CME 213, ME 339 Spring 2023
Introduction to parallel computing using MPI, openMP, and CUDA

Eric Darve, ICME, Stanford

"Controlling complexity is the essence of computer programming"
(Brian Kernigan)

Stanford University

Recap

- Parallel computing and high-performance computing
- Multicore processor architecture; cc-NUMA architectures
- Multithreaded programming
- Process and threads

Processes and threads

Process and threads

- Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads.

- A thread is the basic unit to which the operating system allocates processor time. A thread can execute any part of the process code, including parts currently being executed by another thread.

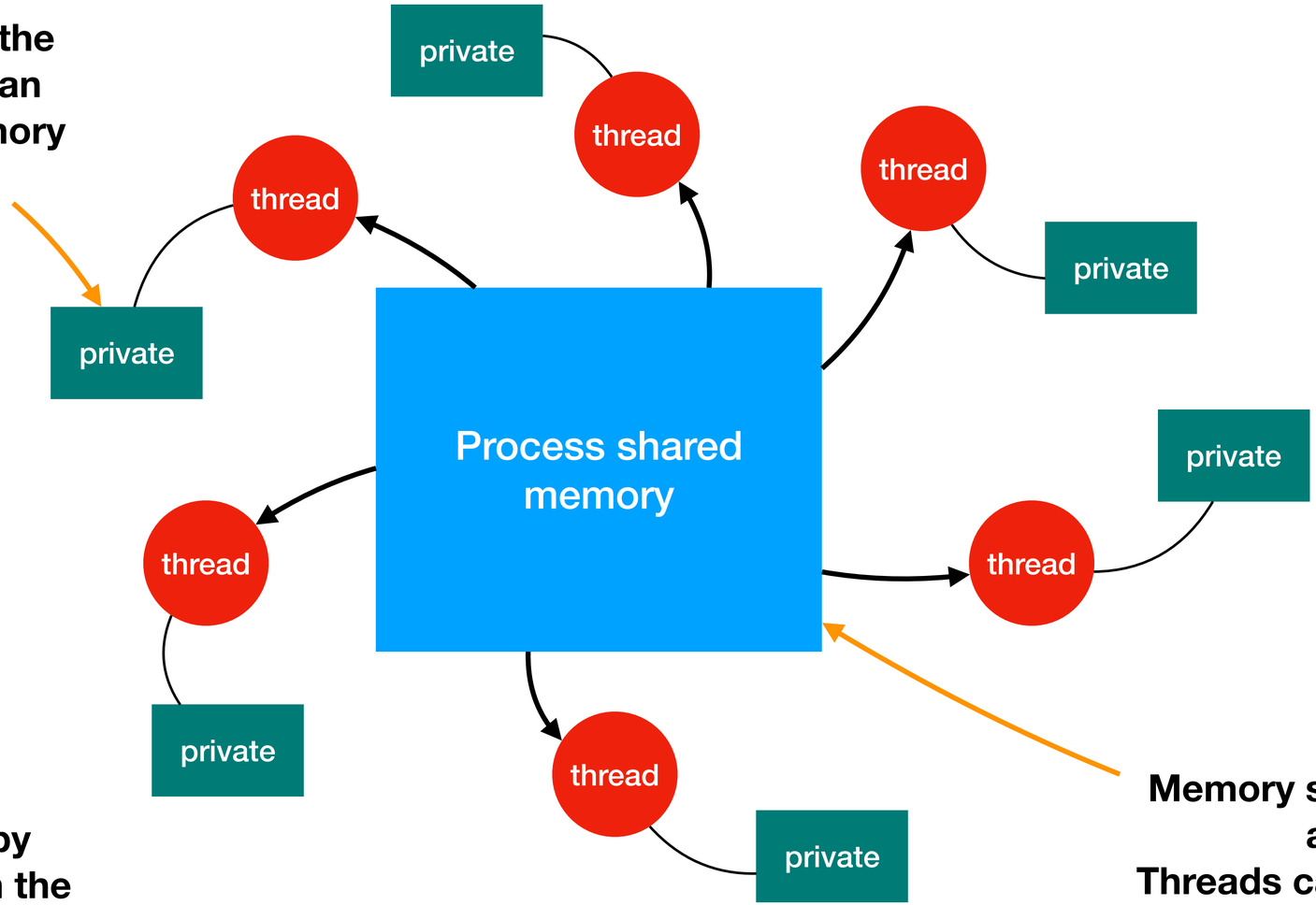
Threads from the programmer's point of view

- In a C++ code, a thread is able to run a procedure in the program independently from the main process thread.

- Imagine a program that contains a number of procedures. Then imagine these **procedures** being able to be scheduled to **run simultaneously and/or independently** by the operating system. This describes **a multi-threaded program.**

Thread communication

This memory is private to the thread. No other thread can read and write to that memory space.



Threads exchange data by writing to and reading from the shared memory space.

**Memory space accessible to all threads.
Threads can read and write to that space.**

Threads are everywhere in all programming languages

- C threads: Pthreads
- Java threads: Thread thread = new Thread();
- Python threads: t = threading.Thread(target=worker)
- Cilk: x = spawn fib (n-1);
- Julia: r = remotecall(rand, 2, 2, 2)
- **C++ threads (11): std::thread**
- **OpenMP**

Let’s dive into our first example

- Live coding demo with C++ threads.
- `C++_threads.ipynb`
- `hello_world_threads.cpp`

Key concepts

- `thread t1(f1);`
- Creates a thread that runs f1.
- `thread t2(f2, m);`
- m is passed as argument to f2.
- `thread t3(f3, ref(k));`
- This is how you pass a reference to a function.

join()

- Since the execution is asynchronous, we need a mechanism to determine when a thread has completed the execution of the function.
- We use join for that:
- `t3.join();`
- The main thread will wait until t3 completes.
- join() must be called before the thread destructor is called. Otherwise, you will get an error during execution.

How can threads exchange data?

- Data is exchanged through the shared memory.
- Due to asynchronous execution, care must be taken.
- There are two main mechanisms:

    1. Modify a shared variable. This is what we did previously. It works well but is limited because we need to use **join()** and **wait** for the thread to finish execution.

    2. A more flexible mechanism is to use **promise.**

promise

`promise` solves two difficulties:

1. Allocating resources to store a return value

2. Querying the return value

std::promise

- **std::promise** provides a facility to **store a value** or an exception that is later acquired asynchronously via a **std::future** object created by the std::promise object.

- The std::promise object is meant to be used only once.

std::future

- std::future provides a mechanism to **access the result** of asynchronous operations:
  - An asynchronous operation (e.g., created via std::promise) can provide a std::future object to the creator of that asynchronous operation.
  - The creator of the asynchronous operation can then use a variety of methods to **query, wait for, or extract a value** from the std::future. These methods **block** if the asynchronous operation has not yet provided a value.

- Live coding example: `cpp_promise.cpp`

promise and future

```
promise<int> sum_promise;  // Will store the int
future<int> sum_future = sum_promise.get_future();
```

- Creates a promise to hold the value of the result.
- Creates a future to query the promise and retrieve the value.

Retrieve the value

```
int sum1 = sum_future.get();
```

- **Blocks** until the promise holds the value.
- **Retrieves** the value from the promise using the future sum_future and assign it to sum1.

Lambda function

- We can simplify a bit the syntax using lambda functions.

```
std::thread([&sum_promise, &vec0] {
  sum_promise.set_value(accumulate(vec0));
}).detach();
```

- This creates a thread that runs the lambda function.
- `detach()` is used to separate the thread of execution from the thread object. This execution continues independently from the main thread.
- Not using `detach()` and not having a `join()` will result in an error during execution.

Summary of promise/future

- Compared to using references or global variables, promise/future offers a more flexible mechanism.
- **As soon as set_value is called on the promise, the value can be acquired using the future.**
- This does not require the thread to complete the execution of the function and using `join()`.
- promise/future allows a **flexible and efficient communication** mechanism between threads.

More information

See https://en.cppreference.com/w/cpp/thread

Thread coordination

- Threads can read and write to shared variables in the shared memory space.
- This can cause errors if two threads try to access/modify the object at the same time.

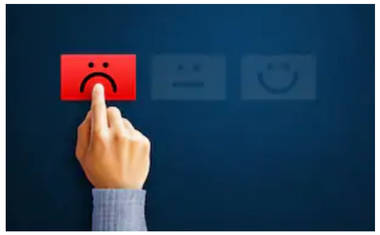
Example of error

A well-known bank company has asked you to implement a multi-threaded code to perform bank transactions

Goal: allow deposits

1. Clients deposit money and the amount gets credited to their accounts.

2. But, a result of having multiple threads running concurrently the following can happen:

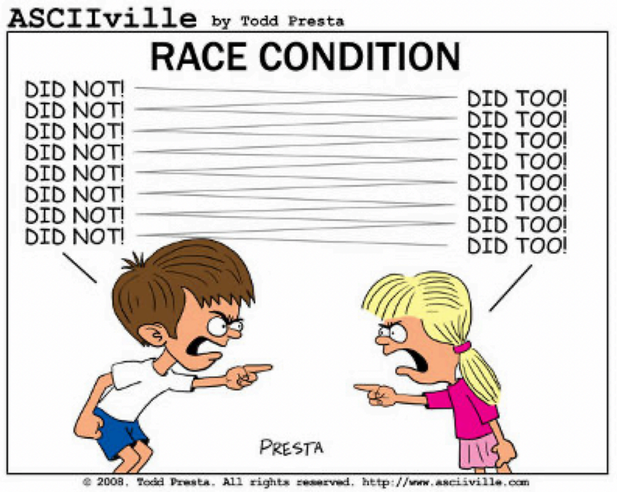


Possible execution scenario

Thread 0	Thread 1	Balance
Client requests a deposit	Client requests a deposit	$1000
Check current balance = $1000		
	Check current balance = $1000	
Ask for deposit amount = $100	Ask for deposit amount = $300	
	Compute new balance = $1300	
Compute new balance = $1100	Write new balance to account	**$1300**
Write new balance to account		**$1100**

Race condition

- This is called a **race condition.**
- The final result depends on the order in which the instructions are executed.



When do you have a race condition?

A race condition occurs when you have a sequence like

- READ/WRITE, or
- WRITE/READ

performed by different threads.

mutex

- A mutex needs to be used to protect access to shared variables.
- Note that: this is only required for READ/WRITE and WRITE/READ situations.
- READ/READ is not a problem.
- mutex can ensure that the correct order of operations is preserved.

Definition of mutex

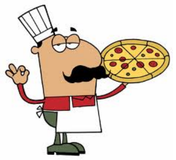
- The mutex class is a **synchronization primitive** that can be used to protect shared data from being simultaneously accessed by multiple threads.
- mutex offers exclusive, non-recursive ownership semantics:
  - A calling thread **owns** a mutex from the time that it successfully calls **lock** until it calls **unlock**.
  - When a thread owns a mutex, all other threads will **block** if they attempt to claim ownership of the mutex.
- A calling thread must not own the mutex prior to calling lock.
- The behavior of a program is undefined if a mutex is destroyed while still owned by any threads, or a thread terminates while owning a mutex.
- `std::mutex` is neither copyable nor movable.

Schematic use case; modifying a linked list

Thread 0	Thread 1
Thread 0 wants to add a new to-do item.	
Thread 0 owns mutex and locks it.	
Thread 0 adds an entry in list.	Thread 1 tries to own the mutex. It blocks.
Thread 0 is done with the to-do list. It unlocks the mutex.	
	Thread 1 is now able to own the mutex. It locks the mutex.
	Thread 1 adds an entry in list.
	Thread 1 unlocks the mutex.

The pizza restaurant

Pizza cook



Receives all the orders.
Prepares all the pizzas.

Ask delivery team to deliver pizzas to customers

Pizza delivery team

- Checks the addresses for customers
- Deliver pizza

Go back; check if there are orders left.

File `mutex_demo.cpp`

```
g_mutex.lock();

while (!g_task_queue.empty()) {
```

The thread must lock the mutex before checking the state of the queue.

```
g_mutex.unlock();

Delivery();

g_mutex.lock();
```

- The thread can unlock the mutex after the entry has been removed from the queue.
- Delivery() can happen while other threads access the queue.
- The thread needs to lock the mutex again before checking whether the queue is empty or not.

```
g_mutex.unlock();
return;
```

- Don’t forget to unlock() before terminating the thread!

lock_guard()

- lock() and unlock() work well and are easy to use.
- There are a few scenarios where this may lead to errors.
- For example, if a function throws an exception, your code may fail to unlock the mutex.

```
std::mutex m;  // global mutex

void oops() {
  m.lock();
  doSomething();
  m.unlock();
}
```

- If doSomething() throws an exception, the mutex remains locked.

How to use lock_guard()

- lock_guard() provides a safer mechanism where the mutex is automatically **unlocked when going out of scope.**
  - The class lock_guard is a mutex wrapper that provides a convenient mechanism for owning a mutex for the duration of a scoped block.
  - When a lock_guard object is created, it attempts to take ownership of the mutex it is given. **When control leaves the scope in which the lock_guard object was created, the lock_guard is destructed and the mutex is released.**

```
void safe_increment() {
  const std::lock_guard<std::mutex> lock(g_i_mutex);
  g_i += 3;  // Increment the shared variable

  std::cout << "g_i: " << g_i << "; in thread #" <<
  std::this_thread::get_id()
            << '\n';

  // g_i_mutex is automatically released when lock
  // goes out of scope
}
```

The lock_guard takes care of locking and unlocking the mutex as needed.

- More information at https://en.cppreference.com/w/cpp/header/mutex

Summary

- C++ threads can run functions asynchronously.
- promise/future can be used to retrieve values computed by a thread.
- A mutex needs to be used to protect access to shared variables.
- lock_guard() is a safer way to use a mutex; the mutex is automatically unlocked when the lock_guard goes out of scope.