Lecture 29: Locking in xv6

Mythili Vutukuru
IIT Bombay
https://www.cse.iitb.ac.in/~mythili/os/
Why locking in xv6?

• **No threads in xv6**, so no two user programs can access same userspace memory image
  - No need for userspace locks like pthreads mutex

• **However, scope for concurrency in xv6 kernel**
  - Two processes in kernel mode in different CPU cores can access same kernel data structures
  - When a process is running in kernel mode, another trap occurs, and the trap handler can access data that was being accessed by previous kernel code

• **Solution: spinlocks used to protect critical sections**
  - Limit concurrent access to kernel data structures that can result in race conditions

• xv6 also has a sleeping lock (built on spinlock, not discussed)
Spinlocks in xv6

• Acquiring lock: uses xchg x86 atomic instruction (test and set)
  – Atomically set lock variable to 1 and returns previous value
  – If previous value is 0, it means free lock has been acquired, success!
  – If previous value is 1, it means lock is held by someone, continue to spin in a busy while loop till success
• Releasing lock: set lock variable to 0
• Must disable interrupts on CPU core before spinning for lock
  – Interrupts disabled only on this CPU core to prevent another trap handler running and requesting same lock, leading to deadlock
  – OK for process on another core to spin for same lock, as the process on this core will release it
  – Disable interrupts before starting to spin (otherwise, vulnerable window after lock acquired and before interrupts disabled)
Disabling interrupts

- Must disable interrupts on CPU core before beginning to spin for spinlock
- Interrupts stay disabled until lock is released
- What if multiple spinlocks are acquired?
  - Interrupts must stay disabled until all locks are released
- Disabling/enabling interrupts:
  - pushcli disables interrupts on first lock acquire, increments count for future locks
  - popcli decrements count, reenables interrupts only when all locks released and count is zero
The process table protected by a lock, any access to ptable must be done with ptable.lock held.

Normally, a process in kernel mode acquires ptable.lock, changes ptable, releases lock.
- Example: when allocproc allocates new struct proc

But during context switch from process P1 to P2, ptable structure is being changed all through context switch, so when to release lock?
- P1 acquires lock, switches to scheduler, switches to P2, P2 releases lock
Every function that calls sched() to give up CPU will do so with ptable.lock held. Which functions invoke sched()?
- **Yield**, when a process gives up CPU due to timer interrupt
- **Sleep**, when process wishes to block
- **Exit**, when process terminates

Every function that swtch switches to will release ptable.lock. What functions does swtch return to?
- **Yield**, when switching in a process that is resuming after yielding is done
- **Sleep**, when switching in a process that is waking up after sleep
- **Forkret** for newly created processes

Purpose of forkret: to release ptable.lock after context switch, before returning from trap to userspace
• Scheduler goes into loop with lock held
• Switch to P1, P1 switches back to scheduler with lock held, scheduler switches to P2, P2 releases lock
• Periodically, end of looping over all processes, releases lock temporarily
  – What if no runnable process found due to interrupts being disabled? Release lock, enable interrupts, allow processes to become runnable.

```c
2757 void
2758 scheduler(void)
2759 { 
2760  struct proc *p;
2761  struct cpu *c = mycpu();
2762  c->proc = 0;
2763  for (;;) {
2764     // Enable interrupts on this processor.
2765     sti();
2766     // Loop over process table looking for process to run.
2767     acquire(&ptable.lock);
2768     for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2769       if (p->state != RUNNABLE)
2770         continue;
2771     }
2772     // Switch to chosen process. It is the process's job
2773     // to release ptable.lock and then reacquire it
2774     // before jumping back to us.
2775     c->proc = p;
2776     switchcpu(p);
2777     p->state = RUNNING;
2778     switch(c->scheduler), p->context);
2779     switchkvm();
2780     // Process is done running for now.
2781     // It should have changed its p->state before coming back.
2782     c->proc = 0;
2783     release(&ptable.lock);
2784   }
2785 }
2786 ```
Summary

• Spinlocks in xv6 based on xchg atomic instruction
• Processes in kernel mode hold spinlock when accessing shared data structures, disabling interrupts on that core while lock is held
• Special ptable.lock held across context switch