HIGH PERFORMANCE COMPUTING FOR SCIENCE AND ENGINEERING

LECTURE 5

Fabian Wermelinger
Harvard University
CS205

Tuesday, February 7th 2023

LAST TIME

- Extension to multicore architectures
- Different kinds of parallelism and how the hardware exploits them
- Differences between processes and threads
- Introduction to race conditions in shared memory

TODAY

Main topic: ***Memory consistency model, critical sections and mutual exclusion***

Details:

- ***Recap:*** what is a race condition?
- Revisit the `Counter` example from last time
- The problem with read and write operations in shared memory
- Memory consistency models
- Sequential consistency and relaxed memory models
- Critical sections and mutual exclusion
- Lock examples for two threads

AGENDA CHECK:

- Project milestone 1 → if you have not found a team yet, please ask on the Ed forum and let the teaching staff know at `cs205-staff@g.harvard.edu`
- Reading assignment for Thursday → OpenMP specification 5.1 until Section 1.4 (inclusive)

RECAP WHAT A DATA RACE IS

High-level (software)

```cpp
#include <iostream>

class Counter
{
public:
    Counter() : count_(0) {}

    // data race because of B!
    void increment() { ++count_; }
    unsigned int get_count() const
    {
        return count_;
    }

private:
    unsigned int count_; // shared because of A
};

int main(void)
{
    Counter counter; // A: main memory (on stack)
#pragma omp parallel for
    for (int i = 0; i < 100; ++i) {
        // B: method call of shared object in
        // region parallel
        counter.increment();
    }
    std::cout << counter.get_count() << '\n';
    return 0;
}
```

Low-level (executable)

- Assume *two* threads `t0` and `t1` run this code
- They both *race* for *write* access to the `count_` attribute
- The executed machine code (***recall:*** *each thread has its own program counter*) at runtime may look like this:

```
mov edx, DWORD PTR [rax]
mov edx, DWORD PTR [rax]
add edx, 1
add edx, 1
mov DWORD PTR [rax], edx
mov DWORD PTR [rax], edx
```

- Why do you think the compiler developers leave such a mess to us programmers? There is a very high potential for *difficult bugs* because of data races (*multi-threaded code only*).

THE ISSUE WITH READ AND WRITE INSTRUCTIONS

- The question you essentially have to ask yourself is:

  Should there be a specific ***order*** among *read* and *write* instructions in code that is executed by multiple threads in parallel?

  Example *read* instruction in the counter code:

```
1 mov edx, DWORD PTR [rax] ; thread 0 loads current value of count_
```

  Example *write* instruction in the counter code:

```
1 mov DWORD PTR [rax], edx ; thread 1 writes register edx to memory
```

- The answer is ***YES!*** → *but why are the compiler people not doing it then?*
- The simple answer for this question is because of ***performance!***

  If you enforce a strict read/write order you take away *substantial* freedom from the compiler to *optimize* code (e.g. when you pass the `-O2` or `-O3` options at compile time). ***Example optimization:*** write instructions are often queued in a *write buffer* for efficiency. *Such an optimization would no longer be possible if a certain order was imposed on write instructions.*

THE ISSUE WITH READ AND WRITE INSTRUCTIONS

We have a conflict of interest:

- We want ***correct*** multi-threaded code (👍)
- This means we must forbid the compiler from doing optimizations (👎)

All the efforts we put in to parallelize code (make it run faster) would be destroyed by slow, non-optimized machine code!

The solution is to come up with a ***memory (consistency) model*** that defines some ***synchronization primitives*** which dictate the compiler whether certain optimizations are allowed to do at this point or not.

Take-away from CS205: *synchronization means serialization*.

Using many synchronization primitives in your code often means your algorithm does not parallelize well. Synchronization is *expensive and not parallel*. You must only use them *where necessary for correctness*.

MEMORY CONSISTENCY MODEL

> *The result of any execution is the same as if the operations of all the processes were executed in some sequential order (total order) and the operations of each individual process appear in this sequence in the order specified by its program (partial order).*
>
> *Lamport, 1979*

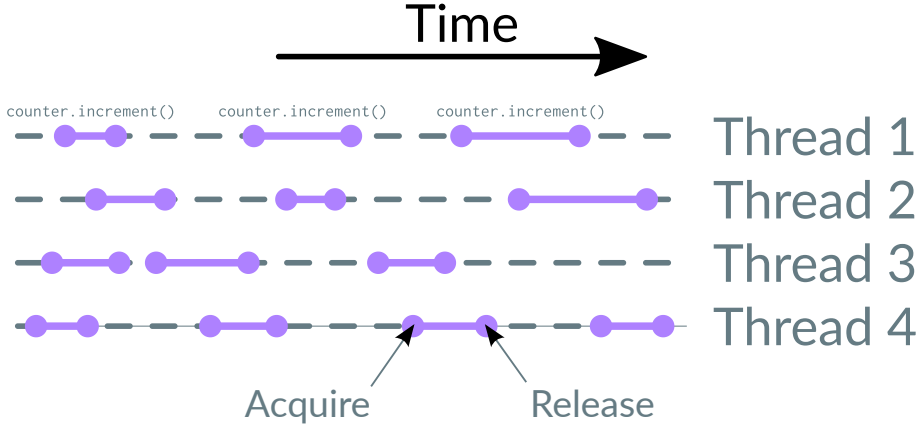
(***Recommended read:*** `lamport1979a.pdf` in class Git repository.)

As we have seen by the `Counter` example, this consistency is obviously not guaranteed for us. What happened is what we now identify as a ***race condition***. As we seek performance in our parallel code, *you (the programmer) must **explicitly** ensure correctness in shared memory codes (through synchronization primitives enabled by the memory consistency model)*. This is the major difficulty added to the shared memory programming task when compared with sequential programming.

BACK TO THE PARALLEL COUNTER

```cpp
#include <iostream>

int main(void)
{
    Counter counter;
#pragma omp parallel for
    for (int i = 0; i < 100; ++i) {
        counter.increment();
    }
    std::cout << counter.get_count() << '\n';
    return 0;
}
```

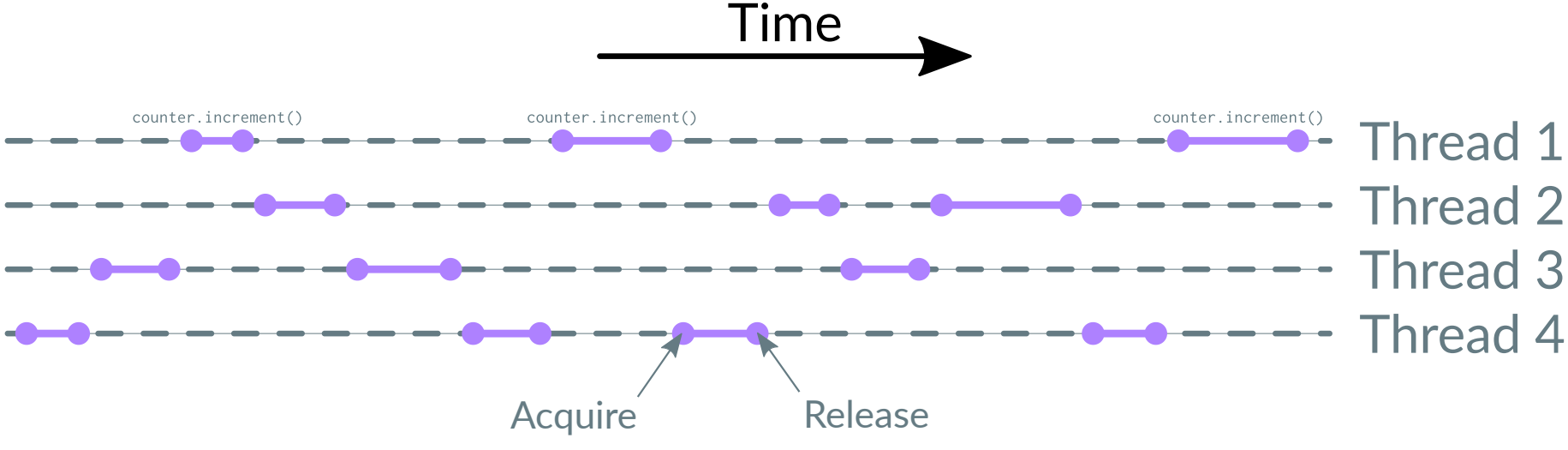


race conditions (memory inconsistent)

- Threads read the current value of `count_` update and write it back to memory
- The time between acquiring and releasing exclusive access to `count_` is called a *critical section*
- The compiler generates machine code where write instructions ***do not*** follow the same order as in the program → *not sequentially consistent*

SEQUENTIAL CONSISTENCY

- A sequentially consistent memory model means that the read/write order will be consistent with a sequential execution flow (*program order*).
- It is the easiest and most intuitive memory model but (most of the time) too restrictive w/r/t compiler optimizations and not widely used in practice.
- If the compiler had followed a *sequentially consistent* memory model, you would not need to insert any synchronization primitives manually!

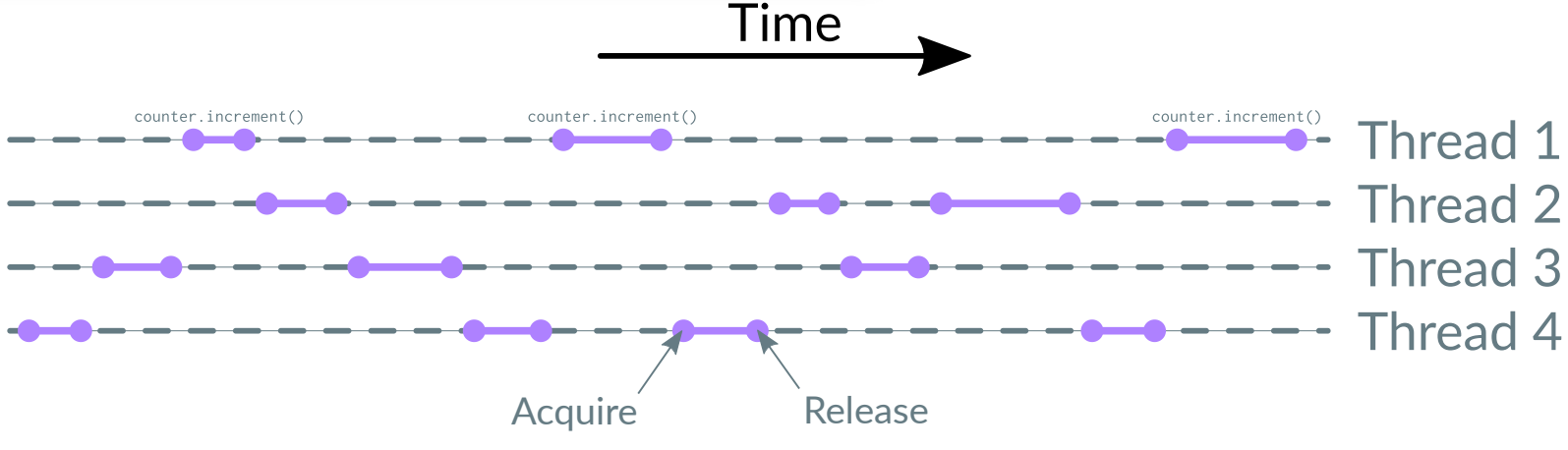


Example for sequentially consistent parallel counter. Note that this *synchronizes* calls to `counter.increment()`. *Our counter is not parallelizable (it's a sequential algorithm).*

ENFORCING MEMORY CONSISTENCY IN OPENMP

```
1  class Counter
2  {
3  public:
4      Counter() : count_(0) {}
5
6      void increment() {
7  #pragma omp atomic
8          ++count_; // this is a critical section
9      }
10     unsigned int get_count() const { return count_; }
11
12 private:
13     unsigned int count_;
14 };
```

- The `#pragma omp atomic` construct is an OpenMP specific statement.
- Below the hood it is part of a *memory consistency model* implemented in OpenMP. *There are other ways to accomplish this!*
- The implementation of a memory model is a specific part of threading library used (e.g. OpenMP and C++ both implement a memory model)

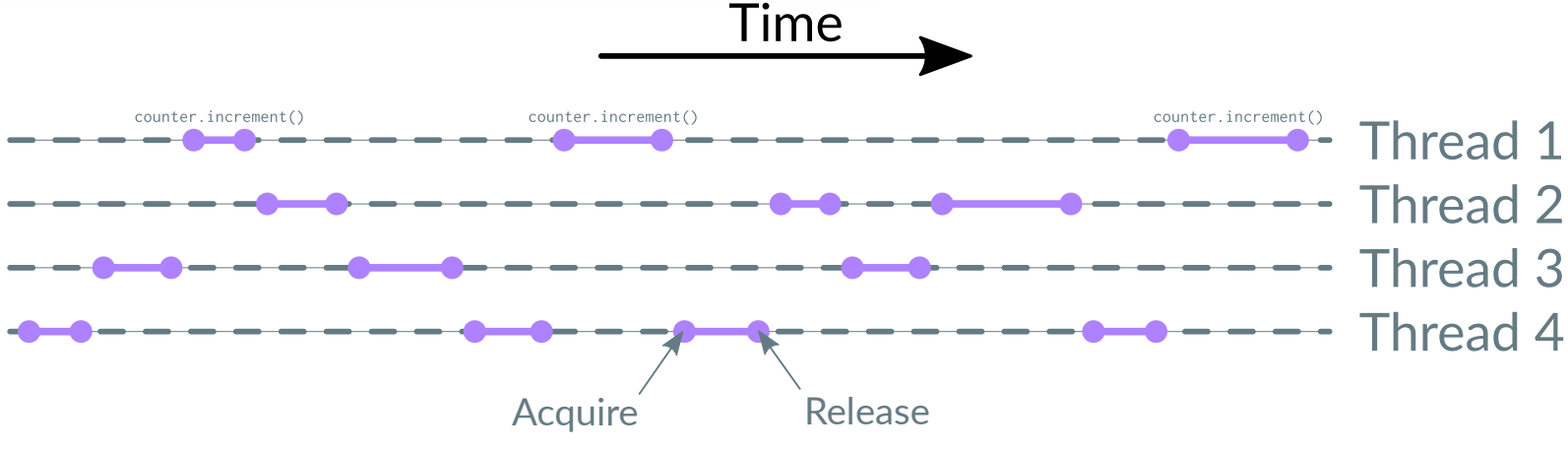


Data-race free (but same wall time as sequential code!)

EXAMPLE: MEMORY CONSISTENCY WITH C++

```
1  #include <atomic> // C++11 and beyond only
2
3  class Counter
4  {
5  public:
6      Counter() : count_(0) {}
7
8      void increment() { ++count_; }
9      unsigned int get_count() const { return count_; }
10
11 private:
12
   std::atomic_uint count_; // thread-safe!
13 };
```

- C++ implements a memory model since the 2011 standard.
- This achieves the same as the OpenMP solution before using only C++ this time.
- *No* new line of code added ***but*** code has been changed in line 12 → the type of `count_` has changed and you now require a compiler that understands this new code!



Data-race free (but same wall time as sequential code!)

SEQUENTIALLY CONSISTENT READ/WRITE ORDER

- The memory consistency model controls when *writes* to a (shared) memory location by one thread become *visible* to another thread.
- Visible means that when the other thread *reads* from that shared memory location, it will "see" new value.
- Consider the following sequence of statements executed by *two* threads (assume initially `A = B = 0` in *shared memory*):

Thread 1

```
1  A = 1;
2  x = B;
```

Thread 2

```
1  B = 1;
2  y = A;
```

- In line 1 the threads ***write*** $A \leftarrow 1$ and $B \leftarrow 1$, respectively (they update two different memory locations).
- In line 2 the threads ***read*** $x \leftarrow B$ and $y \leftarrow A$, respectively. Assume x and y are *thread local* variables (not important for this example) → *what matters are the read/write operations on shared locations A and B.*

SEQUENTIALLY CONSISTENT READ/WRITE ORDER

- Consider the following sequence of statements executed by *two* threads (assume initially `A = B = 0` in *shared memory*):

Thread 1

```
1  A = 1;
2  x = B;
```

Thread 2

```
1  B = 1;
2  y = A;
```

- What are the possible ***total order*** scenarios for *sequential consistency*, given the individual sequence of statements per thread (***partial order***)?

Scenario 1:

A←1

x←B

B←1

y←A

Result: x=0, y=1

Scenario 2:

A←1

B←1

x←B

y←A

Result: x=1, y=1

Note: the two writes to A and B could be transposed as well. The same applies for the writes to x and y. The result does not change.

Scenario 3:

B←1

y←A

A←1

x←B

Result: x=1, y=0

-

RELAXED CONSISTENCY MODELS

- ***Recall:*** *maintaining sequential consistency is expensive*. It means that a *write* to a *shared variable* must be visible immediately to all other threads.
- For a sequentially consistent memory model, no synchronization primitives are needed → *we did not observe this behavior in the* `Counter` *example earlier.*
- Relaxed consistency models *allow* read and write operations to complete ***out of order*** for code optimization reasons. This requires explicit *synchronization* operations to *enforce* sequential consistency such that your parallel program eventually behaves the same as its sequential version.
- You can now see why synchronization primitives *are bad* and you *do not want* to use them unless necessary to enforce correctness. They are bad because you prohibit the compiler from doing optimizations and therefore generate slow code.
- There are different *relaxed memory consistency models* used in practice. They are all related to relaxing the ordering constraints of write operations relative to read operations ***and require you to take action*** to enforce correctness.

RELAXED CONSISTENCY MODELS

- Consider the following sequence of statements executed by *two* threads (assume initially `A = B = 0` in *shared memory*):

Thread 1

```
1  A = 1;
2  x = B;
```

Thread 2

```
1  B = 1;
2  y = A;
```

- What are the possible ***total order*** scenario(s) for *violation* of sequential consistency, given the ***partial order*** above?

Scenario 4:

x←B

y←A

A←1

B←1

Result: x=0, y=0

Note: the two writes to x and y could be transposed as well. The same applies for the writes to A and B. The result does not change.

- This total order ***is not*** sequentially consistent because it does not respect the partial order for the two threads above. This will produce an *unexpected* result due to a race condition!
- Where do you need to add ***synchronization*** to restore sequentially consistent results?

SEQUENTIAL CONSISTENCY EXERCISES

Are the following partial orderings *sequentially consistent*? *All variables are set to `0` initially.* ***Notation: (examples)***

- `W(x,1)`: write the value `1` to memory location of variable `x`
- `R(x,0)`: read value `0` from memory location of variable `x`

1.

Thread 1

```
1 W(x,1);
```

Thread 2

```
1 R(x,0);
2 R(x,1);
```

2.

Thread 1

```
1 W(x,1);
```

Thread 2

```
1 R(x,1);
2 R(x,0);
```

3.

Thread 1

```
1 W(x,1);
```

Thread 2

```
1 W(x,2);
```

Thread 3

```
1 R(x,1);
2 R(x,2);
```

CRITICAL SECTION

```
1  #include <iostream>
2
3  int main(void)
4  {
5      Counter counter;
6  #pragma omp parallel for
7      for (int i = 0; i < 100; ++i) {
8          counter.increment(); // critical section
9      }
10     std::cout << counter.get_count() << '\n';
11     return 0;
12 }
```

- Line 8 in the code on the left is a ***critical section*** because multiple threads try to modify *the same memory location* (the location is in *shared memory*).
- This is dangerous because the relaxed memory consistency model allows for *race conditions* without our supervision.

- We simply fixed this problem by using a *synchronization primitive* via OpenMP (and saw another example where we used the memory model implemented in C++11)
- Synchronization primitives restore sequential consistency (program order)
- In general there are more than just one statement in a critical section. An atomic primitive will not work in that case.

```
1  class Counter
2  {
3  public:
4      Counter() : count_(0) {}
5
6      void increment() {
7  #pragma omp atomic
8      ++count_;
9      }
10
11 private:
12     unsigned int count_;
13 };
```

MUTUAL EXCLUSION

In shared memory programming we need to control *access to critical sections* which requires some form of synchronization primitive. A ***lock*** (sometimes called *mutex*) is an abstract data type that implements ***mutual exclusion*** for critical sections.

The desired properties of a lock are:

- ***Mutual exclusion:*** critical sections of different threads do not overlap
- ***Memory consistent:*** operations are *visible to all threads* after the critical section
- ***Progress:*** if a thread is not inside the critical section, another thread can always enter
- ***Starvation-freedom:*** (implies *deadlock-freedom*) if a thread wants to enter the critical section, it will *eventually* succeed
- ***Fairness:*** if a thread `t0` requests the lock *before* thread `t1`, it will *acquire* the lock *before* `t1`
- ***Performance:*** the lock must *scale* to large numbers of competing threads

MUTUAL EXCLUSION

In shared memory programming we need to control *access to critical sections* which requires some form of synchronization primitive. A ***lock*** (sometimes called *mutex*) is an abstract data type that implements ***mutual exclusion*** for critical sections.

Example use of a lock type:

```cpp
#include <iostream>
#include <Lock.h>

int main(void)
{
    Counter counter; // assume not thread-safe
    Lock mutex;      // some lock data type

#pragma omp parallel for
    for (int i = 0; i < 100; ++i) {
        mutex.lock();        // acquire the lock
        counter.increment(); // critical section
        mutex.unlock();      // release the lock
    }
    std::cout << counter.get_count() << '\n';
    return 0;
}
```

How does such a Lock type work?

We study it on the next slides based on a simple 2-thread only lock implementation.

2-THREAD LOCK: FIRST ATTEMPT

```
1  class LockOne
2  {
3  public:
4      void lock(const int me) // my ID
5      {
6          const int other = 1 - me; // other thread's ID
7          flag[me] = true;          // I would like the lock
8          while (flag[other]) {
9          } // busy-wait
10     }
11
12     void unlock(const int me) { flag[me] = false; }
13
14 private:
15     volatile bool flag[2]; // volatile to enforce busy-wait
16 };
```

- Assume `flag[2]` is `false` initially for both threads
- The lock is *greedy* (the thread that requests the lock wants *precedence*)
- Which variables are shared?
- Does the lock satisfy mutual exclusion?
- This lock is not ***deadlock-free***, why?

2-THREAD LOCK: SECOND ATTEMPT

```
1  class LockTwo
2  {
3  public:
4      void lock(const int me) // my ID
5      {
6          victim = me; // other thread goes first
7          while (victim == me) {
8          } // busy-wait
9      }
10
11     void unlock() const {}
12
13 private:
14     volatile int victim; // volatile enforce busy-wait
15 };
```

- This lock is giving precedence to the other thread first
- Does this lock satisfy mutual exclusion?
- Is this lock ***starvation-free*** (implies deadlock freedom)?

2-THREAD LOCK: THIRD ATTEMPT

LockOne and LockTwo are useless on their own (not deadlock-free). What if we combine them?

```
1  class LockPeterson
2  {
3  public:
4      void lock(const int me) // my ID
5      {
6          const int other = 1 - me; // other thread's ID
7          flag[me] = true;          // I would like the lock
8          victim = me;              // but you go first
9          while (flag[other] && victim == me) {
10         } // busy-wait
11     }
12
13     void unlock(const int me)
14     {
15         flag[me] = false; // I am done
16     }
17
18 private:
19     volatile bool flag[2]; // volatile enforce busy-wait
20     volatile int victim;   // volatile enforce busy-wait
21 };
```

- This lock is ***starvation-free*** which implies deadlock freedom → verify it! (***Recommended read:*** `peterson1981a.pdf` in class Git repository.)
- Elegant example to study mutual exclusion → uses only shared memory for communication
- Why is this lock not optimal from a performance perspective?
- In practice ***we do not*** implement locks ourselves (unless you have a good reason). *Mechanisms for mutual exclusion are provided by the shared memory programming model.*
- See this video for Dekker's algorithm (original solution)

RECAP

- Multithreaded programs in a shared memory architecture require constraints on the read/write instruction order to ensure correct parallel programs
- These restrictions do not appear in sequential programs
- ***You*** must enforce these restrictions (by synchronization primitives implemented by the memory model) to ensure correct parallel programs
- It requires you to ***think*** parallel when you write programs for parallel architectures (there are multiple program flow trajectories in parallel execution)
- Not being aware of this paradigm shift is the major source of bugs for novice programmers. ***Bugs due to data races are hard to debug!*** Make use of address sanitizers when debugging.

Further reading:

- Sections 2.6.1.5 and 2.6.1.6 in *"Introduction to High Performance Scientific Computing"*, V. Eijkhout, free pdf
- Section 5.6 in Hennessy and Patterson, *"Computer Architecture"*, Morgan Kaufmann, 2019
- S. V. Adve and K. Gharachorloo, *"Shared Memory Consistency Models: A Tutorial"*, IEEE, 1996
- G. L. Peterson, *"Myths About the Mutual Exclusion Problem"*, Information Processing Letters, 1981
- L. Lamport, *"How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs"*, IEEE, 1979
- E.W. Dijkstra, *"Solution of a Problem in Concurrent Programming Control"*, CACM, 1965