CS153 (Spring 2016) – xv6 Lab 1 Report

Jeet Thakkar – jthak002

Kevin Nguyen – knguy092

We used Songwei's four test cases: wait_one, wait_more, prio_test and prio_test2. We ran these two test cases multiple times to ensure completeness of our implementation of the code. To run the code, it is fairly standard. Compile and make the code by typing "make" into the command line while in the xv6 directory. After that, type "make qemu-nox" and the system will start. Typing "ls" will make it easier to see all the executable files and programs. Typing "wait_one" or "wait_more" will run Songwei's test cases and the output will be correct. The system should work as desired. There will only be one file of xv6.

Implementation of Part I

The implementation of the exit syscall was extremely easy as we had to change the function signature to include the status argument into the function declaration. Also, in order to save the exit status of the process that exited, we added an int exit status variable to put proc struct in proc.h. The only other change in the xv6 files was changing the exit() function calls to provide an exit status in all the user space programs, which as you can see have been changed to exit(1), for abnormal termination or errors(as in the case mkdir command when no arguments are provided to the syscall), or exit(0) to signify normal execution. The exit syscall code is given below:

```
//modified code shown in bold
// Exit the current process.  Does not return.
// An exited process remains in the zombie state
// until its parent calls wait() to find out it exited.
void
exit(int status)
{
  struct proc *p;
  int fd;
  proc->exitstatus=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;
```

```
    }
  }

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

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  wakeup1(proc->parent);
  if(proc->waitpid_parentindex!=0)
  {
      int i;
      for(i=0; i< proc->waitpid_parentindex;i++)
      {
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        {
          if(p->pid==proc->waitpid_parent[i])
            wakeup1(p);
        }
      }
  }
  // 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);
    }
  }

  // Jump into the scheduler, never to return.
  proc->state = ZOMBIE;

  //cprintf("ola");
  sched();
  panic("zombie exit");

}
```

Implementation of wait() and waitpid()

The wait syscall is already provided as the part of a code in the xv6 project. Our task was to analyze the exit status of the child process and return 0 or -1 through int *status, based on successful execution. The addition of these arguments led to the modification of sysproc.c to add these arguments to the interface of the syscalls. (USING argptr, and argint for dynamic allocation of heap). The code from sysproc.c is given below:

```
int sys_waitpid()
{
  int pid;
  int * status;
  int options;
  argint(0,&pid);
  argptr(0,(char**) &status, sizeof(int*));
  argint(0,&options);
  return waitpid(pid, status, options);
  return 0;//never reaches here
}
int
sys_exit() //ADD_USER:int ext_stat has been added to check effect
{
  int status;
  argint(0, &status);
  exit(status);
  return 0;  // not reached
}

int
sys_wait(void)
{
  int* status;
  argptr(0,(char **) &status, sizeof(int*));
  return wait(status);
}
```

The wait syscall has been modified as shown below. We setting the value of the pointer status to the value of p->exitstatus .

```
// Wait for a child process to exit and return its pid.
// Return -1 if this process has no children.
int
wait(int *status )
{
  struct proc *p;
  int havekids, pid;

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for zombie 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.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
```

```
        p->state = UNUSED;
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        if(status)
          *status=p->exitstatus;
        release(&ptable.lock);

        return pid;
      }
    }

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

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

As for waitpid, we modify the wait function to detect the pid of the process and compare it the the pid argument, instead of finding the child of the calling process. Once, it finds the child it checks if the child has completed execution. If not, the pid of the calling process is added to a special data structure within the child process called waitpid_parent[]. This keeps a record of all the waitpid-parents that need to be woken up when the child process exits. This waitpid_parent being woken up code is added to the exit syscall, right after the exiting process wakes up it's real parent process. The code for waitpid() is given below:

```
//WAITPID--PROGRAMMER ADDED SYSCALL--work needed
int
waitpid(int pid, int *status, int options)
{
  struct proc *p;
  int p_found;

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for zombie children.
    p_found = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if(p->pid != pid)
```

```
        continue;
      p_found = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->state = UNUSED;
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        if(status)
          *status=p->exitstatus;
        release(&ptable.lock);
        return pid;
      }
      else
      {
        p->waitpid_parent[p->waitpid_parentindex]=proc->pid;
        p->waitpid_parentindex++;
        break;
      }
    }

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

    // Wait for children to exit.  (See wakeup1 call in proc_exit.)

    sleep(proc, &ptable.lock);  //DOC: wait-sleep
  }
}
```

Files modified to add new Syscall – proc.h, proc.c, syscall.h, syscall.c, user.h, usys.S, sysproc.c, defs.h, sysfunc.h

PRIORITY SCHEDULING

The code that we modified included the addition of a system call change_priority(int priority) in order to run the test provided by the TA. The scheduler code is the traversal of all the processes from the ptable and then running them based on their priority. Also, the code makes sure that once the priority of a process is changed it is executed first. At the moment our code just creates processes with the lowest

priority i.e. 63. Also, multiple processes with the same priority are executed in the round robin manner, where the process that was created first is executed other processes.

CODE:

```
//setpriority syscall--this syscall sets the priority for a process to
a
//different value
void change_priority(int priority)
{
  struct proc * p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid==proc->pid)
    {
      if(priority<63 && priority>0)
      {
        p->priority=priority;
        p->numtickets=0;
        break;
      }
      else
      {
        p->priority=63;
        p->numtickets=0;
        break;
      }
    }
    else
      continue;
    }
    release(&ptable.lock);
    //enable interrupts

    exit(0);  //call scheduler as soon as priority changes
}




void
scheduler(void)
{
  struct proc *p;

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

```
    int i;

    for(i=0;i<=63;i++)  //for priority levels from 1 to 63
    {
      // Loop over process table looking for process to run.
      acquire(&ptable.lock);    //important to place here..should not
hamper the use of ptable by other processes
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
      {
        if(p->state != RUNNABLE)    //checks if process is runnable--
standard issue
          continue;
        if(p->priority==i)  //finds the priority of level of i--will
move from 0-63
        {
          // 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, proc->context);
          switchkvm();
          // Process is done running for now.
          // It should have changed its p->state before coming back.
          proc = 0;
        }
      }
      release(&ptable.lock);
    }
  }
}
```

Test cases

wait_one:

```
[[jthak002@sledge xv6-master]$ make qemu-nox
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB) copied, 0.0382069 s, 134 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes (512 B) copied, 5.9917e-05 s, 8.5 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
275+1 records in
275+1 records out
141208 bytes (141 kB) copied, 0.000641576 s, 220 MB/s
qemu -nographic -hdb fs.img xv6.img -smp 2 -m 512
Could not open option rom 'sgabios.bin': No such file or directory
xv6...
cpu1: starting
cpu0: starting
init: starting sh
[$ wait_one
pid = 5
pid = 6
pid = 7
pid = 9
pid = 10
pid = 8
pid = 11
pid = 12
pid = 13
pid 13 waiting for 8
success clean 8, status is 0
pid = 14
pid = 15
pid = 16
pid = 17
pid = 18
pid = 19
kill 4
kill 5
kill 6
kill 7
kill 15
kill 9
kill 10
kill 11
kill 12
kill 13
kill 14
kill 16
kill 17
kill 18
kill 19
kill -1
$
```

wait_more:

```
$ wait_more
pid = 23
pid = 24
pid = 25
pid = 27
pid = 28
pid = 29
pid = 30
pid = 26
pid = 31
pid 31 waiting for 26
success clean 26
pid = 32
pid = 33
pid 33 waiting for 26
no more waiting for 26
pid = 34
pid = 35
pid = 36
kill pid = 37
22 process
kill 23 process
kill 24 process
kill 25 process
kill 33 process
kill 27 process
kill 28 process
kill 29 process
kill 30 process
kill 31 process
kill 32 process
kill 34 process
kill 35 process
kill 36 process
kill 37 process
kill -1 process
$
```

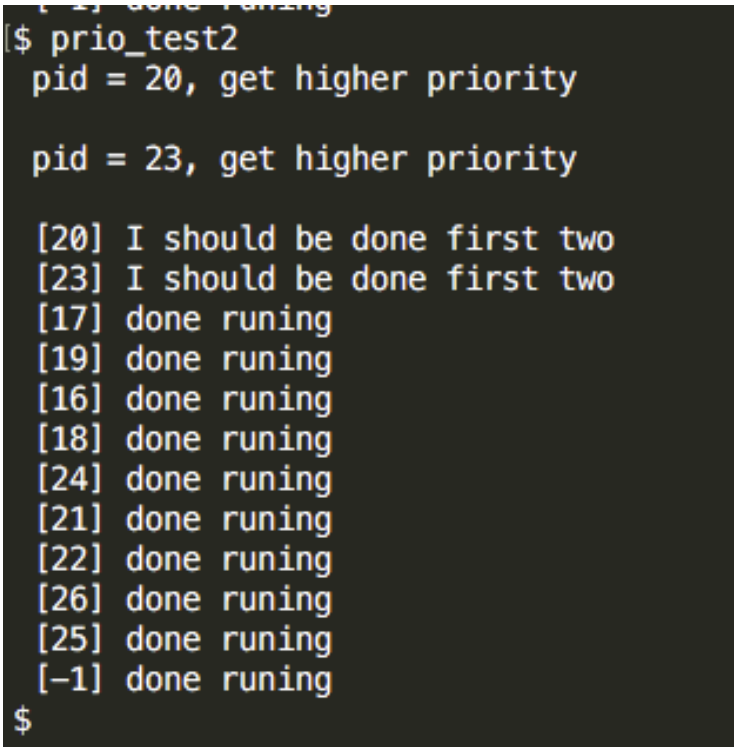
prio_test:

```
-bash: exity: command not found
[jthak002@sledge xv6_part2]$ exit
logout
Connection to sledge.cs.ucr.edu closed.
Jeets-MacBook-Air:Desktop jeet$ ssh jthak002@sledge.cs.ucr.edu
jthak002@sledge.cs.ucr.edu's password:
Last login: Thu May  5 23:20:53 2016 from 104.241.36.27
-bash: warning: setlocale: LC_CTYPE: cannot change locale (UTF-8): No such file or directory
[jthak002@sledge ~]$ clear

[jthak002@sledge ~]$ cd xv6_part2
[jthak002@sledge xv6_part2]$ make qemu-nox
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB) copied, 0.0382408 s, 134 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.000102341 s, 5.0 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
277+1 records in
277+1 records out
141932 bytes (142 kB) copied, 0.000694979 s, 204 MB/s
qemu -nographic -hdb fs.img xv6.img -smp 2 -m 512
Could not open option rom 'sgabios.bin': No such file or directory
xv6...
cpu1: starting
cpu0: starting
init: starting sh
$ prio_test
 pid = 11, get higher priority

 [11] I should be done first
 [4] done runing
 [6] done runing
 [10] done runing
 [8] done runing
 [13] done runing
 [5] done runing
 [7] done runing
 [9] done runing
 [12] done runing
 [14] done runing
 [-1] done runing
$
```

prio_test2:



adding test files to xv6 console:

create two files wait_one.c and wait_more.c. in order to compile them add these entries in the makefile in the “extras” and “UPROGS” section.