CS 134
Operating Systems

Feb 25, 2019

Process, threads, and scheduling

Homework 7: xv6 locks—iderw

- What goes wrong with adding sti after acquire and cli() after release()?
  - Let's see
- What would happen if acquire didn't check holding and panic?
  - Let's see
- What happens to the interrupt in the original code?
- What if IDE interrupt had occurred on a different core?

Spin-locks and interrupts

```
void
acquire(struct spinlock *lk)
{
  pushcli(); // To avoid deadlock.
 …
}
```

```
void
release(struct spinlock *lk)
{
  …
  popcli();
}
```

```
// 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();
}
```

Homework 7: xv6 locks—filealloc

- What happens if interrupts on while holding file table lock?
  - Nothing seems to happen.
  - However, if set breakpoint in gdb while holding locks and interrupts enabled, we can get a panic

Process

- Abstract virtual machine with its own:
  - CPU
  - Memory
- Motivated by isolation
- API:
  - `fork`
  - `exec`
  - `wait`
  - `kill`
  - `sbrk`
  - `getpid`

Challenge: more processes than processors

- E.g., your laptop has two processors and you want to:
  - run editor
  - run compiler
  - play music
- Must multiplex N processes among M (possibly <N) processors
- Called time-sharing (or context switching, or scheduling)

Goals

- Transparent to user processes
  - Doesn’t break virtual machine illusion
- Preemptive for user processes
  - No need to call `yield`
- Preemptive for kernel, where convenient
  - Helps keep system responsive

xv6 solution

- 1 user thread and 1 kernel thread per process
- 1 scheduler thread per CPU
- n processors

- So, 3 processes on 2 processors, how many total threads: 8

What is a thread

- Either:
  - CPU core executing (with registers and stack)
- Or:
  - Saved set of registers and stack that could execute

Overview of xv6 process switching

- User → kernel thread (how?)
- Kernel thread yields, due to preemption or waiting for I/O
- kernel thread → scheduler thread
- scheduler thread finds a RUNNABLE kernel thread

If no RUNNABLE thread could be found?

- scheduler thread → kernel thread
- kernel thread -> user

You cannot go directly from user thread to scheduler, or from scheduler thread to user thread. You would have to go through the kernel thread.

xv6 process states

- proc->state
  - `RUNNING`
  - `RUNNABLE`
  - `SLEEPING`
  - `ZOMBIE`
  - `UNUSED`

Context switching hard to get right

- Interrupts
- Locking
- Multi-core
- Process termination

Demonstrating preemptive switching

- Timer interrupt
- We'll run QEMU with one CPU
- We'll see how xv6 context-switches between the two processes

```
#include "types.h"
#include "user.h"

int main() {
  if (fork() == 0) {
    for (;;) {
    }
  } else {
    for (;;) {
    }
  }
  return 0;
}
```

hog.c

Demonstrating preemptive scheduling

- switch—to scheduler thread
  - a context holds a non-executing kernel thread's saved registers
  - xv6 contexts always live on the stack
  - context pointer is effectively the saved esp
    - Where are user registers?

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

proc.h

Why no need to save eax, ecx, edx?

Demonstrating preemptive scheduling

```
# void swtch(struct context **old, struct context *new);
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

swtch.S

Why not save %eip?

```c
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);
  }
}
```

Question

- What is the scheduling policy?
  - Will the thread that called yield run immediately again?

Question

- Why does scheduler release after loop and re-acquire immediately after?

```
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++){
      …
    }
    release(&ptable.lock);
  }
}
```

Question

- Why does scheduler briefly enable interrupts at beginning of loop?

```
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++){
      …
    }
    release(&ptable.lock);
  }
}
```

Question

- Why does the yield in one thread acquire the ptable.lock, but another thread releases it?

```
void scheduler(void)
{
  …
  for(;;){
    …
   acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      // 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);
  }
}
```

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

Coroutines

- `sched` and `scheduler` are *coroutines*
  - Flow control is passed between the two functions without returning
  - When either one calls `swtch`, the other continues executing where it last left off
  - Each one knows who it is `swtch`ing to, and who it was `swtch`ed from
  - Thus, they can cooperate on locking and unlocking `ptable.lock`

Process invariants

- If a process is RUNNING
  - CPU registers hold process's register values
    - Including %esp and %cr3
- If process is RUNNABLE
  - an idle CPU's scheduler must be able to run it
    - p->context must hold process's kernel thread variables
    - No CPU is executing on the process's kernel stack
    - No CPUs %cr3 holds the process's page table
    - No CPUs `proc` refers to the process

Question

- Is there preemptive scheduling of kernel threads?
  - What if timer interrupt while executing in the kernel?
  - What does the kernel thread stack look like?

Question

- Why no locks (other than ptable.lock) can be held when yielding the CPU?
  - acquire may waste a lot of time spinning, waiting for a lock held by a non-running thread
  - Worse: deadlock can occur since acquire waits with interrupts off

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  …
}
```

Thread cleanup

```
// Kill the process with the given pid.
// Process won't exit until it returns
// to user space (see trap in trap.c).
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;
}
```

Kill doesn't free resources (close open fds, release memory, etc.). Process must kill itself

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");
}
```

Thread cleanup, part 2

```
int wait(void)
{
  …
 acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc)
        continue;
     if(p->state == ZOMBIE){
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        release(&ptable.lock);
        return pid;
      }
    }
    …
  }
}
```

What if parent never waits?

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

  if(curproc == initproc)
    panic("init exiting");
  …
  // 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);
    }
  }
  …
}
```

```
int main(void)
{
  …
  while((wpid=wait()) >= 0 && wpid != pid)
    printf(1, "zombie!\n");
}
```

init.c