Week 5: Better Spinlocks

CSC 469 / CSC 2208

Fall 2018

(with thanks to Bogdan Simion, Tom Hart, Paul McKenney)
• Last week:
  • Processes communicate and coordinate via IPC
  • Pipes, sockets, signals, etc.

• Coordinating shared resources
  • Synchronization problem!
  • Contention and Scalability!
The Synchronization 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
- Need to know about *memory consistency model*
Contention and Scalability

- Locking serializes execution of critical sections
  - Limits ability to use multiple processors
  - Remember Amdahl’s law?
- **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 → increased contention
  - Fine-grained locking reduces contention but requires more locks
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)
- Focus will be on spinlocks

```c
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;
}
```
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 lock is likely to be available
  - Known as test_and_test_and_set (TTAS)

```c
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 – why?

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

- Resolve fairness issues (FIFO order)
- Added to Linux in 2.6.25 (2008)
- Lock consists of two counters (next_ticket, now_serving)

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

- Reduces number of atomic ops
- **Problems? How do we mitigate them?**
**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
  - \( R = \text{running} \)
  - \( S = \text{spinning} \)
- (c) Held lock
  - 2 waiters spinning
MCS locks in a nutshell

- Process 4 arrives, attempting to acquire lock

Diagrams: ©Bill Scherer – Rice University
MCS locks

- Process 4 swaps self into tail pointer
- Acquires pointer to predecessor (3) from swap on tail
- Note: Process 3 can’t leave without noticing that one or more successors will link in behind it because the tail no longer points to 3
MCS locks

- Process 4 links behind predecessor (3)
MCS locks

- Process 4 now spins until 3 signals that the lock is available by setting a flag in 4’s lock record
MCS locks

- Process 1 prepares to release lock
  - If its next field is set, signal successor directly
MCS locks

1. leaving
2. run
3. spin
4. spin

tail
MCS locks

A. Process 1 prepares to release lock
   - Suppose 1’s next pointer is still null (A)
     - attempt a compare_and_swap on the tail pointer; finds that tail no longer points to self
     - waits until successor pointer is valid (B)
     - signal successor (process 2)

B. 1 → 2 spin

University of Toronto, Department of Computer Science
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

```c
struct qnode {
    int locked;
    struct qnode *next;
}

void acquire(struct qnode **tail, struct qnode *my_node) {
    my_node->next = NULL;
    // atomically retrieve old last node, 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;
        pred->next = my_node;
        while(my_node->locked) ; //spin
    }
}
```
Example: Simultaneous Acquire

<table>
<thead>
<tr>
<th>T₀: my_node-&gt;next = NULL;</th>
<th>T₁: my_node-&gt;next = NULL;</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₀: pred = fetch_and_store(tail, my_node);</td>
<td>T₁: pred = fetch_and_store(tail, my_node);</td>
</tr>
</tbody>
</table>

- `fetch_and_store` executes atomically in some order...
- Either T₀’s op completes first, or T₁’s does.

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

*Note:* No additions are lost, but queue may not be fully linked together until all threads complete pred->next update
MCS Lock Release

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

```c
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 it swapped
        // 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
}
```
Ex: 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:

```c
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:

```c
if (my_node->next == NULL) {
    if (CAS(tail, my_node, NULL))
        return;
    while (my_node->next == NULL);
}
my_node->next->locked = FALSE;
```
MCS – concluding notes

- 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, additional code can allocate these dynamically in acquire_lock, and look them up in a table in release_lock.
- Spins only on local locations
  - Cache-coherent and non-cache-coherent machines
- Atomic primitives
  - Needs support for fetch_and_store and (ideally) compare_and_swap
- Key lesson
  - Importance of reducing memory traffic in synchronization
- Widely-used: e.g., monitor locks used in Java VMs are variants of MCS
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 (suggested reading): http://locklessinc.com/articles/locks/

• HP Labs atomic_ops project (Hans Boehm)

• C11 / C++11 language includes atomic ops
  • Supported by the compiler

• Next up: avoiding locking (non-blocking synchronization)
Locking: A necessary evil?

• Locks are an easy to understand solution to critical section problem
  • Protect shared data from corruption due to simultaneous updates
  • Protect against inconsistent views of intermediate states
• But locks have lots of problems
  • 1. Deadlock
  • 2. Priority inversion
  • 3. Not fault tolerant
  • 4. Convoying
  • 5. Expensive, even when uncontended
• Not easy to use correctly!
1. Deadlock

- Textbook definition: Set of threads blocked waiting for event that can only be caused by another thread in the same set
- Classic example:

```
P
Get A...
Get B

A
Get B...
Get A

B
Q
```

- Self-deadlock also a big issue
  - Thread holds lock on shared data structure and is interrupted
    - Interrupt handler needs same lock!
- Solutions exist (e.g., specify lock order, disable interrupts while holding lock) but add complexity
2. Priority Inversion

- Lower priority thread gets spinlock
- Higher priority thread becomes runnable and preempts it
  - Needs lock, starts spinning
  - Lock holder can’t run and release lock

- Solutions exist (e.g. disable preemption while holding spinlock, implement priority inheritance, etc.), but add complexity
3. Not fault tolerant

- If lock holder crashes, or gets delayed, no one makes progress

- Scheduler-conscious synchronization helps with delays (preemption, page faults)
  - Crashes require abort / restart
4. Convoying

- Threads doing similar work, started at different times, occasionally accessing shared data
  - e.g., multi-threaded web server
- Expect access to shared objects to be spread out over time
  - Lock contention should be low
- Delay of lock holder allows other threads to catch up
  - Lock becomes contended and tends to stay that way

=> Convoying
5. Expensive, even when uncontended

<table>
<thead>
<tr>
<th>Operation</th>
<th>Nanoseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>0.24</td>
</tr>
<tr>
<td>Clock Cycle</td>
<td>0.69</td>
</tr>
<tr>
<td>Atomic Increment</td>
<td>42.09</td>
</tr>
<tr>
<td>Cmpxchg Blind Cache Transfer</td>
<td>56.80</td>
</tr>
<tr>
<td>Cmpxchg Cache Transfer and Invalidate</td>
<td>59.10</td>
</tr>
<tr>
<td>SMP Memory Barrier (eieio)</td>
<td>75.53</td>
</tr>
<tr>
<td>Full Memory Barrier (sync)</td>
<td>92.16</td>
</tr>
<tr>
<td>CPU-Local Lock</td>
<td>243.10</td>
</tr>
</tbody>
</table>

McKenney, 2005 – 8-CPU 1.45 GHz PPC
Causes: Deeper Memory Hierarchy

Then:

CPU

CPU

Memory

Now:

CPU

L1

L2

L3

CPU

L1

L2

L3

Memory

• Memory speeds have not kept up with CPU speeds
  • 1984: no caches needed, since instructions slower than memory accesses
  • after ~2005: 3-4 level cache hierarchies, since instructions orders of magnitude faster than memory accesses
• Synchronization ops typically execute at memory speed
Causes: Deeper Pipelines

Then: Fetch → Execute → Retire

Now:  

• 1984: Many cycles per instruction
• 2005: Many instructions per cycle
  • 20 stage pipelines
  • CPU logic executes instructions out-of-order to keep pipeline full
  • Synchronization instructions must not be reordered
  • => synchronization stalls the pipeline
• Deeper pipelines not always better and processors are changing
Main issue with lock performance used to be contention
  - Techniques were developed to reduce overheads in contended case
    - And to reduce contention
  - Today, issue is degraded performance even when locks are always available
    - Together with other concerns about locks