CS6560: Parallel Computer Architecture

Synchronization

Madhu Mutyam

PACE Laboratory
Department of Computer Science & Engineering
Indian Institute of Technology Madras

Mar 14-22, 2016

Synchronization

- To ensure consistency of shared data structures
- Types of synchronization
  - Mutual exclusion (using *lock-unlock* pairs)
  - Point-to-point event synchronization (using *flags*)
  - Global event synchonization (using *barriers*)
- Hardware – speed
- Software – cost, flexibility, and adaptability

Components of a Synchronization Event

- *Acquire* method:
  - A process tries to acquire the right to the synchronization
- *Waiting* method:
  - A process waits for a synchronization to become available
- *Release* method:
  - A process to enable other processors to proceed past a synchronization event
- Waiting method is independent of the type of synchronization
  - Busy-waiting
  - Blocking
  - Hybrid

Mutual Exclusion

- Ensures that only one process enters the critical section
- Hardware locks – lock lines on the bus; lock locations in memory; lock registers
- Simple software lock algorithm:

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

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

Two processes can enter the critical section

Atomic Read-Modify-Write Instructions

- Hardware primates to implement synchronization
  - Atomic *test-and-set*, *swap*, *fetch-and-increment*, etc
- Atomic *test & set* instruction:
  - Value in a memory location is read into a specified register
  - Constant 1 is stored into the location atomically
  - Successful if the value loaded into the register is 0
  - Other constants could be used instead of 0 and 1

```
lock:   t&s     register,   location
        bnz     register,   lock        /* if not 0, try again */
        ret                             /* return control to caller */

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

Variants of Read-Modify-Write Instructions

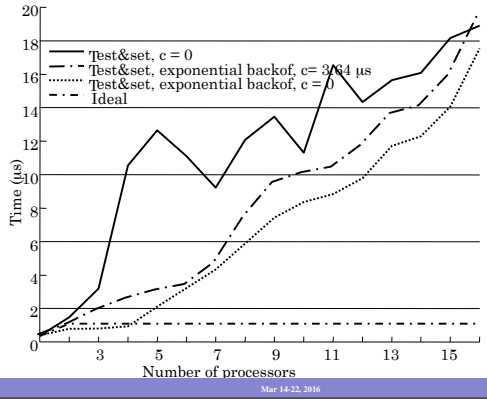
- Swap:
  - Atomically swaps the values in a memory location and a register
- Fetch & op:
  - Atomically reads the current value of a location into a register and writes the new value into the location
  - *fetch&increment*; *fetch&decrement*; *fetch&add*;
- Compare & swap:
  - Three operands: a memory location, register to compare with, register to swap with
  - Not commonly supported by RISC instruction sets

Enhancements to the Simple Lock Algorithm

- Reduce frequency of issuing test&set while waiting
  - *test&set* lock with back off
    - Do not back off too much or will be backed off when lock becomes free
    - Exponential back off works quite well
- Busy-wait with read operations rather than test&set:
  - *Test-and-test&set* lock
    - Keep testing with ordinary load
    - cached lock variable will be invalidated when release occurs
    - When value changes, try to obtain lock with test&set
    - only one attempter will succeed; others will fail and start testing again

Performance of Test&Set Lock

```
lock(L);
critical-section(c); // c is a delay parameter
unlock(L);
```



Instruction Sets Offering Hardware Support for Synchronization

- IBM 370: included atomic *compare&swap* for multiprogramming
- x86: any instruction can be prefixed with a lock modifier
- SPARC: atomic register-memory ops (*swap*, *compare&swap*)
- MIPS, IBM Power: no atomic operations, but pair of instructions (*load-locked*, *store-conditional*)

Load-Locked, Store-Conditional (LL-SC)

- Implementing single atomic memory operation is complex
- Use a pair of instructions to implement synchronization
- LL reads the memory location (synchronization variable) into a register
- Followed by arbitrary instructions to modify the register value
- SC tries to write the data of a register to the memory location
  - if and only if no other write to the location since the processor's LL
  - indicated by condition codes or a return value
- If SC succeeds, all three steps happened atomically
- If fails, does not write or generate invalidations

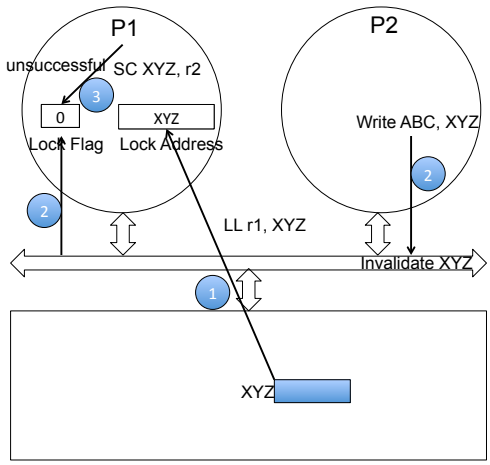
Implementing Lock and Unlock Using LL-SC

```
lock:   ll    r1,   location
        bnz   r1,   lock

        sc    location,  r2
        beqz            lock
        ret

unlock: st    location,  #0
        ret
```

Implementing LL-SC



Implementing LL-SC

- Requires a lock flag and a lock address register at each processor
- An LL operation sets the lock flag and puts the address of the read block in the lock address register
- Incoming invalidation (or update) requests are matched against the lock address register
  - A successful match resets the lock flag
- SC checks the lock flag to know whether an intervening write has occurred
- The lock flag can also be reset when
  - the lock variable is replaced from the cache
  - context switch happens between LL and SC

Performance Goals for Locks

- Low latency – especially when low contention
- Low traffic – especially under high contention
- Scalability – neither latency not traffic should scale with the number of processors
- Low storage cost – the information needed for a lock should be small
- Fairness – starvation or substantial unfairness should be avoided

Parameter	Locking Mechanism			
	T&S	T&S-backoff	T-and-T&S	LL-SC
Latency	Low uncontended	Low uncontended	High uncontended	Low
Traffic	High (if many processors compete)	Moderate	Moderate	Low, but not minimum
Scalability	Poor	Moderate	Good	Good
Storage	Low	Low	Low	Low
Fairness	No	No	No	No

Drawbacks with the Basic Locking Mechanisms

- When a lock is released,
  - all processes attempt to obtain the lock – lack of fairness
  - all processes incur a read miss – unnecessary traffic
- Advanced locking mechanisms:
  - **Ticket lock** – only one process attempts to obtain the lock
  - **Array lock** – only one process incurs a read miss

Ticket Lock

- Only one read-modify-write per acquire
- Two counters per lock (*ticket-number*, *now-serving*)
  - **Acquire**: *fetch&increment ticket-number*
  - **Waiting**: wait for *now-serving* == *ticket-number*
  - **Release**: increment *now-serving*
- Performance
  - low latency for low-contention – if *fetch&increment* cacheable
  - *fetch&increment* can be implemented with LL-SC
  - FIFO order, so it is fair
  - O(p) read misses at release, as all spin on same variable
  - Backoff by a duration proportional to the difference in its *ticket-number* and the *now-serving* number

Array-Based Locks

- A p-entry array for p processes competing for a lock
- **Acquire:**
  - *fetch&inc* obtains address on which to spin (next array element)
  - ensure that the addresses are in different cache lines/memories
- **Waiting:**
  - processes poll on different locations in the p-sized array
- **Release:**
  - set next location in array, thus waking up process spinning on it
- O(1) traffic per acquire with coherent caches
- FIFO ordering, as in ticket lock, but O(p) space per lock
- Processor can spin on a location that may not be in its local memory

Point-to-Point Event Synchronization

- Software algorithms:
  - Busy-waiting: use ordinary variables as flags
  - Blocking: use semaphores
- Hardware support: *full-empty* bit
  - Set when word is "full" with newly produced data on a write
  - Unset when word is "empty" due to being consumed on a read
  - Can be used for word-level producer-consumer synchronization
  - Hardware preserves atomicity of read or write with the manipulation of the *full-empty* bit
  - lack of flexibility
    - multiple consumers, or multiple writes before consumer reads?
    - needs language support to specify when to use

Hardware Barriers

- Wired-AND line separate from address/data bus
- Set input high when arrive, wait for output to be high to leave
- Useful when barriers are global and very frequent
- Difficult to support arbitrary subset of processors or multiple processes per processor
- Difficult to dynamically change the number and to adapt the configuration of participating processors

Centralized Software Barriers

- Implemented using locks, shared counters, and flags

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

Centralized Software Barrier with Sense Reversal

- Prevent process from entering until all have left the previous instance
  - Another counter can be used, but it increases latency and contention
- Sense reversal: wait for the flag to take different value consecutive times

```
BARRIER(bar_name, p) {
 local_sense = !(local_sense);/* toggle private sense
 LOCK(bar_name.lock);          variable */
 bar_name.counter++;
 if(bar_name.counter == p) {
      UNLOCK(bar_name.lock);
      bar_name.counter = 0;
      bar_name.flag = local_sense;}
 else{
      UNLOCK(bar_name.lock);
      while (bar_name.flag!=local_sense){ };
     }
}
```

Performance of Centralized Barriers

- **Latency** – critical path latency is proportional to the # of processors (p)
- **Traffic** – $3p + 1$ bus transactions
- **Scalability** – latency and traffic should increase slowly with the number of processors
- **Storage cost** – centralized counter and a flag
- **Fairness** – same processor should not always be the last one to exit the barrier
  - No such bias in centralized barrier

Thank You