

# Multithreaded Programming Episodes V and VI

Arch D. Robison  
Intel Corporation

# Outline

## Episode

- I Parallelism 101
- II Hardware
- III Decomposition
- IV Specification of parallelism and synchronization
- V Correctness
- VI Practical issues

# Episode V: Correctness

- Get sequential version right first!
  - Design program so that there *is* a sequential version
  - Counter example: some producer-consumer queues
- Peter Principle for Programmers
  - Programs rise to level of their programmer's incompetence.
    - Forced by economic competition
    - Creeping featurism etc.
  - Reserve mental capital for dealing with parallel issues
    - Embarrassing parallelism is good

# Race Conditions

- These are the primary correctness headache.
  - Non-deterministic results
- Example:

Initial State  
int X = 0;

Thread 1  
t1 = X  
t1 = t1 + 1  
X = t1

Thread 2  
t2 = X  
t2 = t2 + 2  
X = t2

Final State  
 $X \in \{1, 2, 3\}$

# Somewhat Less Obvious

Thread 1  
++X

Thread 2  
X+=2

## Hidden

Thread 1  
OtherProgrammersRoutine(1);

Thread 2  
OtherProgrammersRoutine(2);

## Data Dependent

Thread 1  
++X[i];

Thread 2  
++X[j];

# Synchronization

- Low-level
  - Mutexes, condition variables, (dangerous) events
  - Atomic operations
  - Emphasis is on pair of threads
- High-level
  - Parallel loops
  - Pipelines
  - Barriers
  - Queues

# Flavors of Mutual Exclusion

- Mutex, critical section
  - Let one thread in at a time
- Semaphore
  - Let  $\leq N$  threads in at a time.
- Reader-writer lock
  - Let multiple readers or one writer in at a time
  - Useful when reading dominates writing
- Condition variables
  - Allows threads to wait for state protected by mutex to change, without holding the mutex.
  - And without creating timing holes!

# Example of Condition Variable (PThreads)

```
pthread_mutex_t mutex;  
pthread_cond_t cond;  
int count = 0  
int flipflop = 0;
```

```
pthread_mutex_init( &mutex ) ;  
pthread_cond_init( &cond ) ;  
... safe to use mutex and cond ...  
pthread_mutex_destroy( &mutex ) ;  
pthread_cond_destroy( &cond ) ;
```

```
pthread_mutex_lock( &mutex );  
if( ++count==n ) {  
    // Signal rest of threads that they can go.  
    count = 0;  
    flipflop ^=1;  
    pthread_cond_broadcast( &cond );  
} else {  
    int f = flipflop ;  
    // Wait for rest of threads  
    do {  
        pthread_cond_wait( &cond, &mutex );  
    } while( flipflop==f );  
}  
pthread_mutex_unlock( &mutex );
```

Atomically leaves mutex and waits on condition variable.

IEEE Std 1003.1-2001 allows a spurious wakeup, so check predicate. Google "pthread spurious wakeup" to learn more.

This simplified version does not protect against thread cancellation.

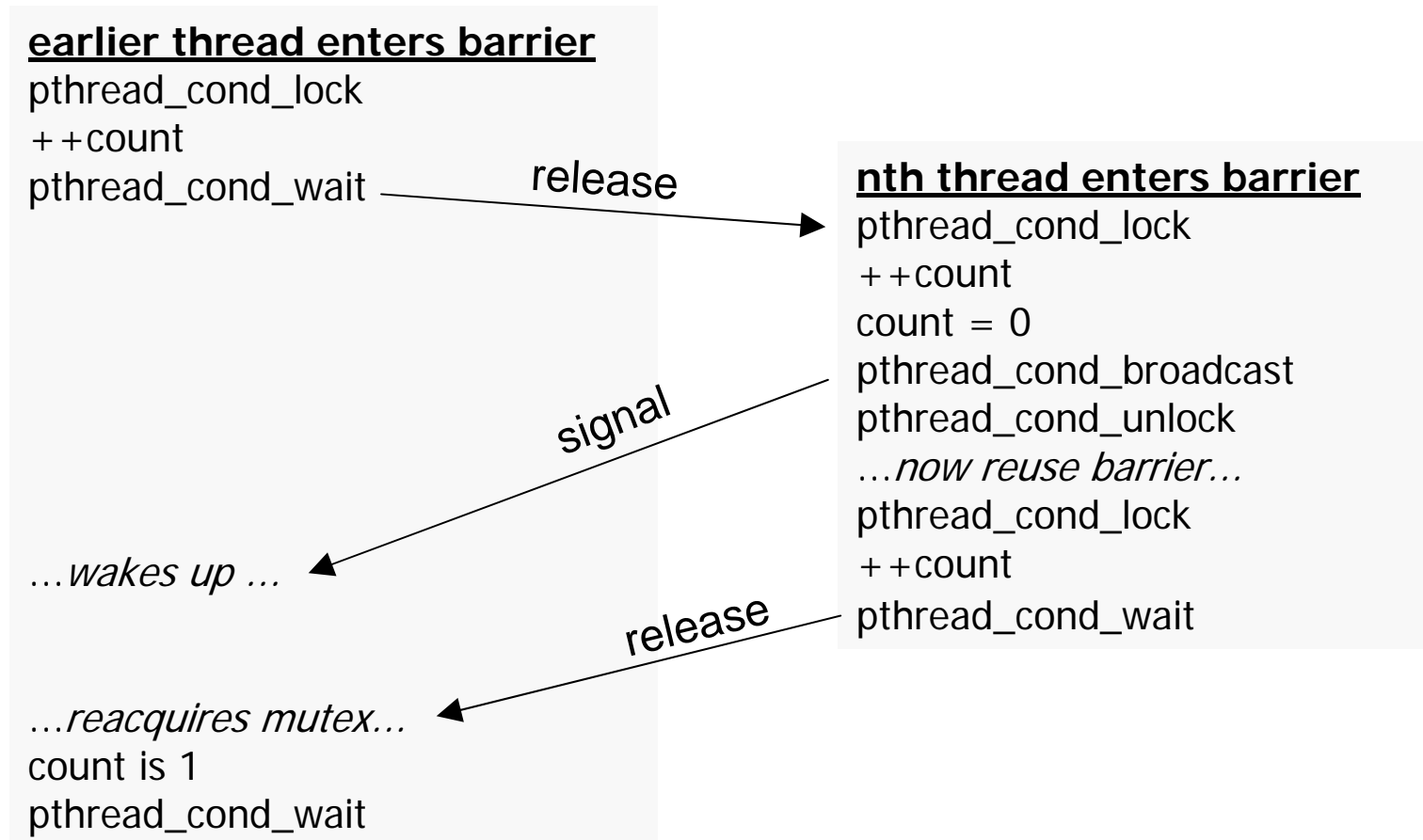
# What's Wrong With This Version?

```
pthread_mutex_lock( &mutex );  
if( ++count==n ) {  
    // Signal rest of threads that they can go.  
    count = 0;  
    pthread_cond_broadcast( &cond );  
} else {  
    // Wait for rest of threads  
    do {  
        pthread_cond_wait( &cond, &mutex );  
    } while( count );  
}  
pthread_mutex_unlock( &mutex );
```

Original version that I presented in in class ☹

Thanks to last-year's students who questioned it!

# Error = “Intercepted Wakeup”



See Section 7.1.1 of *Programming with POSIX Threads* by David R. Butenhof (Addison-Wesley, 1997) for a detailed description of how to implement barriers with conditional variables, including how to protect against thread cancellation.

# Condition Variables on Windows

- Sadly lacking from Windows\* for a long time
- SignalObjectAndWait became available as of NT 4.0.
- Reference that discusses the issues:

Strategies for Implementing POSIX Condition Variables on Win32

<http://www.cs.wustl.edu/~schmidt/win32-cv-1.html>

# Problems With Locks

- Composition
  - Locking lower level operations does not guarantee that higher-level operation is race-free.
- Deadlock
  - Cycle of threads that have each acquired a lock, and are waiting to acquire another's thread's lock.
- Convoying
  - If owner of lock is preempted, other threads wait behind it.
  - If owner of lock crashes, other threads wait forever
- Priority Inversion
  - Can occur with prioritized preemptive scheduling
    - Low-priority thread is preempted while holding lock
    - Medium-priority thread runs in preference to low-priority thread
    - High-priority thread waits forever on lock
  - Mars Pathfinder example  
[http://research.microsoft.com/~mbj/Mars\\_Pathfinder/Mars\\_Pathfinder.html](http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html)

# Composition Problem

- Composing thread-safe operations does not guarantee that result is thread-safe.
- Example: implementing a set using a thread-safe list
  - Invariant is “each key occurs once in list”

```
void add_if_not_present(key) {  
    if( !list.contains(key) )  
        list.append(key);  
}
```

Between the time we check for the key and insert it, another thread might append the same key.

Remember to lock outermost invariant.  
Locks in inside levels just waste time.

This is a challenge for reusable components.

# Avoiding Deadlock

- Try not to hold two locks at the same time
- Acquire multiple locks in same order
- Example orders
  - From “outer” to “inner” in nested system
  - Sorted by numerical address
  - Alphabetical (assumes each lock has a name)

# Recommendations on Mutexes

- Mutex should protect invariant(s)
  - E.g., “list contains no duplicate keys”
- Avoid exposing mutexes to other packages
  - Use private lock objects
  - Avoid holding lock while calling code in another package
  - Acquire locks in fixed order to avoid deadlock

# Atomic Operations

- Fetch-and-add
- Exchange
- Compare-and-swap
- Load linked conditional store
  - Not on IA-32 or IPF. Has implementation scaling issues.

# Example

- Fetch-and-Op atomically reads, modifies, and writes.
- Implement via Compare-And-Swap loop

```
temp = x;  
do {  
    old_x = temp;  
    temp = InterlockedCompareExchange (&x, Op(old_x), old_x);  
} while (temp != old_x);
```

comparand

returns old value of x

new value to store if x  
matches comparand

# Non-Examples!

- Depends on hardware and compiler
- Typical non-atomic accesses:
  - Read or write of structure type
    - Sometimes even if type occupies one word
  - Read or write of floating-point wider than one word
    - “double” on 32-bit machine
    - “long double” on 64-bit machine
- In principle, C/C++ standards do not guarantee that scalar types are accessed atomically
  - But in practice, it’s a safe bet.

# Problems with Lockless Algorithms

- Algorithms on non-trivial data structures are extremely difficult to understand
  - Every one is a new publishable result
- Livelock (non-termination)
- Cache line ping pong
- ABA problem
- Memory reclamation

# ABA Problem

- Lockless list manipulation using Fetch-And-Op
- Push onto list
  - Do Fetch-And-Prepend on root of list

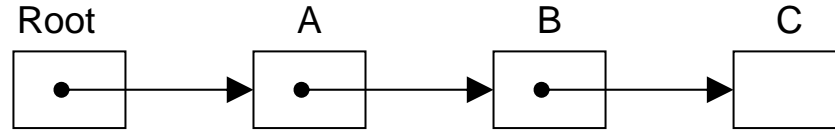
```
Node* Op ( Node* old_x ) {  
    new_item->next = old_x;  
    return new_item;  
}
```

- Pop from list
  - Do Fetch-And-(Set-To-Next)

```
Node* Op( Node* old_x ) {  
    front_item = old_x;  
    return old_x->next;  
}
```

WRONG IF NODES  
ARE RECYCLED!

# Erroneous Sequence



Thread 1 trying to pop

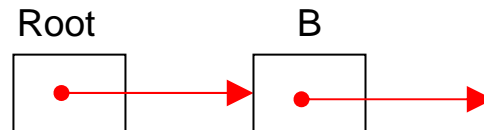
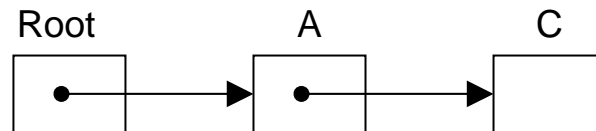
read Root ( $=\&A$ )

read A.next ( $=\&B$ )

compare-and-swap on  
Root succeeds!

Thread 2

pop A; pop B; push A



# Solutions

- Garbage collection
  - Never reuse a node (until garbage collector has determined that there are no more references to it)
  - “Hazard pointers” – like a mini garbage collector
- Tagged pointers
  - One word for pointer and one for serial number
    - Increment serial number as part of pointer update
    - Do back-of-envelope calculation on whether serial number can wrap
  - Requires double-width compare-and-swap or half-width pointer
    - IA-32: `cmpxchg8b`
      - Equivalent `cmpxchg16b` not yet available on Intel® EM64T
    - Itanium® processor: `cmp8xchg16`
      - Compares only 8 bytes (must be the serial number)
      - Only available on very recent chips

# Memory Reclamation

- It is often not safe to call `free()` on a node involved in a lockless algorithm
  - Might still be pointers to it by threads that are depending on compare-and-swap failing
  - Reallocation of node might fill field with value that makes compare-and-swap accidentally work
- There have been many proposed schemes
  - Sometimes subtlety broken
    - E.g., reference counting
  - Sometimes unrealistic assumptions
    - Double-word compare-and-swap instruction
    - Know number of threads up front

**Maged Michael, “Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects”, IEEE Trans. on Parallel and Distributed Systems, 15(6), June 2004**

# Recommendations on Lockless Algorithms

- These are a hot research topic, but have a long way to go before they become widely applicable.
  - Understand their limitations.
  - Do not fall for the hype.
  - Probably require “transactional memory” to go mainstream

Reference: <http://supertech.csail.mit.edu/xaction.html>

- Some trivial lockless algorithms worth using:
  - Counting
  - Fetch-and-update

# Memory Consistency

- Be very careful about accessing shared memory that is unprotected by a mutex.
  - Hardware can reorder accesses
  - Compiler can reorder accesses
    - volatile keyword in C/C++ is *not* sufficient (except on Itanium processors)
    - volatile keyword in C# (and recent Java\*) is sufficient (more or less)
- Definitions
  - sequential consistency
    - All operations happened as if in a particular total order
    - All observers agree on the order
  - relaxed consistency
    - Observers can disagree.
    - Enables significant hardware and compiler optimizations

# Example

```
// Initial state  
volatile int Message=0;  
volatile bool Flag=false;
```

```
// Thread 1  
Message = 42;  
Flag = true;
```

If writes are reordered,  
then Flag is set prematurely.

```
// Thread 2  
while( !Flag )  
    continue;  
if( Message !=42 )  
    abort();
```

If reads are reordered,  
Thread 2 uses out of date  
value of Message.

# Fence Instructions

- When writing lockless code, use fence instructions to prevent reordering
  - Mutexes typically imply these fences
  - So do not worry if using mutexes properly.
- Formal definitions of reordering rules are often hard to understand.
- Good rule of thumb:
  - Think of communicating via shared memory as sending and receiving messages.
  - Sender: write message; store fence; write flag
  - Receiver: read flag; load fence; read message

# Infamous Double Check (Done Correctly)

```
// Initial global state  
T* volatile Flag = NULL;  
volatile T Message;
```

Alas, there is no portable way to write this in C++.

```
// Double-check idiom  
T* f = Flag;  
_asm lfence;  
if( f ==NULL ) {  
    acquire lock  
    if( Flag==NULL ) {  
        Message = ...;  
        _asm sfence;  
        Flag = &Message;  
    }  
    release lock  
}  
...= *f // read Message
```

lfence ensures that Flag is read before Message on all execution paths.


sfence ensures that Message is written before Flag.

sfence unnecessary on IA-32 if “non-temporal” stores are not employed.

# Infamous Double Check (Itanium® Processor or .NET)

```
// Initial global state  
T* volatile Flag = NULL;  
volatile T Message;
```

Memory operations not allowed to move up past volatile read (acquire).

```
// Double-check idiom  
if( Flag==NULL ) {  
    acquire lock  
    if( Flag==NULL ) {  
        Message = ...;   
        Flag = &Message;  
    }  
    release lock  
}  
...= *f // read Message
```

Memory operations not allowed to move down past volatile write.(release)

# Java/C# Alternative

- “On Demand Holder” idiom
  - Exploit lazy initialization of static field
  - Language specification mandates thread-safety

```
class Holder {  
    // Languages dictate lazy initialization  
    public static T Message = new T();  
};
```

```
// Triggers initialization of Message on first use  
... = Holder.Message
```

# References on Memory Consistency

“Shared Memory Consistency Models: A Tutorial”, Sarita V. Adve and Kouros Gharachorloo,  
<ftp://gatekeeper.dec.com/pub/DEC/WRL/research-reports/WRL-TR-95.7.pdf>

<http://www.cs.umd.edu/~pugh/java/memoryModel/>

Java memory model - recently repaired.

“Memory Ordering”, Section 7.2 of IA-32 Intel® Architecture Software Developer’s Manual  
Volume 3: System Programming Guide

<http://www.intel.com/design/mobile/manuals/243192.htm>

C++ and the Perils of Double-Checked Locking, Scott Meyers  
and Andrei Alexandrescu,

[http://www.aristeia.com/Papers/DDJ\\_Jul\\_Aug\\_2004\\_revised.pdf](http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf)

Discusses why portable  
double-check idiom is  
*impossible* in current C/C++.

A Formal Specification of Intel® Itanium® Processor Family Memory Ordering

<http://www.intel.com/design/itanium/downloads/251429.htm>

“Memory Consistency & .NET”, Arch D Robison,  
Dr. Dobb’s Journal, April 2003.

Itanium and ECMA .NET have the  
most relaxed memory consistency in  
the industry. Understand them, and  
you have a solid grip on the issues.

# Correctness Tools

- Assertions
- Code Reviews
  - Many eyeballs and minds really helps
- Interactive breakpoint debuggers
  - Most debuggers understand threads to some degree
- Tracing
- SPIN Model Checker
  - Very useful for verifying lockless algorithms
- Memory checkers
  - BoundsChecker\*, Purify\*
  - Or use type-safe language (C#, Java) that eliminates memory errors
- Intel® Thread Checker
  - Finds potential races

# Heisenbugs

- Adding instrumentation (assertions, tracing) to program can change relative thread speeds
  - Possibly hide or expose race conditions.
  - Extremely frustrating when it happens
  - Try a different machine or optimization level until you can “shake loose” the bug.
  - Use tool like Intel® Thread Checker if possible
    - Finds *potential* races that actually did not happen in a particular run.

# Tracing Via Fast Circular Buffer

- ```
static unsigned long EventCount;

static const long N_Event = 1<<20;

struct Event {
    const char* what;
    const long arg1;
};

static Event EventArray[N_Event];

static void RecordEvent( const char* what, long arg1=0 ) {
    Event& e = EventArray[InterLockedIncrement(&EventCount) % N_Event];
    e.what = what;
    e.arg1 = arg1;
}
```

# Summary of Episode V

- Race conditions are what make writing correct parallel programs hard
  - Mutable shared state should be protected by mutual exclusion
  - Acquire multiple locks in the same order
  - Use reader-writer locks when appropriate
  - Use lockless algorithms only if you know what you are doing
- Use assertions
- Use tools to find race conditions
  - Intel® Thread Checker

# Episode VI: Practical Issues

- Miscellaneous advice for getting performance

# Back of the Envelope Calculations

- Before coding, estimate whether desired performance is plausible
  - Goal might be turn-around, throughput, or capacity
- Get sense on what resources will give out.
  - Computation, bandwidth, or I/O?
  - Might be able to trade one resource for another
  - Allow some slack
  - Rough estimates are usually optimistic

Jon Bentley, “The Back of the Envelope”  
<http://www.cs.bell-labs.com/cm/cs/pearls/bote.html>

Pop question: How old are you  
on your  $10^9$  sec. birthday?

# Example: CD filtering

- CD is 44,100 samples per second
- What if filter needs 512 x 64 multiply-adds per sample?
  - $44,100 \times 512 \times 64 = 1.44 \times 10^9$  multiply-adds/sec
  - On  $3 \times 10^9$  Hz processor, this probably does not leave enough slack for other stuff
  - Consider alternative algorithm, or use dual-core
- What if filter coefficients are each 64-bit floating point?
  - $8 \times 512 \times 64 = 0.25 \times 10^6$  bytes
  - Probably not enough slack on .5 MByte cache.
    - Depends on what else is going on
  - Look for ways to shrink filter. Or buy chip with bigger cache.
  - Might be able to hold in 1 MByte cache

# Keep Track of Your Dependences

- Write squeaky-clean readable code
- Minimize use of shared mutable state
  - Document dependences that you cannot avoid
- Protect hidden shared state with a lock
  - If you give client “separate objects”, it ought to be safe to use them as if they really are separate objects.
  - Example: use atomic reference counting if doing copy-on-write
- Document thread safety and non-safety
- Typical convention for object-oriented programming
  - Instance methods on the same instance are not thread safe
  - Static methods are thread safe
  - Unless otherwise specified (e.g. a concurrent hash table)

# Profile Your Code

- Use a profiler to find where time is spent
  - Do not waste time rewriting portions that contribute little to run time.
  - Intel's VTune™ analyzer has nice graphical interface for exploring call graphs.
  - Poor man's profiler
    - Stop program manually at random times
- If goal is turn-around time, find portions that are on critical path
  - These are what affect total time
  - Intel® Thread Profiler can determine critical path

# Parallelizing Legacy Code

- When faced with a loop nest
  - Try to parallelize at outermost level that does not have a cross-iteration dependence
    - Usually yields best grain size and least communication
  - Loop interchange can sometimes push the parallel loop outwards.
- Consider using OpenMP
  - It's designed for dealing with legacy loopy code.

# Know Platform Quirks

- Cache line size and false sharing
- Hyper-Threading 64K aliasing problem
- Windows non-scaling malloc/free
  - Consider scalable memory allocator
  - Examples:
    - SmartHeap\* (<http://www.microquill.com/>)
    - Hoard (<http://www.hoard.org>)

# Know Your Language And Compiler

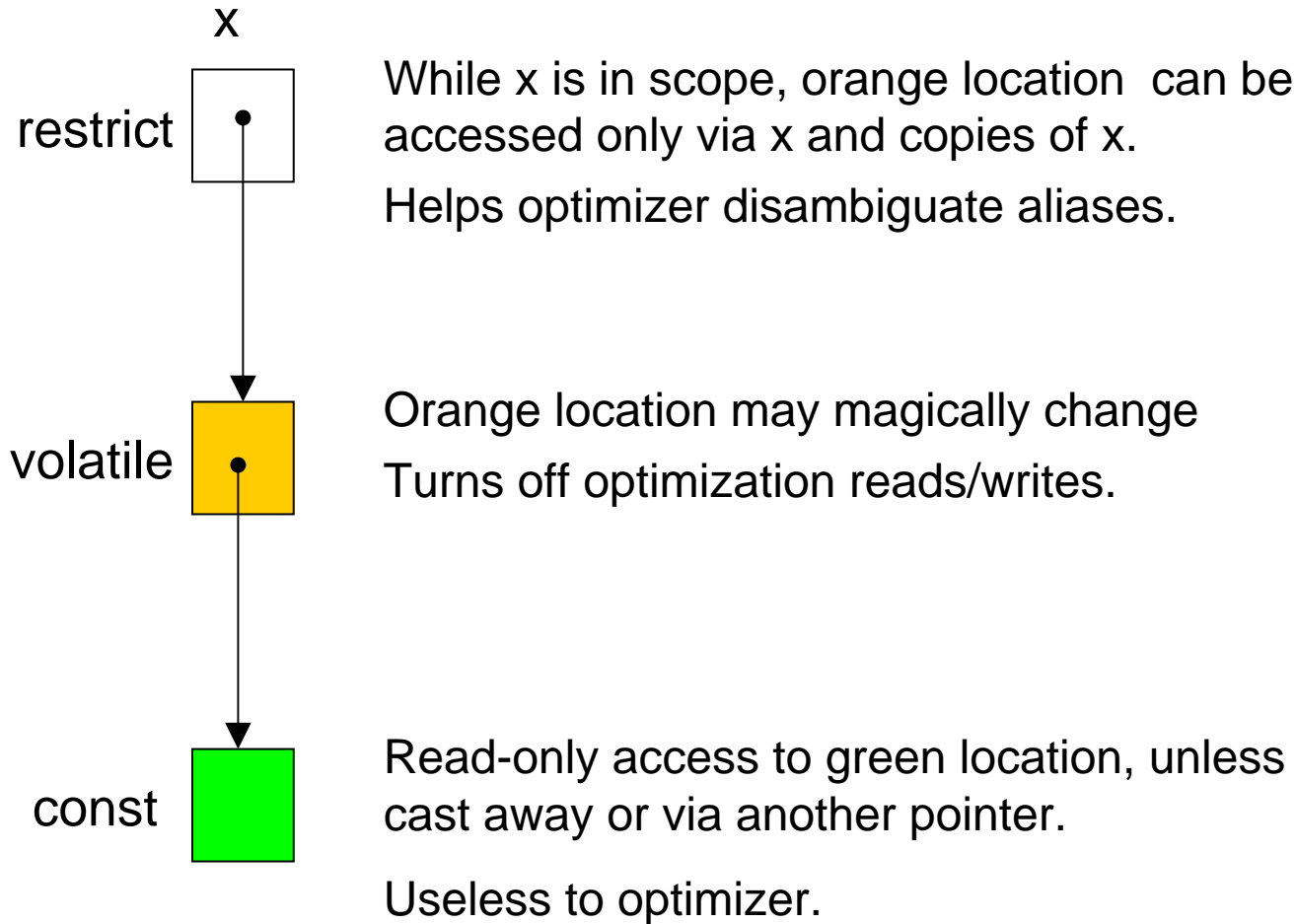
- Be nice to your compiler
  - Some hand optimizations are a nuisance now
    - E.g. pointer-bumping instead of indexing
    - Compiler can usually do better from subscript code
  - Rule of thumb: If its easy for you to read, it's easy for the compiler to read.
- Know when and where to use “volatile”

Pop quiz for people who think they are C experts:  
Describe what “const T \* volatile \* restrict x;” means.

Which qualifier:

- does not help optimization?
- prohibits optimization?
- may help optimization?

# Reading “const T \* volatile \* restrict x”



# Memory Contention

- Reads and writes to shared memory can be expensive
  - Cost of a cache miss
- It may pay to have a thread make a local copy
  - Copy data from shared to private (e.g. thread's stack)
  - Operate on private copy
  - Merge private copies at end (if copies are modified)

# Lock Contention

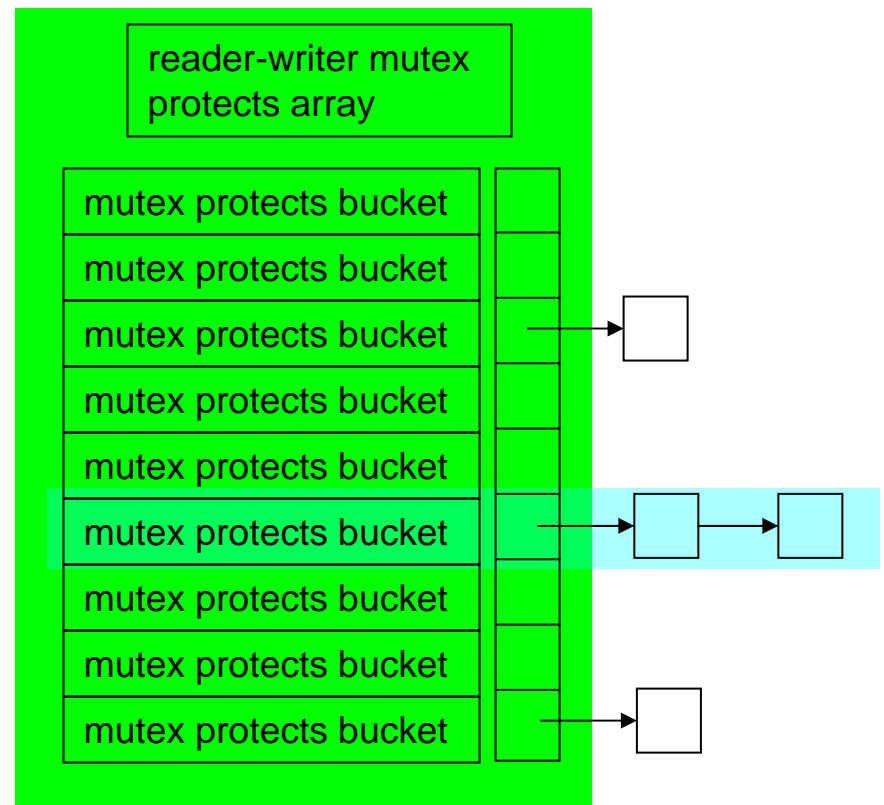
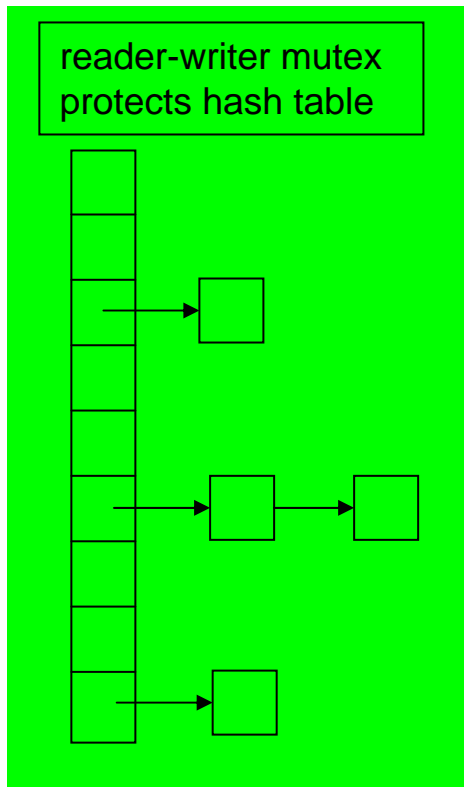
- Don't blame the mutex if program does not scale
  - A faster mutex might help by a constant factor, but it will not improve scaling
    - Atomic operations can improve constant factor for simple stuff like counters
  - By definition, there is no such thing as a scalable mutex!
    - Exception: readers in a reader-writer mutex
- Fix your decomposition to remove the contention
  - Try to spread contention among multiple locks
  - E.g., partition hash table into multiple subtables
    - Use hash function to map keys to subtables

# Coarse-Grain vs. Fine Grain Locking

(Scalability Costs a Constant Factor)

Coarse grain is simple and saves space, but serializes writers.

Fine grain enables concurrent writers, but requires more space. Threads must acquire two locks.



# Summary of Episode VI

- Estimate performance up front
- Know your tools and environment
- If at first you don't succeed, decompose again.