CS100
Introduction to Programming

Lecture 23. Concurrency

Today's learning objectives

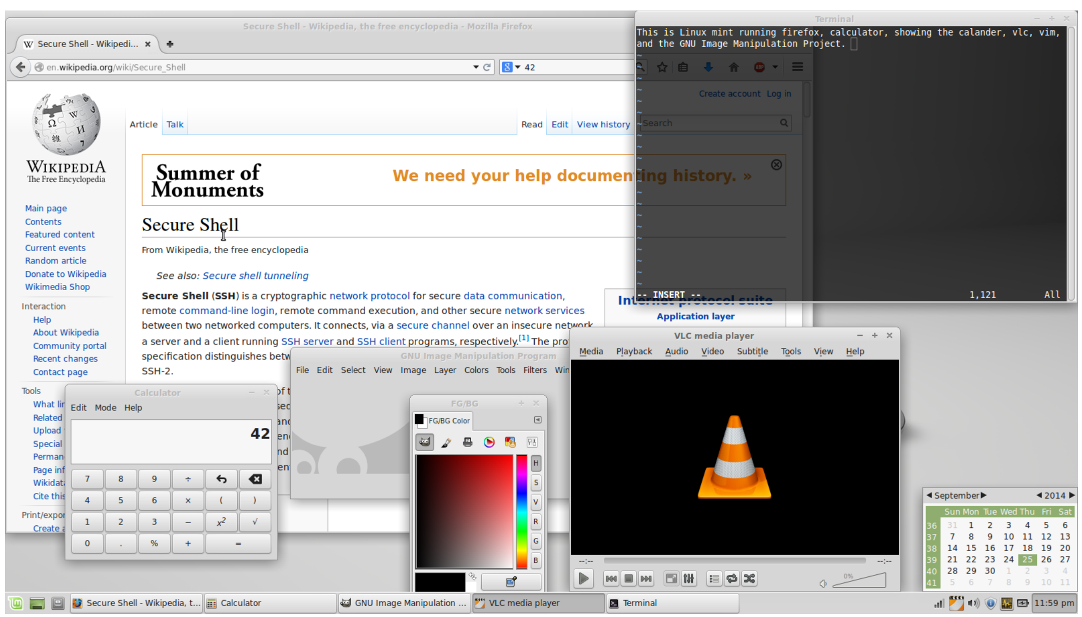
- Understand the need for/meaning of multi-tasking, concurrency, and parallel computation
- Understand the difference between processes and threads
- Learn how to implement them
- Learn about thread safety and how to implement it
- Learn about thread synchronization

Outline

- Multi-tasking, concurrency, and parallel processing
- std::thread
- std::mutex
- std::lock_guard
- std::atomic
- Thread synchronization

Multi-tasking, Concurrency, and Parallel Computing

- What is multi-tasking?
  - Multi-tasking is one of the main functionalities supported by an operating system. It allows the **quasi-parallel** execution of multiple **processes**



Source: Wikipedia

Multi-tasking, Concurrency, and Parallel Computing

- What is concurrency?
  - More general
  - Execution of several **computations** at overlapping times
  - Concurrency can happen at the level of:
    - Network (cloud computing)
    - Computer / OS (multi-tasking, multiple **processes**)
    - Program (multiple **threads**)

Multi-tasking, Concurrency, and Parallel Computing

- What is parallel computing?
  - Strictly parallel execution of (possibly same) computations
  - Requires parallel computing hardware
    - Multi-core processor
    - Graphics Processing Unit (GPU)
    - Field Programmable Gate Array (FPGA)
    - Derived ASICs
    - Specialized software-programmable SoCs (Ambarella etc.)

A process

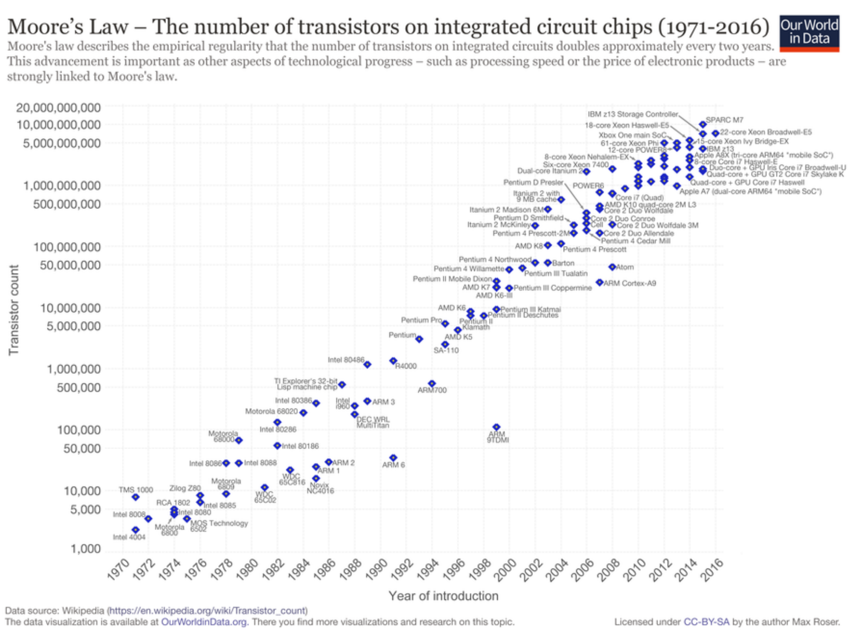
- A process is …
  - … started at the operating system level
  - … assigned a space of individual memory that is typically not shared with other processes
  - … communicating with other processes via other interfaces (network, disk space, etc.)

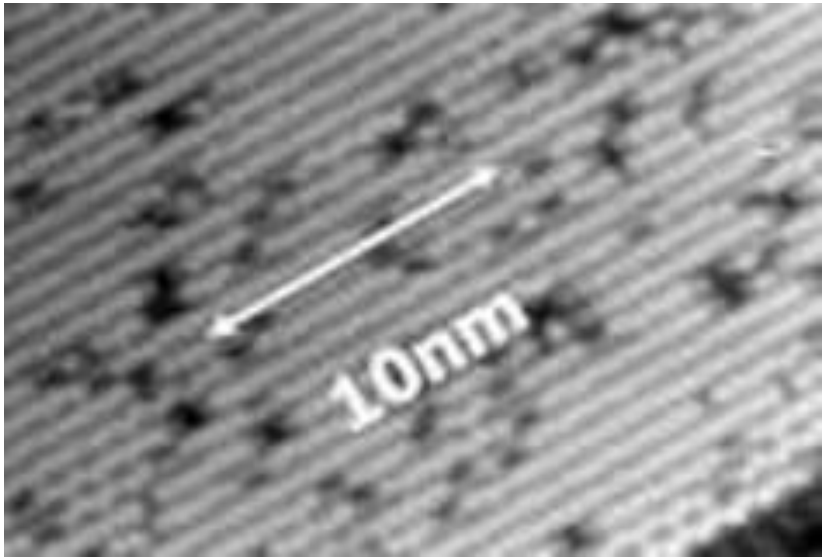
A thread

- A thread is …
  - … started at the process/program level
  - … granted access to the memory space that has been allocated to the process
  - … possibly sharing that memory space with other threads from the same process
  - … able to communicate directly with other threads through the assigned memory
  - … a means to realize parallel computing

Why is concurrency necessary?

- Moore's law is dying
  - Currently ~10nm process technology
    (distance refers to half the distance between identical features in array)
  - That's about 20 silicon atoms!





Why is concurrency necessary?

- Size of gate/transistor limits the processor clock!
  - Higher frequency -> More dissipated heat!
  - Smaller transistors ”can’t take it”

- → We want ever more powerful computing resources
- → Concurrency!

Why is concurrency possible?

- Tasks can often be naturally divided into multiple (often simpler) sub-tasks
  - Divide and Conquer
- Many problems are embarrassingly parallel in their nature
  - Matrix manipulations
  - Image processing
  - …

Outline

- Multi-tasking, concurrency, and parallel processing
- std::thread
- std::mutex
- std::lock_guard
- std::atomic
- Thread synchronization

Concurrency in C++

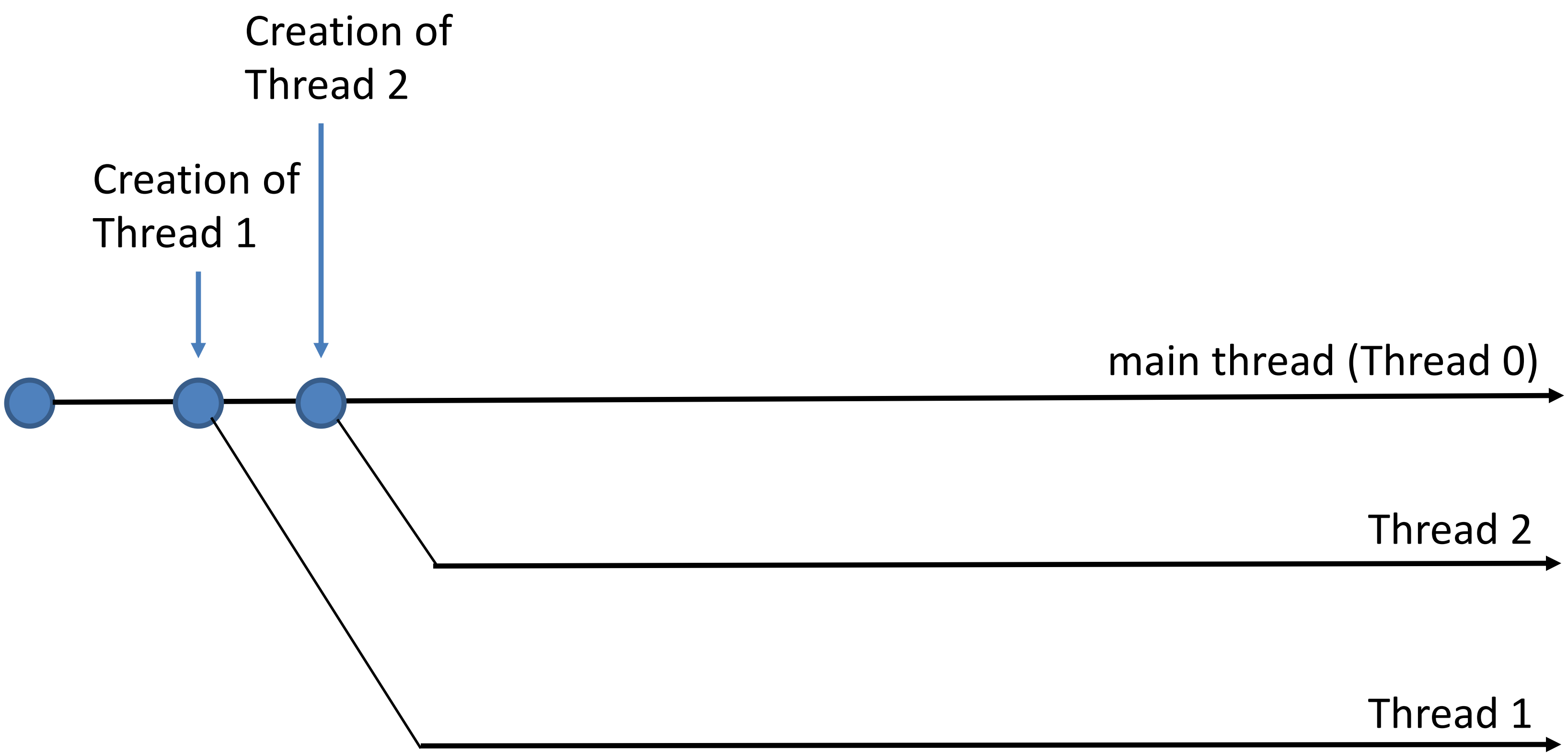
- Original C++ Standard (1998) only supported single thread programming
  - Requirement for other libraries (e.g. pthread) to access POSIX threads functionality

- Since C++ 11:
  - Acknowledges the existence of multi-threaded programs
  - Provides interface to create/synchronize threads
  - Introduces memory models for concurrency

Requirements

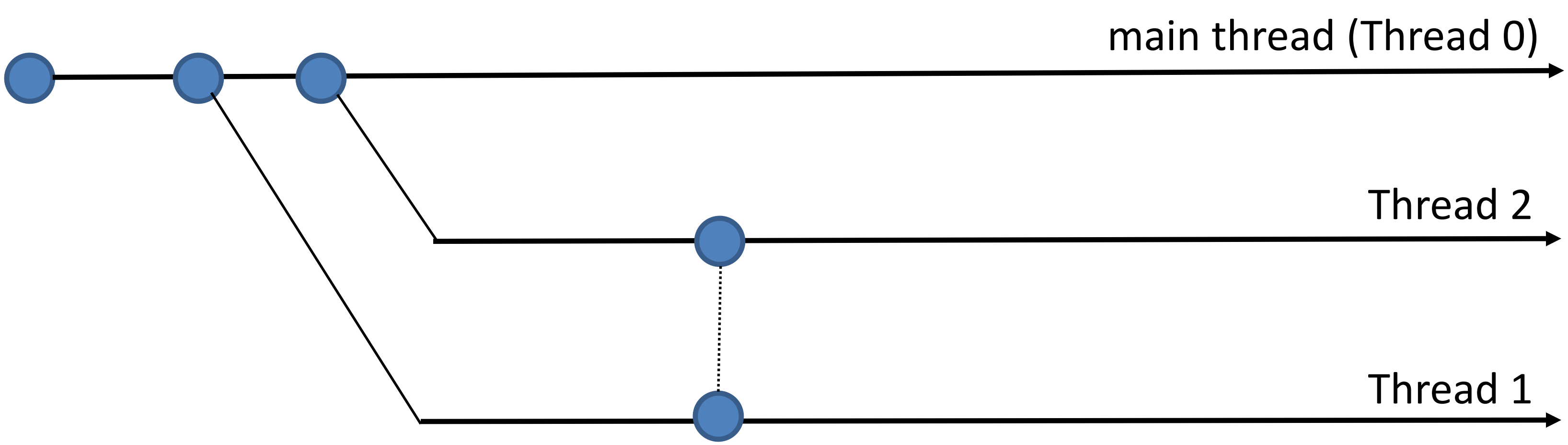
Requirements

- Thread creation
    - Ability to start a new thread (i.e. from the main thread)



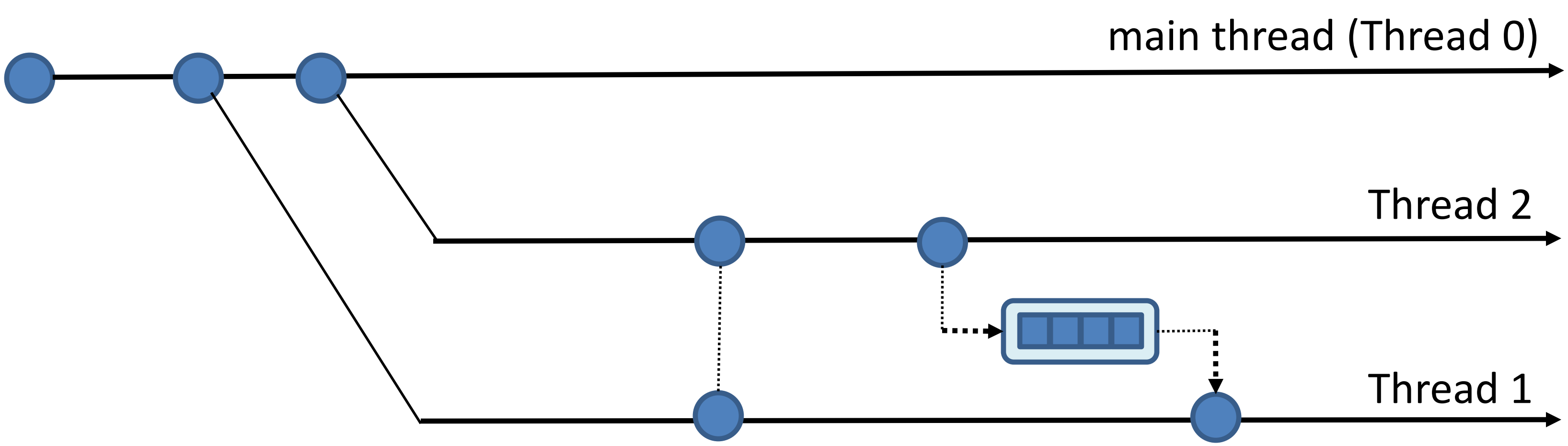
Requirements

- Thread sychronization
  - Ability to establish timing relationships between threads
    - Example: One thread waits until another thread has reached a certain point in its code
    - Example: One thread is ready to transmit information while the other is ready to receive the message, simultaneously



Requirements

- Thread communication
  - Ability to correctly transmit data among different threads



Thread creation

- Use C++ 11!
- Use thread library

```
#include <thread>
```

- Creating an instance of std::thread automatically starts a new thread

```
std::thread th( threadFunction );
```

Thread creation

- Example:

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

void threadFunction() {
        std::cout << "Hello from thread 1\n";
}

int main() {
        std::thread th(threadFunction);
        std::cout << "Hello from thread 0\n";
        th.join();
}
```

Thread creation

- Example:

```cpp
#include <thread>
#include <iostream>

void threadFunction() {
        std::cout << "Hello from thread 1\n";
}

int main() {
        std::thread th(threadFunction);
        std::cout << "Hello from thread 0\n";
        th.join();
}
```

Thread creation

Thread joining

- Main thread waits for other threads to finish!

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

void threadFunction() {
        std::cout << "Hello from thread 1\n";
}

int main() {
        std::thread th(threadFunction);
        std::cout << "Hello from thread 0\n";
        th.join();
}
```

Thread joining

Outline

- Multi-tasking, concurrency, and parallel processing
- std::thread
- std::mutex
- std::lock_guard
- std::atomic
- Thread synchronization

What are critical sections?

- Data is usually shared between threads
- Problem:
  - Multiple threads access the same object at the same time
  - If operation is atomic (i.e. not divisible)
    - No other thread could read/operate on a partial result
    - It is safe!
  - If operation is not atomic (i.e. divisible into several steps)
    - Other threads could read/operate on partial result if switching happens in between
    - It is not safe!

Example

- 5 threads increase the same counter 5000 times

```
#include <vector>
#include <thread>
#include <iostream>

class Counter {
public:
  Counter() {
    m_value = 0;
  };

  int getValue() {
    return m_value;
  };
  void increment() {
    ++m_value;
  };
private:
  int m_value;
};
```

```
void incrementCounterManyTimes(
    Counter & counter ) {
  for( int i = 0; i < 5000; i++ ) {
    counter.increment();
  }
}

int main() {
  Counter counter;
  std::vector<std::thread> threads;
  for( int i = 0; i < 5; i++ ) {
    threads.push_back( std::thread(
        incrementCounterManyTimes,
        std::ref(counter) ) );
  }
  for( int i = 0; i < 5; i++ ) {
    threads[i].join();
  }
  std::cout << counter.getValue() << "\n";
  return 0;
}
```

Reference needed!

Example

- Result
  - Program has synchronization issues!
  - Possible outputs
    - 9840, 6102, 11952, 8740, 10515, 11635, 8490, 15170, 7202, 6218
    - The output is different every time!
  - What has happened?
    - ”increment” is not an atomic operation!
      - It first reads the value
      - It adds one
      - Then copies the result back

Thread switching at either of these points will cause a **data race!**

What are critical sections?

- Critical section
  - A piece of code that accesses/modifies a shared resource, and the access/modification is non-atomic
  - → Access must not be concurrent!
  - → Simultaneous access by multiple threads must be prevented
  - → Access needs to be **mutually exclusive**!

How to protect shared data?

- Solutions
  - Semaphores
  - Mutexes (binary semaphores)
  - Monitors (guarantees only one time can be active within a monitor at a time (supported in Java)
  - Condition variables
  - Compare-and-wrap: The idea is to compare the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory location to a given new value
  - etc.

Mutex

- Mutexes (named after <u>mutual exclusion</u>) enable us to
  - mark critical sections as mutually exclusive
  - if any thread enters that critical section, any other thread that tries to enter a critical section that is marked by the same mutex has to wait!

Mutex

- How does it work?
  - Create a mutex by creating an instance of **`std::mutex`**
  - Lock it with a call to the member function **`lock()`**
  - Unlock it with a call to the member function **`unlock()`**

```
class Counter {
public:
  Counter() { m_value = 0; };

  int getValue() { return m_value; };
  void increment() {
    m_mutex.lock();
    ++m_value;
    m_mutex.unlock();
  };

private:
  int m_value;
  std::mutex m_mutex;
};
```

Output:
25000 every time!

Mutex

- How does it work?
  - A mutex does not directly lock a part of the code
  - A mutex is a resource (i.e. a lock), and we use it passively to protect a critical section
  - **`lock()`** is blocking until it “has the lock”
  - **`unlock()`** ”releases the lock”
  - Only one at a time can have the lock
  - → Bound all critical sections (w.r.t. to the same data) by the same mutex

Outline

- Multi-tasking, concurrency, and parallel processing
- std::thread
- std::mutex
- std::lock_guard
- std::atomic
- Thread synchronization

Problems with Mutex

- It is not wise to call **`lock()`** directly
  - You have to remember to **`unlock()`** on every code path out of a critical section
    (including those due to exceptions)

- Use **`std::lock_guard`**

std::lock_guard

- Example:

```
void increment() {
  std::lock_guard<std::mutex> lock(m_mutex);
  ++m_value;
};
```

- `m_mutex.lock()` is called when the instance is constructed
- `m_mutex.unlock()` is called when the instance is destructed
- → `lock_guard` locks the mutex for the duration of the scope in which `lock_guard` is defined

std::lock_guard

- Also called the RAII idiom
  (resource acquisition is initialization)