```
6.S081 2020 Lecture 10: Locking

Why talk about locking?
  apps want to use multi-core processors for parallel speed-up
  so kernel must deal with parallel system calls
  and thus parallel access to kernel data (buffer cache, processes, &c)
  locks help with correct sharing of data
  locks can limit parallel speedup

What goes wrong if we don't have locks
  Case study: delete acquire/release in kalloc.c
  Boot
    kernel works!
  Run usertests
    all tests pass!
    except we lose some pages
  Why do we lose pages?
    picture of shared-memory multiprocessor
    race between two cores calling kfree()
  BUMMER:
    we need locks for correctness
    but loose performance (kfree is serialized)

The lock abstraction:
  lock l
  acquire(l)
    x = x + 1 -- "critical section"
  release(l)
  a lock is itself an object
  if multiple threads call acquire(l)
    only one will return right away
    the others will wait for release() -- "block"
  a program typically has lots of data, lots of locks
    if different threads use different data,
    then they likely hold different locks,
    so they can execute in parallel -- get more work done.
  note that lock l is not specifically tied to data x
    the programmer has a plan for the correspondence

A conservative rule to decide when you need to lock:
  any time two threads use a memory location, and at least one is a write
  don't touch shared data unless you hold the right lock!
  (too strict: program logic may sometimes rule out sharing; lock-free)
  (too loose: printf(); not always simple lock/data correspondence)

Could locking be automatic?
  perhaps the language could associate a lock with every data object
    compiler adds acquire/release around every use
    less room for programmer to forget!
  that idea is often too rigid:
    rename("d1/x", "d2/y"):
      lock d1, erase x, unlock d1
      lock d2, add y, unlock d2
    problem: the file didn't exist for a while!
      rename() should be atomic
        other system calls should see before, or after, not in between
      otherwise too hard to write programs
    we need:
      lock d1 ; lock d2
      erase x, add y
      unlock d2; unlock d1
  that is, programmer often needs explicit control over
    the region of code during which a lock is held
    in order to hide awkward intermediate states
```

```
Ways to think about what locks achieve
  locks help avoid lost updates
  locks help you create atomic multi-step operations -- hide intermediate states
  locks help operations maintain invariants on a data structure
    assume the invariants are true at start of operation
    operation uses locks to hide temporary violation of invariants
    operation restores invariants before releasing locks

Problem: deadlock
  notice rename() held two locks
  what if:
    core A              core B
    rename(d1/x, d2/y)  rename(d2/a, d1/b)
      lock d1             lock d2
      lock d2 ...         lock d1 ...
  solution:
    programmer works out an order for all locks
    all code must acquire locks in that order
    i.e. predict locks, sort, acquire -- complex!

Locks versus modularity
  locks make it hard to hide details inside modules
  to avoid deadlock, I need to know locks acquired by functions I call
  and I may need to acquire them before calling, even if I don't use them
  i.e. locks are often not the private business of individual modules

Locks and parallelism
  locks *prevent* parallel execution
  to get parallelism, you often need to split up data and locks
    in a way that lets each core use different data and different locks
    "fine grained locks"
  choosing best split of data/locks is a design challenge
    whole FS; directory/file; disk block
    whole kernel; each subsystem; each object
  you may need to re-design code to make it work well in parallel
    example: break single free memory list into per-core free lists
      helps if threads were waiting a lot on lock for single free list
    such re-writes can require a lot of work!

Lock granularity advice
  start with big locks, e.g. one lock protecting entire module
    less deadlock since less opportunity to hold two locks
    less reasoning about invariants/atomicity required
  measure to see if there's a problem
    big locks are often enough -- maybe little time spent in that module
  re-design for fine-grained locking only if you have to

Let's look at locking in xv6.

A typical use of locks: uart.c
  typical of many O/S's device driver arrangements
  diagram:
    user processes, kernel, UART, uartputc, remove from uart_tx_buf,
    uartintr()
  sources of concurrency: processes, interrupt
  only one lock in uart.c: uart_tx_lock -- fairly coarse-grained
  uartputc() -- what does uart_tx_lock protect?
    1. no races in uart_tx_buf operations
    2. if queue not empty, UART h/w is executing head of queue
    3. no concurrent access to UART write registers
  uartintr() -- interrupt handler
    acquires lock -- might have to wait at interrupt level!
    removes character from uart_tx_buf
    hands next queued char to UART h/w (2)
```

```
    touches UART h/w registers (3)

How to implement locks?
  why not:
    struct lock { int locked; }
    acquire(l) {
      while(1){
        if(l->locked == 0){ // A
          l->locked = 1;    // B
          return;
        }
      }
    }
  oops: race between lines A and B
  how can we do A and B atomically?

Atomic swap instruction:

  a5 = 1
  s1 = &lk->locked
  amoswap.w.aq a5, a5, (s1)

  does this in hardware:
    lock addr globally (other cores cannot use it)
    temp = *s5
    *addr = a5
    a5 = temp
    unlock addr
  RISC-V h/w provides a notion of locking a memory location
    different CPUs have had different implementations
    diagram: cores, bus, RAM, lock thing
    so we are really pushing the problem down to the hardware
    h/w implements at granularity of cache-line or entire bus
  memory lock forces concurrent swamp to run one at a time, not interleaved

Look at xv6 spinlock implementation
  acquire(l){
    while(__sync_lock_test_and_set(&lk->locked, 1) != 0)
  }
  if l->locked was already 1, sync_lock_test_and_set sets to 1 (again), returns 1,
    and the loop continues to spin
  if l->locked was 0, at most one lock_test_and_set will see the 0; it will set
    it to 1 and return 0; other test_an_set will return 1
  this is a "spin lock", since waiting cores "spin" in acquire loop

  what is the push_off() about?
      why disable interrupts?
  release():
    sets lk->locked = 0
    and re-enables interrupts

Detail: memory read/write ordering
  suppose two cores use a lock to guard a counter, x
  and we have a naive lock implementation
  Core A:          Core B:
    locked = 1
    x = x + 1      while(locked == 1)
    locked = 0       ...
                   locked = 1
                   x = x + 1
                   locked = 0
  the compiler AND the CPU re-order memory accesses
    i.e. they do not obey the source program's order of memory references
    e.g. the compiler might generate this code for core A:
      locked = 1
```

```
      locked = 0
      x = x + 1
      i.e. move the increment outside the critical section!
    the legal behaviors are called the "memory model"
  release()'s call to __sync_synchronize() prevents re-order
    compiler won't move a memory reference past a __sync_synchronize()
    and (may) issue "memory barrier" instruction to tell the CPU
  acquire()'s call to __sync_synchronize() has a similar effect:
  if you use locks, you don't need to understand the memory ordering rules
    you need them if you want to write exotic "lock-free" code

Why spin locks?
  don't they waste CPU while waiting?
  why not give up the CPU and switch to another process, let it run?
  what if holding thread needs to run; shouldn't waiting thread yield CPU?
  spin lock guidelines:
    hold spin locks for very short times
    don't yield CPU while holding a spin lock
  systems provide "blocking" locks for longer critical sections
    waiting threads yield the CPU
    but overheads are typically higher
    you'll see some xv6 blocking schemes later

Advice:
  don't share if you don't have to
  start with a few coarse-grained locks
  instrument your code -- which locks are preventing parallelism?
  use fine-grained locks only as needed for parallel performance
  use an automated race detector
```