Deadlock

(And how you will likely encounter it)

Deadlock

- Classic Deadlock Definition
    - 2 threads- each have a resource that the other wants

```
mutex m1;
mutex m2;
void f1(int i){
    while(i>0){
        lock_guard<mutex> lg1(m1);
        lock_guard<mutex> lg2(m2);
    }
}

void f2(int i){
    while(i>0){
        lock_guard<mutex> lg1(m2);  //mutexes aquired out of order
        lock_guard<mutex> lg2(m1);
    }
}

int main() {
    //the threaded way with 2 mutexes
    thread t1(f1, NUMB_TIMES);
    thread t2(f2, NUMB_TIMES);
    t1.join();
    t2.join();
}
```

Deadlock

- Classic Deadlock Definition
  - 2 threads- each have a resource that the other wants

```
mutex m1;
mutex m2;
void f1(int i){
    while(i>0){
        lock_guard<mutex> lg1(m1);
        lock_guard<mutex> lg2(m2);
    }
}

void f2(int i){
    while(i>0){
        lock_guard<mutex> lg1(m2);  //mutexes aquired out of order
        lock_guard<mutex> lg2(m1);
    }
}

int main() {
    //the threaded way with 2 mutexes
    thread t1(f1, NUMB_TIMES);
    thread t2(f2, NUMB_TIMES);
    t1.join();
    t2.join();
}
```

If t1 is interrupted here

Deadlock

- Classic Deadlock Definition
  - 2 threads- each have a resource that the other wants

```
mutex m1;
mutex m2;
void f1(int i){
    while(i>0){
        lock_guard<mutex> lg1(m1);
        lock_guard<mutex> lg2(m2);
    }
}

void f2(int i){
    while(i>0){
        lock_guard<mutex> lg1(m2);  //mutexes aquired out of order
        lock_guard<mutex> lg2(m1);
    }
}

int main() {
    //the threaded way with 2 mutexes
    thread t1(f1, NUMB_TIMES);
    thread t2(f2, NUMB_TIMES);
    t1.join();
    t2.join();
}
```

If t1 is interrupted here

Then t2 runs and is interrupted here

Deadlock

- Classic Deadlock Definition
  - 2 threads- each have a resource that the other wants

```
mutex m1;
mutex m2;
void f1(int i){
    while(i>0){
        lock_guard<mutex> lg1(m1);
        lock_guard<mutex> lg2(m2);
    }
}

void f2(int i){
    while(i>0){
        lock_guard<mutex> lg1(m2);  //mutexes aquired out of order
        lock_guard<mutex> lg2(m1);
    }
}

int main() {
    //the threaded way with 2 mutexes
    thread t1(f1, NUMB_TIMES);
    thread t2(f2, NUMB_TIMES);
    t1.join();
    t2.join();
}
```

If t1 is interrupted here

Then t2 runs and is interrupted here

Then the program will stop making forward progress

Deadlock

- Classic Deadlock Definition
  - 2 threads- each have a resource that the other wants

```
mutex m1;
mutex m2;
void f1(int i){
    while(i>0){
        lock_guard<mutex> lg1(m1);
        lock_guard<mutex> lg2(m2);
    }
}

void f2(int i){
    while(i>0){
        lock_guard<mutex> lg1(m2);  //mutexes aquired out of order
        lock_guard<mutex> lg2(m1);
    }
}

int main() {
    //the threaded way with 2 mutexes
    thread t1(f1, NUMB_TIMES);
    thread t2(f2, NUMB_TIMES);
    t1.join();
    t2.join();
}
```

If t1 is interrupted here

Then t2 runs and is interrupted here

Then the program will stop making forward progress

Demo: simple_deadlock project

Deadlock

- Classic Deadlock Definition
  - 2 threads- each have a resource that the other wants

```
mutex m1;
mutex m2;
void f1(int i){
    while(i>0){
        lock_guard<mutex> lg1(m1);
        lock_guard<mutex> lg2(m2);
    }
}

void f2(int i){
    while(i>0){
        lock_guard<mutex> lg1(m2);  //mutexes aquired out of order
        lock_guard<mutex> lg2(m1);
    }
}

int main() {
    //the threaded way with 2 mutexes
    thread t1(f1, NUMB_TIMES);
    thread t2(f2, NUMB_TIMES);
    t1.join();
    t2.join();
}
```

If t1 is interrupted here

Then t2 runs and is interrupted here

Then the program will stop making forward progress

Demo: simple_deadlock project

To fix: always acquire lock objects in same order

Deadlock- Possible Solution

```
//the prevention lock is used to protect
//critical sections where locks are being aquired
prevention.lock();

//then aquire all necessary locks
//knowing that no other thread can be
//in a critical section protected by
//the prevention lock
L1.lock();
l2.lock();

prevention.unlock()


//grab this lock first
prevention.lock();

l2.lock();  //ordering does not matter now
L1.lock();  //given the prevention lock

prevention.unlock()
```

- Put all lock acquisition in critical sections
- Good
  - Cannot be interrupted while acquiring locks
- Bad
  - Additional lock (prevention) to manage
  - Have to know ahead of time what locks we need
  - Decreasses concurrency as we are likely acquiring locks early. Critical sections are larger than needed

Deadlock- Possible Solution 2

```
mutex m1;
mutex m2;
void f1(int i){
    while(i>0){
        lock_guard<mutex> lg1(m1);
        lock_guard<mutex> lg2(m2);
    }
}

void f2(int i){
    while(i>0){
        lock_guard<mutex> lg2(m1);
        lock_guard<mutex> lg1(m2);
    }
}

int main() {
    //the threaded way with 2 mutexes
    thread t1(f1, NUMB_TIMES);
    thread t2(f2, NUMB_TIMES);
    t1.join();
    t2.join();
}
```

- Always acquire locks in same order
- Good
  - Cannot deadlock
- Not as good
  - Locking is not so simple. Locks are spread out amongst classes, functions and libraries with many conditional statements.
  - Its often hard to predict what the ordering will be.
- But probably about as good as it will get

Things that act like deadlock

```
mutex m;
void fun2(){
    m.lock();
}
void fun1(){
    m.lock();
    fun2();
}
int main() {
    // superficial blocking? Its actually undefined
    //from the C++ 11 standard
    //30.4.1.2.1/4 [Note: A program may deadlock if the
    //thread that owns a mutex object calls lock() on that object.]
    //it MAY, or it may not block.  It may work on 1 compiler and
    //not another
    m.lock();
    m.lock();

    //the way locking twice without
    //really happens
    fun1();
}
```

This causes thread to block and wait to acquire a lock that it already owns

- Locking twice on the same thread without an intervening unlock

Things that act like deadlock

```
mutex m;
void fun2(){
    m.lock();
}
void fun1(){
    m.lock();
    fun2();
}
int main() {
    // superficial blocking? Its actually undefined
    //from the C++ 11 standard
    //30.4.1.2.1/4 [Note: A program may deadlock if the
    //thread that owns a mutex object calls lock() on that object.]
    //it MAY, or it may not block.  It may work on 1 compiler and
    //not another
    m.lock();
    m.lock();

    //the way locking twice without
    //really happens
    fun1();
}
```

This is how it happens in real code

- Locking twice on the same thread without an intervening unlock

Demo: simple_deadlock project

Deadlock – the real world

- Locking is not so simple, especially if you have more than 1 lock.
- Locks are spread out amongst classes, functions and libraries with many conditional statements
- Its often hard to see what the lock ordering will be
- Once a program deadlocks it stops, it does not consume processor cycles, or any more memory than it had when the deadlock occurred. But it will never exit. It must be killed and restarted.