CS5460/6460: Operating Systems

Lecture 14: Scalability techniques

Anton Burtsev
February, 2014
Recap: read and write barriers

```c
void foo(void)
{
    a = 1;
    smp_wmb();
    b = 1;
}

void bar(void)
{
    while (b == 0)
        continue;
    smp_rmb();
    assert(a == 1);
}
```
scheduler(void)
{
    for(;;) {
        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        {
            if(p->state != RUNNABLE)
                continue;
            p->state = RUNNING;
            swtch(&cpu->scheduler, proc->context);
        }
        release(&ptable.lock);
    }
}
Scalable spinlocks
Exim collapse
Spinlock collapse

- We discussed two solutions:
  - Per-core hash-table
  - Read copy update

- Is it possible to build scalable spinlocks?
struct qnode {
    volatile void *next;
    volatile char locked;
};

typedef struct {
    struct qnode *v;
} mcslock_t;

arch_mcs_lock(mcslock_t *l, volatile struct qnode *mynode) {
    struct qnode *predecessor;
    mynode->next = NULL;
    predecessor = (struct qnode *)xchg((long *)&l->v, (long)mynode);
    if (predecessor) {
        mynode->locked = 1;
        barrier();
        predecessor->next = mynode;
        while (mynode->locked) ;
    }
}
arch_mcs_lock(mcslock_t *l, volatile struct qnode *mynode) {
    struct qnode *predecessor;
    mynode->next = NULL;
    predecessor = (struct qnode *)xchg((long *)&l->v, (long)mynode);
    if (predecessor) {
        mynode->locked = 1;
        barrier();
        predecessor->next = mynode;
        while (mynode->locked) ;
    }
}

arch_mcs_unlock(mcslock_t *l, volatile struct qnode *mynode) {
    if (!mynode->next) {
        if (cmpxchg((long *)&l->v, (long)mynode, 0) == (long)mynode)
            return;
        while (!mynode->next) ;
    }
    ((struct qnode *)mynode->next)->locked = 0;
}
Why does this scale?
Ticket spinlock

```c
void spin_lock(spinlock_t *lock) {
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket) 
        ; /* Spin */
}
```

```c
void spin_unlock(spinlock_t *lock) {
    lock->current_ticket++;
}
```

```c
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

- How many cache messages are needed to acquire the lock?
Ticket spinlock

- How many cache messages are needed to acquire the lock?
  - Proportional to the number of cores
    - 1 message for atomic_inc()
    - N messages from other cores which hold the lock and update current_ticket upon release

```c
void spin_lock(spinlock_t *lock) {
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket) {
        /* Spin */
    }
}

void spin_unlock(spinlock_t *lock) {
    lock->current_ticket++;
}

struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```
arch_mcs_lock(mcslock_t *l, volatile struct qnode *mynode) {
    struct qnode *predecessor;
    mynode->next = NULL;
    predecessor = (struct qnode *)xchg((long *)&l->v, (long)mynode);
    if (predecessor) {
        mynode->locked = 1;
        barrier();
        predecessor->next = mynode;
        while (mynode->locked);
    }
}

arch_mcs_unlock(mcslock_t *l, volatile struct qnode *mynode) {
    if (!mynode->next) {
        if (cmpxchg((long *)&l->v, (long)mynode, 0) == (long)mynode)
            return;
        while (!mynode->next);
    }
    ((struct qnode *)mynode->next)->locked = 0;
}
struct qnode {
    volatile void *next;
    volatile char locked;
    char __pad[0] __attribute__((aligned(64)));
};

typedef struct {
    struct qnode *v __attribute__((aligned(64)));
} mcslock_t;
Exim: MCS vs ticket lock

Throughput (messages/sec)

Cores

Ticket lock
MCS lock
Hardware transactional memory
insert(int data)
{
    struct list *l;
    l = malloc(sizeof *l);
    l->data = data;
    l->next = list;
    list = l;
}
insert(int data)
{
    struct list *l;
    l = malloc(sizeof *l);
    l->data = data;
    acquire(&listlock);
    l->next = list;
    list = l;
    release(&listlock);
}

We protected list with locks
9 insert(int data)
10 {
11   struct list *l;
13   l = malloc(sizeof *l);
14   l->data = data;
15   l->next = list;
16   list = l;
17 }
Intel Transactional Synchronization Extensions (TSX)

- Two modes of execution
  - Restricted transactional memory (RTM)
  - Hardware lock elision (HLE)
_retry: xbegin _abort

// critical section
xend

_abort:

// Fallback path, retry
// transaction or acquire a lock
Restricted transactional memory

- Some instructions and events may cause aborts
  - Uncommon instructions, interrupts, faults
- Software must provide non-transactional path
Hardware lock elision

- Is it possible to use transactional memory without changing the code?
  - Hint: use existing locks as hints for transactions
Hardware lock elision

mov eax, 1

_try:  xacquire lock xchg lock, eax
cmp eax, 0
jz _success

_spin: pause
  cmp lock, 1
  jz _spin
jmp _try

// critical section

xrelease mov lock, 0
Hardware lock elision

- Try to execute lock code in the transactional manner
- In case of abort, do a transparent restart
  - Execute same software code without elision
Scalable commutativity rule
Thinking about scalability

- Scalability is typically viewed as a property of implementation
- Is it possible to detect scalability bottlenecks at the level of interfaces
Whenever interface operations commute, they can be implemented in a way that scales
Designing commutative interfaces

• Decompose compound operations
  • fork()
    - Creates a new process and snapshots its entire memory, file descriptors, signal masks
    - Fails to commute with memory writes, address space operations, and many file descriptor operations
  • stat()
    - Retrieves many stats simultaneously
    - Fails to commute with any operation that changes any attribute returned by stat, e.g., link, chmod, chown, write, and even read
Designing commutative interfaces (2)

- Embrace specification non-determinism
  - Lowest available file-descriptor
- Permit weak ordering
  - Local domain sockets
    - send and receive operations do not commute
    - Unnecessary in case of multiple readers and writers
- Release resources asynchronously
  - munmap requires expensive TLB shootdowns before it can return
One more scalability technique: sloppy counters
Reference counting

- Reference counting is used to keep track of object users
  - Increment counter for every new user
  - Decrement counter when users leave
  - Deallocate object when counter is 0, e.g. there are no users
Atomic increment on 64 cores

Reference counting is a big problem
Sloppy counters

- Observation: kernel rarely needs true value of a reference counter
Sloppy counters

- Observation: kernel rarely needs true value of a reference counter
Sloppy counters

- Observation: kernel rarely needs true value of a reference counter
Sloppy counters

- Observation: kernel rarely needs true value of a reference counter
Sloppy counters

- Observation: kernel rarely needs true value of a reference counter
Sloppy counters

- Observation: kernel rarely needs true value of a reference counter
Sloppy counters

- Observation: kernel rarely needs true value of a reference counter
Sloppy counters

- Observation: kernel rarely needs true value of a reference counter
Sloppy counters

- Observation: kernel rarely needs true value of a reference counter
Sloppy counters

- Observation: kernel rarely needs true value of a reference counter
Sloppy counters

- Observation: kernel rarely needs true value of a reference counter
Sloppy counters

- Observation: kernel rarely needs true value of a reference counter
Sloppy counters

- Observation: kernel rarely needs true value of a reference counter

```
Core 0
```

```
Core 1
```

```
rm /tmp/foo
```

```

per-core 1 per-core

shared

dentry sloppy counter
```
Sloppy counters

- Observation: kernel rarely needs true value of a reference counter
Exim: more scalability with sloppy counters
Conclusion
Thank you!