CS 347M (Operating Systems Minor)

Spring 2022

Lecture 15: Sleep and wakeup in xv6

Mythili Vutukuru

CSE, IIT Bombay

Locks, sleep/wakeup in xv6

- xv6 does not have userspace threads, only single threaded processes
- But multiple processes may be kernel mode on different CPU
  - Need locks to protect access to shared kernel data structures
- OS also needs a mechanism to let processes sleep (e.g., when process makes blocking disk read syscall) and wakeup when some events occur (e.g., disk has raised interrupt and data is ready)
  - Needs sleep/wakeup functions for processes in kernel mode (not userspace)
  - Process P1 in kernel mode calls sleep to give up CPU, gets blocked until event
  - Another process P2 (in kernel mode) wakes up P1 when the event occurs
- This lecture: more on xv6 locks, sleep, wakeup, …

Recap: Context switching in xv6 (1)

- Every CPU has a scheduler thread (special process that runs scheduler code)
- Scheduler goes over list of processes and switches to one of the runnable ones
- The special function “swtch” performs the actual context switch
  - Save context on kernel stack of old process
  - Restore context from kernel stack of new process

```
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);

  }
}
```

Recap: Context switching in xv6 (2)

- After running for some time, the process switches back to the scheduler thread, when:
  - Process has terminated (exit system call)
  - Process needs to sleep (e.g., blocking read system call)
  - Process yields after running for long (timer interrupt)
- Process calls “sched” which calls “swtch” to switch to scheduler thread again
- Scheduler thread runs its loop and picks next process to run, and the story repeats

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

```
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;

  sched();

```

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

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 in some way, 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

P1 $\longrightarrow$ scheduler $\longrightarrow$ P2

Acquire ptable.lock | Release ptable.lock

ptable.lock (2)

- Every function that calls sched() to give up CPU will do so with ptable.lock held
- Which functions invoke sched() to give up CPU?
  - Yield: process gives up CPU due to timer interrupt
  - Sleep: when process wishes to block
  - Exit: when process terminates
- Every function where a process resumes after being scheduled release ptable.lock
- What functions does a process resume after swtch?
  - Yield: resuming process after yield is done
  - Sleep: resuming process that is waking up after sleep
  - Forkret: for newly created processes
- Purpose of forkret: to release ptable.lock
  - New process then returns from trap like its parent

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

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

  if (first) {
    // Some initialization functions must be run i
    // of a regular process (e.g., they call sleep
    // be run from main().
    first = 0;
    iinit(ROOTDEV);
    initlog(ROOTDEV);
  }
```

ptable.lock (3)

P1 → scheduler → P2

Acquire ptable.lock

Release ptable.lock

- Scheduler goes into loop with lock held
- Acquire ptable.lock in P1 → scheduler picks P2 → release in P2
- Later, acquire ptable.lock in P2 → scheduler picks P3 → release in P3
- 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);

  }
}
```

Sleep and wakeup in xv6

- A process P1 that wishes to block and give up CPU calls "sleep" function
  - Example: process reads a block from disk, must block until disk read completes
  - Read syscall → sleep → sched() to give up CPU
- Another process P2 calls "wakeup" when event to block P1 occurs
  - P2 calls wakeup → marks P1 as runnable, no context switch immediately
  - Example: disk interrupt occurred when P2 is running, P2 runs interrupt handler, which will call wakeup
- How does P2 know which process to wake up? When P1 sleeps, it sets a channel (void * chan) in its struct proc, P2 calls wakeup on same channel
  - Channel = any value known to both P1 and P2
  - Example: channel value for disk read can be address of disk block
- Spinlock protects atomicity of sleep: P1 calls sleep with some spinlock L held, P2 calls wakeup with same spinlock L held

Sleep function

- Two arguments: channel to sleep on, a spinlock to protect atomicity of sleeping
- Acquire ptable.lock, release the lock given to sleep (make it available for wakeup)
  - Unless lock given is ptable.lock itself, in which case no need to acquire again
  - One of two locks held at all times
- Sleep calls sched() to give up CPU
  - Needs to hold ptable.lock when calling sched()
- Calls sched(), switched out of CPU, resumes again when woken up and ready to run
- Reacquires the lock given to sleep and returns back
  - Code that invoked sleep with lock held returns with lock held again

```
// Atomically release lock and sleep on chan.
// Reacquires lock when awakened.
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  if(p == 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){
    acquire(&ptable.lock);
    release(lk);
  }
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;

  sched();

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

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

Wakeup function

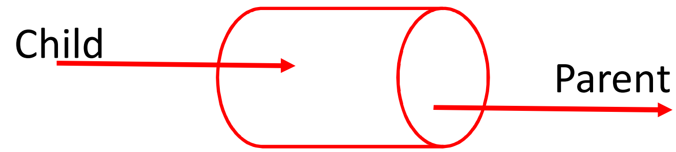
- Sleep and wakeup called by processes with same lock held (protect atomicity of sleep)
- Wakeup acquires ptable.lock, changes ptable to mark process matching channel as runnable, releases ptable.lock
  - If lock protecting atomicity of sleep is ptable.lock itself, then directly call wakeup1
- Sleep holds one of sleep's lock or ptable.lock at all times, so a wakeup cannot run in between sleep
- Wakes up all processes sleeping on a channel in ptable (more like signal broadcast of condition variables)
  - Good idea to check condition is still true upon waking up (use while loop while calling sleep)

```
// 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;
}

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

Example: pipes (1)



- xv6 provides anonymous pipes for IPC between parent and child processes
  - E.g., Parent P and child C share anonymous pipe
  - Child C writes into pipe, parent P reads from pipe
- Internal implementation inside kernel
  - Common shared buffer, protected by a spinlock
  - Write system call stores data in shared buffer
  - Read system call returns data from shared buffer
- Sleep and wakeup involved in read/write
  - Pipe read sleeps if pipe is empty, pipe write wake up
  - Pipe write sleeps if pipe is full, pipe read wakes up

```
struct pipe {
  struct spinlock lock;
  char data[PIPESIZE];
  uint nread;      // number of bytes read
  uint nwrite;     // number of bytes written
  int readopen;    // read fd is still open
  int writeopen;   // write fd is still open
};
```

```
//userspace code

int fd[2]
pipe(fd) //syscall to create pipe

int ret = fork()

if(ret == 0) {//child
   close(fd[0]) //close read end
   write(fd[1], message, ..)
}
else {//parent
   close(fd[1]) //close write end
   read(fd[0], message, ..)
}
```

Example: pipes (2)

- Implementation of pipe read and write system calls using sleep/wakeup
  - Similar to producer-consumer logic of last class
  - Channel for sleep/wakeup = address of pipe structure variables (can be

```
int
pipewrite(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  for(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;
}
```

pipe is full

writer's channel for sleep is address of nwrite variable

```
int
piperead(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  while(p->nread == p->nwrite && p->writeopen){
    if(myproc()->killed){
      release(&p->lock);
      return -1;
    }
    sleep(&p->nread, &p->lock);
  }
  for(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;
}
```

pipe is empty

pipe lock protects atomicity of sleep

Example: wait and exit

- If wait called in parent while children are running, parent calls sleep and gives up CPU
  - Here, channel is parent struct proc address, lock given to sleep is ptable.lock

```
// Wait for children to exit.  (See wakeup1 call in proc_exit.)
sleep(curproc, &ptable.lock);
```

- In exit, child acquires ptable.lock and wakes up sleeping parent using its channel

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

- Here, lock given to protect atomicity of sleep is ptable.lock itself (convenient to do so)
  - Double locking of ptable.lock is avoided during sleep and wakeup
- Why is terminated process memory cleaned up by parent?
  - When a process calls exit, kernel stack, page table etc are in use, all this memory cannot be cleared until terminated process has been taken off the CPU
  - Parent code in wait is a good place to clean up child memory after child has stopped running