Lecture 5: Locks and Avoiding Locks

Better spinlocks
Non-blocking Synchronization
Read-Copy Update
Transactional Memory

CSC 469 / CSC 2208
Fall 2017

(with thanks to Tom Hart, Paul McKenney & Angela Demke Brown)
• 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 \(\rightarrow\) 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?
  - What sort of problem can this run into?

```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 = running
S = spinning

(c) Held lock
2 waiters spinning
MCS locks in a nutshell

- Process 4 arrives, attempting to acquire lock
MCS locks

- Process 4 swaps self into tail pointer
- Acquires pointer to predecessor (3) from swap on tail
- Note: 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

- 4 links behind predecessor (3)
MCS locks

• 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
  - Suppose 1’s next pointer is still null
    - attempt a compare_and_swap on the tail pointer; finds that tail no longer points to self
    - waits until successor pointer is valid (already points to 2 in diagram)
    - signal successor (process 2)
MCS locks

1 - leaving
2 - run
3 - spin
4 - spin

tail
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 previous 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

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

```
T1: my_node->next = NULL;
T1: pred = fetch_and_store(
    &tail, 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 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.

<table>
<thead>
<tr>
<th>T0 acquire:</th>
<th>T1 release:</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct qnode *pred = FAS(&amp;tail, my_node);</td>
<td></td>
</tr>
<tr>
<td>if (pred != NULL) { //queue !empty</td>
<td>if (my_node-&gt;next == NULL) {</td>
</tr>
<tr>
<td>my_node-&gt;locked = TRUE;</td>
<td>if (CAS(&amp;tail, my_node, NULL))</td>
</tr>
<tr>
<td>pred-&gt;next = my_node;</td>
<td>return;</td>
</tr>
<tr>
<td>while (my_node-&gt;locked); //spin</td>
<td>while (my_node-&gt;next == NULL);</td>
</tr>
<tr>
<td>}</td>
<td>my_node-&gt;next-&gt;locked = FALSE;</td>
</tr>
</tbody>
</table>
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
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 and others
  - `__pthread_lock`
    - First does adaptive number of spins, using TTAS
    - If not successful, adds self to linked list and suspends self
  - `__pthread_unlock`
    - wakes waiting thread with highest priority, if any
  - `__pthread_alt_lock / unlock` functions use `wait_node`
    - Similar in structure to MCS locks, used differently
    - Main benefit is ability for waits to timeout
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](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/](http://locklessinc.com/articles/locks/)

- HP Labs atomic_ops project (Hans Boehm)

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

- 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

- 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
- \( \Rightarrow \) synchronization stalls the pipeline

Deeper pipelines not always better and processors are changing
Performance

• 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
Critical section efficiency

• Assuming little to no contention, and no caching effects in CS

\[
\text{Efficiency} = \frac{T_c}{T_c + T_a + T_r}
\]

• Even if lock contention is negligible, critical section efficiency must be addressed!
Locks: A necessary evil?

Idea: Don’t lock if we don’t need to!

- Non-Blocking Synchronization (NBS)
  - Use term “lockless” to describe strategies that avoid locking
NBS Basics

• Make change optimistically, roll back and retry if conflict detected

```c
atomic_inc(int *counter) {
    int value;
    do {
        value = *counter;
        } while (!CAS(counter, value, value+1);
}
```

• Complex updates (e.g. modifying multiple values in a structure) are hidden behind a single commit point using atomic instructions
Example: Stack Data Structure

- Lock-based synchronization:

```c
/* definitions */

typedef struct node_s {
    int val;
    struct node_s *next;
} node_t;

typedef struct stack_s {
    node_t *top;
    lock_t *stack_lock;
} stack_t;

void push(stack_t *S, node_t *n) {
    lock(S->stack_lock);
    n->next = S->top; S->top=n;
    unlock(S->stack_lock);
}

node_t* pop(stack_t *S) {
    node_t *n = NULL;
    lock(S->stack_lock);
    if (S->top != NULL) {
        n = S->top;
        S->top = S->top->next;
    }
    unlock(S->stack_lock);
    return n;
}
```
Non-blocking stack (take 1)

```c
/* definitions */

typedef struct node_s {
    int val;
    struct node_s *next;
} node_t;

/* Stack type is just a * pointer to a node. */
typedef node_t *stack_t;

void push(stack_t *S, node_t *n) {
    node_t *first;
    do {
        first = *S;
        n->next = first;
    } while (!CAS(S,first,n));
}

node_t* pop(stack_t *S) {
    node_t *first, *second;
    do {
        first = *S;
        if (first != NULL) {
            second = first->next;
        } else return NULL;
    } while (!CAS(S,first,second));
    return first;
}
```

What’s wrong?
ABA Problem

- Ti, Tj both doing pops and pushes, interleaved as follows:

```plaintext
SS

Ti: pop()  first
    second
    (interrupt)

Tj: a = pop();
    b = pop();
```

![Diagram showing the ABA problem with Ti and Tj operations interleaved and the sequence of stack operations indicated.](image-url)
ABA Problem

- CAS(x, y, z) succeeds if value stored at x matches y

`CAS(S, first, second)`

- Ti: pop()
  - first
  - second
  - (interrupt)

- Tj:
  - a = pop();
  - b = pop();
  - push(n);
  - push(a);

S

Ti: pop() first second (interrupt)

S

Tj:

A

B

C

A

N

B

C (not on stack)
One Solution

- Include a version number with every pointer
  - pointer_t = <pointer, version>
  - Increment version number (atomically) every time you modify pointer
  - Change to version number guarantees CAS will fail if pointer has changed
  - Requires double-word CAS operation (not every architecture provides this)
  - Use garbage collection to reclaim memory later
    - May restrict reuse of memory