Lab Report 1

April 30th, 2017

Bradley Evans and Dharti Tarapara

CS153 Operating Systems

Part 1: Adding System Calls

Hello World

Implementing a hello world function and executing it in the xv6 shell.

The Hello World function is implemented by changing the following files.

- `defs.h`
    - A function prototype is added here.
    - `void hello(void);`
- `syscall.h`
    - Define a system call here.
    - `#define SYS_hello 22`
- `user.h`
    - Define a function prototype.
    - `int hello(void);`
- `proc.c`
    - Define the actual hello function.
    - `void hello(void) { cprintf("hello!\n"); }`
- `sysproc.c`
    - Define the system call here. This will simply call our `hello()` program in `proc.c`.
    - `int sys_hello(void) { hello(); return 0; }`
- `hello.c`
    - This is added to the root directory.

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

int main(int argc, char * argv[]) {
  hello();
  exit(-1);

  return 0;
}
```

Implemented in this way, the user can now execute a `hello` from the command line. This will trigger a system call, which will eventually call `hello()` from `proc.c`, displaying the hello world message.

Editing `wait()` and `exit()`

The `exit()` and `wait()` functions were modified to take integers.

Each instance of `exit()` in user programs was changed to `exit(0)` to reflect the change in type of exit from `void` to `int`. `exit` (and, in the same way, `wait`) now return a status.

To make use of this new parameter, `exit(int)` was changed to the following:

```
void
exit(int status)
{
  struct proc *p;
  int fd;

  // cprintf("exit status %d", proc->status);

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

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

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

  acquire(&ptable.lock);

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

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

```
  proc->status = status; // MOD - 4/18
  // Jump into the scheduler, never to return.
  proc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

Also, `wait()` was changed to the following:

```
int
wait(int * status)
{
  struct proc *p;
  int havekids, pid;

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

        // MOD - 4/29
        if (p->status != 0) {
          *status = p->status;
        } else *status = 0;

        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 || proc->killed){
      release(&ptable.lock);
      return -1;
    }

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

Part 2: Schedulers

Implementing Priority Scheduling

Prior to this lab, xv6 used a *round robin* style scheduler. We modified it to use a *priority* scheduler, changing the scheduler code to the following.

```
void
scheduler(void)
{
  struct proc *p;
  int priority = 0;   // hold priority value

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

    /* *** BEGIN MOD: 4/30 PRISCHED
    for(priority = 0; priority < maxpriority; priority++) {
      acquire(&ptable.lock);
      // Loop over process table looking for process to run.
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE)
          continue;
        // PRISCHED: Now check and see if that process matches our current
        // priority level.
        if (p->priority == priority) {
          // If it does, get it running.
          proc = p;
          switchuvm(p);
          p->state = RUNNING;
          swtch(&cpu->scheduler, proc->context);
          switchkvm();
        }
        // Process is done running for now.
        // It should have changed its p->state before coming back.
        proc = 0;
      }
      release(&ptable.lock);
    }*/

    // 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;
      }

      release(&ptable.lock);

      priority = getprocpriority();

      acquire(&ptable.lock);

      if (priority < p->priority) {
        p->priority = priority;
      }
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);
  }
}
```

In addition, a `priority` variable was added to to `struct proc` to give variables a means of storing a priority code.

The processes need a way to change priority. A mutator is required. In `proc.c` this is defined as `setpriority`:

```
int
setpriority(int num)
{
  if (num < 0) {
    num = 0;
  } else if (num > 63) {
    num = 63;
  }s priority

  proc->priority = num;

  return 0;
}
```

This is tied to a `setpriority` system call, added into the system in a similar way to the earlier `hello` call.

Also, a `getprocpriority` function will return the current seniormost priority level.

```
int
getprocpriority(void)
{
  struct proc *p;
  int priority = 65;

  // go through ptable
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    // look for runnable process
    if(p->state != RUNNABLE) {
          continue;
    }

    if(priority == 0) {
      priority = p->priority;
    } else {
      if (p->priority < priority) {
        priority = p->priority;
      }
    }
  }

  release(&ptable.lock);

  return priority;
}
```

Priority Donation and Inheritence

A problem with priority schedulers is deadlock (more often called "unbounded priority inversion"), where lower-priority processes do not yield resources to a higher-priority process. A way to mitigate this is via *priority donation and inheritence*. If a higher priority task (H) attempts to access resource in use by a lower priority task (L), the lower priority task "inherits" a higher priority so that it can complete execution of some critical, uninterruptable portion of its programming. This prevents L from being pre-empted by a medium priority task M, which might then block H from executing.

To make this happen, you could use the `setpriority` mutator whenever a task enters the queue below a lower priority task. `setpriority` would manually change the lower priority's `priority` value to be equal to that of the high-priority task. When the high-priority task is allowed to execute, it can use `setpriority` to return the low-priority task to its original priority level.