

Threads

Race conditions

Atomic Variables

Critical sections

BTW

- Read the week 10 advice on course website

Race Conditions

- Launch 2 threads, the outcome depends on which finishes first
 - Spurious, tough to reproduce (may need exacting set of conditions)
 - Because of this non-determinism they are **Tough to debug**

```
//A global int  
int i=0;
```

Each of these instructions
are really 3 assembly instructions

//Thread 1
i++;

//Thread 2
i--;

Race Conditions

- Launch 2 threads, the outcome depends on which finishes first
 - Spurious, tough to reproduce (may need exacting set of conditions)
 - Because of this non-determinism they are **Tough to debug**

```
//A global int  
int i=0;
```

Each of these instructions
are really 3 assembly instructions

The problem is you can't guarantee
that all three will run to completion
without being interrupted

//Thread 1
i++;

//Thread 2
i--;

Race Conditions

- Launch 2 threads, the outcome depends on which finishes first
 - Spurious, tough to reproduce (may need exacting set of conditions)
 - Because of this non-determinism they are **Tough to debug**

```
//A global int  
int i=0;
```

Each of these instructions
are really 3 assembly instructions
The problem is you can't guarantee
that all three will run to completion
without being interrupted.

Want an atomic operation; a sequence
of 1 or more operations that appear
indivisible. No other process can see
an intermediate state, once started
cannot be interrupted

//Thread 1
i++;

//Thread 2
i--;

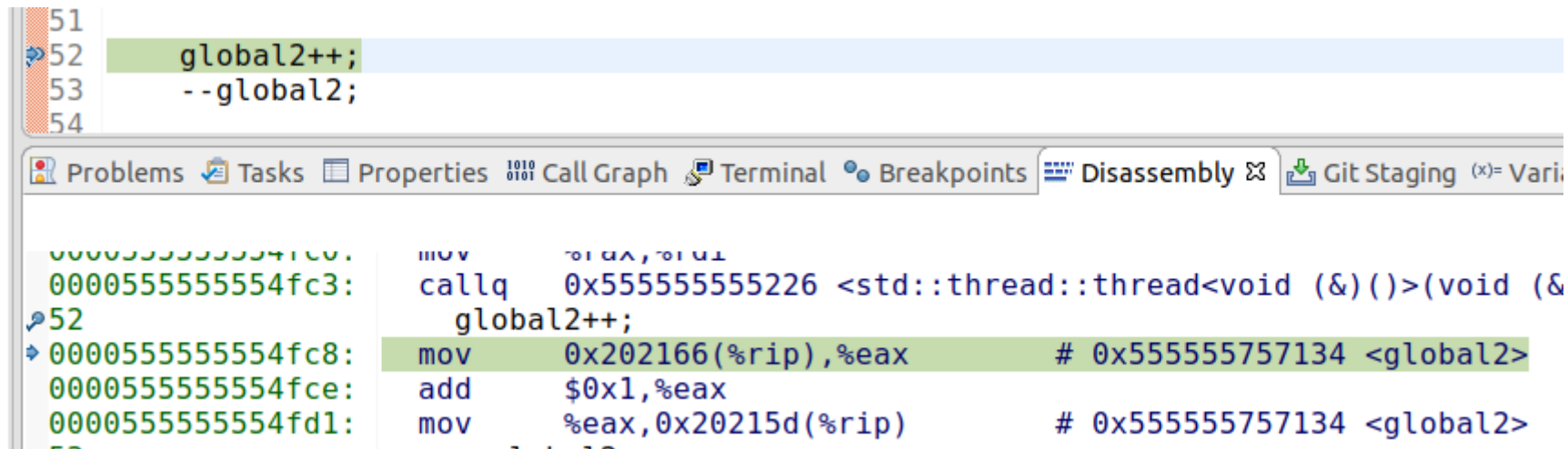
So, is this atomic?

```
global2++;  
--global2;
```

See https://github.com/CNUClasses/thread_problem_atomic_solution.git

No, its 3, interruptible, machine instructions

```
global2++;  
--global2;
```



The screenshot shows a debugger window with two panes. The top pane displays C++ source code with line numbers 51 to 54. Line 52, `global2++;`, is highlighted. The bottom pane shows the corresponding assembly instructions. The instruction at address `000055555554fc8:` is `mov 0x202166(%rip),%eax`, which is highlighted in green. This instruction corresponds to the `global2++;` line in the source code. The assembly pane also shows a `callq` instruction at `000055555554fc3:` and another `mov` instruction at `000055555554fd1:`.

```
51  
52 global2++;  
53 --global2;  
54  
000055555554fc3: callq 0x55555555226 <std::thread::thread<void (&())>(void (&  
52 global2++;  
000055555554fc8: mov 0x202166(%rip),%eax # 0x555555757134 <global2>  
000055555554fce: add $0x1,%eax  
000055555554fd1: mov %eax,0x20215d(%rip) # 0x555555757134 <global2>
```

Go to this project and demo non deterministic behaviour



See https://github.com/CNUClasses/thread_problem_atomic_solution.git

OK make it atomic

Go From this

```
#include <thread>

using namespace std;
const int NUMB_TIMES = 100000;

//global variable
int global2 = 0;
```


OK make it atomic

Go From this

```
#include <thread>

using namespace std;
const int NUMB_TIMES = 100000;

//global variable
int global2 = 0;
```

To this

```
#include <thread>
#include <atomic>

using namespace std;
const int NUMB_TIMES = 100000;

//atomic variable
std::atomic<int> global2(0);
```

OK make it atomic

Go From this

```
#include <thread>

using namespace std;
const int NUMB_TIMES = 100000;

//global variable
int global2 = 0;
```

To this

```
#include <thread>
#include <atomic>

using namespace std;
const int NUMB_TIMES = 100000;

//atomic variable
std::atomic<int> global2(0);
```

Atomic types are types that encapsulate a value whose access is guaranteed to not cause data races and can be used to synchronize memory accesses among different threads.

OK make it atomic

Go From this

```
#include <thread>

using namespace std;
const int NUMB_TIMES = 100000;

//global variable
int global2 = 0;
```

To this

```
#include <thread>
#include <atomic>

using namespace std;
const int NUMB_TIMES = 100000;

//atomic variable
std::atomic<int> global2(0);
```

Go to this project and demo atomic solution



See https://github.com/CNUClasses/thread_problem_atomic_solution.git

But...

- Atomics protect single lines of code only.
- What if you have 3 lines that must be uninterruptable?

But...

- Atomics protect single lines of code only.
- What if you have 3 lines that must be uninterruptable? Like this

```
//starting balance
int bal =50;

void withdrawmoney(int amt){
    if (bal>amt){
        cout<<"approved!"<<endl;
        bal -=amt;
    }
    else
        cout<<"denied!";
}
```

```
int main() {
    thread t1(withdraw, 40);
    thread t2(withdraw, 25);
```

Go to this project to see this code



See https://github.com/CNUClasses/Thread_Race_condition.git


But...

- Atomics protect single lines of code only.
- What if you have 3 lines that must be uninterruptable? Like this

```
//starting balance
int bal =50;

void withdrawmoney(int amt){
    if (bal>amt){
        cout<<"approved!"<<endl;
        bal -=amt;
    }
    else
        cout<<"denied!";
}
```

What happens if you are interrupted
right after the if conditional check



```
int main() {
    thread t1(withdraw, 40);
    thread t2(withdraw, 25);
}
```

But...

- Atomics protect single lines of code only.
- What if you have 3 lines that must be uninterruptable? Like this

```
//starting balance
int bal =50;

void withdrawmoney(int amt){
    if (bal>amt){
        cout<<"approved!"<<endl;
        bal -=amt;
    }
    else
        cout<<"denied!";
}
```

```
int main() {
    thread t1(withdraw, 40);
    thread t2(withdraw, 25);
}
```

What happens if you are interrupted
right after the if conditional check

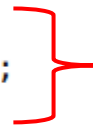
Will not help to make bal an atomic (why?)

But...

- Atomics protect single lines of code only.
- What if you have 3 lines that must be uninterruptable? Like this

```
//starting balance  
int bal =50;
```

```
void withdrawmoney(int amt){  
    if (bal>amt){  
        cout<<"approved!"<<endl;  
        bal -=amt;  
    }  
    else  
        cout<<"denied!";  
}
```



What is needed is to make these three lines uninterruptable

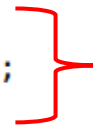
```
int main() {  
    thread t1(withdraw, 40);  
    thread t2(withdraw, 25);  
}
```


But...

- Atomics protect single lines of code only.
- What if you have 3 lines that must be uninterruptable? Like this

```
//starting balance
int bal =50;

void withdrawmoney(int amt){
    if (bal>amt){
        cout<<"approved!"<<endl;
        bal -=amt;
    }
    else
        cout<<"denied!";
}
```



What is needed is to make these three lines uninterruptable
We call this a “critical section”

```
int main() {
    thread t1(withdraw, 40);
    thread t2(withdraw, 25);
}
```

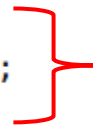
Critical Section: Code that accesses a shared resource, that must complete without interruption. BTW make them as small as possible

But...

- Atomics protect single lines of code only.
- What if you have 3 lines that must be uninterruptable? Like this

```
//starting balance
int bal =50;

void withdrawmoney(int amt){
    if (bal>amt){
        cout<<"approved!"<<endl;
        bal -=amt;
    }
    else
        cout<<"denied!";
}
```



What is needed is to make these three lines uninterruptable
We call this a “critical section”

Critical Section: Code that accesses a shared resource, that must complete without interruption. BTW make them as small as possible

Question: Can you have a critical section in a single threaded environment?

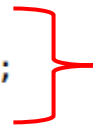
```
int main() {
    thread t1(withdraw, 40);
    thread t2(withdraw, 25);
}
```

But...

- Atomics protect single lines of code only.
- What if you have 3 lines that must be uninterruptable? Like this

```
//starting balance
int bal =50;

void withdrawmoney(int amt){
    if (bal>amt){
        cout<<"approved!"<<endl;
        bal -=amt;
    }
    else
        cout<<"denied!";
}
```



What is needed is to make these three lines uninterruptable
We call this a “critical section”

Critical Section: Code that accesses a shared resource, that must complete without interruption. BTW make them as small as possible

```
int main() {
    thread t1(withdraw, 40);
    thread t2(withdraw, 25);
}
```

Question: Can you have a critical section in a single threaded environment? **NO**

Critical Section

- Critical Section: Code that accesses a shared resource, where only 1 thread can be at a time.
- Make them as small as possible! Why? Because in the critical section you potentially go from a multithreaded application, to a single threaded application where the other threads are blocked waiting to get in.

Where are critical sections in the following code?

```
int g=0;

void fun(){
    g++;
}

int main(){
    thread t1(fun);

    int i=g;

    i++;

    g=i;

    t1.join();
    :
}
```

Where are critical sections in the following code?

```
int g=0;
```

```
void fun(){  
    g++;  
}
```

```
int main(){  
    //thread t1(fun);
```

```
    int i=g;
```

```
    i++;
```

```
    g=i;
```

```
    //t1.join();
```

```
    :  
}
```

If no threads?

Where are critical sections in the following code?

```
int g=0;

void fun(){
    g++;
}

int main(){
    //thread t1(fun);

    int i=g;

    i++;

    g=i;

    //t1.join();
    :
}
```

If no threads?

If no threads then single threaded, no critical sections.

Where are critical sections in the following code?

```
int g=0;

void fun(){
    int a=g;
}

int main(){
    thread t1(fun);

    int i=g;

    i++;

    g=i;

    t1.join();
}
```

If no threads?

If no threads then single threaded, no critical sections.

If fun() just reads g?

Where are critical sections in the following code?

```
int g=0;

void fun(){
    int a=g;
}

int main(){
    thread t1(fun);

    int i=g;

    i++;

    g=i;

    t1.join();
}
```

If no threads?

If no threads then single threaded, no critical sections.

If fun() just reads g?

Where are critical sections in the following code?

```
int g=0;

void fun(){
    int a=g;
}

int main(){
    thread t1(fun);

    int i=g;

    i++;

    1 g=i;

    t1.join();
}
```

If no threads?

If no threads then single threaded, no critical sections.

If fun() just reads g?

g is being written at
need protection

1

If 1 write then all reads and writes

Where are critical sections in the following code?

```
int g=0;

void fun(){
    int a=g;
}

int main(){
    thread t1(fun);

    int i=g;
    i++;
    g=i;

    t1.join();
}
```

If no threads?

If no threads then single threaded, no critical sections.

If fun() just reads g?

g is being written at

1

If 1 write then all reads and writes

need protection

See code in the rounded rectangle for critical sections

Where are critical sections in the following code?

```
int g=0;

void fun(){
    int a=g;
}

int main(){
    thread t1(fun);

    int i=g;

    i++;

    1 g=i;

    t1.join();
}
```

If no threads?

If no threads then single threaded, no critical sections.

If fun() just reads g?

g is being written at

1

If 1 write then all reads and writes

need protection

Will these smaller critical sections work?

Where are critical sections in the following code?

```
int g=0;

void fun(){
    int a=g;
}

int main(){
    thread t1(fun);

    int i=g;

    i++;

    1 g=i;

    t1.join();
}
```

If no threads?

If no threads then single threaded, no critical sections.

If fun() just reads g?

g is being written at

1

If 1 write then all reads and writes

need protection

Will these smaller critical sections work?

NO

What if I receives g, then you increment i,
Then another thread chages g, then this thread writes
I back to g? Answer: you overwrite other threads changes

Where are critical sections in the following code?

```
int g=0;

void fun(){
    g++;
}

1 → int main(){
    thread t1(fun);

    int i=g;

    i++;

    g=i;

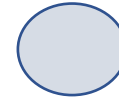
    t1.join();
}
```

If no threads?

If no threads then single threaded, no critical sections.

If fun() just reads g?

g is being written at



If 1 write then all reads and writes need protection

See code in the rounded rectangle for critical sections

If thread starts in position



Where are critical sections in the following code?

```
int g=0;
```

```
void fun(){
```

```
    g++;
```

```
}
```

```
int main(){
```

```
    thread t1(fun);
```

```
    int i=g;
```

```
    i++;
```

```
    g=i;
```

```
    t1.join();
```

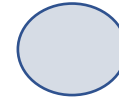
```
}
```

If no threads?

If no threads then single threaded, no critical sections.

If fun() just reads g?

g is being written at



If 1 write then all reads and writes need protection

See code in the rounded rectangle for critical sections

If thread starts in position

1

Where are critical sections in the following code?

```
int g=0;

void fun(){
    g++;
}

int main(){
    int i=g;

    2 → thread t1(fun);

    i++;

    g=i;

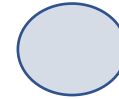
    t1.join();
}
```

If no threads?

If no threads then single threaded, no critical sections.

If fun() just reads g?

g is being written at



If 1 write then all reads and writes need protection

See code in the rounded rectangle for critical sections

If thread starts in position

2

Where are critical sections in the following code?

```
int g=0;

void fun(){
    g++;
}

int main(){
    int i=g;

    2 → thread t1(fun);

    i++;

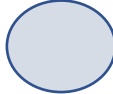
    g=i;

    t1.join();
}
```

If no threads?

If no threads then single threaded, no critical sections.

If fun() just reads g?

g is being written at  If 1 write then all reads and writes need protection

See code in the rounded rectangle for critical sections

If thread starts in position

2

Where are critical sections in the following code?

```
int g=0;

void fun(){
    g++;
}

int main(){
    int i=g;

    i++;

    g=i;

    3 → thread t1(fun);

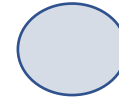
    t1.join();
}
```

If no threads?

If no threads then single threaded, no critical sections.

If fun() just reads g?

g is being written at



If 1 write then all reads and writes need protection

See code in the rounded rectangle for critical sections

If thread starts in position

3

Where are critical sections in the following code?

```
int g=0;

void fun(){
    g++;
}

int main(){
    int i=g;

    i++;

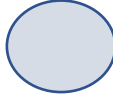
    g=i;

    3 → thread t1(fun);
        t1.join();
}
```

If no threads?

If no threads then single threaded, no critical sections.

If fun() just reads g?

g is being written at  If 1 write then all reads and writes need protection

See code in the rounded rectangle for critical sections

If thread starts in position 

There are no critical sections for position 3 as the global is never Accessed in a multithreaded environment

BTW...When only reading global variables

- If all you do is read a global variable, then there is no critical section and no need to protect access to the global variable.

```
//A global int  
int i=0;
```

```
//Thread 1  
int j=i;
```

```
//Thread 2  
int k=i;
```

BTW... When only reading global variables

- If all you do is read a global variable, then there is no critical section and no need to protect access to the global variable.
- BUT, if you write a global variable at all. Even if just 1 write and 10000 reads.
- Then all 10001 operations are critical and all 10001 must be protected.

```
//A global int  
int i=0;
```

```
//Thread 1  
int j=i;
```

```
//Thread 2  
int k=i;
```

Race Condition again- what happens here?

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

void doZero(){}
void doNotZero(){}

int global=2;
void fun(){
    if(global==0)
        doZero();
    else
        doNotZero();
}

int main() {
    std::thread t1(fun);
    global=0;
    t1.join();
    return 0;
}
```

Diagram illustrating the execution flow of the code:

- Execution starts at the `main` function (labeled 1).
- `main` creates a thread `t1` (labeled 2) and sets `global=0`.
- The thread `t1` (labeled 2) calls `fun()`, which checks the value of `global`.
- Since `global` is 0, `fun()` calls `doZero()`.

- Do you execute `doZero()` or `doNotZero()`?
- If 1 happens before 2
 - Then `doZero()`
- If 2 happens before 1
 - Then `doNotZero()`

How can you tell what happens?

Race Condition again- what happens here?

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

void doZero(){}
void doNotZero(){}

int global=2;
void fun(){
    if(global==0)
        doZero();
    else
        doNotZero();
}

int main() {
    std::thread t1(fun);
    global=0;
    t1.join();
    return 0;
}
```

Diagram illustrating the race condition:

- Execution 1 (labeled 1) starts at `main()` and reaches `global=0;`.
- Execution 2 (labeled 2) starts at `fun()` and reaches the `if(global==0)` condition.

- Do you execute `doZero()` or `doNotZero()`?
- If 1 happens before 2
 - Then `doZero()`
- If 2 happens before 1
 - Then `doNotZero()`

How can you tell what happens?

You cannot as written.

Race Condition again- what happens here?

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

void doZero(){}
void doNotZero(){}

int global=2;
void fun(){
    if(global==0)
        doZero();
    else
        doNotZero();
}

int main() {
    std::thread t1(fun);
    global=0;
    t1.join();
    return 0;
}
```

Diagram illustrating the race condition:

- Execution 2 (blue circle) points to the `if(global==0)` check in the `fun()` function.
- Execution 1 (blue circle) points to the `global=0;` assignment in the `main()` function.

- Do you execute `doZero()` or `doNotZero()`?
- If 1 happens before 2
 - Then `doZero()`
- If 2 happens before 1
 - Then `doNotZero()`

How can you tell what happens?

You cannot as written.

You can however use condition variables to synchronize these operations to do either (later)

Race Condition again- what happens here?

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

void doZero(){}
void doNotZero(){}

int global=2;
void fun(){
    if(global==0)
        doZero();
    else
        doNotZero();
}

int main() {
    std::thread t1(fun);
    global=0;
    t1.join();
    return 0;
}
```

Diagram illustrating the race condition:

- Blue circle 1 points to the `main()` function, specifically the line `global=0;`.
- Blue circle 2 points to the `fun()` function, specifically the line `if(global==0)`.
- Red circle 3 points to the `main()` function, specifically the line `t1.join();`.

- Do you execute `doZero()` or `doNotZero()`?
- If 1 happens before 2
 - Then `doZero()`

- If 2 happens before 1
 - Then `doNotZero()`

How can you tell what happens?

You cannot as written.

You can however use condition variables to synchronize these operations to do either (later)

Or move 1 to position 3

Race Condition again- A bogus solution

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

```
void doZero(){}
void doNotZero(){}

```

```
int global=2;
void fun(){
    if(global==0)
        doZero();
    else
        doNotZero();
}
```

```
int main() {
    std::thread t1(fun);

    //when you see delays like this in the code with
    //comments like "wait for deposit to occur first"
    //or "wait for system stabalization" be very
    //suspicious of the code quality since this often means the
    //original developer has no idea how to coordinate thread activities
    //hint (use condition variables- coming soon)
    std::this_thread::sleep_for(std::chrono::milliseconds(500));    global=0;

    t1.join();
    return 0;
}
```

PSA- you may see code that “fixes” this with delays (see left). This is a cheesy, non scalable solution. (Why?)

DO NOT DO THIS!

Summary

- Race conditions- where they occur, learn to recognize them
- Atomics and problems they solve (single line only)
- Critical Sections- an area of code where only 1 thread can be at a time. Learn how to recognize them, make them small (since only one thread should be in them at a time)
 - Question- If you launch no threads, can you have critical sections?
 - Question-if you only read global variables, can you have critical sections?