

# Lecture 13

**The C++ Memory model**

**Synchronization variables**

**Implementing synchronization**

# Announcements

## Today's lecture

- Memory locality in the cardiac simulator
- C++ memory model
- Synchronization variables
- Implementing Synchronization

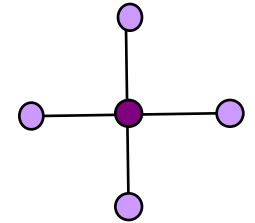
## Improving performance

- We can apply multithreading
- We can reduce the number of cache misses
- Next time: using vectorization

```
for (j=1; j<=m+1; j++){                                     // PDE SOLVER
    for (i=1; i<=n+1; i++) {
        E[j,i] = Eprv[j,i]+ $\alpha$ *(Eprv[j,i+1]+Eprv[j,i-1]-4*Eprv[j,i]+Eprv[j+1,i]+Eprv[j-1,i]);
    }
}
for (j=1; j<=m+1; j++){                                     // ODE SOLVER
    for (i=1; i<=n+1; i++) {
        E[j][i] += -dt*(kk*E[j,i]*(E[j,i]-a)*(E[j,i]-1)+E[j,i]*R[j,i]);
        R[j][i] += dt*( $\epsilon$ +M1* R[j,i] ( E[j,i]+M2))*(-R[j,i]-kk*E[j,i]*(E[j,i]-b-1));
    }
}
```

# Visualizing cache locality

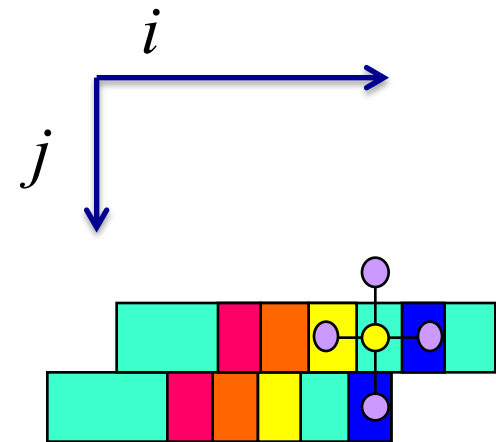
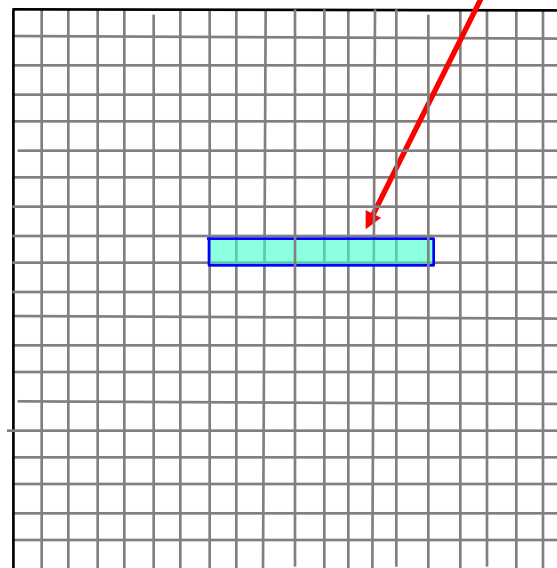
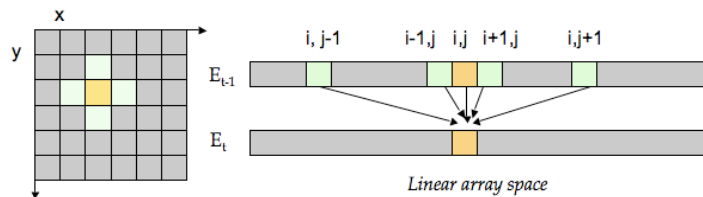
- The stencil's bottom point traces the cache miss pattern:  $[j+1, i]$
- This is called the “frontier” of the stencil update



```

for (j=1; j<=m+1; j++){
    for (i=1; i<=n+1; i++) {
        E[j,i] = Eprev[j,i]+
            α*(Eprev[j,i+1] + Eprev[j,i-1] - 4*Eprev[j,i] + Eprev[j+1,i] + Eprev[j-1,i]);
    }
}
    
```

// PDE SOLVER



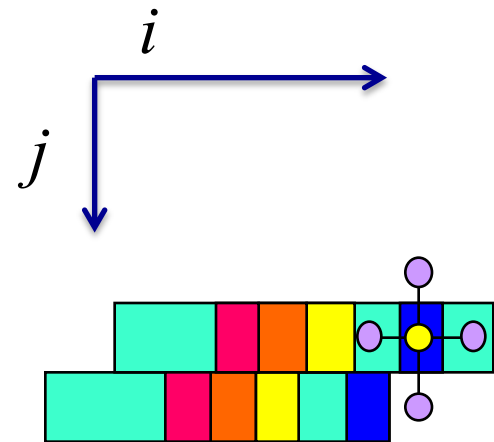
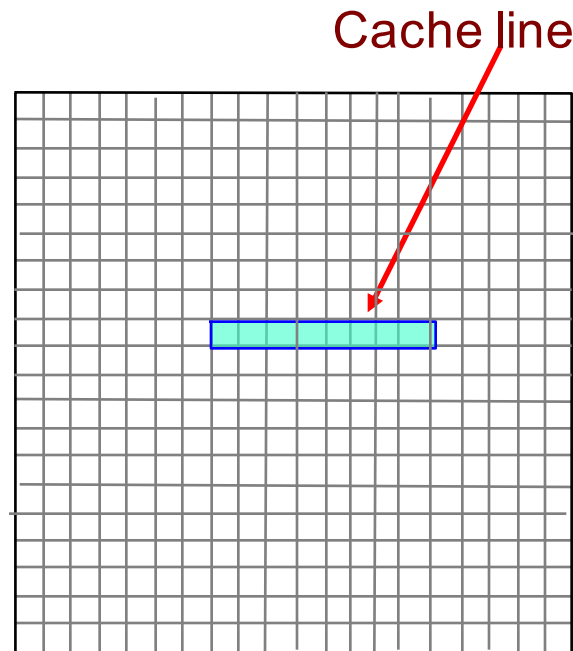
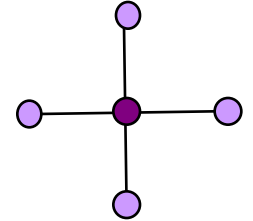
# Visualizing cache locality

- The stencil's bottom point traces the cache miss pattern:  $[i, j+1]$
- There are 6 reads per innermost iteration
- One miss every 8<sup>th</sup> access (8 doubles=1 line)
- We predict a miss rate of  $(1/6)/8 = 2.1\%$

```

for (j=1; j<=m+1; j++){
    for (i=1; i<=n+1; i++) {
        E[j,i] = Eprev[j,i]+
             $\alpha \cdot (Eprev[j,i+1] + Eprev[j,i-1] - 4 \cdot Eprev[j,i] + Eprev[j+1,i] + Eprev[j-1,i]);$ 
    }
}
    
```

// PDE SOLVER



# Where is the time spent?

- The memory addresses are linearized: a 2D ordered pair (i,j) maps to the address  $(i-1)*(m+3)+j$
- There are 12 reads per innermost iteration

```
Command:      ./apf -n 255 -i 2000
Data file:    cachegrind.out.18164
              Dr          Dlmr
```

```
-----
1,382,193,768 50,592,402  PROGRAM TOTALS
1,381,488,017 50,566,005  solve.cpp:solve( ...)
      .      .      .      // Fills in the TOP Ghost Cells
      10,000      1,999  for (i = 0; i < (n+3); i++)
      516,000      66,000      Eprev[i] = Eprev[i + (n+3)*2];
                          // Fills in the RIGHT Ghost Cells
      10,000          0  for (i = (n+2); i < (m+3)*(n+3); i+=(m+3))
      516,000      504,003      Eprev[i] = Eprev[i-2];
                          // Solve for the excitation, a PDE
      1,064,000      8,000  for (j = m+3+1; j <= (((m+3)*(n+3)-1)-(m+1))-(n+3); j+=(m+3)){
      1,024,000      2,000      for (i = 0; i <= n; i++) {
      721,920,001 16,630,000          Eij = Eprev[i+j]+alpha*(Eprev[i+1+j] +
                          Eprev[i-1+j]-4*Eprev[i+j]+Eprev[i+(n+3)+j]+Eprev[i-(n+3)+j]);
                          }
                          // Solve the ODEs
      4,000          4,000  for (j=m+3+1; j <= (((m+3)*(n+3)-1)-(m+1))-(n+3); j+=(m+3)){
                          for (i = 0; i <= n; i++) {
      262,144,000 33,028,000          Eij +=-dt*(kk*Eij*(Eij-a)*(Eij-1)+Eij*Rij);
      393,216,000      4,000          Rij += dt*(ε+M1*Rij/(Eij+M2))*(-Rij-kk+Eij*(Eij-b-1));
                          }
                          }
```

Shorthand:  $E[i+j] \equiv E_{ij}$

Looking at the cache miss counts,  
how many frontier accesses are there  
(reads and writes)?

A. 1 out of 12 total

B. 2 out of 12 total

C. 12 out of 12 total

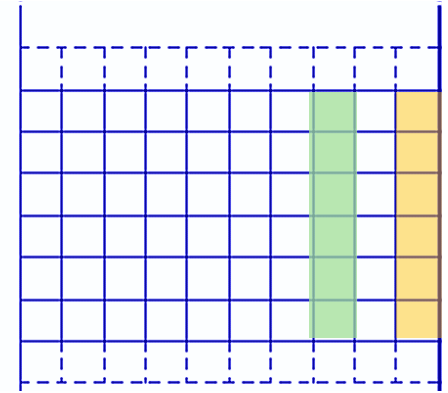
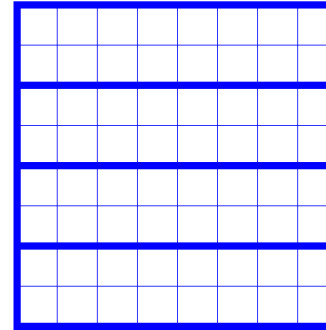
| Dr                | Dlmr       |                                                             |
|-------------------|------------|-------------------------------------------------------------|
| 1,382,193,768     | 50,592,402 | PROGRAM TOTALS                                              |
| 1,381,488,017     | 50,566,005 | solve.cpp:solve( ...)                                       |
| // Solve the ODEs |            |                                                             |
| 4,000             | 4,000      | for (j=m+3+1; j <=((m+3)*(n+3)-1)-(m+1))-(n+3); j+=(m+3)) { |
|                   |            | for (i = 0; i <= n; i++) {                                  |
| 262,144,000       | 33,028,000 | Eij += -dt*(kk*Eij*(Eij-a)*(Eij-1)+Eij*Rij);                |
| 393,216,000       | 4,000      | Rij += dt*(ε+M1*Rij/(Eij+M2))*(-Rij-kk+Eij*(Eij-b-1));      |
|                   |            | }                                                           |

Shorthand:  $E[i+j] \equiv E_{ij}$   
 $R[i+j] \equiv R_{ij}$

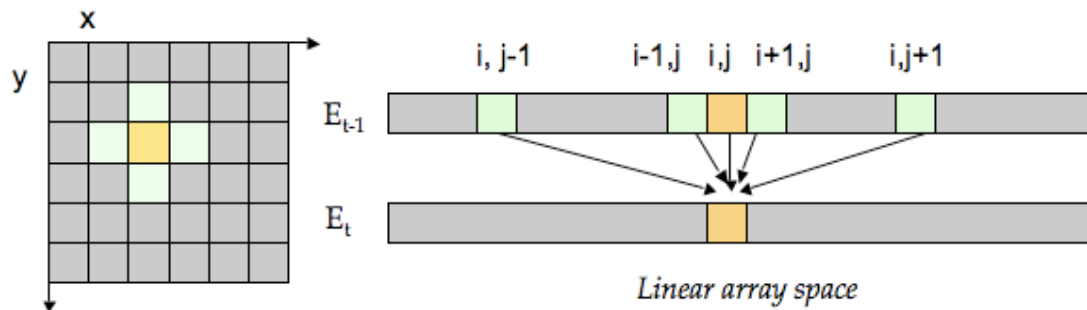
# Which Loop fills in the RIGHT SIDE?

A. Blue loop (top)

B. Red loop (bottom)



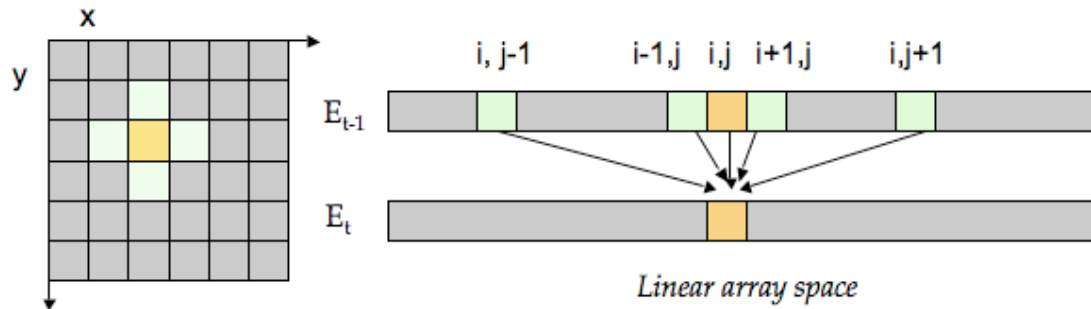
| Dr            | Dlmr       |                                                            |
|---------------|------------|------------------------------------------------------------|
| <hr/>         |            |                                                            |
| 1,381,488,017 | 50,566,005 | <code>solve.cpp:solve( ...)</code>                         |
| 10,000        | 1,999      | <code>for (i = 0; i &lt; (n+3); i++)</code>                |
| 516,000       | 66,000     | <code>  Eprev[i] = Eprev[i + (n+3)*2];</code>              |
| 10,000        | 0          | <code>for (i = (n+2); i &lt; (m+3)*(n+3); i+=(m+3))</code> |
| 516,000       | 504,003    | <code>  Eprev[i] = Eprev[i-2];</code>                      |



# Memory strides

- Some nearest neighbors that are nearby in space are far apart in memory
- *Stride* = memory distance along the direction we are moving: N along the vertical dimension
- Miss rate much higher when moving vertical strips of data than horizontal ones –the padding code

| Dr            | Dlmr       |                                                            |
|---------------|------------|------------------------------------------------------------|
| <hr/>         |            |                                                            |
| 1,381,488,017 | 50,566,005 | <code>solve.cpp:solve( ...)</code>                         |
| 10,000        | 1,999      | <code>for (i = 0; i &lt; (n+3); i++)</code>                |
| 516,000       | 66,000     | <code>    Eprev[i] = Eprev[i + (n+3)*2];</code>            |
|               |            | <code>// Fills in</code>                                   |
|               |            | <code>// TOP RIGHT</code>                                  |
| 10,000        | 0          | <code>for (i = (n+2); i &lt; (m+3)*(n+3); i+=(m+3))</code> |
| 516,000       | 504,003    | <code>    Eprev[i] = Eprev[i-2];</code>                    |
|               |            | <code>// RIGHT SIDE</code>                                 |



What problems may arise when copying the left and right sides, assuming each thread gets a rectangular region and it shares values with neighbors that own the outer dashed region?

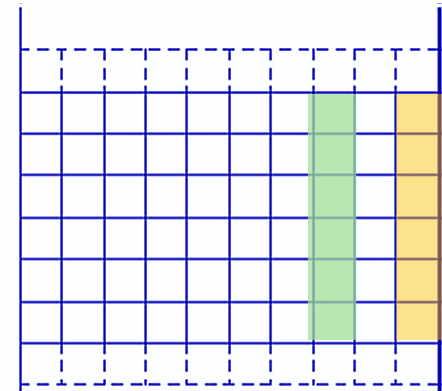
A. False sharing

B. Poor data reuse in cache

C. Data races

D. A & B only

E. All



```
for (i = (n+2); i < (m+3)*(n+3); i+=(m+3))  
    Eprev[i] = Eprev[i-2];
```

What problems may arise when copying the top and bottom sides, assuming each thread gets a rectangular region and it shares values with neighbors that own the outer dashed region?

A. False sharing Some false sharing is possible, though not significant

B. Poor data reuse in cache

C. Data races

D. A & B only

E. None



```
for (i = 0; i < (n+3); i++)  
    Eprev[i] = Eprev[i + (n+3)*2];
```

# Today's lecture

- Memory locality in the cardiac simulator
- C++ memory model
- Synchronization variables
- Implementing Synchronization

# Recalling from last time: atomics

- Assignment involving atomics is restricted
  - No copy or assignment constructors, these are illegal  
`atomic<int> x=7;      // Some C++ documentation permits this!`  
`atomic<int> u = x`  
`atomic<int> y(x);`
  - We can assign to, or copy from, or to a non-atomic type  
`x=7;`  
`int y = x;`
  - We can also use direct initialization involving constants  
`atomic<int> x(0)`
- We will use the sequentially consistent variant (default)  
`memory_order_seq_cst`
- We only need to use the `atomic::load()` and `store()` functions if we require another *memory consistency model*; the default can penalize performance [http://en.cppreference.com/w/cpp/atomic/memory\\_order](http://en.cppreference.com/w/cpp/atomic/memory_order)  
`memory_order_relaxed`

# Memory models

- Earlier we discussed cache coherence and consistency
- Cache coherence is a *mechanism*, a hardware protocol to ensure that memory updates propagate to other cores. Cores will then be able to agree on the values of information stored in memory, *as if* there were no cache at all
- *Cache consistency* defines a *programming model*: **when** do memory writes become visible to other cores?
  - ▶ Defines the ordering of memory updates
  - ▶ A contract between the hardware and the programmer: if we follow the rules, the results of memory operations are guaranteed be predictable

# The C++11 Memory model

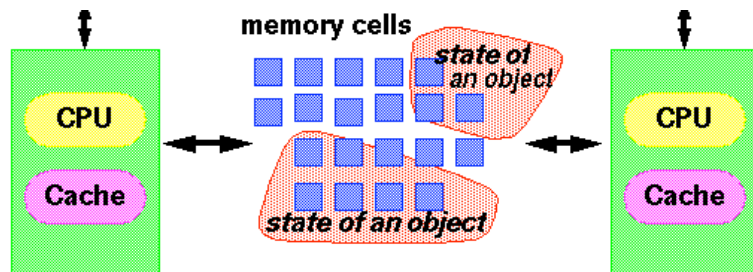
- C++ provides a layer of abstraction over the hardware, so we need another model, i.e. a contract between the hardware and the *C++11 programmer*
  - Ensure that multithreaded programs are *portable*: they will run correctly on different hardware
  - Clarify which optimizations will or will not break our code
- We need these rules, for example, to understand when we can have a data race, so we can know when our program is correct, and that it will run correctly by all compliant C++11 compilers

| Thread 1                          | Thread 2                          |
|-----------------------------------|-----------------------------------|
| r1 = X;<br>if (r1 == 1)<br>Y = 1; | r2 = Y;<br>if (r2 == 1)<br>X = 1; |

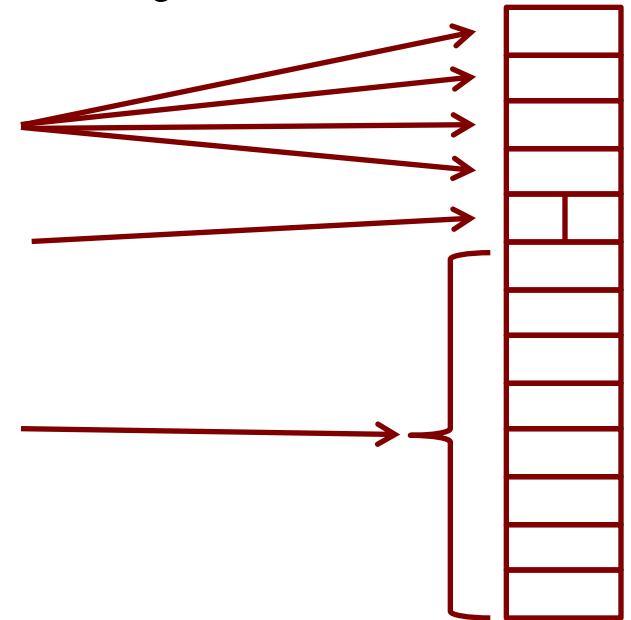
- For example, we might ask:  
“If  $X=Y=1$ , is it possible for the outcome of this program to be  $r1 = r2 = 1$ ?”

# Preliminaries

- The C++11 memory model describes an abstract relation between threads and memory
- Provides guarantees about the interaction between instruction sequences and variables in memory
- Every variable occupies 1 memory location
  - Bit fields and arrays are different; don't load all of `c[ ]` as a 32 bit word
- A write to one location can't affect writes to adjacent ones



```
struct s {  
    char c[4];  
    int i:3, j:4;  
    struct in {  
        double d;  
    } id;  
};
```



# Why don't we want to load all of the `c[ ]` array as one word?

A. Because each element is considered an “variable”

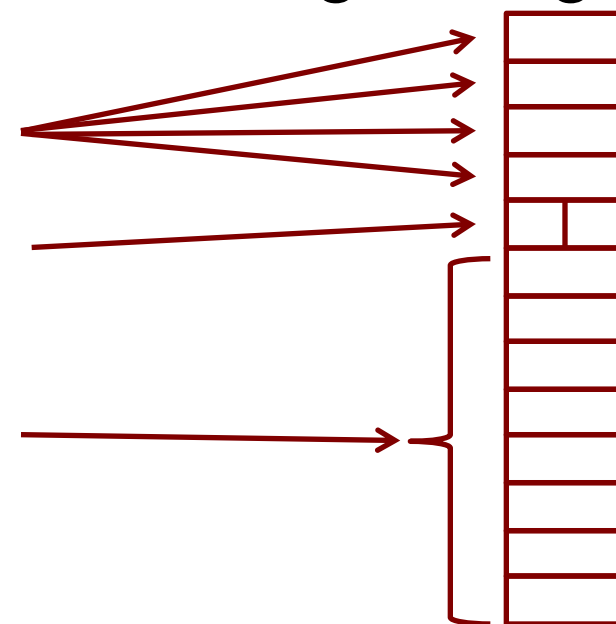
B. Because another thread could be writing a single element

C. Because another thread could be reading a single element

D. A and B

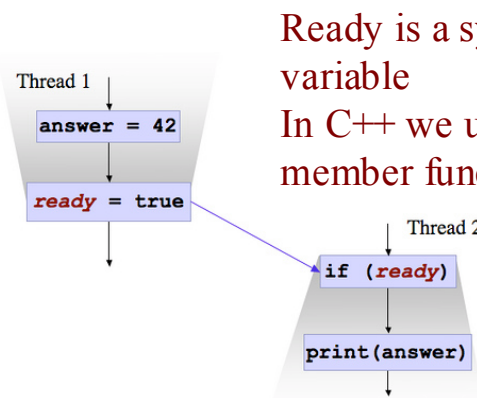
E. B and C

```
struct s {  
    char c[4];  
    int i:3, j:4;  
  
    struct in {  
        double d;  
    } id;  
};
```



# Communication

- Memory writes made by one thread *can* become visible, but ....
- ... special mechanisms are needed to guarantee that communication happens between threads
- *Without explicit communication, you can't guarantee which writes get seen by other threads, or even the order in which they will be seen*
- The C++ atomic variable (and the Java **volatile** modifier) constitutes a special mechanism to guarantee that communication happens between threads
- When one thread writes to a *synchronization variable* (e.g. an *atomic*) and another thread sees that write, the first thread is telling the second about **all** of the contents of memory up until it performed the write to that variable



Ready is a synchronization variable

In C++ we use load and store member functions

*All the memory contents seen by T1, before it wrote to ready, must be visible to T2,*

*after it reads the value true for ready.*

<http://jeremymanson.blogspot.com/2008/11/what-volatile-means-in-java.html>

# The effects of synchronization

- Synchronization can be characterized in terms of 3 properties  
**Atomicity, Visibility, Ordering**
- All changes made in one synchronized variable or code block are atomic and visible with respect to other synchronized variables and blocks employing the same lock, and processing of synchronized methods or blocks within any given thread is in program-specified order
- Out of order processing cannot matter to other threads employing synchronization
- When synchronization is not used or is used inconsistently, answers are more complex
- Imposes additional obligations on programmers attempting to ensure object consistency relations that lie at the heart of exclusion
- Objects must maintain invariants as seen by all threads that rely on them, not just by the thread performing any given state modification

# The 3 Properties

- Of most concern when values must be transferred between main memory and per-thread memory
- **Atomicity.** Which instructions must have indivisible effects?
- **Visibility.** Under what conditions are the effects of one thread visible to another? The effects of interest are: writes to variables, as seen via reads of those variables
- **Ordering.** Under what conditions can the effects of operations appear out of order to any given thread? In particular, reads and writes associated with sequences of assignment statements.

# What kinds of variables require atomic updates?

A. Instance variables and static variables

B. Array elements. Depends on the access pattern

C. Local variables inside methods

D. A & B

E. B & C

# Data races

- We say that a program allows a *data race* on a particular set of inputs if there is a *sequentially consistent execution*, i.e. an interleaving of operations of the individual threads, in which two conflicting operations can be executed “simultaneously” (Boehm)
- We say that operations can be executed “simultaneously” if they occur next to each other in the interleaving, and correspond to different threads
- We can guarantee sequential consistency only when the program avoids data races
- Consider this program, with  $x == y == 0$  initially

| Thread 1              | Thread 2              |
|-----------------------|-----------------------|
| $x = 1;$<br>$r1 = y;$ | $y = 1;$<br>$r2 = x;$ |

# Does this program have a data race?

A. Yes

B. No

$x = y = 0$  initially

**Atomic<int> x; int y;**

| Thread 1              | Thread 2              |
|-----------------------|-----------------------|
| $x = 1;$<br>$r1 = y;$ | $y = 1;$<br>$r2 = x;$ |

# Data races

- We say that a program allows a *data race* on a particular set of inputs if there is a *sequentially consistent execution*, i.e. an interleaving of operations of the individual threads, in which two conflicting operations can be executed “simultaneously” (Boehm)
- We say that operations can be executed “simultaneously” if they occur next to each other in the interleaving, and correspond to different threads
- We can guarantee sequential consistency only when the program avoids data races
- This program has a data race ( $x = y = 0$  initially)

| Execution                                                                                      |
|------------------------------------------------------------------------------------------------|
| <pre>x = 1;<br/>r1 = y;<br/>y = 1;<br/>r2 = x;<br/>// r1 = 1 <math>\wedge</math> r2 == 1</pre> |

| Thread 1                      | Thread 2                      |
|-------------------------------|-------------------------------|
| <pre>x = 1;<br/>r1 = y;</pre> | <pre>y = 1;<br/>r2 = x;</pre> |

# “Happens-before”

- Fundamental concept in understanding the memory model
- Consider these 2 threads, with counter = 0
  - A: counter++;
  - B: prints out counter
- Even if B executes after A, we cannot guarantee that B will see 1 ...
- ... **unless** we establish a *happens-before relationship* between these two statements running in different threads
- What guarantee is made by a *happens-before* relationship?  
A guarantee that memory writes by one specific statement are visible to another specific statement
- Different ways of accomplishing this: synchronization, atomics, variables, thread creation and completion, e.g.
  - thread tA = thread(A);    tA.join();
  - thread tB = thread(B);    tB.join();



# Establishing a happens-before relationship

- C++ and Java provide *synchronization* variables to communicate between threads, and are intended to be accessed concurrently: the *atomic* types, mutexes
- Such concurrent accesses are not considered data races
- Thus, sequential consistency is guaranteed so long as the only conflicting concurrent accesses are to synchronization variables
- Any write to a synchronization variable establishes a *happens-before* relationship with subsequent reads of that same variable: `x_ready=true` happens-before the read of `x_ready` in Thread 2.
- A statement sequenced before another *happens-before* it  
`x=42` happens-before `x_ready=true`
- *Happens-before* is transitive: everything sequenced before a write to synchronization variable also *happens-before* the read of that synchronization variable by another thread. Thus, assignment `x=42` (T1) is visible after the read of `x_ready` by Thread 2, e.g. the assignment to `r1`

| global: int x; atomic<bool> x_ready; |                                |
|--------------------------------------|--------------------------------|
| Thread 1                             | Thread 2                       |
| x = 42;<br>x_ready = true;           | while (!x_ready) {}<br>r1 = x; |

# Does this program have a race condition?

A. Yes

B. No

| global: int x; atomic<bool> x_ready; |                                |
|--------------------------------------|--------------------------------|
| Thread 1                             | Thread 2                       |
| x = 42;<br>x_ready = true;           | while (!x_ready) {}<br>r1 = x; |