524P

Parallel Programming

Mark Oskin
oskin@cs.washington.edu
206-293-9456 (text only)

Emily Furst
eafurst@cs.washington.edu

Renat Bekbolatov
renatbek@cs.washington.edu

Schedule

- (10/3) Shared Memory Multiprocessing (threads & locks)
- (10/10) Advanced SM: Transactions, Cilk, OpenMP, etc
- 10/17 - **No class (MICRO conference)**
- 10/24 - GPUs: Vectors, OpenCL, C++Amp, CUDA
- 10/31 - MPI
- 11/7 - *possibly no class, have to play it by ear for now*
- 11/14 - PGAS languages
- 11/21 - Map/Reduce
- 11/28 - TBD (FPGAs? Cancel because of Thanksgiving++?)
- 12/5 - TBD (DSL's?)

Four assignments:

- Write: dense Matrix-Vector multiply, and breadth first search graph traversal on these four programming stacks:
  - threads/locks
  - GPU
  - MPI
  - Map/Reduce/Spark
- You MUST do the threads, MPI and Map/Reduce/Spark assignments in the VM. You can do the GPU one on whatever platform you have.
- You are graded on correctness and efficiency of implementation.
- More details forthcoming.

In the book

- Note, there is no great book on this topic.
- I tried to choose one that I thought would be relevant to most people in this class long after you left here.
- We will cover a lot of ground in this class. About 3/4s of the material is *not* in any (one) or any book at all.
- Read Chapters 1,3 & 6 quickly. But read the rest at your leisure/need.

There is no final exam in this class.

Your grade is based on your assignments ***AND*** **your participation in class**

SPEAK!

Who am I?!?

Who are you?

- Companies
  - 1/2 MSFT
  - 1/5 AMZN
  - 1/6 BOEING
  - 1 salesforce
  - 1 oracle
  - 1 Facebook
  - 1 GOOG
- About 1/3 work on parallel code
- Most all in non-scripting languages (C, C#, java)
- 1/8 have used a map/reduce style framework
- Handful MPI
- 1/4 have used threads and locks
- ~ 5 of us get up before 6am

What hardware/OS are you going to use for this class?

- Mostly window
- And then Mac
- handful run Linux

Why do we have more than one CPU?

- Only so many transistors
- So we can be lazier programmers
- So we can do more than one thing at a time
- Heat
- Cheaper to scale horizontally than vertically
- Speeds of a single core stopped increasing

Topics for today/next week

- Threads
- Locks (Mutex)
- Semaphore
- Reader/Writer lock
- Condition variables
- Barrier
- Monitors
- Lock free (“Live free or Die!”)
- Consistency
- User-mode threads
- OpenMP
- Transactions

What is a thread?

- Lightweight process
- Stack + IP
- Sequential execution of a list of instructions
- Something you can map onto a processor

- Hardware: IP, register set, misc context, address space
- Language: (hopefully) a well defined execution context
- OS: something you can schedule and run

Example

Common bugs with threads

- Failure to join
- Race on start state
- Failure to synchronize on library calls (POSIX 1 vs. POSIX 1c (1996))
- Using shared memory incorrectly (much more on this)

Common performance problems with threads

- Too fine
- Too coarse
- Too few
- Too many
- Move around too much (no affinity)

Stretch!

What is a lock?

- Mechanism to prevent other threads from using a resource

- Piece of shared memory to achieve mutual exclusion

- Meeting point for threads

- data-structure to maintain ownership

- Something that can be “owned” by only one thread

A (broken) lock implementation

```
void lock(int *the_lock) {
 while (*the_lock == 1)
   ;
 *the_lock = 1;
}
```

```
void unlock(int *the_lock) {
 *the_lock = 0;
}
```

How are locks implemented?

- Core necessity: *Either* a bit of private state per acquirer (Petersen lock), ***Or***, an **atomic read/write** operation (how we tend to do it in SM systems)

- Processors tend to support a variety of atomic operations useful for constructing locks:

  - LOCK XCHG # Exchange

    - Swap a register and memory location value

  - LOCK CMPXCHG # Compare and xchange

    - Write to memory a register value if and only if the memory is equal to a given value

A lock implementation

```
void lock(int *the_lock) {
 while (__sync_val_compare_and_swap(the_lock, 0, 1)
  == 1)
  ;
}
```

```
void unlock(int *the_lock) {
 *the_lock = 0;
}
```

A slightly better implementation

```
void lock(int *the_lock) {
 while (__sync_val_compare_and_swap(the_lock, 0, 1)
  == 1)
  asm volatile ("pause");
}
```

A slightly better implementation

```
#define MAX_BACK_OFF   (1<<12)

void lock(int *the_lock) {
 int back_off = 1, i;
 while (__sync_val_compare_and_swap(the_lock, 0, 1)
  == 1) {

  for (i = 0; i < back_off; i++)
    asm volatile ("pause");

  if (back_off <= MAX_BACK_OFF)
    back_off = back_off << 1;
}
```

A slightly slightly better implementation

```
#define MAX_BACK_OFF   (1<<12)

void lock(int  *the_lock) {
 int back_off = 1, i;
 while (1) {
  // TEST
  while (*the_lock != 1) {
   asm volatile (“pause"); // Tell the CPU we are spinning
   asm volatile ( ::: "memory"); // address expose
  }
  // TEST AND SET
  if (__sync_val_compare_and_swap(the_lock, 0, 1) == 0)
      return;
  // BACK OFF
  for (i = 0; j < back_off; i++)
    asm volatile ("pause");
  if (back_off <= MAX_BACK_OFF)
    back_off = back_off << 1;
}
```

An even slightly slightly better implementation

```
#define MAX_BACK_OFF   (1<<12)
#define CACHE_LINE_SIZE (64)

typedef union _lock {
    int lock_state;
    char padding[CACHE_LINE_SIZE];
} lock;

void lock(lock *the_lock) {
 int back_off = 1, i;
 while (1) {
  while (the_lock->lock_state != 1) {
   asm volatile ("pause");
   asm volatile ( ::: “memory");
  }
  if (__sync_val_compare_and_swap(the_lock->lock_state, 0, 1) == 0)
   return;
  for (i = 0; j < back_off; i++)
    asm volatile ("pause");
  if (back_off <= MAX_BACK_OFF)
    back_off = back_off << 1;
}
```

Example

Common bugs with locks

- Failure to use one
- Failure to initialize them
- Failure to acquire all of them that you need
- Failure to unlock
- Releasing them too early
- Failure to acquire in a consistent order (deadlock)
- Acquiring them twice
- Releasing them when a thread ends abnormally
- Holding a lock over blocking I/O that may block “indefinitely”

You should never deadlock

- It’s possible to write a piece of code known as a “lock witness”.

- Group locks into classes.

- Classes can only be acquired in order

- An acquisition of locks out of order, *regardless of whether a deadlock occurred,* is detectable and should be signaled to the developer.

- Every large project needs a lock witness. Go take it from FreeBSD.

Common performance issues with locks

- Holding them for too long a time
- Holding them for too short a time
  - (acquiring/release the same lock in an inner loop)
- Holding more of them than you actually need
- Spinning when you should be queueing
- Queuing when you should be spinning
- Multiple locks in the same cache-line

Stretch!

Semaphores

- Generalization of locks

```
typedef int semaphore_t;

down(semaphore_t *x) {
 while (1) {
  atomic {
    if ((*x) != 0) {
      —(*x);
     return;
    }
}

up(semaphore_t *x) {
  atomic {
   ++(*x);
  }
}
```

Your turn!

Reader/Writer locks

- Basic idea: some data-structures support concurrent reads, but not concurrent writes.

    - hash-tables, trees, maps, whatev’s.

- A reader/writer lock permits multiple readers but only one writer.

  ```
  pthread_rwlock_rdlock(pthread_rwlock_t *);
  pthread_rwlock_wrlock(pthread_rwlock_t *);
  pthread_rwlock_unlock(pthread_rwlock_t *);
  ```

Example

A good interview question…

Implement a *fair* reader/writer lock

Why is this a hard question?

How would you solve it?

```
struct rw_lock {
    lock    reader_lock;
    lock    writer_lock;
    int     readers;
};

void lock_rd(struct rw_lock *rwl) {
    lock(&rwl->reader_lock);
    ++rwl->readers;
    if (rwl->readers == 1)
        lock(&rwl->writer_lock);
    unlock(&rwl->reader_lock);
}

void unlock_rd(struct rw_lock *rwl) {
    lock(&rwl->reader_lock);
    --rwl->readers;
    if (rwl->readers == 0)
        unlock(&rwl->writer_lock);
    unlock(&rwl->reader_lock);
}

void lock_wr(struct rw_lock *rwl) {
    lock(&rwl->writer_lock);
}

void unlock_wr(struct rw_lock *rwl) {
    unlock(&rwl->writer_lock);
}
```

Stretch!

Condition Variables

- Probably the least understood but most important synchronization primitive there is.

- pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *lock);

    - Thread **must** hold the specified lock. The thread **releases** the lock and is **blocked** until the condition is **signaled**.

- pthread_cond_signal(pthread_cond_t *cond);

- pthread_cond_broadcast(pthread_cond_t *cond);

Condition variables

- Condition variables are important because they all you to synchronize without spinning, and hence wasting resources.
- Condition variables can be extremely challenging for programmers to use but if you follow two simple rules then you will be fine:
  - 1) **A signal before wait is lost.** Always acquire the mutex the wait will wait on, and then signal.
  - 2) When you are signaled, you are **not necessarily assured the condition you really want to be true is true**; it depends on the condition, the number of waits, what they do, etc.

Example

Barriers

- Barriers synchronize a *group* of threads.
- Threads are stopped until all threads that should enter the barrier have done so.
- pthread_barrier_init(pthread_barrier_t *barrier, attr, unsigned int count);
- pthread_barrier_wait(barrier);

Your turn!

```
barrier_init(struct barrier *barrier, int waiters) { }

barrier_wait(struct barrier *barrier) { }
```

Barriers, continued

- Should barriers support a barrier_signal or barrier_broadcast, “early release” mechanism?
- Should barriers support a time-out?

Common bugs/issues with barriers

- A wayward thread never enters
  - Perhaps it’s waiting on a lock or data to be produced from a thread that has already entered the barrier!
- *Advice:* Barriers are useful at the mesoscale of your code.
  - If you enter the barrier from different points in your code, think hard, but it still might be ok
  - If you enter a barrier after executing a giant section of code, think hard, but it still might be ok.

Monitors

- When I write data-structures for parallel systems I leave the locks out; **I make no promises to the rest of the code about the safety of concurrent use.**

- But others may disagree…

- A *Monitor* is (essentially) one big lock around a class that is implicitly acquired/released on invocation of a function of that class.

- Sounds simple right?

What makes monitors hard to implement & use…

- Class functions call other class functions:

```
monitor_class A {
 int my_function1() { B.my_function3(); }
  int my_function2(…) { my_function1(); }
…
```

- Class functions call functions from other classes that may use functions of the given class:

```
monitor_class B {
   int my_function3() { A.my_function2(); }
```

- Synchronization within the monitor

```
monitor_class C {
  int add(item p) { stack.push(); }
  item remove() { while (stack.empty()) ;
     return stack.pop(); }
```

Lock-Free Algorithms

- There is an important concept in parallel programing, or rather, a concept some people think is important, known as lock-free algorithms
  - *Lock-free does not mean synchronization free*
- Lock-free generally means a data-structure is designed such that synchronization becomes inherent in the manipulation of the data.
- I'm pushing 1MM lines+ in my lifetime so far. I do not write lock-free data-structures. I do write "lock free" flags now and then but less so as I get older and hopefully wiser.

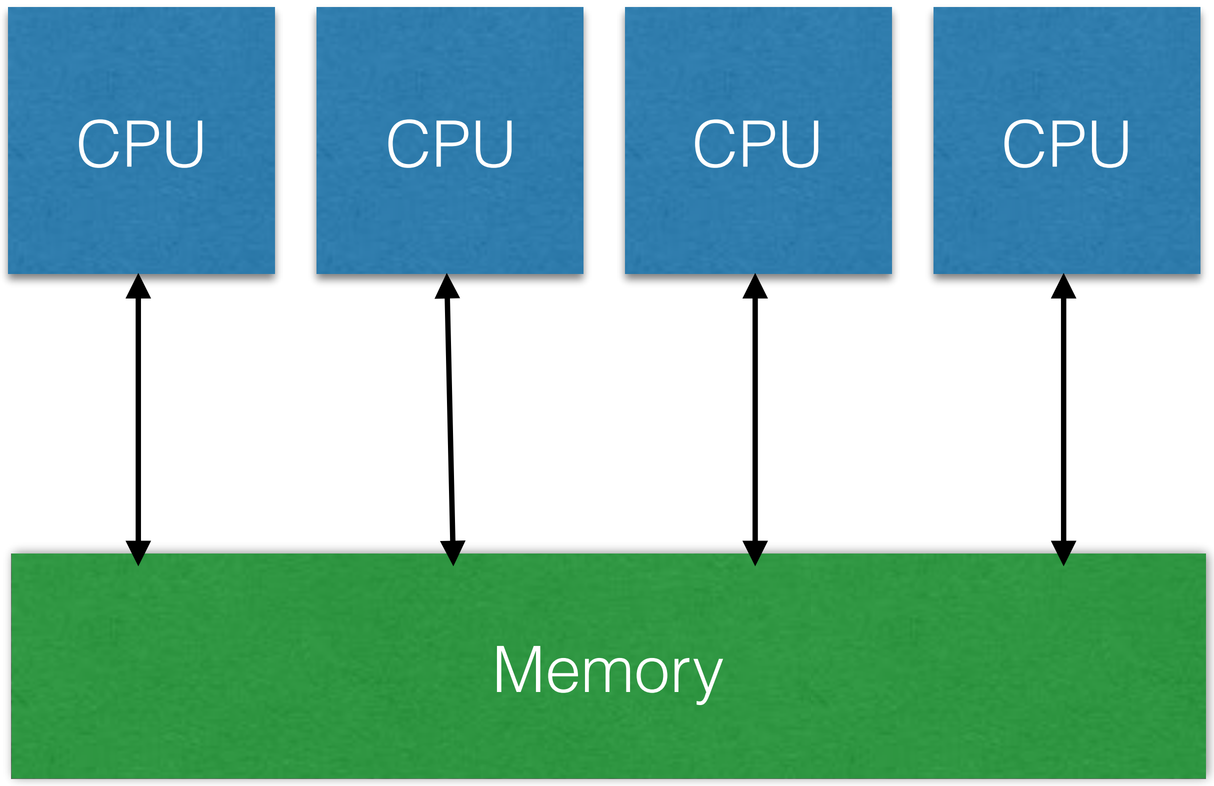
Example

Lock-free issues

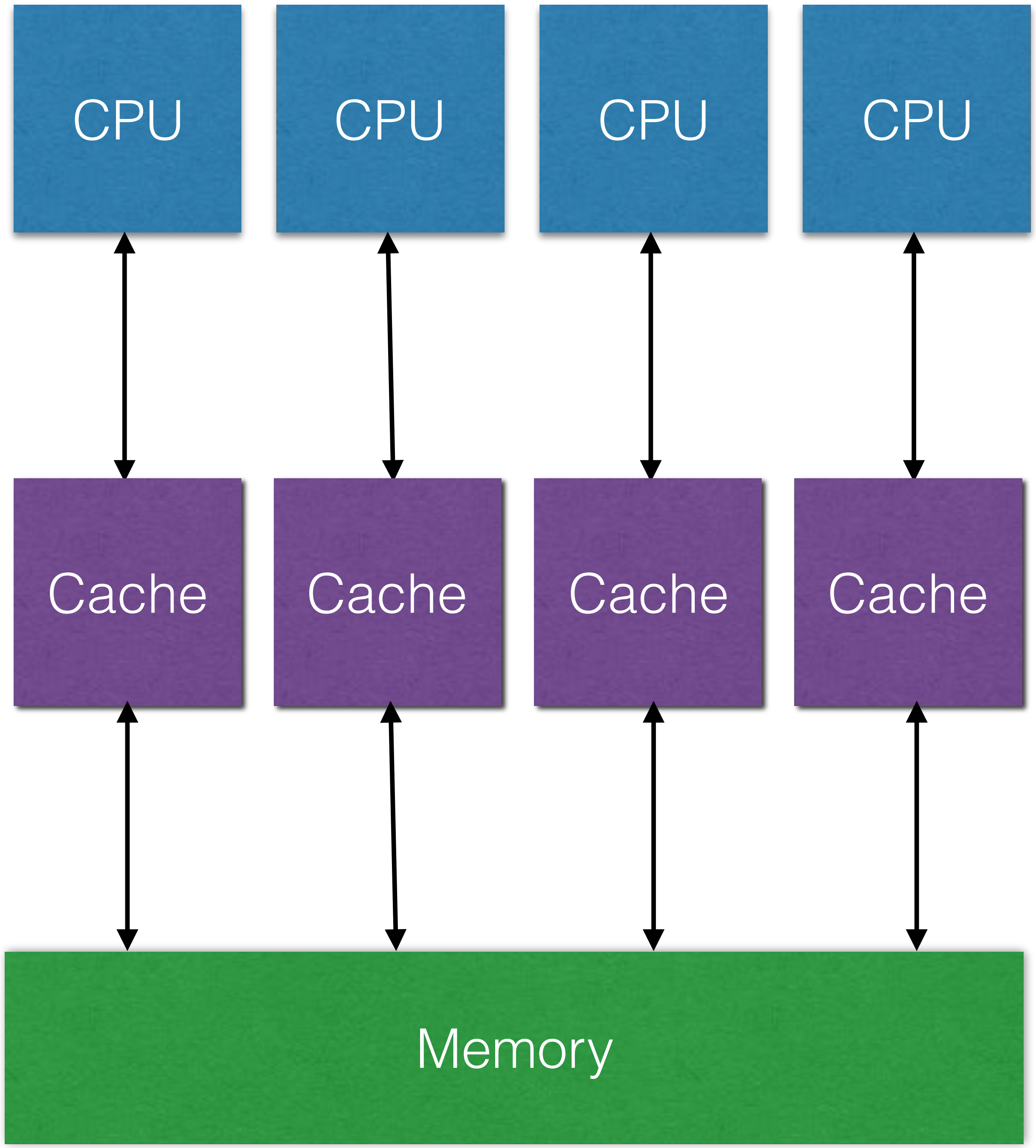
- Just being lock-free does not make you fast
- Lock-free is very challenging to get correct. Think of it as the extreme end of super-fine-grained locking
- Lock-freedom *does not mean* wait-freedom. Wait-free algorithms (which are necessarily lock-free) do exist for some things
- People still write papers that appear in top places on lock free algorithms. Frankly I think they are bit like doing algorithmic research on quantum computing. Hard, and maybe relevant some day, but not so much now. (But then again, I'm starting to get back into QC…)

Stretch!

How programmers want to see the world



How programmers need to see the world



Consistency

- Unfortunately for you, hardware doesn't behave like you think.
- Up until now, we've been writing code that will execute correctly on x86, ARM, etc.
- Broadly speaking, there are four consistency models in the world:
  - sequential, how you think concurrency works until you're told it's not so
  - processor, how x86 works, except for where it doesn't
  - release, how ARM works and a few other esoteric ISAs
  - scope (used for GPUs).
- The consistency model presented to the programmer is a function of the **language.** It is more or less easy to implement that consistency model on different hardware.
  - Easy for an academic to say. In practice, it's the wild west out there and you have to know what you are working with.
- C11 and C++11 are what is known as SC for DRF programs; or sequentially consistent for "data race free" programs.

Say what?

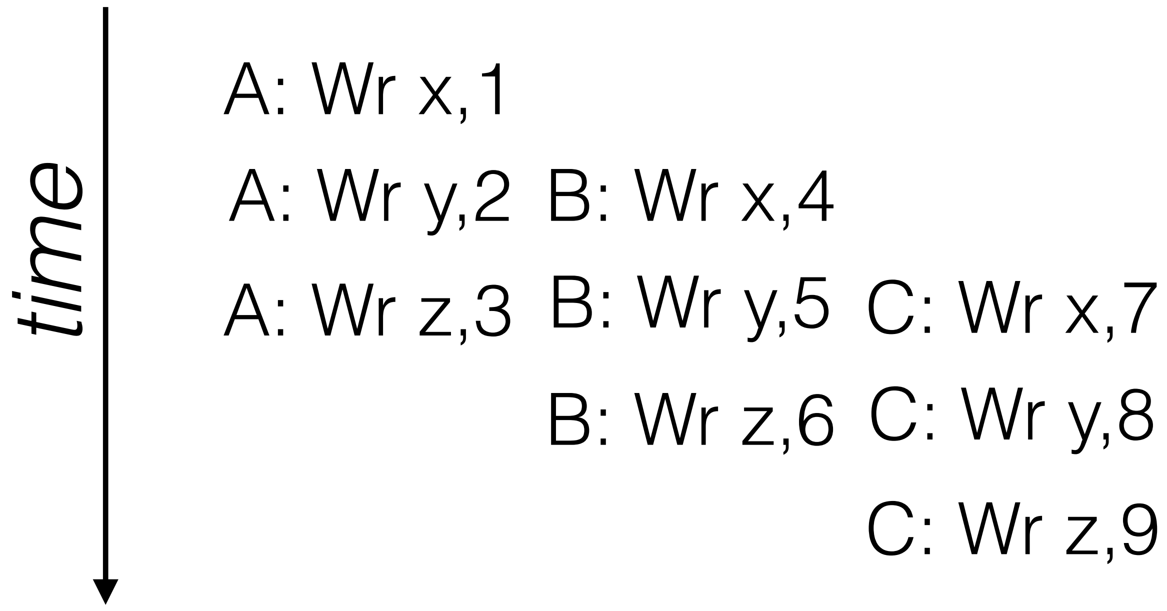
```
int a = 1;
int b = 2;

thread1() {
  a = 3;
  b = 4;
}

thread 2 {
  printf(“%d”, b);
  printf(“%d”, a);
}
```

What are the possible outputs?

Sequential Consistency



A: Wr x,1

A: Wr y,2

B: Wr x,4

B: Wr y,5

A: Wr z,3

C: Wr x,7

C: Wr y,8

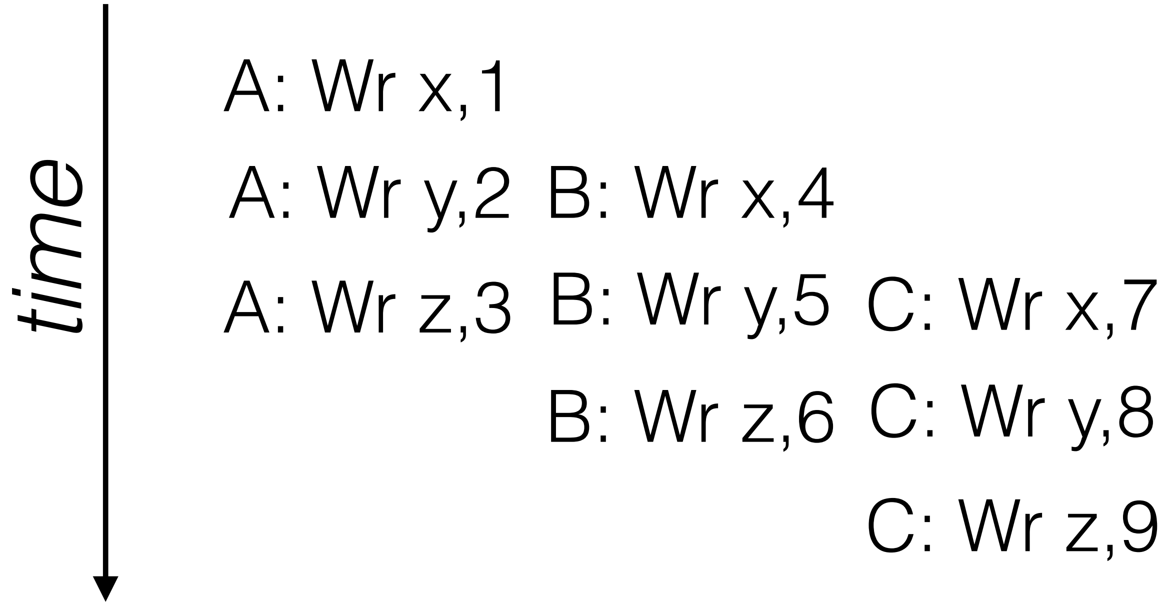
B: Wr z,6

C: Wr z,9

All processors see all writes in the *same* order

This is just one possible SC order.

Processor Consistency

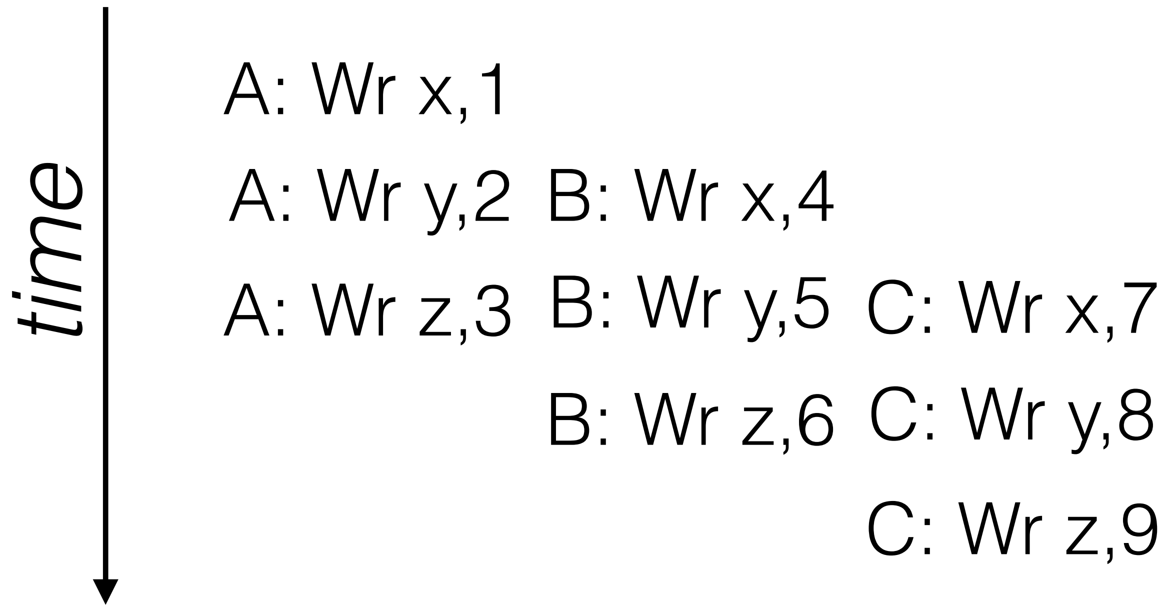


A: Wr x,1		
A: Wr y,2	B: Wr x,4	
A: Wr z,3	B: Wr y,5	C: Wr x,7
	B: Wr z,6	C: Wr y,8
		C: Wr z,9

How A can see the world	How B can see the world
A: Wr x,1	A: Wr x,1
A: Wr y,2	B: Wr x,4
B: Wr x,4	A: Wr y,2
B: Wr y,5	B: Wr y,5
A: Wr z,3	C: Wr x,7
C: Wr x,7	A: Wr z,3
C: Wr y,8	B: Wr z,6
B: Wr z,6	C: Wr y,8
C: Wr z,9	C: Wr z,9

A processor sees *all* writes from another processor in the order that processor performs them

Release Consistency



A: Wr x,1

A: Wr y,2 B: Wr x,4

A: Wr z,3 B: Wr y,5 C: Wr x,7

B: Wr z,6 C: Wr y,8

C: Wr z,9

A: Wr x,1
A: Wr y,2
B: Wr x,4
B: Wr y,5
A: Wr z,3
C: Wr x,7
C: Wr y,8
B: Wr z,6
C: Wr z,9

How A can see the world

A: Wr x,1
B: Wr x,4
A: Wr z,3
A: Wr y,2
B: Wr y,5
C: Wr x,7
B: Wr z,6
C: Wr z,9
C: Wr y,8

How B can see the world

More or less, release consistency promises nothing without a fence

What would Brian Boitano do?

- **Don’t write code that synchronizes outside of existing synchronization primitives**
    - Or if you must, wrap your shared memory accesses with fences: MFENCE (fence it all), LFENCE (load fence), or SFENCE (store-fence).
- In my experience, this kind of stuff will not make your code run fast. The problems that hinder parallel code bases are always much bigger than the micro-scale synchronization that is used.
    - **Go for maintainability and don’t check in nor allow anyone else to check in code that uses subtle synchronization.**

Wrap-up

Some practical advice

- Use synchronization, don't try and go lock-free; Write DRF code, don't bother with so much as a flag
- Co-architect the way you will parallelize your code with the way that code is synchronized; it's not just that it's parallel, it has to work.
- Clean up after yourself. Nobody likes an application that leaks, and that sloppiness is a sign of poor design, leading to more bugs.
    - Be very careful about having memory allocated in one thread, de-allocated in another. Can you re-architect the code?
- #define volatile _do_not_compile_this_volatile
  You want that compiler error so you don't checkin code with volatile in it. Have explicit read() and write() operations that use the address expose primitive.
- Don't prematurely optimize your locking/threading, etc. Keep it simple (sequential) until you have to, and then take only baby steps into parallelization.
    - Think of the children.
- Violate the rule above if you have to, and there will be times when you have to. Some code is fundamentally parallel.

Stretch!

User mode threads

- “Threads” as we have been discussing them are:
  - pre-emptive
  - their state is more or less equivalent to a processor state (register set, IP, stack, address space, etc)
  - co-managed by a user-space runtime system and the operating system
- *User mode threads* are (generally):
  - co-operatively multitasked (yield())
  - Their state can be squeezed quite small
  - Can be entirely managed by a user mode runtime

Threading packages

- Windows has built in support for user mode threads, namely in the form of “user mode scheduling”, and “fibers”.
    - UMS threads have a complete context, and if blocked in the kernel another UMS thread can run
    - fibers are more like user mode threads on Linux
- With Linux systems you must first choose a package
    - “unfortunately”, writing a user mode threading package is a common undergraduate OS assignment. So you have to sift through a lot of crap on the internet on the topic.

This is why we professors use this as a homework assignment

```
// void switch_stacks(u_int64_t *old_context, u_int64_t *new_context);
switch_stacks:
        push    %rbx            # save the registers on the stack
        push    %rbp
        push    %r12
        push    %r13
        push    %r14
        push    %r15
        push    %rdx
        push    %rcx
        push    %rdi
        push    %rsi

        mov     %rsp, (%rdi)    # save old stack

        mov     (%rsi), %rsp    # load new stack

        pop     %rsi            # pop our saved values off the stack
        pop     %rdi
        pop     %rcx
        pop     %rdx
        pop     %r15
        pop     %r14
        pop     %r13
        pop     %r12
        pop     %rbp
        pop     %rbx
        ret
```

Or call get/set context :)

A complete user mode thread package offers:

- create, join,
- **void yield();**
- *possibly* yield_to(thread_t);
- lock, unlock
- cond_wait, cond_signal, cond_broadcast
- *possibly,* although it is highly desirable if they do, some form of I/O

Why should you care?

- pthread_create /join are *NOT* fast.
- Even context switching pthreads is not fast ~ 750ns
- Suppose you want to specify your work as *millions* of tiny work units (tasks if you will) and then dynamically bind them to threads (stacks) for execution.
    - Useful technique because it can allow code to be performant in the face of a variety of machine architectures.
- Or suppose you want to overlap I/O. Your life will be easier with pthreads but UMS on Windows or a rich library on Linux can help.

Stretch!

OpenMP

- An extension to C/C++ that gives you a parallel for loop (and a lot more).

- Cross-platform and widely supported

- Bit of a grotesque syntax, but it can get the job done

- http://bisqwit.iki.fi/story/howto/openmp/

- http://openmp.org/mp-documents/omp-hands-on-SC08.pdf

Why do you want a parallel for loop?

- Recall the simple code we had before:

```
for (i = 0; i < n; i++)
    a[i] = rand() % 10;
for (i = 0; i < n; i++)
    sum += a[i];
```

- …and remember how that became 120 lines of code

OpenMP version of sum fits on a slide

-
  ```
  #include <omp.h>
  #include <stdio.h>

  int main(int argc, char *argv[]) {
      int     *array;
      int     i, sum = 0;
      int     array_size = atoi(argv[1]);

      array = malloc(sizeof(*array) * array_size);

      #pragma omp parallel for private(i)
      for (i = 0; i < array_size; i++) {
          int r;
      #pragma omg critical
          r = rand();
          array[i] = r % 10;
      }

      #pragma omp parallel for reduction(+:sum) private(i)
      for (i = 0; i < array_size; i++)
          sum += array[i];

      printf("Sum=%d\n", sum);
      return 0;
  }
  ```

OpenMP version of sum fits on a slide

-
  ```
  #include <omp.h>
  #include <stdio.h>

  int main(int argc, char *argv[]) {
      int     *array;
      int     i, sum = 0;
      int     array_size = atoi(argv[1]);

      array = malloc(sizeof(*array) * array_size);

      #pragma omp parallel for private(i)
      for (i = 0; i < array_size; i++) {
          int r;
      #pragma omg critical
          r = rand();
          array[i] = r % 10;
      }

      #pragma omp parallel for reduction(+:sum) private(i)
      for (i = 0; i < array_size; i++)
          sum += array[i];

      printf("Sum=%d\n", sum);
      return 0;
  }
  ```

….but runs 75x slower than sequential :)

Example

OpenMP has grown substantially over time

- parallel for
  - Lots of ways to control the schedule of iterations
- reduction
- SIMD
  - For use with MMX, etc
- generalized tasks
- partitioned address spaces
  - For accelerator offload