Lecture 8: Multiprocessor OSs & Synchronization

CSC 469H1F / CSC 2208H1F
Fall 2007
Angela Demke Brown

The Problem

- Coordinated management of shared resources
  - Resources may be accessed by multiple threads
  - Need to control accesses, prevent races
- Two main problems
  - 1) atomic access to shared data
    - preventing corruption or inconsistent views
  - 2) enforcing order
    - Condition synchronization (wait until X is true)
    - Barrier synchronization (all threads complete phase N before beginning phase N+1)
- We'll focus on shared data problem
  - Code that needs synchronized access to shared data is a *critical section*

Uniprocessor Solutions

- Protecting data shared between:
  - Multiple kernel threads
    - Disable / don't allow context switches in critical sections
  - Kernel threads and interrupt handlers
    - Disable interrupts *and* disallow context switches in critical sections
- Works because there is no *true concurrency*
- FreeBSD (at least to 5.3), Linux pre-2.6 had no kernel preemption
  - Only had to synchronize with interrupt handlers

Multiprocessors

- True concurrency – code executes simultaneously on multiple CPUs, possibly accessing shared data
  - Disable/disallow context switch doesn't help since multiple contexts are executing anyway
  - Disable interrupts only affects local CPU
- Need some help from the hardware
  - Simple ops can be done with special *atomic instructions*
    - E.g. set/increment/decrement variable
  - Grouping multiple instructions requires *locking*
    - Hardware atomic test_and_set (TAS), compare_and_swap (CAS) or load-linked/store-conditional instructions assist

Lock Options

- Spinlocks – loop testing lock variable until available
  - Good if you have nothing else to do
  - Or if expected wait is short (< 2 context switches)
  - Or if you aren't allowed to block (like in interrupt handler)
  - Or to build sleep lock primitives
- Focus today on spinlocks

```
boolean lock;

boolean TAS(boolean *lock)
{ /* pseudocode for HW atomic */
      boolean old = *lock;
      *lock = TRUE;
      return old;
}

void acquire(boolean *lock) {
      while(TAS(lock));
}

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

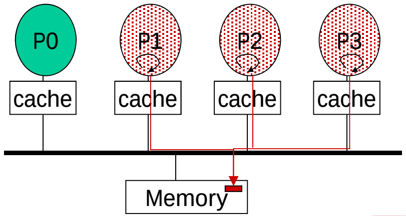
Contention and Scalability

- Locking serializes execution of critical sections
  - Limits ability to use multiple processors
- *Contention* refers to a lock that is held when another thread tries to acquire it
- *Scalability* refers to ability to expand size of a system
- Locks that are frequently contended limit scalability
  - Coarse-grained locking, large critical sections lead to increased contention
  - First multiprocessing support in Linux, FreeBSD, others, treated entire kernel as critical section protected by a single giant lock

---

Cost of Locking

- TAS(lock) operates on memory location atomically
- Leads to extra traffic and contention on memory bus
  - Slows down other memory operations as well



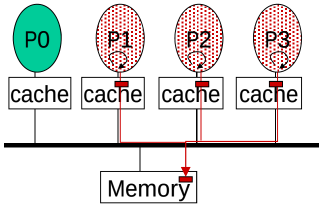
Building a better spinlock

- Idea: spin in cache, access memory only when likely that lock is available
  - Known as test_and_test_and_set

```
boolean lock;

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

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



Spinlock with backoff

- Idea: if lock is held, wait awhile before probing again
  - Best performance uses exponential backoff
  - Can cause fairness problems (new arrivals have shorter backoffs, more likely to detect free lock)

```
void acquire(boolean *lock) {
    int delay = 1;
    while(TAS(lock) == TRUE) {
        pause(delay)
        delay = delay * 2;
    }
}
```

Ticket Locks

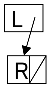
- Resolve fairness issues (FIFO order)
- Reduces number of atomic ops
- Lock consists of two counters
  - num_requests and num_releases

```
struct lock {
    int next_ticket = 0;
    int now_serving = 0;
}
void acquire(struct lock *l) {
    int my_ticket = FAA(&l->next_ticket);
    while(l->now_serving != my_ticket) ; //spin
}
void release(struct lock *l) {
    l->now_serving++;
}
```

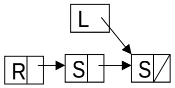
Queuing Locks

- Idea: Each CPU spins on a different location
  - Reduces cache coherence traffic, memory contention
  - Release unblocks next waiter only
  - Guarantees FIFO ordering
  - Lock acquire adds 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

MCS Lock Pseudocode

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

```
struct qnode {
    int locked;
    struct qnode *next;
}
void acquire(struct qnode *lock, struct qnode *my_node) {
    my_node->next = NULL;
    // atomically retrieve previous last node, and make
    // lock point to my_node
    struct qnode *pred = fetch_and_store(&lock, my_node);
    if (pred != NULL) { // queue not empty
        my_node->locked = TRUE;
        pred->next = my_node;
        while(my_node->locked) ; //spin
    }
}
```

MCS Lock Release

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

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

Example: Simultaneous Acquire

```
initial: lock=NULL;
T0: my_node->next = NULL;
T0: pred =
      fetch_and_store(
      &lock, my_node);
```

```
T1: my_node->next = NULL;
T1: pred =
      fetch_and_store(
      &lock, my_node);
```

fetch_and_store executes atomically in some order… either T0's op completes first, or T1's does.

If T0 first: old value of l is NULL, so pred = NULL and l is set to point at T0's qnode. For T1, old value of l (pred) is T0's qnode.

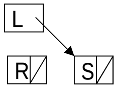
→ T0 acquires the lock and T1 spins on its qnode's locked value

If T1's fetch_and_store completes first, the situation is reversed

→ No additions are lost, but queue may not be fully linked together until all threads complete pred->next update

---

Ex: Simultaneous Release and Acquire



acquire() has completed fetch_and_store, knows pred, but has not updated pred->next.
release() sees no waiters (next == NULL), but knows acquire is in progress since lock is not pointing at releaser's qnode.

- Suppose lock is held, and there are no waiters when an acquire and release happen simultaneously
- Either acquirer's fetch_and_store or releaser's compare_and_swap will complete atomically before the other
  - If no waiters, lock and my_node must point to same location
  - If fetch_and_store has completed, lock will point to new qnode
    - compare_and_swap returns false, releaser waits for new waiter to finish updating next, then completes release

---

Resources

- Pseudocode for the locks in this lecture and other variants on Michael Scott's webpage
  - http://www.cs.rochester.edu/research/synchronization/pseudocode/ss.html
    - See CLH and IBM K42 MCS variants
- HP Labs atomic_ops project (Hans Boehm)
  - http://www.hpl.hp.com/research/linux/atomic_ops/
- Next time: avoiding locking and the Linux RCU (Read-Copy-Update) API