COMPUTATIONAL MATH, SCIENCE AND ENGINEERING DEPARTMENT

MICHIGAN STATE UNIVERSITY

Concurrency1

The big problem

The problem is the writing to a shared data structure by multiple writers/threads.

If all the data structures are read-only, there is no problem with concurrency.

Concurrent writes are the problem

synchronization

We need to synchronize access to avoid race conditions and to get the results we expect.

We will examine shared memory sync in these sections.

race condition

A race condition is essentially an indeterminacy on a piece of data.

If we cannot know deterministically what a read will provide giving multiple writers, then what we read is a race between multiple writers.

```
int main (){
  char c;
  long ms_since_epoch = std::chrono::system_clock::now().time_since_epoch() / std::chrono::milliseconds(1);
  thread t1(thread_fun, 3, ms_since_epoch, '*');
  ms_since_epoch = std::chrono::system_clock::now().time_since_epoch()  / std::chrono::milliseconds(1);
  thread t2(thread_fun, -3, ms_since_epoch, '-');
  c=getchar();
  stop = true;
  t1.join();
  t2.join();
  cout << "Global result is:"
<<global_var << endl;
}
```

```
long global_var = 0;
bool stop = false;

void thread_fun(long inc, int seed, char c){
  mt19937_64 reng(seed);
  uniform_int_distribution<long> dist;
  decltype(dist.param()) small(100,500);
  decltype(dist.param()) large(600,1500);
  if (inc > 0)
    dist.param(small);
  else
    dist.param(large);

  while(!stop){
    cout << c;
    global_var += inc;
    cout << "Global is now:"<<global_var<<endl<<flush;
    std::this_thread::sleep_for(std::chrono::milliseconds(dist(reng)));
```

your basic stack

```
stack<int> s;
if (!s.empty() )){
    int const value = s.top();
    s.pop();
    my_fun(value);
}
```

possible ordering

Table 3.1 A possible ordering of operations on a stack from two threads

Thread A	Thread B
`if(!s.empty())`	
	`if(!s.empty())`
`int const value=s.top();`	
	`int const value=s.top();`
`s.pop();`	
`do_something(value);`	`s.pop();`
	`do_something(value);`

This is sneaky hard

This kind of problem is sneaky, hard to see as you write the algorithm and hard to find as you debug.

It requires a different way of thinking about the problem.

what can go wrong

- unsynchronized access
  - which of multiple statements goes first/last
- half written data
  - the underlying operations might not have completed
- reordered statements
  - each statement might be OK, but their order for the overall process is off

how to solve

Two broad categories of how to solve the problem

- atomicity
  - an operation happens as a single unit, an atom, such that only 1 thread can use it
- order
  - use programmer elements to enforce order

how to address

- critical sections, programmatically gate a section of code
  - mutexs, lock_guard,
- atomic types
  - use operations that are guaranteed to be indivisible, only one thread at a time.
- barriers (C++20)
  - hold all threads until all arrive at this point in their execution

how to address(2)

- conditions, have thread wait on some condition
  - condition_variable

Note, design is really important!

- We can use these tools, but things can still go wrong because we did a bad design
- We'll see this later

mutual exclusion

critical section

A critical section is a piece of code that accesses a piece of shared resource and therefore must be protected from concurrent access

mutex

A `mutex` is essentially a way to serialize access to a critical section. It has two basic methods (plus more of course)

- `.lock()` : Lock the mutex. If successful the locking thread has access. If not, thread waits (blocks) to access the section (is queued).
- `.unlock()` : remove block, let another thread in

mutex is shared

A mutex is a shared object among locks. This is required so that the behavior is what we expect (one thread in at a time).

tricker than it looks

What if you forget to unlock?

- or an exception occurs before you unlock

What if you need to get two locks?

- if each thread has one of the two needed locks, you get deadlock

C++ cannot get around bad design

But it can help to some extent

There is an approach to dealing with resource problems called RAII

- Resource Acquisition is Initialization
  - Bjarne Stroustroup

RAII

From the Wik:

"Resources are acquired during initialization, when there is no chance of them being used before they are available, and released with the destruction of the same objects, which is guaranteed to take place even in case of errors."

locks

- a lock utilizes a mutex to create a critical section
- the lock can be a local variable to the thread application, but the mutex must be shared.

lock_guard

A `lock_guard` is an RAII approach to mutex.

It takes a `mutex` as an object and, when created (when constructed), it locks the `mutex`.

When the `lock_guard` is destroyed, before it is gone it unlocks the mutex

3.4

```
#include <future>
#include <mutex>
#include <iostream>
#include <string>

std::mutex printMutex;  // enable synchronized output with print()

void print (const std::string& s)
{
    std::lock_guard<std::mutex> l(printMutex);
    for (char c : s) {
        std::cout.put(c);
    }
    std::cout << std::endl;
}

int main(){
    auto f1 = std::async (std::launch::async,
                          print, "Hello from a first thread");
    auto f2 = std::async (std::launch::async,
                          print, "Hello from a second thread");
    print("Hello from the main thread");
}
```

scope of the lock_guard

mutex locked here

unlocked when lock_guard out of scope(destroyed)

lock is not enough

Just because you lock the data does not mean that things can't go wrong.

- pointers and references to shared data can get around a `mutex`
- passing functions around can do the same thing

What gets locked?

It is important to think about what a mutex, a mutual exclusion of whatever kind, locks.

It locks a piece of code, not memory!

If you want to establish mutual exclusion then it must be the case that all code access to a data item go through the same mutual exclusion

This is a programmatic issue.

```
class some_data{
   int a;
   std::string b;
public:
   void do_something(){
     std::cout << "Doing something"
                  <<std::endl;
   }
};

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

```
some_data* unprotected; // global pointer to a some_data
data_wrapper x;  // global data_wrapper

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

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

int main(){
   foo();
}
```

The parallel programming tension

- To avoid things like races and deadlocks we can serialize with locks, barriers and atomics
- To go fast we want to avoid serialization

Hard to get the balance right.

Coarse vs Fine

Where do we put locks in a program? And how many locks should there be? These questions have motivated the designs of several different locks and synchronization mechanisms.
The most basic choice is between having *few coarse-grained locks* and *many fine-grained locks*.

Where?

Few coarse-grained locks
(1 lock protects many resources)

+ Correctness is easier (with only one lock, there's less chance of grabbing the wrong lock, and less risk of deadlock)
– Performance is lower (not much concurrency)

Many fine-grained locks
(1 lock protects a small number of resources)

+ Good concurrency/parallelism = good performance
– Correctness is harder (it's easier to make a mistake and forget to grab the lock required to access a resource)
– Higher overhead from having many locks