When critical section is very short

FINE-GRAINED SYNCHRONIZATION
Agenda: mutex locks

- Locks
  - Why do we need them?
  - How do we implement them? (In xv6 in particular)
  - How do we use them?

Why locks?

- The point of multiprocessors / multicores?
  - Goal: increase performance
  - Means: running different tasks concurrently

- Problem
  - Simultaneous use of the same data structure(s) might cause incorrect results

- Solution
  - Synchronize & coordinate
    - So as to serialize access where needed
  - To do that, we will use spinlocks
Example

```
struct List {
    int data;
    struct List *next;
};

List *list = 0;

insert(int data) {
    List *l = new List;
    l->data = data;
    l->next = list;
    list = l;
}
```

- Works correctly when serial
- Breaks when two cores doing it simultaneously

Race condition

Core #1:

```
insert(int data) { // data=3
    List *l = new List;
    l->data = data;
    l->next = list; // = null
    list = l; // list = [3 -> null]
}
```

Core #2:

```
insert(int data) { // data=2
    List *l = new List;
    l->data = data;
    l->next = list; // = null
    list = l; // list = [2 -> null]
}
```
Race condition

- Why is it called a “race”?
- Nondeterministic nature
  - Debug printing might
    - Eliminate the race, or introduce a new one...
    - This nondeterministic behavior makes it very **hard to debug**
- How to avoid races?
  - Introduce a lock to protect a collection of *invariants*
    - Data structure properties that are maintained across ops
    - For linked list:
      - ‘list’ points to head + each ‘next’ points to next element
    - Op might temporarily violate invariants
      - Which is why we need it to be “*atomic*”

```c
insert(int data) {
    acquire( &list_lock );
    List *l = new List;
    l->data = data;
    l->next = list;
    list = l;
    release( &list_lock )
}
```

- **Serialize with lock abstraction**
- **Associate a lock object with the list**
- **2 ops + 2 matching states:**
  1. acquire => “locked”
  2. release => “unlocked”
- **Semantics:**
  - Only one thread can ‘acquire’ lock
  - Other threads must wait until it’s ‘release’-d
Serialize with lock abstraction

insert(int data) {
    List *l = new List;
    l->data = data;

    acquire( &list_lock );

    l->next = list;
    list = l;

    release( &list_lock )
}

Would this work?

Why would we want to do it?

LET’S TRY TO IMPLEMENT THE ABSTRACTION
Acquire & Release (broken)

```c
struct Lock {
    int locked; // 0=free; 1=taken
};

void broken_acquire(Lock *lock) {
    while(1) {
        if(lock->locked == 0) {
            lock->locked = 1;
            break;
        }
    }
}

void release (Lock *lock) {
    lock->locked = 0;
}
```

Can be acquired simultaneously (or TOCTTOU).
We want these 2 lines to be atomic.
**Atomic instructions**

- **Most CPUs provide atomic operations**
  - Often doing exactly what we needed in the previous slide

- **In x86**
  - `xchng %eax, addr`
  - Is equivalent to
    1. freeze any memory activity for address `addr`
    2. `temp := *addr`
    3. `*addr := %eax`
    4. `%eax = temp`
    5. un-freeze other memory activity

```c
void release (Lock *lock) {
    xchg(&lock->locked, 0);
    // why not do:
    //     lock->locked = 0
    // ?
    // [see next slide…]
}
```

```c
void acquire(Lock *lock) {
    while(1)
        if( xchg(&lock->locked, 1) == 0 )
            break;
}
```

```c
int xchg(addr, value) {
    %eax = value
    xchg %eax, (addr)
    return %eax
}
```

```c
int xchg(addr, value) {
    %eax = value
    xchg %eax, (addr)
    return %eax
}
```
Memory ordering:
(“memory fence” aka “memory barrier”)

```c
insert(int data) {
    acquire( &list_lock ); // 0
    List *l = new List;
    l->data = data;
    l->next = list; // 1
    list = l; // 2
    release( &list_lock )
}
```

- Both processor & compiler can reorder operations for out-of-order execution, to maximize ILP
- Must make sure, e.g., that lines 1–2 executed by thread 1 are not reorder with line 0 of thread 2
- xchg serves as a full memory “fence” or “barrier”
  - Full fence: all reads/writes prior to fence occur before any reads/writes after fence
  - Partial: all write before fence occur before any reads after
  - (also, xchg is “asm volatile”)

Spin or block?

- **Constructs like while(1) \{ xchg \} are called**
  - “Busy waiting” or “spinning”
  - Indeed, the lock we’ve used is called a “spinlock”
- **Not advisable when the critical section is long**
  - (critical section = everything between the acquire-release)
  - Because other threads can utilize the core instead
- **The alternative: blocking**
  - In xv6: sleep() & wakeup()
  - Not advisable when wait is short (otherwise induces overhead)
- **There’s also two-phase waiting**
  - The “ski problem”
- **Note that kernel & processes are different in this context**
  - Kernels can hold lock with interrupts disables
  - Sleeping might not make sense (or be possible) for kernel
Locking granularity

- How much code should we lock?
- Once, Linux had a “big lock” (for the entire kernel!)
  - (Completely removed only in 2011…)
- What’s the appropriate locking granularity for file-system:
  - All of it with a single lock? Every directory? File? Block?
- Coarse granularity:
  - ☑ Pros: simple, easier, tends to be more correct…
  - ☠ Cons: performance degradation, negates concurrency
- Fine granularity:
  - ☑ If done right, allows for concurrency most of the time
  - ☠ Messes up the code, hard to get right, hard to debug
- Xv6
  - ☑ To simplify, tends to favor coarse (but no “kernel big lock”)
  - ☑ E.g., locking entire process table instead of individual entries

Lock ordering

- When utilizing more than one lock
  - ☑ All code paths must acquire locks in same order
  - ☑ Otherwise => deadlock!

- Xv6
  - ☑ Simple / coarse => not a lot of locks held together
  - ☑ Longest chain is 2, e.g.,
    - When removing a file: lock dir => lock file (in that order)
    - ideinter() / ide lock => wakeup / ptable lock
    - …
Synchronizing with interrupts

- A core uses locks to synchronize with other cores
  - (interrupt + non-interrupt context)

- A core must also synchronize with itself
  - Namely, its non-interrupt + interrupt code
  - Both are essentially different threads of execution that might manipulate the same data
  - Example: both of the following use tickslock (3063)
    - tick handler (timer interrupt; 3116), and
    - sys_sleep (3473)

- xv6 solution (again, favors simplicity over performance)
  - Disable interrupts while holding a lock
  - Thus, interrupts will never be processed with a lock
  - Which ensures no deadlocks due to interrupts

xv6 locking code

- Pages 14 – 15
Locks mess up modularity

◆ In principle…
  ▶ When calling function F, best if we don’t know if it uses a lock
  ▶ Supposedly should be “internal details”
◆ Alas, in the kernel, it oftentimes doesn’t work that way
  ▶ Performance is paramount
    □ Typically not trading off elegance for performance
  ▶ If I hold lock L and invoke F, it must not attempt to acquire L
  ▶ If F acquires L, I must not attempt to call it while holding L
  ⊗ => Locks break modularity and contaminate the OS code
  ⊗ => Consider locks as part of the function specification
    □ “call this function while holding L…”
    □ “if that function returns -2 release L2…”
    □ Ordering of locks needs to be know too

Real world

◆ Locking is hard
  ▶ Despite years of experience, it’s still very challenging!
  ▶ Deadlocks are common when developing even for experts
◆ Our spinlock, with xchg, keeps memory busy => suboptimal
  ▶ We can sometimes do better (see later lecture)
  ▶ RCU (read-copy-update) optimizes for mostly reading shared state; heavily used in Linux
◆ Synchronization is an active field of research
  ▶ Transactional memory (via software, hardware, or both)
  ▶ Lock-free data structures
  ▶ A newly proposed research OS (2010) that is always deterministic (see later lecture)
  ▶ Many papers about automatically identifying races
END

In remaining time (~20 mins)

- Memory management stuff from the last lecture we didn’t have time to finish
- Or barrier sync presentation
- Or next week’s lecture