Building Concurrency Primitives

CS 450: Operating Systems
Sean Wallace <swallac6@iit.edu>
Previously...

1. Decided concurrency was a useful (sometimes necessary) thing to have.

2. Assumed the presence of concurrent programming “primitives” (e.g., locks).

3. Showed how to use these primitives in concurrent programming scenarios.
...but how are these primitives actually constructed?

- as usual: responsibility is shared between kernel and hardware.
Agenda

• The mutex lock

• xv6 concurrency mechanisms

• Code review: implementation & usage
The mutex lock
Thread A

```
   a1 count = count + 1
```

Thread B

```
   b1 count = count + 1
```

```
   use
   acquire
   allocated

T_A
count
T_B
```
• Basic requirement:

• Prevent other threads from entering their crucial section while one thread holds the lock

• I.e., execute critical section in mutex
Lock-polling—“spinlock”

```c
struct spinlock {
    int locked;
};

void acquire(struct spinlock *l) {
    while (1) {
        if (!l->locked) {
            l->locked = 1;
            break;
        }
    }
}

void release(struct spinlock *l) {
    l->locked = 0;
}
```
Problem: thread can be preempted between test & set operations

• Again, must guarantee execution of test & set in mutex...

• Using a...lock!?
Time for an alternative strategy!
Recognize that *preemption* is carried out by a hardware *interrupt*…

… so, disable interrupts!
• x86: interrupt flag (IF) in FLAGS register
  • Cleared/set by cli/sti instructions
  • Restored by iret instruction
  • Note: above are all \textit{privileged} operations — i.e., must be performed by kernel
One possible setup:

```
begin_mutex();
/* critical section */
end_mutex();
```

```
asm("cli");
asm("sti");
```
• Horrible idea!

• User code *cannot be preempted*; kernel effectively neutered

• Also, prohibits all concurrency (not just for related critical sections)
Should only block interrupts in kernel space, and minimize blocked timeframe

```c
void acquire(struct spinlock *l) {
    int done = 0;

    while (!done) {
        asm("cli");

        if (!l->locked) {
            done = l->locked = 1;
            break;
        }

        asm("sti");
    }
}

void release(struct spinlock *l) {
    l->locked = 0;
}
```
• But!

  • Preventing interrupts only helps to avoid concurrency *due to preemption*

  • Insufficient on a multiprocessor system!
    • Where we have true *parallelism*
    • Each processor has its own interrupts
asm("cli");
if (!l->locked) {
    done = l->locked = 1;
}
asm("sti");
Instead of a general mutex, recognize that all we need is to make test (read) & set (write) operations on lock atomic

```c
asm("cli");
if (!l->locked) {
    done = l->locked = 1;
}
asm("sti");
```
• E.g., x86 atomic exchange instruction (xchg)

• Atomically swaps register/memory content

• Guarantees no out-of-order execution

# note: pseudo-assembly!

loop:
  movl $1, %eax  # set up "new" value in reg
  xchgl l->locked, %eax  # swap values in reg & lock
  test %eax, %eax
  jne loop  # spin if old value != 0
void acquire(struct spinlock *lk)
{
    if(holding(lk))
        panic("acquire");

    while(xchg(&lk->locked, 1) != 0);

    ...
}

void release(struct spinlock *lk)
{
    if(!holding(lk))
        panic("release");

    lk->pcs[0] = 0;
    lk->cpu = 0;

    xchg(&lk->locked, 0);

    ...
}
xv6 uses spin locks *internally*

e.g., to protect proc array in scheduler:

```c
void scheduler(void) {
    ...
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE)
            continue;

        proc = p;
        switchuvm(p);
        p->state = RUNNING;
        swtch(&cpu->scheduler, proc->context);
    switchkvm();

        proc = 0;
    }
    release(&ptable.lock);
}
}
```

Maintains mutex across parallel execution of scheduler on separate CPUs
In theory, scheduler execution may also be interrupted by the clock...which causes the current thread to **yield**:

```c
// Give up the CPU for one scheduling round.
void yield(void)
{
    acquire(&ptable.lock);  //DOC: yieldlock
    proc->state = RUNNABLE;
    sched();
    release(&ptable.lock);
}
```
Ok, right?

```c
void scheduler(void) {
    acquire(&ptable.lock);
    ...
    release(&ptable.lock);
}

void yield(void) {
    acquire(&ptable.lock);
    ...
    release(&ptable.lock);
}
```
No!

Designed to enforce mutex *between threads*. If one thread tries to acquire a lock more than once, it will have to *wait for itself* to release the lock…

…which it can’t/won’t. Deadlock!
• xv6’s (ultra-conservative) policy:
  • Never hold a lock when interrupts are enabled
  • Corollary: Can only enable interrupts when all locks have been released (may hold more than one at any time)
void acquire(struct spinlock *lk) {
    pushcli();
    if(holding(lk))
        panic("acquire");

    while(xchg(&lk->locked, 1) != 0);
    . . .
}

void release(struct spinlock *lk) {
    if(!holding(lk))
        panic("release");
    . . .
    xchg(&lk->locked, 0);
    popcli();
}

void pushcli(void) {
    int eflags;

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

void popcli(void) {
    if(readeflags()&FL_IF)
        panic("popcli - interruptible");
    if(--cpu->ncli < 0)
        panic("popcli");
    if(cpu->ncli == 0 && cpu->intena)
        sti();
}
• Spinlock usage:
  • When to lock?
  • How long to hold onto a lock?
• Spinlocks are very inefficient!

• Lock polling is indistinguishable from “application” logic (e.g., scheduling)

• Scheduler will allocate entire time quanta to perform lock polling
Would like “blocking” logic

I.e., threads block on some condition and are not re-activated until necessary

- Push notification vs. continuous polling
xv6 implements `sleep` and `wakeup` mechanism for blocking threads on semantic “channels” (`proc.c`)

- distinct scheduler state (`SLEEPING`) prevents re-activation
```c
void sleep(void *chan, struct spinlock *lk)
{
    if(proc == 0)
        panic("sleep");

    if(lk == 0)
        panic("sleep without lk");

    // Must acquire ptable.lock in order to
    // change p->state and then call sched.
    // Once we hold ptable.lock, we can be
    // guaranteed that we won't miss any wakeup
    // (wakeup runs with ptable.lock locked),
    // so it's okay to release lk.
    if(lk != &ptable.lock){ //DOC: sleeplock0
        acquire(&ptable.lock); //DOC: sleeplock1
        release(lk);
    }

    // Go to sleep.
    proc->chan = chan;
    proc->state = SLEEPING;
    sched();

    // Tidy up.
    proc->chan = 0;

    // Reacquire original lock.
    if(lk != &ptable.lock){ //DOC: sleeplock2
        release(&ptable.lock);
        acquire(lk);
    }
}

// Wake up all processes sleeping on chan.
void wakeup(void *chan)
{
    acquire(&ptable.lock);
    wakeup1(chan);
    release(&ptable.lock);
}
```

```
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
static void wakeup1(void *chan)
{
    struct proc *p;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == SLEEPING && p->chan == chan)
            p->state = RUNNABLE;
```
Sample usage: \texttt{wait} (synchronize on termination, reap child) / \texttt{exit} (process termination)
// Wait for a child process to exit
// and return its pid.
// Return -1 if this process has no children.

int
wait(void)
{
    struct proc *p;
    int havekids, pid;

    acquire(&ptable.lock);
    for(;;){
        havekids = 0;
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->parent != proc)
                continue;
            havekids = 1;
            if(p->state == ZOMBIE){
                // Found one.
                pid = p->pid;
                kfree(p->kstack);
                p->kstack = 0;
                freevm(p->pgdir);
                p->state = UNUSED;
                p->pid = 0;
                p->parent = 0;
                p->name[0] = 0;
                p->killed = 0;
                release(&ptable.lock);
                return pid;
            }
        }
        if(!havekids || proc->killed){
            release(&ptable.lock);
            return -1;
        }

        sleep(proc, &ptable.lock);
    }
}

// Exit the current process. Does not return.
// An exited process remains in the zombie state
// until its parent calls wait() to find out it exited.
void
exit(void)
{
    struct proc *p;
    int fd;

    if(proc == initproc)
        panic("init exiting");

    // Close all open files.
    for(fd = 0; fd < NOFILE; fd++){
        if(proc->ofile[fd]){
            fileclose(proc->ofile[fd]);
            proc->ofile[fd] = 0;
        }
    }

    iput(proc->cwd);
    proc->cwd = 0;

    acquire(&ptable.lock);

    // Parent might be sleeping in wait().
    wakeup1(proc->parent);

    // Pass abandoned children to init.
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->parent == proc){
            p->parent = initproc;
            if(p->state == ZOMBIE)
                wakeup1(initproc);
        }
    }

    proc->state = ZOMBIE;
    sched();
    panic("zombie exit");