Lecture 29: Locking in xv6

Mythili Vutukuru

IIT Bombay

https://www.cse.iitb.ac.in/~mythili/os/

Why locking in xv6?

P1 P2

kernel data

- 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

    P1 → kernel → intr
  - 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)

```
// Mutual exclusion lock.
struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters)
                     // that locked the lock.
};
```

```
void
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __sync_synchronize();

  // Record info about lock acquisition for debugging.
  lk->cpu = mycpu();
  getcallerpcs(&lk, lk->pcs);
}
```

=0 ✓ success

spin

C1 P1 → L → ~~intr.~~

C2 P2 → L spinning ✓

Disabling interrupts

L1 L2

- 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

```
// Pushcli/popcli are like cli/sti except that they are matched:
// it takes two popcli to undo two pushcli.  Also, if interrupts
// are off, then pushcli, popcli leaves them off.

void
pushcli(void)
{
  int eflags;

  eflags = readeflags();
  cli();
  if(mycpu()->ncli == 0)
    mycpu()->intena = eflags & FL_IF;
  mycpu()->ncli += 1;
}

void
popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli - interruptible");
  if(--mycpu()->ncli < 0)
    panic("popcli");
  if(mycpu()->ncli == 0 && mycpu()->intena)
    sti();
}
```

ptable.lock (1)

```
2409 struct {
2410   struct spinlock lock;
2411   struct proc proc[NPROC];
2412 } ptable;
```

- 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

ptable.lock (2)

- 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

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

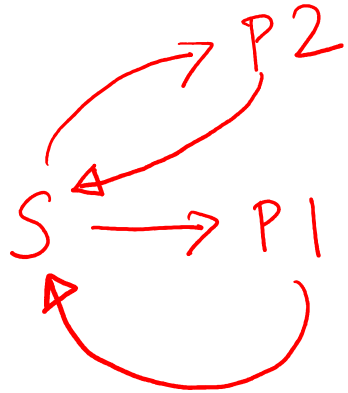
P1 → P2

```
void
forkret(void)
{
  static int first = 1;
  // Still holding ptable.lock from scheduler.
  release(&ptable.lock);

  if (first) {
    // Some initialization functions must be run in the context
    // of a regular process (e.g., they call sleep), and thus cannot
    // be run from main().
    first = 0;
    iinit(ROOTDEV);
    initlog(ROOTDEV);
  }
```

trapret

ptable.lock (3)



- 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.

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&ptable.lock);

  }
}
```

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