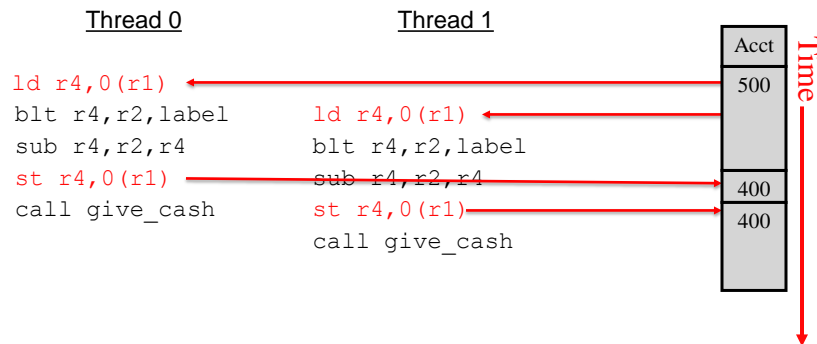# Synchronization

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 does a *series of accesses* to 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

Spring 2012   CSE 471 - Synchronization   1

Critical Sections: Motivating Example

Thread 0	Thread 1	Acct	
ld r4,0(r1)		500	Time
blt r4,r2,label	ld r4,0(r1)		
sub r4,r2,r4	blt r4,r2,label		
st r4,0(r1)	sub r4,r2,r4	400	
call give_cash	st r4,0(r1)	400	
	call give_cash		

Spring 2012   CSE 471 - Synchronization   2

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 shared data at a time
- to execute a critical section, a thread
  - acquires a lock that guards it
  - executes its code
  - releases the lock

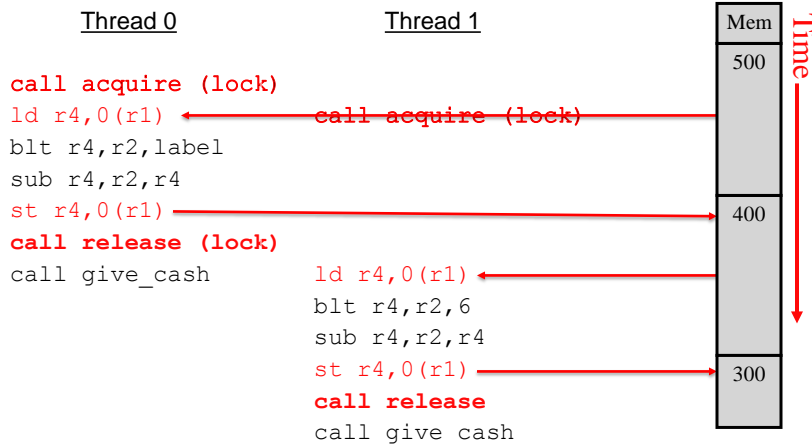The effect is to synchronize or order the access of threads with respect to their accessing shared data

Spring 2012   CSE 471 - Synchronization   3

Critical Sections: Correct Example

Thread 0   Thread 1   Mem

call acquire (lock)
ld r4,0(r1) ←   call acquire (lock)   ← 500
blt r4,r2,label
sub r4,r2,r4
st r4,0(r1) →   → 400
call release (lock)
call give_cash   ld r4,0(r1) ←
                      blt r4,r2,6
                      sub r4,r2,r4
                      st r4,0(r1) →   300
                      call release
                      call give_cash

Time

Spring 2012   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 threads
    - each thread decrements the counter
    - when the counter value becomes 0, all threads have crossed the barrier
- code that implements the counter must be a critical section
- useful for:
    - programs that execute in (semantic) phases
    - synchronizing after a parallel loop

Spring 2012   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 2012   CSE 471 - Synchronization   6

Acquiring a Lock

Atomic exchange instruction: swap a value in memory & a value in a register as 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 */
```

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
- in general, a lock implementation is known as **atomic read-modify-write** to a location in memory

Releasing a Lock

Store a 0 in the lock

Load-locked & 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 pair
- **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 2012   CSE 471 - Synchronization   9

Load-locked & 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
- if 0, then processor no longer has the lock & has to try again
- why would the lock-flag register be cleared?
  - if the lock is written by another processor
  - if a context switch or interrupt

Spring 2012   CSE 471 - Synchronization   10

5

Synchronization APIs

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

- efficient **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 & testing the lock & updating 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
```

Synchronization APIs

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

Inter-thread 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

Spring 2012   CSE 471 - Synchronization   13

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 (could be software)
  - each processor spins on a different location (in a queue)
  - when a lock is released, only the next processor in the queue see its lock go "unlocked"
  - other processors continue to spin/block
  - lock is effectively passed from one processor to the next
  - also addresses fairness (locks acquired in FIFO order)

Spring 2012   CSE 471 - Synchronization   14

Trickiness

Writing programs that are both correct and parallel

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

end_transaction :
- Commit the writes to memory, clear bits

Implemented with cache block state,: read & write bits
- Set bits on read or write
- Clear bits on commit
- If any block with read or write bit set is invalidated, abort the transaction by restoring the checkpoint & re-executing.

Spring 2012   CSE 471 - Synchronization   17

Transactional Memory

+ Has the programming simplicity of coarse-grain locks
  - execute transactions speculatively
+ Higher concurrency (parallelism) of fine-grain locks
  - abort if a conflict
    only serialized if data is actually write-shared
+ No lock acquisition overhead

Spring 2012   CSE 471 - Synchronization   18

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 do we automatically "transactionify" existing lock-based programs?
- Should transactions be implemented in hardware, software or both?

Important Issues

Red & Green

- role of coherency protocol vs. role of thread synchronization
- critical section
- mutual exclusion
- barrier synchronization
- how locks work
- efficient atomic operations
- another example of snooping
- spinning vs. blocking
- another illustration of trading latency for throughput
- efficient busywaiting
- another use of speculation
- precise interrupts & transactional memory (another roll-back situation)