

ECE 454

Computer Systems Programming

Better Locking

Jon Eyolfson
Courtesy: Ashvin Goel
ECE Dept, University of Toronto

With thanks to Angela Demke Brown, Tom Hart, Paul McKenney

Overview

- Overview of locking implementations
- Spinlocks
- Cost of locking
- Ticket locks
- Queuing locks
- MCS locks
- Some parallel programming techniques

Review

- Processes communicate and coordinate via IPC
 - Pipes, sockets, signals, etc.
- Threads communicate and coordinate via memory
 - Requires mutual exclusion to prevent data races, inconsistencies
 - Use locks
 - Requires synchronization to enforce ordering
 - Use barriers, condition variables, semaphores

Questions

- How are locks implemented?
- What is the cost of locking?
- How can we develop more efficient locks?

Uniprocessor Locking Solutions

- Within kernel:
 - When data is shared between multiple threads
 - Disallow context switches in critical sections
 - When data is shared between threads and interrupt handlers
 - Disable interrupts and disallow context switches in critical sections
- At user level:
 - Use blocking locks
 - Implemented by the kernel using the mechanisms described above
- Works because there is no true parallelism

Multiprocessor Locking Solutions

- True concurrency, i.e., parallelism – code executes simultaneously on multiple CPUs
 - Disabling interrupts only affects local CPU
 - Disallowing context switch doesn't help since multiple threads are executing anyway
- Need some help from hardware
 - Hardware provides special atomic instructions such as atomic test_and_set (TAS), compare_and_swap (CAS), etc.
 - Atomic operations performed using these instructions directly
 - E.g. set/increment/decrement variable
 - Mutual exclusion for multiple instructions requires locking
 - Use atomic instructions to implement spinlocks

Atomic Instructions

test_and_set (TAS):

```
boolean TAS(boolean *lock)
{ /* pseudocode: atomic inst. */
  boolean old = *lock;
  *lock = TRUE;
  return old;
}
```

compare_and_swap (CAS):

```
int CAS(int *p, int old, int new)
{ /* pseudocode: atomic inst. */
  if (*p == old) {
    *p = new;
    return TRUE;
  } else {
    return FALSE;
  }
}
```

fetch_and_increment:

```
int fetch_and_increment(int *value)
{ /* pseudocode: atomic inst. */
  int old = *value;
  *value = *value + 1;
  return old;
}
```

fetch_and_store (FAS):

```
int FAS(int *p, int value)
{ /* pseudocode: atomic inst. */
  int old = *p;
  *p = value;
  return old;
}
```

Spinlocks

- Loop, testing lock variable until available
- When to use it (vs blocking locks)?
 - Good if nothing else to do
 - Or if expected wait is short
 - < 2 context switches
 - Or if you aren't allowed to block
 - E.g., interrupt handler
- Rest of the slides focus on improving spinlock performance

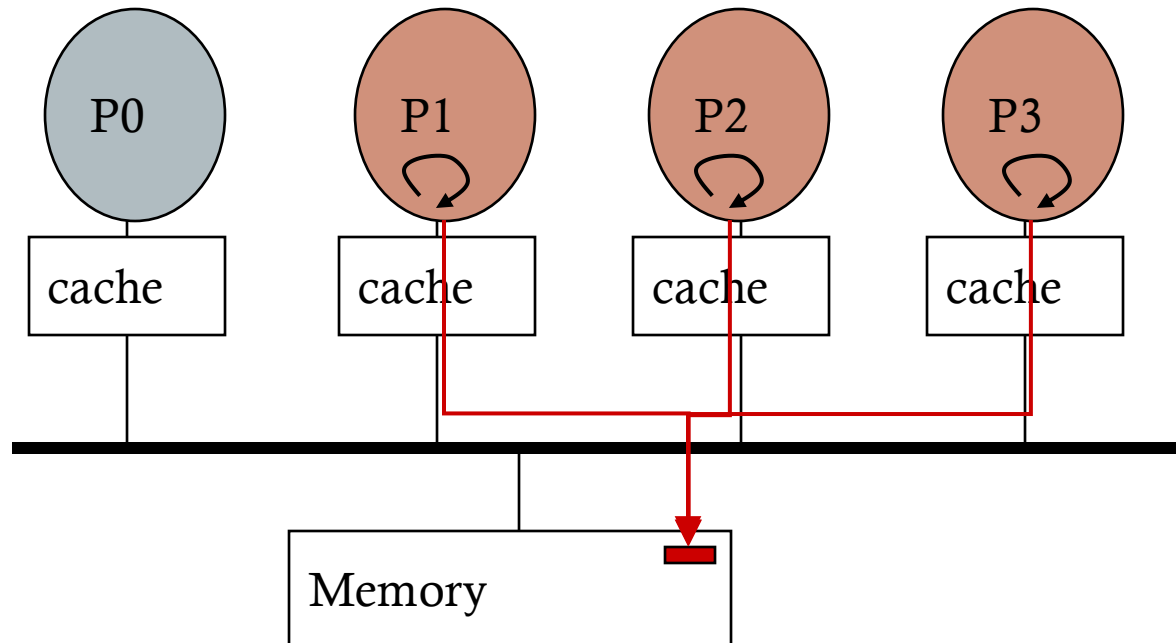
```
boolean lock;  
  
void acquire(boolean *lock) {  
    while(TAS(lock));  
}  
  
void release(boolean *lock) {  
    *lock = false;  
}
```


Cost of Locking

- TAS(lock) operates on memory location atomically
- Hardware implementation
 - Read-exclusive (invalidations)
 - Modify (change state)
 - Memory barrier (ensures that reads/writes before/after atomic instruction are not reordered)
 - complete all the mem. op. before this TAS
 - cancel all the mem. op. after this TAS
- Read-exclusive broadcasts invalidations to all caches
- Modify marks local cache dirty
- With contention, cache line ping pongs with each TAS operation!

Cost of Locking

- Leads to significant cache traffic, contention on memory bus
 - Slows down other memory operations as well



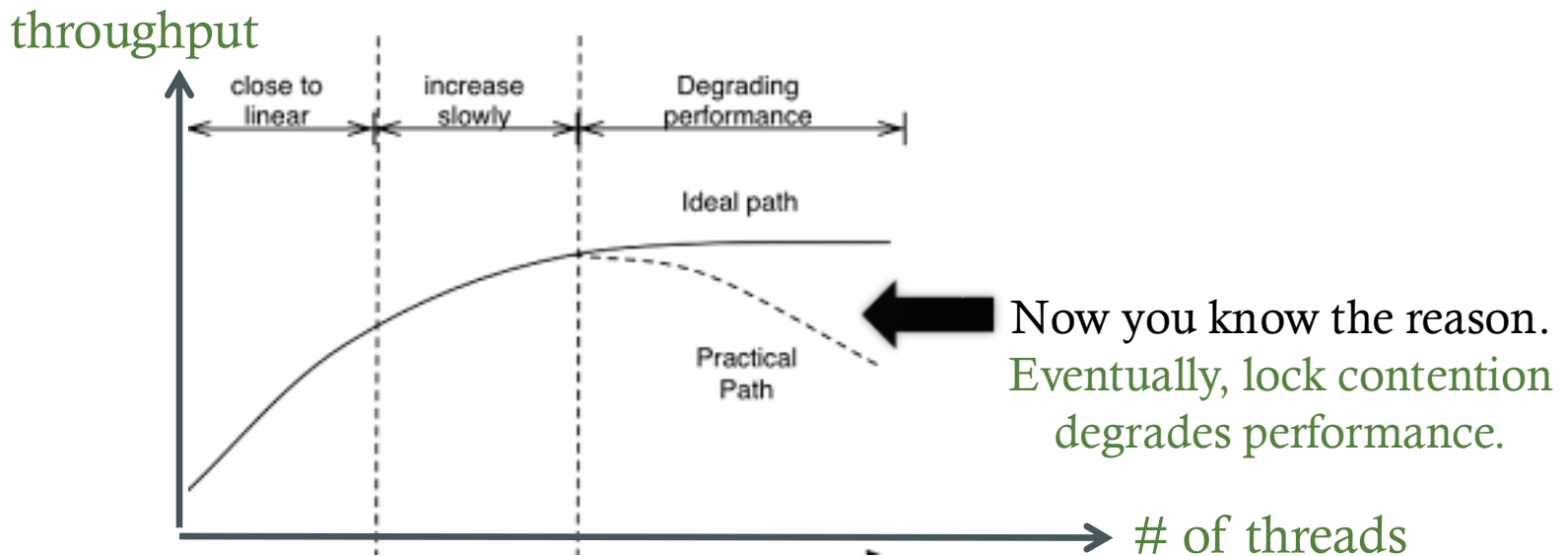
How Bad is it?

System		Opteron			Xeon				Opteron	Xeon
State	Hops	same die	one hop	two hops	same die	one hop	two hops	L1	3	5
	TAS		Modified	110		216	296	120	324	430
Store	Modified	83	191	273	115	320	431	LLC	40	44
								RAM	136	355

- Recall: TAS essentially is a Store + Memory Barrier
- Takeaway: heavy lock contention may lead to worse performance than serial execution that accesses local cache

Big Picture

- We know that we need parallelization
- But will more parallelization always yield better performance?



Building Better Spinlocks

But how?

Spinlock with Backoff

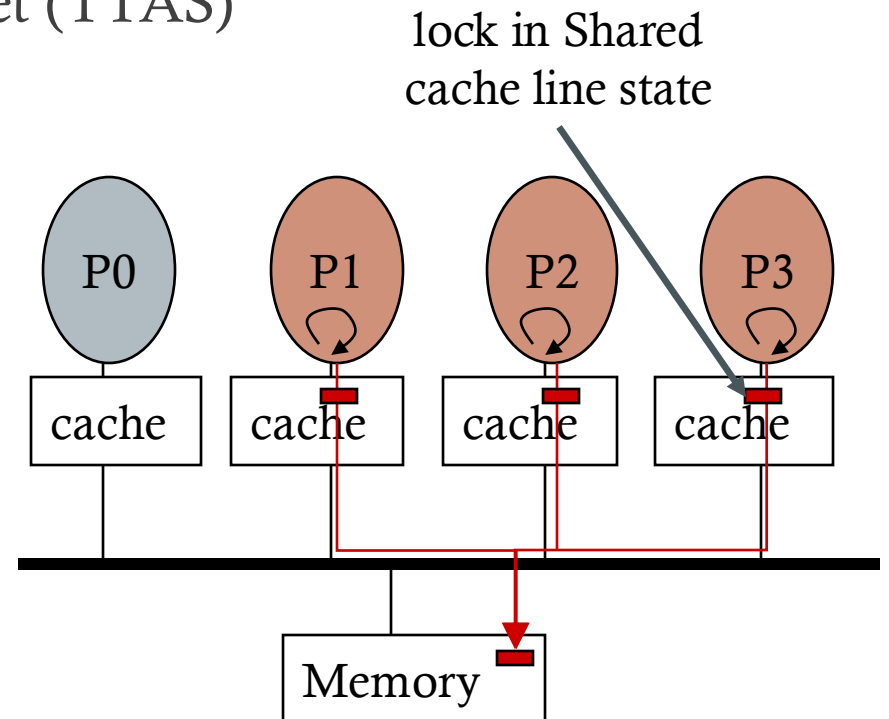
- Idea: if lock is held, wait awhile before probing again
 - Best performance uses exponential backoff
 - Can cause fairness problems – why?

```
void acquire(boolean *lock) {
    int delay = 1; // not shared by threads
    while(TAS(lock)) {
        pause(delay);
        delay = delay * 2;
    }
}
```

TTAS Spinlock

- Idea: spin in cache, access memory only when lock is likely to be available
 - Known as test_and_test_and_set (TTAS)

```
boolean lock;  
  
void acquire(boolean *lock) {  
    do {  
        while(*lock == TRUE);  
    } while (TAS(lock));  
}  
  
void release(boolean *lock) {  
    *lock = false;  
}
```



Ticket Locks

- Resolve fairness issues with previous spinlocks (FIFO order)
- Lock consists of two counters: `next_ticket`, `now_serving`

```
struct lock {  
    int next_ticket = 0;  
    int now_serving = 0;  
};
```

atomically increments `next_ticket`,
returns old value of `next_ticket`



```
void acquire(struct lock *l) {  
    int my_ticket = fetch_and_increment(&l->next_ticket);  
    while(l->now_serving != my_ticket) ; //spin, only reads performed  
}  
void release(struct lock *l) {  
    l->now_serving++; // why not atomic?  
}
```


Ticket Locks

- Reduces # of atomic operations compared to TTAS locks
- Problems? How can we mitigate them?

```
struct lock {  
    int next_ticket = 0;  
    int now_serving = 0;  
};
```

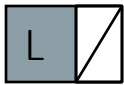
atomically increments next_ticket,
returns old value of next_ticket



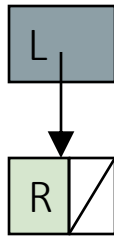
```
void acquire(struct lock *l) {  
    int my_ticket = fetch_and_increment(&l->next_ticket);  
    while (l->now_serving != my_ticket) ; //spin  
}  
void release(struct lock *l) {  
    l->now_serving++;  
}
```

Queuing Locks

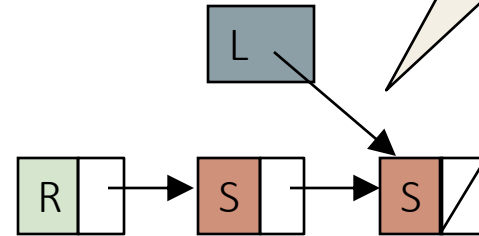
- Idea: Each CPU spins on a different location
 - Release unblocks next waiter only
 - Guarantees FIFO ordering (similar to ticket locks)
 - Reduces cache coherence traffic, memory contention, why?
- Lock L points to tail of list
 - Lock acquire: add node for processor to tail of list
 - Lock release: unblocks next node in list



(a) Free lock
(null pointer)



(b) Held lock
no waiters

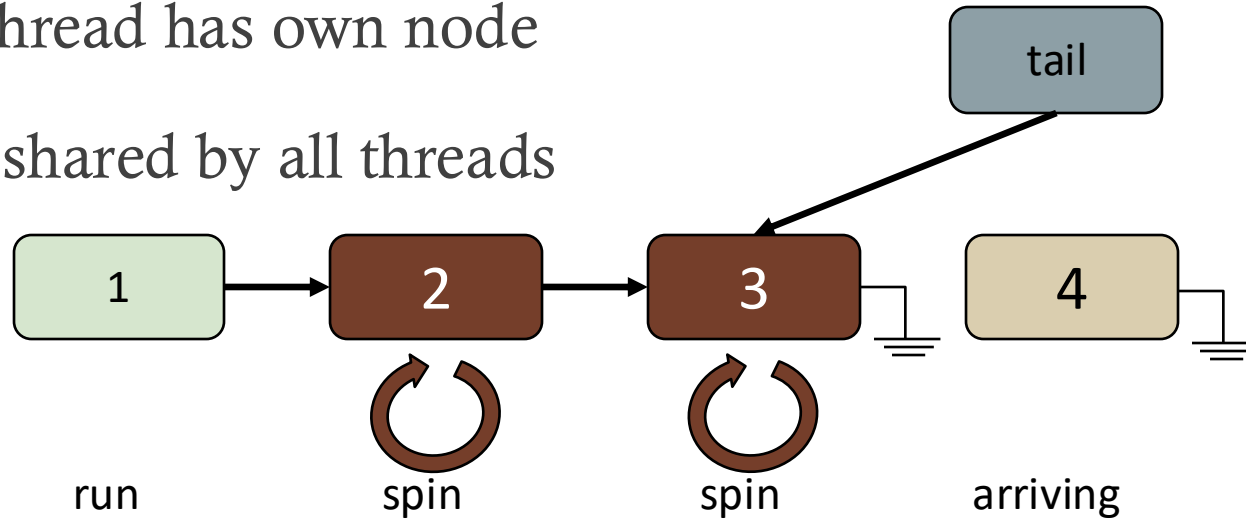


(c) Held lock
2 waiters spinning

L = lock
R = running
S = spinning

MCS Lock Operations

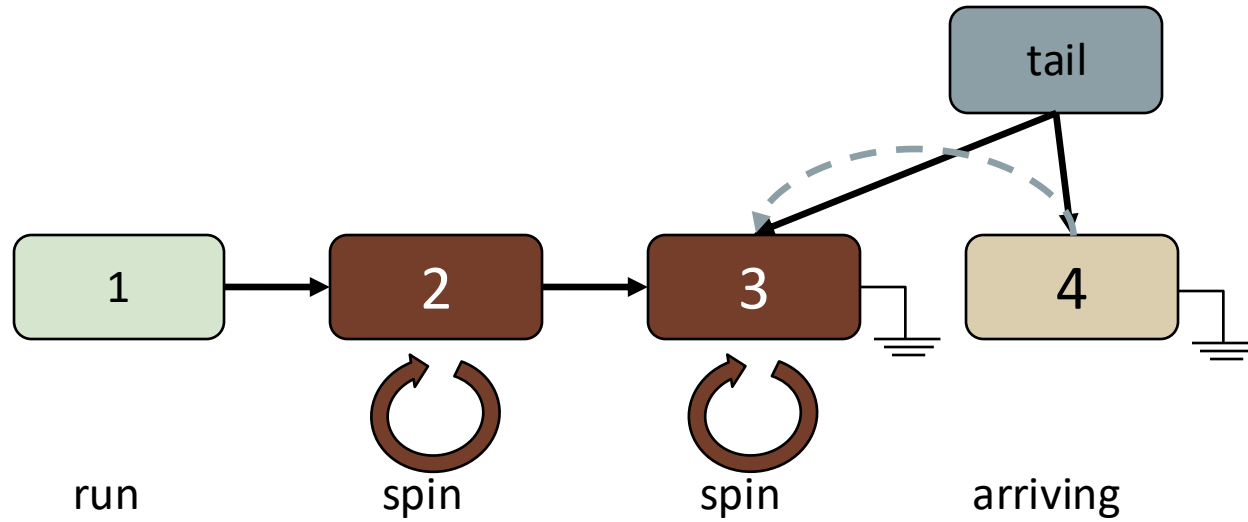
- Each thread has own node
- Tail is shared by all threads



- 4 arrives, attempting to acquire lock

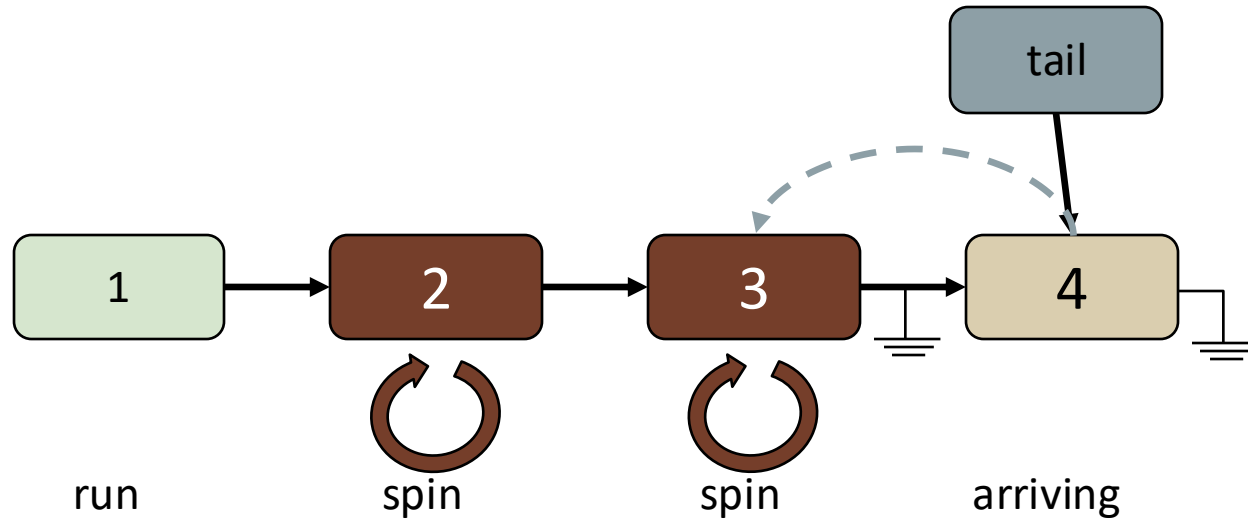
Diagrams: Redrawn from originals ©Bill Scherer – Rice University

MCS Lock Operations: acquire()



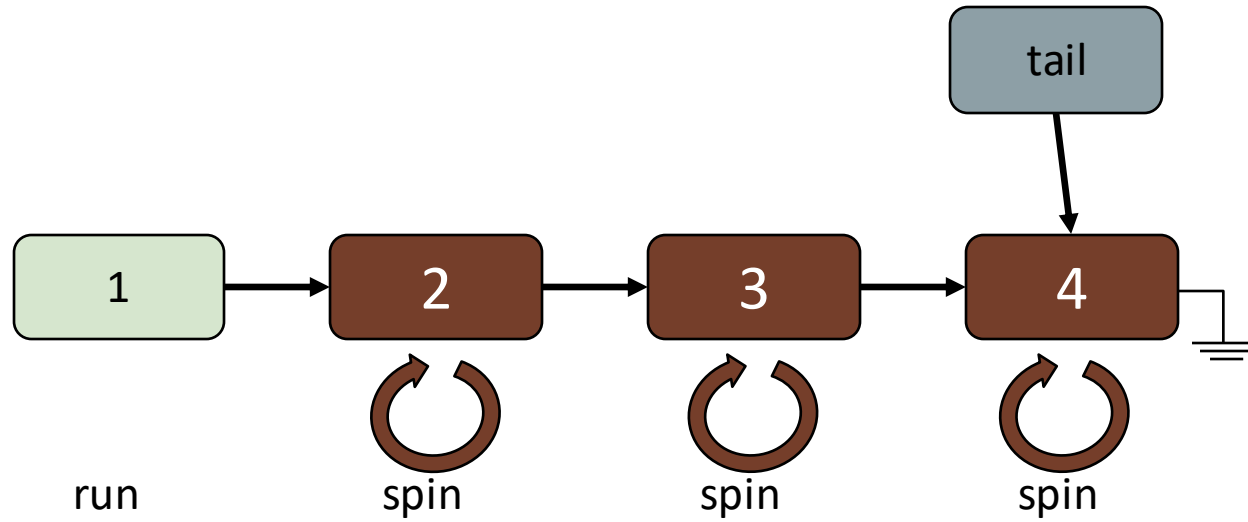
- 4 swaps tail pointer to point to own node
- Acquires pointer to 3 (predecessor) from swap on tail
- **Note:** 3 can't leave immediately because tail no longer points to 3

MCS Lock Operations: acquire()



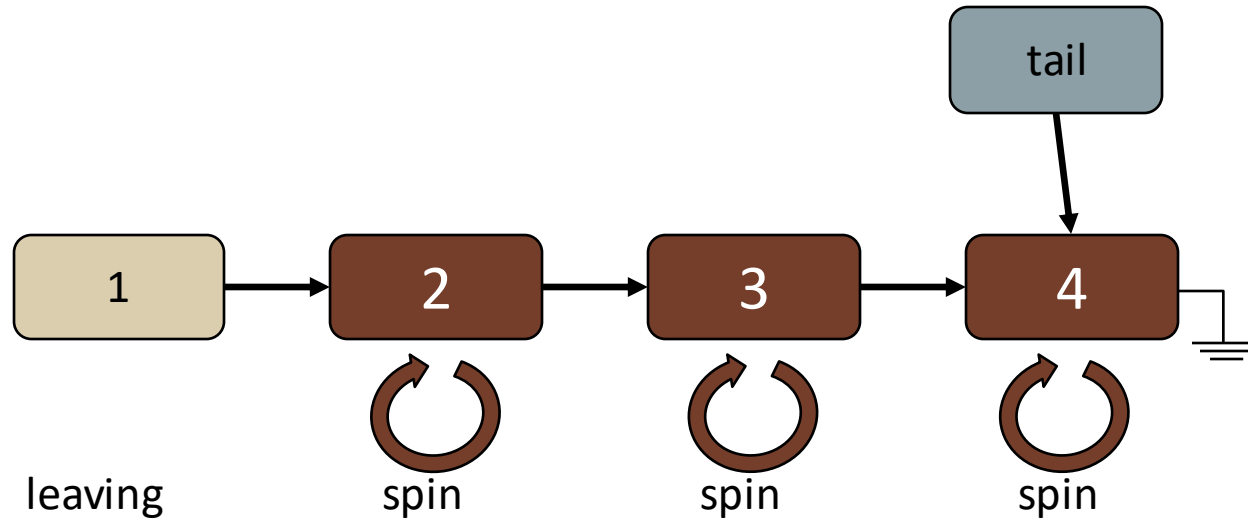
- 4 links behind 3 (predecessor)

MCS Lock Operations: acquire()



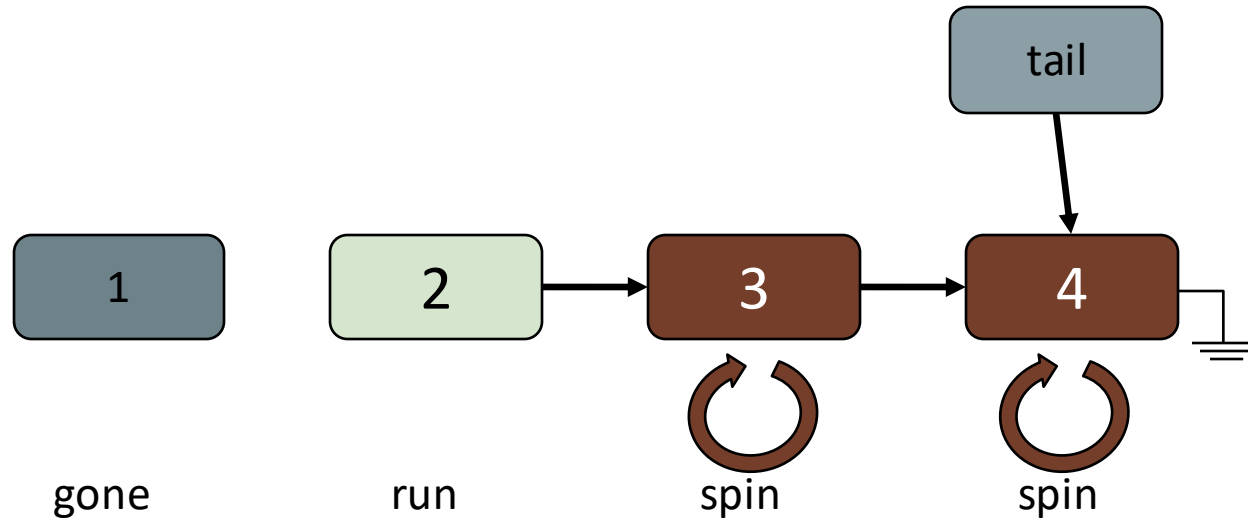
- 4 now spins until 3 signals that the lock is available by setting a flag in 4's node

MCS Lock Operations: release()



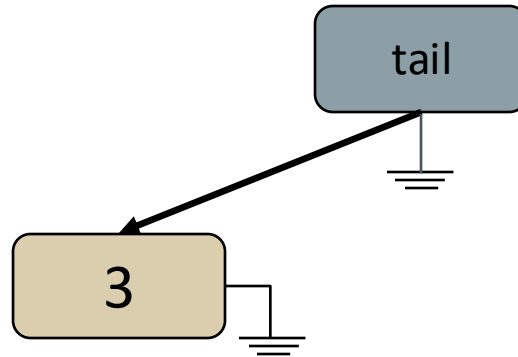
- 1 prepares to release lock
 - Its next field is set (in this diagram), so signal successor directly

MCS Lock Operations: release()



- 2 can now run, holds the lock
- 2 will signal 3 in turn, when it is done with lock
- No other process sees that lock holder has changed

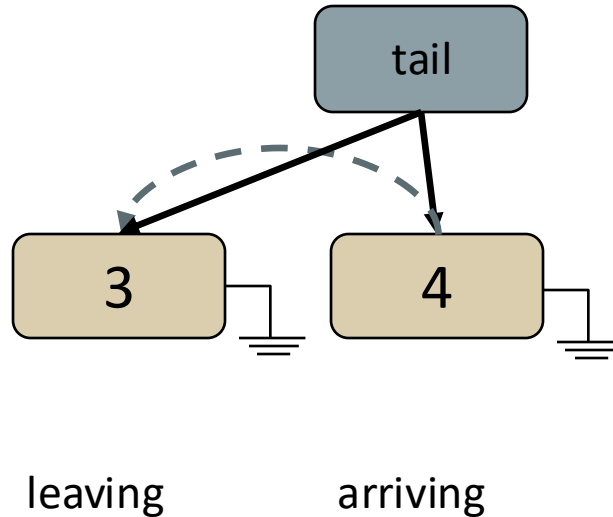
MCS Lock Operations: release()



leaving

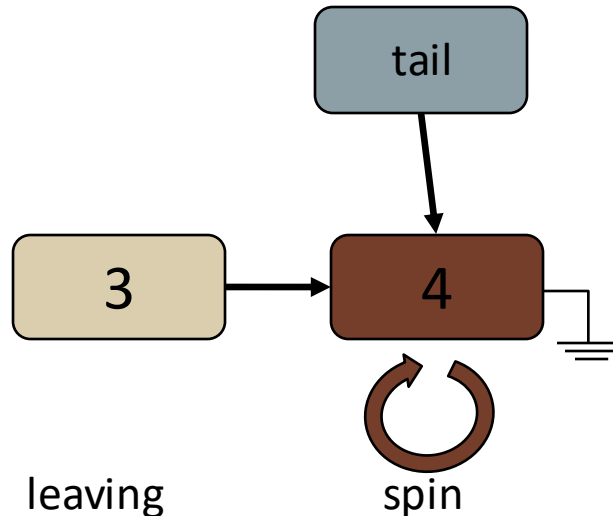
- Suppose 3 has lock, and it is the only process in the queue
- When it leaves, it sets the tail to NULL

MCS Lock Operations: release()



- Suppose 3 (last process) has lock and is leaving as Process 4 arrives
- Process 4 sets tail to itself, but Process 3's next pointer is still NULL
- Process 3 attempts `compare_and_swap` on tail pointer to set it to NULL, but finds that tail no longer points to self

MCS Lock Operations: release()



- Suppose 3 (last process) has lock and is leaving as Process 4 arrives
- Process 4 sets tail to itself, but Process 3's next pointer is still NULL
- Process 3 attempts `compare_and_swap` on tail pointer to set it to NULL, but finds that tail no longer points to self
- 3 now waits until its successor pointer is valid; 3 signals 4

MCS Lock Pseudocode

- Shared variable “tail” is a pointer to last qnode in list
 - i.e. “tail” stores address of last qnode
 - Need to pass address of tail to modify tail pointer itself

```
struct qnode {
    int locked;           // lock flag
    struct qnode *next;  // next node in linked list
}
void acquire(struct qnode **tail, struct qnode *my_node) {
    my_node->next = NULL;
    // atomically retrieve old tail, and make tail point to my_node
    struct qnode *pred = fetch_and_store(tail, my_node);
    if (pred != NULL) { // queue not empty
        my_node->locked = TRUE; // initialize to locked
        pred->next = my_node; // append my_node to queue
        while (my_node->locked); //spin until pred sets locked to FALSE
    }
}
```

Example: Simultaneous Acquire

Initial: tail == NULL

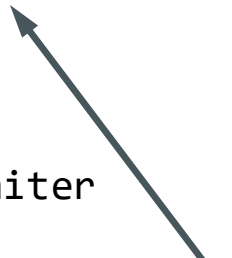
```
T0: my_node->next = NULL;  
T0: pred = FAS(tail, my_node);
```

```
T1: my_node->next = NULL;  
T1: pred = FAS(tail, my_node);
```

- fetch_and_store (FAS) executes atomically in some order...
 - Either T₀'s FAS operation completes first, or T₁'s does
- Suppose T₀ first:
 - For T₀, old value of tail is NULL, so pred == NULL
 - Tail is set to point at T₀'s qnode
 - T₀ acquires the lock
 - For T₁, old value of tail (pred) is T₀'s qnode
 - T₁ spins on its qnode's locked value
- Note: No additions are lost, but queue may not be fully linked together until all threads complete pred->next update

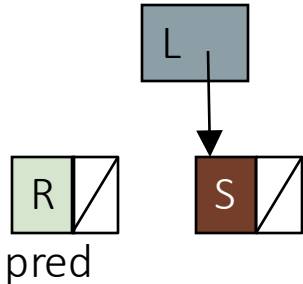
MCS Lock Release

```
struct qnode {
    int locked;
    struct qnode *next;
}
void release(struct qnode **tail, struct qnode *my_node) {
    if (my_node->next == NULL) {
        // no known successor, check if tail still points to me
        if (compare_and_swap(tail, my_node, NULL))
            return; // CAS returns TRUE iff tail updated to NULL
        // CAS fails if someone else is adding themselves to the list
        // wait for them to finish
        while (my_node->next == NULL) ; //spin
    }
    my_node->next->locked = FALSE; // release next waiter
}
```



Release may happen after new waiter makes 'tail' point to its qnode, but before waiter updates the predecessor (lock holder) qnode's next field

Simultaneous Release and Acquire



acquire() has completed `fetch_and_store`, knows `pred`, but **has not updated `pred->next` yet**

release() sees no waiters (`next == NULL`), but **knows acquire is in progress since the tail is not pointing at its own qnode**

T0 acquire:

```
my_node->next = NULL;
struct qnode *pred = FAS(&tail, my_node);
if (pred != NULL){ //queue !empty
    my_node->locked = TRUE;

    pred->next = my_node;
    while (my_node->locked); //spin
}
```

T1 release:

```
if (my_node->next == NULL) {
    if (CAS(&tail, my_node, NULL))
        return;

    while (my_node->next == NULL);
}
my_node->next->locked = FALSE;
```

Are any memory ordering instructions needed?

MCS Conclusions

- Grants requests in FIFO order
- Space: $2p + n$ words of space (p processes and n locks)
- Requires a local "queue node" to be passed in as a parameter
 - Alternatively, allocate these nodes dynamically in `acquire_lock`, and look them up in a table in `release_lock`
- Atomic primitives: Need `fetch_and_store`, `compare_and_swap`
- Spins only on local locations
 - Key lesson: Important to reduce memory traffic during synchronization
- Widely-used: e.g., monitors in Java VMs are variants of MCS

What about Pthreads?

- Most widely used API for multithreaded C code
 - Basic lock is `pthread_mutex_t`
- How is it implemented in Linux glibc?
 - Mix of techniques discussed here
 - `__pthread_lock`
 - First does adaptive number of spins, using TTAS
 - If not successful, adds self to linked list and suspends self
 - Similar in structure to MCS locks, used differently
 - Main benefit is ability for waits to timeout and set priority
 - `__pthread_unlock`
 - wakes waiting thread with highest priority, if any

Some Parallel Programming Techniques

- **Counter used by N threads**
 - Basic operation: counter++
 - Needs to be in critical section: lock; counter++; unlock, but only if exact counter value is critical
 - Alternative, cache friendlier approach if #incr >> #reads:
 - Use 1 counter per thread, properly padded
 - Increment only local counter → no locks needed
 - Approximate reads: sum up all local counters → no locks needed
- **Barriers**
 - N threads: lock; b_counter++; unlock; while (b_counter < N):
 - Alternative: use tree of barriers

Structuring Data for Caches

- **S1: segregate read-mostly data from frequently modified data**
 - E.g., if linked list payloads modified often, then separate linked list pointers and payload
 - Exact opposite of what you'd do on a uniprocessor
- **S2: segregate independently accessed data from each other**
 - Avoids false sharing
- **S3: use per core data wherever possible**
 - E.g., one ready queue per core on Linux, jemalloc arenas
- **S4: privatize write-mostly data**
 - E.g., counter example above

Locking Data

- **L1: use per-core reader/writer locks for read-mostly critical sections**
 - For read access, acquire local lock
 - For write access, acquire all locks (writes are expensive)
- **L2: segregate contended locks from associated data**
 - Prevents threads that are trying to acquiring lock from interfering with writing thread

Resources

- Pseudocode for the locks in this lecture and other variants on Michael Scott's webpage
 - <https://www.cs.rochester.edu/research/synchronization/pseudocode/queues.html>
 - See CLH and IBM K42 MCS variants
 - Other references: <http://locklessinc.com/articles/locks/>
- HP Labs atomic_ops project (Hans Boehm)
 - http://shiftright.com/mirrors/www.hpl.hp.com/research/linux/atomic_ops/index.php4
- C11/C++11 language includes atomic ops
 - Supported by the compiler