Introduction to Concepts in Multi-Threading

Vakho Tsulaia (LBL)

*Based on the presentation given by **Charles Leggett (LBL)** in September 2015 edition of the ATLAS Software Development Tutorial*

ATLAS Software Development Tutorial
CERN, September 23-27, 2019

What are threads?

- Mechanism to perform multiple tasks simultaneously within a single program
- Each task exists within the context of the same process
  - *cf.* multi-processing, where each task exists within a separate process
- Program is split into two or more parts:
  - **Process**: contains resources shared across the whole program
    - Program instructions
    - Global data
    - Resources
  - **Thread(s)**: contains information related to the execution state
    - Program counter
    - Stack

Multi-threaded program

- Lightweight: fewer system resources used when spawning new thread compared to forking new process
- Easy to share data and communicate between threads
  - No need in Inter Process Communication mechanisms (pipes, shared memory, sockets, etc)
- Easy to screw up data sharing
  - Deadlocks, data races, non-deterministic behavior
- Control of threads is difficult
  - Hard to stop a “bad” thread

Single vs Multiple Threads

- There must be a really good reason for turning a solid single threaded program into a multi threaded one

- Multi threaded code is much harder to understand
  - There is a much higher cost in maintaining
  - Bugs are much harder to find, and may only appear under unpredictable circumstances

- Each thread uses resources
  - Duplicate stack, scheduling, kernel resources, etc.

- Context switching takes time
  - If the program keeps increasing the number of threads, at some point more time will be spent by the system switching between threads than actually executing them

Thread libraries

- Posix threads (`pthread`)
  - The original
- Boost (`boost::thread`)
  - Based on pthread with few extra functions (e.g. futures)
- Intel Thread Building Blocks (TBB)
  - More task oriented
- C++ threads (`std::thread`)
  - Similar to Boost threads, plus atomics

Simple threaded program

- The example is based on C++ threads
- Start a thread by creating a `std::thread` object and initialize it with a function
  - The thread will execute this function and finish
- Wait for all threads to finish their tasks

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

using namespace std;

void do_one_thing(int *);
void do_another_thing(int *);
void wrap_up(int,int);

int r1=0, r2=0;

main () {

  std::thread t1(do_one_thing, &r1);
  std::thread t2(do_another_thing, &r2);

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

  wrap_up(r1,r2);

}
```

Thread Safety

- Functions that can be called from multiple threads without destructive/unreproducible behavior are called **Thread Safe**
- Use of global variables (external or static), or static local variables, makes a function **Thread Unsafe**
- Make threads safe by
  - Surrounding critical functions and data with **Locks**
- Functions can be thread safe, but not yet **Reentrant**
  - Reentrancy guarantees consistent behavior if the function can be interrupted in the middle of its execution and safely called again (e.g. **recursive function**)

Protecting shared data

- If multiple threads modify same memory location simultaneously, unpredictable results occur
  - **Race conditions:** often load dependent. Makes it very challenging to debug, as problems often disappear when running with debugger, as timing is slowed down

- Race conditions are **major source of bugs** in multithreaded programs

- Various ways of avoiding race conditions
  - **Mutex**: blocks another thread from modifying a piece of memory while current thread is working on it
  - **Lock free programming**: modifications to data is done as a series of indivisible changes, preserving invariants
  - **Transactions**: data modifications are stored in a transaction log, committed as a single step.

Examples of data races

- **Simple race**

```
int var;

void Thread1() {  // Runs in one thread.
  var++;
}
void Thread2() {  // Runs in another thread.
  var++;
}
```

- **Thread-hostile reference counting**

```
// Ref() and Unref() may be called from several threads.
// Last Unref() destroys the object.
class RefCountedObject {
 ...
 public:
  void Ref() {
    ref_++;  // Bug!
  }
  void Unref() {
    if (--ref_ == 0)  // Bug! Need to use atomic decrement!
      delete this;
  }
 private:
  int ref_;
};
```

Mutex

- **Mut**ual **ex**clusion
- Must be locked and unlocked

```
#include <mutex>

std::mutex list_mutex;

void add_to_list(int new_value) {
  list_mutex.lock();
  some_list.push_back(new_value)
  list_mutex.unlock();
}
```

- Very rarely used by themselves
  - Better to use wrappers (e.g. `std::lock_guard`)

Mutex is not a panacea

```
class Data {
   int a;
   std::string b;
 public:
   void do_something();
};

class data_wrapper {
 private:
   Data data;
   std::mutex m;
 public:
   template<typename Function>
   void process_data(Function func) {
     std::lock_guard<std::mutex> l(m);
     func(data);
   }
};
```

- Calling user functions on protected data can also be dangerous. The function can leak internals!
- Don’t pass pointers and references to protected data outside the scope of the lock!

```
Data* unprotected;

void malicious_function(Data& protected_data) {
  unprotected = &protected_data;
}

data_wrapper x;

void foo() {
  x.process_data(malicious_function);
  unprotected->do_something();
}
```

More about mutexes

- **Deadlock**
  - When two or more threads need multiple locks to perform an operation
  - If one thread holds **mutex A** and needs to acquire **mutex B**, while another thread holds **mutex B** and needs to acquire **mutex A**, none can progress
  - Avoid this by always locking mutexes in the same order

- **Lock granularity**
  - Try to lock at smallest appropriate granularity (i.e. don't lock the entire tree when you can lock nodes)
  - On the other hand, locking at too fine granularity can cause problems too
  - Only hold a lock for a minimum possible time to perform an operation
  - Don't do file I/O when locked

Thread-local variables

- If a data object is shared between threads, it can sometimes be very useful to have separate instances of one of the objects' variables for each thread
  - Declared with keyword `thread_local`
  - Scope:
    - Namespace / File
    - Static data members of classes
    - Local variables

```
// Namespace scope
thread_local int x;

// Static class data member
class X {
   static thread_local std::string s;
};

// Local variable
void foo() {
 static thread_local std::vector<int> v;
}
```

Some resources

- This talk only scratched the surface

- Online resources
  - http://www.cpluplus.com
  - http://www.isocpp.com

- Books
  - ***C++ Concurrency in Action*** by Anthony Williams
  - ***pthreads Programming*** by Bradford Nichols