Recap: Synchronization

Disabling IRQs – use to protect against concurrent access by IRQ handler

Locks – use to protect against concurrent access by other threads

Direct implementation of locks on uniprocessor

- Requires disable_preemption
- Involves state change of thread if contended

Today: multiprocessor locks, locking strategies

Multiprocessor Locks

Can't stop threads running on other processors

- too expensive (interprocessor irq)
- also would create conflict with protection (locking = unprivileged op, stopping = privileged op), involving the kernel in *every* acquire/release

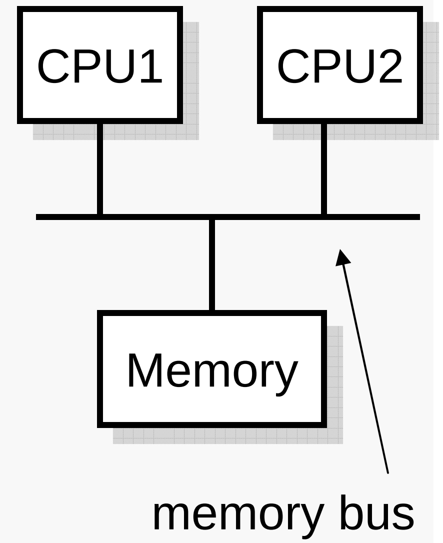
Instead: use atomic instructions provided by hardware

- E.g.: test-and-set, atomic-swap, compare-and-exchange, fetch-and-add
- All variations of "read-and-modify" theme

Locks are built on top of these

Atomic Swap


```
// In C, an atomic swap instruction would like this
void
atomic_swap(int *memory1, int *memory2)
{
        [ disable interrupts in CPU;
          lock memory bus for other processors ]
        int tmp = *memory1;
        *memory1 = *memory2;
        *memory2 = tmp;
         [ unlock memory bus; reenable interrupts ]
}
```



Spinlocks

```
lock_acquire(struct lock *l)
{
    int lockstate = LOCKED;
    while (lockstate == LOCKED) {
      atomic_swap(&lockstate,
                          &l->state);
    }
}
```

```
lock_release(struct lock *l)
{
   l->state = UNLOCKED;
}
```

Thread spins until it acquires lock

- Q1: when should it block instead?
- Q2: what if spin lock holder is preempted?

Spinning vs Blocking

Blocking has a cost

- Shouldn’t block if lock becomes available in less time than it takes to block

Strategy: spin for time it would take to block

- Even in worst case, total cost for lock_acquire is less than 2*block time

Spinlocks vs Disabling Preemption

What if spinlocks were used on single CPU? Consider:

- thread 1 takes spinlock
- thread 1 is preempted
- thread 2 with higher priority runs
- thread 2 tries to take spinlock, finds it taken
- thread 2 spins forever $\rightarrow$ deadlock!

Thus in practice, usually combine spinlocks with disabling preemption

- E.g., spin_lock_irqsave() in Linux
  - UP kernel: reduces to disable_preemption
  - SMP kernel: disable_preemption + spinlock

Spinlocks are used when holding resources for small periods of time (same rule as for when it's ok to disable irqs)

Spinlocks (Faster)


```
lock_acquire(struct lock *l)
{
    int lockstate = LOCKED;
    while (lockstate == LOCKED) {
        while (l->state == LOCKED)
            continue;
        atomic_swap(&lockstate,
                    &l->state);
    }
}
```

```
lock_release(struct lock *l)
{
    l->state = UNLOCKED;
}
```

Only try “expensive” atomic_swap instruction if you’ve seen lock in unlocked state

Locks: Ownership & Recursion

Locks typically (not always) have notion of ownership

- Only lock holder is allowed to unlock
- See Pintos lock_held_by_current_thread()

What if lock holder tries to acquire locks it already holds?

- Nonrecursive locks: deadlock!
- Recursive locks:
  - inc counter
  - dec counter on lock_release
  - release when zero

Implementing Locks: Practical Issues

How expensive are locks?

Two considerations:

- Cost to acquire uncontended lock
  - UP Kernel: disable/enable irq + memory access
  - In other scenarios: needs atomic instruction (relatively expensive in terms of processor cycles, especially if executed often)
- Cost to acquire contended lock
  - Spinlock: blocks current CPU entirely (if no blocking is employed)
  - Regular lock: cost at least two context switches, plus associated management overhead

Conclusions

- Optimizing uncontended case is important
- “Hot locks” can sack performance easily

Using Locks

Associate each shared variable with lock L

- “lock L protects that variable”

```
static struct list usedlist; /* List of used blocks */
static struct list freelist;  /* List of free blocks */

static struct lock listlock; /* Protects usedlist & freelist */
```

```
void *mem_alloc(...)
{
    block *b;
    lock_acquire(&listlock);
    b = alloc_block_from_freelist();
    insert_into_usedlist(&usedlist, b);
    lock_release(&listlock);
    return b->data;
}
```

```
void mem_free(block *b)
{
    lock_acquire(&listlock);
    list_remove(&b->elem);
    coalesce_into_freelist(&freelist, b);
    lock_release(&listlock);
}
```

How many locks should I use?

Could use one lock for all shared variables

- Disadvantage: if a thread holding the lock blocks, no other thread can access *any* shared variable, even unrelated ones
- Sometimes used when retrofitting non-threaded code into threaded framework
- Examples:
  - "BKL" Big Kernel Lock in Linux
  - fslock in Pintos Project 2

Ideally, want fine-grained locking

- One lock only protects one (or a small set of) variables – how to pick that set?

Multiple locks, the wrong way


```
static struct list usedlist; /* List of used blocks */
static struct list freelist;  /* List of free blocks */

static struct lock alloclock; /* Protects allocations */
static struct lock freelock; /* Protects deallocations */
```

```
void *mem_alloc(...)
{
    block *b;
    lock_acquire(&alloclock);
    b = alloc_block_from_freelist();
    insert_into_usedlist(&usedlist, b);
    lock_release(&alloclock);
    return b->data;
}
```

```
void mem_free(block *b)
{
    lock_acquire(&freelock);
    list_remove(&b->elem);
    coalesce_into_freelist(&freelist, b);
    lock_release(&freelock);
}
```

Wrong: locks protect data structures, not code blocks! Allocating thread & deallocating thread could collide

Multiple locks, 2nd try

```
static struct list usedlist; /* List of used blocks */
static struct list freelist;  /* List of free blocks */

static struct lock usedlock; /* Protects usedlist */
static struct lock freelock;  /* Protects freelist */
```

```
void *mem_alloc(...)
{
    block *b;
    lock_acquire(&freelock);
    b = alloc_block_from_freelist();
    lock_acquire(&usedlock);
    insert_into_usedlist(&usedlist, b);
    lock_release(&freelock);
    lock_release(&usedlock);
    return b->data;
}
```

```
void mem_free(block *b)
{
    lock_acquire(&usedlock);
    list_remove(&b->elem);
    lock_acquire(&freelock);
    coalesce_into_freelist(&freelist, b);
    lock_release(&usedlock);
    lock_release(&freelock);
}
```

Also wrong: deadlock!
Always acquire multiple locks in same order -
Or don't hold them simultaneously

Multiple locks, correct (1)

```
static struct list usedlist; /* List of used blocks */
static struct list freelist;  /* List of free blocks */

static struct lock usedlock; /* Protects usedlist */
static struct lock freelock;  /* Protects freelist */
```

```
void *mem_alloc(...)
{
    block *b;
    lock_acquire(&usedlock);
    lock_acquire(&freelock);
    b = alloc_block_from_freelist();
    insert_into_usedlist(&usedlist, b);
    lock_release(&freelock);
    lock_release(&usedlock);
    return b->data;
}
```

```
void mem_free(block *b)
{
    lock_acquire(&usedlock);
    lock_acquire(&freelock);
    list_remove(&b->elem);
    coalesce_into_freelist(&freelist, b);
    lock_release(&freelock);
    lock_release(&usedlock);
}
```

Correct, but inefficient!
Locks are always held simultaneously,
one lock would suffice

Multiple locks, correct (2)


```
static struct li
static struct li

static struct lo
static struct lo
```

Correct, but not necessarily better!

On uniprocessor:

No throughput from fine-grained locking, since no blocking inside critical sections – but pay twice the price compared to one-lock solution

On multiprocessor:

Gain from being able to manipulate free & used lists in parallel, but increased risk of contended locks

```
void *mem_alloc(...)
{
    block *b;
    lock_acquire(&freelock);
    b = alloc_block_from_freelist();
    lock_release(&freelock);
    lock_acquire(&usedlock);
    insert_into_usedlist(&usedlist, b);
    lock_release(&usedlock);
    return b->data;
}
```

```
void mem_free(block *b)
{
    lock_acquire(&usedlock);
    list_remove(&b->elem);
    lock_release(&usedlock);
    lock_acquire(&freelock);
    coalesce_into_freelist(&freelist, b);
    lock_release(&freelock);
}
```

Conclusion

Choosing which lock should protect which shared variable(s) is not easy – must weigh:

- Whether all variables are always accessed together (use one lock if so)
- Whether code inside critical section can block (if not, no throughput gain from fine-grained locking on uniprocessor)
- Whether there is a consistency requirement if multiple variables are accessed in related sequence (must hold single lock if so)
  - See “Subtle race condition in Java” below
- Cost of multiple calls to lock/unlock (increasing parallelism advantages may be offset by those costs)

