

CS350: Operating Systems

Lecture 5: Synchronization

Ali Mashtizadeh

University of Waterloo

Outline

- ① Synchronization and memory consistency review
- ② C11 Atomics
- ③ Cache coherence – the hardware view
- ④ Deadlock
- ⑤ OS Implementation

Motivation

$$T(n) = T(1) \left(B + \frac{1}{n}(1 - B) \right)$$

- Amdahl's law
 - ▶ $T(1)$: the time one core takes to complete the task
 - ▶ B : the fraction of the job that must be serial
 - ▶ n : the number of cores
- Suppose n were infinity!
- Amdahl's law places an ultimate limit on parallel speedup
- Problem: synchronization increases serial section size
- Scalable Commutativity Rule: *"Whenever interface operations commute, they can be implemented in a way that scales"* [Clements]

Locking Basics

```
pthread_mutex_t m;  
  
pthread_mutex_lock(&m);  
cnt = cnt + 1; /* critical section */  
pthread_mutex_unlock(&m);
```

- Only one thread can hold a lock at a time
- Makes critical section atomic
- When do you need a lock?
 - ▶ Anytime two or more threads touch data and at least one writes
- Rule: Never touch data unless you hold the right lock

Fine-grained Locking

```
struct list_head *hash_tbl[1024];

/* Coarse-grained Locking */
mutex_t m;
mutex_lock(&m);
struct list_head *pos = hash_tbl[hash(key)];
/* walk list and find entry */
mutex_unlock(&m);

/* Fine-grained Locking */
mutex_t bucket[1024];
int index = hash(key);
mutex_lock(&bucket[index]);
struct list_head *pos = hash_tbl[index];
/* walk list and find entry */
mutex_unlock(&bucket[index]);
```

- Which of these is better?

Memory reordering danger

- Suppose no sequential consistency & don't compensate
- Hardware could violate program order

<u>Program order on CPU #1</u>	<u>View on CPU #2</u>
read/write: v->lock = 1; read: register = v->val; write: v->val = register + 1; write: v->lock = 0;	v->lock = 1; v->lock = 0; /* danger */ v->val = register + 1;

- If `atomic_inc` called at `/* danger */`, bad val ensues!

Ordering requirements

```
void atomic_inc (var *v) {  
    while (test_and_set (&v->lock))  
        ;  
    v->val++;  
    /* danger */  
    v->lock = 0;  
}
```

- Must ensure all CPUs see the following:
 1. v->lock was set *before* v->val was read and written
 2. v->lock was cleared *after* v->val was written
- How does #1 get assured on x86?
 - ▶ Recall test_and_set uses `xchgl %eax, (%edx)`
- How to ensure #2 on x86?

Ordering requirements

```
void atomic_inc (var *v) {  
    while (test_and_set (&v->lock))  
        ;  
    v->val++;  
    /* danger */  
    v->lock = 0;  
}
```

- Must ensure all CPUs see the following:
 1. v->lock was set *before* v->val was read and written
 2. v->lock was cleared *after* v->val was written
- How does #1 get assured on x86?
 - ▶ Recall test_and_set uses xchgl %eax, (%edx)
 - ▶ xchgl instruction always “locked,” ensuring barrier
- How to ensure #2 on x86?

Ordering requirements

```
void atomic_inc (var *v) {  
    while (test_and_set (&v->lock))  
        ;  
    v->val++;  
    asm volatile ("sfence" ::: "memory");  
    v->lock = 0;  
}
```

- Must ensure all CPUs see the following:
 1. v->lock was set *before* v->val was read and written
 2. v->lock was cleared *after* v->val was written
- How does #1 get assured on x86?
 - ▶ Recall test_and_set uses xchgl %eax, (%edx)
 - ▶ xchgl instruction always “locked,” ensuring barrier
- How to ensure #2 on x86?
 - ▶ Might need fence instruction after, e.g., non-temporal stores

MIPS Spinlocks

- LL rt, offset(rb) – Load Linked
 - ▶ $rt \leftarrow \text{memory}[rb+\text{offset}]$
- SC rt, offset(base) – Store conditional (sets rt 0 if not atomic)
 - ▶ if atomic w.r.t. prior LL $\text{memory}[rb+\text{offset}] \leftarrow rt$, $rt \leftarrow 1$
 - ▶ else $rt \leftarrow 0$

```
# spinlock_data_t spinlock_data_testandset(spinlock_data_t *sd)
ll    v0, 0(a0)      # v0 = *sd (Load Linked)
addi  t1, zero, 1    # t1 = 1
sc    t1, 0(a0)      # *sd = t1 (Store Conditional)
bne   t1, zero, 1f   # if SC not failed
nop                   # branch delay slot
addi  v0, zero, 1    # return 1 on failure
1: j   ra            # return to caller
nop                   # branch delay slot
```

- MIPS I (SYS/161) is sequentially consistent \rightarrow no barriers needed
- Later MIPS processors need SYNC memory barrier

OS/161 Spinlock Acquire

```
void spinlock_acquire(struct spinlock *lk)
{
    struct cpu *mycpu;

    splraise(IPL_NONE, IPL_HIGH);

    /* this must work before curcpu initialization */
    if (CURCPU_EXISTS()) {
        mycpu = curcpu->c_self;
        if (lk->lk_holder == mycpu) {
            panic("Deadlock on spinlock %p\n", lk);
        }
    } else {
        mycpu = NULL;
    }
    ...
}
```

OS/161 Spinlock Acquire Con't

```
void spinlock_acquire(struct spinlock *lk)
{
    ...
    while (1) {
        /*
         * First check if the lock is busy to reduce
         * coherence traffic (more on this later).
         */
        if (spinlock_data_get(&lk->lk_lock) != 0) {
            continue;
        }
        /* Attempt to acquire the lock */
        if (spinlock_data_testandset(&lk->lk_lock) != 0) {
            continue;
        }
        break;
    }
    lk->lk_holder = mycpu;
}
```


Outline

- 1 Synchronization and memory consistency review
- 2 **C11 Atomics**
- 3 Cache coherence – the hardware view
- 4 Deadlock
- 5 OS Implementation

Atomics and Portability

- Lots of variation in atomic instructions, consistency models, compiler behavior
- Results in complex code when writing portable kernels and applications
- Still a big problem today: Your laptop is x86, your cell phone is ARM
 - ▶ x86: Total Store Order Consistency Model, CISC
 - ▶ arm: Relaxed Consistency Model, RISC
- Fortunately, the new C11 standard has builtin support for atomics
 - ▶ Enable in GCC with the `-std=c11` flag
- Also available in C++11, but not discussed today...

- Portable support for synchronization
- New atomic type: e.g., `_Atomic(int) foo`
 - ▶ All standard ops (e.g., `+`, `-`, `/`, `*`) become sequentially consistent
 - ▶ Plus new intrinsics available (`cmpxchg`, atomic increment, etc.)
- `atomic_flag` is a special type
 - ▶ Atomic boolean value without support for loads and stores
 - ▶ Must be implemented lock-free
 - ▶ All other types might require locks, depending on the size and architecture
- Fences also available to replace hand-coded memory barrier assembly

Memory Ordering

- several choices available
 1. `memory_order_relaxed`: no memory ordering
 2. `memory_order_consume`
 3. `memory_order_acquire`
 4. `memory_order_release`
 5. `memory_order_acq_rel`
 6. `memory_order_seq_cst`: full sequential consistency
- What happens if the chosen model is mistakenly too weak? Too Strong?
- Suppose thread 1 **releases** and thread 2 **acquires**
 - ▶ Thread 1's preceding **writes** can't move past the **release** store
 - ▶ Thread 2's subsequent **reads** can't move before the **acquire** load
 - ▶ Warning: other threads might see a completely different order

Example 1: Atomic Counters

```
_Atomic(int) packet_count;  
  
void recv_packet(...) {  
    ...  
    atomic_fetch_add_explicit(&packet_count, 1,  
        memory_order_relaxed);  
    ...  
}
```

Example 2: Producer, Consumer

```
struct message msg_buf;
_Atomic(_Bool) msg_ready;

void send(struct message *m) {
    msg_buf = *m;
    atomic_thread_fence(memory_order_release);
    atomic_store_explicit(&msg_ready, 1,
        memory_order_relaxed);
}

struct message *recv(void) {
    _Bool ready = atomic_load_explicit(&msg_ready,
        memory_order_relaxed);
    if (!ready)
        return NULL;
    atomic_thread_fence(memory_order_acquire);
    return &msg_buf;
}
```

Example 3: A Spinlock

- Spinlocks are similar to Mutexes
- Kernel's use these for small critical regions
 - ▶ Busy wait for others to release the lock
 - ▶ No sleeping and yielding to other Threads

```
void spin_lock(atomic_flag *lock) {  
    while(atomic_flag_test_and_set_explicit(lock,  
        memory_order_acquire)) {}  
}
```

```
void spin_unlock(atomic_flag *lock) {  
    atomic_flag_clear_explicit(lock, memory_order_release);  
}
```

Outline

- ① Synchronization and memory consistency review
- ② C11 Atomics
- ③ Cache coherence – the hardware view
- ④ Deadlock
- ⑤ OS Implementation

- Coherence
 - ▶ concerns accesses to a single memory location
 - ▶ makes sure stale copies do not cause problems
- Consistency
 - ▶ concerns apparent ordering between multiple locations

Multicore Caches

- Performance requires caches
- Caches create an opportunity for cores to disagree about memory
- Bus-based approaches
 - ▶ “Snoopy” protocols, each CPU listens to memory bus
 - ▶ Use write through and invalidate when you see a write bits
 - ▶ Bus-based schemes limit scalability
- Modern CPUs use networks (e.g., hypertransport, UPI)
- Cache is divided into chunks of bytes called *cache lines*
 - ▶ 64-bytes is a typical size

3-state Coherence Protocol (MSI)

- Each cache line is one of three states:
- Modified (sometimes called Exclusive)
 - ▶ One cache has a valid copy
 - ▶ That copy is stale (needs to be written back to memory)
 - ▶ Must invalidate all copies before entering this state
- Shared
 - ▶ One or more caches (and memory) have a valid copy
- Invalid
 - ▶ Doesn't contain any data
- Transitions can take 100–2000 cycles

Core and Bus Actions

- Core has three actions:
- Read (load)
 - ▶ Read without intent to modify, data can come from memory or another cache
 - ▶ Cacheline enters shared state
- Write (store)
 - ▶ Read with intent to modify, must invalidate all other cache copies
 - ▶ Cacheline in shared (some protocols have an exclusive state)
- Evict
 - ▶ Writeback contents to memory if modified
 - ▶ Discard if in shared state

Implications for Multithreaded Design

- Lesson #1: Avoid false sharing
 - ▶ Processor shares data in cache line chunks
 - ▶ Avoid placing data used by different threads in the same cache line
- Lesson #2: Align structures to cache lines
 - ▶ Place related data you need to access together
 - ▶ Alignment in C11/C++11: `alignas(64) struct foo f;`
- Lesson #3: Pad data structures
 - ▶ Arrays of structures lead to false sharing
 - ▶ Add unused fields to ensure alignment
- Lesson #4: Avoid contending on cache lines
 - ▶ Reduce costly cache coherence traffic
 - ▶ Advanced algorithms spin on a cache line local to a core (e.g., MCS Locks)

Outline

- ① Synchronization and memory consistency review
- ② C11 Atomics
- ③ Cache coherence – the hardware view
- ④ **Deadlock**
- ⑤ OS Implementation

The deadlock problem

```
mutex_t m1, m2;
```

```
void f1(void *ignored) {  
    lock(m1);  
    lock(m2);  
    /* critical section */  
    unlock(m2);  
    unlock (m1);  
}
```

```
void f2 (void *ignored) {  
    lock(m2);  
    lock(m1);  
    /* critical section */  
    unlock(m1);  
    unlock(m2);  
}
```

- Lesson: Dangerous to acquire locks in different orders

More deadlocks

- Same problem with condition variables
 - ▶ Suppose resource 1 managed by c_1 , resource 2 by c_2
 - ▶ A has 1, waits on c_2 , B has 2, waits on c_1
- Or have combined mutex/condition variable deadlock:

```
mutex_t a, b;
```

```
cond_t c;
```

```
- lock(a); lock(b); while (!ready) wait(b, c);
```

```
unlock(b); unlock (a);
```

```
- lock(a); lock(b); ready = true; signal(c);
```

```
unlock(b); unlock(a);
```

- Lesson: Dangerous to hold locks when crossing abstraction barriers!
 - ▶ I.e., lock (a) then call function that uses condition variable

Deadlock conditions

1. Limited access (mutual exclusion):
 - ▶ Resource can only be shared with finite users
2. No preemption:
 - ▶ Once resource granted, cannot be taken away
3. Multiple independent requests (hold and wait):
 - ▶ Don't ask all at once
(wait for next resource while holding current one)
4. Circularity in graph of requests
 - All of 1–4 necessary for deadlock to occur
 - Two approaches to dealing with deadlock:
 - ▶ Pro-active: prevention
 - ▶ Reactive: detection + corrective action

Prevent by eliminating one condition

1. Limited access (mutual exclusion):
 - ▶ Buy more resources, split into pieces, or virtualize to make "infinite" copies
 - ▶ Threads: threads have copy of registers = no lock
2. No preemption:
 - ▶ Physical memory: virtualized with VM, can take physical page away and give to another process!
3. Multiple independent requests (hold and wait):
 - ▶ Wait on all resources at once (must know in advance)
4. **Circularity in graph of requests**
 - ▶ Single lock for entire system: (problems?)
 - ▶ Partial ordering of resources (next)

Cycles and deadlock

- View system as graph
 - ▶ Processes and Resources are nodes
 - ▶ Resource Requests and Assignments are edges
- If graph has no cycles → no deadlock
- If graph contains a cycle
 - ▶ Definitely deadlock if only one instance per resource
 - ▶ Otherwise, maybe deadlock, maybe not
- Prevent deadlock with partial order on resources
 - ▶ E.g., always acquire mutex m_1 before m_2
 - ▶ Statically assert lock ordering (e.g., VMware ESX)
 - ▶ Dynamically find potential deadlocks [[Witness](#)]

Outline

- ① Synchronization and memory consistency review
- ② C11 Atomics
- ③ Cache coherence – the hardware view
- ④ Deadlock
- ⑤ OS Implementation

Wait Channels

- OS locks (except spinlocks) use wait channels to manage sleeping threads
- `void wchan_sleep(struct wchan *wc);`
 - ▶ Blocks calling thread on wait channel `wc`
 - ▶ Causes a context switch (e.g., `thread_yield`)
- `void wchan_wakeall(struct wchan *wc);`
 - ▶ Unblocks all threads sleeping on the wait channel
- `void wchan_wakeone(struct wchan *wc);`
 - ▶ Unblocks one threads sleeping on the wait channel
- `void wchan_lock(struct wchan *wc);`
 - ▶ Lock wait channel operations
 - ▶ Prevents a race between sleep and wakeone

OS/161 Semaphores

```
P(struct semaphore *sem) {
    spinlock_acquire(&sem->sem_lock);
    while (sem->sem count == 0) {
        /* Locking the wchan prevents a race on sleep */
        wchan_lock(sem->sem wchan);
        /* Release spinlock before sleeping */
        spinlock_release(&sem->sem_lock);
        /* Wait channel protected by it's own lock */
        wchan_sleep(sem->sem wchan);
        /* Recheck condition, no locks held */
        spinlock_acquire(&sem->sem_lock);
    }
    sem->sem count--;
    spinlock_release(&sem->sem_lock);
}
```

```
V(struct semaphore *sem) {
    spinlock_acquire(&sem->sem_lock);
    sem->count++;
    wchan_wakeone(sem->sem wchan);
    spinlock_release(&sem->sem_lock);
}
```