Finished in 0.298942 seconds (wall clock).
(32,16)
Finished in 0.592317 seconds (wall clock).
```

```
(32,32)
Finished in 1.18443 seconds (wall clock).
```

Listing 8: Reported runtimes for all values of T and N

Plotted graph

The plotted times on threads, size of array and its affect on time is shown In figure 2

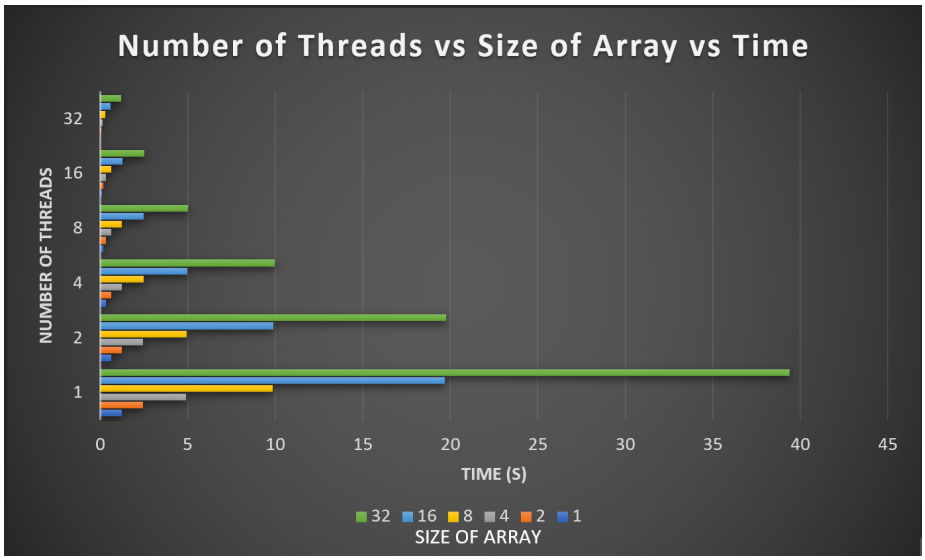

Figure 2: Threads and array size vs time.

Discussion and explanation

A quick look at the graph shows us that as expected as we increase the size of the array (while keeping thread size constant), the time taken to run the program increases almost linearly (it takes twice the amount of time to run the program for array size n2). This pattern continues no matter how many threads we add to the program (and I presume represents the work that cannot be paralleled). Once we start varying the amount of threads, things get interesting, as we see an almost inverse relationship to above. For every thread we add, we see the time taken halved (it takes 0.5 the amount of time of array size n with threads 2t). And this is a great demonstration of the speed parallelizing an application can yield, as the amount of time required to complete array size 32 with 32 threads is lower than the amount of time taken to run the application for array size one on a single thread.

Dining Philosophers

Dining philosophers is a known example problem in computer science formulated by Edsger Dijkstra in 1965 to illustrate a problem that can arise from concurrent programs. In the assignment there has been provided a program which simulates the dining philosophers problem. After compilation and a few trial runs we get a output similar to this which halts:

```
Philosopher 0 is thinking.
Philosopher 0 picked up her left fork.
Philosopher 0 picked up her right fork.
Philosopher 0 is eating.
Philosopher 0 is putting down her right fork.
Philosopher 0 is putting down her left fork.
Philosopher 0 is thinking.
Philosopher 0 picked up her left fork.
Philosopher 0 picked up her right fork.
Philosopher 0 is eating.
Philosopher 0 is putting down her right fork.
Philosopher 0 is putting down her left fork.
Philosopher 0 is thinking.
Philosopher 0 picked up her left fork.
Philosopher 0 picked up her right fork.
Philosopher 0 is eating.
Philosopher 1 is thinking.
Philosopher 0 is putting down her right fork.
Philosopher 0 is putting down her left fork.
Philosopher 0 is thinking.
Philosopher 0 picked up her left fork.
Philosopher 1 picked up her left fork.
```

Listing 9: Output from the dining philosophers program.

The program above had two philosophers and two forks. What has happened is that philosopher 0 picks up her left fork and philosopher 1 picks up her left fork. This means that both philosophers are at step 2 and will wait indefinitely for the other to put down their fork. This means that we have a deadlock, a state in which each member waits for the other member, including itsef, to take action.

To solve the dining philosophers problem, we have implemented the Arbitrator solution. This means that we introduced a mutex that can be thought of as a waiter that does not allow a philosopher to pick up both forks or none. This comes at the cost of lost parallellism as well as adding a new mutex.

The change that we did is shown in the listing below:

```
    arbitrator.lock();
    left->lock();
    std::cout << "Philosopher " << n << " picked up her left fork."
     << std::endl;

    right->lock();
    std::cout << "Philosopher " << n << " picked up her right fork.
    " << std::endl;
```

```
7     arbitrator.unlock();
```

Listing 10: Arbitrator/Waiter code modification