Building Concurrency Primitives

CS 450: Operating Systems
Michael Lee <lee@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
basic requirement: prevent other threads from entering their critical 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
- again, must guarantee execution of test & set in mutex ...
  (using a lock?!)
recognize that *preemption* is caused by a hardware *interrupt* …

… so, disable interrupts!
recall: x86 interrupt flag (IF) in FLAGS register

- cleared/set by cli/sti instructions

- restored by iret instruction

- note: above are all *privileged* operations — i.e., must be performed by kernel
can try to avoid spinlocks altogether:

```
asm("cli");

asm("sti");
```

```
begin_mutex();

/* critical section */

end_mutex();
```
horrible idea!

- user code cannot be preempted; kernel effectively neutered
- also, prohibits all concurrency (not just for related critical sections)
ought only block interrupts in kernel space, and minimize blocked time frame

```c
void acquire(struct spinlock *l) {
    int done = 0;
    while (!done) {
        asm ("cli");
        if (!l->locked)
            done = l->locked = 1;
        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
(fail)

asm ("cli");
if (!l->locked)
    done = l->locked = 1;
asm ("sti");
instead of general mutex, recognize that all we need is to make test (read) & set (write) operations on lock atomic

```c
if (!l->locked)
    done = l->locked = 1;
```
enter: x86 *atomic exchange* instruction (*xchg*)
- atomically *swaps* reg/mem content
- guarantees no out-of-order execution

```assembly
# 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
```
xv6: spinlock.c

```c
void acquire(struct spinlock *lk) {
    ...
    // keep looping until we atomically “swap” a 0 out of the lock
    while(xchg(&lk->locked, 1) != 0) ;
}

void release(struct spinlock *lk) {
    xchg(&lk->locked, 0);
    ...
}
```
xv6 uses spinlocks *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;
        swtch(&cpu->scheduler, proc->context);
    }
    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 \texttt{yield}:

```c
void yield(void) {
    acquire(&ptable.lock);
    proc->state = RUNNABLE;
    sched();
    release(&ptable.lock);
}
```
what could go wrong?

```c
void yield(void) {
    acquire(&ptable.lock);
    proc->state = RUNNABLE;
    sched();
    release(&ptable.lock);
}

void scheduler(void) {
    acquire(&ptable.lock);
    ...
    release(&ptable.lock);
}
```
Locks are 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 with interrupts enabled

- corollary: can only enable interrupts when *all* locks have been released (may hold more than one at any time)

- must be careful about re-enabling interrupts prematurely when releasing a lock
// maintain a "stack" of cli/sti calls

void pushcli(void) {
    int eflags;

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

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

void acquire(struct spinlock *lk) {
    pushcli();
    while(xchg(&lk->locked, 1) != 0) ;
    ...
}

void release(struct spinlock *lk)
{
    ...
    xchg(&lk->locked, 0);
    popcli();
}
spinlock usage:
- when to lock?
- how long to hold onto a lock?
spinlocks are very inefficient!

- lock polling is indistinguishable from application logic — will burn through scheduler time quanta

- not intended for long-term synchronization (e.g., “blocking”)
a “blocked” thread shouldn’t consume CPU cycles until some condition(s) necessary for it to run are true

e.g., data from I/O request is ready; child process ready for reaping by parent (via wait)
xv6 implements `sleep` and `wakeup` mechanism for blocking threads on semantic “channels” (`proc.c`)

- distinct scheduler state (`SLEEPING`) prevents re-activation
// Put calling process to sleep on chan
void
sleep(void *chan)
{
    proc->chan = chan;
    proc->state = SLEEPING;
    sched(); // context switch away from proc
    proc->chan = 0;
}

// Wake up all processes sleeping on chan.
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;
}

Q: What happens if sleep and wakeup are called simultaneously?

A: Race condition! Wakeup may be “lost”.
void
sleep(void *chan, struct spinlock *lk)
{
    // Acquire ptable.lock so we don't miss
    // and wakeups
    if(lk != &ptable.lock){
        acquire(&ptable.lock);
        release(lk);
    }

    // Go to sleep.
    proc->chan = chan;
    proc->state = SLEEPING;
    sched(); // note: scheduler releases lock

    proc->chan = 0;

    // Reacquire original lock.
    if(lk != &ptable.lock){
        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: `wait / exit`
// Wait for a child process to exit.
int
wait(void)
{
    struct proc *p;
    int havekids, pid;

    acquire(&ptable.lock);
    for(;;){
        for(p=ptable.proc; p<&ptable.proc[NPROC]; p++){
            if(p->parent != proc)
                continue;
            if(p->state == ZOMBIE){
                pid = p->pid;
                release(&ptable.lock);
                return pid;
            }
        }

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

// Exit the current process.
// An exited process remains a zombie until its
// parent calls wait() to find out it exited.
void
exit(void)
{
    struct proc *p;
    acquire(&ptable.lock);

    wakeup1(proc->parent);

    // Pass orphaned 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");
}