Lecture 5: Synchronization

• Topics: synchronization primitives and optimizations
Synchronization

• The simplest hardware primitive that greatly facilitates synchronization implementations (locks, barriers, etc.) is an atomic read-modify-write

• Atomic exchange: swap contents of register and memory

• Special case of atomic exchange: test & set: transfer memory location into register and write 1 into memory

• lock: t&s register, location
  bnz register, lock
  CS
  st location, #0
Improving Lock Algorithms

• The basic lock implementation is inefficient because the waiting process is constantly attempting writes → heavy invalidate traffic

• Test & Set with exponential back-off: if you fail again, double your wait time and try again

• Test & Test & Set: read the value, if it has not changed, don’t bother doing the test&set – heavy bus traffic only when the lock is released

• Different implementations trade-off one of these lock properties: latency, traffic, scalability, storage, fairness
Load-Linked and Store Conditional

• LL-SC is an implementation of atomic read-modify-write with very high flexibility

• LL: read a value and update a table indicating you have read this address, then perform any amount of computation

• SC: attempt to store a result into the same memory location, the store will succeed only if the table indicates that no other process attempted a store since the local LL

• SC implementations may not generate bus traffic if the SC fails – hence, more efficient than test&test&set
Load-Linked and Store Conditional

lockit:    LL R2, 0(R1) ; load linked, generates no coherence traffic
BNEZ R2, lockit ; not available, keep spinning
DADDUI R2, R0, #1 ; put value 1 in R2
SC R2, 0(R1) ; store-conditional succeeds if no one
            ; updated the lock since the last LL
BEQZ R2, lockit ; confirm that SC succeeded, else keep trying
Further Reducing Bandwidth Needs

• Even with LL-SC, heavy traffic is generated on a lock release and there are no fairness guarantees

• Ticket lock: every arriving process atomically picks up a ticket and increments the ticket counter (with an LL-SC), the process then keeps checking the now-serving variable to see if its turn has arrived, after finishing its turn it increments the now-serving variable – is this really better than the LL-SC implementation?

• Array-Based lock: instead of using a “now-serving” variable, use a “now-serving” array and each process waits on a different variable – fair, low latency, low bandwidth, high scalability, but higher storage
Barriers

• Barriers require each process to execute a lock and unlock to increment the counter and then spin on a shared variable

• If multiple barriers use the same variable, deadlock can arise because some process may not have left the earlier barrier – sense-reversing barriers can solve this problem

• A tree can be employed to reduce contention for the lock and shared variable

• When one process issues a read request, other processes can snoop and update their invalid entries
Barrier Implementation

LOCK(bar.lock);
if (bar.counter == 0)
    bar.flag = 0;
mycount = bar.counter++;
UNLOCK(bar.lock);
if (mycount == p) {
    bar.counter = 0;
    bar.flag = 1;
}
else
    while (bar.flag == 0) {};

local_sense = !(local_sense);
LOCK(bar.lock);
mycount = bar.counter++;
UNLOCK(bar.lock);
if (mycount == p) {
    bar.counter = 0;
    bar.flag = local_sense;
}
else {
    while (bar.flag != local_sense) { }
}
Title

• Bullet