Synchronization

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Types of Synchronization

• Mutual Exclusion
  – Locks

• Event Synchronization
  – Global or group-based (barriers)
  – Point-to-point
Busy-Waiting vs. Blocking

- Busy-waiting is preferable when:
  - Scheduling overhead is larger than expected wait time
  - Processor resources are not needed for other tasks
  - Schedule-based blocking is inappropriate

- E.g. inside OS kernel
## A Simple Lock

```assembly
lock:    ld    r1, memory-location
cmp    r1, #0
bnz     lock
st      memory-location, #1
ret

unlock:  st    memory-location, #0
        ret
```

- Any problem?
Need Atomic Primitive!

- Two operations but inseparable instruction
  - Test&Set
  - Swap
  - Fetch&Op
    - Fetch&Incr, Fetch&Decr
  - Compare&Swap
Test&Set based Lock

```c
lock:      t&s  r1, memory-location
           bnz    lock
           ret

unlock:   st      memory-location, #0
           ret
```

- **Test&Set (t&s)**
  - t&s  `reg, mem-addr`
    - **Atomic operation:**
      ```c
      {  
        reg = *mem-addr;
        *mem-addr = #1;
        compare reg, #0
      }
      ```
Test&Set Lock: Coherence Traffic

Processor 1

BusRdX
Update line in cache (set to 1)
Invalidate line

Processor 2

Invalidate line

BusRdX
Update line in cache (set to 1)
Invalidate line

Processor 3

Invalidate line

BusRdX
Update line in cache (set to 1)
Invalidate line

BusRdX
Update line in cache (set to 1)
Invalidate line

BusRdX
Update line in cache (set to 1)
Invalidate line

BusRdX
Update line in cache (set to 1)
Invalidate line

test&set
lock owner
Test&Set with Backoff

• Upon failure, delay for a while before retrying
  – Either constant delay or exponential backoff

• Tradeoffs
  – Pros: much less network traffic
  – Cons: exponential backoff can cause starvation for high-contention locks (since new requestors back off for shorter times)

• But, exponential found to work best in practice
Test&Set Lock Performance

- `lock(L); critical-section(c); unlock(L);`
  - `c` is actually delay(μsec) to determine the size of critical-section
  - Same total # of lock calls as `p` increases
  - Same # of tasks dequeued from central task queue
  - Measure time per lock-unlock pair
Test and Test&Set

lock:  ld r1, memory-location
       cmp r1, #0
       bnz lock
       t&s r1, memory-location
       bnz lock
       ret

• Pros
  – While spinning, accesses lock variable in cache

• Cons
  – Can still generate a lot of traffic, when many processors perform test&set
  – Theoretically, still starvation can occur
Test Test&Set Lock: Coherence Traffic

Processor 1

- BusRdX
- Update line in cache (set to 1)
- Invalidate line

Processor 2

- Invalidate line
- BusRd

Processor 3

- Invalidate line
- BusRd

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Processor 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>BusRdX</td>
<td>Invalidate line</td>
<td>Invalidate line</td>
</tr>
<tr>
<td>Update line in cache (set to 0)</td>
<td>BusRd</td>
<td>BusRd</td>
</tr>
<tr>
<td>Invalidate line</td>
<td>BusRd</td>
<td>BusRd</td>
</tr>
<tr>
<td>test&amp;set</td>
<td>Update line in cache (set to 1)</td>
<td>Update line in cache (set to 1)</td>
</tr>
<tr>
<td>lock owner</td>
<td>Invalidate line</td>
<td>Invalidate line</td>
</tr>
</tbody>
</table>
Test&Set with Update Protocol

• Test&Set on update-based cache coherence
  – t&s sends updates to processors that cache the lock

• Tradeoffs
  – Pros: good for bus-based machines
  – Cons: still lots of traffic on distributed networks

• Main problem with test&set based schemes
  – A lock release causes all waiters to try to get the lock, using test&set
Load-Locked and Store-Conditional

- Improved ISA primitives on modern processors
  - Not generating “invalidation” on failed attempts
  - Can build a range of atomic operations: read-modify-write
    - test&set, fetch&op, compare&swap
- A pair of special instructions
  - LL (load-locked or load-linked)
    - Load a synch variable into register,
    - Arbitrary instructions, which manipulate register value, follows
  - SC (store-conditional):
    - Write the register back to the memory location (the synch variable), if and only if no other processor has written to that location (or cache block) since it completes the LL
  - i.e. LL-SC is atomically read-modify-write, if SC succeeds
    Otherwise, store fails (without any “invalidation”)

LL and SC based Lock

lock:  ll  r1, memory-location
       bnz  lock
       sc  memory-location, r2  # r2 = 1
       beqz lock  # if fail, goto lock
       ret

Unlock:  st  memory-location, #0
          ret

- Lock and unlock implementation with LL and SC
  - Spinning in cache without “invalidation”
    - Similar effect to test-and-test&set,
    - but needs $O(p)$ traffic when unlocked (it will invalidates all others) and get the lock variable from memory
  - No “invalidation” when contend to get the lock (i.e. failed SCs)
    - Test-and-test&set generates “invalidation” on test&set phase
    - One successful SC “invalidates” others : $O(1)$ traffic
  - Theoretically, starvation is still possible
Ticket Lock

- **Two counters**
  - `next_ticket` (number of requestors)
  - `now_serving` (number of releases that have happened)
  - E.g. teller line in bank
    - get the ticket number and wait until your number is on the LED display

- **Lock**
  - First do a `fetch&incr` on `next_ticket` (serialize the waiting line)
    - `my_ticket = fetch&inc(next_ticket)`
  - When release happens, poll the value of `now_serving` (busy-waiting)
    - `If now_serving == my_ticket, then I got the lock`

- **Unlock**
  - Increment `now_serving` (no atomic operation is needed)
Ticket Lock (cont’d)

```c
struct lock {
    volatile int next_ticket;
    volatile int now_serving;
};

void lock(lock* l) {
    int my_ticket = atomic_inc(&l->next_ticket);
    while (my_ticket != l->now_serving);
}

void unlock(lock* l) {
    l->now_serving++;
    // no atomic op. needed
}
```
Ticket Lock (cont’d)

- **Pros**
  - Guaranteed FIFO order (fairness), no starvation possible
  - Latency can be low
    - Fetch&incr when the process first arrives at the lock – presumably arrival times are dispersed, not at the same time
    - But, test&set attempts on lock release, which can be heavily contended
  - Traffic can be quite low (comparable to LL-SC lock)
    - Similar to LL-SC lock, only one process issues “invalidation”

- **Cons**
  - Traffic is not guaranteed to be $O(1)$ per lock acquire --- but $O(p)$
    - When `now_serving` is updated, all competing processors get read-misses
    - All competing processors need to read `now_serving` from the memory
    - Could reduce contention by backoff proportional to the $\text{diff}(\text{my_ticket}, \text{now_serving})$
Array-Based Locks

- Every process spins on a unique location (i.e. different location)
  - Ticket lock spins on a single `now_serving` variable, vs.
  - Array-based lock structure provides $p$ locations for $p$ processes

- Lock
  - Using fetch&incr, get the next available location in the array with wraparound

- Unlock
  - Write “unlocked” value to the next location of its own
  - Only the processor spinning on that location gets invalidated
Array-Based Locks (cont’d)

```c
struct lock {
    volatile int status[P];
    volatile int head;
};

int my_element;
void lock(lock* l) {
    my_element = atomic_inc(&l->head); // circular inc
    while ( l->status[my_element] == 1 );
}

void unlock(lock* l) {
    l->status[next(my_element)] = 0;
}
```
Array-Based Locks (cont’d)

• Pros
  – Guaranteed **FIFO order** (same as ticket lock)
  – **O(1) traffic** with coherence cache (unlike ticket lock)
    • Unlock (release) updates one location of the array on which the process getting lock was spinning

• Cons
  – Require space per lock proportional to \( p \)
    • \( O(p*n) \) space for \( p \) processes and \( n \) locks
  – Spin can run on remote memory
    • Bad when remote data caching is disabled
Queue-based Lock

• Each process spin on a unique location
  – A distributed liked list (or a queue) of waiters on a lock
  – Head node is a lock-holder
  – Every other nodes spin on their local memory
• Spinning variables are linked of each other
Queue-based Lock (cont’d)

```c
struct qnode {
    struct qnode* next;
    int locked;
};
typedef struct qnode lock; // tail

void lock(lock* lock, struct qnode *i) {
    i->next = NULL;
    qnode *prev = fetch_and_store(lock, i); // add to tail
    if ( prev != NULL ) {
        i->locked = 1;
        prev->next = i;
        while (i->locked); // wait until prev unlocks me
    }
}

void unlock(lock *lock, struct qnode *i) {
    if (i->next == NULL) {
        if ( compare_and_swap(lock, i, NULL) ) // atomic exit when there is no waiter
            return;
        while (i->next == NULL); // wait until the linked list is consistent
    }
    i->next->locked = 0;
}
```
Queue-based Lock (cont’d)

P1 → P2 → P3 → P4

run
spin
spin
incoming

P1 → P2 → P3 → P4

run
spin
spin
spin

P1 → P2 → P3 → P4

leaving
spin
spin
spin

P2 → P3 → P4

run
spin
spin
Queue-based Lock (cont’d)

• Pros
  – Guaranteed **FIFO order** (same as array-based lock)
  – O(1) traffic with coherence cache (same as array-based lock)
    • Unlock (release) updates one location of the array on which the process getting lock was spinning
  – Spin on local memory
  – O(p+n) space for p processes and n locks

• Cons
  – Requires a local “queue node” to be passed in as a parameter
Point-to-Point Event Synchronization

• Implementation
  – Busy-waiting on ordinary variable (1:1 synch, not 1:p-1 synch)
  – Blocking uses semaphore with a sleep queue

• Software algorithm

```plaintext
P1
a = f(x); // produce a
flag = 1;

P2
while (flag == 0);
b = g(a); // consume a
```

• Hardware support
  – A full-empty bit associated with every word in memory
  – Write to a location only if its full-empty bit is empty and set to full
  – Read from a location only if its full-empty bit is full and set to empty
Global Event Synchronization

• **Barrier** – global event synchronization
  – In parallel applications, keep pace of parallel threads
  – Parallel regions in OpenMP (parallel loops, sections)

• **Types of barrier implementation**
  – Centralized
  – Software combining tree
  – Hardware supported barrier
Centralized Barrier

- **Centralized software barrier**
  - A shared counter – # of processes that have arrived at the barrier
  - Poll the shared counter until all have arrived
- **Simple, but incorrect barrier implementation**

```c
Barrier (bar, p) {
    // bar: barrier struct,  p: total #processes
    LOCK(bar.lock);
    if (bar.counter == 0) bar.flag = 0;  // reset flag if first to reach
    mycount = bar.counter++;
    UNLOCK(bar.lock);

    if (mycount == p) {
        bar.counter = 0;  bar.flag = 1;  // reset counter for next barrier
    }
    else while (bar.flag == 0);  // busy wait for release
}
```

- If two `Barrier()`’s are invoked with the same `bar` variable consecutively, flag can be reset to 0 before all processes leave the earlier barrier
- Need to make sure all processes have left previous barrier
Centralized Barrier (cont’d)

- **Sense reversal**
  - Instead of waiting flag turns into 1 (indication of barrier end)
  - Waiting for different values in consecutive instances
    - Wait for the flag turns into 1 in one instance
    - Wait for the flag turns into 0 in the next instance

```c
Barrier (bar, p) {
    local_sense = !local_sense; // toggle private sense variable
    LOCK(bar.lock);
    mycount = bar.counter++;
    if (bar.counter == p) {
        UNLOCK(bar.lock);
        bar.counter = 0;
        bar.flag = local_sense;
    } else {
        UNLOCK(bar.lock);
        while (bar.flag != local_sense); // busy wait for release
    }
}
```
Software Combining Tree Barrier

• Problem in centralized barrier
  – All processes contend for the same lock and flag variables

• Combining tree
  – Works well on distributed networks, but not on bus
  – A tree with p leaves has $2^p - 1$ nodes, generates more traffic on a bus than centralized barrier
Hardware Supported Barriers

• Hardware primitives
  – Lock implementation supports
  – Piggybacking on the first read miss response
    • Reduce the impact of multiple bus miss transactions on flag at release

• Hardware barrier
  – Wired-AND line separated from system bus
  – Each process turn the line ON
  – In practice, more wires required for consecutive barriers
Lock-Free Algorithms

- Multiple threads share the same data structure
  - But carefully designed algorithms do not require lock
    - Instead, use atomic instructions
    - No lock/atomic inst. for 1 producer and 1 consumer

- Lock-Free Queue/Stack
- Lock-Free Ring buffer
- Lock-Free Linked list
Lock-Free Algorithms

- Use atomic instructions
  - Add/delete in LIFO, FIFO queues implemented with lists

- Compare-and-swap (CAS)
  - Atomic exchange after
  - CAS(a, b, c)

```c
add_head(node *F) {
    node *Q;
    do {
        Q = head.ptr;
        F.ptr = Q;
        c = CAS(head.ptr, Q, F);
    } while (!c)
}
```

CAS(a, b, c)
if (a == b) { a = c; return True; }
return False;
Lock-Free Algorithms

- Circular FIFO buffer
  - Single reader, single writer
  - Neither lock nor atomic instruction is needed

Enqueue(n) {
    if (((tail+1)%size +1)%size == head)
        return -1; // full
    tail = (tail+1)% size;
    buffer[tail] = n;
    return tail;
}

Dequeue(&n) {
    if (((tail+1)%size == head)
        return -1; // empty
    *n = buffer[head];
    head = (head+1)% size;
    return head;
}
Summary

• Synchronization
  – Mutual exclusion – lock
  – Event synchronization – point-to-point synch, barrier

• Lock
  – ISA supported – atomic instruction (test&set, fetch&op, cmp&swap)
  – Algorithms (test and test&set, LL-SC, ticket lock, array-based lock)

• Barrier
  – Centralized vs. combining tree

• Lock-free algorithms
  – List, circular queue algorithms, etc.
Performance in Real System

- **Environment**
  - Two Intel Xeon e5-2680 (12 cores per socket)
  - Snoop-based cache coherence
  - L1, L2 and L3 are write-back caches
Performance in Real System

- **BusRdX latency to a modified cacheline**
  - Ex. Core 1 performs test&set and then Core 2 performs test&set
Performance in Real System

• BusRdX latency to a modified cacheline
  – Ex. Core 1 performs test&set and then Core 2 performs test&set

  – Local access
    • L1: 1.3ns, L2: 3.4ns, L3: 13ns
  – On the same processor
    • L1: 28ns, L2: 25ns, L3: 13ns
  – On the remote processor
    • 102-109 ns

  – Non-contending case
Tested Workload

- Each thread executes the following code

```c
register int count = 0;
while (1) {
    lock(); count++; unlock();
}
```

- All threads begin at the same time, run for 5 seconds and stop

- Tested locks
  - spinlock (just test&set lock)
  - spinlock with back-off (not exponential)
  - ticket spinlock
  - mcs (queue-based) spinlock
Lock Performance

- Lock acquisition-release time
Lock Fairness

- # of locks acquired by each thread
ipc
References

- COMP422: Parallel Computing by Prof. John Mellor-Crummey @ Rice Univ.
- CMU 15-428: Parallel Computer Architecture and Programming by Prof. Kayvon Fatahalian @ CMU