Threading

PIC 10B, UCLA

Threading

In programming, **multithreading** is a technique to break a process into smaller chunks that can be done concurrently. This can greatly speed up overall runtime.

Just imagine doing a time-intensive process on a **std::vector** with 1,000,000 elements. Given enough computing power, we could divide the work up into, say 2, 3, 4. ..., separate sets of instructions (**threads**) running independently. Then our work could be done 2, 3, 4. ..., times faster (or thereabouts).

Terminology

A few pieces of terminology that are relevant:

The **CPU** of a computer can contain one or more **cores**. A **core** is a part of the CPU that can be given instructions and perform calculations based on those instructions.

A **process** is the set of instructions a computer program is carrying out. A **process** can be divided into multiple **threads** of instruction that can be executed on different **cores**, possibly sharing resources like global variables, etc.

thread

To work with **threads** in C++, we include the **<thread>** library.

Before using threads, one somewhat useful function to call is:

```
std::thread::hardware_concurrency();
```

It returns an estimate for how many threads are supported. There is no way to determine the exact number.

thread

A **std::thread** has a **variadic constructor**. One must provide a functor (lambda, function pointer, callable class, etc.). Then zero or more arguments are provided as input arguments to that functor.

The **join** member function ensures a thread is done executing before the process continues.

Unless the thread has been detached (beyond this course), if its destructor is called before it has **join**ed, an exception is thrown and the program will terminate.

thread

A simple example:

```
#include<thread>

void foo(int *i) { *i = 0; }

int main() {
    int i = 5;
    std::thread t( foo, &i );
    t.join();
    return i; // returns 0
}
```

This trivial example is for illustration only: we created a thread that calls **foo** with the argument of **&i**: it changes **i** to **0**. We **join** the thread and return the value of **i**.

thread

The program below will crash due to the thread not being joined!

```
#include<thread>

void foo(int *i) { *i = 0; }

int main() {
    int i = 5;
    std::thread t( foo, &i );
    // bad, no joining so t's destructor called while still joinable
    return i;
}
```

thread

By default, all the functor inputs passed to the **thread** constructor are **passed by value, even if the functor itself accepts arguments by reference**!!!

There is a way around this but we won't delve into it here. So in our examples, we will pass around pointers.

Summing in Parallel I

The function below does some “busy work”. Given a range of **int**s, it computes the sum of all sums of consecutive **int**s up to each **int** in the original range. There’s nothing special about this: it just keeps the computer busy, to represent a complex process that requires threading.

```
void busy_work(const int* begin, const int* end, long long* res) {
    long long total = 0; // local total

    for (const int* p = begin; p != end; ++p) { // for every int
        for (int i = 0; i <= *p; ++i) { // add consecutive ints up to *p to total
            total += i;
        }
    }

    *res = total;
}
```

Summing in Parallel II

Here we make ourselves a vector storing a lot of random values where we can apply threading to sum all the sums of consecutive values in the vector.

```
int main() {
    constexpr int MAX = 10000; // max int we consider
    constexpr size_t BIG = 100000; // elements in vector

    std::vector<int> v;     v.reserve(BIG);

    for (size_t i = 0; i < BIG; ++i) { // set all the values
        v.push_back(std::rand() % MAX);
    }

    int * const vfirst = &(v[0]); // the first element of v
```

Summing in Parallel III

Below we use 2 threads.

```
long long res1 = 0, res2 = 0; // to store results from the threads

std::thread t1(busy_work, vfirst, vfirst + BIG / 2, &res1); // first half

// second half
std::thread t2(busy_work, vfirst + BIG/2, vfirst + BIG, &res2);

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

long long thread_total = res1 + res2;
```

Summing in Parallel IV

Below we do not use threads.

```
    long long no_thread_total = 0;

    busy_work(vfirst, vfirst + BIG, &no_thread_total);

    return 0;
}
```

Speed

Using **threads** doesn't automatically make code faster! There is some system overhead in setting up a thread. Consider the previous example with different values of **BIG**.

BIG	2 threads	no threading
10^7	8.3×10^7 μs	1.4×10^8 μs
10^6	7.8×10^6 μs	1.3×10^7 μs
10^5	1.1×10^6 μs	1.6×10^6 μs
10^4	3.7×10^5 μs	2.4×10^5 μs
10^3	4.4×10^4 μs	1.6×10^4 μs

For easy jobs, **do not use threads**.

Speed

The time it takes for threaded processes is not described perfectly by being inversely proportional to N, the number of threads. In addition, once the number of threads exceeds the maximum that can run concurrently, the threads have to share a core.

With **BIG** $= 10^6$ and **std::thread::hardware_concurrency() = 4**, we have:

threads	time	speedup factor
1	1.3×10^7 μs	n/a
2	7.8×10^6 μs	1.7
3	5.8×10^6 μs	2.2
4	5.0×10^6 μs	2.6
5	4.9×10^6 μs	2.7
6	4.8×10^6 μs	2.7

Race Conditions

When multiple threads are accessing the same data and at least one of the threads is writing to the data, this is called a **race condition**. Because they can all operate simultaneously, the state of the variable at any given time is anyone's guess.

Race Conditions

Consider:

```
void print(int i) {
    for(int c = 0; c < 10; ++c) { std::cout <<i; }
}
```

and then the threading:

```
std::thread t1(print, 1); // should print ten 1's
std::thread t2(print, 2); // should print ten 2's
t1.join();
t2.join();
```

Both threads are writing to **std::cout**. The output could be anything, for example:

```
12111222111222121212
```

Race Conditions

In threading, a lot of functions that we take for granted can lead to race conditions. For example, **std::rand()** is not thread-safe. Working with **std::cin** and **std::cout** is in general not thread-safe.

We cover a very brief example of how to fix a race condition. In general, it involves locking a set of instructions to only allow one thread access at a time. To write good code, these locked sections should be kept to a minimum or the whole code behaves like a single process.

std::mutex

A **std::mutex** is a resource that “locks” a block of code, only allowing one thread access. Other threads have to wait their turn before accessing the function. It is defined in the **<mutex>** header.

It obeys **RAII**: once locked, the code block is locked to other threads until the mutex unlocks it. Forgetting to unlock a mutex is akin to a memory leak.

Due to the system-wide behaviour of mutexes, *it is common, and even encouraged, for them to be global variables!*

On the next slide we present valid code that ensures all the 1’s will be printed together and all the 2’s will be printed, without weird mixes.

std::mutex

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

std::mutex MUTEX; // global

void print(int i) {
    MUTEX.lock(); // code starting here locked, do the printing...
    for(int c = 0; c < 10; ++c) { std::cout <<i; }
    MUTEX.unlock(); // unlock so the other threads can print
}

int main() {
    std::thread t1(print, 1);
    std::thread t2(print, 2);
    t1.join();     t2.join();
    return 0;
}
```

Aside: False Sharing

For very technical reasons relating to compiler optimization, sometimes when two threads are writing to disjoint, independent pieces of memory, they can still interfere with each other if their addresses are close. This is sometimes called **false sharing**.

It is the reason why in the **busy_work** function, we made a local variable rather than adding directly to ***res** within the loops.

Summary

- A **thread** is an object that can set a process in motion that can be done in parallel with other threads' processes.
- The constructor is variadic: it accepts a functor and a list of arguments to use.
- Often a thread should be **join**ed before it is destructed.
- When multiple threads are accessing the same data and at least one of them is writing, there is a **race condition**.
- A **mutex** is a resource to lock coding instructions to fix race conditions, but it must be unlocked: they obey RAII.
- It is common in threads to make local variables before writing to an argument due to **false sharing**.

Exercises

1. Use **std::thread_hardware_concurrency()** to break the **busy_work** example into (approximately) as many threads as possible. Avoid hard-coding: let the number of concurrent threads be a variable and work with it.
2. Redo the **busy_work** example with threads, but pass the address of the total variable to all the threads (like was done without threading). The total can start at zero outside of the function calls. Only add to it at the end of the function call. Use a **std::mutex** to guard against race conditions. Your final program should not require the intermediate values **res1**, **res2**, etc.