Paul Kummer

Dr. Hanku Lee

Operating Systems CSIS 430-01

11/19/2021

Lab 09: Implementing Priority Scheduler

Purpose

This lab's purpose is to teach students how to implement a priority scheduler in xv6. Additionally, students will learn about why the priority scheduler works.

Code

For making the priority scheduler work, only the proc.c file needed to be modified. Other files were reviewed, but not changed.

Main.c: main procedure

```
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // detect other processors
  lapicinit();     // interrupt controller
  seginit();       // segment descriptors
  picinit();       // disable pic
  ioapicinit();    // another interrupt controller
  consoleinit();   // console hardware
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  ideinit();       // disk
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  mpmain();        // finish this processor's setup
}
```

In the Main.c file there is a main procedure that gets called when starting the OS. This procedure initializes many other component to the OS and at the end it calls the mpmain() procedure.

Main.c: mpmain procedure

```
// Common CPU setup code.
static void
mpmain(void)
{
  cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
  idtinit();       // load idt register
  xchg(&(mycpu()->started), 1); // tell startothers() we're up
  scheduler();     // start running processes
}
```

Near the end of the mpmain() procedure the scheduler is called, which will pick what processes should run.

Proc.c: Scheduler

```
void scheduler(void)
{
  struct proc *p;
  struct proc *p1;
  struct proc *highP;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;)
  {
    // Enable interrupts on this processor.
    sti();
    highP = NULL;

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      //skip over any process that isn't runnable and select the highest priority
      if(p->state != RUNNABLE)
      {
        continue;
      }

      highP = p;

      for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++)
      {
        //skip over any process that isn't runnable and select the highest priority
        if(p1->state != RUNNABLE)
        {
          continue;
        }
        if(highP->priority < p1->priority)
        {
          highP = p1;
        }
      }

      // 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 = highP;
      switchuvm(highP);
      highP->state = RUNNING;

      swtch(&(c->scheduler), highP->context);
      switchkvm();
```

The scheduler picks what process will run next. In the above picture, the scheduler has been modified to pick processes based on a priority algorithm. The scheduler will run forever and when its running, it will iterate over every process that exists. It will then compare every runnable process against every other runnable process to determine what has the highest priority level. Once the highest priority has been found the scheduler will switch to that process for a given quantum or until an interrupt happens.

Trap.c

```
97    // Force process exit if it has been killed and is in user space.
98    // (If it is still executing in the kernel, let it keep running
99    // until it gets to the regular system call return.)
100   if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
101     exit();
102
103   // Force process to give up CPU on clock tick.
104   // If interrupts were on while locks held, would need to check nlock.
105   if(myproc() && myproc()->state == RUNNING &&
106      tf->trapno == T_IRQ0+IRQ_TIMER)
107     yield();
108
109   // Check if the process has been killed since we yielded
110   if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
111     exit();
```

When a process has used its quantum, the trap procedure will be called, causing the yield procedure to be called, which makes the process give up the CPU.

Proc.c: Yield procedure

```
455   // Give up the CPU for one scheduling round.
456   void
457   yield(void)
458   {
459     acquire(&ptable.lock);  //DOC: yieldlock
460     myproc()->state = RUNNABLE;
461     sched();
462     release(&ptable.lock);
463   }
```

The yield function will cause the current process to give up the CPU for one scheduling round. It does that by changing the process' state to runnable and then saving the process' registers. This will give other processes a chance to run, especially if a processes priority may have changed to something higher than the current process in a priority scheduling algorithm.

Execution

```
ps
name     pid     state           priority
init     1       SLEEPING        10
sh       2       SLEEPING        10
myfork   24      RUNNING         10
ps       25      RUNNING         10
myfork   20      SLEEPING        10
myfork   21      RUNNABLE        10
myfork   23      SLEEPING        10
$ cpr 21 11;
 pid=21, pr=11
$ ps
name     pid     state           priority
init     1       SLEEPING        10
sh       2       SLEEPING        10
myfork   24      RUNNABLE        10
ps       28      RUNNING         10
myfork   20      SLEEPING        10
myfork   21      RUNNING         11
myfork   23      SLEEPING        10
$ cpr 24 40;
 pid=24, pr=40
ps
ps
ps
ps
zombie!
$ name   pid     state           priority
init     1       SLEEPING        10
sh       2       SLEEPING        10
myfork   24      RUNNING         40
ps       31      RUNNING         10
myfork   23      SLEEPING        10
$ name   pid     state           priority
```

In this example, the process 21 originally is runnable and other processes are running. However, its priority is changed to a priority higher than all other processes and it begins to run. Next process 24 has its priority changed to the highest priority, and it eventually starts running after the zombie process. The zombie process happens when a process dies. The process'

parent is suppose to be notified that its child has died and call wait(). This allows the parent to clean up the dead process and retrieve information about the process. If the parent doesn't clean up the dead process by waiting, then the zombie doesn't get removed.

Conclusion

This lab taught me how to create a simple priority scheduler for xv6. Also, it taught me about some of the other procedures, and how and why they work. I did a little more follow up reading to investigate why the zombie processes occur and what is done about them. I do not know why the OS doesn't allow an interrupt after the second priority is changed. I think it may have something to do with the zombie process.