Synchronization

Nima Honarmand

(Based on slides by Don Porter and Mike Ferdman)
What is Synchronization?

• Code on multiple CPUs coordinate their operations

• Examples:
  – Locking provides mutual exclusion
    • CPU A locks CPU B’s run queue to steal tasks
      • Otherwise CPU B may start running a task that CPU A is stealing
  – Threads wait at barrier for completion 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
Kernel Locking History

- Traditionally, didn’t worry about it
  - Most machines were single processor

- Eventually started supporting multi-processors
  - Called kernels “SMP” around this time
  - Typically had a few (one?) lock
    - Called “Giant” lock

- Giant lock became a bottleneck
  - Switches to fine-grained locking
    - With many different types of locks

- Grew tools to dynamically detect/fix locking bugs
  - E.g., FreeBSD “WITNESS” infrastructure
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

- Perfect Scalability
- Not Scalable
- Somewhat Scalable

Ideal: Time halves with 2x CPUS
Performance Scalability (more visually intuitive)

Slope $= 1 ==$ perfect scaling

- **Perfect Scalability**
- **Not Scalable**
- **Somewhat scalable**
Coarse-Grained Locking

• A single lock for everything
  – Idea: Before touching any shared data, grab the lock
  – Problem: completely unrelated operations serialized
    • Adding CPUs doesn’t improve performance
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?

• Locks are addresses in **shared memory**
  – To check if locked, read value from location
  – To unlock, write value to location to indicate unlocked
  – To lock, write value to location to indicate locked
    • If already locked, keep reading value until unlock observed

• Use hardware-provided **atomic instruction**
  – Determines who wins under contention
  – Requires waiting strategy for the loser(s)

• Also need a waiting strategy for the loser(s)
Atomic Instructions

• Regular memory accesses don’t work

```c
lock: movq [lock], %rax
cmpq %rax, 1
je lock
movq 1, [lock]
```

• Atomic Instructions guarantee atomicity
  – Perform **Read, Modify, and Write** together (RMW)
  – Many flavors in the real world (**lock** prefix on x86)
    • **Compare and Swap** (CAS)
    • **Fetch and Add**
    • **Test and Set**
    • **Load Linked / Store Conditional**
Atomic Instruction Examples

• Atomic increment/decrement
  – $x = x \pm 1$
  – 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 $(x == y) \{x = z; \text{return 1;}\} \text{ else } \{\text{return 0;}\}$
  – 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
  – Poll lock in a busy loop
  – When lock is free, try to acquire it

• Blocking
  – Put process on wait queue and go to sleep
    • CPU may do useful work
  – Winner (lock holder) wakes up loser(s)
    • After releasing lock
  – Same thing as used to wait on I/O
Which Strategy to Use?

- Expected waiting time vs. time of 2 context switches
  - If lock will be held a long time, blocking makes sense
  - If the lock is only held momentarily, spinning makes sense

- Adaptive sometimes works
  - Try to spin a bit
    - If successful, great
    - If unsuccessful, block
  - Can backfire (if spin is never successful)
Linux Lock Types

• Non-blocking: spinlocks, seqlocks, completions
• Blocking: mutex, semaphore
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);
    }
}
Reader/Writer Lock

• If everyone is reading, no need to block
  – Everyone reads at the same time
• Writers require mutual exclusion
  – For anyone to write, wait for all readers to give up lock
Linux RW-Spinlock

- Low 24 bits count active readers
  - Unlocked: 0x01000000
  - To read lock: atomic_dec_unless(count, 0)
    - 1 reader: 0x:00ffffff
    - 2 readers: 0x00fffffffe
    - Etc.
    - Readers limited to $2^{24}$

- 25th bit for writer
  - Write lock – CAS 0x01000000 -> 0
    - Readers will fail to acquire the lock until we add 0x10000000
Readers Starving Writers

- Constant stream of readers starves writer
- We may want to prioritize writers over readers
  - For instance, when readers are polling for the write
Seqlock

• 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

% Time for CSE 506: 70%
% Time for All Else: 30%

Invariant: Must add up to 100%
Seqlock Example

% Time for CSE 506 80
% Time for All Else 20

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();
Seqlock

• 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
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
Lock Composition

• Need to touch two data structures (A and B)
  – Each is protected by its own lock
• 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 code convention

• Developers gather, eat lunch, plan order of locks
  – Potentially worse: gather, drink beer, plan order of locks

• Nothing prevents violating convention
  – Research topics on making this better:
    • Finding locking bugs
    • Automatically locking things properly
    • Transactional memory
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)
 */