Concurrency Bugs

Nima Honarmand
(Based on slides by Prof. Andrea Arpaci-Dusseau)

Concurrency Bugs are Serious

The Therac-25 incident (1980s)

"The accidents occurred when the high-power electron beam was activated instead of the intended low power beam, and without the beam spreader plate rotated into place. Previous models had hardware interlocks in place to prevent this, but Therac-25 had removed them, depending instead on software interlocks for safety. The software interlock could fail due to a **race condition**."

"…in three cases, the injured patients **later died**."

Source: en.wikipedia.org/wiki/Therac-25

Concurrency Bugs are Serious (2)

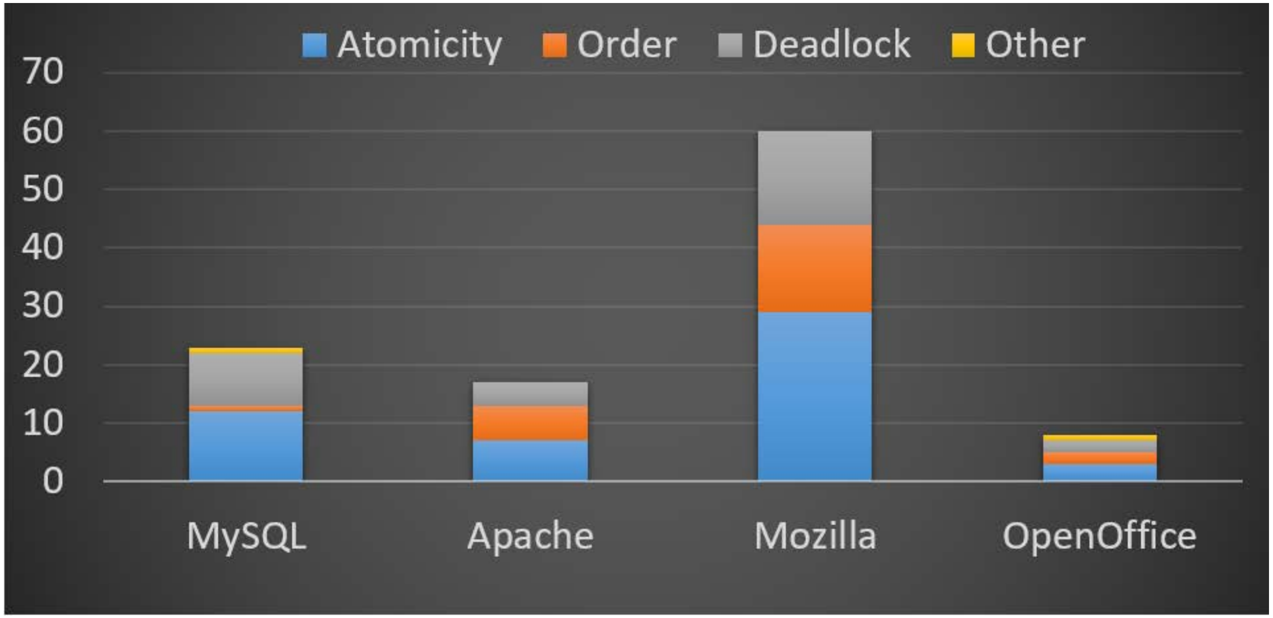
Northeast blackout of 2003

“The Northeast blackout of 2003 was a widespread power outage that occurred throughout parts of the Northeastern and Midwestern United States and the Canadian province of Ontario on Thursday, August 14, 2003, just after 4:10 p.m. EDT.”

The blackout's primary cause was a bug in the alarm system... The lack of an alarm left operators unaware of the need to re-distribute power after overloaded transmission lines hit unpruned foliage, triggering a "race condition" in the energy management system... What would have been a manageable local blackout cascaded into massive widespread distress on the electric grid.”

Source: en.wikipedia.org/wiki/Northeast_blackout_of_2003

Concurrency Study from 2008



For four major projects, search for concurrency bugs among > 500K bug reports. Analyze small sample to identify common types of concurrency bugs.

Source: Lu et. al, "Learning from mistakes — a comprehensive study on real world concurrency bug characteristics"

Atomicity Violation Bugs

"The desired serializability among multiple memory accesses is violated (i.e. a code region is intended to be atomic, but the atomicity is not enforced during execution)"

MySQL Example

Thread 1

```
if (thd->proc_info) {
    …
    fputs(thd->proc_info, …);
    …
}
```

Thread 2

```
thd->proc_info = NULL;
```

- What's wrong?
- How to fix?
  - Use a lock

Ordering Violation Bugs

"The desired order between two (groups of) memory accesses is flipped (i.e., A should always be executed before B , but the order is not enforced during execution)"

Mozilla Example

Thread 1

```
void init() {
  …
  mThread =
    PR_CreateThread(mMain, …);
  …
}
```

Thread 2

```
void mMain(…) {
  …
  mState = mThread->State;
  …
}
```

- What's wrong?
- How to fix?
  - Use a condition variable

Ordering Violation Bugs (2)

Thread 1

```
void init() {
  …
  mThread =
    PR_CreateThread(mMain, …);
  mutex_lock(&mtLock);
  mtInit = 1;
  cond_signal(&mtCond);
  mutex_unlock(&mtLock);
    …
}
```

Thread 2

```
void mMain(…) {
  …
  mutex_lock(&mtLock);
  while (mtInit == 0)
    cond_wait(&mtCond, &mtLock);
  mutex_unlock(&mtLock);

  mState = mThread->State;
  …
}
```

- Why are we using a new flag (`mtInit`) instead of `mThread` itself?

Fixing Concurrency Bugs: Easy?

- If all we had to do was adding locks and cond vars, concurrent programming would be quite simple
- Problems?

1) Adding too many locks increase the danger of deadlocks
2) How about having just a few big locks then?
   - Causes performance problems because it reduces concurrency

Locking Granularity

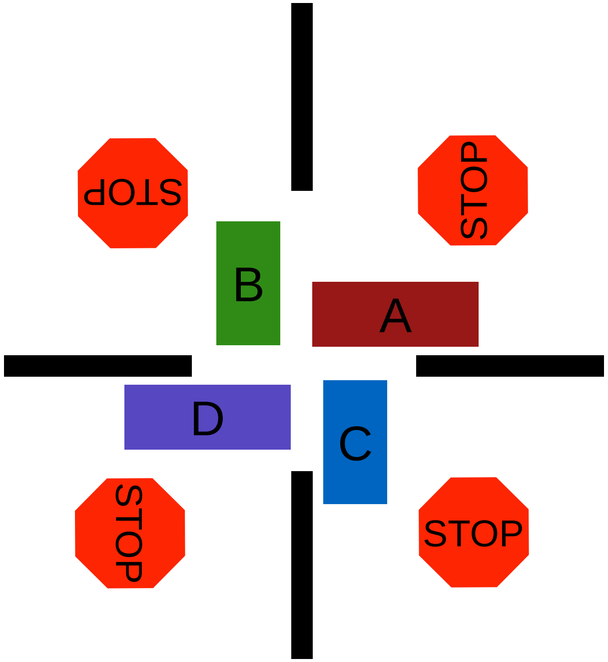
- ***Coarse-grain locking***
  - Have one (or a few) locks that protect all (or big chunks) of shared state
  - Example: early Linux's BKL (Big Kernel Lock)
    - One big lock protecting all kernel data
    - Only one processor code execute kernel code at any point of time; others would have to wait
  - Significant contention over big locks → hurts performance
- ***Fine-grain locking***
  - Have many small locks, each protecting one (or a few) objects
  - Reduces contention → better performance
  - Increases deadlock risk

Deadlock Bugs

- Deadlock: No progress can be made because two or more threads are waiting for the other to take some action and thus neither ever does
- Could arise when we need to coordinate access to more than one shared resources
  - Means we need to grab and hold multiple locks simultaneously

Deadlock Theory

- Deadlocks can only occur when all four conditions are true:
  1) Mutual exclusion
  2) Hold-and-wait
  3) Circular wait
  4) No preemption



- <u>Eliminate deadlock by eliminating any one condition</u>

1) Mutual Exclusion

- Definition: “Threads claim exclusive control of resources that they require (e.g., thread grabs a lock)”
- Strategy: eliminate locks
  - Try to use atomic instructions instead

Concurrent Counter Example

Code with locks

```
void add (int *val, int amt)
{
  mutex_lock(&m);
  *val += amt;
  mutex_unlock(&m);
}
```

Code with Compare-and-Swap (CAS)

```
void add (int *val, int amt)
{
  do {
    int old = *value;
  } while(!CAS(val, old, old+amt));
}
```

Example: Lock-Free Linked List Insert

Code with locks

```
void insert (int val)
{
  node_t *n =
    malloc(sizeof(*n));
  n->val = val;
  mutex_lock(&m);
  n->next = head;
  head = n;
  mutex_unlock(&m);
}
```

Code with Compare-and-Swap (CAS)

```
void insert (int val)
{
  node_t *n = malloc(sizeof(*n));
  n->val = val;
  do {
    n->next = head;
  } while (!CAS(&head, n->next, n));
}
```


2) Hold-and-Wait

- Definition: "Threads hold resources allocated to them (e.g., locks they have already acquired) while waiting for additional resources (e.g., locks they wish to acquire)."
- Strategy: release currently held resources when waiting for new ones

Example with `trylock`

```
top:
  pthread_mutex_lock(A);
  if (pthread_mutex_trylock(B) != 0)
  {
    pthread_mutex_unlock(A);
    goto top;
  }
…
```

Problem w/ This Strategy

- Potential for ***Livelock***: no process makes forward progress, but the state of involved processes constantly changes
- Can happen if all processes release resources and then try to re-acquire, fail, and keep doing this
- Classic solution: back-off techniques
  - ***Random back-off***: wait for a random amount of time before retrying
  - ***Exponential back-off***: wait for exponentially increasing amount of time before retrying

3) Circular Wait

- Definition: “There exists a circular chain of threads such that each thread holds a resource (e.g., lock) being requested by next thread in the chain.”
- Usually the easiest deadlock requirement to attack
- Strategy: impose a well-documented order of acquiring locks
  - Decide which locks should be acquired before others
  - If A before B, never acquire A if B is already held!
  - Document this, and write code accordingly
- Works well if system has distinct layers

Simple Example

Thread 1

```
lock(&A);
lock(&B);
```

Thread 2

```
lock(&B);
lock(&A);
```

How would you fix this code?

Thread 1

```
lock(&A);
lock(&B);
```

Thread 2

```
lock(&A);
lock(&B);
```

Example: 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)
. . .
```

Encapsulation Makes Ordering Difficult

- Encapsulation, and emphasis on code modularity, make things difficult
  - Can't control the order in which locks are acquired when we calling a function in another module

- What could go wrong in this code?

```
set_t *intersect(set_t *s1, set_t *s2)
{
    set_t *rv = malloc(sizeof(*rv));
    mutex_lock(&s1->lock);
    mutex_lock(&s2->lock);
    for(int i=0; i<s1->len; i++) {
        if(set_contains(s2, s1->items[i])
            set_add(rv, s1->items[i]);
    mutex_unlock(&s2->lock);
    mutex_unlock(&s1->lock);
}
```

Deadlock possible if one thread calls `intersect(s1, s2)` and another thread `intersect(s2, s1)`

One Possible Solution

- Acquire the locks in the order of their virtual addresses when possible

```
set_t *intersect(set_t *s1, set_t *s2) {
    set_t *rv = malloc(sizeof(*rv));
    if ((uint)&s1->lock < (uint)&s2->lock) {
        mutex_lock(&s1->lock);
        mutex_lock(&s2->lock);
    } else {
        mutex_lock(&s2->lock);
        mutex_lock(&s1->lock);
    }
    for(int i=0; i<s1->len; i++) {
        if(set_contains(s2, s1->items[i])
            set_add(rv, s1->items[i]);
    mutex_unlock(&s2->lock);
    mutex_unlock(&s1->lock);
}
```

You may also want to change the order of `unlock()`s to be reverse of `lock()`s.

Other Complications

- Sometimes can't know all virtual addresses in advance
- Example: when traversing a linked list where each object has a separate lock

Linux Example: `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);
}
```

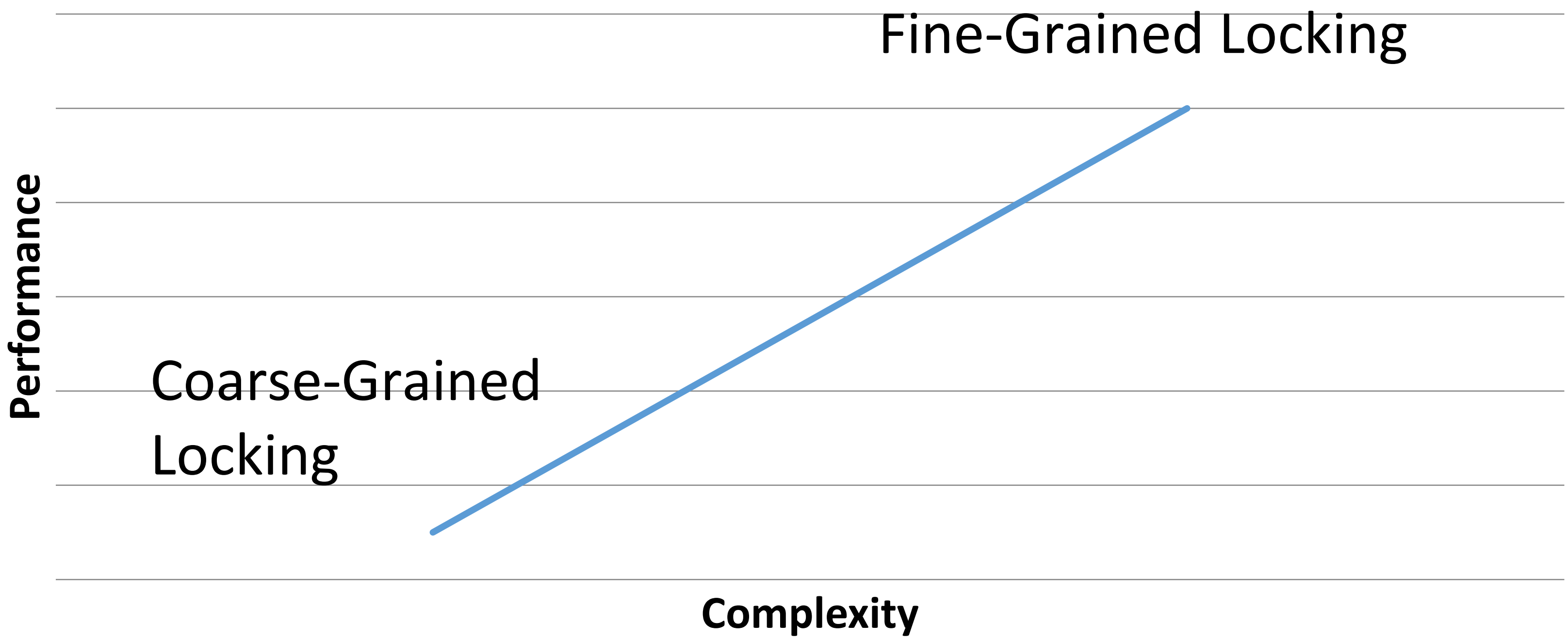
Make sure inode lock is acquired before dentry locks

When a list element is removed, have to restart from beginning because order of items has changed.

4) Deadlock Detection and Recovery

- Database systems use many, many locks
  - Very difficult to always avoid deadlocks in general in such a system
- Last-resort strategy: detect deadlocks, and recover
  - Detection usually involves looking out for locks that are held for too long
  - Recovery usually requires a restart of the database app
- An example of breaking the “No preemption” condition
  - By restarting, we are forcibly releasing the resource

Summary: Current Reality



Unsavory trade-off between synchronization complexity and performance

Locking in Kernel

- All locking stuff we discussed so far applies equally to kernel and user code
  - Spinlocks
  - Blocking locks
  - Granularity
  - Deadlock
  - Etc.
- However, there is one form of concurrency that’s (almost) only found in kernel, remember?
  - Yes, interrupts!

Locks and Interrupts

- Suppose you are in the disk driver (say, serving a `read()` syscall) and holding a disk-related lock
- Say, a disk interrupt happens, and you need to grab the same lock in the interrupt service routine (ISR)
- What would happen?
  - Yes, deadlock
    - Can’t finish the ISR without grabbing the lock
    - Can’t return to driver code (to release the lock) without finishing ISR
- Can you identify the multiple resources that are involved in the deadlock?
  1) Lock
  2) CPU

Solution

- How can we solve this problem?
- Two part solution:
  1) Only use spinlocks in ISRs — never call, directly or indirectly, a routine that would use a blocking lock
  2) When acquiring a spinlock in kernel, disable interrupts on the current processor
- Why just on this processor? Is it okay to get an interrupt on other processors?
- This is why xv6 kernel spinlocks disable interrupts