Lecture 25: Thread Level Parallelism -- Synchronization ~~and Memory Consistency~~

CSCE 513 Computer Architecture

Department of Computer Science and Engineering

Yonghong Yan

yanyh@cse.sc.edu

https://passlab.github.io/CSCE513

Topics for Thread Level Parallelism (TLP)

- Parallelism (centered around …)
  - Instruction Level Parallelism
  - Data Level Parallelism
  - Thread Level Parallelism

- TLP Introduction
  - 5.1
- SMP and Snooping Cache Coherence Protocol
  - 5.2
- Distributed Shared-Memory and Directory-Based Coherence
  - 5.4
- Synchronization Basics ~~and Memory Consistency Model~~
  - 5.5, 5.6

Data Racing in a Multithread Program

Consider:

```
/* each thread to update shared variable best_cost */

if (my_cost < best_cost)
    best_cost = my_cost;
```

- two threads,
- the initial value of `best_cost` is 100,
- the values of `my_cost` are 50 and 75 for threads t1 and t2

T1	T2
`if (my_cost (50) < best_cost)` `best_cost = my_cost;`	`if (my_cost (75) < best_cost)` `best_cost = my_cost;`

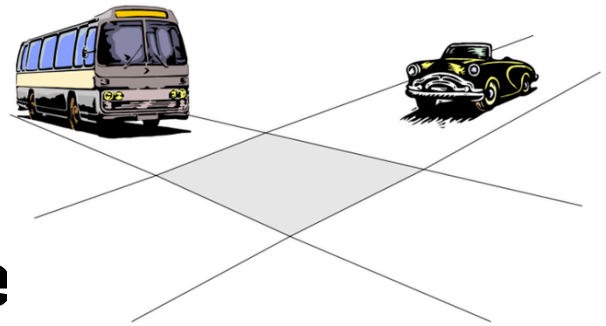
- The value of `best_cost` could be 50 or 75!
- The value 75 does not correspond to any serialization of the two threads.

3

Critical Section and Mutual Exclusion

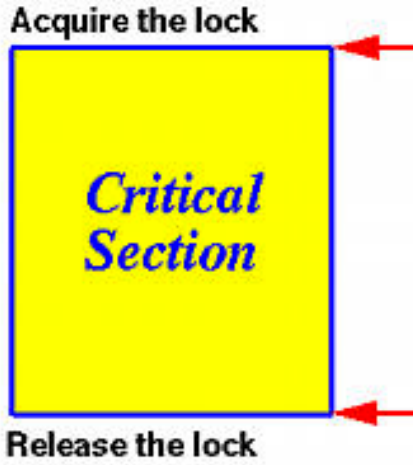
- Critical section = a segment that must be executed by only one thread at any time

```
if (my_cost < best_cost)
    best_cost = my_cost;
```



- Mutex locks protect critical se Pthreads
  - locked and unlocked
  - At any point of time, only one thread can acquire a mutex lock

- Using mutex locks
  - request lock before executing critical section
  - enter critical section when lock granted
  - release lock when leaving critical section



4

Mutual Exclusion using Pthread Mutex

```
int pthread_mutex_lock (pthread_mutex_t *mutex_lock);
int pthread_mutex_unlock (pthread_mutex_t *mutex_lock);
int pthread_mutex_init (pthread_mutex_t     *mutex_lock
    const pthread_mutexattr_t *lock_attr);
```


```
pthread_mutex_t cost_lock;
int main() {
    ...
  pthread_mutex_init(&cost_lock, NULL);
  pthread_create(&thhandle1, NULL, find_best, …);
  pthread_create(&thhandle2, NULL, find_best, …);
}
void *find_best(void *list_ptr) {
  ...
  pthread_mutex_lock(&cost_lock); // enter CS
  if (my_cost < best_cost)
    best_cost = my_cost;
  pthread_mutex_unlock(&cost_lock); // leave CS
}
```

Critical Section

pthread_mutex_lock blocks the calling thread if another thread holds the lock

When **pthread_mutex_lock** call returns

1. Mutex is locked, enter CS
2. Any other locking attempt (call to thread_mutex_lock) will cause the blocking of the calling thread

When **pthread_mutex_unlock** returns

1. Mutex is unlocked, leave CS
2. One thread who blocks on thread_mutex_lock call will acquire the lock and enter CS

Components of a Synchronization Event

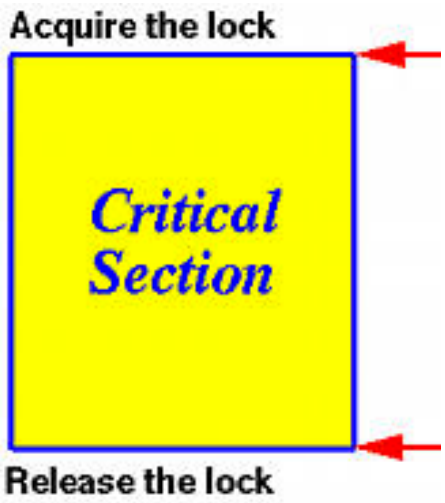
- Method for acquiring and making it exclusive
  - Acquire right to the synch
    - » enter critical section, go past event
  - Make sure no others enter CS
- Waiting algorithm
  - Wait for synch to become available when it isn't
  - busy-waiting, blocking, or hybrid
- Release method
  - Enable other processors to acquire right to the synch
- Waiting algorithm is independent of type of synchronization
  - makes no sense to put in hardware

Strawman Lock Implementation

Busy-Wait

```
lock:   ld   R1, mem[cost_lock]     /* copy cost_lock to register */
        cmp  R1, #0                 /* compare with 0 */
        bnz  lock                   /* if not 0, try again */
        st   mem[cost_lock], #1     /* store 1 to mark it exclusive */
        ret                         /* return control to caller */

unlock: st   mem[cost_lock], #0     /* write 0 to cost_lock */
        ret                         /* return control to caller */
```



Why doesn't the acquire method work?

For example: when two threads (cores) try to acquire the lock, they both execute the ld instruction when the mem[cost_lock] = 0

Atomic Instructions

Exchange data between register and memory atomatically

- **Specifies a location, register, & atomic operation**
  - Value in location read into a register
  - Another value (function of value read or not) 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
  - Other constants could be used instead of 1 and 0
- **How to implement test&set in distributed cache coherent machine?**
  - Wait until have write privileges, then perform operation without allowing any intervening operations (either locally or remotely)

Choices of Hardware Primitives for Synchronizations -- 1

- Test&Set

```
test&set (&address) {
    result = M[address];
    M[address] = 1;
    return result;
}
```

- Swap

```
swap (&address, register) { /* x86 */
    temp = M[address];
    M[address] = register;
    register = temp;
}
```

Choices of Hardware Primitives for Synchronizations -- 2

- Compare and Swap (CAS)

```
compare&swap (&address, reg1, reg2) {
  if (reg1 == M[address]) {
     M[address] = reg2;
     return success;
  } else {
     return failure;
  }
}
```

- Load-linked/reserved/locked and Store-conditional

```
load-linked&store conditional(&address) {
    loop:
      ll r1, M[address];
      movi r2, 1; … /* Can do arbitrary ops */
      sc r2, M[address];
      beqz r2, loop;
}
```

https://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Atomic-Builtins.html

Improved Hardware Primitives: LL-SC

- **Goals:**
  - Test with reads
  - Failed read-modify-write attempts don't generate invalidations
  - Nice if single primitive can implement range of r-m-w operations
- ***Load-Locked*** **(or -linked),** ***Store-Conditional***
  - LL reads variable into register
  - Follow with arbitrary instructions to manipulate its value
  - SC tries to store back to location
  - succeed if and only if no other write to the variable since this processor's LL
    - » indicated by condition codes;
- **If SC succeeds, all three steps happened "atomically"**
- **If fails, doesn't write or generate invalidations**
  - must retry to acquire

Simple Lock with LL-SC

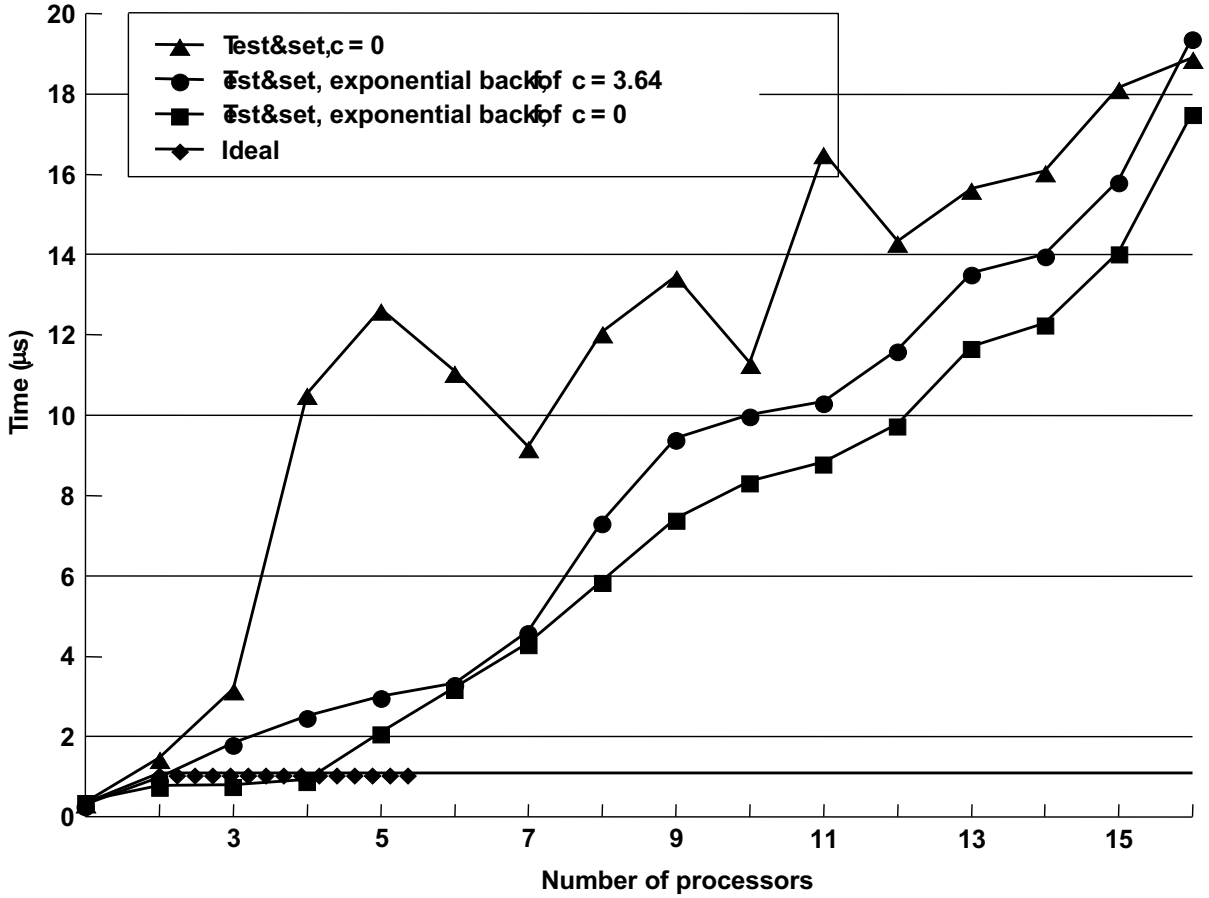
```
lock:   ll     R1, mem[cost_lock]   /* LL location to reg1 */
        sc     mem[cost_lock], R2   /* SC reg2 into mem */
        beqz   R2, lock             /* if failed, start again */
        ret

unlock: st     mem[cost_lock], #0   /* write 0 to location */
        ret
```

- Can do more fancy atomic ops by changing what's between LL & SC
  - But keep it small so SC likely to succeed
  - Don't 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
  - But can use directly to implement simple operations on shared variables

Test&Set Lock Microbenchmark: SGI Chal.

```
lock:       t&s   R1, mem[cost_location]
            bnz   lock                      /* if not 0, try again */
            ret                           /* return control to caller */
unlock:     st    mem[cost_location], #0    /* write 0 to location */
            ret                           /* return control to caller */
```




```
lock;
delay(c);
unlock;
```

Cost of Atomic and Hardware Locks

- **Expensive, e.g. X86 lock could causes multiple 100 cycles**

" `lock cmpxchg [rsp - 8], rdx` " (both with comparison match and mismatch),

" `lock xadd [rsp - 8], rdx` ",

" `lock bts qword ptr [rsp - 8], 1` "

- **Hardware atomic increment/decrement could cost multiple 10 cycles**

- **Because it may involve locking the memory bus so no others can use**

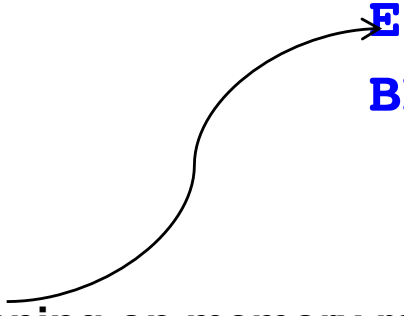
Mini-Instruction Set debate

- atomic read-modify-write instructions
  - IBM 370: included atomic compare&swap for multiprogramming
  - x86: any instruction can be prefixed with a lock modifier
  - High-level language advocates want hardware locks/barriers
    - » but it's goes against the "RISC" flow,and has other problems
  - SPARC: atomic register-memory ops (swap, compare&swap)
  - MIPS, IBM Power: no atomic operations but pair of instructions
    - » load-locked, store-conditional
    - » later used by PowerPC and DEC Alpha too
  - 68000: CCS: Compare and compare and swap
    - » No-one does this any more
- Rich set of tradeoffs

Busy-wait Lock

- Also called spin lock
  - Keep trying to acquire lock until read
  - Very low latency/processor overhead!
  - Very high system overhead!
    - » Causing stress on network while spinning
    - » Processor is not doing anything else useful

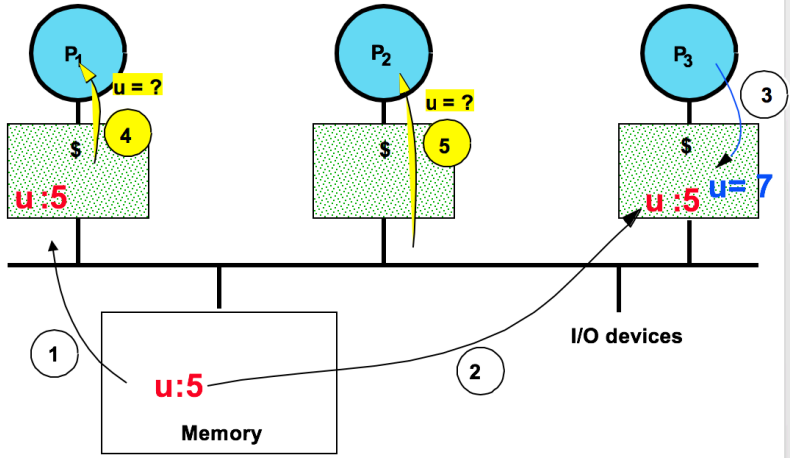
```
lockit:  DADDUI R2, R0, #1
         EXCH R2, 0(R1)     ;atomic exchange
         BNEZ R2, lockit    ;already locked?
```



Spinning on memory read

Busy-wait Lock Leveraging Cache Coherence

- Also called spin lock
  - Keep trying to acquire lock until read
  - Very low latency/processor overhead!
  - Very high system overhead!




```
lockit:  LD R2, 0(R1)      ;load of lock
         BNEZ R2, lockit   ;not available-spin
         DADDUI R2, R0, #1 ;load locked value
         EXCH R2, 0(R1)    ;swap
         BNEZ R2, lockit   ;branch if lock wasn't 0
```

Spinning on cache read until cache miss
(because of invalidation)

Busy-wait Lock Leveraging Cache Coherence

```
lockit:   LDR2,0(R1)       ;load of lock
          BNEZR2,lockit    ;not available-spin
          DADDUIR2,R0,#1   ;load locked value
          EXCHR2,0(R1)     ;swap
          BNEZR2,lockit    ;branch if lock wasn't 0
```

Step	P0	P1	P2	Coherence state of lock at end of step	Bus/directory activity
1	Has lock	Begins spin, testing if lock = 0	Begins spin, testing if lock = 0	Shared	Cache misses for P1 and P2 satisfied in either order. Lock state becomes shared.
2	Set lock to 0	(Invalidate received)	(Invalidate received)	Exclusive (P0)	Write invalidate of lock variable from P0.
3		Cache miss	Cache miss	Shared	Bus/directory services P2 cache miss; write-back from P0; state shared.
4		(Waits while bus/ directory busy)	Lock = 0 test succeeds	Shared	Cache miss for P2 satisfied
5		Lock = 0	Executes swap, gets cache miss	Shared	Cache miss for P1 satisfied
6		Executes swap, gets cache miss	Completes swap: returns 0 and sets lock = 1	Exclusive (P2)	Bus/directory services P2 cache miss; generates invalidate; lock is exclusive.
7		Swap completes and returns 1, and sets lock = 1	Enter critical section	Exclusive (P1)	Bus/directory services P1 cache miss; sends invalidate and generates write-back from P2.
8		Spins, testing if lock = 0			None

Busy-wait vs Blocking Lock

- Busy-wait: I.e. spin lock
  - Keep trying to acquire lock until read
  - Very low latency/processor overhead!
  - Very high system overhead!
    - » Causing stress on network while spinning
    - » Processor is not doing anything else useful
- Blocking:
  - If can't acquire lock, deschedule process (I.e. unload state)
  - Higher latency/processor overhead (1000s of cycles?)
    - » Takes time to unload/restart task
    - » Notification mechanism needed
  - Low system overheadd
    - » No stress on network
    - » Processor does something useful
- Hybrid:
  - Spin for a while, then block
  - 2-competitive: spin until have waited blocking time

Blocking Lock

- while (!finished()) cpu_pause();
  - Pause CPU so not consuming cycles/energy, but still occupying the CPU
  - ISA support
- while (!finished()) sched_yield();
  - Yield the CPU from the kernel, i.e. give up the slice in time-sharing
  - API/Kernel support
- mutex_wait() and mutex_wake()
  - Completely surrender the CPU for doing other work
  - API/Kernel support via pthread_cond_t and pthread_mutex_t
    - » Pthread Condition variable and mutex

Lock-Free Synchronization

- What happens if process grabs lock, then goes to sleep???
  - Page fault
  - Processor scheduling
  - Etc
- Lock-free synchronization:
  - Operations do not require mutual exclusion of multiple insts
- Nonblocking:
  - Some process will complete in a finite amount of time even if other processors halt
- Wait-Free (Herlihy):
  - Every (nonfaulting) process will complete in a finite amount of time
- Systems based on LL&SC can implement these

Synchronization Summary

- Rich interaction of hardware-software tradeoffs
- Must evaluate hardware primitives and software algorithms together
  - primitives determine which algorithms perform well
- Evaluation methodology is challenging
  - Use of delays, microbenchmarks
  - Should use both microbenchmarks and real workloads
- Simple software algorithms with common hardware primitives do well on bus
  - Will see more sophisticated techniques for distributed machines
  - Hardware support still subject of debate
- Theoretical research argues for swap or compare&swap, not fetch&op
  - Algorithms that ensure constant-time access, but complex

Class Lectures End Here!

Memory Consistency Model

- One of the most confusing topics (if not the most) in computer system, parallel programming and parallel computer architecture

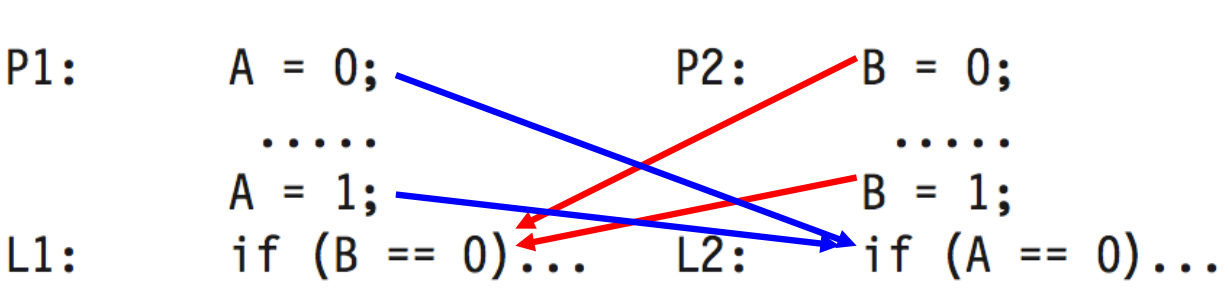
Setup for Mem. Consistency

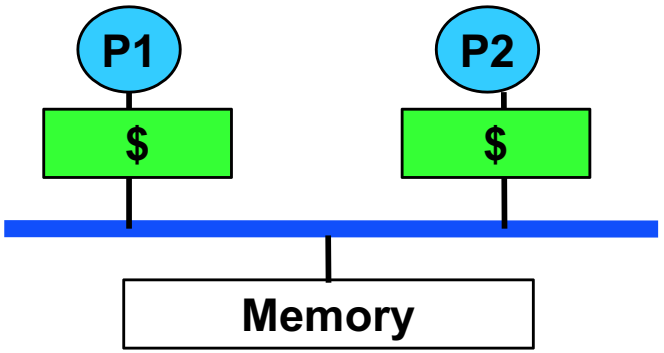
- Coherence ⇒ Writes to a location become visible to all in the same order
  - But when does a write become visible?
  - Immediately or visible when needed?

- How do we establish orders between a write and a read by different processors?
  - use event synchronization

- Typically use more than one location!

Example





- **Under cache coherence, if write is immediately available**
  - **Not possible for both L1 and L2 to be true**

- **For cache-coherent systems, if write invalidates are delayed and processor allows to continue under delay**
  - **It is possible for both L1 and L2 to be true**

Another Example of Ordering?

- What's the intuition?
  - Whatever it is, we need an ordering model for clear semantics
    - » across different locations as well
    - » so programmers can reason about what results are possible

P_1	P_2
/*Assume initial values of A and B are 0 */	
`(1a) A = 1;`	`(2a) print B;`
`(1b) B = 2;`	`(2b) print A;`

- Expect memory to respect order between accesses to *different* locations issued by a given process
  - to preserve orders among accesses to same location by different processes
- Coherence is not enough!
  - pertains only to single location

Memory Consistency Model

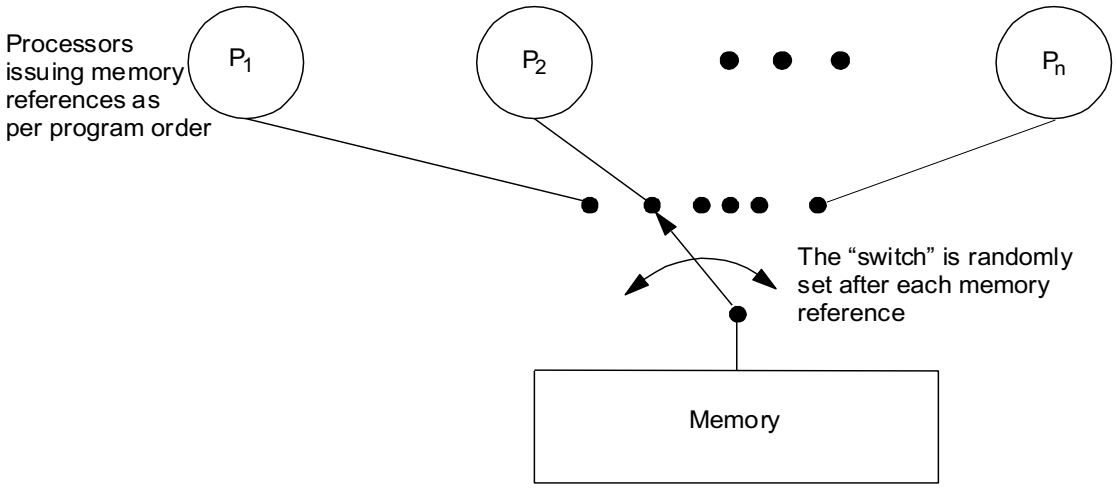
- Specifies constraints on the order in which memory operations (from any process) can appear to execute with respect to one another
  - What orders are preserved?
  - Given a load, constrains the possible values returned by it
- Implications for both programmer and system designer
  - Programmer uses to reason about correctness and possible results
  - System designer can use to constrain how much accesses can be reordered by compiler or hardware
- Contract between programmer and system

Sequential Consistency

- Memory operations from a proc become visible (to itself and others) in program order
- There exists a total order, consistent with this partial order - i.e., an interleaving
  - the position at which a write occurs in the hypothetical total order should be the same with respect to all processors
- Said another way:
  - For any possible individual run of a program on multiple processors
  - Should be able to come up with a serial interleaving of all operations that respects
    - » Program Order
    - » Read-after-write orderings (locally and through network)
    - » Also Write-after-read, write-after-write

Sequential Consistency

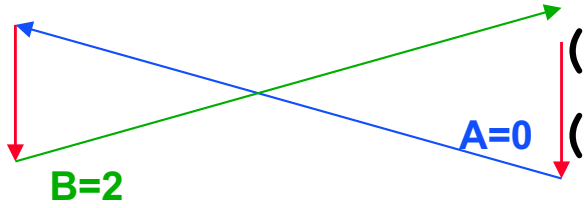
- **Total order achieved by *interleaving* accesses from different processes**
  - **Maintains *program order*, and memory operations, from all processes, appear to [issue, execute, complete] atomically w.r.t. others**
  - **as if there were no caches, and a single memory**
- "A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."
  [Lamport, 1979]



SC Example

- What matters is order in which operations *appear to execute,* not the chronological order of events
- Possible outcomes for (A,B): (0,0), (1,0), (1,2)
- What about (0,2) ?
  - program order ⇒ 1a->1b and 2a->2b
  - A = 0 implies 2b->1a, which implies 2a->1b
  - B = 2 implies 1b->2a, which leads to a contradiction (cycle!)
- Since there is a cycle⇒no sequential order that is consistent!

P_1	P_2
/*Assume initial values of A and B are 0*/	
(1a) A = 1;	(2a) print B;
(1b) B = 2;	(2b) print A;



Implementing SC

- Two kinds of requirements
  - Program order
    - » memory operations issued by a process must appear to execute (become visible to others and itself) in program order
  - Atomicity
    - » in the overall hypothetical total order, one memory operation should appear to complete with respect to all processes before the next one is issued
    - » guarantees that total order is consistent across processes
  - tricky part is making writes atomic
- How can compilers violate SC?
  - Architectural enhancements?

Sequential Consistency

- Bus imposes total order on xactions for all locations
- Between xactions, procs perform reads/writes (locally) in program order
- So any execution defines a natural partial order
  - M_j subsequent to M_i if
    - » (i) M_j follows M_i in program order on same processor,
    - » (ii) M_j generates bus xaction that follows the memory operation for M_i
- In segment between two bus transactions, any interleaving of local program orders leads to consistent total order
- Within segment writes observed by proc P serialized as:
  - Writes from other processors by the previous bus xaction P issued
  - Writes from P by program order
  - Insight: only one cache may have value in "M" state at a time…

Sufficient conditions

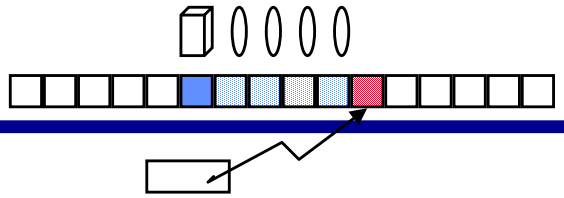
- **Sufficient Conditions**
  - issued in program order
  - after write issues, the issuing process waits for the write to complete before issuing next memory operation
  - after read is issues, the issuing process waits for the read to complete and for the write whose value is being returned to complete (globally) before issuing its next operation
- **Write completion**
  - can detect when write appears on bus (flush) appears
- **Write atomicity:**
  - if a read returns the value of a write, that write has become visible to all others already
    - » Either: it is being read by the processor that wrote it and no other processor has a copy (thus any read by any other processor will get new value via a flush
    - » Or: it has already been flushed back to memory and all processors will have the value

More on Synchronizations

Enhancements to Simple Lock

- Reduce frequency of issuing test&sets while waiting
  - *Test&set lock with backoff*
  - Don't back off too much or will be backed off when lock becomes free
  - Exponential backoff works quite well empirically: i^{th} time = $k*c^i$
- 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 (to 0), try to obtain lock with test&set
    - » only one attemptor will succeed; others will fail and start testing again

Ticket Lock



- Only one r-m-w per acquire
- Two counters per lock (next_ticket, now_serving)
  - **Acquire:** fetch&inc next_ticket;
    wait for now_serving == next_ticket
    - » atomic op when arrive at lock, not when it's free (so less contention)
  - Release: increment now-serving
- Performance
  - low latency for low-contention - if fetch&inc cacheable
  - O(p) read misses at release, since all spin on same variable
  - FIFO order
    - » like simple LL-SC lock, but no inval when SC succeeds, and fair
  - Backoff?
- Wouldn't it be nice to poll different locations ...

Array-based Queuing Locks

- Waiting processes poll on different locations in an array of size *p*
  - Acquire
    - » fetch&inc to obtain address on which to spin (next array element)
    - » ensure that these addresses are in different cache lines or memories
  - 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
  - Not so great for non-cache-coherent machines with distributed memory
    - » array location I spin on not necessarily in my local memory
- Example: MCS lock (Mellor-Crummey and Scott)

Point to Point Event Synchronization

- Software methods:
  - Interrupts
  - Busy-waiting: use ordinary variables as flags
  - Blocking: use semaphores
- Full hardware support: *full-empty bit* with each word in memory
  - Set when word is "full" with newly produced data (i.e. when written)
  - Unset when word is "empty" due to being consumed (i.e. when read)
  - Natural for word-level producer-consumer synchronization
    - » producer: write if empty, set to full; consumer: read if full; set to empty
  - Hardware preserves atomicity of bit manipulation with read or write
  - Problem: flexibility
    - » multiple consumers, or multiple writes before consumer reads?
    - » needs language support to specify when to use
    - » composite data structures?

Barriers

- Software algorithms implemented using locks, flags, counters
- 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

A Simple Centralized Barrier

- Shared counter maintains number of processes that have arrived
  - increment when arrive (lock), check until reaches numprocs
  - Problem?

```
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 */
}
```

A Working Centralized Barrier

- Consecutively entering the same barrier doesn't work
  - Must prevent process from entering until all have left previous instance
  - Could use another counter, but increases latency and contention
- Sense reversal: wait for flag to take different value consecutive times
  - Toggle this value only when all processes reach

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

Centralized Barrier Performance

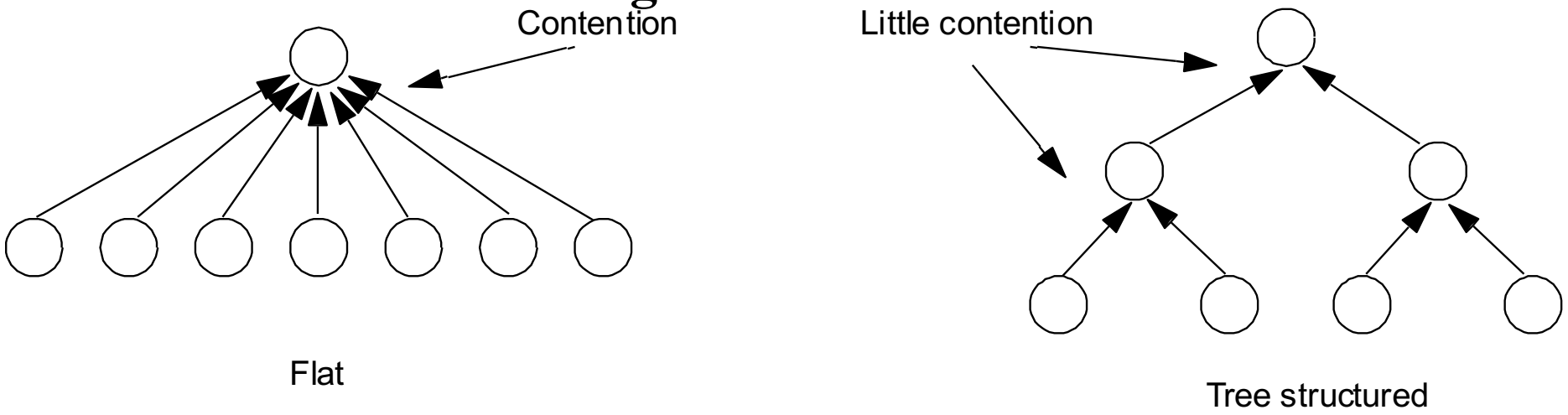
- Latency
  - Centralized has critical path length at least proportional to *p*
- Traffic
  - About $3p$ bus transactions
- Storage Cost
  - Very low: centralized counter and flag
- Fairness
  - Same processor should not always be last to exit barrier
  - No such bias in centralized
- Key problems for centralized barrier are latency and traffic
  - Especially with distributed memory, traffic goes to same node

Improved Barrier Algorithms for a Bus

- Separate arrival and exit trees, and use sense reversal
- Valuable in distributed network: communicate along different paths
- On bus, all traffic goes on same bus, and no less total traffic
- Higher latency (*log p* steps of work, and *O(p)* serialized bus xactions)
- Advantage on bus is use of ordinary reads/writes instead of locks

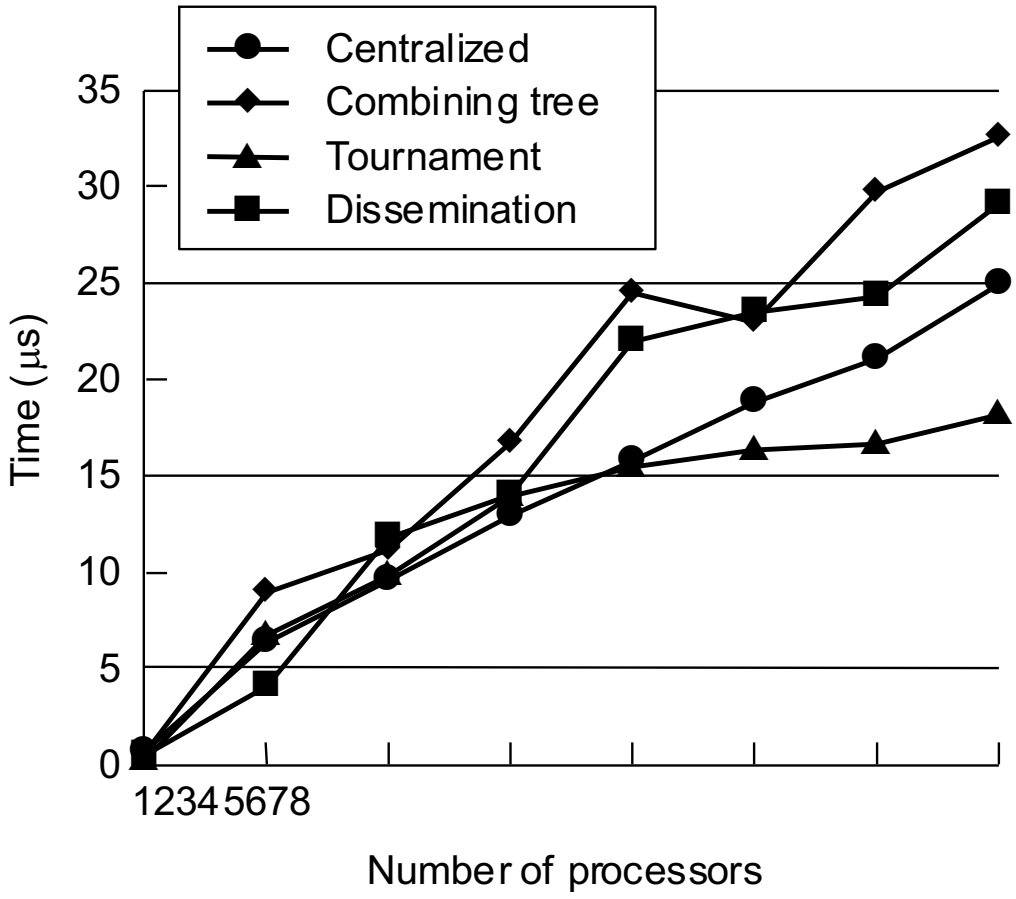
Software combining tree

- **Only *k* processors access the same location, where *k* is degree of tree**



Barrier Performance on SGI Challenge

- Centralized does quite well
  - » Will discuss fancier barrier algorithms for distributed machines
- Helpful hardware support: piggybacking of reads misses on bus
  - » Also for spinning on highly contended locks



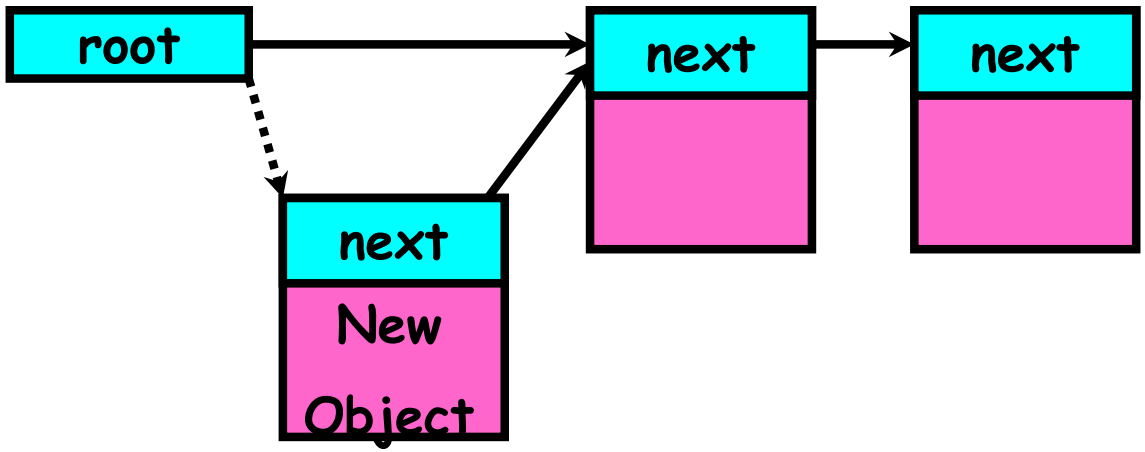
Using of Compare&Swap for queues

```
compare&swap (&address, reg1, reg2) { /* 68000 */
    if (reg1 == M[address]) {
       M[address] = reg2;
       return success;
    } else {
       return failure;
    }
}
```

Here is an atomic add to linked-list function:

```
addToQueue(&object) {
    do {                          // repeat until no conflict

       ld r1, M[root]             // Get ptr to current head
       st r1, M[object]           // Save link in new object
    } until (compare&swap(&root,r1,object));
}
```



Transactional Memory

- Transaction-based model of memory
  - Interface:
    start transaction();
    read/write data
    commit transaction():
  - If conflicts detected, commit will abort and must be retried
  - What is a conflict?
    - » If values you read are written by others before commit
- Hardware support for transactions
  - Typically uses cache coherence protocol to help process

Brief discussion of Transactional Memory

- LogTM: Log-based Transactional Memory
  - Kevin Moore, Jayaram Bobba, Michelle Moravan, Mark Hill & David Wood
  - Use of Cache Coherence protocol to detect transaction conflicts
- Transactional Interface:
  - `begin_transaction()`: Request that subsequent statements for a transaction
  - `commit_transaction()`: Ends successful transaction begun by matching `begin_transaction()`. Discards any transaction state saved for potential abort
  - `abort_transaction()`: Transfers control to a previously register conflict handler which should undo and discard work since last `begin_transaction()`

```
for(i=0; i<10000; ++i){
   begin_transaction();
     new_total = total.count + 1;
     private_data[id].count++;
     total.count = new_total;
   commit_transaction();
   think();
}
```

Figure 4. Shared-counter microbenchmark (main loop)

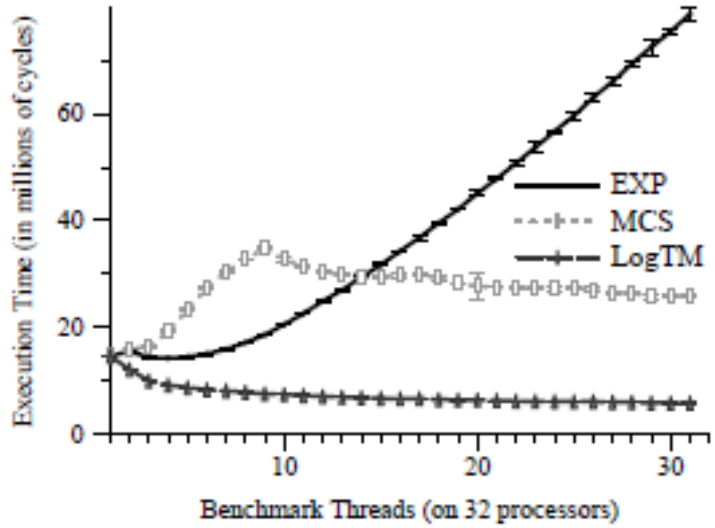


Figure 5. Execution time for LogTM transactions, test-and-test-and-set locks with exponential backoff (EXP) and MCS locks (MCS).