Locking Details in xv6

Computer Science 432 — Lecture 13 — Duane Bailey

April 6, 2022

Announcements

- (Optional) small group meetings are in the Microscopy Lab,
  Hopper Basement Tunnel City Corner (Room G10)
  Wed 11-12:15 (A), 1:10-2:25 (B), 2:35-3:50 (C)
  Thu 9:55-11:10 (D), 1:10-2:25(E), 2:35-3:50 (F)

- O/S Conference topic contracts due today. Please turn these in this morning.

- Chapter 7 is due for Monday. We'll be thinking about threading in our next lab.

Lock design

- Recall that a lock involves a *shared* memory location capable of holding an integer.
  - The most obvious way that two processes could share a location is in the kernel.
  - We have seen several different lock variables:
    - the lock that governs access to process state,
    - the lock that controls access to the freelist, and
    - the lock that controls access to the nextpid.
  - Each of these governs access to critical sections that *update* some shared state.

What makes a critical section?

- An important part of using locks, of course, is the identification of *critical sections*.
  - Typically, it involves a shared state that is written
    and
    possibly read or written concurrently.
  - An example: The nextpid variable.
    - Process identifiers are allocated by
      - (1) Reading the nextpid variable.
      - (2) Updating the nextpid variable.
    - Clearly it is important that pid values are unique.
    - It's not so important that they be consecutive.
    - The worry is that two different threads will execute pid = nextpid at the same time.

```
int allocpid(void)
{
   int pid;

   pid = nextpid;
   nextpid = nextpid+1;

   return pid;
}
```

What makes a critical section?

- An important part of using locks, of course, is the identification of *critical sections.*
  - Typically, it involves a shared state that is written
    and
    possibly read or written concurrently.
  - An example: The nextpid variable.
    - Process identifiers are allocated by
      - (1) Reading the nextpid variable.
      - (2) Updating the nextpid variable.
    - Clearly it is important that pid values are unique.
    - It's not so important that they be consecutive.
    - The worry is that two different threads will execute pid = nextpid at the same time.

```
int allocpid(void)
{
    int pid;
    acquire(&pid_lock);
    pid = nextpid;
    nextpid = nextpid+1;
    release(&pid_lock);
    return pid;
}
```

How about reference counting?

- When we are sharing physical pages in page tables, we use a reference counter
  - By design, there are several places where the reference counter is updated
  - In kshare(pa), we *increment* the counter.
  - In kfree(pa), we *decrement* the counter.
  - What's important, here, is that the reference counter accurately reflects the number of copies.
  - *Any place a reference counter is modified, it becomes a part of a race.*
  - Reference counting should be guarded by a lock.

```
in kshare(pa)
{ . . .
    // saturating

    if (*counter < 255)
       *counter += 1;

}
```

```
in kfree(pa)
{ . . .
    // saturating

    if (*counter > 0 && *counter < 255)
       *counter -= 1;
    if (*counter == 0) . . .

}
```

How about reference counting?

- When we are sharing physical pages in page tables, we use a reference counter
  - By design, there are several places where the reference counter is updated
  - In kshare(pa), we *increment* the counter.
  - In kfree(pa), we *decrement* the counter.
  - What's important, here, is that the reference counter accurately reflects the number of copies.
  - *Any place a reference counter is modified, it becomes a part of a race.*
  - Reference counting should be guarded by a lock.

```
in kshare(pa)
{ . . .
    // saturating
    acquire(&kmem.lock);
    if (*counter < 255)
        *counter += 1;
    release(&kmem.lock);
}
```

```
in kfree(pa)
{ . . .
    // saturating
    acquire(&kmem.lock);
    if (*counter > 0 && *counter < 255)
        *counter -= 1;
    if (*counter == 0) . . .
    release(&kmem.lock);
}
```

Lock granularity.

- Any shared value that is potentially in a race must be protected by a lock
  - Fine-grained approach: Each shared variable is guarded by a dedicated lock
    - e.g. pid_lock guards nextpid
    - e.g. p->lock guards state of process p.
  - Coarse-grained approach: An entire structure is guarded
    - e.g. kmem.lock guards the entire free list
      (subtle: the freelist variable could be subject to a race, but *also* every run.next)
  - *Very* coarse-grained approach: The *entire kernel* is guarded!
    - e.g. Linux's *Big Kernel Lock* (BKL).

Locks in xv6

- The xv6 operating system uses two different types of *mutex* (mutual exclusion) locks:
  - Spinlocks (see spinlock.c). We saw these in the last class.
    - These locks use tight loops with atomic amoswap operations.
    - Looping (in acquire) is a waste of time.
    - Spinlocks cannot yield the CPU, if they did there might be deadlock.
    - Spinlocks cannot be interrupted, for similar reason. Long holds are problematic.
  - Sleep-locks (see sleeplock.c).
    - These locks cause the process to sleep while waiting for acquire.
    - They require additional complications: wait and notify.
    - It's possible to yield the CPU while holding a sleep-lock.

Locks in xv6

Lock	Description
bcache.lock	Protects allocation of block buffer cache entries
cons.lock	Serializes access to console hardware, avoids intermixed output
ftable.lock	Serializes allocation of a struct file in file table
itable.lock	Protects allocation of in-memory inode entries
vdisk_lock	Serializes access to disk hardware and queue of DMA descriptors
kmem.lock	Serializes allocation of memory
log.lock	Serializes operations on the transaction log
pipe's pi->lock	Serializes operations on each pipe
pid_lock	Serializes increments of next_pid
proc's p->lock	Serializes changes to process's state
wait_lock	Helps wait avoid lost wakeups
tickslock	Serializes operations on the ticks counter
inode's ip->lock	Serializes operations on each inode and its content
buf's b->lock	Serializes operations on each block buffer

Figure 6.3: Locks in xv6

Chains of Locks

- All complex systems have multiple, interacting, critical sections.
- When a multiple locks must be held, *they must always be acquired in the same order.*
  - Example: Suppose an airport has a *runway* and 2 *taxiways*. All require a lock.
    - An *arriving* plane first wants to claim the runway and then,
      after landing it wants to claim the appropriate taxiway.
    - A *departing* plane first wants its taxiway and then,
      after taxiing it wants to claim the runway.
    - Two planes—one arriving and one departing—may **deadlock**.
    - As a result, all required resources are acquired in the same order.
      e.g. You cannot arrive or depart until you have (1) a taxiway and then (2) the runway.
    - All the job of air traffic controllers….

Spinlocks and Interrupts

- When you hold a spinlock, it's important you not surrender the cpu; interrupts must be turned off.
  - acquire(lk) — turns off interrupts, if they're on.
  - release(lk) — turns interrupt on, *if this lock turned them off*.
  - allows the holding of a sequence of spinlocks.

Sleep-locks.

- Sleep-locks are more complex structures that require coordination with the scheduler.
  - The sleep lock contains a field, locked, that indicates someone is holding the lock.
  - The sleep lock's lock is protected by a spinlock, lk.
    - You must hold lk to change lock.
  - When the sleep-lock is attempting to acquire a lock in use
    - It goes to sleep with the lock as its *channel*,
      but only after sleep releases the spinlock.
  - When the sleep-lock is attempting to release a lock it holds
    - It calls wakeup to re-schedule the process sleeping on the lock,
      re-acquiring the spinlock

- Pay attention to the discussion of sleep and wakeup in sections 7.5-7.7.

Dijkstra's Semaphores: Locks that Count

- A *semaphore* is a synchronization primitive that governs *n* resources
  - Two operatations P(sem) and V(sem)
    EWD was Dutch: *probeer* for "try", and *verhoog* for "increase" or *vrij* for "free".
  - The semaphore sem is initialized with the value *n*
    reflecting the fact there are *n* resources available.
  - P(sem) waits until sem > 0 and then atomically decrements it.
  - V(sem) atomically increments sem.
  - Note that if *n* is 1, this is a *mutex*.
- Producer-Consumer relationship (e.g. bounded buffers in pipes)
  - The value of the semephore keeps track of how many more objects may be produced before one must be consumed. It's initialized to the size of the pipe.
  - The producer gets a resource ready, calls P(sem) and when it returns,
    it puts the item in the buffer.
  - The consumer takes a resource from the buffer and then calls V(sem), freeing a slot