CS 134
Operating Systems

Feb 27, 2019

Sleep and Wakeup

Outline

- User-level thread switch homework
- Sequence coordination
  - xv6: sleep & wakeup
  - lost wakeup problem
  - termination

HW 8: xv6 uthreads

```
/* Switch from current_thread to next_thread. Make next_thread
 * the current_thread, and set next_thread to 0.
 * Use eax as a temporary register; it is caller saved.
 */
        .globl thread_switch
thread_switch:

        pushal                          /* save general registers */
        movl current_thread, %eax
        movl %esp, (%eax)

        movl next_thread, %eax
        movl %eax, current_thread
        movl (%eax), %esp
        popal                           /* pop general registers */
        ret                             /* pop return address */
```

HW 8: xv6 uthreads

```
void thread_schedule(void)
{
  thread_p t;
  /* Find another runnable thread. */
  next_thread = 0;
  for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
    if (t->state == RUNNABLE && t != current_thread) {
      next_thread = t;
      break;
    }
  }
  if (t >= all_thread + MAX_THREAD && current_thread->state == RUNNABLE) {
    /* The current thread is the only runnable thread; run it. */
    next_thread = current_thread;
  }
  if (next_thread == 0) {
    printf(2, "thread_schedule: no runnable threads\n");
    exit();
  }
  if (current_thread != next_thread) {         /* switch threads?  */
    next_thread->state = RUNNING;
    thread_switch();
  } else
    next_thread = 0;
}
```

Sequence coordination

- Threads need to wait for specific events or conditions:
  - Wait for disk read to complete
  - Wait for pipe reader(s) to make space in the pipe
  - Wait for any child to exit

- Don’t want a spin lock
  - Chews up CPU time

- Better: coordination primitives that yield the CPU
  - sleep/wakeup (xv6)
  - condition variables (HW 9), barriers (HW 9), etc.

Sleep and wakeup

- `sleep(chan, lock)`
  - sleeps on a “channel”: an address to name the condition we are sleeping on
- `wakeup(chan)`
  - wakes up all threads sleeping on `chan`
  - May wake more than on thread
  - No formal connection to the condition the sleeper is waiting on
    - `sleep()` may return even if the condition is true
    - Caller must treat `sleep()` returns as a hint

```
while (!condition)
  sleep(chan, lock);
```

Sleep/wakeup use in ide

```
void iderw(struct buf *b)
{
  …
  acquire(&idelock);
  …
  // Wait for request to finish.
  while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
    sleep(b, &idelock);
  }
  release(&idelock);
}
```

```
void ideintr(void)
{
  acquire(&idelock);
  …
  // Wake process waiting for b.
  b->flags |= B_VALID;
  b->flags &= ~B_DIRTY;
  wakeup(b);
  …
  release(&idelock);
}
```

Lost wakeup

```
void iderw(struct buf *b)
{
  …
  acquire(&idelock);
  …
  // Wait for request to finish.
  while((b->flags & (B_VALID|B_DIRTY))
      != B_VALID){
    release(&idelock);
    broken_sleep(b);
  }
  release(&idelock);
}
```

```
void ideintr(void)
{
  acquire(&idelock);
  …
  // Wake process waiting for b.
  b->flags |= B_VALID;
  b->flags &= ~B_DIRTY;
  wakeup(b);
  …
  release(&idelock);
}
```

```
void
broken_sleep(void *chan)
{
  struct proc *p = myproc();
  if(p == 0)
    panic("sleep");
  // Must acquire ptable.lock in order to
  // change p->state and then call sched.
  acquire(&ptable.lock);
  p->chan = chan;
  p->state = SLEEPING;
  sched();
  // Tidy up.
  p->chan = 0;
  release(&ptable.lock);
}
```

```
void wakeup(void *chan)
{
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC];
      p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
  release(&ptable.lock);
}
```

Solution to lost wakeup

- Goal: lock out wakeup for entire time between condition check and `state = SLEEPING`
- Release the condition lock while asleep
- xv6 strategy:
  - Require `wakeup` to hold lock on condition and ptable.lock
  - sleeper at all times holds one or the other lock
    - can release condition lock after it holds the ptable lock
  - While `wakeup` checks for `SLEEPING` threads, both locks are held.

Solution to lost wakeup

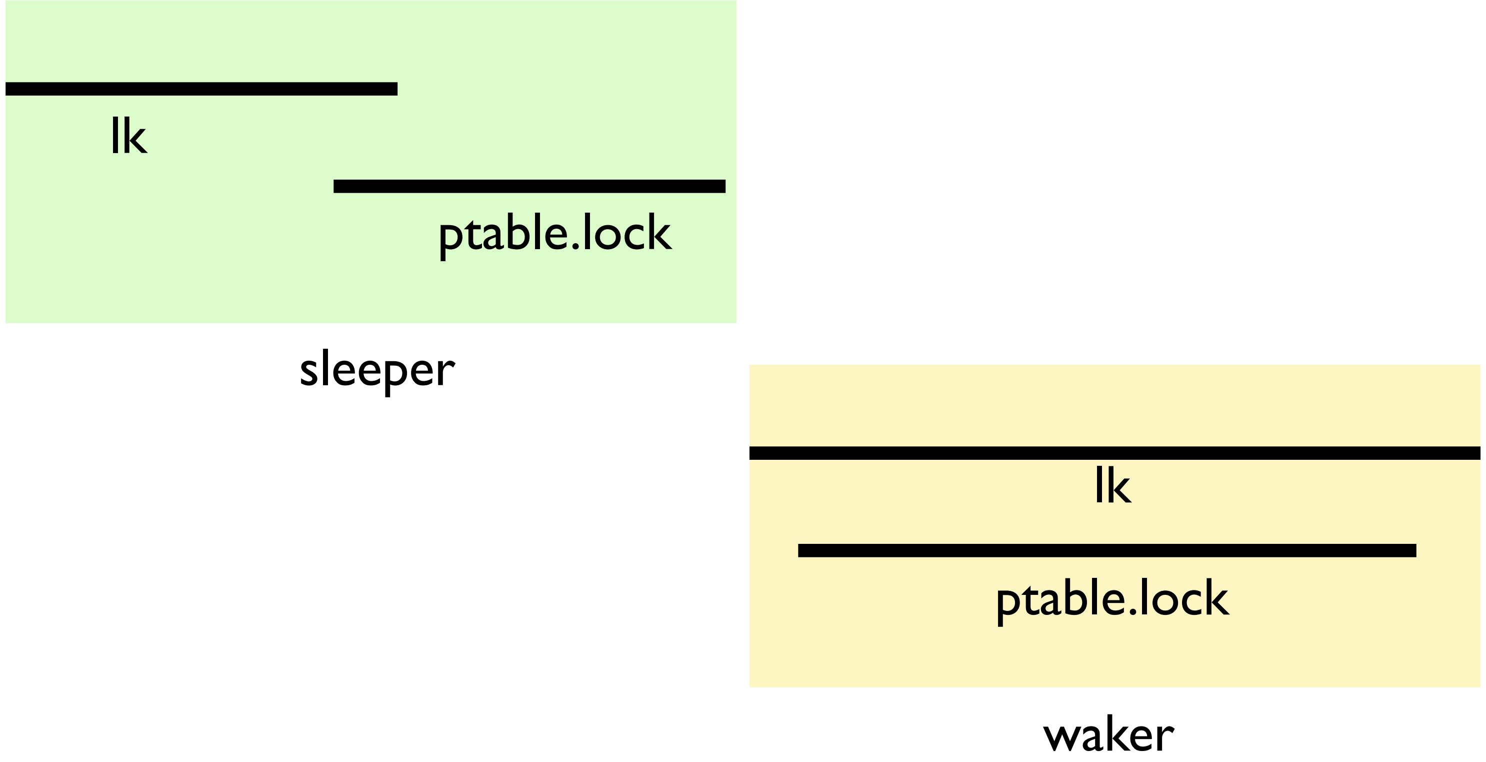
```
void iderw(struct buf *b)
{
  …
  acquire(&idelock);
  …
  // Wait for request to finish.
  while((b->flags & (B_VALID|B_DIRTY))
        != B_VALID){
    sleep(b, &idelock);
  }
  release(&idelock);
}
```

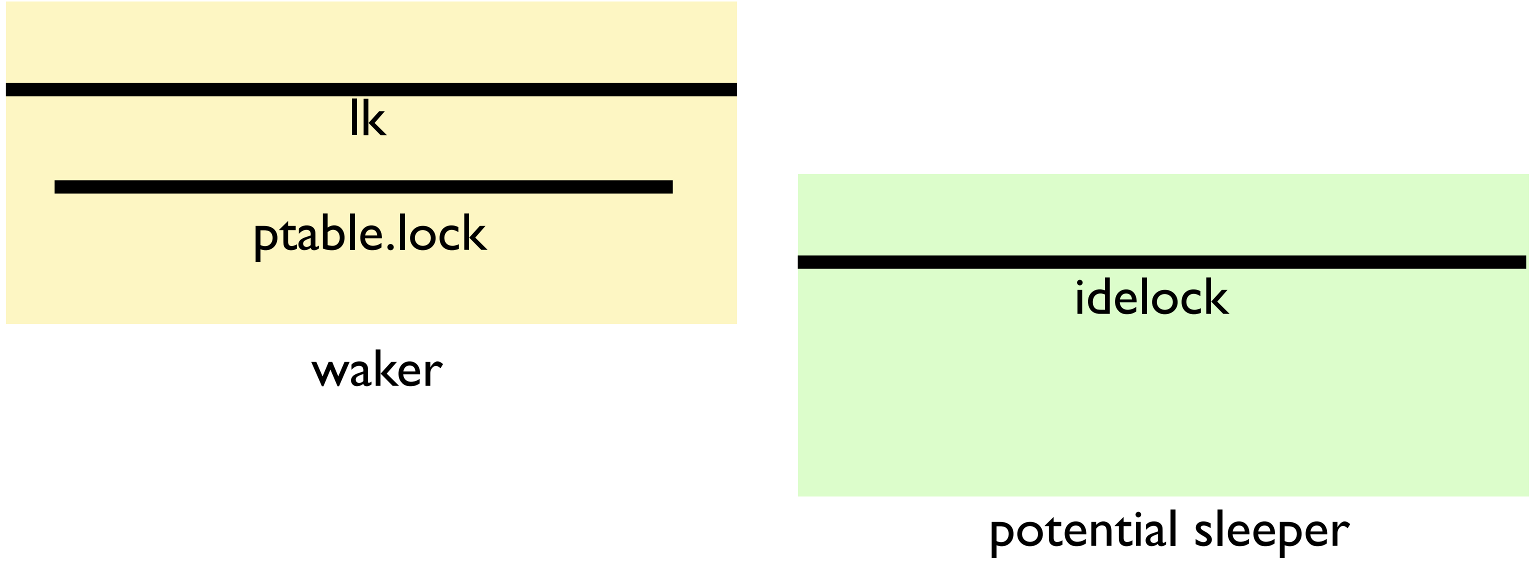
```
void sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();
  if(p == 0)
    panic("sleep");
  // Must acquire ptable.lock in order to
  // change p->state and then call sched.
  acquire(&ptable.lock);
  release(lk)
  p->chan = chan;
  p->state = SLEEPING;
  sched();
  // Tidy up.
  p->chan = 0;
  release(&ptable.lock);
  acquire(lk);
}
```

```
void ideintr(void)
{
  acquire(&idelock);
  …
  // Wake process waiting for b.
  b->flags |= B_VALID;
  b->flags &= ~B_DIRTY;
  wakeup(b);
  …
  release(&idelock);
}
```

```
void wakeup(void *chan)
{
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC];
      p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
  release(&ptable.lock);
}
```

Solution to lost wakeup





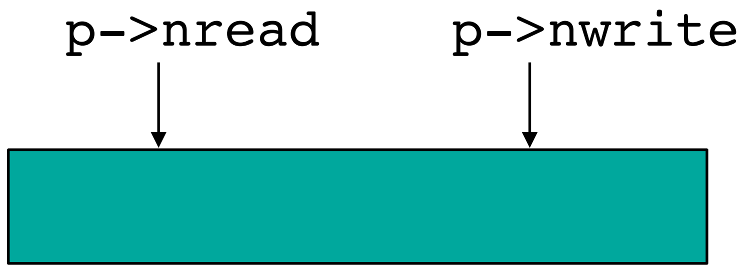
Many sequence-coordination primitives

- Counting semaphores
- Condition variables (similar to sleep/wake)
- Wait queues (Linux kernel)

Another sequence coordination problem: pipe

```
int pipewrite(struct pipe *p, char *addr, int n)
{
  acquire(&p->lock);
  for(int i = 0; i < n; i++){
    while(p->nwrite == p->nread + PIPESIZE){
      if(p->readopen == 0 || myproc()->killed){
        release(&p->lock);
        return -1;
      }
      wakeup(&p->nread);
      sleep(&p->nwrite, &p->lock);
    }
    p->data[p->nwrite++ % PIPESIZE]
      = addr[i];
  }
  wakeup(&p->nread);
  release(&p->lock);
  return n;
}
```

```
int piperead(struct pipe *p, char *addr,
  int n)
{
  acquire(&p->lock);
  while(p->nread == p->nwrite &&
      p->writeopen){
    if(myproc()->killed){
      release(&p->lock);
      return -1;
    }
    sleep(&p->nread, &p->lock);
  }
  for(int i = 0; i < n; i++){
    if(p->nread == p->nwrite) break;
    addr[i] =
      p->data[p->nread++ % PIPESIZE];
  }
  wakeup(&p->nwrite);
  release(&p->lock);
  return i;
}
```



circular buffer: p->data

Another sequence coordination problem: terminating a sleeping thread

- May not be safe to forcibly terminate process
  - Might be executing in kernel w/ kernel stack, PT
  - Might be in critical section (needs to restore invariants)
  - Can’t immediately terminate it

- Tell proc to exit at next convenient point
  - Gets to keep running until next system call or timer interrupt

```
int kill(int pid)
{
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC];
      p++){
    if(p->pid == pid){
      p->killed = 1;
      // Wake process from sleep if necessary.
      if(p->state == SLEEPING)
        p->state = RUNNABLE;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

Thread cleanup

```
void trap(struct trapframe *tf) {
   if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
  …
  if(myproc() &&
     myproc()->killed &&
     (tf->cs&3) == DPL_USER)
    exit();
  …
}
```

```
void exit(void)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

  if(curproc == initproc)
    panic("init exiting");
  // clean up open file descriptors
  // Parent might be sleeping in wait().
  wakeup1(curproc->parent);

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

  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

What if kill target is sleeping?

- Could be waiting for console input, or in `wait()`, or in `iderw()`
- Wake it up (change from SLEEPING to RUNNABLE)
  - Want it to exit immediately
  - But, maybe sleeping target is halfway through complex operation that (for consistency) must complete (e.g., creating a file)

What if kill target is sleeping? xv6 solution

- Some sleep locks check for killed (`piperead`, `pipewrite`, `consoleread`, `sys_sleep`)

```
int pipewrite(struct pipe *p, char *addr, int n)
{…
  while(p->nwrite == p->nread + PIPESIZE){
      if(p->readopen == 0 || myproc()->killed){
        release(&p->lock);
        return -1;
      }
     sleep(&p->nwrite, &p->lock);
    }…
}
```

- Some don't: `iderw`
  - If reading, FS expects to see data in disk buf
  - If writing, FS might be in the middle of a `create`

```
void iderw(struct buf *b)
{ …
 while((b->flags & (B_VALID|B_DIRTY))
      != B_VALID){
    sleep(b, &idelock);
  }…
}
```

xv6 spec for `kill`

- If target is in user code:
  - Will exit next system call or timer interrupt
- If target is in kernel code:
  - Won’t ever execute more user code
  - But may spend a while in kernel code

How does JOS deal with these problems?

- Lost wakeup:
  - JOS interrupts are disabled in the kernel
    - so wakeup can’t sneak in between condition check and sleep
- Termination while blocking:
  - JOS has only a few system calls and they are simple
    - No blocking multi-step operations like `create`
      - No file system or disk driver in the kernel
    - Really only one blocking call: IPC `ipc_recv`
    - `ipc_recv` leaves env in an `ENV_NOT_RUNNABLE` state where it can be safely destroyed