Implementing Synchronization

- Topics
  - Implementing synchronization primitives
  - Hardware support for synchronization
- Learning Objectives:
  - Select appropriate hardware primitives to implement synchronization primitives.
  - Implement several different synchronization primitives

Spinlocks

- States of a spinlock:
  - Zero when unlocked
  - Non-zero when locked
- Proposed implementation:

```
1. while (lock_var != 0);
2. lock_var = 1;
```

Spinlocks: Race Condition!

- Proposed implementation:

```
1. while (lock_var != 0);
2. lock_var = 1;
```

Thread 1	Thread 2
Line1: lock_var == 0	
… descheduled …	Line 1: lock_var == 0
	Line 2: Sets lock_var = 1 (Thinks it has the lock.)
Line 2: Sets lock_var = 1 (Thinks it has the lock)	… descheduled …

Hardware to the Rescue

- Brute force: turn off interrupts
  - If a thread cannot be interrupted, then its operations are atomic, right?
  - What if you have multiple thread contexts, cores, or processors?
    - Would need to disable interrupts on ALL of them – BAD!!!
- Not quite so brute force: change *interrupt priority levels* (IPL)
  - Assign different interrupts different priority levels.
  - The faster a device, the higher its priority. (Why?)

Set Priority Level (SPL)

- Real operating systems define many priority levels
  - NetBSD: approximately 12:
    - lock, serial, sched, clock, statclock, vm, tty, softserial,net, bio, softnet, softclock
  - Linux: approximately 8:
    - bio, clock, imp, net, softclock, softtty, statclock, tty
- OS/161 has only two: low and high. The APIs you have are:
  - spl0: sets the IPL to 0 (interrupts on)
  - splhigh: sets the IPL to its highest value (interrupts off)
  - splx(s): sets IPL to whatever state S represents.
- Canonical use

```
s = splhigh();
Do stuff
splx(s);
```

Hardware Primitive: TAS

- Interrupts are a big hammer; we can do better with an atomic instruction.
- Test-and-set (TAS)
  - Provides an atomic instruction equivalent to:
    1. `return_value = lock_var;`
    2. lock_var = 1;

  - If return value is 0, then you succeeded in acquiring the test-and-set.
  - If return value is non-0, then you did not succeed.
  - How do you "unlock" a test-and-set?

- Test-and-set on Intel:

    ```
    xchg dest, src
    ```

  - Exchanges destination and source.
  - How do you use it?

Hardware Primitive: TAS

- Interrupts are a big hammer; we can do better with an atomic instruction.
- Test-and-set (TAS)
  - Provides an atomic instruction equivalent to:

```
1. return_value = lock_var;
2. lock_var = 1;
```

  - If return value is 0, then you succeeded in acquiring the test-and-set.
  - If return value is non-0, then you did not succeed.
  - How do you "unlock" a test-and-set?

- Test-and-set on Intel:

```
xchg dest, src
```

  - Exchanges destination and source.
  - How do you use it?

src = 1

```
xchg lock_var, src
```

If src == 0, you got the lock.

Hardware Primitive: LL/SC

- LL: load link (sticky load) returns the value in a memory location.
- SC: store conditional: stores a value to the memory location ONLY if that location hasn't changed since the last load-link.
- If update has occurred, store-conditional will fail.
- Example: LL/SC on the MIPS (register 1 contains address of the spinlock)

```
LL r2,(r1) # Load value ref'd by r1 into r2
if r2 is 0 (i.e., unlocked)
        SC r3,(r1) # Store "locked" into loc ref'd by r1
        # r3 contains 0 on failure
if (r2 is non-zero OR r3 is 0)
        goto retry
```

Hardware Primitive: CAS

- Compare and Swap
- Compares the contents of a memory location with a value and if they are the same, then modifies the memory location to a new value.
- CAS on Intel:

```
cmpxchg loc, val
```

- Compare value stored at memory location `loc` to contents of the Compare Value Application Register.
  - If they are the same, then set loc to val.
  - ZF flag is set if the compare was true, else ZF is 0

Using CAS

- Set Compare Value Application Register to 0
- Let “loc” be the address of the memory location of your spinlock.

```
cmpxchg loc, 1
```

- Check ZF flag:
  - If ZF is 1, then the compare was true and you have the lock
  - If ZF is 0, then you failed and should retry.

Fancier Hardware Support: Transactional memory

- Introduced by Herlihy and Moss in 1993.
- Finally starting to get some traction in the past few years.
- Idea:
  - Implement an entire critical section exploiting hardware to make it atomic.
  - Code up the set of operations you want and then "try" to apply them all at once atomically -- that will either succeed or fail.
- Specify a set of "transactional operations”
  - load-transactional (LT): read memory into a register
  - load-transactional-exclusive (LTX): read memory into a register and hint that you’ll be updating it (optimization)
  - store-transactional (ST): write value into a memory location
- Specify a set of transaction control instructions
  - begin: start a sequence of atomic instructions
  - commit: try to apply all the updates from the transaction. If possible, apply them and the transaction succeeds. If not possible, apply none and transaction fails.
  - abort: throw away all the current transactional changes.
  - validate: check if this transaction has aborted.

Implementing Transactional Memory

- Maintain a *read-set*: set of all memory locations read during a transaction (all locations accessed by LT).
- Maintain a *write-set*: set of all memory locations written during a transaction (all locations accessed by LTX and ST).
- *Data-set* is the union of read-set and write-set.
- Commit check that:
  - no other transaction has modified any item in this transaction's data set.
  - no other transaction has read anything in this transaction's write set.
- If commit check fails, restore everything to its initial state.

Uniprocessor Semaphores using SPL (1)

```
struct semaphore {
    char *name;
    volatile int count;
}
```

Uniprocessor Semaphores using SPL (2)

```
P (struct semaphore *sem)
{
        int spl;

        while (1) {
                spl = splhigh();
                if (sem->count > 0)
                        break;
                thread_sleep(sem);
                splx(spl);
        }

        sem->count--;
        splx(spl);
        return;
}
```

Uniprocessor Semaphore using SPL (3)

```
V(struct semaphore *sem)
{
    int spl;

    spl = splhigh();
    sem->count++;
    thread_wakeup(sem);
    splx(spl);
}
```

Multiprocessor: Attempt (1)

- Let's start with a uniprocessor solution and add a TAS to protect the count.

```
struct semaphore {
    char *name;
    volatile int count;
    volatile int tas;
}
```

Multiprocessor Attempt (2)

```
V(struct semaphore *sem)
{
    int spl;

    spl = splhigh();
    while (TAS(sem->tas != 0)); /* spin */
    sem->count++;
    sem->tas = 0;
    thread_wakeup(sem);
    splx(spl);
}
```

Multiprocessor Attempt: (3)

```
P (struct semaphore *sem)
{
        int spl;
        spl = splhigh();
        while (1) {
                spl = splhigh();
                while (TAS(sem->tas) != 0);
                if (sem->count > 0)
                        break;
                sem->tas = 0;
                thread_sleep(sem);
                splx(s);
        }
        sem->count--;
        sem->tas = 0;
        splx(spl);
        return;
}
```

Multiprocessor Attempt: (4)

```
P (struct semaphore *sem)
{
        int spl;
        spl = splhigh();
        while (1) {
                spl = splhigh();
                while (TAS(sem->tas) != 0);
                if (sem->count > 0)
                        break;
                sem->tas = 0;
                thread_sleep(sem);
                splx(s);
        }
        sem->count--;
        sem->tas = 0;
        splx(spl);
        return;
}
```

If another processor sets the semaphore here and does the Wakeup right away, this thread will never be woken up.

Multiprocessor Semaphore (1)

- This is the OS/161 implementation (formatted differently to fit on slides)

```
struct semaphore {
        char *sem_name;
        struct wchan *sem_wchan;
        struct spinlock sem_lock;
        volatile int sem_count;
};
```

Multiprocessor Semaphore (2)

```
struct semaphore *sem_create(const char *name, int initial_count)
{
        struct semaphore *sem;
        if ((sem = kmalloc(sizeof(struct semaphore))) == NULL)
                return (NULL);
        if ((sem->sem_name = kstrdup(name)) == NULL) {
                kfree(sem);
                return (NULL);
        }
        if ((sem->sem_wchan = wchan_create(sem->sem_name)) == NULL) {
                kfree(sem->sem_name);
                kfree(sem);
                return(NULL);
        }
        spinlock_init(&sem->sem_lock);
        sem->sem_count = initial_count;
        return (sem);
}
```

Multiprocessor Semaphore (3)

```
void V(struct semaphore *sem)
{
        KASSERT(sem != NULL);

        spinlock_acquire(&sem->sem_lock);

        sem->sem_count++;
        KASSERT(sem->sem_count > 0);
        wchan_wakeone(sem->sem_wchan);

        spinlock_release(&sem->sem_lock);
}
```

Multiprocessor Semaphore (4)

```
void P(struct semaphore *sem)
{
        KASSERT(sem != NULL);
        KASSERT(curthread->t_in_interrupt == false);
        spinlock_acquire(&sem->sem_lock);
        while (sem->sem_count == 0) {
                wchan_lock(sem->sem_wchan);
                spinlock_release(&sem->sem_lock);
                wchan_sleep(sem->sem_wchan);

                spinlock_acquire(&sem->sem_lock);
        }
        KASSERT(sem->sem_count > 0);
        sem->sem_count--;
        spinlock_release(&sem->sem_lock);
}
```

Multiprocessor Semaphore (5)

```
void P(struct semaphore *sem)
{
        KASSERT(sem != NULL);
        KASSERT(curthread->t_in_interrupt == false);
        spinlock_acquire(&sem->sem_lock);
        while (sem->sem_count == 0) {
                wchan_lock(sem->sem_wchan);
                spinlock_release(&sem->sem_lock);
                wchan_sleep(sem->sem_wchan);

                spinlock_acquire(&sem->sem_lock);
        }
        KASSERT(sem->sem_count > 0);
        sem->sem_count--;
        spinlock_release(&sem->sem_lock);
}
```

Make sure we never block in a signal handler

Multiprocessor Semaphore (6)

```
void P(struct semaphore *sem)
{
        KASSERT(sem != NULL);
        KASSERT(curthread->t_in_interrupt == false);
        spinlock_acquire(&sem->sem_lock);
        while (sem->sem_count == 0) {
                wchan_lock(sem->sem_wchan);
                spinlock_release(&sem->sem_lock);
                wchan_sleep(sem->sem_wchan);

                spinlock_acquire(&sem->sem_lock);
        }
        KASSERT(sem->sem_count > 0);
        sem->sem_count--;
        spinlock_release(&sem->sem_lock);
}
```

Make sure we never block in a signal handler

Note that we do not maintain strict FIFO ordering of threads going through the semaphore; that is, we might "get" it on the first try even if other threads are waiting

Wait Channels

- An abstraction that lets a thread wait on a certain event.
- Includes a lock and a queue.
- Does this sound like a familiar abstraction to you?
- Homework: Figure out how a wait channel differs from a CV.