Chapter 9
Hardware Support for Synchronization

Module 9.1 – Simple Lock Implementation

Synchronization

- "A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast."

- Types of Synchronization
  - *Lock:* for ensuring mutual exclusion
  - Event synchronization
    - *point-to-point*
  - Group synchronization
    - *global (barriers)*

Typical Modern Implementation

HW:

- Level 1: LL/SC (not always needed)
- Level 2: Atomic exchange, fetch and increment, test and set, compare and swap

SW (as libraries)

- Level 3: Lock
- Level 4: Barriers, semaphores, monitors

Level i can construct synchronization support on level i+1

Lock Implementation Performance Criteria

- Uncontended Latency
  - Time to acquire a lock when there is no contention
- Traffic
  - Lock acquisition attempt when lock is acquired
  - Lock acquisition attempt when lock is free
  - Lock release
- Storage
  - As a function of # of threads/processors
- Fairness
  - Degree in which a thread can acquire a lock with respect to others

First Attempt at Simple Software Lock

```
void lock (int *lockvar) {
  while (*lockvar == 1) {} ;  // wait until released
  *lockvar = 1;               // acquire lock
}

void unlock (int *lockvar) {
  *lockvar = 0;
}
```

In machine language, it looks like this:

```
lock: ld R1, &lockvar     // R1 = MEM[&lockvar]
      bnz R1, lock        // jump to lock if R1 != 0
      sti &lockvar, #1    // MEM[&lockvar] = 1
      ret                 // return to caller
unlock: sti  &lockvar, #0 // MEM[&lockvar] = 0
        ret               // return to caller
```

Needed: ld to sti sequence executed atomically. Atomic implies:

- The entire sequence appears to execute in its entirity
- Multiple sequences are serialized

Example of Atomic Instructions

- `test-and-set Rx, M`
  - read the value stored in memory location M, test the value against a constant (e.g. 0), and if they match, write the value in register Rx to the memory location M.
- `fetch-and-op M`
  - read the value stored in memory location M, perform `op` to it (e.g., increment, decrement, addition, subtraction), then store the new value to the memory location M.
- `exchange Rx, M`
  - atomically exchange (or swap) the value in memory location M with the value in register Rx.
- `compare-and-swap Rx, Ry, M`
  - compare the value in memory location M with the value in register Rx. If they match, write the value in register Ry to M, and copy the value in Rx to Ry.

How is Atomicity Guaranteed?

- Use coherence protocol to ensure one atomic instruction is executed at a time

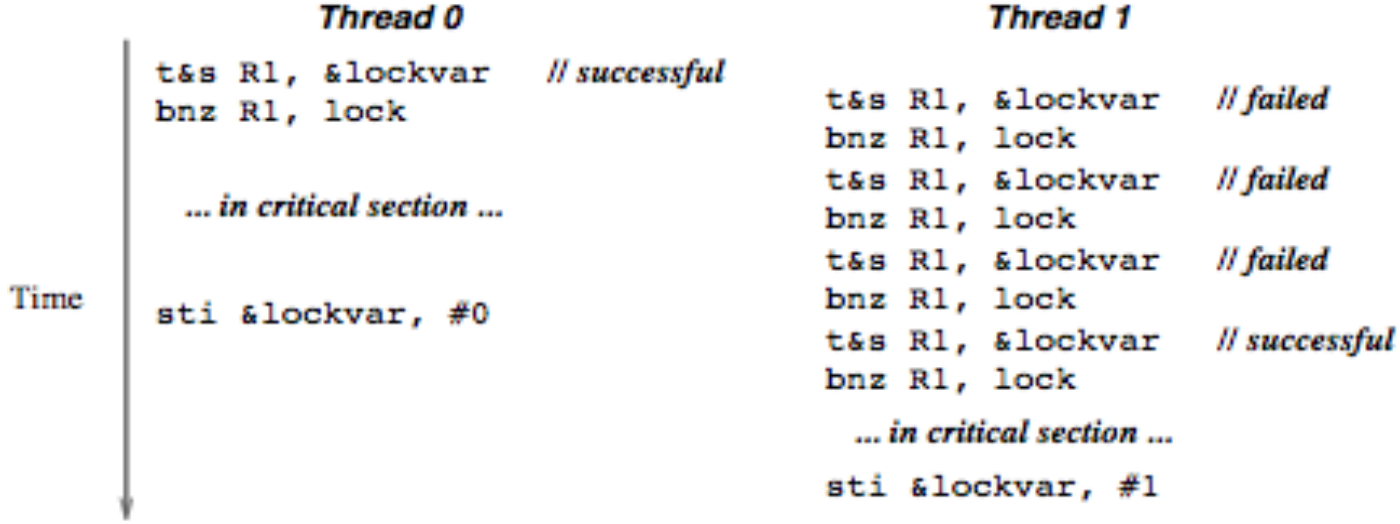
- How to ensure one atomic instruction is executed at a time:
- 1. Reserve the bus until done
  - Other atomic instructions cannot get to the bus
- 2. Reserve the cache block involved until done
  - Obtain exclusive permission (e.g. "M" in MESI)
  - Reject or delay any invalidation or intervention requests until done
- 3. Provide "illusion" of atomicity instead
  - Using LL/SC (to be discussed later)

Simple Test&Set Lock

- lockvar =
  - 1 if the lock is already acquired
  - 0 if the lock is free

```
lock: t&s R1, &lockvar  // R1 = MEM[&lockvar];
                        // if (R1==0) MEM[&lockvar]=1
      bnz R1, lock;     // jump to lock if R1 != 0
      ret               // return to caller
unlock: st &lockvar, #0 // MEM[&lockvar] = 0
        ret             // return to caller
```

Illustration of test-and-set lock Execution



- t&s of thread 0 is successful
- t&s of thread 1 (repeatedly attempted) fails until lock is released by thread 0

Interaction with Cache Coherence

Request	P1	P2	P3	BusRequest
Initially	-	-	-	-
t&s1	M	-	-	BusRdX
t&s2	I	M	-	BusRdX
t&s3	I	I	M	BusRdX
t&s2	I	M	I	BusRdX
unlock1	M	I	I	BusRdX
t&s2	I	M	I	BusRdX
t&s3	I	I	M	BusRdX
t&s3	I	I	M	-
unlock2	I	M	I	BusRdX
t&s3	I	I	M	BusRdX
unlock3	I	I	M	-

Drawbacks

- Very high traffic
  - many coherence transactions even when a lock is not free
- They compete with traffic of the thread that has the lock
  - critical section may be lengthened
- Strategy for reducing traffic
  - Back-off: pause for a while after a failed acquisition and retry later
    - back off by too little: contention still high
    - back off by too much: missed opportunity
  - Exponential back-off: back-off interval raising exponentially with each failure

Better: test-and-test-and-set lock (TTSL)

- Busy-wait with read operations
  - Keep testing with ordinary load
    - cached lock variable will be invalidated when release occurs
  - When value changes (to 0), try to obtain lock with test&set
    - only one attempter will succeed; others will fail and start testing again

TTSL Implementation

```
lock: ld R1, &lockvar  // R1 = MEM[&lockvar]
      bnz R1, lock;    // jump to lock if R1 != 0
      t&s R1, &lockvar // R1 = MEM[&lockvar];
                       // if (R1==0)MEM[&lockvar]=1
      bnz R1, lock;    // jump to lock if R1 != 0
      ret              // return to caller

unlock: st &lockvar, #0 // MEM[&lockvar] = 0
        ret             // return to caller
```

ce

Request	P1	P2	P3	BusRequest
Initially	-	-	-	-
ld1	E	-	-	BusRd
t&s1	M	-	-	-
ld2	S	S	-	BusRd
ld3	S	S	S	BusRd
ld2	S	S	S	-
unlock1	M	I	I	BusUpgr
ld2	S	S	I	BusRd
t&s2	I	M	I	BusUpgr
ld3	I	S	S	BusRd
ld3	I	S	S	-
unlock2	I	M	I	BusUpgr
ld3	I	S	S	BusRd
t&s3	I	I	M	BusUpgr
unlock3	I	I	M	-

TTSL vs. test-and-set lock

- Successful lock acquisition:
  - 2 bus transactions in TTSL
    - 1 BusRd to intervene a remotely cached block
    - 1 BusUpgr to invalidate all remote copies
  - vs. only 1 in test&set lock
    - 1 BusRdX to invalidate all remote copies
- Failed lock acquisition:
  - 1 bus transaction in TTSL
    - 1 BusRd to read a copy
    - then, loop until lock becomes free
  - vs. unlimited with test&set lock
    - Each attempt generates a BusRdX
- Overall, test&set lock performance can be really bad
  - Time in critical section likely > time to acquire a free lock

Improving Lock Implementation

- TTSL is an improvement over test-and-set lock
- But complex to implement
- Bus locking
  - has a limited applicability
  - not scalable with fine grain locks
- Cache block locking
  - Expensive, must rely on buffering or NACK
- Instead of providing atomicity, can we provide an illusion of atomicity instead?
  - Detect violation to atomicity
  - Cancel the store if that happens (must not allow store's new value to be visible to other processors)
  - Repeat all other instructions (load, branch, etc.)

Two New Instructions

- Load Linked/Locked (LL)
  - reads a word from memory, and
  - the address is stored in a special LL register
- Anything that may break atomicity clears out the LL register
  - Context switch occurs
  - Invalidation to the address in LL register occurs
- Store Conditional (SC)
  - tests if the address in the LL register matches the store address
  - if so, it stores into a word in memory
  - else, the store is canceled

Load Linked/Store Conditional Lock

```
lock: LL R1, &lockvar // R1 = MEM[&lockvar];
                      // LINKREG = &lockvar
      bnz R1, lock;   // jump to lock if R1 != 0
      addi R1, #1     // R1 = 1
      SC R1, &lockvar // MEM[&lockvar] = R1;
      bnz R1, lock;   // jump to lock if SC
  fails
      ret             // return to caller

unlock: st &lockvar, #0 // MEM[&lockvar] = 0
      ret               // return to caller
```

LL/SC vs. TTSL

- Similar bus traffic
  - Spinning using loads => no bus transactions when the lock is not free
  - Successful lock acquisition involves two bus transactions
- But a failed SC does not generate bus transaction vs. t&s

Request	P1	P2	P3	BusRequest
Initially	-	-	-	-
ll1	E	-	-	BusRd
sc1	M	-	-	-
ll2	S	S	-	BusRd
ll3	S	S	S	BusRd
ll2	S	S	S	-
unlock1	M	I	I	BusUpgr
ll2	S	S	I	BusRd
sc2	I	M	I	BusUpgr
ll3	I	S	S	BusRd
ll3	I	S	S	-
unlock2	I	M	I	BusUpgr
ll3	I	S	S	BusRd
sc3	I	I	M	BusUpgr
unlock3	I	I	M	-

Lock Implementation Problems

- A lock highly contended by p threads
- There are O(p) attemps to acquire & release a lock
- A single release invalidates O(p) caches
- Causing subsequent O(p) cache misses
- Hence, each critical section causes $O(p^2)$ bus traffic

- Fairness problem: no guarantee that a thread that contends for a lock will eventually acquire it

- Better lock implementations?

Module 9.2 – Scalable Lock, Barrier

Ticket Lock

- Ensures fairness
- but still incur $O(p^2)$ traffic

- Uses a concept of a queue
- A thread attempting to acquire a lock is given a ticket number in the queue
- Lock acquisition order follows the queue order

Implementation

```
ticketLock_init(int *next_ticket, int *now_serving)
{
 *now_serving = *next_ticket = 0;
}

ticketLock_acquire(int *next_ticket, int *now_serving)
{
 my_ticket = fetch_and_inc(next_ticket);
 while (*now_serving != my_ticket) {};
}

ticketLock_release(int *next_ticket, int *now_serving)
{
  now_serving++;
}
```

Illustration

Steps	next_ticket	now_serving	my_ticket		
			P1	P2	P3
Initially	0	0	-	-	-
P1: f&i	1	0	0	-	-
P2: f&i	2	0	0	1	-
P3: f&i	3	0	0	1	2
P1:now_serving++	3	1	0	1	2
P2:now_serving++	3	2	0	1	2
P3:now_serving++	3	3	0	1	2

Array-Based Queueing Lock (ABQL)

- Ensures fairness
- Each release invalidates only one cache

- Similar to ticket lock, uses a concept of a queue
- A thread attempting to acquire a lock is given a ticket number in the queue
- Lock acquisition order follows the queue order

Implementation

```
ABQL_init(int *next_ticket, int *can_serve)
{
 *next_ticket = 0;
 for (i=1; i<MAXSIZE; i++)
   can_serve[i] = 0;
 can_serve[0] = 1;
}
ABQL_acquire(int *next_ticket, int *can_serve)
{
 *my_ticket = fetch_and_inc(next_ticket);
 while (can_serve[*my_ticket] != 1) {};
}
ABQL_release(int *next_ticket, int *can_serve)
{
 can_serve[*my_ticket + 1] = 1;
 can_serve[*my_ticket] = 0; // prepare for next time
}
```

Illustration

Steps	next_ticket	can_serve[]	my_ticket		
			P1	P2	P3
Initially	0	1000	-	-	-
P1: f&i	1	1000	0	-	-
P2: f&i	2	1000	0	1	-
P3: f&i	3	1000	0	1	2
P1:can_serve[1]=1	3	0100	0	1	2
P2:can_serve[2]=1	3	0010	0	1	2
P3:can_serve[3]=1	3	0001	0	1	2

Comparison

Criteria	Test&set	TTSL	LL/SC	Ticket	ABQL
Uncontested latency	Lowest	Lower	Lower	Higher	Higher
1 release max traffic	O(p)	O(p)	O(p)	O(p)	O(1)
Wait traffic	High	-	-	-	-
Storage	O(1)	O(1)	O(1)	O(1)	O(p)
Fairness guaranteed?	no	no	no	yes	yes

Discussion

- Design must balance latency vs. scalability
  - ABQL not necessarily best
  - Often LL/SC locks perform very well
  - Scalable programs rarely use highly-contended locks (e.g. think of linked list parallelization using a fine grain approach)
- Fairness sounds good in theory, but
  - Must ensure no context switch or any long delay of the next lock holder

Barriers

- In current systems, typically implemented in software using locks, flags, counters
- Adequate for small systems
- Not scalable for large systems

Example of usage

```
..
parallel region
BARRIER
parallel region
BARRIER
..
```

A Simple (But Wrong) Centralized Barrier

```
// variables used in a barrier and their initial values
int numArrived = 0;
lock_type barLock = 0;
int canGo = 0;

// barrier implementation
void barrier () {
 lock(&barLock);
  if (numArrived == 0)  // first thread sets flag
    canGo = 0;
  numArrived++;
  myCount = numArrived;
 unlock(&barLock);

 if (myCount < NUM_THREADS) {
  while (canGo == 0) {}; // wait for last thread }
 else { // last thread to arrive
  numArrived = 0; // reset for next barrier
  canGo = 1; // release all threads
 }
}
```

Centralized Barrier with Sense Reversal

```
// variables used in a barrier and their initial values
int numArrived = 0;
lock_type barLock = 0;
int canGo = 0;

// barrier implementation
void barrier () {
  valueToWait = 1 - valueToWait; // toggle it
  lock(&barLock);
     numArrived++;
     myCount = numArrived;
  unlock(&barLock);

  if (myCount < NUM_THREADS) {
    while (canGo != valueToWait) {}; // wait for last thread }
  else { // last thread to arrive
    numArrived = 0; // reset for next barrier
    canGo = valueToWait; // release all threads
  }
}
```

Other Topics

- Combining Tree barrier => see Section 9.2.2
- Hardware barrier network => see Section 9.2.3