Final Boss

Intro to Parallelization with Multithreading

Parallelization & Concurrency Definitions

- Programs are executed on a system in one of two ways:
  - In Serial
  - In Parallel

- Serial programs are what we have been writing all semester: every statement in our program is executed in order, top to bottom.
  - These programs are straightforward and simple.

- Parallel programs have multiple pieces of code that are all executing *simultaneously*.
  - These programs can be much more complex!

- Parallel program typically use *multithreading*, a common technique for achieving parallelism.

Parallelization & Concurrency Definitions

- Concurrency refers to the ability to process multiple tasks *over some span of time*, but not necessarily at the same time.

- Concurrency without parallelization uses context switching
  - e.g., I can tend to my garden and clean my yard, but never at the same time. I can go back and forth between the two tasks, but I do not need to finish one to start the other, and I can stop one to work on the other until they are both finished.

- Concurrency with parallelization uses multithreading
  - e.g., I can cook a meal while talking on the phone, these are happening at the same time. I can easily do both at the same time, and it does not matter exactly the order in which I complete them.

Parallelization

- A more common example is updating a matrix using some operation.
  - e.g., scalar multiplication

$$5 \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

we might see code like this:

```
for (auto &row : matrix) {
    for (auto &element : row) {
        element *= 5;
    }
}
```

- This works perfectly fine, even for relatively large matrices (100x100, 1000x1000). As we get larger though this solution becomes increasingly problematic. Even worse, there is no good way to optimize more.

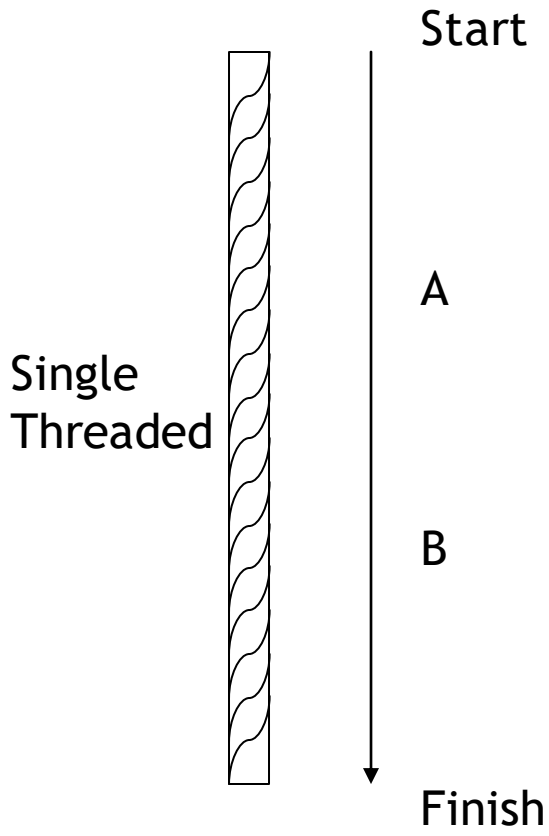
- This method effectively reads: *update row 1, then update row 2, then update row 3*

Parallelization

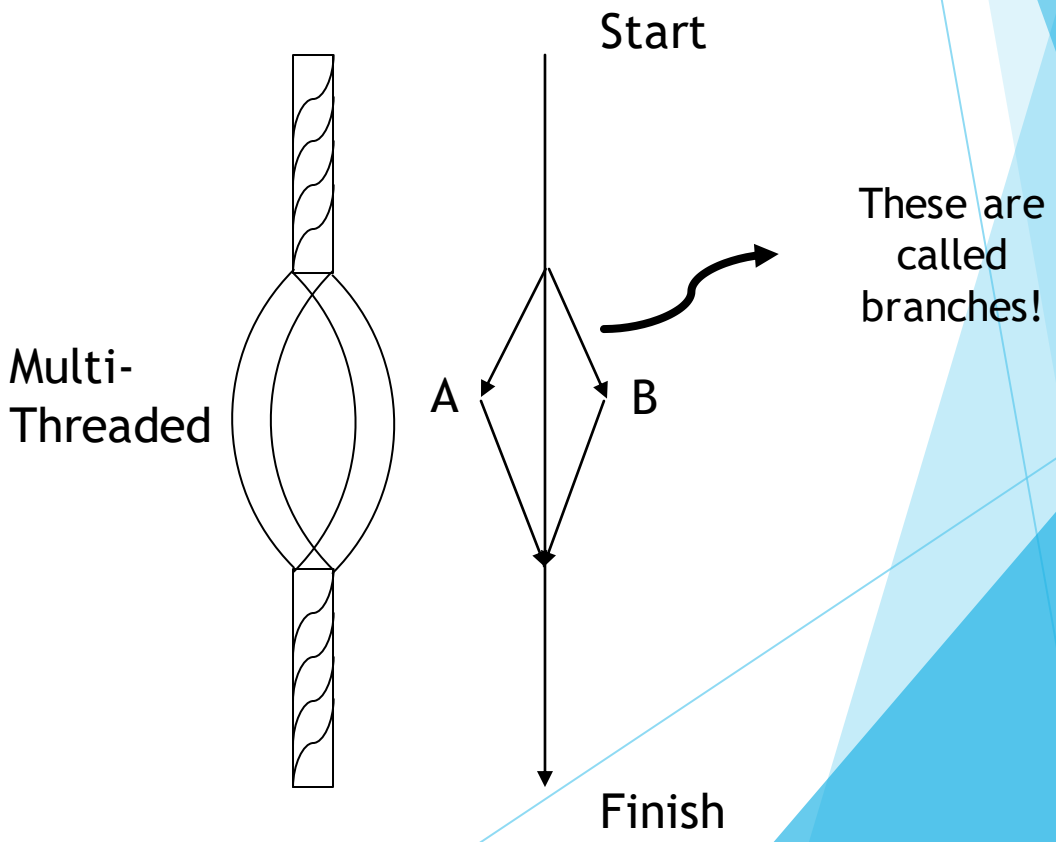
- Conceptually, it would be nicer if we could do the following tasks in parallel.
  - *update row 1 while we also update rows 2 and 3*
- But how do we tell the computer to perform the update of row 1 while also updating rows 2 and 3... all simultaneously?
- We use *threads*. As we execute a program, think of the entire program as one long thread that accounts for all the code and functions and variables. We travel along the thread from start to finish. It is continuous and we cannot jump ahead. Our thread though can split and unwind a little, allowing multiple tasks to be carried out at once.

Parallelization

Serial Execution



Parallel Execution



These are called branches!

Multithreading

- Multithreading is the process in which multiple **processors** are leveraged to execute multiple branches of code simultaneously.
  - The number of threads that can live concurrently is dependent on a combination of the hardware and operating system. We must differentiate between hardware and software threads!

- Once a thread is created it begins running immediately. At this point the thread can be left alone to be ***joined*** later or it can be ***detached***.
  - joining a thread halts the current thread to wait for the joining thread to finish.
  - detaching a thread separates the thread from the current thread altogether to finish on its own.

Multithreading

- Threading in C++ on the surface just requires one new header:

```
#include <thread>
```

  - There are others needed depending on the complexity of usage.

- We create new threads using the `std::thread` object.
  - Its constructor takes in a *function pointer* and all the arguments for that function.

- Once we create a thread, we can choose to detach it or hold onto it to join it later.

Example

```
#include <chrono>
#include <iostream>
#include <thread>

using namespace std::literals::chrono_literals;

auto foo() -> void {
    std::this_thread::sleep_for(2s);
    std::cout << "Thread #" << std::this_thread::get_id() << " done!"
              << std::endl;
}

auto main() -> int {
    auto t1 = std::thread{foo};
    auto t2 = std::thread{foo};
    auto t3 = std::thread{foo};
    t1.join();
    t2.join();
    t3.join();
}
```

Example Cont'd

- Here is a possible output from our example:

```
Thread #2Thread #1 done!
 done!Thread #3 done!
```

Example Cont'd

- Here is a possible output from our example:

```
Thread #2Thread #1 done!
 done!Thread #3 done!
```

- Our program is interleaving the output!

- This is because `std::cout` is shared between all threads, and each line using it is broken down into multiple actions. Each usage of << is an individual action, even though it is part of a single statement!

- We can resolve this by using a singular action to print!

Example Cont'd

```
#include <chrono>
#include <thread>
#include <fmt/format.h>
#include <fmt/std.h>

using namespace std::literals::chrono_literals;

auto foo() -> void {
    std::this_thread::sleep_for(2s);
    fmt::print("Thread #{} done\n", std::this_thread::get_id());
}

auto main() -> int {
    auto t1 = std::thread{foo};
    auto t2 = std::thread{foo};
    auto t3 = std::thread{foo};
    t1.join();
    t2.join();
    t3.join();
}
```

Example Cont'd

- Here is a possible output from our example:

```
Thread #2 done!
Thread #1 done!
Thread #3 done!
```

Example Cont'd

- Here is a possible output from our example:

```
Thread #2 done!
Thread #1 done!
Thread #3 done!
```

- Why did thread #2 come before thread #1?

- Starting threads is expensive and indeterministic!

Multithreading Issues

- Multithreading brings with it a brand-new set of issues when dealing with memory and timing.
- What happens if one thread updates a variable as another one reads it? Does the second thread get the old or the new value? This is called a *race condition*.
- When dealing with multiple threads we have ensure that we are sharing memory safely among them and properly synchronizing them.

Protecting Shared Memory

- We can protect shared memory from improper access and usage using a *mutex*.
  - this is short for *mutual exclusion*
- A thread locks a mutex while it performs some set of actions and all other threads must wait until the mutex is unlocked.
- Threads are responsible for unlocking any mutex they lock, otherwise the program can experience a *deadlock*.
  - Every thread is waiting for a mutex to unlock and thus your program is frozen.
- We must include the ***mutex*** header so that we can create *std::mutex* objects. When appropriate we must ***lock*** the mutex, and then ***unlock*** it.

Example

- What is wrong here?

```
#include <mutex>
#include <thread>
#include <fmt/format.h>

auto m = std::mutex{};  // global, shared between
                        // all threads
auto foo() -> void {
    m.lock();
    fmt::print("all done!\n");
}

auto main() -> int {
    auto t1 = std::thread{foo};
    auto t2 = std::thread{foo};
    auto t3 = std::thread{foo};
    t1.join();
    t2.join();
    t3.join();
}
```

Example Cont'd

- What is wrong here?

```
#include <mutex>
#include <thread>
#include <fmt/format.h>

auto m = std::mutex{};  // global, shared between
                        // all threads
auto foo() -> void {
    m.lock();
    fmt::print("all done!\n");
}

auto main() -> int {
    auto t1 = std::thread{foo};
    auto t2 = std::thread{foo};
    auto t3 = std::thread{foo};
    t1.join();
    t2.join();
    t3.join();
}
```

- We never unlock our mutex!
- One of the threads will acquire and lock m, but because it never unlocks it, the other threads will wait indefinitely for the lock to become available.
- This is called a *deadlock*. *They are bad*.

Example Cont'd

- What is wrong here?

```
#include <mutex>
#include <thread>
#include <fmt/format.h>

auto m = std::mutex{};  // global, shared between
                        // all threads
auto foo() -> void {
    m.lock();
    fmt::print("all done!\n");
    m.unlock();
}

auto main() -> int {
    auto t1 = std::thread{foo};
    auto t2 = std::thread{foo};
    auto t3 = std::thread{foo};
    t1.join();
    t2.join();
    t3.join();
}
```

- We never unlock our mutex!
- One of the threads will acquire and lock m, but because it never unlocks it, the other threads will wait indefinitely for the lock to become available.
- This is called a *deadlock*. *They are bad*.
- **To fix this we can unlock the mutex!**

Protecting Shared Memory

- When we want to use a mutex, we should also consider using a *lock guard*.
  - A lock guard is a special object that uses **RAII** to lock a mutex, and later unlock it when it goes out of scope.
  - RAII is a technique to automatically execute commands when objects go out of scope (like closing files, destructing stale memory, and yes, *unlocking mutexes*!)

- Below we create a lock using a mutex; the lock handles locking and unlocking the mutex automatically!

  e.g.,

  ```
  auto my_lock = std::lock_guard{m};
  ```

- Here, my_lock will lock the mutex m until my_lock is deleted (goes out of scope).

Example

```
#include <chrono>
#include <mutex>
#include <thread>
#include <fmt/format.h>
using namespace std::literals::chrono_literals;

auto m = std::mutex{};

auto foo() -> void {
    std::this_thread::sleep_for(2s);
    auto lock = std::lock_guard{m};
    std::cout << "Thread #" << std::this_thread::get_id() << " done!"
              << std::endl;
}

auto main() -> int {
    auto t1 = std::thread{foo};
    auto t2 = std::thread{foo};
    auto t3 = std::thread{foo};
    t1.join(); t2.join(); t3.join();
}
```

Example Cont'd

- Here is a possible output from our example:

```
Thread #3 done!
Thread #2 done!
Thread #1 done!
```

Example

- What's wrong here?

```
#include <thread>
#include <vector>
#include <fmt/format.h>

auto foo(std::vector<int> &d) -> void
{
    d.push_back(1);
}

auto main() -> int
{
    auto d = std::vector<int>{};
    auto t1 = std::thread(foo, std::ref(d));
    auto t2 = std::thread(foo, std::ref(d));
    auto t3 = std::thread(foo, std::ref(d));
    auto t4 = std::thread(foo, std::ref(d));

    fmt::print("{}\n", fmt::join(d, ","));
}
```

Example

- We have two race-conditions!

```
#include <thread>
#include <vector>
#include <fmt/format.h>

auto foo(std::vector<int> &d) -> void
{
    d.push_back(1);
}

auto main() -> int
{
    auto d = std::vector<int>{};
    auto t1 = std::thread(foo, std::ref(d));
    auto t2 = std::thread(foo, std::ref(d));
    auto t3 = std::thread(foo, std::ref(d));
    auto t4 = std::thread(foo, std::ref(d));

    fmt::print("{}\n", fmt::join(d, ","));
}
```

- Every time we push back into the vector another push back could be happening, and if the vector is moving its data during a resize, we could end up losing data or putting data in an invalid location!
- We do not ensure that our threads are complete before trying to read from the vector! We could still be writing to the vector!

Protecting Shared Memory

- Other ways we can protect our memory is to just be *smarter*.
  - Allocate/reserve memory ahead of time
  - **Pass slices of objects that do not overlap**
  - Do not mix reading and writing tasks on the same object(s)

Final Notes

- Splitting a task into multiple threads does not guarantee a proportional reduction in runtime.
  - We must remember the limits of the system and hardware and understand the diminishing returns on adding more threads.
  - Too many people working the same task could ultimately make it take longer!
- Simply starting a new thread requires tremendous system resources, which can mitigate the benefit in simple cases.
- Your program is likely doing *more* work than its single threaded version, but that work is distributed and thus still *completes* more quickly.