CS315A/EE382B: Lecture 2

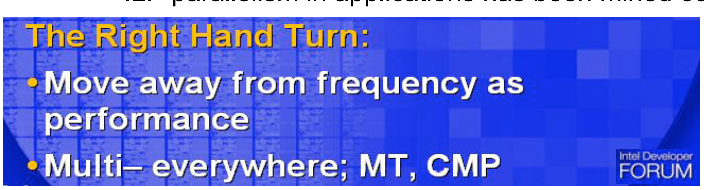
Shared Memory Programming

Kunle Olukotun
Stanford University

http://eeclass.stanford.edu/cs315a

Review: Single Processor Performance is Reaching Limits

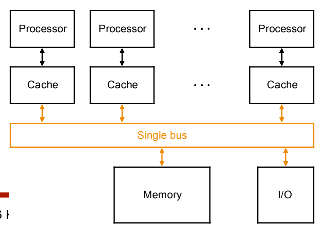
- This has been said before, but it is really happening now!
- ILP and deep pipelining have run out of steam:
  - Frequency scaling is now driven by technology
  - The power and complexity of microarchitectures taxes our ability to cool and verify
  - ILP parallelism in applications has been mined out

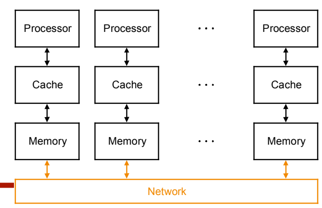


Now need parallel applications to take full advantage of microprocessors!

Review: Key Multiprocessor Questions

- How do parallel processors share data?
  - single address space: Symmetric MP (SMP) vs. NonUniform Memory Architecture (NUMA)
  - message passing: clusters, massively parallel processors (MPP)
- How do parallel processors coordinate?
  - synchronization (locks, semaphores)
  - built into send / receive primitives
- How are the processors interconnected?





Today's Outline

- **Threads:** How we divide an application into parallel regions
  - What is a thread?
  - How can we use them for parallel programming?
- **Locks:** How we control access to shared memory
  - Protecting critical variable accesses
  - Variations on basic locks
- **Synchronization:** How we sequence portions of threads
  - Barriers
  - Condition variables
- Mechanics of **pthreads** and **OpenMP** usage

Shared Address Model

- Each processor can access every physical memory location in the machine
- Each process is aware of all data it shares with other processes
- Data communication between processes is implicit: memory locations are updated
- Processes are allowed to have local variables that are not visible by other processes

Shared Memory vs. Message Passing

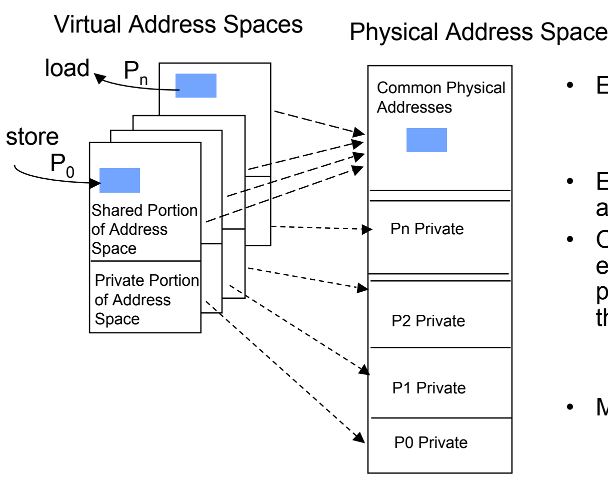
Feature	SM (OMP)	MP (MPI)
Apply parallelism in steps	YES	NO
Scale to large number of processors	MAYBE	YES
Code Complexity increase	small increase	major
Code length increase	2-80%	30-500%
Runtime environment	$ compilers	FREE
Cost of hardware	CHEAP-$$	$$

Threads

- A thread is any *independent control flow* through an application
  - Has its own program counter
  - Has its own active register space
  - Has its own stack
    - Local variables
    - Function call frames
  - Typically shares some heap variables with other threads

- Threads are used for many things:
  - Threads are often used to run "background tasks"
    - Spell checkers, I/O handlers, etc.
  - But we're going to use them to *partition* a single task
  - Two models have been used over the years . . . .

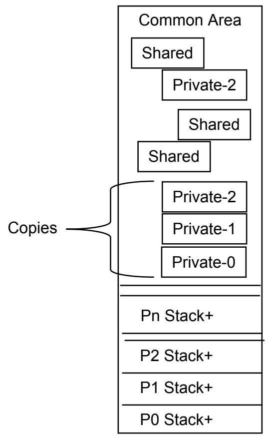
"Heavyweight" Thread Model



- Each thread is a UNIX process
  - Created with `fork()`
  - Expensive to create
- Each has its own virtual address space
- OS maps some pages from each process to common physical addresses to "share" those pages
  - With calls such as `shmget()` and `shmat()`
- Most memory non-shared
  - *Some* heap shared
  - Stack & globals separate

“Lightweight” Thread Model

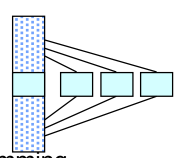
Common **Virtual** Address Space

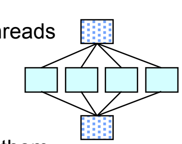


- Each thread is just a PC, registers, and stack
  - Often made with `pthread_create()`
- Usually *all* memory shared
  - Same page table, so no separation
  - Globals are completely shared
- **Pros:**
  - Easier sharing
    - No need for separate `malloc()` calls
    - Now pointer usage controls sharing
  - *Much* less OS overhead (factor 10–100x)
- **Con:** Non-shared data just by copying vars
  - Pointer errors may be able to corrupt other processors’ data (ouch!)

So how do we use them?

- First, figure out where there is parallel work in an application
  - Main topic of the next two lectures
- Next, choose a programming model
  - **Pthreads**: Low-level threading *library*
    - Uses fork-join model, like processes
    - Allows arbitrary code division
  - **OpenMP**: *Compiler directives* for parallel programming
    - Uses “parallel region” model to simplify threads
    - Divide up iterations of data parallel loops among threads
    - Is often much easier to use, but not as general
  - *Others:* Many other choices are available
    - System-specific threads: Solaris, NT, etc.
    - System-specific directives: Solaris compilers have them
    - Parallel languages: Java, HP Fortran, UPC, Cilk, Titanium, etc.





Simple Example

```
{
    /* NUM_PROCS Parallel Work Here */
}
```

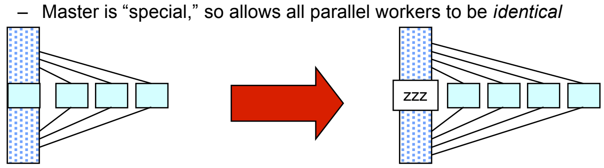
- How do I make this parallel?

Pthreads Example Implementation

```
#include <pthread.h>
main(){
   pthread_t p_threads[NUM_PROCS];
   input_buffer_struct_t input[NUM_PROCS]; /* Minimally, a processor ID */
   output_buffer_struct_t *output_bufptr;
   . . .
   for (i=1; i < NUM_PROCS; i++)
        pthread_create(&p_threads[i], &attr, Parallel_Work,
                (void *) &input[i]);
   Parallel_Work(&(input[0])); /* Optional */
   for (i=1; i < NUM_PROCS; i++)
        pthread_join(p_threads[i], &output_bufptr);
   . . .
}
void *Parallel_Work(void *input_buffer) {
   /* Parallel Work Here */
   return &output_buffer;
}
```

Subtle Pthreads Details I

- Number of threads ≠ number of processors
  - You *can* have a different number of threads and processors
  - Don't really need for simple parallel loops . . .
    - Just wastes memory & causes overhead
  - But useful in some cases:
    - Arbitrary forked-off parallel tasks (like database queries)
    - "Sleeping" master thread (used in book examples)
      - Master is "special," so allows all parallel workers to be *identical*



- A library, so just link to it (may require extra compiler flag)

Subtle Pthreads Details II

- You *can* pass data in/out through I/O pointers
  - Good for "processor ID," at least
  - Other values can be passed through shared variables easier
- Thread characteristics are controlled through an attributes struct
  - Similar to a C++ object
  - Set your preferences before creation
  - Important for us: Uniprocessor vs. Multiprocessor threads!
    - Solaris normally maps "background" threads to the same CPU
    - We want to have "parallel" threads on different CPUs
- Threads normally terminate when the parallel function returns
  - But you can end earlier with `pthread_exit()` (for self-kill) or `pthread_cancel()` (for "killing" other threads)
    - Just like UNIX `exit()` and `kill()` calls

What is OpenMP?

- OpenMP is a pragma based API that provides a simple extension to C/C++ and FORTRAN
- It is exclusively designed for shared memory programming
- *However, some vendors (Intel) are developing virtual shared memory compilers that will support OpenMP*
- Ultimately, OpenMP is a very simple interface to threads based programming

OpenMP: Where did it come from?

- Prior to 1997, vendors all had their own proprietary shared memory programming commands
- Programs were not portable from one SMP to another
- Researchers were calling for some kind of portability
- ANSI X3H5 (1994) proposal tried to formalize a shared memory standard – but ultimately failed
- OpenMP (1997) worked because the vendors got behind it and there was new growth in the shared memory arena

OpenMP Example Implementation

```
#include <omp.h>
main(){
   . . .
   #pragma omp parallel for default(private) \
        num_threads(NUM_PROCS) . . . << var info >> . . .
   for (i=0; i < NUM_PROCS; i++)
   {
        /* Parallel Work Here */
   }
   . . .
}
```

Important!!

- Simpler than pthreads for this basic `for` example
  - But harder for less structured parallelism (like webservers)
  - Just "attaches" to the following `for` loop & runs it in parallel
  - Be careful: These are *preprocessor directives!*

Loop Level Parallelism with OMP

- Consider the single precision vector add-multiply operation
  Y=a**X**+**Y** ("SAXPY")

```
for (i=0;i<n;++i) {
  Y[i]+=a*X[i];
}

#pragma omp parallel for \
     private(i) shared(X,Y,n,a)
for (i=0;i<n;++i) {
  Y[i]+=a*X[i];
}
```

Privatizing Variables

- Critical to performance!
- Simple in pthreads: Just use different variables!
  - Easy concept, but can sometimes complicate code
    - May require many `variable[processor_id]`-like accesses
- More work in OpenMP pragmas:
  - Designed to make parallelizing sequential code easier
  - Makes copies of "private" variables *automatically*
    - And performs some automatic initialization, too
  - Must specify shared/private per-variable in `parallel`
    - `private`: Uninitialized private data
    - `first/lastprivate`: Private, initialize@input & output@end
    - `shared`: All-shared data
    - `threadprivate`: "Static" private for use across several `parallel` regions

OpenMP Extras

- Parallel threads can also do different things with `sections`
  - Use instead of `for` in the `pragma`, and no attached loop
  - Contains several `section` blocks, one per thread
- You can also have a "multi-part" parallel region
  - Allows easy alternation of serial & parallel parts
  - Doesn't require re-specifying # of threads, etc.

```
#pragma omp parallel . . .
{
 #pragma omp for
 . . . Loop here . . .
 #pragma omp single
 . . . Serial portion here . . .
 #pragma omp sections
 . . . Sections here . . .
}
```

Race Conditions: A Concurrency Problem

- We must be able to *control* access to *shared* memory
  - Unpredictable results called races can happen if we don't (Eg. `x++`)

CPU 1	CPU 2
ld r1, x	**. . .**
add r1, r1, 1	**ld r1, x**
—	**add r1, r1, 1**
—	**st r1, x**
st r1, x	**. . .**

Dealing with Race Conditions

- Need mechanism to ensure updates to single variables occur within a *critical section*
- Any thread entering a critical section blocks all others
- Critical sections can be established by using:
  - Lock variables (single bit variables)
  - Semaphores (Dijkstra 1968)
  - Monitor procedures (Hoare 1974, used in Java)

Coordinating Access to Shared Data: Locks

CPU 1	CPU 2
ld r1, x	. . .
add r1, r1, 1	ld r1, x
—	add r1, r1, 1
—	st r1, x
st r1, x	. . .

→

CPU 1	CPU 2
LOCK X	. . .
ld r1, x	LOCK X
add r1, r1, 1	stall
st r1, x	stall
UNLOCK X →	unstall
	ld r1, x
	etc.

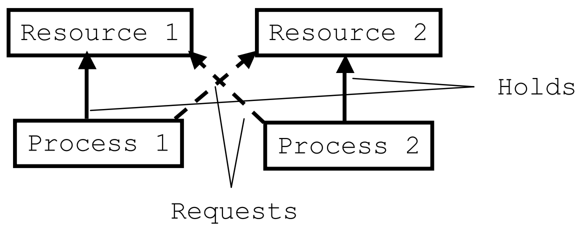
- Locks are a simple primitive to assert control
  - Put lock/unlock (acquire/release) pair *around* each critical region
  - Basis of all more complex variable control & synchronization
    - Semaphores, monitors, condition variables

"Fun" with Locking

- Basic idea of locks is simple:
  - Assign a lock to each shared variable (or variable groups)
  - *Initialize* the lock before you use it
  - *Always* use the lock when you access variables
- But details can get tricky:
  - Need to minimize the time processors spend stalled
  - Need to carefully select groupings of variables
    - Want to minimize # of locks to reduce overhead
    - But want to maximize available parallelism
  - Must be careful to always nest lock acquires correctly
    - Can cause **deadlock** if you're not careful!
  - *Moral:* Privatize as much as possible to avoid locking!

Deadlocks: The pitfall of locking

- Must ensure a situation is not created where requests in possession create a deadlock:



- Nested locks are a classic example of this
- Can also create problem with multiple processes - `deadly embrace'

Locks: Performance vs. Correctness

- Few locks
  - Coarse grain locking
  - Easy to write parallel program
  - Processors spend a lot of time stalled waiting to acquire locks
  - Poor performance
- Many locks
  - Fine grain locking
  - Difficult to write parallel program
  - Higher chance of incorrect program (deadlock)
  - Good performance
- Make parallel programming difficult
  - How do you know what level of lock granularity to use?
  - Will discuss further in upcoming lectures . . . .

Non-blocking Locks

- Structuring parallel programs correctly will be our main weapon against lock stall overhead
- But another one is *non-blocking* locks
  - Try to grab the lock, if possible
  - Do other, non-critical work if you can't get it

```
while (nonblocking_lock(&lock) != GOT_LOCK) {
  /* Do something else non-critical */
}
/* critical region here */
unlock(&lock);
```

- Performance is limited by availability of non-critical work

Locks in Pthreads and OpenMP

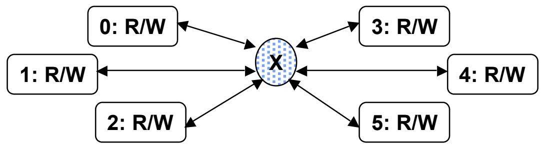
- Both have *equivalent* lock APIs:

Lock Task	Pthreads Version	OpenMP Version
Lock Object Type	`pthread_mutex_t`	`omp_lock_t`
Initialize New Lock	`pthread_mutex_init`	`omp_init_lock`
Destroy Lock	`pthread_mutex_destroy`	`omp_destroy_lock`
Blocking Lock Acquire	`pthread_mutex_lock`	`omp_set_lock`
Lock Release	`pthread_mutex_unlock`	`omp_unset_lock`
Non-blocking Lock Acquire	`pthread_mutex_trylock`	`omp_test_lock`

- For programming assignments, can define some macros and switch between the two with a `#define`
  - Avoid the OpenMP `critical` directive unless you're OpenMP-only (just a lock, but completely different syntax)

Lock Variations

- Basic locks are designed for *equal* read-write access:

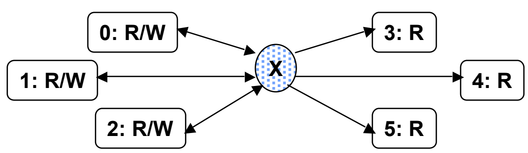


  - Only *one* processor can access at a time
  - Each has full read-write permission during its critical region

- But we may not always need this kind of access

Read-Write Locks I

- What if many of the processors only *read* shared data?



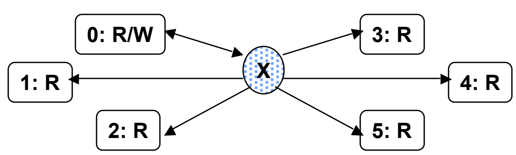
  - Only *one writer* should be able access at a time
  - Readers *could* all access simultaneously
    - But must also be locked out while a writer is accessing
  - Very useful for a structure that is being searched & updated
    - Searching processors only need read-only access
    - Get write access before updating the structure
    - Example: hash table

Read-Write Locks II

- Read-Write Locks are just normal locks + more
  - Need separate queues of waiting readers & writers
    - Pop 1 writer off at a time to give it access to the lock
    - Pop all readers off at once when no writers around
  - Straightforward implementation in pthreads
    - See §7.8.1 in text for an example
    - Uses *condition variables* to manage queues
  - Harder to build in OpenMP
    - Requires a fair amount of hand-built code to manage queues
    - No equivalent signaling mechanism
- RW Locks are very similar to cache coherence protocols
  - Will talk about these more in a few lectures!

Single-Writer "Locks" I

- What if one processor writes and others only *read* shared data?



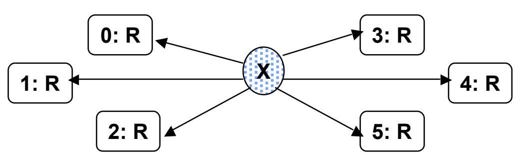
  - Locks *may* be avoidable!
  - Single writer can effectively "always have the lock"
  - For single variable, readers just get latest value written
  - Use locks if several values *must* be updated *together*
    - Without locks, readers may see partially-updated results
    - Use simplified R/W lock to lock out readers during updates
    - Should optimize so single writer can grab lock quickly

Single-Writer "Locks" II

- With no lock, need to occasionally *flush* writes out
  - Compiler may register-allocate values
  - Shared memory hardware *only* keeps *memory* coherent
  - We need to make sure writes are propagated out to memory
- Easy in OpenMP
  - Use `#pragma omp flush [variable-name]`
- No direct support in pthreads
  - Must pass through a mutex lock/unlock
    - These have *implicit* memory barriers included
  - Just acquire a "fake" lock
    - Just for memory barrier purposes, doesn't protect anything

Read-only Shared Data

- What if shared data is only read?



  - Some data isn't updated during every program phase
    - Writes are all on the other side of a "barrier"
  - No locking needed, since no updates
    - Barrier synchronization effectively acts like a lock
  - Use replication where possible to avoid this
    - Could have cache problems (false sharing)
    - But useful for large data structures

Coordinating Access to Shared Data: Synchronization

- We often want to control *sequencing* of *parts* of threads:
  - To impose a sequential order on a code block
    - When a few lines just can't be parallelized
  - To wake up stalled threads
    - When stalled at a lock, for example
  - To control producer-consumer access to data
    - Producer signals consumer when output is ready
    - Consumer signals producer when it needs more input
  - To globally get all processors to the same point in the program
    - Divides a program into easily-understood *phases*
    - Generally called a **barrier**

Simple Problem

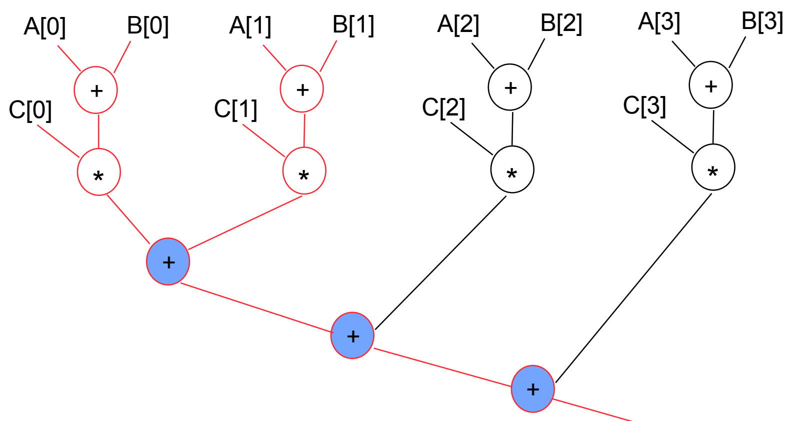
```
for i = 1 to N
      A[i] = (A[i] + B[i]) * C[i]
      sum = sum + A[i]
```

- Split the loops
  - Independent iterations

```
for i = 1 to N
      A[i] = (A[i] + B[i]) * C[i]
for i = 1 to N
      sum = sum + A[i]
```

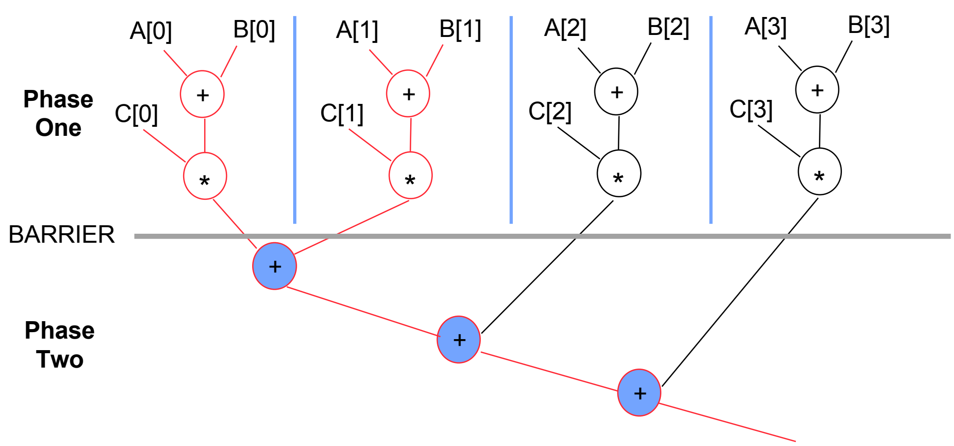
- Data flow graph?

Data Flow Graph



2 + N-1 cycles to execute on N processors
what assumptions?

Partitioning of Data Flow Graph



Barriers: Pros & Cons

- **Pro:** Program phases *ease debugging*
  - Eliminates cases of processors in different code regions
    - Otherwise we may have to consider nasty race conditions!
  - Generally easier to reason about
- **Pro:** Program phases *reduce the need for locks*
  - Only need to use the strongest type of lock *for that phase*
    - Normal or full R/W when many/everyone is modifying
    - Switch to single writer lock or read-only when possible
    - *Example:* A[i] array is *read-only* in phase 2 of example!
  - Can eliminate most of lock overhead for large structures
- **Con:** OVERHEAD
  - "Fast" processors are stalled waiting at the barrier
  - Barrier code itself can be expensive (see §7.8.2!)

OpenMP Synchronization

- OpenMP provides for a few useful "common cases"
- `barrier` implements an arbitrary barrier
  - A barrier anyplace in one line!
  - Note that many other primitives *implicitly* add barriers, too
- `ordered` locks *and* sequences a block
  - Acts like a lock around a code block
  - Forces loop iterations to run block in "loop iteration" order
  - Only one allowed per loop
  - Good for handling reductions manually, when necessary
    - `sum[i] = sum[i-1];`
- `single/master` force only *one* thread to execute a block
  - Acts like a lock
  - Only allows one thread to run the critical code
  - Good for computing a common, global value or handling I/O

Pthreads Synchronization: Condition Variables

- Pthreads offers a lower-level interface to synchronization: *Condition Variables*
  - Provide simple "can I go?" and "go now" signaling calls
    - Should be thought of as "go if X" and "X has changed"
  - Can be used to build:
    - Barriers
    - Producer-consumer queues
    - Read-write locks
    - And just about any other communication primitive . . . .
- Is tied implicitly to a single lock & flag variable
  - Lock protects the condition variable during use
  - Flag allows condition to be tested independently

CV API: Test-and-Wait

- `pthread_cond_wait(CV, lock)` to say "can I go?"
  - Always use *inside* the associated lock
  - Always use in a `while` loop that tests the flag variable:

```
while(!flag)
  pthread_cond_wait(&my_cv, &my_cv_lock);
```

CPU 0	CPU 1	CPU 2
Acquire lock ←	**pthread_cond_wait**	
flag = TRUE	. . . waiting . . .	. . .
Release lock →	CV Released! →	Acquire lock
. . .	. . . stalled . . .	flag = FALSE
	Acquire lock ←	Release lock
	Oops! flag FALSE	. . .

- `pthread_cond_timedwait(CV, lock, time)` limits waits
  - Allows you to do something else after awhile

CV API: Signaling

- `pthread_cond_signal(CV)` to say "next CPU go!"
  - Always use within the lock (*writing* to CV!)
  - Always *set the flag variable* before leaving the lock
- `pthread_cond_broadcast(CV)` to say "all CPUs go!"
  - Same restrictions as above
  - Useful for building barriers, but . . .
  - Still a delay after broadcast due to readers getting lock
    - All broadcast receivers must serialize on the lock acquisition
    - Could be lengthy if a lot of receivers
  - May want to consider a single-writer model in this case
    - Single written flag can eliminate serial reader locks
    - Useful if readers aren't interested in critical region anyway

Summary & A Look Ahead

- Three main portions of SM programming models
  - Threads to divide up work
  - Locks to protect shared data
  - Synchronization primitives for sequencing threads
- These constructs are the basis of shared memory programming
  - All SM assignments will build upon this
  - Some assignments will have you examine details
- Will continue on to see how these work in full applications
  - Dividing up applications into threads
  - Dividing up data to minimize communication and synchronization
  - Avoiding common bugs