

Synchronization with Locks

Lei Qiao

ECE 401
Fall 2006

Locks – A user-level synchronization operation

- An example:

Suppose a function in an ATM machine to handle withdrawals from a bank account:

```
withdraw (account, amount) {  
    balance = get_balance(account);  
    balance = balance – amount;  
    put_balance(account, balance);  
    return balance;  
}
```

Locks – A user-level synchronization operation (cont)

- For shared-memory MP machines, the problem is similar and the solution is also similar
- Implementation with instructions
 - 0 means lock is free
 - 1 means lock is unavailable

Lockit:	LD	R2, 0(R1)
	BNEZ	R2, lockit
	ADDI	R2, R0, #1
	SD	R2, 0(R1)

Key: Atomic Operation

- Locks has to be atomic. That is, the processor should atomically test and set the clock
- During the operation, a processor can simultaneously read and write a memory location
- This prevents other processors from writing or reading memory until atomic operation is complete.

Basic Hardware Primitives

- We need basic hardware primitives capable of atomically reading and modifying the memory location to implement locks
- Atomic exchange
Interchange a value in a register for a value in memory
- Test-and-Set
Test a value and set it if the value pass the test
- Fetch-and-Increment
It returns the value in memory and atomically increments it

How to implement this primitives with instructions

- Load Linked & Store Conditional
- LL read the value in the memory, and put the memory address into link register.
- SC checks if the address matches the one in the link register; if so, SC succeed (return 1)
- If an interrupt occurs or if the cache block containing the address is invalidated, the link register is set to 0 and cause SC fails (return 0)

Implementing atomic exchange with LL & SC

```
try: OR    R3,R4,R0    ; move exchange value into R3
      LL    R2,0(R1)   ; load the value at 0(R1) into R2
      SC    R3,0(R1)   ; store conditional and set R3
      BEQZ  R3,try     ; branch if the value set in R3 is
                        ; changed to 0 (store fails)
      MOV   R4,R2     ; put load value in R4
```



```
EXCH  R4, 0(R1)    ; atomic exchange
```

Implementing locks using atomic exchange

- Basic idea
- Each processor try to set the lock by exchanging 1 in the register with lock value in the memory
- After exchange, the register of each processor will return 1 or 0 to determines if the processor succeeds in setting the lock: 0 means succeed; 1 means fail

Spin locks & implementation

- Spin locks

Locks that a processor continuously tries to acquire, spinning around a loop trying to get the lock

- Implementation

```
lockit:  ADDI    R2,R0,#1    ; load the immediate #1
        EXCH   R2,0(R1)  ; exchange #1 with the
                           memory location of lock
        BNEZ   R2,lockit ; branch if the lock is unavailable
```

Implementing locks using coherence

- In machines that support cache coherence, we hope to cache
- The spin-lock operation is performed on a local cached copy rather than to require a global memory access on each attempt to acquire the lock
- Since there is often locality in lock access, caching reduces time to acquire the lock

Implementing locks with coherence (cont)

- **Problem:**

Exchange include a write which requires an exclusive invalidate all other copies; this will generate considerable bus traffic

- **Solution:**

Start by simply repeatedly reading the lock variable; when it changes, then try exchange

Implementing locks with coherence (cont)

- In this loop, we use read instead of exchange and swap only when the lock becomes free

```
lockit: LD      R2,0(R1) ; read the lock
        BNEZ   R2,lockit ; branch & keep reading the lock
                          ; if lock is not available
        ADDI  R2,R0,#1  ; load locked value
        EXCH  R2,0(R1) ; race to exchange & obtain a 0
        BNEZ  R2,lockit ; branch if other processor get
                          ; it first
```

<u>Step</u>	<u>P0</u>	<u>P1</u>	<u>P2</u>	<u>Coherence State of lock</u>	<u>Bus activity</u>
1	Has Lock	Spins	Spins	Shared	None
2	Set Lock = 0	Invalidate received	Invalidate received	Exclusive(P0)	Write Invalidate from P0
3		Cache Miss	Cache Miss	Shared	WB from P0.
4		Waits while bus busy	Lock = 0	Shared	Cache Miss (P2) satisfied
5		Lock = 0	Swap → Cache Miss	Shared	Cache Miss (P1) satisfied
6		Swap → Cache miss	Completes swap returns 0 and L=1	Exclusive(P2)	Write Invalidate from P2
7		Completes swap returns 1 and set L=1	Enter critical section	Exclusive(P1)	Write back
8		Spins			None

Reference

- Computer Architecture: A Quantitative Approach, Third Edition, John L. Hennessy, David A. Patterson
- Slide of Prof. Huang's lecture from U of Rochester
- Slide of Prof. Patterson's lecture from U.C. Berkeley