Synchronization

CSE 661 – Parallel and Vector Architectures

Dr. Muhamed Mudawar

Computer Engineering Department

King Fahd University of Petroleum and Minerals

Types of Synchronization

- "A parallel computer is a collection of processing elements that **cooperate** and communicate to solve large problems fast"
- Types of Synchronization
  - Mutual Exclusion
    - Lock – Unlock
  - Event synchronization
    - Point-to-point
    - Group
    - Global (barriers)

History and Perspectives

- Much debate over hardware primitives over the years
  - High-level language advocates want hardware locks/barriers
    - But it goes against the "RISC" philosophy and has other problems
  - Speed versus flexibility
- But how much hardware support?
  - Atomic read-modify-write instructions
  - Separate lock lines on the bus (older computers, inflexible)
  - Specialized lock registers shared by processors (Cray XMP)
  - Lock locations in memory
  - Hardware full/empty bits (Tera)

Role of User and System

- User wants to use high-level synchronization operations
  - Locks, barriers, . . . etc.
  - Doesn't care about implementation
- System designer
  - How much hardware support in implementation?
  - Speed versus cost and flexibility
  - Waiting algorithm difficult to implement in hardware
- Popular trend:
  - System provides simple atomic read-modify-write primitives
  - Software libraries implement lock, barrier algorithms using these
  - But some propose and implement full-hardware synchronization

Atomic Read-Modify-Write Operations

- Most modern methods use a form of atomic read-modify-write
- IBM 370
  - Atomic **compare&swap** for multiprogramming
  - Three operands: location, register to compare with, register to swap with
- Intel x86
  - Instructions can be prefixed with a **lock modifier** (atomic execution)
- SPARC
  - Atomic register-memory **swap** and **compare&swap** instructions
- MIPS and IBM Power: pair of instructions
  - **Load-Locked** (LL) and **Store-Conditional** (SC)
  - Later used by PowerPC and DEC Alpha too
  - Allows a variety of higher-level RMW operations to be constructed

Components of a Synchronization Event

- Acquire method
  - Acquire right to the synchronization
    - Enter critical section or proceed past event synchronization
- Waiting algorithm
  - Wait for synchronization to become available when it isn't
    - Lock is not free, or event has not yet occurred
- Release method
  - Enable other processors to acquire right to the synchronization
    - Implementation of unlock
    - Notifying waiting process at a point-to-point event that event has occurred
    - Last process arriving at a barrier to release the waiting processes
- Waiting algorithm is independent of type of synchronization

Waiting Algorithms

- Blocking
  - Waiting process or thread is de-scheduled
  - Overhead of context-switch, suspending/resuming a process/thread
  - Processor becomes available to other processes or threads
- Busy-waiting
  - Waiting process repeatedly test a location until it changes value
  - Releasing process sets the location
  - Consumes processor resources, cycles, and cache bandwidth
- Busy-waiting better when
  - Scheduling overhead is larger than the expected wait time
  - Processor resources are not needed for other processes
- Hybrid methods: busy-wait for a while, then block

Mutual Exclusion: Hardware Locks

- Separate lock lines on the bus
  - Holder of a lock asserts the line
  - Priority mechanism for multiple requestors
- Inflexible, so not popular for general purpose use
  - Few locks can be in use at a time (one per lock line)
  - Hardwired waiting algorithm
- Lock registers (Cray XMP)
  - Set of registers shared among processors
  - Primarily used to provide atomicity for higher-level software locks

Simple Software Lock: First Attempt

```
lock:   ld   register, location   /* load register ← location */
        bnz  register, lock       /* if register not 0, try again */
        st   location, #1         /* store 1 to mark it locked */
        ret                       /* return control to caller */

unlock: st   location, #0         /* store 0 to unlock location */
        ret                       /* return control to caller */
```

- Problem: lock needs atomicity in its own implementation
  - Load (test) and Store (set) of lock variable by a process not atomic
- Solution: **atomic read-modify-write** or **exchange** instructions
  - Atomically test value of location and set it to another value

Atomic Exchange Instructions

- **`Test&Set register, location`**
  - Value in *location* read into specified *register*
  - Constant 1 stored into *location*
- **`Swap register, location`**
  - Contents of memory *location* and *register* are exchanged
- **`Fetch&Op register, location`**
  - Value in *location* is read into specified *register*
  - Apply operation *Op* in Fetch&Op and update memory *location*
  - Examples: **`Fetch&Inc, Fetch&Dec, Fetch&Add,`** etc.
- Atomicity of instruction is ensured
  - The read and write components cannot be split
- Atomic instructions can be used to build locks

Simple Test&Set Lock

```
lock:   t&s  register, location   /* test-and-set location */
        bnz  register, lock       /* if not 0, try again */
        ret                       /* return control to caller */
unlock: st   location, #0         /* store 0 into location */
        ret                       /* return control to caller */
```

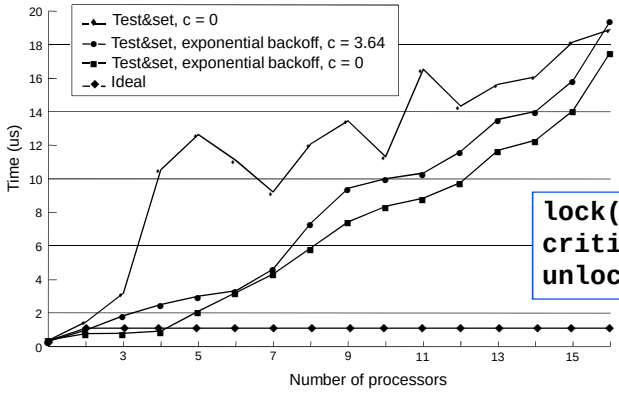
- Test&Set succeeds if a zero is loaded into register
  - Otherwise, busy wait until lock is released by another process
- Performance: *test&set* generates bus traffic (for coherence)
  - Happens when multiple processors compete to acquire the same lock
  - *Test&Set* causes bus transaction when cache block is shared or invalid
    - Bus read-exclusive transaction that invalidates others (invalidation protocol)
    - Bus-update transaction (update protocol)

Test&Set Lock With Backoff

```
backoff: compute backoff delay
         busy wait loop
lock:    t&s  register, location   /* test-and-set location */
         bnz  register, backoff    /* if not 0, try again */
         ret                       /* return control to caller */
unlock:  st   location, #0         /* store 0 into location */
         ret                       /* return control to caller */
```

- Basic Idea of Backoff
  - Insert delay after an unsuccessful attempt to acquire lock
  - Reduce frequency of issuing test&sets while waiting
  - Linear backoff: unsuccessful i^{th} time delay = $k \times i$ (constant k)
  - Exponential backoff: i^{th} time delay = k^{i} (works quite well empirically)

Test&Set Lock Performance



On SGI Challenge

```
lock(l);
critical_section(c);
unlock(l);
```

- Same total number of lock calls, independent of # of processors *p*
- Measure time per lock transfer; time in critical section not counted
- Irregular nature is due to timing dependence of the bus contention effects

Performance Criteria for Locks

- Low Latency
  - If a lock is free and no other processor is trying to acquire it
- Low Traffic
  - If many processors are trying to acquire a lock
  - Processors should do so with as little bus traffic as possible
  - Contention can slow down lock acquisition and unrelated bus transactions
- Scalability
  - Latency and bus traffic should not increase with number of processors
- Low Storage Cost
  - Lock storage should be very small, independent of *p*
- Fairness
  - Ideally, processors should acquire locks in same order as their requests

Test-and-Test&Set Lock

```
lock:   ld   register, location     /* load first from location*/
        bnz  register, lock         /* if not 0, try again */
        t&s  register, location     /* test-and-set */
        bnz  register, lock         /* if not 0, try again */
        ret                         /* return to caller */
unlock: st   location, #0           /* write 0 to location */
        ret                         /* return to caller */
```

- Improved busy-waiting with *load* rather than *test&set*
  - Keep testing with ordinary load in cache ⇒ **No bus transactions**
  - Cached lock variable will be invalidated when release occurs
  - When value changes to 0, try to obtain lock with *test&set*
  - Only one attempt will succeed; others will fail and start testing again

Challenges

- Test-and-Test&Set Lock
  - Slightly higher latency than simple *Test&Set*, but less traffic
  - But still all processors rush out on release to …
    - Read miss on *load* and then *test&set*
  - Traffic is $O(p^2)$ for p processors, each to acquire the lock once
  - **Each one** of the p releases causes an invalidation that results in
    - $O(p)$ cache misses on load
    - $O(p)$ processes trying to perform the *test&set* resulting in $O(p)$ transactions
- Synchronization may have different needs at different times
  - Lock accessed with low or high contention
  - Different requirements: low latency, high throughput, fairness
- Rich area of software-hardware interactions
- Luckily, better hardware primitives as well as algorithms exist

Improved Hardware Primitives: LL-SC

- **Load-Locked** or **Load-Linked** (LL)
  - Load variable into a register and keep track of address
  - Address is saved in a **lock address register** and a **lock flag** is set
  - LL may be followed by instructions that modify the register value
- **Store-Conditional** (SC)
  - Tries to store back the same memory location read by load-locked
    - Provided no one has written to the variable since this processor's LL
  - Checks the **lock flag** and **address register** for an intervening write
  - If the flag has been reset by another write, then SC fails
  - If SC succeeds, instructions between LL-SC happened atomically
  - If it fails, doesn't write nor generate invalidations (need to retry LL)
  - Success or failure indicated by a return value

Load-Locked & Store-Conditional - contd

- Store-Conditional fails when
  - Detects intervening write to same address (lock flag is reset)
  - A context switch happened before SC (lock flag is reset)
  - Cache block was replaced (lock flag is reset)
  - However, failure of SC does not generate a bus transaction
- LL-SC are not lock-unlock respectively
  - Completion of Load-Locked does not mean obtaining exclusive access
  - Successful LL-SC pair does not even guarantee exclusive access
  - Only guarantee no conflicting write to lock variable between them
  - Failure of SC will have to retry the LL-SC code sequence

Simple Lock with LL-SC

```
lock:   ll      reg, location     /* load locked reg ← location */
        bnz     reg, lock         /* try again if not zero */
        move    reg, #1           /* reg ← 1 */
        sc      location, reg     /* store cond location ← 1 */
        beqz    reg, lock         /* try again if sc failed*/
        ret
unlock: st      location, #0      /* write 0 to location */
        ret
```

- Test with LL, but failed SC attempts don't generate invalidations
- Better than Test-and-Test&Set, but still not a fair lock
  - **Traffic is reduced from $O(p^2)$ to $O(p)$ per lock acquisition for p processors**
  - **Each one** of the p releases causes an invalidation that results in
    - $O(p)$ bus read transactions to shared *location*
      - Separate read misses to same block can be combined if all caches observe bus read
    - Only one bus transaction for successful SC that causes invalidations
    - Failed SC do not generate bus transactions (advantage over *test&set*)

Implementing Atomic Operations

- LL-SC pair can be used to implement atomic r-m-w operations
  - *Test&Set*, *Swap*, *Fetch&Op*, and other custom operations
  - Better than to build a lock-unlock around simple shared variable update
- Example on Implementing **Fetch&Inc** with LL-SC pair:

```
Fetch&Inc:
    ll      reg, location     /* load locked reg ← location */
    add     reg, reg, #1      /* increment register */
    sc      location, reg     /* store cond location ← reg */
    beqz    reg, Fetch&Inc    /* retry if sc failed*/
    ret
```

- But keep it small so SC is likely to succeed
- **Don't include store instructions between LL & SC**
  - **Cannot be undone**

Advanced Locking Algorithms

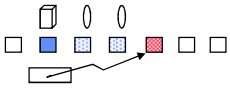
- Problem with Simple LL-SC lock
  - Doesn't reduce traffic to minimum and not a fair lock
  - Read misses by all waiting processes after lock release and SC by winner
  - Can use backoff with LL-SC to reduce bursty bus read traffic
- Advanced algorithms (useful for test&set and LL-SC)
  - Only one process to attempt to acquire lock upon release
    - Rather than all processes rushing to do test&set and issue invalidations
  - Only one process to have read miss when a lock is released
    - Rather than all processes rushing to read the lock variable (useful for LL-SC)
  - **Ticket lock** achieves first
  - **Array-based lock** achieves both, but with a cost in memory space
  - Both are fair (FIFO) locks as well

Ticket Lock

- Works like waiting line at bank
  - Two shared counters per lock:
    - **next_ticket** and **now_serving**
  - Acquire: **fetch&inc my_private_ticket ← next_ticket**
    - Atomic operation when acquiring lock only, can be implemented with LL-SC
  - Waiting: busy wait until **my_private_ticket** is equal to **now_serving**
    - No atomic operation for checking since each process has a unique ticket
  - Release: **increment now_serving**
- Fair FIFO order
- *O(p)* read misses at release, since all spin on **now-serving**
  - Like simple LL-SC lock, but key difference is fairness
  - Backoff can reduce number of read misses on release
    - Backoff proportional to **my_private_ticket – now_serving** may work well

Array-Based Lock

- Waiting processes poll on different locations in an array of size *p*
  - Requires a shared array **lock[ ]** and a shared **next_address** per lock
  - Distant **lock[ ]** elements in different cache blocks to avoid false sharing
- Acquire:
  - `fetch&add my_address ← next_address, block_size`
- Wait until `lock[my_address] == 1`
- Release:
  - `lock[my_address] = 0;`
  - `lock[my_address + block_size] = 1;` *// to wakeup next*



- Performance:
  - *O(1)* traffic per acquire with coherent caches
  - FIFO ordering as in ticket lock, but *O(p)* space per lock

Lock Performance on SGI Challenge

- All locks are implemented using LL-SC
- Delays are inserted as processor cycles, converted to μsecs

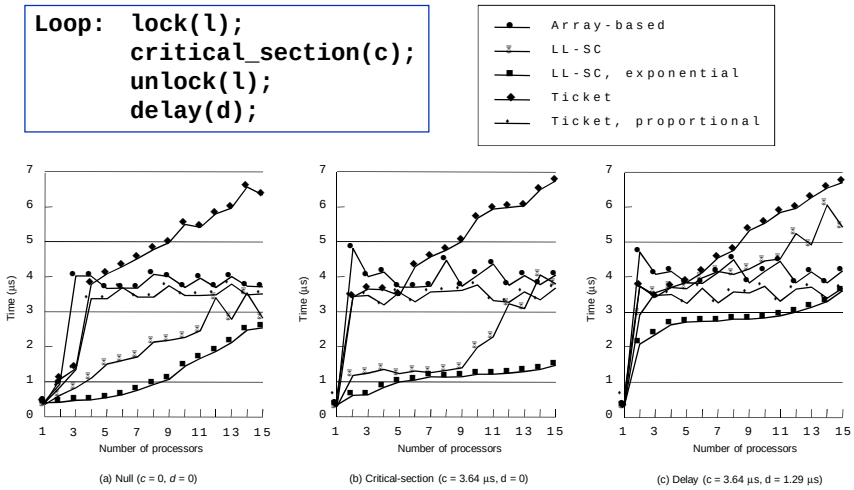
```
Loop:  lock(l);
       critical_section(c);
       unlock(l);
       delay(d);
```

- Fixed number of lock calls are executed, independent of *p*
- Delays *c* and *d* are subtracted out of total time
  - So that only time for lock acquisition, release, and contention is measured
- Three scenarios:

  (1) $c = 0$, $d = 0$ (2) $c = 3.64$ μs, $d = 0$, and (3) $c = 3.64$ μs, $d = 1.29$ μs

Lock Performance on SGI Challenge

```
Loop:  lock(l);
       critical_section(c);
       unlock(l);
       delay(d);
```



Lock Performance – cont'd

- Simple LL-SC lock seems to perform better
- LL-SC does best at small *p* due to unfairness
  - Especially when same processor does SC followed directly by LL
  - Same processor acquires the lock before another processor gets chance
  - With delay after release, performance of simple LL-SC deteriorates
  - LL-SC with exponential backoff performs best, but not necessarily fair
- Ticket lock with proportional backoff scales well
  - Backoff duration proportional to: **myTicketNumber – nowServing**
- Array-based lock also scales well
  - Only correct processor issues a read
  - Time per lock acquire is constant and does not increase with *p*
- Real benchmarks required for better comparisons

Point-to-Point Event Synchronization

- Software Methods:
  - Busy-waiting: use ordinary variables as flags
  - Blocking: use semaphores and system support
- Software Algorithms:
  - Flags as control variables

P1	P2

a = f(x); // set a
flag = 1;
	while (flag == 0) ; // do nothing – busy wait
	b = g(a); // use a

  - Value to control synchronization: when initial value (say 0) changes

P1	P2

a = f(x); // set a	while (a == 0) ; // do nothing – busy wait
	b = g(a); // use a

Hardware Support

- **Full/Empty bit** with each word in memory
  - Fine-grained word-level producer-consumer synchronization
  - Set to "full" when word is written (special store) with produced data
    - Producer does so when bit is "empty" and then set it to "full"
  - Clear to "empty" when word is consumed (special load)
    - Consumer reads word if "full" and then clears it to "empty"
  - Hardware preserves atomicity of bit manipulation with read or write
  - Concerns about flexibility
    - Single-producer-multiple-consumer synchronization
    - Multiple writes before consumer reads
    - Language/compiler support for the use of the special load/store

Global Barrier Event Synchronization

- Hardware barriers
  - Wired-AND line separate from address/data bus
  - Set input high when arrive, wait for output to be high to leave
  - In practice, multiple wires to allow reuse
  - Useful when barriers are global and very frequent
  - Difficult to support arbitrary subset of processors
    - even harder with multiple processes per processor
  - Difficult to dynamically change number and identity of participants
    - e.g. latter due to process migration
  - Not common today on bus-based machines
- Software algorithms implemented using locks, flags, counters
  - Let's look at software algorithms with simple hardware primitives

A Simple Centralized Software Barrier

```
struct bar_type {
   int counter = 0; lock_type lock; int flag = 0;
}

BARRIER (bar, p) {
   LOCK(bar.lock);
   if (bar.counter == 0)
      bar.flag = 0;                  // reset flag if first to reach
   mycount = bar.counter++;          // mycount is private
   UNLOCK(bar.lock);
   if (mycount == p) {               // last to arrive
      bar.counter = 0;               // reset for next barrier
      bar.flag = 1;                  // release waiters
   }
   else while (bar.flag == 0) {};    // busy wait for release
}
```

Simple Centralized Barrier – cont'd

- Barrier *counter* counts number of processes that have arrived
  - Lock barrier and increment counter when a process arrives
  - First process to arrive resets flag on which to busy-wait
  - Last process resets counter and sets flag to release *p*–1 waiting processes
- However, there is one problem with the *flag*

```
some computation …
BARRIER(bar,p)
some more computation …
BARRIER(bar,p)
```

- First process to exit first barrier enters second barrier
  - Resets *bar.flag* to 0 when other processes did not exit first barrier
  - Some processes might get stuck and will never be able to leave first barrier

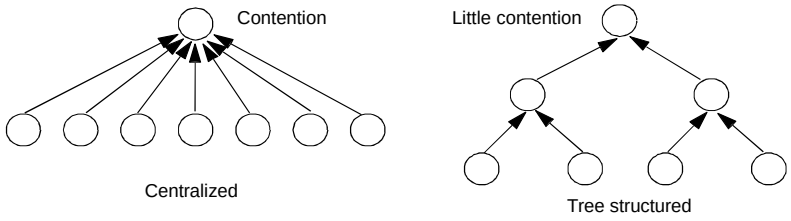
Centralized Barrier with Sense Reversal

- Sense reversal: toggles flag value (0 and 1) in consecutive times
  - Waiting condition changes in consecutive instances of barrier

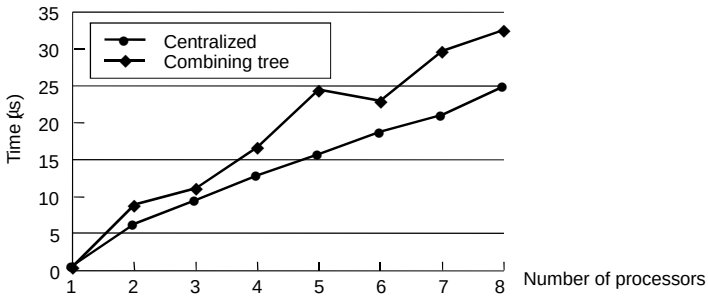
```
BARRIER (bar, p) {
    local_sense = !(bar.flag);          // private sense variable
    LOCK(bar.lock);
    mycount = bar.counter++;            // private variable
    UNLOCK(bar.lock);
    if (mycount == p) {                 // last process to arrive
        bar.counter = 0;
        bar.flag = local_sense;         // release waiters
    }
    else {
        while (bar.flag != local_sense); // busy wait
    }
}
```

Improving Barrier Algorithms

- Centralized Barrier: all *p* processors contend for same lock-flag
- Software Combining Tree
  - Only 2 processors access the same location in a binary tree
  - Valuable in distributed network: communicate along parallel paths
  - Ordinary reads/writes instead of locks at each node



Barrier Performance on SGI Challenge



- Centralized barrier involving *p* processors does quite well
  - Requires *O(p)* bus transactions
- Combining tree has similar total number of bus transactions
  - Can perform much better in a distributed network with parallel paths

Synchronization Summary

- Rich interaction of hardware-software tradeoffs
  - Hardware support still subject of debate
- Flexibility of LL and SC made them increasingly popular
- Must evaluate hardware primitives/software algorithms together
  - Primitives determine which algorithms perform well
- Evaluation methodology is challenging
  - Use of delays in micro-benchmarks
  - Should use both micro-benchmarks and real workloads
- Simple software algorithms do well on a bus
  - Algorithms that ensure constant-time access do exist, but more complex
  - More sophisticated techniques for distributed machines