

Synchronization

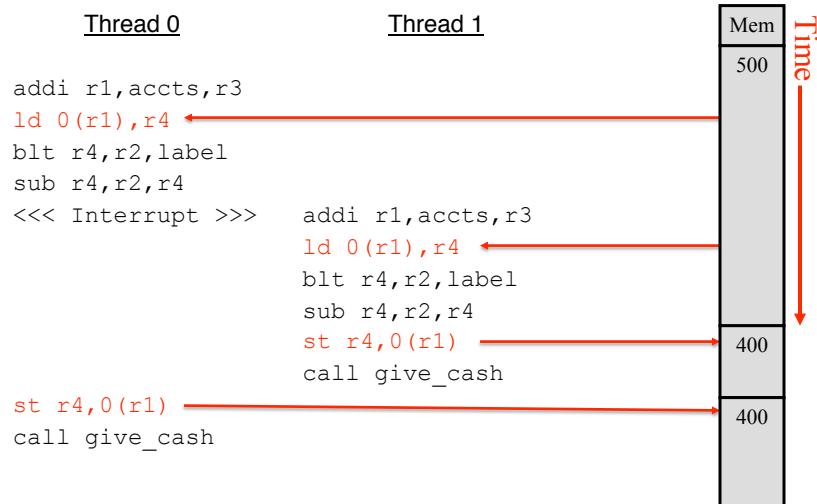
Coherency protocols guarantee that a reading processor (thread) sees the most current update to shared data.

Often we want to follow program behaviors that are on a higher plane than an individual access

Coherency protocols **do not** regulate access to shared data:

- Do not ensure that only one thread operates on shared data or a shared hardware or software resource at a time
- **Critical sections** order thread access to shared data
- Do not force threads to start executing particular sections of code together
- **Barriers** force threads to start executing particular sections of code together

Critical Sections: Motivating Example



Critical Sections

A **critical section**

- a sequence of code that only one thread can execute at a time
- provides **mutual exclusion**
 - a thread has exclusive access to the code & the data that it accesses
 - guarantees that only one thread can update the data at a time
- to execute a critical section, a thread
 - acquires a lock that guards it
 - executes its code
 - releases the lock

The effect is to synchronize or order the access of threads with respect to their accessing shared data

Spring 2009

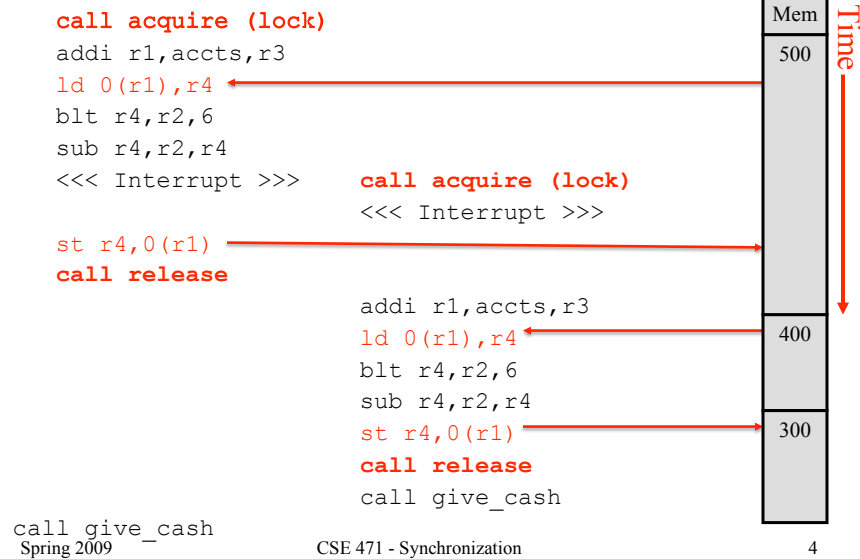
CSE 471 - Synchronization

3

Critical Sections: Correct Example

Thread 0

Thread 1



CSE 471 - Synchronization

4

Barriers

Barrier synchronization

- a **barrier**: point in a program which all threads must reach before any thread can cross
 - threads reach the barrier & then wait until all other threads arrive
 - all threads are released at once & begin executing code beyond the barrier
- example implementation of a barrier:
 - set a lock-protected counter to the number of processors
 - each thread (assuming 1/processor) decrements it
 - when the counter value becomes 0, all threads have crossed the barrier
- code that implements a barrier is a critical section
- useful for:
 - programs that execute in (semantic) phases
 - synchronizing after a parallel loop

Spring 2009

CSE 471 - Synchronization

5

Locking

Locking facilitates access to a critical section & shared data.

Locking protocol:

- **synchronization variable or lock**
 - 0: lock is available
 - 1: lock is unavailable because another thread holds it
- a thread obtains the lock before it can enter a critical section or access shared data
 - sets the lock to 1
- thread releases the lock before it leaves the critical section or after its last access to shared data
 - clears the lock

Spring 2009

CSE 471 - Synchronization

6

Acquiring a Lock

Atomic exchange instruction: swap a value in a register & a value in memory in one operation

- set the register to 1
- swap the register value & the lock value in memory
- new register value determines whether got the lock

```
AcquireLock:
    li    R3, #1          /* create lock value
    swap  R3, 0(R4)       /* exchange register & lock
    bnez  R3, AcquireLock /* have to try again */
```

- also known as **atomic read-modify-write** a location in memory

Other examples

- test & set: tests the value in a memory location & sets it to 1
- fetch & increment/decrement: returns the value of a memory location + 1

Releasing a Lock

Store a 0 in the lock

Load-linked & Store Conditional

Performance problem with atomic read-modify-write:

- 2 memory operations in one
- must hold the bus until both operations complete

Pair of instructions *appears* atomic

- avoids need for uninterruptible memory read & write
- **load-locked & store-conditional**
 - load-locked returns the original (lock) value in memory
 - if the contents of lock memory has not changed when the store-conditional is executed, the processor still has the lock
 - store-conditional returns a 1 if successful

```

GetLk:    li    R3, #1          /* create lock value
ll     R2, 0(R1)          /* read lock variable
...
sc     R3, 0(R1)          /* try to lock it
beqz    R3, GetLk          /* cleared if sc failed
... (critical section)

```

Spring 2009

CSE 471 - Synchronization

9

Load-linked & Store Conditional

Implemented with special processor registers: **lock-flag register & lock-address register**

- load-locked sets lock-address register to lock's memory address & lock-flag register to 1
- store-conditional returns lock-flag register value
- if still 1, then processor has the lock
- lock-flag register is cleared if the lock is written by another processor
- lock-flag register cleared if context switch or interrupt

Spring 2009

CSE 471 - Synchronization

10

Synchronization APIs

User-level software synchronization library routines constructed with atomic hardware primitives

- **spin locks**
 - **busywaiting** until obtain the lock
 - contention with atomic exchange causes invalidations (for the write) & coherency misses (for the rereads)
 - avoid if separate reading the lock & testing it
 - spinning done in the cache rather than over the bus

```

getLk:                li    R2, #1
spinLoop:             ll    R1, lockVariable
                     blbs  R1, spinLoop
                     sc    R2, lockVariable
                     beqz  R2, getLk
                     .... (critical section)
                     st    R0, lockVariable
  
```

Spring 2009

CSE 471 - Synchronization

11

Synchronization APIs

User-level software synchronization library routines constructed with atomic hardware primitives

- **blocking locks**
 - block the thread immediately
 - block the thread after a certain number of spins

Spring 2009

CSE 471 - Synchronization

12

Synchronization Strategy

An example overall synchronization/coherence strategy:

- design cache coherency protocol for little interprocessor contention for locks (the common case)
- add techniques to avoid performance loss if there is contention for a lock & still provide low latency if no contention

Synchronization Strategy

Have a race condition for acquiring a lock when it is unlocked

- $O(p^2)$ bus transactions for p contending processors with write-invalidate

Two techniques to avoid $O(p^2)$

- **exponential back-off** - software solution
 - each processor retries at a different time
 - successive retries done an exponentially increasing time later
- **queuing locks** - hardware solution
 - each processor spins on a different location (a queue)
 - when a lock is released, only the next processor see its lock go "unlocked"
 - other processors continue to spin
 - lock is effectively passed from one processor to the next
 - also addresses fairness (locks acquired in FIFO order)

Trickiness

Writing programs that are both correct and parallel

- Choosing the right kind of lock
- Choosing the right locking granularity
 - Coarse-grain are simple to get correct, but limit parallelism
 - Fine-grain the opposite
- Acquiring & releasing nested locks in the correct order, or deadlock
- Avoiding locks when they aren't really needed

Transactional Memory

The idea:

- No locks, just shared data
- Execute critical sections speculatively
- Abort on conflicts

```
begin_transaction();  
if (accts[id_from].bal >= amt) {  
    accts[id_from].bal -= amt;  
    accts[id_to].bal += amt; }  
end_transaction();
```


Transactional Memory

```
begin_transaction();
if (accts[id_from].bal >= amt) {
    accts[id_from].bal -= amt;
    accts[id_to].bal += amt; }
end_transaction();
```

begin_transaction :

- Checkpoint the registers
- Track all read addresses
- Buffer all the writes so they're invisible to other processors

end_transaction :

- Any writes to the tracked read addresses"
 - no: commit the writes
 - yes: abort the transaction by restoring the checkpoint & re-executing

Transactional Memory

- + Has the programming simplicity of coarse-grain locks
- + Higher concurrency (parallelism) of fine-grain locks
 - critical sections usually execute in parallel
 - only serialized if data is actually shared
- + No lock acquisition overhead

Transactional Memory

Issues:

- What if reads/writes don't fit in the cache?
- What if the transaction gets swapped out in the middle?
- What if the transaction does a (not-abortable) I/O or syscall?
- How transactionify existing lock-based programs?
- Should transactions be implemented in hardware, software or both?