

Multiprocessor Synchronization

CSE 240B

Dean Tullsen

Synchronization

- Why Synchronize?

CSE 240B

Dean Tullsen

Synchronization

- Why Synchronize?
- Why do we need hardware support for synchronization?

CSE 240B

Dean Tullsen

Synchronization

- Why Synchronize?
- Why do we need hardware support for synchronization?
- What do we need that hardware support to give us to make (basic) synchronization work?

CSE 240B

Dean Tullsen

Synchronization

- Why Synchronize?
- Why do we need hardware support for synchronization?
- What do we need that hardware support to give us to make (basic) synchronization work?
- What do we need to make a **lock** work?

Locking

P1	P2
lock(L)	lock(L)
load sharedvar	load sharedvar
modify sharedvar	modify sharedvar
store sharedvar	store sharedvar
release(L)	release(L)

Uninterruptable Instruction to Fetch and Update Memory

0 => synchronization variable is free

1 => synchronization variable is locked and unavailable

- **Atomic exchange**: interchange a value in a register for a value in memory
 - Set register to 1 & swap
 - New value in register determines success in getting lock
 - 0 if you succeeded in setting the lock (you were first)
 - 1 if other processor had already claimed access
 - Write 0 to release lock
 - Key is that exchange operation is indivisible
 - Can be used to do more powerful things than implement locks.

Uninterruptable Instruction to Fetch and Update Memory

- **Test-and-set**: reads a value and sets it atomically
 - Special case of atomic exchange
 - Most common sync primitive
 - 0 means lock free, 1 means locked
 - Test-and-set reads the lock variable, and sets it to one. If the value read was 0, you have acquired the lock. If it was 1, you did not.
 - Write 0 to release lock
 - Pretty much just used to enable locks.

Uninterruptable Instruction to Fetch and Update Memory

- **Fetch-and-increment**: it returns the value of a memory location and atomically increments it
 - 0 => synchronization variable is free, >0 implies locked
 - Write 0 to release lock
 - Can do more powerful things than implement locks

CSE 240B

Dean Tullsen

Example

```
lock:  lw    R1, lockaddress
      bnez  R1, lock
      addi R1, R0, 1
      sw   R1, lockaddress
      lw   R2, varaddress
      addi R2, R2, 1
      sw   R2, varaddress
      add  R1, R0, R0
release: sw   R1, lockaddress
```

- Why doesn't this work?

CSE 240B

Dean Tullsen

Example

lock:	lw	R1, lockaddress							
	bnez	R1, lock			lock:	T&S	R1, lockaddress		
	addi	R1, R0, 1				bnez	R1, lock		
	sw	R1, lockaddress				lw	R2, varaddress		
	lw	R2, varaddress				addi	R2, R2, 1		
	addi	R2, R2, 1				sw	R2, varaddress		
	sw	R2, varaddress				add	R1, R0, R0		
	add	R1, R0, R0				sw	R1, lockaddress		
release:	sw	R1, lockaddress							

→

- This works because *test-and-set* is atomic
- Notice this could be done with one instruction if we have *fetch-and-increment*.

CSE 240B

Dean Tullsen

Uninterruptable *Instructions* to Fetch and Update Memory

- Hard to have read & write in 1 instruction: use 2 instead
- **Load linked** (or load locked) + **store conditional**
 - Load linked returns the initial value
 - Store conditional only completes the store if no other store to same memory location since preceding load linked. The SC returns 1 if it succeeds and 0 otherwise.

CSE 240B

Dean Tullsen

Uninterruptable *Instructions* to Fetch and Update Memory

- Example doing atomic swap with LL & SC:

```
try: mov    R3,R4      ; mov exchange value
      ll     R2,0(R1)  ; load linked
      sc     R3,0(R1)  ; store
      beqz   R3,try    ; branch store fails
      mov    R4,R2     ; put load value in R4
```

- Example doing fetch & increment with LL & SC:

```
try: ll     R2,0(R1)  ; load linked
      addi   R2,R2,#1 ; increment (OK if reg-reg)
      sc     R2,0(R1)  ; store
      beqz   R2,try    ; branch store fails
```

CSE 240B

Dean Tullsen

LL-SC

- This is an example of something called *non-blocking (lock-free) synchronization*. Why? What's the big advantage?

CSE 240B

Dean Tullsen

Compare and Swap (CAS)

- CAS(address, oldvalue, newvalue)
 - Atomic

```
if (*address == oldvalue) {
    *address = newvalue
    return 1
}
else
    return 0
```

- Power PC supports this

CSE 240B

Dean Tullsen

Fetch and Add with CAS?

CSE 240B

Dean Tullsen

Big Weakness of Compare and Swap?

- (Not a problem with *fetch and add* example)

User Level Synchronization—Operation Using These Primitives

- Spin locks: processor continuously tries to acquire, spinning around a loop trying to get the lock

```

lockit:    li      R2,#1
           exch   R2,0(R1)      ;atomic exchange
           bnez  R2,lockit      ;already locked?
    
```

- What about MP with cache coherency?
 - Want to spin on cache copy to avoid full memory latency
 - Likely to get cache hits for such variables
- Problem: exchange includes a write, which invalidates all other copies; this generates considerable bus traffic

User Level Synchronization—Operation Using These Primitives

- Solution: start by simply repeatedly reading the variable; when it changes, then try exchange (“test and test&set”):

```

try:      li      R2,#1
lockit:   lw      R3,0(R1)      ;load var
           bnez  R3,lockit      ;not free=>spin
           exch  R2,0(R1)      ;atomic exchange
           bnez  R2,try        ;already locked?
    
```

Steps for Invalidate Protocol

Step	P0	\$	P1	\$	P2	\$	Bus/Direct activity
1.	Has lock	Sh	spins	Sh	spins	Sh	None
2.	Lock<- 0	Ex		Inv		Inv	P0 Invalidates lock
3.		Sh	miss	Sh	miss	Sh	WB P0; P2 gets bus
4.		Sh	waits	Sh	lock = 0	Sh	P2 cache filled
5.		Sh	lock=0	Sh	exch	Sh	P2 cache miss(WI)
6.		Inv	exch	Inv	r=0;l=1	Ex	P2 cache filled; Inv
7.		Inv	r=1;l=1	Ex	locked	Inv	WB P2; P1 cache
8.		Inv	spins	Ex		Inv	None

For Large Scale MPs, Synchronization Can Be a Bottleneck

- 20 processors spin on lock held by 1 proc, 50 cycles for bus
 - 1525 bus operations, over 30,000 cycles for 20 processors to pass through the lock
 - Problem is contention for lock and serialization of lock access: once lock is free, all compete to see who gets it (each causing an invalidate storm)
- Alternative: exponential backoff. Why does this help?
- Another alternative: create a list of waiting processors, go through list: called a “queuing lock”

CSE 240B

Dean Tullsen

Barrier Synchronization

- A very common synchronization primitive
- Wait until all threads have reached a point in the program before any are allowed to proceed further.

```
computation;  
barrier()  
communication;  
barrier()  
repeat:
```

CSE 240B

Dean Tullsen

Barriers

Hardware Support?

CSE 240B

Dean Tullsen

Barriers

Hardware Support?
Software?

CSE 240B

Dean Tullsen

A Methodology for Implementing Highly Concurrent Data Structures

Maurice Herlihy

CSE 240B

Dean Tullsen

Wait-Free vs Lock-Free (non-blocking)

Non-blocking?

Wait-Free?

Do I Care??

CSE 240B

Dean Tullsen

Non-blocking: Single-word Objects

Sequential:

```
FETCH&ADD(x: integer, v: integer)  
returns(integer, integer)  
return (x+v, x)  
end FETCH&ADD
```

CSE 240B

Dean Tullsen

Non-blocking: Single-word Objects

Parallel:

```
fetch&add(obj: object, v: integer) returns(integer)  
success: boolean := false  
loop exit when success  
old: integer := obj  
new: integer, r: value := FETCH&ADD(old, v)  
success := compare&swap(x, old, new);  
end loop  
return r  
end fetch&add
```

non-blocking

CSE 240B

Dean Tullsen

Small Objects

What makes objects larger than one word hard?


CSE 240B

Dean Tullsen

Non-blocking: Small Objects

Sequential:

```
stack = record[size: integer, data: array[item]]
POP(s: block*) returns(block*, item)
  if s.size = 0
    then return (s, "empty") result for empty stack
  end if
  r: item := s.data[s.size] compute non-empty POP result
  s' := alloc() allocate new stack
  s'.size := s.size - 1
  for i in 1..s'.size do
    s'.data[i] := s.data[i] copy old stack suffix
  end for into new stack
  free(s) free old stack
  return (s', r) return (new state, result) tuple
end POP
```



CSE 240B

Small Objects

What is the naïve parallel version?

CSE 240B

Dean Tullsen

Small Objects

What is the naïve parallel version?

What is the problem?

CSE 240B

Dean Tullsen

Non-blocking: Small Objects

Parallel:

```
pop(root: block*) returns(value)
  success: boolean := false
  loop exit when success
1:   old: block* := freeze(root)      freeze obj current version
2:   new: block*, res: value := POP(old) compute new version
   success := true
3:   if old ≠ new                      if new object state differs
     then success :=
       compare&swap(obj, old, new)    update object state
     end if
abort: unfreeze blocks in frozen_list  unfreeze block | froze
4:   if success
     then unfreeze blocks in commit_set  unfreeze block | freed
     else unfreeze blocks in abort_set    unfreeze block | allocated
     end if
5:   reset frozen_list, commit_set and abort_set  reset private vars
  end loop
  return result
end pop
```

CSE 240B

24

Large Objects

Example: skew heap (tree), but assume any linked data structure

What would happen if we apply the small object protocol?

Ignoring the complexity of the code shown, what in principle can you do with this kind of structure to avoid full copying, but never let the tree be in an inconsistent state?

CSE 240B

Dean Tullsen

Wait-Free

What additional constraints are placed on the computing nodes?

What are some of the strategies you can employ?

CSE 240B

Dean Tullsen