CS4223 Synchronization

Trevor E. Carlson
National University of Singapore
tcarlson@comp.nus.edu.sg
(Slides from Tulika Mitra)

Learning Objectives

- Why do we need synchronization?
- Building block: Atomic read-modify-write instructions
- Lock implementations with increasing complexity:
  - Test & Set lock
  - Test and Test & Set Lock
  - Locks based on LL-SC primitive
- Fair lock implementations:
  - Ticket lock
  - Array-based queuing lock
- Barrier implementation

Synchronization

- A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast
- Synchronization for co-operation
  - Mutual exclusion (locks)
  - Event synchronization
    - Point-to-point
    - Global (barrier)

Shared memory code

```
begin parallel    // spawn a child thread
private int start_iter, end_iter, i;
shared int local_iter = 4;
shared double sum = 0.0, a[], b[], c[];
shared lock_type mylock;

start_iter = getid() * local_iter;
end_iter = start_iter + local_iter;
for (i=start_iter; i<end_iter; i++)
    a[i] = b[i] + c[i];
barrier;

for (i=start_iter; i<end_iter; i++)
      if (a[i] > 0) {
          lock (mylock);
          sum = sum + a[i];
          unlock (mylock);
      }
barrier;
end parallel // kill the child thread
print sum;
```

First attempt at simple software lock

```
lock:ld register, location  /* copy location to register */
     cmp location, #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 */
```

- Problem: lock needs atomicity in its implementation
  - Read (test) and write (set) of lock variable by a process not atomic
- Solution: atomic read-modify-write instruction
  - Atomically test value of location and set it to another value, return success or failure

Atomic Read-Modify-Write Instruction

- Specifies a location and register. In atomic operation:
  - Value in location read into a register
  - Another value stored into location
- Many variants
  - Varying degrees of flexibility in second part
- Simple example: test & set
  - Value in location read into a specified register
  - Constant 1 stored into location
  - Successful if value loaded into register is 0
- Can be used to build locks

How is atomicity ensured?

- Rely on cache coherence
- Obtain exclusive ownership of memory location M
- Do not allow the block to be stolen during atomic instruction execution
    - For the duration of the atomic instruction execution, lock the bus
    - Alternatively, the cache controller of the processor executing atomic instruction defer responding to all requests to the block until the atomic instruction completes execution

Other Read-Modify-Write Primitives

- Exchange Rx, M:
  - Atomically exchange the value in M with value in Rx
- Fetch & op M:
  - Read value stored in memory location M
  - Perform op (increment, decrement, addition, subtraction) to it
  - Store the new value to memory location M
- Compare & swap Rx, Ry, M
  - Three operands: location, register to compare , register to swap
  - Compare value in M with value in register Rx
  - If they match, write value in Ry to M
  - Copy the value in Rx to Ry

Simple Test & Set Lock

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

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

Simple T&S Performance

- Un-contended lock-acquisition latency low
  - One atomic instruction + branch
- Traffic requirement: high
  - Each lock acquisition attempt causes invalidation of all cached copies regardless of whether acquisition is successful or not

Test & Set with backoff

- Reduce frequency of issuing t&s while waiting
  - test & set lock with backoff
  - Don’t back off too much or will be in backed off state when lock becomes free
  - Exponential backoff works quite well empirically:

    i^{th} time = $k*c^i$

Test and Test & Set Lock

- Busy-wait with read operations rather than t&s
- Keep testing with ordinary load
  - cached lock variable invalidated when release occurs
- When value changes to 0, try to obtain lock with t&s
  - only one processor will succeed
  - others will fail and start testing again

Test and Test & Set lock

```
lock: ld register, location  /* copy location to register */
      cmp register, #0       /* compare with 0 */
      bnz lock               /* if not 0, try again */
      t&s register, location
      bnz lock
      ret                    /* return control to caller */

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

Test and Test & Set lock drawbacks

- All processor still rush out to read miss and t&s on release of the lock
- Poor fairness; can cause starvation

Load-Locked (Linked) Store-Conditional

- Goals:
  - Test with reads
  - Failed read-modify-write attempts do not generate invalidations
  - single primitive can implement range of r-m-w operations

LL-SC

- LL reads variable into register
- Follow with arbitrary instructions to manipulate its value
- SC tries to store back to location if and only if no one else has written to the variable since this processor’s LL
  - If SC succeeds, means all three steps happened atomically
  - If fails, doesn’t write or generate invalidations (retry LL)
  - Success indicated by condition codes

Implementation of LL/SC

- LL loads a block into register
- Records address in a special register linked register
- SC succeeds only if the address matches the address stored in linked register
- Linked register is cleared on
  - An invalidation to the linked register address
  - Context switching
- When SC fails, the store is canceled
  - Does not generate bus transaction

Simple Lock with LL-SC

```
lock:      ll     reg1, location     /* LL location to reg1 */
           bnz    reg1, lock
           sc     location, reg2     /* SC reg2 into location*/
           beqz   lock               /* if failed, start again */
           ret
unlock:    st     location, #0       /* write 0 to location */
           ret
```

Finer details

- More fancy atomic ops by changing what is between LL-SC
  - Keep it small so SC likely to succeed
  - Do not include instructions that would need to be undone (e.g., stores)
- SC can fail (without putting transaction on bus) if:
  - Detects intervening write even before trying to get bus
  - Tries to get bus but another processor’s SC gets bus first
- LL, SC are not lock, unlock respectively
  - Only guarantee no conflicting write to lock variable between them
  - Can be used directly to implement simple operations on shared variables

LL-SC Lock drawbacks

- No invalidation on failure, but read misses by all waiters after both release and successful SC by winner
- Does not reduce traffic to minimum and not a fair lock
- How to improve?
  - Fair lock
  - Only one processor to have read miss upon release and success

Drawbacks of earlier solutions

- Locks are not fair
- Lot of traffic on successful acquire and release due to cache miss effect

How to introduce fairness?

- Maintain the order in which the threads attempt to acquire lock
- On release, ensure that the first thread that asked for the lock acquires it

next-ticket now-serving

Ticket Lock

- Works like waiting line at deli or bank
- Two counters per lock: next_ticket, now_serving
- Acquire:
  fetch&inc next_ticket; wait for now_serving to equal it
- Release: increment now-serving
- Atomic operation when arrive at lock; not for release

- FIFO order: fair
- Unlike LL-SC lock, no invalidation when succeeds
- Still read misses at release; all spin on same variable

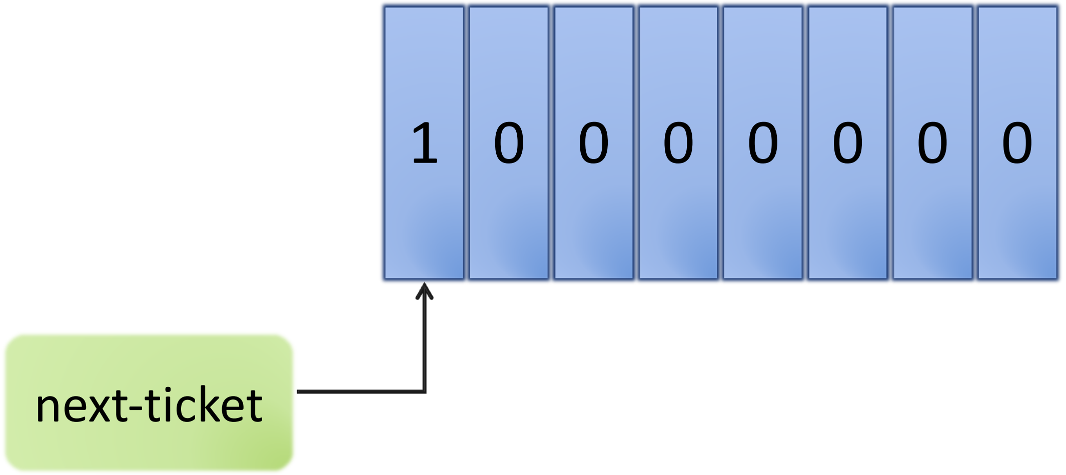
Ticket lock implementation

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

ticketLock_acquire(int *next_ticket, int *now_serving)
{
    my_ticket = fetch&inc(next_ticket);
    while (*now_serving != my_ticket) {};
}

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

How to avoid read miss on release?

- Make each process wait on a different variable rather than a single variable

- Create array of N bits when N is no of threads



Array-based Queuing Locks

- Waiting processes poll on different locations in array
- Acquire
  - fetch&inc to obtain next array element to spin on
  - Ensure array elements are in different memory blocks
- Release
  - set next location in array, waking up process spinning on it
- O(1) traffic per acquire with coherent caches
- FIFO ordering as in ticket lock
- O(p) space per lock

Array-based Queuing Lock Implementation

```
init(int *next_ticket, int *can_serve){
    *next_ticket = 0;
    for (i=1; i<MAXSIZE; i++) can_serve[i] = 0;
     can_serve[0] = 1;
}

acquire(int *next_ticket, int *can_serve){
   my_ticket = fetch&inc(next_ticket);
   while (can_serve[my_ticket] ! = 1){};
}

Release(int *can_serve){
  can_serve[my_ticket + 1] = 1;
  can_serve[my_ticket] = 0;
}
```

Point to Point Event Synchronization

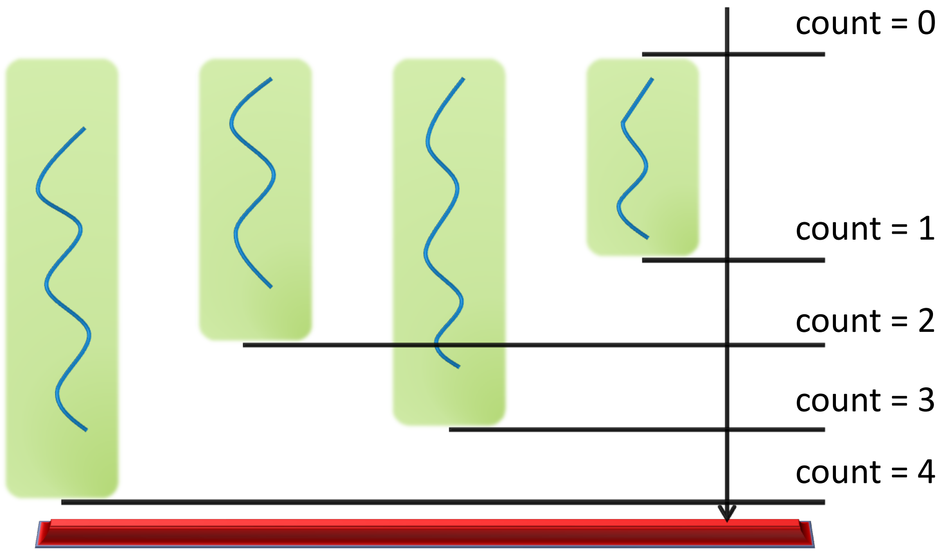
- Software methods:
  - Interrupts
  - Busy-waiting: use ordinary variables as flags

```
P1                              P2
a = f(x); /* set a */           while (flag == 0) do nothing;
flag = 1;                       b = g(a);  /* use a */
```

Shared memory code

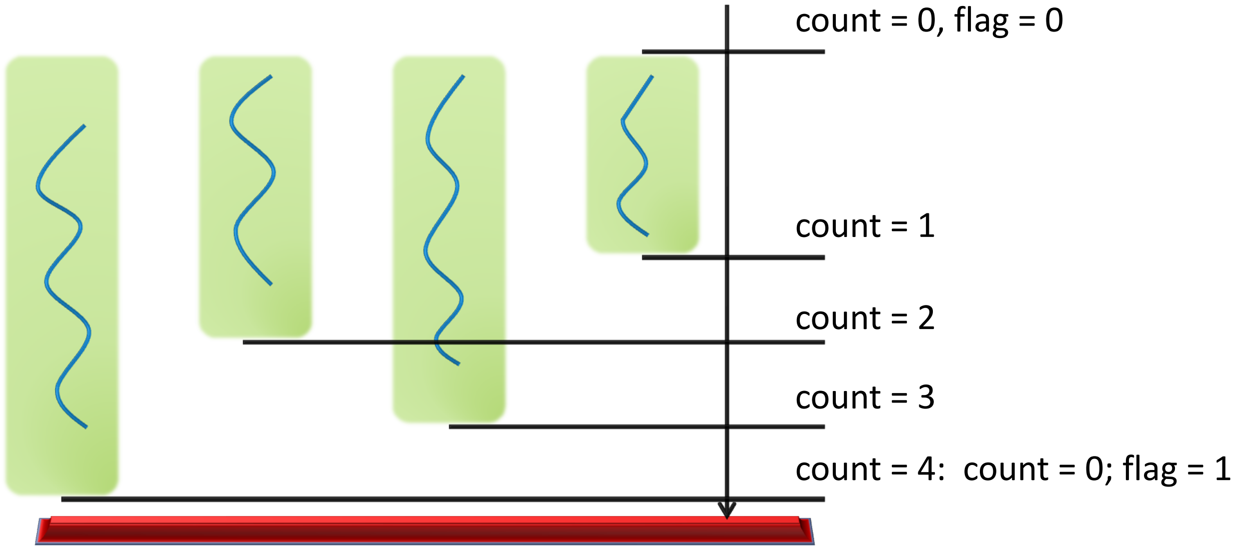
```
begin parallel    // spawn a child thread
private int start_iter, end_iter, i;
shared int local_iter = 4;
shared double sum = 0.0, a[], b[], c[];
shared lock_type mylock;

start_iter = getid() * local_iter;
end_iter = start_iter + local_iter;
for (i=start_iter; i<end_iter; i++)
    a[i] = b[i] + c[i];
barrier;

for (i=start_iter; i<end_iter; i++)
      if (a[i] > 0) {
          lock (mylock);
          sum = sum + a[i];
          unlock (mylock);
      }
barrier;
end parallel // kill the child thread
print sum;
```

The idea of barrier



- Keep count of how many threads have reached barrier
- When count = N, threads can go past barrier

Simple centralized barrier

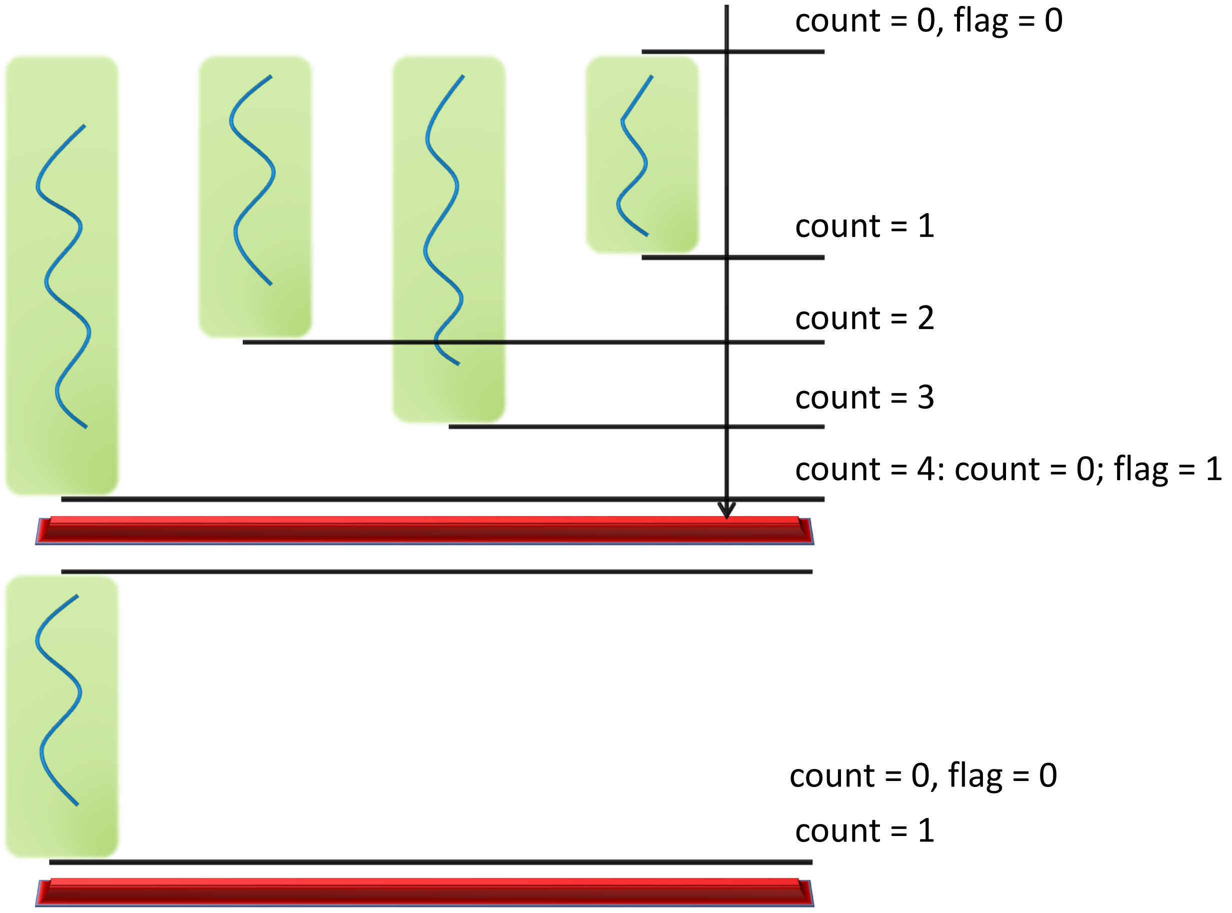


- First thread sets flag = 0
- Keep count of how many threads have reached barrier
- When count = N, set flag = 1 releasing barrier

A simple centralized barrier

```
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*/
}
```

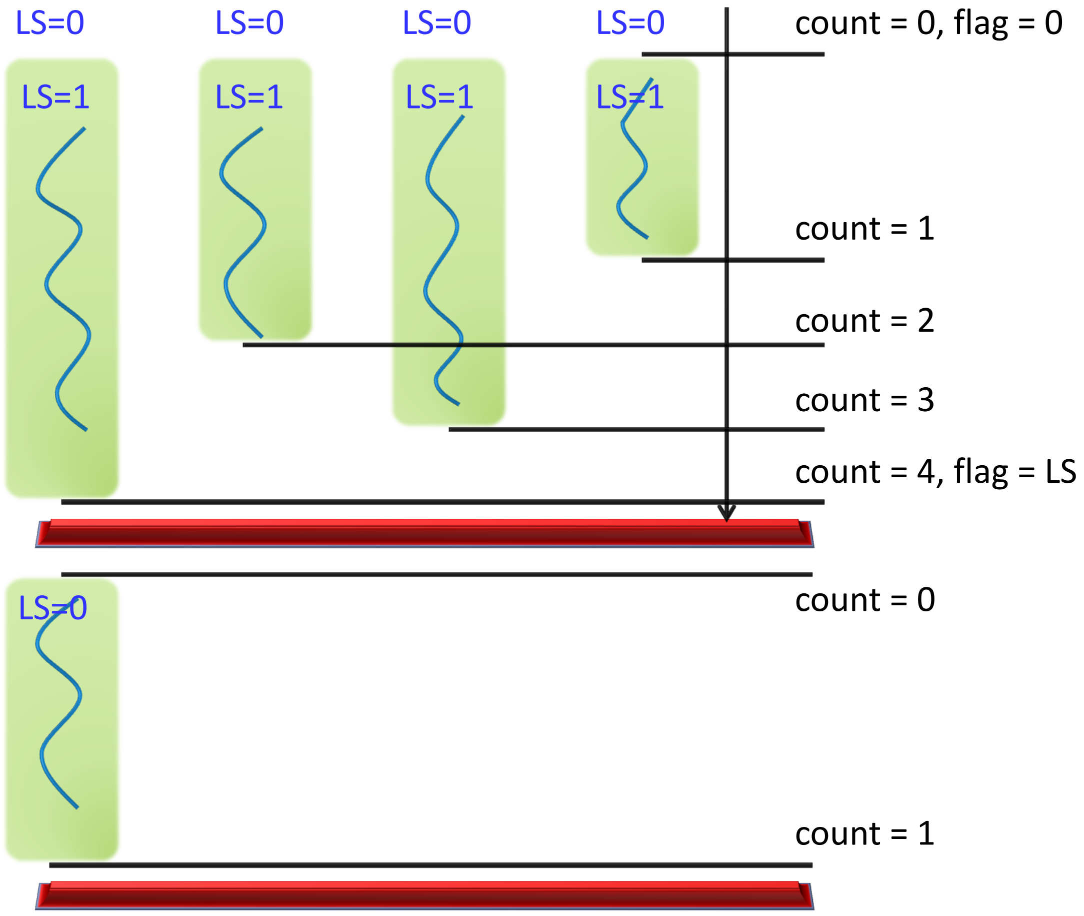
Simple centralized barrier problem



A Working Centralized Barrier

- Consecutively entering the same barrier doesn't work
  - Must prevent process from entering until all have left previous instance
- Sense reversal: wait for flag to take different value consecutive times
  - Toggle this value only when all processes reach

Sense reversal barrier



Sense-reversal Barrier

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

Summary

- Locks and barriers for synchronization
- Low-level primitives in hardware
- Sophisticated synchronization algorithms in software