COMP6771
Advanced C++ Programming

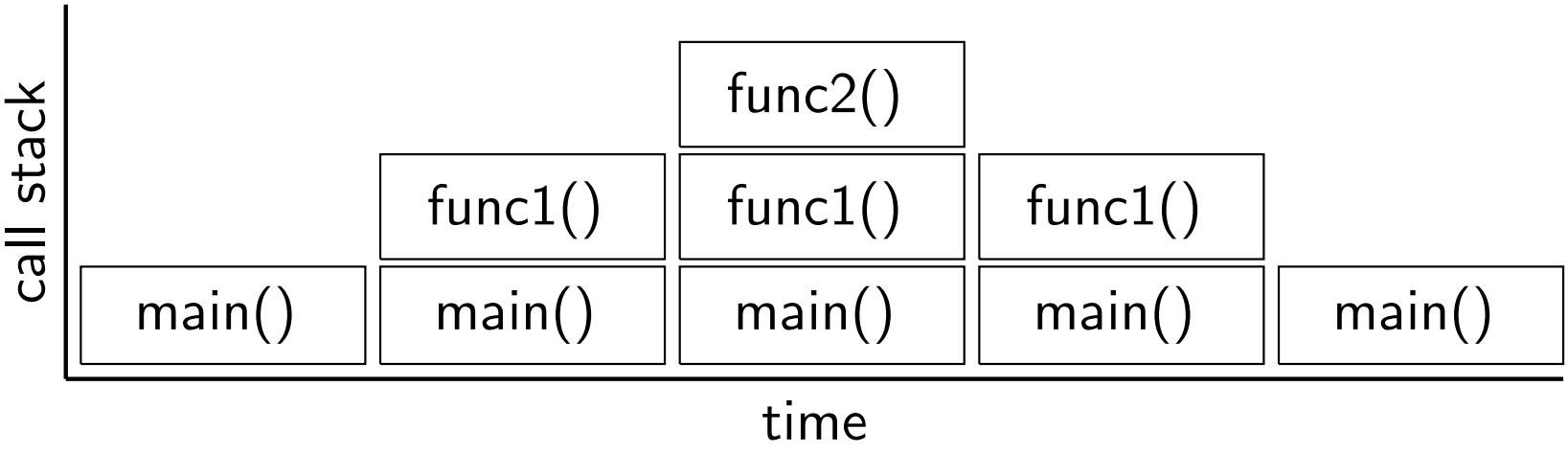
Week 9
Multithreading

2016

www.cse.unsw.edu.au/~cs6771

Single Threaded Programs

- All programs so far this semester have been single threaded
- They have a single call stack, which:
  - increases when a function is called
  - decreases when a function returns



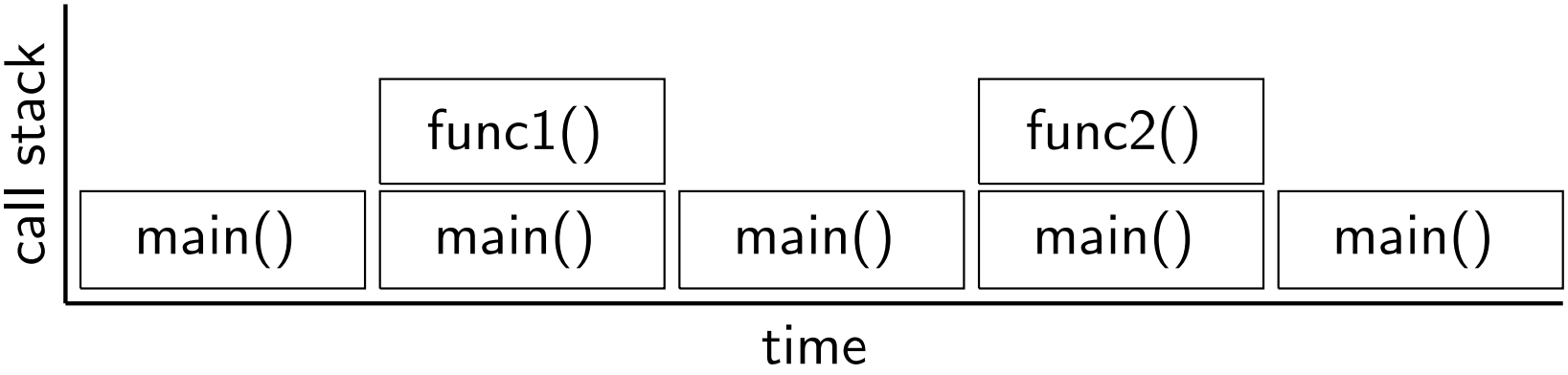
- Only one command executes at any given time

Multi-threaded Prograns

- Most modern CPUs are multi-threaded or have multiple cores
- This means they can execute multiple commands at the same time
- Dual and Quad core CPUs are common
- GPUs (graphical processing units) can have thousands of cores
- Programs that use multiple threads can be many times faster than single-threaded programs

Multi-threading in C++

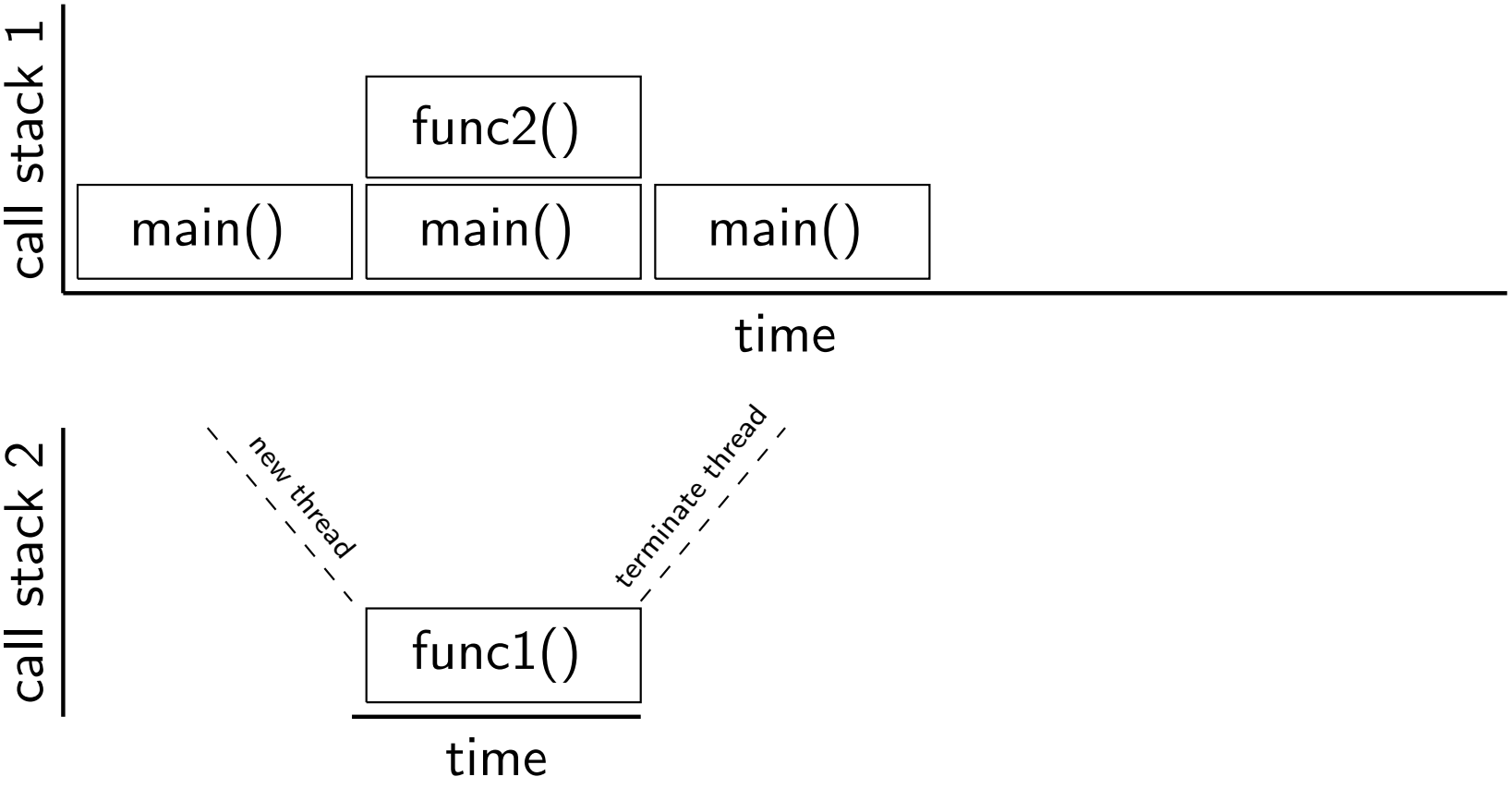
- Each program begins with one thread
- Can use the thread library in C++11 to create additional threads when required
- Each thread has it's own call stack.
- Consider a single threaded call stack, that calls two unrelated functions:



- func2 can't be executed until after func1 has been executed

Multi-threading in C++

- Now consider a multi-threaded C++ program that creates a new thread for func1:



- func1 and func2 run in parallel, speeding up the overall program execution time

Common types of program workloads

- Sequential

  e.g., Random number generation, the previous number(s) are used to calculate the next number
- Parallel

  e.g., Compressing a file, the file can be split into chunks and a different processor core can independently compress each chunk (note: some sequential code may be required to split and join the chunks)
- Embarrassingly parallel

  e.g., A multi-threaded webserver that serves files to each user using a different processor. No communication between threads is required.

Speedup

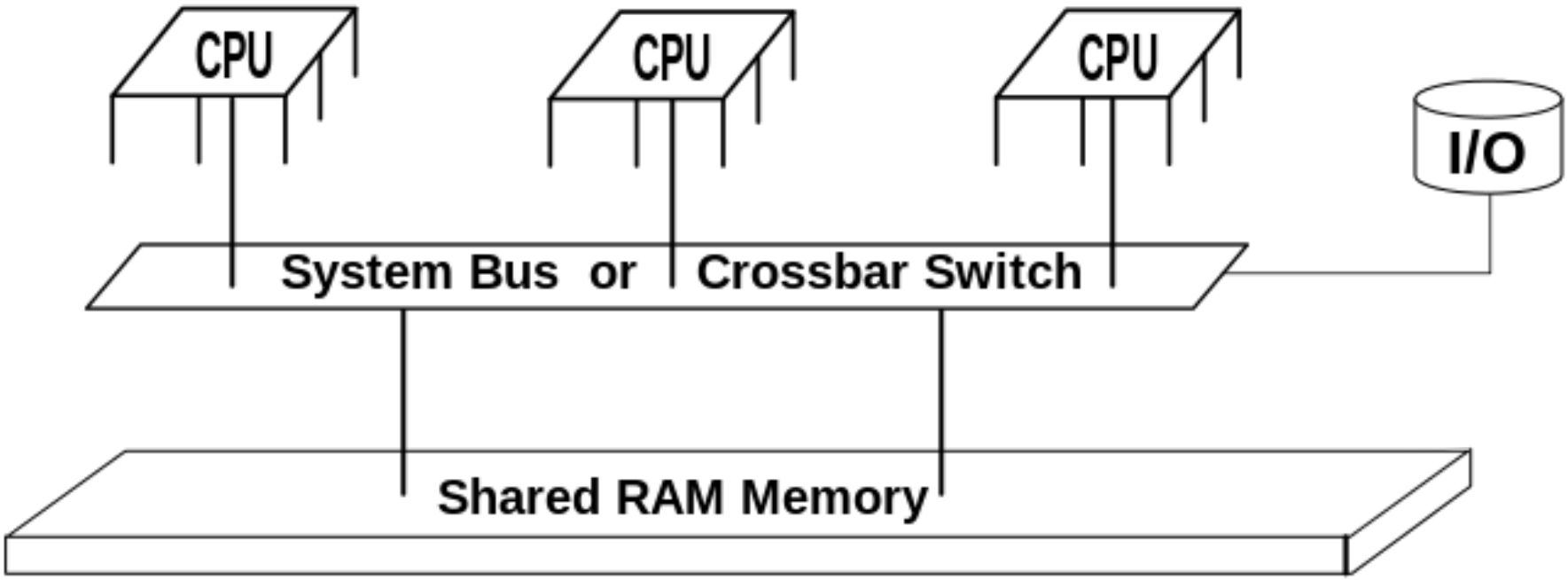
A metric to measure the increase in performance of parallel over sequential task execution. Super-linear speedup can occur in embarrassingly parrallel problems.

Memory Models

- Shared Memory
  All threads share the same memory. Need to ensure that one thread doesn't write to the same memory location that another thread is reading from at the same time
- Local Memory
  Each thread has it's own exclusive memory that other threads cannot access
- Message Passing
  Each thread has it's own exclusive memory and can share data with other threads by sending them data objects through messages

Symmetric multiprocessing (SMP)

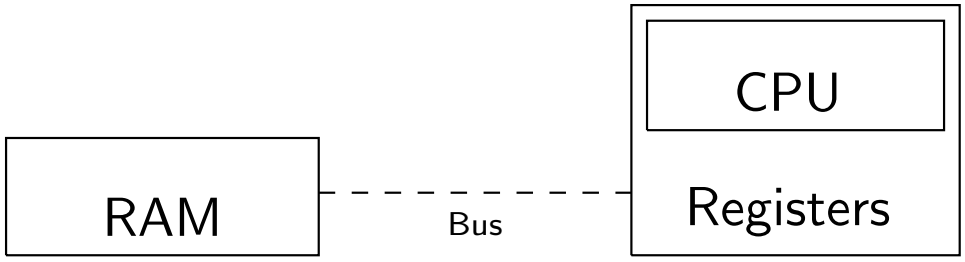
Most consumer multi-core CPUs have this architecture.



https://en.wikipedia.org/wiki/Symmetric_multiprocessing

Atomic Operations

- An operation that, generally, takes one CPU clock cycle. Most memory writes are atomic.
- Consider the command: `i++`
- When you compile the machine code may look like:
  1. Load value from RAM location 0x42 into register r1
  2. Increase value in register r1
  3. Store value from register r1 into RAM location 0x42



- Each machine instruction takes one clock cycle
- Only one thread can write to a specific memory location each cycle (but they may be able to write to different locations)

Race Conditions

- Consider: `int i = 1;`
- Thread 1 wants to increment i: `i++;`
- Thread 2 wants to decrememnt i: `i--;`
- The two commands may consist of the following machine instructions:

// Incremement:

1. Load value from RAM location 0x42 into register r1
2. Increase value in register r1
3. Store value from register r1 into RAM location 0x42

// Decrement:

1. Load value from RAM location 0x42 into register r2
2. Decrease value in register r2
3. Store value from register r2 into RAM location 0x42

Three possible outcomes

- Because the writing into the memory location 0x42 is atomic, there are three possible final values for i
- Case 1: increment happens entirely before the decrement
- Afterwards, i == 1;

```
// Thread 1:
Load 0x42 into register r1
Increase register r1
Store register r1 into 0x42
```

```
// Thread 2:
Load 0x42 into register r2
Decrease register r2
Store register r2 into 0x42
```

Three possible outcomes

- Because the writing into the memory location 0x42 is atomic, there are three possible final values for i
- Case 2: decrement happens entirely before the incremement
- Afterwards, i == 1;

```
// Thread 1:
Load 0x42 into register r1
Increase register r1
Store register r1 into 0x42
```

```
// Thread 2:
Load 0x42 into register r2
Decrease register r2
Store register r2 into 0x42
```

Three possible outcomes

- Because the writing into the memory location 0x42 is atomic, there are three possible final values for i.
- Case 3: The increment happens in parallel with the decrement.
- Afterwards, i == 0 (or 2);

```
// Thread 1:
Load 0x42 into register r1
Increase register r1
Store register r1 into 0x42
```

```
// Thread 2:
Load 0x42 into register r2
Decrease register r2

Store register r2 into 0x42
```

- Problem! incrementing and decrementing i should == 1! Not 0 (or 2).
- Example: `threadLamdaRace.cpp`

threadLamdaRace.cpp

```
1  #include <iostream>
2  #include <thread>
3
4  // NOTE: do not compile this with -O2 it optimises out the ++
5  // call and prevents the race condition from happening.
6
7  int main() {
8    int i = 1;
9    const long numIterations = 1000000;
10   std::thread t1{[&] {
11     for (int j = 0; j < numIterations; ++j) {
12       i++;
13     }
14   }};
15   std::thread t2{[&] {
16     for (int j = 0; j < numIterations; ++j) {
17       i--;
18     }
19   }};
20   t1.join();
21   t2.join();
22   std::cout << i << std::endl;
23 }
```

Mutual Exclusion: Critical Section

- We need a way to ensure that these three machine instructions always happen as one block operation.
- This block of machine instructions is called a critical section.
- A Mutex allows us to only let a single thread at a time access a critical section.
- A Mutex is an object that is shared between threads.
- Generally you have a Mutex object for each critical section (or shared variable) in your code.

Mutual Exclusion: Locking

- A Mutex can act as a lock on a critical section of code. A thread locks a mutex when it wants to execute a section of code and unlocks it when it is finished with that section.
- If a second thread wants to lock the mutex while it is already locked, the second thread has to wait for the first thread to unlock it. Forgetting to unlock mutexes is a major source of bugs!
- Additionally, you can write code that directly accesses a critical section without using a mutex, but this is very dumb.

Deadlocks

- Consider the following:
- You have two shared objects A and B.
- You have two shared mutexes: mutexA and mutexB.
- Thread 1 wants to call the function: std::swap(A,B)
- Thread 2 wants to call the function: std::swap(B,A)
- The code may look something like this:

```
1 // Thread 1:
2 mutexA.lock();
3 mutexB.lock();
4 std::swap(A,B);
5 mutexA.unlock();
6 mutexB.unlock();
```

```
1 // Thread 2:
2 mutexB.lock();
3 mutexA.lock();
4 std::swap(B,A);
5 mutexB.unlock();
6 mutexA.unlock();
```

- What is the problem with this code?

Deadlocks

```
1 // Thread 1:
2 mutexA.lock();
3 mutexB.lock();
4 std::swap(A,B);
5 mutexA.unlock();
6 mutexB.unlock();
```

```
1 // Thread 2:
2 mutexB.lock();
3 mutexA.lock();
4 std::swap(B,A);
5 mutexB.unlock();
6 mutexA.unlock();
```

- What is the problem with this code?
- Thread 1 tries to lock mutexB which is already locked by thread 2 so waits for it to be unlocked.
- Thread 2 tries to lock mutexA which is already locked by thread 1 so waits for it to be unlocked.
- The code is deadlocked as both threads are waiting for each other.
- Deadlocks are a major source of bugs and are not easy to test for.

C++11 Threads

- Defined in <thread> header file
- Need to link against pthread under g++ on linux
  `g++ -std=c++11 -Wall -Werror -O2 -pthread -o file file.cpp`
- New threads can be started on functions, function objects, lamdas, or member functions.

Threads with Function Pointer

```
1  #include <iostream>
2  #include <thread>
3
4  void counter(int numIterations) {
5    for (int i = 0; i < numIterations; ++i) {
6      std::cout << "Thread id: " << std::this_thread::get_id();
7      std::cout << " value = " << i << std::endl;
8    }
9  }
10
11 int main() {
12   std::thread t1{counter, 6}; // start two threads
13   std::thread t2{counter, 4};
14   counter(2); // call counter on the main thread too
15   t1.join(); // wait for the two threads to end.
16   t2.join();
17 }
```

join()

The call to join() for each thread is required for this code to work. It blocks the main thread which waits until the other threads have finished. Note: This isn't always the best approach to take.

Threads with Function Object

```
1  #include <iostream>
2  #include <thread>
3
4  struct Counter {
5    int numIterations;
6    Counter(int n) : numIterations{n} {} // constructor
7    void operator()() {
8      for (int i = 0; i < numIterations; ++i) {
9        std::cout << "Thread id: " << std::this_thread::get_id();
10       std::cout << " value = " << i << std::endl;
11     }
12   }
13 };
14
15 int main() {
16   std::thread t1 { Counter { 10 }};
17   // named variable
18   Counter c5;
19   std::thread t2{c};
20   t1.join();
21   t2.join();
22 }
```


See: p.750 Gregoire, M., Professional C++ (2014).

Threads with Lamda Functions

```
1  #include <iostream>
2  #include <thread>
3
4  int main() {
5    auto lamdaFunc = [](int numIterations) {
6      for (int i = 0; i < numIterations; ++i) {
7        std::cout << "Thread id: " << std::this_thread::get_id();
8        std::cout << " value = " << i << std::endl;
9      }
10   };
11
12   std::thread t1{lamdaFunc,10};
13   std::thread t2{lamdaFunc,5};
14   t1.join();
15   t2.join();
16 }
```

Threads with Lamda Functions 2

Be careful if you have lamda captures - by reference.

```
1  #include <iostream>
2  #include <thread>
3
4  int main() {
5    int numIterations = 10;
6
7    auto lamdaFunc = [&numIterations]() {
8      for (int i = 0; i < numIterations; ++i) {
9        std::cout << "Thread id: " << std::this_thread::get_id();
10       std::cout << " value = " << i << std::endl;
11     }
12   };
13
14   std::thread t1{lamdaFunc};
15   numIterations = 5;
16   std::thread t2{lamdaFunc};
17   t1.join();
18   t2.join();
19 }
```

How many times does `thread t1` print?

Threads with Lamda Functions 3

Be careful if you have lamda captures - by value.

```
1  #include <iostream>
2  #include <thread>
3
4  int main() {
5    int numIterations = 10;
6
7    auto lamdaFunc = [numIterations]() {
8      for (int i = 0; i < numIterations; ++i) {
9        std::cout << "Thread id: " << std::this_thread::get_id();
10       std::cout << " value = " << i << std::endl;
11     }
12   };
13
14   std::thread t1{lamdaFunc};
15   numIterations = 5;
16   std::thread t2{lamdaFunc};
17   t1.join();
18   t2.join();
19 }
```

How many times does `thread t1` print?

Threads with Member Function

```
1  #include <iostream>
2  #include <thread>
3
4  struct Iterator {
5    void counter(int numIterations) {
6      for (int i = 0; i < numIterations; ++i) {
7        std::cout << "Thread id: " << std::this_thread::get_id();
8        std::cout << " value = " << i << std::endl;
9      }
10   }
11 };
12
13 int main() {
14   Iterator i; // create an object
15   // call the member function through a thread
16   std::thread t1 { &Iterator::counter, &i, 10 };
17   // call the same member function on the same object in parallel
18   i.counter(5);
19   t1.join();
20 }
```

Atomic Operations

- Atomic types automatically synchronise and prevent race conditions
- The C++11 standard provides atomic primative types:

Typedef name	Full specialisation
atomic_bool	atomic<bool>
atomic_char	atomic<char>
atomic_int	atomic<int>
atomic_uint	atomic<unsigned int>
atomic_long	atomic<long>

Atomic Operations Example

Our problematic race condition fixed with an atomic int:

```
1  #include <iostream>
2  #include <thread>
3  #include <atomic>
4  int main() {
5    std::atomic<int> i {1};
6    const long numIterations = 1000000;
7    std::thread t1([&i,numIterations] {
8      for (int j = 0; j < numIterations; ++j) {
9        i++;
10     }
11   });
12   std::thread t2([&i,numIterations] {
13     for (int j = 0; j < numIterations; ++j) {
14       i--;
15     }
16   });
17   t1.join();
18   t2.join();
19   std::cout << i << std::endl;
20 }
```

Is this a good approach?

Atomic Performance Problem

- The purpose of multi-threaded code is to improve performance
- Atomic operations slow down our code as they need to synchronise through locking
- How you structure multi-threaded code has a large influence on performance
- In our example we lock every time we need to increment, how might we do this better?

Improved Performance Example

```
1  int main() {
2    std::atomic<int> i {1};
3    const long numIterations = 1000000;
4    std::thread t1([&i,numIterations] {
5      int k = 0; // use a local variable
6      for (int j = 0; j < numIterations; ++j) {
7        k++;  // generally more complex math
8      }
9      i += k;
10   });
11   std::thread t2([&i,numIterations] {
12     int k = 0;
13     for (int j = 0; j < numIterations; ++j) {
14       k--;
15     }
16     i -= k;
17   });
18   t1.join();
19   t2.join();
20   std::cout << i << std::endl;
21 }
```

Only write to the atomic int once per thread!

Atomic Operations Library

- The standard defines a number of operations on atomic types.
- These operations will be faster than your code.
- Example, `fetch_add()`:

```
1  #include <iostream>
2  #include <atomic>
3  #include <thread>
4
5  int main() {
6    std::atomic<int> i{10};
7    std::cout << "Initial Value: " << i << std::endl;
8    int fetchedValue = i.fetch_add(5);
9    std::cout << "Fetched Value: " << fetchedValue << std::endl;
10   std::cout << "Current Value: " << i << std::endl;
11 }
```

- Ouput:

```
1  Initial Value: 10
2  Fetched Value: 10
3  Current Value: 15
```