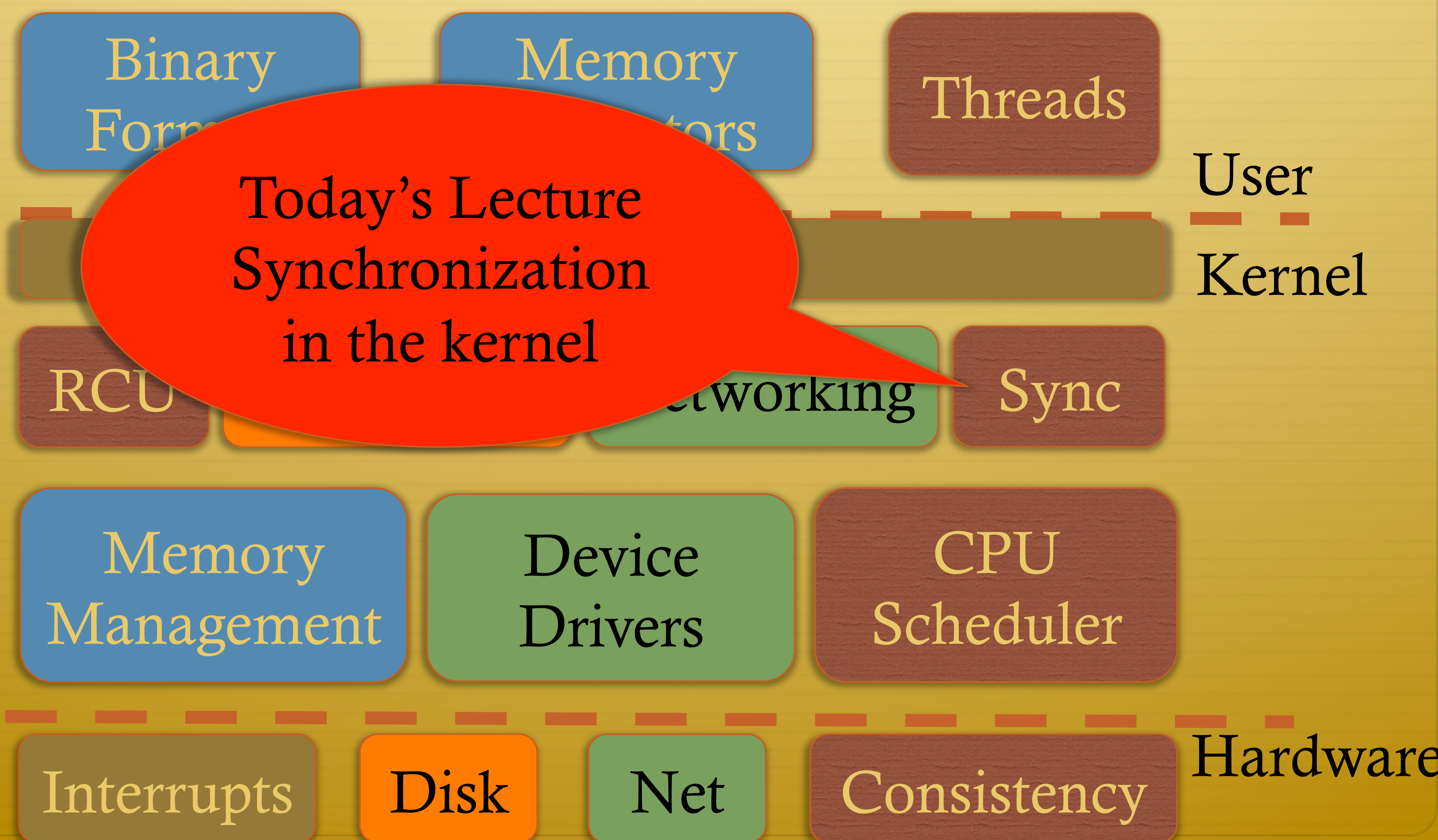
Linux kernel synchronization

Don Porter
CSE 506

Logical Diagram



Warm-up

- What is synchronization?
  - Code on multiple CPUs coordinate their operations
- Examples:
  - Locking provides mutual exclusion while changing a pointer-based data structure
  - Threads might wait at a barrier for completion of a phase of computation
  - Coordinating which CPU handles an interrupt

Why Linux synchronization?

- A modern OS kernel is one of the most complicated parallel programs you can study
  - Other than perhaps a database
- Includes most common synchronization patterns
  - And a few interesting, uncommon ones

Historical perspective

- Why did OSes have to worry so much about synchronization back when most computers have only one CPU?

The old days: They didn't worry!

- Early/simple OSes (like JOS, pre-lab4): No need for synchronization
  - All kernel requests wait until completion – even disk requests
  - Heavily restrict when interrupts can be delivered (all traps use an interrupt gate)
  - No possibility for two CPUs to touch same data

Slightly more recently

- Optimize kernel performance by blocking inside the kernel
- Example: Rather than wait on expensive disk I/O, block and schedule another process until it completes
  - Cost: A bit of implementation complexity
    - Need a lock to protect against concurrent update to pages/inodes/etc. involved in the I/O
    - Could be accomplished with relatively coarse locks
    - Like the Big Kernel Lock (BKL)
  - Benefit: Better CPU utilitzation

A slippery slope

- We can enable interrupts during system calls
  - More complexity, lower latency
- We can block in more places that make sense
  - Better CPU usage, more complexity
- Concurrency was an optimization for really fancy OSes, until…

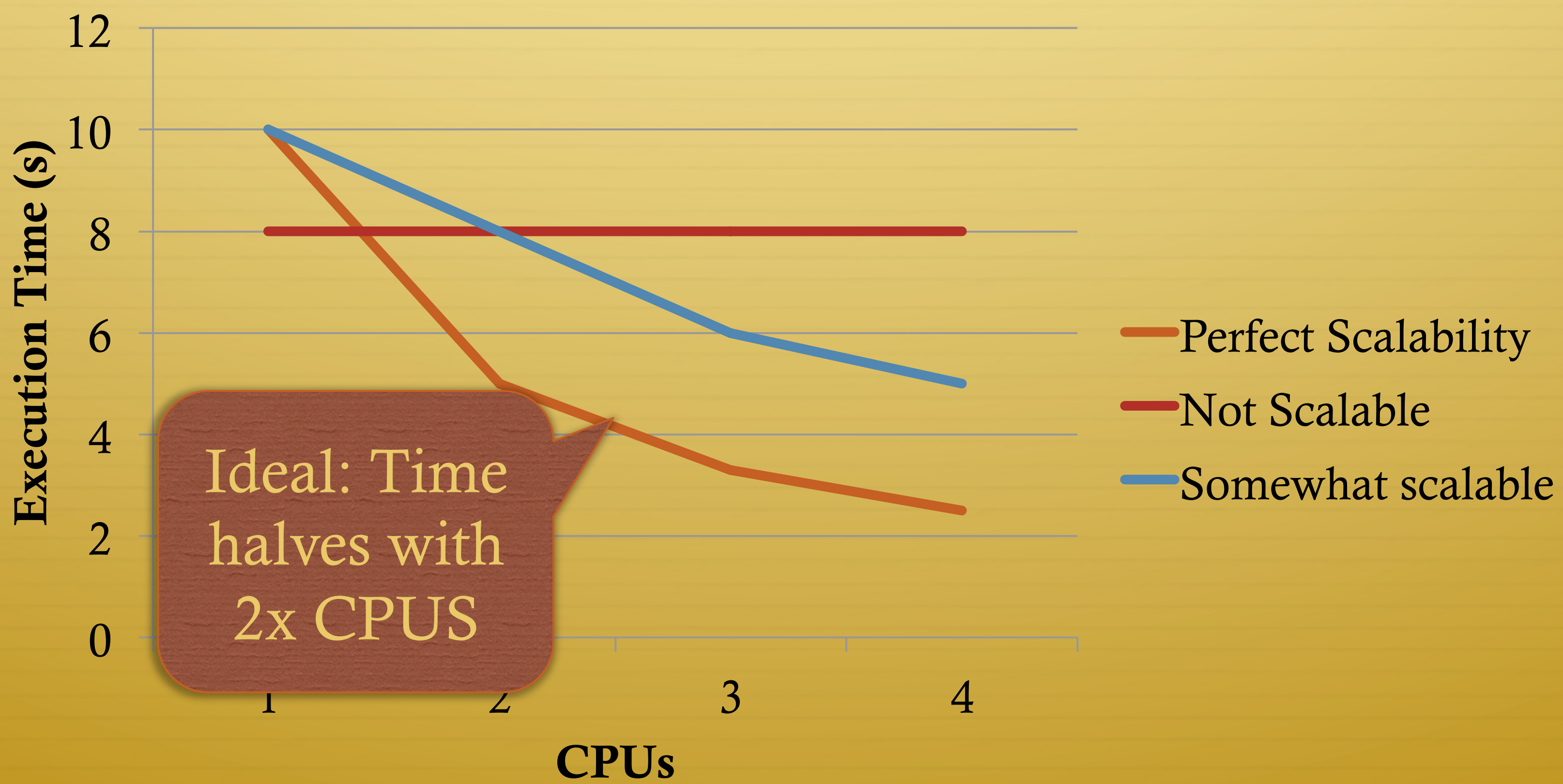
The forcing function

- Multi-processing
  - CPUs aren't getting faster, just smaller
  - So you can put more cores on a chip
- The only way software (including kernels) will get faster is to do more things at the same time

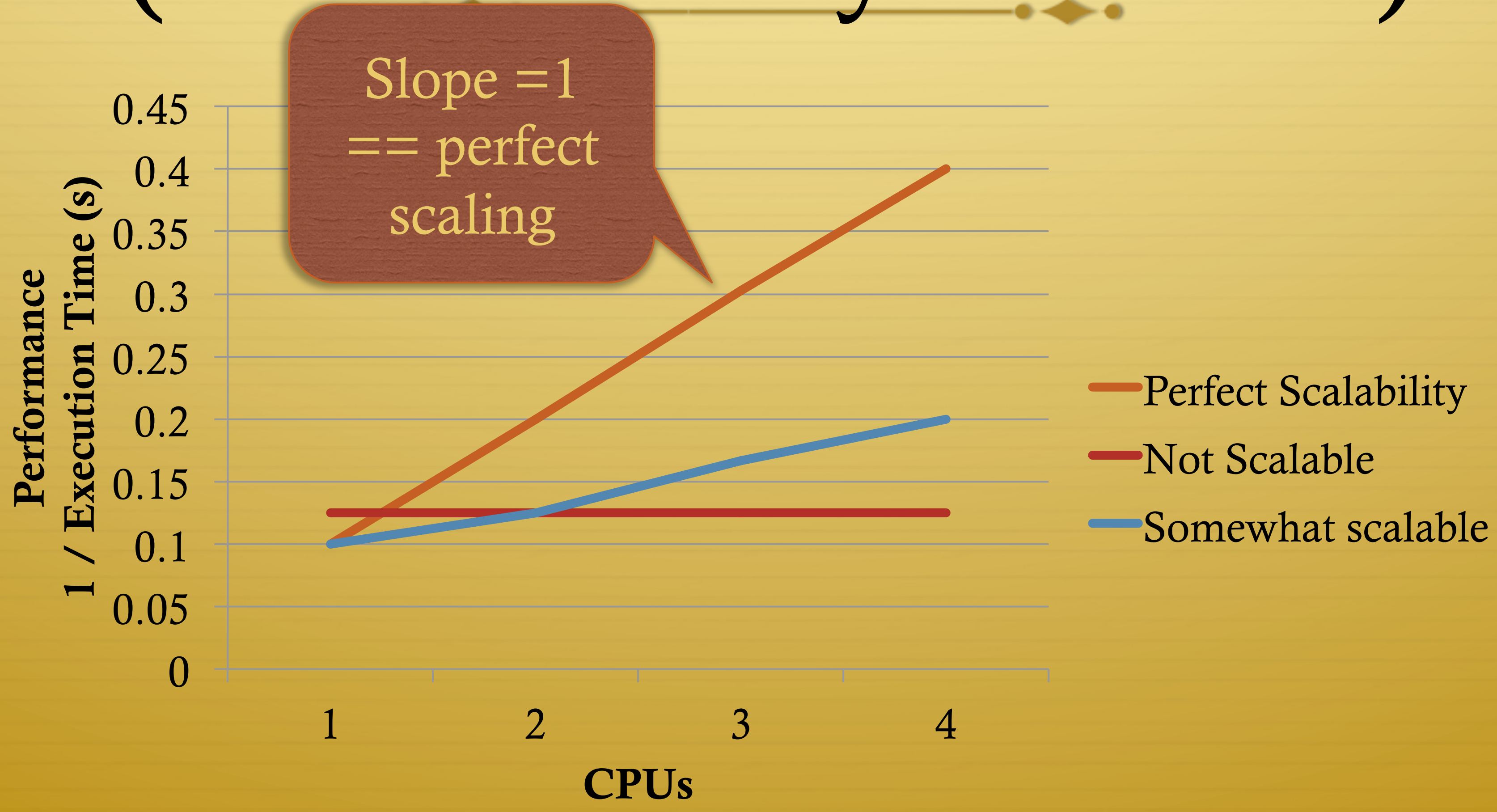
Performance Scalability

- How much more work can this software complete in a unit of time if I give it another CPU?
  - Same: No scalability---extra CPU is wasted
  - 1 -> 2 CPUs doubles the work: Perfect scalability
- Most software isn’t scalable
- Most scalable software isn’t perfectly scalable

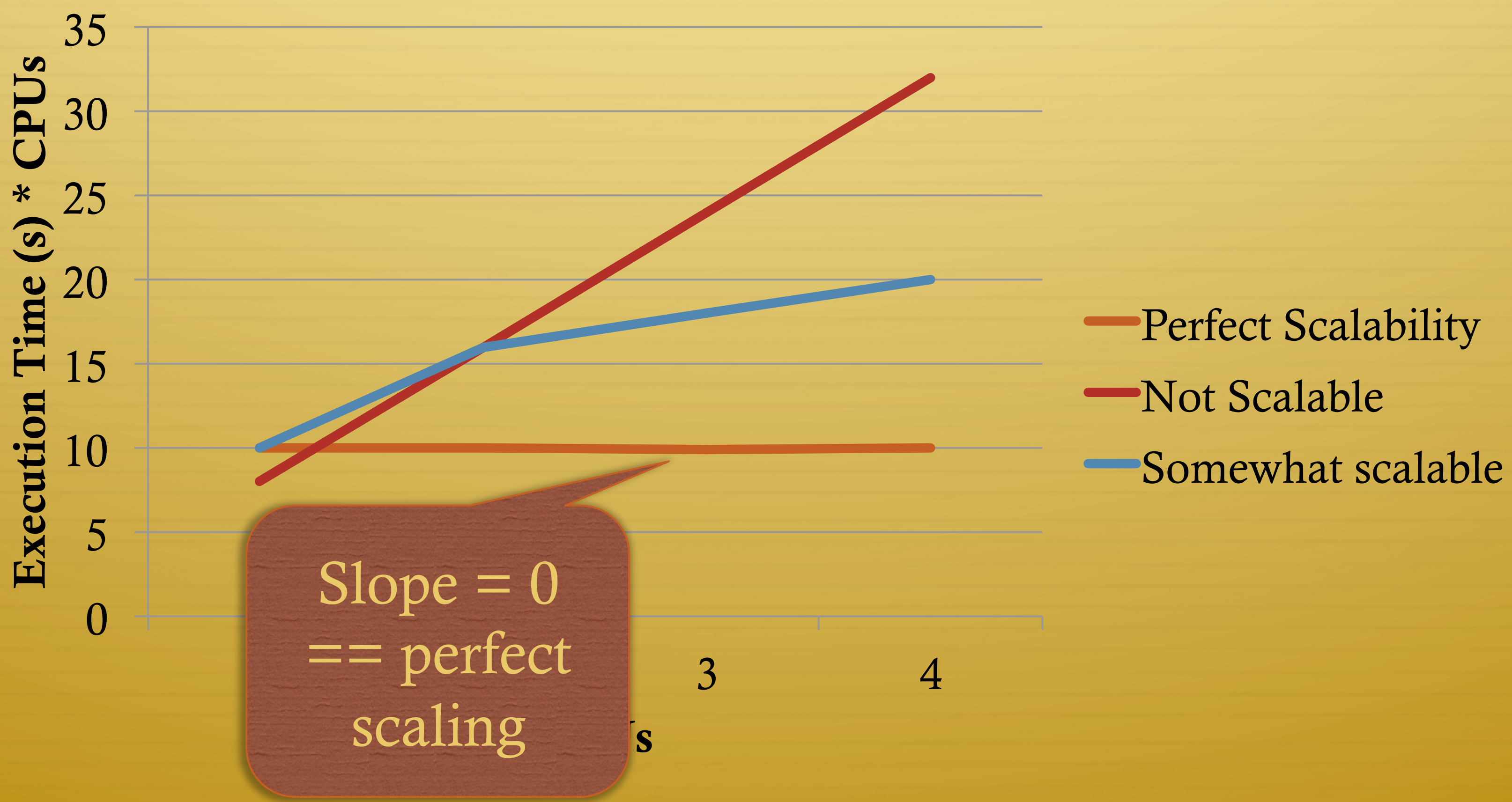
Performance Scalability



Performance Scalability (more visually intuitive)



Performance Scalability (A 3rd visual)



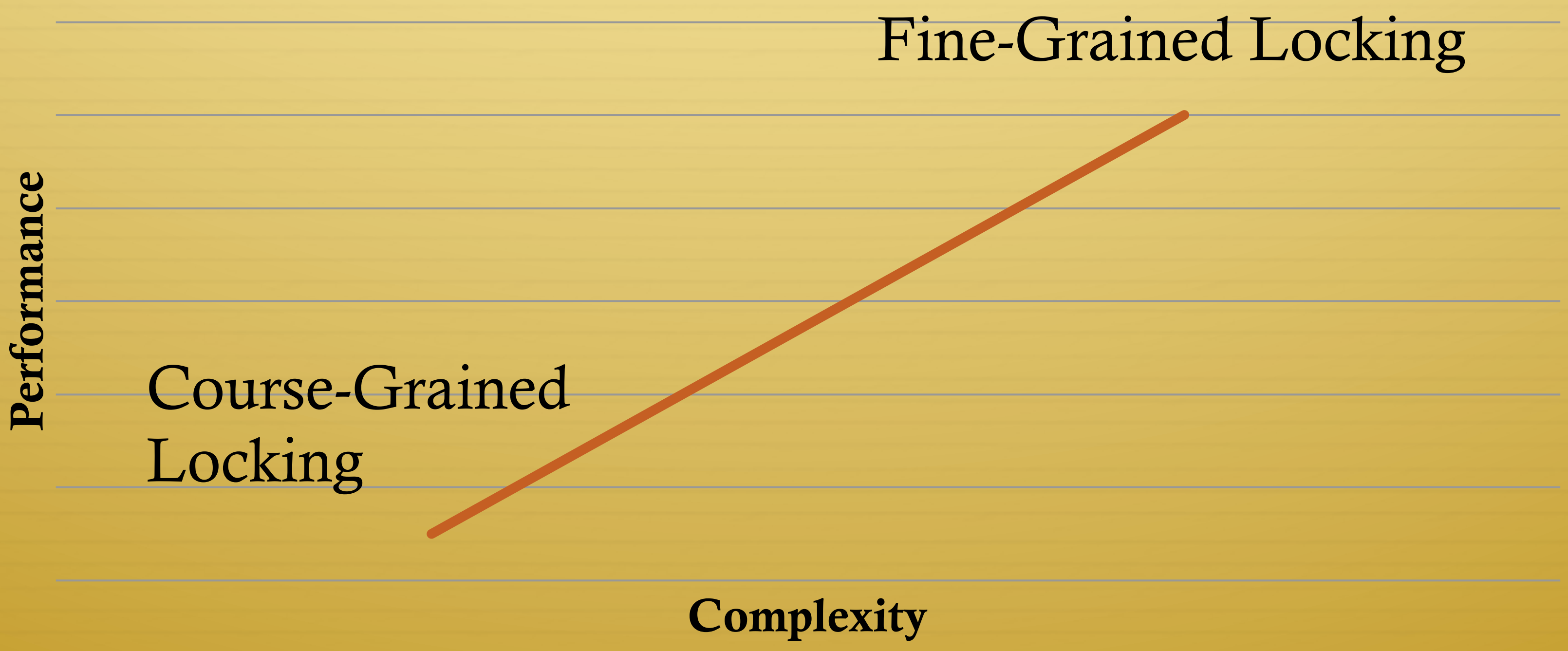
Coarse vs. Fine-grained locking

- Coarse: A single lock for everything
  - Idea: Before I touch any shared data, grab the lock
  - Problem: completely unrelated operations wait on each other
    - Adding CPUs doesn't improve performance

Fine-grained locking

- Fine-grained locking: Many "little" locks for individual data structures
  - Goal: Unrelated activities hold different locks
    - Hence, adding CPUs improves performance
  - Cost: complexity of coordinating locks

Current Reality



- Unsavory trade-off between complexity and performance scalability

How do locks work?

- Two key ingredients:
  - A hardware-provided atomic instruction
    - Determines who wins under contention
  - A waiting strategy for the loser(s)

Atomic instructions

- A “normal” instruction can span many CPU cycles
  - Example: ‘a = b + c’ requires 2 loads and a store
  - These loads and stores can interleave with other CPUs’ memory accesses
- An atomic instruction guarantees that the entire operation is not interleaved with any other CPU
  - x86: Certain instructions can have a ‘lock’ prefix
  - Intuition: This CPU ‘locks’ all of memory
  - Expensive! Not ever used automatically by a compiler; must be explicitly used by the programmer

Atomic instruction examples

- Atomic increment/decrement (x++ or x--)
  - Used for reference counting
  - Some variants also return the value x was set to by this instruction (useful if another CPU immediately changes the value)
- Compare and swap
  - if (x == y) x = z;
  - Used for many lock-free data structures

Atomic instructions + locks

- Most lock implementations have some sort of counter
- Say initialized to 1
- To acquire the lock, use an atomic decrement
  - If you set the value to 0, you win! Go ahead
  - If you get < 0, you lose. Wait ☹
  - Atomic decrement ensures that only one CPU will decrement the value to zero
- To release, set the value back to 1

Waiting strategies

- Spinning: Just poll the atomic counter in a busy loop; when it becomes 1, try the atomic decrement again
- Blocking: Create a kernel wait queue and go to sleep, yielding the CPU to more useful work
  - Winner is responsible to wake up losers (in addition to setting lock variable to 1)
  - Create a kernel wait queue – the same thing used to wait on I/O
    - Note: Moving to a wait queue takes you out of the scheduler's run queue

Which strategy to use?

- Main consideration: Expected time waiting for the lock vs. time to do 2 context switches
  - If the lock will be held a long time (like while waiting for disk I/O), blocking makes sense
  - If the lock is only held momentarily, spinning makes sense
- Other, subtle considerations we will discuss later

Linux lock types

- Blocking: mutex, semaphore
- Non-blocking: spinlocks, seqlocks, completions

Linux spinlock (simplified)

```
1: lock; decb slp->slock        // Locked decrement of lock var
   jns 3f                       // Jump if not set (result is zero) to 3
2: pause                        // Low power instruction, wakes on
                                // coherence event
   cmpb $0,slp->slock           // Read the lock value, compare to zero
   jle 2b                       // If less than or equal (to zero), goto 2
   jmp 1b                       // Else jump to 1 and try again
3:                              // We win the lock
```

Rough C equivalent

```
while (0 != atomic_dec(&lock->counter)) {
        do {
                // Pause the CPU until some coherence
                // traffic (a prerequisite for the counter changing)
                //  saving power

        } while (lock->counter <= 0);
}
```

Why 2 loops?

- Functionally, the outer loop is sufficient
- Problem: Attempts to write this variable invalidate it in all other caches
  - If many CPUs are waiting on this lock, the cache line will bounce between CPUs that are polling its value
    - This is VERY expensive and slows down EVERYTHING on the system
  - The inner loop read-shares this cache line, allowing all polling in parallel
- This pattern called a Test&Test&Set lock (vs. Test&Set)

Reader/writer locks

- Simple optimization: If I am just reading, we can let other readers access the data at the same time
  - Just no writers
- Writers require mutual exclusion

Linux RW-Spinlocks

- Low 24 bits count active readers
  - Unlocked: 0x01000000
  - To read lock: atomic_dec_unless(count, 0)
    - 1 reader: 0x:00ffffff
    - 2 readers: 0x00fffffe
    - Etc.
    - Readers limited to 2^24. That is a lot of CPUs!
- 25th bit for writer
  - Write lock – CAS 0x01000000 -> 0
    - Readers will fail to acquire the lock until we add 0x1000000

Subtle issue

- What if we have a constant stream of readers and a waiting writer?
  - The writer will starve
- We may want to prioritize writers over readers
  - For instance, when readers are polling for the write
  - How to do this?

Seqlocks

- Explicitly favor writers, potentially starve readers
- Idea:
  - An explicit write lock (one writer at a time)
  - Plus a version number – each writer increments at beginning and end of critical section
- Readers: Check version number, read data, check again
  - If version changed, try again in a loop
  - If version hasn't changed and is even, neither has data

Seqlock Example

70

% Time for CSE 506

30

% Time for All Else

0

Version

Lock

Invariant: Must add up to 100%

Seqlock Example

80

% Time for CSE 506

20

% Time for All Else

What if reader executed now?

```
Reader:
do {
    v = version;
    a = cse506;
    b = other;
} while (v % 2 == 1  &&
         v != version);
```

```
Writer:
    lock();
    version++;
    other = 20;
    cse506 = 80;
    version++;
    unlock();
```

Seqlocks

- Explicitly favor writers, potentially starve readers
- Idea:
  - An explicit write lock (one writer at a time)
  - Plus a version number – each writer increments at beginning and end of critical section
- Readers: Check version number, read data, check again
  - If version changed, try again in a loop
  - If version hasn't changed and is even, neither has data

Composing locks

- Suppose I need to touch two data structures (A and B) in the kernel, protected by two locks.
- What could go wrong?
  - Deadlock!
  - Thread 0: lock(a); lock(b)
  - Thread 1: lock(b); lock(a)
- How to solve?
  - Lock ordering

Lock Ordering

- A program code convention
- Developers get together, have lunch, plan the order of locks
- In general, nothing at compile time or run-time prevents you from violating this convention
  - Research topics on making this better:
    - Finding locking bugs
    - Automatically locking things properly
    - Transactional memory

How to order?

- What if I lock each entry in a linked list. What is a sensible ordering?
  - Lock each item in list order
  - What if the list changes order?
  - Uh-oh! This is a hard problem
- Lock-ordering usually reflects static assumptions about the structure of the data
  - When you can't make these assumptions, ordering gets hard

Linux solution

- In general, locks for dynamic data structures are ordered by kernel virtual address
  - I.e., grab locks in increasing virtual address order
- A few places where traversal path is used instead

Lock ordering in practice

From Linux: fs/dcache.c

```
void d_prune_aliases(struct inode *inode) {
        struct dentry *dentry;
        struct hlist_node *p;
restart:
        spin_lock(&inode->i_lock);
        hlist_for_each_entry(dentry, p, &inode->i_dentry, d_alias) {
                spin_lock(&dentry->d_lock);
                if (!dentry->d_count) {
                        __dget_dlock(dentry);
                        __d_drop(dentry);
                        spin_unlock(&dentry->d_lock);
                        spin_unlock(&inode->i_lock);
                        dput(dentry);
                        goto restart;
                }
                spin_unlock(&dentry->d_lock);
        }
        spin_unlock(&inode->i_lock);
}
```

Care taken to lock inode before each alias

Inode lock protects list; Must restart loop after modification

mm/filemap.c lock ordering

```
/*
 * Lock ordering:
 *  ->i_mmap_lock               (vmtruncate)
 *    ->private_lock            (__free_pte->__set_page_dirty_buffers)
 *      ->swap_lock             (exclusive_swap_page, others)
 *        ->mapping->tree_lock
 *  ->i_mutex
 *    ->i_mmap_lock             (truncate->unmap_mapping_range)
 *  ->mmap_sem
 *    ->i_mmap_lock
 *      ->page_table_lock or pte_lock   (various, mainly in memory.c)
 *        ->mapping->tree_lock  (arch-dependent flush_dcache_mmap_lock)
 *  ->mmap_sem
 *    ->lock_page               (access_process_vm)
 *  ->mmap_sem
 *    ->i_mutex                 (msync)
 *  ->i_mutex
 *    ->i_alloc_sem             (various)
 *  ->inode_lock
 *    ->sb_lock                 (fs/fs-writeback.c)
 *    ->mapping->tree_lock      (__sync_single_inode)
 *  ->i_mmap_lock
 *    ->anon_vma.lock           (vma_adjust)
 *  ->anon_vma.lock
 *    ->page_table_lock or pte_lock     (anon_vma_prepare and various)
 *  ->page_table_lock or pte_lock
 *    ->swap_lock               (try_to_unmap_one)
 *    ->private_lock            (try_to_unmap_one)
 *    ->tree_lock               (try_to_unmap_one)
 *    ->zone.lru_lock           (follow_page->mark_page_accessed)
 *    ->zone.lru_lock           (check_pte_range->isolate_lru_page)
 *    ->private_lock            (page_remove_rmap->set_page_dirty)
 *    ->tree_lock               (page_remove_rmap->set_page_dirty)
 *    ->inode_lock              (page_remove_rmap->set_page_dirty)
 *    ->inode_lock              (zap_pte_range->set_page_dirty)
 *    ->private_lock            (zap_pte_range->__set_page_dirty_buffers)
 *  ->task->proc_lock
 *    ->dcache_lock             (proc_pid_lookup)
 */
```

Semaphore

- A counter of allowed concurrent processes
  - A mutex is the special case of 1 at a time
- Plus a wait queue
- Implemented similarly to a spinlock, except spin loop replaced with placing oneself on a wait queue

Ordering blocking and spin locks

- If you are mixing blocking locks with spinlocks, be sure to acquire all blocking locks first and release blocking locks last
  - Releasing a semaphore/mutex schedules the next waiter
    - On the same CPU!
  - If we hold a spinlock, the waiter may also try to grab this lock
  - The waiter may block trying to get our spinlock and never yield the CPU
  - We never get scheduled again, we never release the lock

Summary

- Understand how to implement a spinlock/semaphore/ rw-spinlock
- Understand trade-offs between:
  - Spinlocks vs. blocking lock
  - Fine vs. coarse locking
  - Favoring readers vs. writers
- Lock ordering issues