Scalable Locking

Barriers
Outline

Scalable Locking

Barriers
An alternative lock – ticket lock

- Each lock has a ticket associated with it
- Locks and tickets are initialized to 0

```c
lock(l):
    // atomic_add returns previous value
    my_ticket = atomic_add(l.ticket, 1);
    while(l != my_ticket);

unlock(l):
    l += 1;   // increase now serving, could also be an atomic_add
```
## Performance tradeoffs of ticket locks

<table>
<thead>
<tr>
<th>Operation</th>
<th>Atomics</th>
<th>Reads/Writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock</td>
<td>1</td>
<td>unbounded</td>
</tr>
<tr>
<td>Unlock</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Review of a ticket lock

- 1 atomic per `lock()`, so $O(n)$ atomics when $n$ threads contend
- $O(1)$ space per lock
- Unbounded reads/writes
  - Lock can be in remote cache
  - Generates cache coherence traffic while threads are waiting
- Fair
  - Threads are granted locks in FIFO order
  - But what happens when threads in queue are pre-empted?
Internals of a ticket lock

- On a lock
  - Atomic to increment the ticket
  - Read on “now-serving” ticket
- On an unlock
  - Increase of “now-serving” ticket
  - Broadcast of this update to all readers
Queues vs Broadcast

If you’re standing in a queue, who do you need to monitor?
Queueing Locks

- Use a queue to maintain waiting threads
- Threads in queue acquire locks in FIFO order
- A thread that releases a lock only notifies the next thread in the queue
  - Essentially transferring ownership of the lock
def lock:
    create queue entry for this thread
    add entry to queue for lock

    if this thread is the first in queue:
        we have lock!
    else:
        wait for lock to be passed to us

def unlock:
    pass lock to next thread in queue (if any)
Data Structure for Queue

```
struct queue_entry {
    struct queue_entry *next;
    int waiting;
};
```

- *next is pointer to next entry in queue
- waiting is flag (initially 1) that set to zero when thread is given ownership of the lock
struct lock {
    struct queue_entry *tail = NULL;
};

• What do lock and unlock methods look like?
The Lock method

```c
void acquire_lock(lock *l) {
    struct queue_entry me;
    me.next = NULL;
    me.waiting = 1;
    prev = atomic_swap(l->tail, &me);
    if(prev == NULL) {
        // nobody waiting for lock
        return;
    } else {
        prev->next = &me;
        while(me.waiting);
    }
}
```

What bugs do you see here?
void acquire_lock(lock *l, struct queue_entry *me) {
    me->next = NULL;
    me->waiting = 1;

    write_to_all_fence();

    prev = atomic_swap(l->tail, &me);

    if(prev == NULL) {
        // nobody waiting for lock
        me->waiting = 0; // not really needed
    } else {
        prev->next = &me;
        while(me.waiting);
    }

    // prevent ops in critical section
    // from executing before the lock is acquired

    read_to_all_fence();
}
void release_lock(lock *l, struct queue_entry *me) {
    if(me->next == NULL) {
        // nobody waiting after me
        atomic_CAS(l->tail, me, NULL);
        return;
    } else {
        me->next->waiting = 0;
    }
}

What bugs do you see here?
The Correct Unlock method

```c
void release_lock(lock *l, struct queue_entry *me) {
    struct queue_entry *succ = me->next;

    // make all operations in critical section
    // visible
    all_to_write_fence();

    if(succ == NULL) {
        if atomic_CAS(l->tail, me, NULL) == me return;
        while((succ = me->next) == NULL);
    }
    succ->waiting = 0;
}
```
What we have just described is the Mellor-Crummey–Scott (MCS) lock.

- How does the MCS lock compare to the ticket lock?
  - In atomics?
  - In reads?
  - In writes?
  - In space?
  - In API/interface?
Outline

Scalable Locking

Barriers
Barriers

- Barriers (and Condition variables) are synchronization mechanisms like locks
- Generally not used for mutual exclusion
- Mostly used for communication/“synchronization”
  - Threads wait for other threads to arrive at a barrier
Barrier Interface

- **Creation**
  - `barrier.create(n)` where `n` is number of threads participating

- **Waiting for other threads to arrive**
  - `barrier.sync()` - blocks until all participating threads have invoked sync
  - Barriers are commonly used many times

- **Uncommonly used, but useful sometimes:**
  - `barrier.arrive()` – thread has arrived at barrier and moved on
  - Not discussing this, but you will study them

- **Destruction**
struct barrier {
    int nthreads;
    int arrived;
};

create(n) {
    struct barrier *c = calloc(1, sizeof(struct barrier));

    c->nthreads = n;
    c->arrived = 0;

    return c;
}
def sync(b):
    prev = b.arrived++ // atomic fetch and add

    while (b.arrived < b.nthreads);

    if(prev == b.threads - 1) {
        b.arrived = 0;
    }

What’s wrong here?

- You need a barrier between leaving the while loop and the reset to b.arrived
  - Otherwise, threads may not all exist
  - Cannot distinguish sync() immediately followed by sync()
Sense-reversing barriers

- Separate:
  - Count of arriving threads
  - Waiting
Sync for a sense-reversing barrier

struct barrier {
    int nthreads;
    int arrived;
    int sense;
};

def sync(b)
    prev = b.arrived++

    if(prev == b.threads - 1) {
        b.arrived = 0;
        b.sense = 1;
    }

    while(b.sense != 1);

    b.sense = 0;

• Correct?
  • No, same problem as before!
Sync for a sense-reversing barrier: Correct version

```c
struct barrier {
    ...
    int sense;
    int *local_sense;
};

def create(n):
    // initializes local_sense to zero
    b->local_sense = calloc(n, sizeof(int));
    b->sense = 0;
    ...

Continued ...
```
def sync(b):
    s = not b->local_sense[me];
    b->local_sense[me] = s;

    prev = b.arrived++
    if(prev == b.threads - 1) {
        b.arrived = 0;

        // make sure all writes are visible
        // before write to sense

        all_to_write_fence();
        b.sense = s;
    }

    // different invocations of sync now
    // wait for different values of s
    while(b.sense != s);

    // do not allow operations after barrier to happen
    // before this fence
    read_to_all_fence();
Scalable Barriers?

- Do all threads need to read `sense`?
- Barriers are computing a sum
  - Can this be done in parallel?