Paul Kummer

Dr. Hanku Lee

Operating Systems CSIS 430-01

10/21/2021

Lab 06: XV6 Function Overview

Purpose of Lab

The purpose of this lab is to review the files and code of the operating system xv6. The functions in the files pertain to processes and other system calls.

Functions Within Files

proc.h

Contains the declarations of structs to be used in proc.c. The first struct stores information for each cpu used by the OS. It holds information about the scheduler, state of the tasks, interrupts, and what process is currently being run, if any. Next there is an enum that holds different states that a process can be in. Then there is a struct that holds information about processes. It stores the size of the process, page table, process status, process identifier, open files, current working directory, and the process name.

cpu

```
// Per-CPU state
struct cpu {
  uchar apicid;                // Local APIC ID
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  volatile uint started;       // Has the CPU started?
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?
  struct proc *proc;           // The process running on this cpu or null
};
```

Holds the information about each cpu used by the OS. It holds a scheduler and information about its current running process.

proc

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

Holds information about a process. Some information that a process contains is its current working directory, process id, its size, current running state, and its parent.

procstate

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

Holds the five different states that a process can be in

proc.c

Has more structs to be used by processes and defines functions to be used by processes. The file defines how a process can read and disable interrupts. Also, it allows a process to be stored in an empty spot in the process table if one exists by calling allocproc(). Also, when it accesses the ptable, it acquires a lock while reading or modifying the table and then releases the lock after a process is done. There is userinit(), which sets up the first process. Then there is a growproc(), which expands the amount of memory allocated for a process. A current process can be accessed with the myproc() function. The most renowned function a process can use is the fork() function. This function will copy the current process and create a new duplicated process as a child of the original. The process id will be different than its parent, and the pid is returned upon completion of the fork() call. Another function, used to terminate a process is exit(). The exiting process will wait to close until the parent knows it is going to terminate, and be in a zombie state. A wait() function is used to make a parent process wait for its child process to finish. If the wait() is not successful, -1 will be returned. To determine what process should fun next, the function scheduler is called. The scheduler function, scheduler(), runs immediately after a cpu is setup, and runs indefinitely. When a process is changed, the current process will call a switch function. There is a sched() function that performs a similar function as scheduler().

allocproc

```
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;

  release(&ptable.lock);

  // Allocate kernel stack.
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;

  return p;
}
```

Allocates a process into a part of memory. It searches for any empty space and puts the process in that location. Then the process state is changed to embryo.

cpuid

```
int
cpuid() {
  return mycpu()-cpus;
}
```

Returns the pid of the current process.

exit

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

  if(curproc == initproc)
    panic("init exiting");

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(curproc->cwd);
  end_op();
  curproc->cwd = 0;

  acquire(&ptable.lock);

  // 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");
}
```

Begins terminating a process. A parent process will wake up in order to let the parent know the child is being exited. All of a process's children will be sent to init.

fork

```
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  pid = np->pid;

  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);

  return pid;
}
```

Duplicates a current process but will be a child of the process that called fork. The new process will also have its own pid.

growproc

```
int
growproc(int n)
{
  uint sz;
  struct proc *curproc = myproc();

  sz = curproc->sz;
  if(n > 0){
    if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  } else if(n < 0){
    if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  }
  curproc->sz = sz;
  switchuvm(curproc);
  return 0;
}
```

Allocates more space for the current process.

kill

```
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;
}
```

Terminates the current process. The process will acquire a lock on the ptable and change its kill state to true. Then when allocation is needed to be done, the scheduler will know it can use this old process for a new process.

mycpu

```
struct cpu*
mycpu(void)
{
  int apicid, i;

  if(readeflags()&FL_IF)
    panic("mycpu called with interrupts enabled\n");

  apicid = lapicid();
  // APIC IDs are not guaranteed to be contiguous. Maybe we should have
  // a reverse map, or reserve a register to store &cpus[i].
  for (i = 0; i < ncpu; ++i) {
    if (cpus[i].apicid == apicid)
      return &cpus[i];
  }
  panic("unknown apicid\n");
}
```

Returns the currently running cpu.

myproc

```
struct proc*
myproc(void) {
  struct cpu *c;
  struct proc *p;
  pushcli();
  c = mycpu();
  p = c->proc;
  popcli();
  return p;
}
```

Returns the currently running process.

pinit

```
void
pinit(void)
{
  initlock(&ptable.lock, "ptable");
}
```

Acquires a lock on the ptable so a process can be initialized.

ptable

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

Tells the cpu what process currently has access to its resources.

sched

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

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

Another version of scheduler, that handles certain conditions.

scheduler

```
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);

  }
}
```

Determines what process gets to use the cpu next. A process will be selected from the ptable, and the currently running process will be switched out.

sleep

```
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){  //DOC: sleeplock0
    acquire(&ptable.lock);  //DOC: sleeplock1
    release(lk);
  }
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;

  sched();

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

  // Reacquire original lock.
  if(lk != &ptable.lock){  //DOC: sleeplock2
    release(&ptable.lock);
    acquire(lk);
  }
}
```

Locks the ptable and suspends a process from running.

userinit

```
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();

  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  // this assignment to p->state lets other cores
  // run this process. the acquire forces the above
  // writes to be visible, and the lock is also needed
  // because the assignment might not be atomic.
  acquire(&ptable.lock);

  p->state = RUNNABLE;

  release(&ptable.lock);
}
```

Initializes a new process and sets its state to runnable, so the scheduler can pick it to run.

wait

```
int
wait(void)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();

  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;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        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;
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || curproc->killed){
      release(&ptable.lock);
      return -1;
    }

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

Stops a process from continuing execution until its children have finished. If there are not children, nothing will happen and an error value of -1 will be returned.

wakeup

```
void
wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}
```

Acquires locks on the ptable so that a process can wakeup from a sleep correctly. Wakeup1 will be called, which will change the process's state to runnable.

yield

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

Causes a process to give up its cpu time to the next process the scheduler picks.

sysproc.c

This file provides the OS with a way of starting up some processes. It utilizes many of the functions that were defined in the proc.c file.

sys_sbrk

```
int
sys_sbrk(void)
{
  int addr;
  int n;

  if(argint(0, &n) < 0)
    return -1;
  addr = myproc()->sz;
  if(growproc(n) < 0)
    return -1;
  return addr;
}
```

Increases the size in memory for the current process.

syscall.c

This file provides system calls used by the kernel. There are fetch functions that will retrieve and integer from an address or the length of a string and set a pointer to point to the string. Also, there are functions that deal with command line arguments.

argint, argptr, argstr

```
// Fetch the nth 32-bit system call argument.
int
argint(int n, int *ip)
{
  return fetchint((myproc()->tf->esp) + 4 + 4*n, ip);
}

// Fetch the nth word-sized system call argument as a pointer
// to a block of memory of size bytes.  Check that the pointer
// lies within the process address space.
int
argptr(int n, char **pp, int size)
{
  int i;
  struct proc *curproc = myproc();

  if(argint(n, &i) < 0)
    return -1;
  if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz)
    return -1;
  *pp = (char*)i;
  return 0;
}

// Fetch the nth word-sized system call argument as a string pointer.
// Check that the pointer is valid and the string is nul-terminated.
// (There is no shared writable memory, so the string can't change
// between this check and being used by the kernel.)
int
argstr(int n, char **pp)
{
  int addr;
  if(argint(n, &addr) < 0)
    return -1;
  return fetchstr(addr, pp);
}
```

These three system calls all work off of a command line arguments. The argint function will return an int and fetch the nth argument. The argptr function will return a pointer to the nth argument. Finally, the argstr will return a pointer to the string value of the nth argument.

wc.c

The wc.c file is the code for counting characters, words, and lines from whatever file is passed into it.

Conclusion

This lab gave an overview of some of the files that we have been working with in our previous assignments. This allowed us to have a greater understanding of what functions are used in the OS and where they are located. Some of the previous labs already worked with the information in this lab, so I had learned what they did by exploring the files before. The overview did help with putting some things together in the bigger picture of how the OS works. Personally, I think this lab should be done earlier, but we may not have enough understanding of the OS to

make sense of what is shown. I appreciate knowing what I will have to work on before working on it, so this lab will probably be helpful for some future assignments. I do have a better understanding now of what some functions are doing.