Locking in xv6

xv6 runs on a multiprocessor and allows multiple CPUs to execute concurrently inside the kernel. These usually correspond to system calls made by processes running on different CPUs. There might also be interrupt handlers running at the same time as other kernel code. An interrupt can occur on any of the CPUs; both user-level processes and processes inside the kernel can be interrupted. xv6 uses spin-locks to coordinate how these concurrent activities share data structures.

How do xv6 locks work? Let's look at acquire() in spinlock.c. [1474]

- what does pushcli() do? disables interrupts, increments per-CPU variable called ncli. release() calls popcli() which decrements counter and enables interrupts if counter equals zero.
- why does pushcli()/popcli() count nested calls? To ensure that interrupts are disabled only when all locks have been released.
- so why disable interrupts on the current processor while holding the lock? To guarantee atomicity with respect to the interrupt handlers.
- but won't the interrupt handlers also acquire a lock before accessing shared data? yes, they will but that's even worse because if a thread is already holding a lock and an interrupt occurs that tries to acquire the same lock, it will result in a deadlock.
- holding() is true if the current CPU already holds the lock. why is it bad for a CPU to re-acquire a lock it already has? because the lock implementation is not recursive.
- Wouldn't recursive locks be better? No, because that encourages bugs. For example, if interrupts were kept enabled within the critical section and recursive locks were allowed, an interrupt handler would be able to acquire the lock even if the interrupt occurred in the middle of the critical section --- bad!

And release():

- 1519: why not just lock->locked = 0? We need to ensure that instructions do not get reordered (either by compiler or by the hardware at runtime). The xchg instruction acts as an implicit barrier, preventing instructions from getting reordered across it.

Sleep/Wakeup

Sleep and wakeup are xv6's incarnation of condition variables. Here are the signatures of the `sleep()` and `wakeup()` functions:

```
void sleep(void *channel, struct lock *mutex);
void wakeup(void *channel);
```

The `sleep()` function is similar to the `wait()` function and the `wakeup()` function is similar to the `notify` function (recall condition variables). The difference is that instead of declaring a condition variable explicity (recall `struct cv`), we simply use any number (or void pointer) as our "channel" to wait on. Recall that because a condition variable is stateless, we do not really need to declare it, and can simply use any program variable's address to act as the "condition variable" (or channel, as called in xv6). The channel is just a number, so callers of sleep and wakeup must agree on a convention so they correctly synchronize between themselves.

The semantics of sleep()/wakeup() are identical to those of condition variables. The sleep() function goes to sleep on the channel releasing the mutex atomically, and the wakeup() function wakes up all threads sleeping on the channel. Below we describe the semantics of sleep and wakeup using code (assuming xv6 process table structure):

```
void sleep(void *chan, struct lock *lk):
  //begin{atomic}

  curproc->chan = chan       //curproc points to the PCB of current process
  curproc->state = SLEEPING
  release(lk);
  sched()                    //call the scheduler, which will context switch
                               to another process
  //end{atomic}

wakeup(chan):
  //begin{atomic}

  foreach p in ptable[]:     //ptable is an array containing all PCBs
    if p->chan == chan and p->state == SLEEPING:
      p->state = RUNNABLE

  //end{atomic}
```

The sleep() and wakeup() functions need to be atomic with respect to each other (also indicated by the begin-atomic and end-atomic annotations in the pseudo-code) and also with respect to other accesses to the process table. Recall that xv6 uses the `ptable.lock` to implement mutual-exclusion for accesses to the `ptable`.

```
void sleep(void *chan, struct lock *lk):
  acquire(&ptable.lock);
  curproc->chan = chan       //curproc points to the PCB of current process
  curproc->state = SLEEPING
  release(lk);
  sched()                    //recall that the scheduler (or the next process)
```

```
                             //releases ptable.lock

void wakeup(void *chan):
  acquire(&ptable.lock);
  foreach p in ptable[]:     //ptable is an array containing all PCBs
    if p->chan == chan and p->state == SLEEPING:
      p->state = RUNNABLE
  release(&ptable.lock);
```

Notice that it is also okay to release the lock `lk` before accessing `curproc` (as `ptable.lock` is now protecting accesses to `curproc`).

However, we need to ensure that our implementation cannot result in a deadlock (given that a thread can hold two locks at the same time). Firstly, we should check if `lk` is the same as `ptable.lock` --- if so, we do not need to reacquire it again. Secondly, and more importantly, we need to ensure that there is a global ordering on the lock acquisition. Notice that `ptable.lock` is acquired after the acquisition of `lk` (which must have been acquired by the caller of sleep()). The invariant followed by xv6 is that the `ptable.lock` will always be the inner-most lock, i.e., no other lock acquisition will be attempted while `ptable.lock` is held. This ensures that the `ptable.lock` is always the least priority, and so no deadlocks can result from cycles involving `ptable.lock`. Here is xv6's (correct) implementation of sleep and wakeup:

```
void sleep(void *chan, struct lock *lk):
  if (lk != &ptable.lock) {
    acquire(&ptable.lock);
    release(lk);
  }
  curproc->chan = chan       //curproc points to the PCB of current process
  curproc->state = SLEEPING
  sched()                    //recall that the scheduler (or the next process)
                             //releases ptable.lock

void wakeup(void *chan):
  acquire(&ptable.lock);
  foreach p in ptable[]:     //ptable is an array containing all PCBs
    if p->chan == chan and p->state == SLEEPING:
      p->state = RUNNABLE
  release(&ptable.lock);
```

Notice that it is the programmer's responsibility to ensure that `wakeup()` is not called with `ptable.lock` held.

Use example: exit and wait

Recall API: Suppose process P1 has forked child process P2. If P1 calls wait(), it should return after P2 calls exit().

wait() in proc.c looks for any of its children that have already exited; if any exist, clean them up and return. If none, it calls sleep().

exit() calls wakeup(), marks itself as dead (ZOMBIE), and calls sched().

What lock should protect wait() and exit() from missing each other? `ptable.lock` as these functions manipulate the ptable.

What should be the channel convention? Using the process-ID of the waiting process seems like a good one. The waiting process will wait on its own process-ID. The exiting processes will wakeup processes waiting on their *parent's* process-ID.

What if we used a common channel for all processes? e.g., let's say we always used channel `(void *)1` (recall that a channel is just a number). Then exiting children of other processes may cause a waiting process to wakeup from sleep (inside wait). This may not be a correctness problem if the woken-up process again checks if it has any zombie children, and goes to sleep otherwise. However this causes poor performance, as all waiting processes will be woken up on every exit, and all but one of them will go back to sleep. Using the parent's process ID as the channel avoids this extra work.

What if some unrelated part of the kernel decides to sleep and wakeup on the same channel(s)? This will not be a problem from a correctness standpoint (irrelevant sleeping processes will wakeup only to go back to sleep). However this is not desirable from a performance standpoint.

Let's look at the xv6 code for wait and exit. Why does wait() free the child's memory (e.g., pgdir, kstack), and not exit()? Because the exit() function is currently using the process's pgdir and kstack, and so cannot deallocate them. Instead the process's parent can deallocate these structures inside wait.

Things to think about (relates to semantics and typical usage pattern of sleep/wakeup):

What if exit is called before wait?

What if wait is called just after exit calls wakeup?