Coherence and Synchronization

- Topics: synchronization primitives (Sections 5.4-5.5)
Constructing Locks

• Applications have phases (consisting of many instructions) that must be executed atomically, without other parallel processes modifying the data

• A lock surrounding the data/code ensures that only one program can be in a critical section at a time

• The hardware must provide some basic primitives that allow us to construct locks with different properties

• Lock algorithms assume an underlying cache coherence mechanism – when a process updates a lock, other processes will eventually see the update
Race conditions

• Example:
  • Disk driver maintains a list of outstanding requests
  • Each process can add requests to the list
```c
struct list {
    int data;
    struct list *next;
};

struct list *list = 0;

insert(int data) {
    struct list *l;
    l = malloc(sizeof *l);
    l->data = data;
    l->next = list;
    list = l;
}
```

**List implementation (no locks)**

- List
  - One data element
  - Pointer to the next element
```c
1 struct list {
2   int data;
3   struct list *next;
4 }
...
6 struct list *list = 0;
...
9 insert(int data)
10 {
11   struct list *l;
12
13   l = malloc(sizeof *l);
14   l->data = data;
15   l->next = list;
16   list = l;
17 }
```

List implementation (no locks)

- Global head
List implementation
(no locks)

- Insertion
  - Allocate new list element
List implementation (no locks)

- Insertion
  - Allocate new list element
  - Save data into that element

```c
1 struct list {
2   int data;
3   struct list *next;
4 };...
6 struct list *list = 0;
...
9 insert(int data)
10 {
11   struct list *l;
12
13   l = malloc(sizeof *l);
14   l->data = data;
15   l->next = list;
16   list = l;
17 }
```
List implementation (no locks)

- Insertion
  - Allocate new list element
  - Save data into that element
  - Insert into the list
Now what happens when two CPUs access the same list
Request queue (e.g. pending disk requests)

- Linked list, list is pointer to the first element
CPU1 allocates new request
CPU2 allocates new request
CPUs 1 and 2 update
next pointer
CPU1 updates head pointer

list = l

Diagram showing a list structure with nodes and pointers.
CPU2 updates head pointer

list = 1
State after the race
(red element is lost)
Mutual exclusion

- Only one CPU can update list at a time
List implementation with locks

- Critical section
How can we implement acquire()?
void acquire(struct spinlock *lk) {
  for(;;) {
    if(!lk->locked) {
      lk->locked = 1;
      break;
    }
  }
}

- Spin until lock is 0
- Set it to 1
Still incorrect

21 void
22 acquire(struct spinlock *lk)
23 {
24     for(;;) {
25         if(!lk->locked) {
26             lk->locked = 1;
27             break;
28         }
29     }
30 }

• Two CPUs can reach line #25 at the same time
  • See not locked, and
  • Acquire the lock
• Lines #25 and #26 need to be atomic
  • I.e. indivisible
Synchronization

• The simplest hardware primitive that greatly facilitates synchronization implementations (locks, barriers, etc.) is an atomic read-modify-write

• Atomic exchange: swap contents of register and memory

• Special case of atomic exchange: test & set: transfer memory location into register and write 1 into memory

• acquire: t&s register, location
  bnz register, acquire
  CS
release: st location, #0
Caching Locks

- Spin lock: to acquire a lock, a process may enter an infinite loop that keeps attempting a read-modify till it succeeds.

- If the lock is in memory, there is heavy bus traffic → other processes make little forward progress.

- Locks can be cached:
  - cache coherence ensures that a lock update is seen by other processors.
  - the process that acquires the lock in exclusive state gets to update the lock first.
  - spin on a local copy – the external bus sees little traffic.
SMP/UMA/Centralized Memory Multiprocessor
Coherence Traffic for a Lock

- If every process spins on an exchange, every exchange instruction will attempt a write → many invalidates and the locked value keeps changing ownership

- Hence, each process keeps reading the lock value – a read does not generate coherence traffic and every process spins on its locally cached copy

- When the lock owner releases the lock by writing a 0, other copies are invalidated, each spinning process generates a read miss, acquires a new copy, sees the 0, attempts an exchange (requires acquiring the block in exclusive state so the write can happen), first process to acquire the block in exclusive state acquires the lock, others keep spinning
Test-and-Test-and-Set

- lock: test register, location
  bnz register, lock
  t&s register, location
  bnz register, lock
  CS
  st location, #0
Load-Linked and Store Conditional

- LL-SC is an implementation of atomic read-modify-write with very high flexibility

- LL: read a value and update a table indicating you have read this address, then perform any amount of computation

- SC: attempt to store a result into the same memory location, the store will succeed only if the table indicates that no other process attempted a store since the local LL (success only if the operation was “effectively” atomic)

- SC implementations do not generate bus traffic if the SC fails – hence, more efficient than test&test&set
Spin Lock with Low Coherence Traffic

lockit:    LL         R2, 0(R1)    ; load linked, generates no coherence traffic
BNEZ    R2, lockit     ; not available, keep spinning
DADDUI R2, R0, #1 ; put value 1 in R2
SC         R2, 0(R1)   ; store-conditional succeeds if no one
               ; updated the lock since the last LL
BEQZ    R2, lockit    ; confirm that SC succeeded, else keep trying

• If there are i processes waiting for the lock, how many
  bus transactions happen?
Spin Lock with Low Coherence Traffic

lockit:    LL         R2, 0(R1)    ; load linked, generates no coherence traffic
BNEZ    R2, lockit     ; not available, keep spinning
DADDUI R2, R0, #1 ; put value 1 in R2
SC         R2, 0(R1)   ; store-conditional succeeds if no one
                    ; updated the lock since the last LL
BEQZ    R2, lockit    ; confirm that SC succeeded, else keep trying

• If there are i processes waiting for the lock, how many bus transactions happen?
  1 write by the releaser + i read-miss requests + i responses + 1 write by acquirer + 0 (i-1 failed SCs) + i-1 read-miss requests + i-1 responses
Further Reducing Bandwidth Needs

• Ticket lock: every arriving process atomically picks up a ticket and increments the ticket counter (with an LL-SC), the process then keeps checking the now-serving variable to see if its turn has arrived, after finishing its turn it increments the now-serving variable

•
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 ++;
}
What is really wrong with locks?

• Scalability
48-core AMD server
Exim collapse

![Graph showing throughput vs cores with a notable collapse at 40 cores highlighted.](image-url)
# Oprofile results

<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>

**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>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>

**48 cores: 4000 msg/sec**
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;
}
```
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

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 */
}
```

```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;
}
```

Allocate a 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;
}
```
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 */
    }
}

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 */
    }
}

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?
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
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;
}
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

The graph shows the throughput (messages/sec) against the number of cores for two lock mechanisms:
- Ticket lock
- MCS lock

The graph indicates that the throughput increases with the number of cores for both mechanisms, with a slight advantage for the MCS lock until a certain point, after which the throughput decreases significantly for the ticket lock.
Thank you!