CS238P: Operating Systems

Lecture 12: Synchronization (and Scalability)

Anton Burtsev
March, 2018
Spinlocks
struct spin_lock {
    int locked;
};

insert(int data)
{
    struct list *l;
    l = malloc(sizeof *l);
    spinlock_acquire(&spin_lock);
    l->data = data;
    l->next = list;
    list = l;
    spinlock_release(&spin_lock);
}

Critical sections with spinlocks

- Critical section
Correct implementation

1573 void
1574 spinlock_acquire(struct spinlock *lk)
1575 {
1576   ...
1580   // The xchg is atomic.
1581   while(xchg(&lk->locked, 1) != 0) ;
1582   ...
1592 }
static inline uint xchg(volatile uint *addr, uint newval) {
    uint result;

    // The + in "+m" denotes a read-modify-write operand.
    asm volatile("lock; xchgl %0, %1"
                 : "+m" (*addr), "=a" (result)
                 : "1" (newval)
                 : "cc");

    return result;
}
void acquire(struct spinlock *lk) {
...

   // The xchg is atomic.
   while(xchg(&lk->locked, 1) != 0) ;

   // Tell the C compiler and the processor to not move loads or stores
   // past this point, to ensure that the critical section’s memory
   // references happen after the lock is acquired.
   __sync_synchronize();

...
Mutexes
struct mutex {
    int locked;
};

insert(int data) {
    struct list *l;
    l = malloc(sizeof *l);
    mutex_acquire(&spin_lock);
    l->data = data;
    l->next = list;
    list = l;
    mutex_release(&spin_lock);
}

• Critical section
void mutex_acquire(struct mutex *m) {
    // The xchg is atomic.
    while(xchg(&m->locked, 1) != 0)
        yield();
    __sync_synchronize();
}

void mutex_unlock(struct mutex *m) {
    __sync_synchronize();
    m->locked = 0;
}
Conditional variables
Conditional variables

```c
pthread_mutex_lock(&m);
while (count < 10) {
    pthread_cond_wait(&cv, &m);
}
pthread_mutex_unlock(&m);
```

```c
while (1) {
    pthread_mutex_lock(&m);
    count++;
    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&m);
}
```
Semaphores
Suppose a library has 10 identical study rooms, to be used by one student at a time. Students must request a room from the front desk if they wish to use a study room. If no rooms are free, students wait at the desk until someone relinquishes a room. When a student has finished using a room, the student must return to the desk and indicate that one room has become free.
typedef struct sem_t {
    int count;
    mutex_t m;
    condition_t cv;
} sem_t;

int sem_init(sem_t *s, int value) {
    s->count = value;
    mutex_init(&s->m, NULL);
    cond_init(&s->cv, NULL);
    return 0;
}
sem_wait(sem_t *s) {
    mutex_lock(&s->m);
    while (s->count == 0) {
        cond_wait(&s->cv, &s->m);
    }
    s->count--;
    mutex_unlock(&s->m);
}

sem_post(sem_t *s) {
    mutex_lock(&s->m);
    s->count++;
    cond_signal(&s->cv);
    pthread_mutex_unlock(&s->m);
}
Semaphores as mutexes

- Mutexes can be implemented as binary semaphores
  - count = 1
  - However watch out for recursive use of the mutex
Back to spinlocks
void spinlock_acquire(struct spinlock *lk) {
...
// The xchg is atomic.
while(xchg(&lk->locked, 1) != 0) ;
...
What is really wrong with locks?
What is really wrong with locks?

- Scalability
48-core AMD server
Exim collapse

Throughput (messages/second) vs Cores

The graph shows the throughput of Exim in messages per second as a function of the number of cores. The throughput increases with the number of cores up to a certain point, after which it experiences a collapse, highlighted by the red circle.
# Oprofile results

## 40 cores: 10000 msg/sec

<table>
<thead>
<tr>
<th>samples</th>
<th>%</th>
<th>app name</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2616</td>
<td>7.3522</td>
<td>vmlinux</td>
<td>radix_tree_lookup_slot</td>
</tr>
<tr>
<td>2329</td>
<td>6.5456</td>
<td>vmlinux</td>
<td>unmap_vmas</td>
</tr>
<tr>
<td>2197</td>
<td>6.1746</td>
<td>vmlinux</td>
<td>filemap_fault</td>
</tr>
<tr>
<td>1488</td>
<td>4.1820</td>
<td>vmlinux</td>
<td>___do_fault</td>
</tr>
<tr>
<td>1348</td>
<td>3.7885</td>
<td>vmlinux</td>
<td>copy_page_c</td>
</tr>
<tr>
<td>1182</td>
<td>3.3220</td>
<td>vmlinux</td>
<td>unlock_page</td>
</tr>
<tr>
<td>966</td>
<td>2.7149</td>
<td>vmlinux</td>
<td>page_fault</td>
</tr>
</tbody>
</table>

## 48 cores: 40000 msg/sec

<table>
<thead>
<tr>
<th>samples</th>
<th>%</th>
<th>app name</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>13515</td>
<td>34.8657</td>
<td>vmlinux</td>
<td>lookup_mnt</td>
</tr>
<tr>
<td>2002</td>
<td>5.1647</td>
<td>vmlinux</td>
<td>radix_tree_lookup_slot</td>
</tr>
<tr>
<td>1661</td>
<td>4.2850</td>
<td>vmlinux</td>
<td>filemap_fault</td>
</tr>
<tr>
<td>1497</td>
<td>3.8619</td>
<td>vmlinux</td>
<td>unmap_vmas</td>
</tr>
<tr>
<td>1026</td>
<td>2.6469</td>
<td>vmlinux</td>
<td>___do_fault</td>
</tr>
<tr>
<td>914</td>
<td>2.3579</td>
<td>vmlinux</td>
<td>atomic_dec</td>
</tr>
<tr>
<td>896</td>
<td>2.3115</td>
<td>vmlinux</td>
<td>unlock_page</td>
</tr>
</tbody>
</table>
Exim collapse

• \texttt{sys\_open} eventually calls:

```c
struct vfsmount *lookup_mnt(struct path *path)
{
    struct vfsmount *mnt;
    spin_lock(&vfsmount_lock);
    mnt = hash_get(mnts, path);
    spin_unlock(&vfsmount_lock);
    return mnt;
}
```
Exim collapse

• sys_open eventually calls:

```c
struct vfsmount *lookup_mnt(struct path *path)
{
    struct vfsmount *mnt;
    spin_lock(&vfsmount_lock);
    mnt = hash_get(mnts, path);
    spin_unlock(&vfsmount_lock);
    return mnt;
}
```

• spin_lock and spin_unlock use many more cycles than the critical section
struct spinlock_t {
    int current_ticket ;
    int next_ticket ;
}

void spin_lock ( spinlock_t *lock) {
    int t = atomic_fetch_and_inc (&lock -> next_ticket );
    while (t != lock -> current_ticket )
        ; /* spin */
}

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

```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;
}
```
## Spin lock implementation

**Allocate a ticket**

```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;
}
```
Spin lock implementation

```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;
}
```
Spin lock implementation

```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;
}
```
Spin lock implementation

```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;
}
```

Allocate a ticket

120-420 cycles
Spin lock implementation

```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;
}
```
Spin lock implementation

Bunch of cores are spinning

```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;
}
```
Spin lock implementation

```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++;
}
```

Broadcast message (invalidate the value)
Spin lock implementation

```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++;
    lock->next_ticket;
}
```

Cores don't have the value of current_ticket
Spin lock implementation

```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++;
}
```

Re-read the value
Spin lock implementation

```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;
}
```
• In most architectures, the cache-coherence reads are serialized (either by a shared bus or at the cache line’s home or directory node)

• Thus completing them all takes time proportional to the number of cores.

• The core that is next in line for the lock can expect to receive its copy of the cache line midway through this process.
  • N/2
Atomic synchronization primitives do not scale well
Atomic increment on 64 cores
What can we do about it?
Solution #1: per-core mount tables

- Observation: mount table is rarely modified

```c
struct vfsmount *lookup_mnt(struct path *path)
{
    struct vfsmount *mnt;
    if ((mnt = hash_get(percore_mnts[cpu()], path)))
        return mnt;
    spin_lock(&vfsmount_lock);
    mnt = hash_get(mnts, path);
    spin_unlock(&vfsmount_lock);
    hash_put(percore_mnts[cpu()], path, mnt);
    return mnt;
}
```
Solution #1: per-core mount tables

- Observation: mount table is rarely modified

```c
struct vfsmount *lookup_mnt(struct path *path)
{
    struct vfsmount *mnt;
    if ((mnt = hash_get(percore_mnts[cpu()], path)))
        return mnt;
    spin_lock(&vfsmount_lock);
    mnt = hash_get(mnts, path);
    spin_unlock(&vfsmount_lock);
    hash_put(percore_mnts[cpu()], path, mnt);
    return mnt;
}
```
Solution #1: per-core mount tables

- Observation: mount table is rarely modified

```c
struct vfsmount *lookup_mnt(struct path *path)
{
    struct vfsmount *mnt;
    if ((mnt = hash_get(percore_mnts[cpu()], path)))
        return mnt;
    spin_lock(&vfsmount_lock);
    mnt = hash_get(mnts, path);
    spin_unlock(&vfsmount_lock);
    hash_put(percore_mnts[cpu()], path, mnt);
    return mnt;
}
```

- Fast path: local hash lookup
Solution #1: per-core mount tables

- Observation: mount table is rarely modified

```c
struct vfsmount *lookup_mnt(struct path *path)
{
    struct vfsmount *mnt;
    if ((mnt = hash_get(percore_mnts[cpu()], path)))
        return mnt;
    spin_lock(&vfsmount_lock);
    mnt = hash_get(mnts, path);
    spin_unlock(&vfsmount_lock);
    hash_put(percore_mnts[cpu()], path, mnt);
    return mnt;
}
```

- Slow path: lookup global mount table, then update local, per-core copy
Solution #2:
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) ;
    }
}
MCS lock

MCS lock

CPU 1

CPU 2

CPU 1

CPU 2

CPU 2
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 */
}

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

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

- Remember O(N) re-fetch messages after invalidation broadcast
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;
}
Cache line isolation

```c
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

![Graph showing throughput (messages/sec) vs cores. The graph compares Ticket lock and MCS lock. TheTicket lock line shows a steady increase in throughput with an inflection point at around 42 cores, while the MCS lock line maintains a consistent increase until it also shows a decrease at around 42 cores.](image)
RCU: Read Copy Update
Read copy update

- Goal: remove “cat” from the list
  - There might be some readers of “cat”
- Idea: control the pointer dereference
  - Make it atomic
Read copy update (2)

• Remove “cat”
  • Update the “boa” pointer
  • All subsequent readers will get “gnu” as boa->next
Read copy update (2)

- Wait for all readers to finish
  - synchronize_rcu()
Read copy update (3)

- Readers finished
  - Safe to deallocate “cat”
Read copy update (4)

- New state of the list
How can we build this?

- Disable preemption while using the RCU data
  - `rcu_lock()`, `rcu_unlock()`
- Wait for all RCU readers to finish
  - Schedule something on each CPU
  - If you managed to run on a CPU
    - You preempted a thread on that CPU
    - Thus this thread exited the RCU lock/unlock section
void rcu_read_lock()
{
    preempt_disable[cpu_id()]++;
}

void rcu_read_unlock()
{
    preempt_disable[cpu_id()]--;
}

void synchronize_rcu(void)
{
    for_each_cpu(int cpu)
    {
        run_on(cpu);
    }

RCU implementation
What does it mean to run on a CPU?

- In xv6 scheduler() function goes through a list of all processes
  - If we keep a mask of allowable CPUs for each process
  - On each CPU the scheduler() function will pick processes with a proper mask
- run_on(cpu)
  - sets the mask for the current process
  - Invokes scheduler()
    - Calls yield(), which in turn calls swtch()
In practice...

- Linux just waits for all CPUs to pass through a context switch
  - Instead of scheduling the updater on each CPU
struct vfsmount *lookup_mnt(struct path *path)
{
    struct vfsmount *local_mnts;
    struct vfsmount *mnt;

    rcu_read_lock();
    local_mnts = rcu_dereference(mnts);
    mnt = lookup_mnt(local_mnts, path);
    rcu_read_unlock();

    return mnt;
}

RCU example: lookup_mnt()
Why do we need rcu_dereference()?

```c
struct vfsmount *lookup_mnt(struct path *path)
{
    ...
    rcu_read_lock();
    local_mnts = rcu_dereference(mnts);
    mnt = lookup_mnt(local_mnts, path);
    rcu_read_unlock();
    ...
}
```
Memory barriers

```c
#define __rcu_assign_pointer(p, v, space) \
    do { \\ 
        smp_wmb(); \\ 
        (p) = (typeof(*v) __force space *)(v); \\ 
    } while (0)
```
syscall_t *table;
spinlock_t table_lock;

int invoke_syscall(int number, void *args...)
{
    syscall_t *local_table;
    int r = -1;

    rcu_read_lock();
    local_table = rcu_deference(table);
    if (local_table != NULL)
        r = local_table[number](args);
    rcu_read_unlock();

    return r;
}
void retract_table()
{
    syscall_t *local_table;
    spin_lock(&table_lock);
    local_table = table;
    rcu_assign_pointer(&table, NULL);
    spin_unlock(&table_lock);
    synchronize_rcu();
    kfree(local_table);
}
Recap: read copy update
Conclusion

- What RCU is good for?
Conclusion

• What RCU is good for?
  • Read-heavy workload
  • Updates are rare
    - synchronize_rcu is slow
  • System call example:
    - You acquire a lock every time you execute a system call
    - But really the table might never change

• What if you need fast updates?
Conclusion

• What RCU is good for?
  • Read-heavy workload
  • Updates are rare
    – synchronize_rcu is slow
  • System call example:
    – You acquire a lock every time you execute a system call
    – But really the table might never change

• What if you need fast updates?
  • Fine-grained, scalable spinlocks
  • Lock-free synchronization
  • Transactional memory
Thank you!