

COMPUTATIONAL MATH, SCIENCE AND ENGINEERING DEPARTMENT

MICHIGAN STATE
UNIVERSITY

Concurrency 1

```

5312 0000488B 15470B00 004889D6 4889C7E8 6E050000 488B1535 0B000048 89D64889 C7E85C05 0000488D 35790600 00488B05 140B0000 4889C7E8 7C050000 BE080000
5376 004889C7 E85D0500 00488B15 000B0000 4889D648 89C7E827 050000E8 10050000 4889C348 8D354A06 0000488B 05070A00 004889C7 E83F0500 004889D6 4889C7E8
5440 1C050000 488B15C5 0A000048 89D64889 C7E8EC04 0000E8CF 04000048 89C3488D 351E0600 00488B05 9C0A0000 4889C7E8 04050000 4889D648 89C7E8E1 04000048
5504 8B158A0A 00004889 D64889C7 E8B10400 00488B15 780A0000 4889D648 89C7E89F 04000048 8D35E705 0000488B 05570A00 004889C7 E8BF0400 000B0800 00004889
5568 C7E8A004 0000488B 15430A00 004889D6 4889C7E8 6A040000 488B1531 0A000048 89D64889 C7E85804 0000488D 35B70500 00488B05 100A0000 4889C7E8 78040000
5632 BE040000 004889C7 E8590400 00488B15 FC090000 4889D648 89C7E823 040000E8 F4030000 660F7EC3 488D3588 05000048 8B05D209 00004889 C7E83A04 0000660F
5696 6EC34889 C7E8A004 0000488B 15BF0900 004889D6 4889C7E8 E6030000 E8B10300 00660F7E C3488D35 5B050000 488B0595 09000048 89C7E8FD 03000066 0F6EC348
5760 89C7E8CD 03000048 8B158209 00004889 D64889C7 E8A90300 00488D35 36050000 488B0561 09000048 89C7E8C9 030000BE 18000000 4889C7E8 9E030000 488B154D
5824 09000048 89D64889 C7E87403 0000488B 153B0900 004889D6 4889C7E8 62030000 488D3509 05000048 8B051A09 00004889 C7E88203 0000BE08 00000048 89C7E863
5888 03000048 8B150609 00004889 D64889C7 E82D0300 00E8E602 00006648 0F7EC348 8D35DA04 C000488B 05D80800 004889C7 E8430300 0066480F 6EC34889 C7E80603
5952 0000488B 15C70800 004889D6 4889C7E8 EE020000 E8A10200 0066480F 7EC3488D 35AC0400 00488B05 9C080000 4889C7E8 04030000 66480F6E C34889C7 E8C70200
6016 00488B15 88080000 4889D648 89C7E8AF 02000048 8D358704 0000488B 05670800 004889C7 E8CF0200 00BE3500 00004889 C7E8A402 0000488B 15530800 004889D6
6080 4889C7E8 7A020000 488B1541 08000048 89D64889 C7E86802 0000488D 355B0400 00488B05 20808000 4889C7E8 88020000 BE100000 004889C7 E8690200 00488B15
6144 0C080000 4889D648 89C7E89F 04000048 89D64889 C7E85804 0000488B 05570A00 004889C7 E8480200 000B6D00 DB3C2448 89C7E813 02000048 8B15CE07
6208 00004889 D64889C7 E8B10400 00488B15 780A0000 4889D648 89C7E89F 04000048 8D35E705 0000488B 05570A00 004889C7 E8BF0400 000B0800 00004889
6272 4889D648 89C7E89F 04000048 8D35E705 0000488B 05570A00 004889C7 E8BF0400 000B0800 00004889 C7E8A004 0000488B 155B0700 004889D6 4889C7E8 82010000
6336 8B060000 004883C4 48585DC3 554889E5 4883EC10 897DFC89 75F88370 FC017532 47DF8FF FF000075 29488D3D F8070000 E87D0100 00488D15 04E7FFFF 488D35E5
6400 07000048 8B05F606 00004889 C7E86C01 0000C9C3 554889E5 BEFFFF00 00BF0100 0000E8A5 FFFFFF5D C3554889 E8B00800 FFFF5DC3 554889E5 B8FF7F00 005DC355
6464 4889E588 00000080 5DC35548 89E5B8FF FFFF7F5D C3554889 E5488B00 00000000 0000805D C3554889 E5488BFF FFFF7F5D C3554889 E58B0589 01000066
6528 0F6EC05D C3554889 E58B057D 01000066 0F6EC05D C3554889 E5488B00 00000000 00100066 480F6EC0 5DC35548 89E5488B FFFFFFFF FFFF7F5D 66480F6E C05DC355
6592 4889E548 B8000000 00000000 00B0A010 00004889 45F08955 F8DB6D0F 5DC35548 89E548C7 C0FFFFFF FFBABE7F 00004889 45F08955 F8DB6D0F 5DC3FF25 28060000
6656 FF252A06 0000FF25 2C060000 FF252E06 0000FF25 30060000 FF253206 0000FF25 34060000 FF253606 0000FF25 38060000 FF253A06 0000FF25 3C060000 FF253E06
6720 0000FF25 40060000 FF254206 0000FF25 44060000 FF254606 0000FF25 48060000 FF254A06 0000FF25 4C060000 FF254E06 0000FF25 50060000 FF255206 0000FF25
6784 54060000 4C8D1D95 05000041 53FF2585 05000090 68000000 00E9E6FF FFFF6819 000000E9 DCFFFFFFF 682B0000 00E9D2FF FFFF683D 000000E9 54000000
6848 00E9BEFF FFFF6861 000000E9 B4FFFFFF 68730000 00E9AAFF FFFF6885 000000E9 A0FFFFFF 68970000 00E996FF FFFF68B7 000000E9 8CFFFFFF 68F70000 00E982FF
6912 FFFF6800 00000000 FFFF77FF FFFF7FFF 53697A65 206F6620 73686F72 743A0053 6D616C6C 65737420 73686F72 743A004C 61726765 73742073 686F7274 3A005369
6976 7A65206F 6620696E 743A0053 6D616C6C 65737420 696F743A 004C6172 70696734 3A005369 66206C6F 66E73A00 65737420 206C6F6E 743A0053 6D616C6C 65737420
7040 673A004C 61726765 7374206C 6F6E673A 0053697A 65206F66 206C6F6E 67206C6F 6E672069 6E743A00 53697A65 206F6620 666C6F61 743A0053 6D616C6C 65737420
7104 666C6F61 743A004C 61726765 73742066 6C6F6174 3A004469 67697473 20696E20 6D617469 7376612C 20666C6F 61743A00 53697A65 206F6620 646F7562 6C653A00
7168 536D616C 6C657374 20646F75 626C653A 004C6172 67657374 20646F75 626C653A 00446967 69747320 696E206D 61746973 73612C20 646F7562 6C653A00 53697A65
7232 206F6620 6C6F6E67 20646F75 626C653A 00536D61 6C6C6573 74206C6F 6E67206A 6F75626C 653A004C 61726765 7374206C 6F6E6720 646F7562 6C653A00 00000000
7296 44696769 74732069 6E206D61 74697373 612C206C 6F6E6720 646F7562 6C653A00 14000000 00000000 017A5200 01781001 100C0708 90010000 34000000 1C000000
7360 69FCFFFF FFFFFFFF 0B000000 00000000 00040100 00000E10 86020403 00000000 06040600 00000C07 08000000 00000000 00000000 34000000 54000000 3CFCFFFF FFFFFFFF
7424 0B000000 00000000 00040100 00000E10 86020403 00000000 06040600 00000C07 08000000 00000000 00000000 00000000 34000000 0FFCFFFF FFFFFFFF 00000000
7488 00040100 00000E10 86020403 00000000 06040600 00000C07 08000000 00000000 34000000 C4000000 E2FBFFFF FFFFFFFF 0B000000 00000000 00040100 00000E10
7552 86020403 00000000 06040600 00000C07 08000000 00000000 34000000 FC000000 B5FBFFFF FFFFFFFF 10000000 00000000 00040100 00000E10 86020403 00000000
7616 06040B00 00000C07 08000000 00000000 34000000 34010000 8DFBFFFF FFFFFFFF 10000000 00000000 00000000 00040100 00000E10 86020403 00000000 06040B00
7680 08000000 00000000 34000000 6C010000 65FBFFFF FFFFFFFF 10000000 00000000 00040100 00000E10 86020403 00000000 06040B00 00000C07 08000000 00000000
7744 34000000 A4010000 3DFBFFFF FFFFFFFF 10000000 00000000 00040100 00000E10 86020403 00000000 06040B00 00000C07 08000000 00000000 34000000 DC010000
7808 15FBFFFF FFFFFFFF 15000000 00000000 00040100 00000E10 86020403 00000000 06041000 00000C07 08000000 00000000 00000000 34000000 14020000 F2FAFFFF FFFFFFFF
7872 15000000 00000000 00040100 00000E10 86020403 00000000 06041000 00000C07 08000000 00000000 00000000 CFFAFFFF FFFFFFFF 1F000000 00000000
7936 00040100 00000E10 86020403 00000000 06041A00 00000C07 08000000 00000000 34000000 84020000 B6FAFFFF FFFFFFFF 1C000000 00000000 00040100 00000E10
8000 86020403 00000000 06041700 00000C07 08000000 00000000 34000000 BC020000 02FAFFFF FFFFFFFF 6A050000 00000000 00040100 00000E10 86020403 00000000
8064 06040500 00000303 04600500 000C0708 34000000 F4020000 34F9FFFF FFFFFFFF 48000000 00000000 00040100 00000E10 86020403 00000000 06044300 00000C07
8128 08000000 00000000 34000000 2C030000 44F9FFFF FFFFFFFF 15000000 00000000 00040100 00000E10 86020403 00000000 06041000 00000C07 08000000 00000000
8192 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
8256 BF190000 01000000 75190000 01000000 85190000 01000000 4A190000 01000000 3F190000 01000000 65190000 01000000 55190000 01000000 34190000 01000000
8320 29190000 01000000 9E1A0000 01000000 A31A0000 01000000 B21A0000 01000000 BC1A0000 01000000 C61A0000 01000000 D01A0000 01000000 D81A0000 01000000
8384 DA1A0000 01000000 E41A0000 01000000 EE1A0000 01000000 F81A0000 01000000 14190000 01000000 00000000 00000000 00000000 00000000 00000000
8448 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
8512 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
8576 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

```

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
<pre>if(!s.empty()) int const value=s.top(); s.pop(); do_something(value);</pre>	<pre>if(!s.empty()) int const value=s.top(); s.pop(); do_something(value);</pre>



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



COMPUTATIONAL MATH, SCIENCE AND ENGINEERING DEPARTMENT

MICHIGAN STATE
UNIVERSITY

mutual exclusion

5312 0000488B 15470B00 004889D6 4889C7E8 6E050000 488B1535 0B000048 89D64889 C7E85C05 0000488D 35790600 00488B05 140B0000 4889C7E8 7C050000 BE080000
5376 004889C7 E85D0500 00488B15 000B0000 4889D648 89C7E827 050000E8 10050000 4889C348 8D354A06 0000488B 05070A00 004889C7 E83F0500 004889D6 4889C7E8
5440 1C050000 488B15C5 0A000048 89D64889 C7E8EC04 0000E8CF 04000048 89C3488D 351E0600 00488B05 9C0A0000 4889C7E8 04050000 4889D648 89C7E8E1 04000048
5504 8B158A0A 00004889 D64889C7 E8B10400 00488B15 780A0000 4889D648 89C7E89F 04000048 8D35E705 0000488B 05570A00 004889C7 E8BF0400 000B0800 00004889
5568 C7E8A004 0000488B 15430A00 004889D6 4889C7E8 6A040000 488B1531 0A000048 89D64889 C7E85804 0000488D 35B70500 00488B05 100A0000 4889C7E8 78040000
5632 BE040000 004889C7 E8590400 00488B15 FC090000 4889D648 89C7E823 040000E8 F4030000 660F7EC3 488D3588 05000048 8B05D209 00004889 C7E83A04 0000660F
5696 6EC34889 C7E8A004 0000488B 15BF0900 004889D6 4889C7E8 E6030000 E8B10300 00660F7E C3488D35 5B050000 488B0595 09000048 89C7E8FD 03000066 0F6EC348
5760 89C7E8CD 03000048 8B158209 00004889 D64889C7 E8A90300 00488D35 36050000 488B0561 09000048 89C7E8C9 030000BE 18000000 4889C7E8 9E030000 488B154D
5824 09000048 89D64889 C7E87403 0000488B 153B0900 004889D6 4889C7E8 62030000 488D3509 05000048 8B051A09 00004889 C7E88203 0000BE08 00000048 89C7E863
5888 03800048 8B150609 00004889 D64889C7 E82D0300 00E8E602 00006648 0F7EC348 8D35DA04 C000488B 05D80800 004889C7 E8430300 0066480F 6EC34889 C7E80603
5952 0000488B 15C70800 004889D6 4889C7E8 EE020000 E8A10200 0066480F 7EC3488D 35AC0400 00488B05 9C080000 4889C7E8 04030000 66480F6E C34889C7 E8C70200
6016 00488B15 88080000 4889D648 89C7E8AF 02000048 8D358704 0000488B 05670800 004889C7 E8CF0200 00BE3500 00004889 C7E8A402 0000488B 15530800 004889D6
6080 4889C7E8 7A020000 488B1541 08000048 89D64889 C7E86802 0000488D 355B0400 00488B05 20080000 4889C7E8 88020000 BE100000 004889C7 E8690200 00488B15
6144 0C080000 4889D648 89C7E89F 04000048 F8040000 857D0448 8D35E705 0000488B 05570A00 0000488B 090B0600 DB3C2448 89C7E813 02000048 8B15CE07
6208 00004889 D64889C7 E8B10400 0000488B 153B0900 004889D6 4889C7E8 62030000 488D3509 05000048 8B051A09 00004889 C7E88203 0000BE08 00000048 89C7E863
6272 4889D648 89C7E8AF 02000048 8D358704 0000488B 05670800 004889C7 E8CF0200 00BE3500 00004889 C7E8A402 0000488B 15530800 004889D6
6386 8B080000 004889C7 E8590400 00488B15 FC090000 4889D648 89C7E823 040000E8 F4030000 660F7EC3 488D3588 05000048 8B05D209 00004889 C7E83A04 0000660F
6400 07000048 8B05F606 00004889 C7E86C01 0000C9C3 554889E5 BEFFF000 00BF0100 0000E8A5 FFFFFF5D C3554889 E5B80080 FFFF5DC3 554889E5 B8FF7F00 005DC355
6464 4889E588 00000080 5DC35548 89E5B8FF FFFF7F5D C3554889 E5488B00 00000000 0000805D C3554889 E5488BFF FFFF7F5D C3554889 E58B0589 01000066
6528 0F6EC05D C3554889 E58B057D 01000066 0F6EC05D C3554889 E5488B00 00000000 00100066 480F6EC0 5DC35548 89E5488B FFFF7F5D FFFF7F5D 66480F6E C05DC355
6592 4889E548 B8000000 00000000 00B8A010 00004889 45F08955 F8DB6D0F 5DC35548 89E548C7 C0FFFFF7 FFBABE7F 00004889 45F08955 F8DB6D0F 5DC35548 89E5488B FFFF7F5D
6656 FF252A06 0000FF25 2C060000 FF252E06 0000FF25 30060000 FF253206 0000FF25 34060000 FF253606 0000FF25 38060000 FF253A06 0000FF25 3C060000 FF253E06
6720 0000FF25 40060000 FF254206 0000FF25 44060000 FF254606 0000FF25 48060000 FF254A06 0000FF25 4C060000 FF254E06 0000FF25 50060000 FF255206 0000FF25
6784 54060000 4C8D1D95 05000041 53FF2585 05000090 68000000 00E9E6FF FFFF6819 000000E9 DCF7FFFF 682B0000 00E9D2FF FFFF683D 000000E9 E8092FFF
6848 00E9BEFF FFFF6861 000000E9 B4FFFFF7 68730000 00E9AAFF FFFF6885 000000E9 A0FFFFF7 68970000 00E996FF FFFF68B7 000000E9 8CFFFFF7 68F70000 00E982FF
6912 FFFF0000 00000000 FFFF77F7 FFFF77FF 53697A65 206F6620 73686F72 743A0053 6D616C6C 65737420 73686F72 743A004C 61726765 73742073 686F7274 3A005369
6966 7A65206F 6620696E 743A0053 6D616C6C 65737420 696E743A 004C6172 706967374 3A005369 66206C6F 66E73A00 686F7274 206C6F6E
7040 673A004C 61726765 7374206C 6F6E673A 0053697A 65206F66 206C6F6E 67206C6F 6E672069 6E743A00 53697A65 206F6620 666C6F61 743A0053 6D616C6C 65737420
7104 666C6F61 743A004C 61726765 73742066 6C6F6174 3A004469 67697473 20696E20 6D617469 737612C2 20666C6F 61743A00 53697A65 206F6620 646F7562 6C653A00
7168 536D616C 6C657374 20646F75 626C653A 004C6172 67657374 20646F75 626C653A 00446967 69747320 696E206D 61746973 73612C20 646F7562 6C653A00 53697A65
7232 206F6620 6C6F6E67 20646F75 626C653A 00536D61 6C6C6573 74206C6F 6E67206A 6F75626C 653A004C 61726765 7374206C 6F6E6720 646F7562 6C653A00 00000000
7296 44696769 74732069 6E206D61 74697373 612C206C 6F6E6720 646F7562 6C653A00 14000000 00000000 017A5200 01781001 100C0708 90010000 34000000 1C000000
7360 69FCFFFF FFFFFFFF 0B000000 00000000 00040100 00000E10 86020403 00000000 06040600 00000C07 08000000 00000000 00000000 34000000 34000000
7424 0B000000 00000000 00040100 86020403 00000000 06040600 00000C07 08000000 00000000 34000000 00000000 00000000 00000000 00000000 00000000
7488 00040100 00000E10 86020403 00000000 06040600 00000C07 08000000 00000000 34000000 C4000000 E2FBFFFF FFFFFFFF 0B000000 00000000 00040100 00000E10
7552 86020403 00000000 06040600 00000C07 08000000 00000000 34000000 FC000000 B5FBFFFF FFFFFFFF 10000000 00000000 00040100 00000E10 86020403 00000000
7616 06040B00 00000C07 08000000 00000000 34000000 34010000 8DFBFFFF FFFFFFFF 10000000 00000000 00000000 00040100 86020403 00000000 06040B00 00000C07
7680 08000000 00000000 34000000 6C010000 65FBFFFF FFFFFFFF 10000000 00000000 00040100 00000E10 86020403 00000000 06040B00 00000C07 08000000 00000000
7744 34000000 A4010000 3DFBFFFF FFFFFFFF 10000000 00000000 00040100 00000E10 86020403 00000000 06040B00 00000C07 08000000 00000000 34000000 DC010000
7808 15FBFFFF FFFFFFFF 15000000 00000000 00040100 00000E10 86020403 00000000 06041000 00000C07 08000000 00000000 00000000 00000000 00000000
7872 15000000 00000000 00040100 86020403 00000000 06041000 00000C07 08000000 00000000 34000000 00000000 CFFAFFFF FFFFFFFF 1F000000 00000000
7936 00040100 00000E10 86020403 00000000 06041A00 00000C07 08000000 00000000 34000000 84020000 B6FAFFFF FFFFFFFF 1C000000 00000000 00040100 00000E10
8000 86020403 00000000 06041700 00000C07 08000000 00000000 34000000 BC020000 02FAFFFF FFFFFFFF 6A050000 00000000 00040100 00000E10 86020403 00000000
8064 06040500 00000303 04680500 000C0708 34000000 F4020000 34F9FFFF FFFFFFFF 48000000 00000000 00040100 00000E10 86020403 00000000 06044300 00000C07
8128 08000000 00000000 34000000 2C030000 44F9FFFF FFFFFFFF 15000000 00000000 00040100 00000E10 86020403 00000000 06041000 00000C07 08000000 00000000
8192 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
8256 BF190000 01000000 75190000 01000000 85190000 01000000 4A190000 01000000 3F190000 01000000 65190000 01000000 55190000 01000000 34190000 01000000
8320 29190000 01000000 9E1A0000 01000000 8A1A0000 01000000 B21A0000 01000000 BC1A0000 01000000 C61A0000 01000000 D01A0000 01000000 001A0000 01000000
8384 DA1A0000 01000000 E41A0000 01000000 EE1A0000 01000000 F81A0000 01000000 14190000 01000000 00000000 00000000 00000000 00000000 00000000 00000000
8448 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
8512 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
8576 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

CMSE 822, FS21, W.F. Punch

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 Stroustrup



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()
```

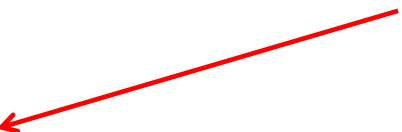
```
void print (const std::string& s)
```

```
{
    std::lock_guard<std::mutex> l(printMutex);
    for (char c : s) {
        std::cout.put(c);
    }
    std::cout << std::endl;
}
```


scope of the
lock_guard



mutex
locked
here



unlocked when
lock_guard out
of scope(destroyed)



```
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");
}
```

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

