Concurrency in C/C++

10/17/17
Topic Overview

- What’s a thread?
- Creating threads in C/C++
- What’s concurrency?
- Concurrency example
- Race conditions
- Synchronization and locks
- Locking example
- Advanced synchronization primitives
What’s a thread?

When starting a program, a process is created.

Every process has a unique set of virtual memory.

The program’s code is stored somewhere in this memory.

A process consists of one or more threads that run the code.

Every thread in a process accesses the same memory while running.

When changing the value of a global variable in one thread, all threads then see the new value.
Single-threaded vs. Multi-threaded

Every process has at least one thread. This is created when the program starts.

The first thread is the main thread. It runs main().

A thread can create new threads. These are child threads.

Child threads can run arbitrary functions (with some restrictions).

A process with only one thread is called a single-threaded process.

A process with more than one thread is called a multi-threaded process.
Single-threaded vs. Multi-threaded

http://www.pling.org.uk/cs/opsimg/threadmodel.png
Creating threads in C/C++

Most system calls use an object called **pthreads**, or “process threads”.

C++ has the std::thread library, which is a wrapper to pthreads.

Both of these libraries are suitable for Project 1.
Creating a pthread

```c
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void **(start_routine) (void *), void *arg);
```

- **thread**: The object where information about the new thread is stored. Used in other pthread system calls.
- **attr**: Used to set advanced attributes on the thread, like core pinning. For Project 1, just use `NULL` (default attributes).
- **start_routine**: the function the thread will run. The function’s type must be `void* some_func(void *arg) {}`.
- **arg**: The argument passed to `start_routine`. 
Joining a pthread

To wait until a thread is finished and get its return value, **join** with the thread:

```c
int pthread_join(pthread_t thread, void **retval);
```

- **thread**: The `pthread_t` object given to `pthread_create` earlier.
- **retval**: A pointer to where the return value of the thread will be stored.
Code example

See `thread_creation/main.c`
What’s concurrency?

All threads in a process run at the same time, or **concurrently**.

**Concurrency** is using multiple threads to accomplish disjoint tasks in a program.

A **concurrent web server** is a web server that handles multiple clients simultaneously using threads.
Concurrency example

See concurrency/main_nolocks.cpp
Concurrency example

The final value of `counter` should be zero.

Run the program multiple times. What happens?
Race conditions

How is the value different every time? Each thread was only repeating one line of code the same number of times!

Simple lines of code often require several assembly instructions to implement.

Strange behavior in concurrent programs is usually caused by a race condition.

A race condition is when multiple threads try to access the same data simultaneously. The final result of a race condition is unpredictable.
Race conditions

Remember that threads run concurrently.

This means that a thread could execute *any* number of CPU instructions between only *one* instruction of any other thread!

What is the actual assembly code for `counter++`?
Race conditions

**Incrementer Thread**

- LW $3, 0(counter)
- ADDIU $3, $3, 1
- SW $3, 0(counter)

... 

**Decrementer Thread**

- LW $3, 0($counter)
- ADDI $3, $3, -1
- SW $3, 0($counter)

... 

$3

??

counter

0

??
Race conditions

**Incrementer Thread**

LW $3, 0(counter)
ADDIU $3, $3, 1
SW $3, 0(counter)

... $3 ??

**Decrementer Thread**

LW $3, 0($counter)
ADDI $3, $3, -1
SW $3, 0($counter)

... $3 0
counter 0
Race conditions

Incrementer Thread

LW $3, 0(counter)
ADDIU $3, $3, 1
SW $3, 0(counter)
...

Decrementer Thread

LW $3, 0($counter)
ADDI $3, $3, -1
SW $3, 0($counter)
...
Race conditions

**Incrementer Thread**

LW $3, 0(counter)
ADDIU $3, $3, 1
SW $3, 0(counter)
...

$3

1

**Decrementer Thread**

LW $3, 0($counter)
ADDI $3, $3, -1
SW $3, 0($counter)
...

$3

0
Race conditions

**Incrementer Thread**

LW $3, 0(counter)
ADDIU $3, $3, 1
SW $3, 0(counter)

\[ \text{\ldots} \]

**Decrementer Thread**

LW $3, 0($counter)
ADDI $3, $3, -1
SW $3, 0($counter)

\[ \text{\ldots} \]
Race conditions

**Incrementer Thread**

LW $3, 0(counter)
ADDIU $3, $3, 1
SW $3, 0(counter)

...  

$3 ??

**Decrementer Thread**

LW $3, 0($counter)
ADDI $3, $3, -1
SW $3, 0($counter)

...  

$3 -1

counter 1
Race conditions

Incrementer Thread

LW $3, 0(counter)
ADDIU $3, $3, 1
SW $3, 0(counter)

Decrementer Thread

LW $3, 0($counter)
ADDI $3, $3, -1
SW $3, 0($counter)
Race conditions

Incrementer Thread

LW $3, 0(counter)
ADDIU $3, $3, 1
SW $3, 0(counter)
...

$3 ??

The value of the counter is no longer 0 at the start of the loop!

Decrementer Thread

LW $3, 0($counter)
ADDI $3, $3, -1
SW $3, 0($counter)
...

$3 ??

counter -1

This is just one of the possible outcomes.
Race conditions

This is just one of several outcomes that could occur in reality.

That’s why the output of the program changes every time it’s run!

Remember: you can’t predict how concurrent threads will behave.
Synchronization and locks

To avoid race conditions, we need to use synchronization.

**Synchronization** uses special objects to make threads work in unison.

These objects are called **synchronization primitives**.

These primitives have **invariants** that make it possible to avoid race conditions.

The simplest object is a **lock**, and is usually called a **mutex** (“mutual exclusion”).
Mutexes

A mutex has two states: locked, and unlocked.

A thread that has locked a mutex is said to be **holding the lock**.

A mutex has certain rules, or **invariants**:

1. Only one thread hold a lock at a time.
2. A mutex can only be locked if it is currently unlocked.
3. Only the thread holding the lock can unlock the mutex.
4. A thread trying to lock an already locked mutex must wait until it is unlocked.

This prevents other threads from modifying objects at the same time.
Locking example

See concurrency/main-locks.cpp

C also has mutexes; see pthread_mutex.

Notice that the value is now always 0!
Advanced synchronization primitives

There are two other widely used synchronization primitives.

**Semaphores** have a counter that starts at some counter. Threads can increase the counter for free, or decrease it only if it is above zero.

**Condition variables** allow threads to explicitly signal they are waiting for something, and will go to sleep. Another thread can wake up one or more waiting threads.