

## LAB ASSIGNMENT: xv6 SYSTEM CALL

ARPIT AGARWAL

IIT2019139

### BOOTING xv6

```
Machine View

The code in the files that constitute xv6 is
Copyright 2006-2018 Frans Kaashoek, Robert Morris, and Russ Cox.

ERROR REPORTS

We don't process error reports (see note on top of this file).

BUILDING AND RUNNING Xv6

To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run
"make". On non-x86 or non-ELF machines (like OS X, even on x86), you
will need to install a cross-compiler gcc suite capable of producing
x86 ELF binaries (see https://pdos.csail.mit.edu/6.828/).
Then run "make TOOLPREFIX=i386-jos-elf-". Now install the QEMU PC
simulator and run "make qemu". $ grep run README
To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run
Then run "make TOOLPREFIX=i386-jos-elf-". Now install the QEMU PC
$ cat README | grep run | wc
2 22 130
$ echo MY NEW FILE > newfile
$ cat newfile
MY NEW FILE
$ _
```

### **Part I : System Call Tracing**

The following is the code for syscall.c where changes are to be done.

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
#include "syscall.h"
```

```

// User code makes a system call with INT T_SYSCALL.
// System call number in %eax.
// Arguments on the stack, from the user call to the C
// library system call function. The saved user %esp points
// to a saved program counter, and then the first argument.

// Fetch the int at addr from the current process.
int
fetchint(uint addr, int *ip)
{
    struct proc *curproc = myproc();

    if(addr >= curproc->sz || addr+4 > curproc->sz)
        return -1;
    *ip = *(int*)(addr);
    return 0;
}

// Fetch the nul-terminated string at addr from the current process.
// Doesn't actually copy the string - just sets *pp to point at it.
// Returns length of string, not including nul.
int
fetchstr(uint addr, char **pp)
{
    char *s, *ep;
    struct proc *curproc = myproc();

    if(addr >= curproc->sz)
        return -1;
    *pp = (char*)addr;
    ep = (char*)curproc->sz;
    for(s = *pp; s < ep; s++){
        if(*s == 0)
            return s - *pp;
    }
    return -1;
}

// 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);
}

extern int sys_chdir(void);
extern int sys_close(void);
extern int sys_dup(void);
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
extern int sys_mkdir(void);
extern int sys_mknod(void);
extern int sys_open(void);
extern int sys_pipe(void);
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);

```

```

extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_cps(void);

static int (*syscalls[])(void) = {
[SYS_fork]      sys_fork,
[SYS_exit]      sys_exit,
[SYS_wait]      sys_wait,
[SYS_pipe]      sys_pipe,
[SYS_read]      sys_read,
[SYS_kill]      sys_kill,
[SYS_exec]      sys_exec,
[SYS_fstat]     sys_fstat,
[SYS_chdir]     sys_chdir,
[SYS_dup]       sys_dup,
[SYS_getpid]    sys_getpid,
[SYS_sbrk]      sys_sbrk,
[SYS_sleep]     sys_sleep,
[SYS_uptime]    sys_uptime,
[SYS_open]      sys_open,
[SYS_write]     sys_write,
[SYS_mknod]     sys_mknod,
[SYS_unlink]    sys_unlink,
[SYS_link]      sys_link,
[SYS_mkdir]     sys_mkdir,
[SYS_close]     sys_close,
[SYS_cps]       sys_cps,
};

// Declaring array of char pointer
static char* syscallnames[] = {
[SYS_fork]      "fork",
[SYS_exit]      "exit",
[SYS_wait]      "wait",
[SYS_pipe]      "pipe",
[SYS_read]      "read",
[SYS_kill]      "kill",
[SYS_exec]      "exec",
[SYS_fstat]     "fstat",
[SYS_chdir]     "chdir",
[SYS_dup]       "dup",
[SYS_getpid]    "getpid",
[SYS_sbrk]      "sbrk",
[SYS_sleep]     "sleep",
[SYS_uptime]    "uptime",
[SYS_open]      "open",
[SYS_write]     "write",

```

```

[SYS_mknod]    "mknod",
[SYS_unlink]   "unlink",
[SYS_link]     "link",
[SYS_mkdir]    "mkdir",
[SYS_close]    "close",
[SYS_cps]      "cps"
};

void
syscall(void)
{
    int num;
    struct proc *curproc = myproc();

    num = curproc->tf->eax;
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {

        curproc->tf->eax = syscalls[num]();
        cprintf("%s->%d\n", syscallnames[num], curproc->tf->eax);
    } else {
        cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
        curproc->tf->eax = -1;
    }
}

```

**/\* The line in bold is to be added in syscall.c (void syscall(void) function) \*/**

***\*\* Screenshots of output are on NEXT page\*\****

---

```
arpit@arpit: ~/xv6-public
417+1 records out
213676 bytes (214 kB, 209 KiB) copied, 0.00206623 s, 103 MB/s
arpit@arpit:~/xv6-public$ make qemu
qemu-system-x86_64 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.i
mg,index=0,media=disk,format=raw -smp 2 -m 512
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
exec->0
open->0
dup->1
dup->2
iwrite->1
nwrite->1
iwrite->1
twrite->1
:write->1
write->1
swrite->1
twrite->1
awrite->1
rwrite->1
twrite->1
iwrite->1
nwrite->1
gwrite->1
write->1
swrite->1
hwrite->1

write->1
fork->2
exec->0
open->3
close->0
$write->1
write->1
█
```

```
QEMU

Machine View
iwrite->1
twrite->1
:write->1
write->1
swrite->1
twrite->1
awrite->1
rwrite->1
twrite->1
iwrite->1
nwrite->1
gwrite->1
write->1
swrite->1
hwrite->1

write->1
fork->2
exec->0
open->3
close->0
$write->1
write->1
```

## Part II : ps System Call

Modifications to be done in the files written below. We are making a new syscall call with the name of **cps** and it is to be added in these files.

Suppose the syscall that we make is called “**cps**” which prints out **current process states (PID)** in the system

```
// syscall.h
// defs.h
// user.h
// sysproc.c
```

Now we have to add this function call to sysproc.c

```
int
sys_cps (void)
{
    return cps();
}

// usys.S
// syscall.c
// proc.c -> the function below is to be added...

// current process status
int
cps() {
    struct proc *p;
    // enable interrupts on this processor.
    sti();

    // loop over process table looking for process with pid.
    acquire(&ptable.lock);
    cprintf("Name \t PID \t State \t \n");
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state == SLEEPING)
            cprintf("%s \t %d \t SLEEPING \t \n ", p->name, p->pid);
        else if (p->state == RUNNING)
            cprintf("%s \t %d \t RUNNING \t \n ", p->name, p->pid);
        else if (p->state == RUNNABLE)
            cprintf("%s \t %d \t RUNNABLE \t \n", p->name, p->pid);
    }

    release(&ptable.lock);

    return 22;
}
```

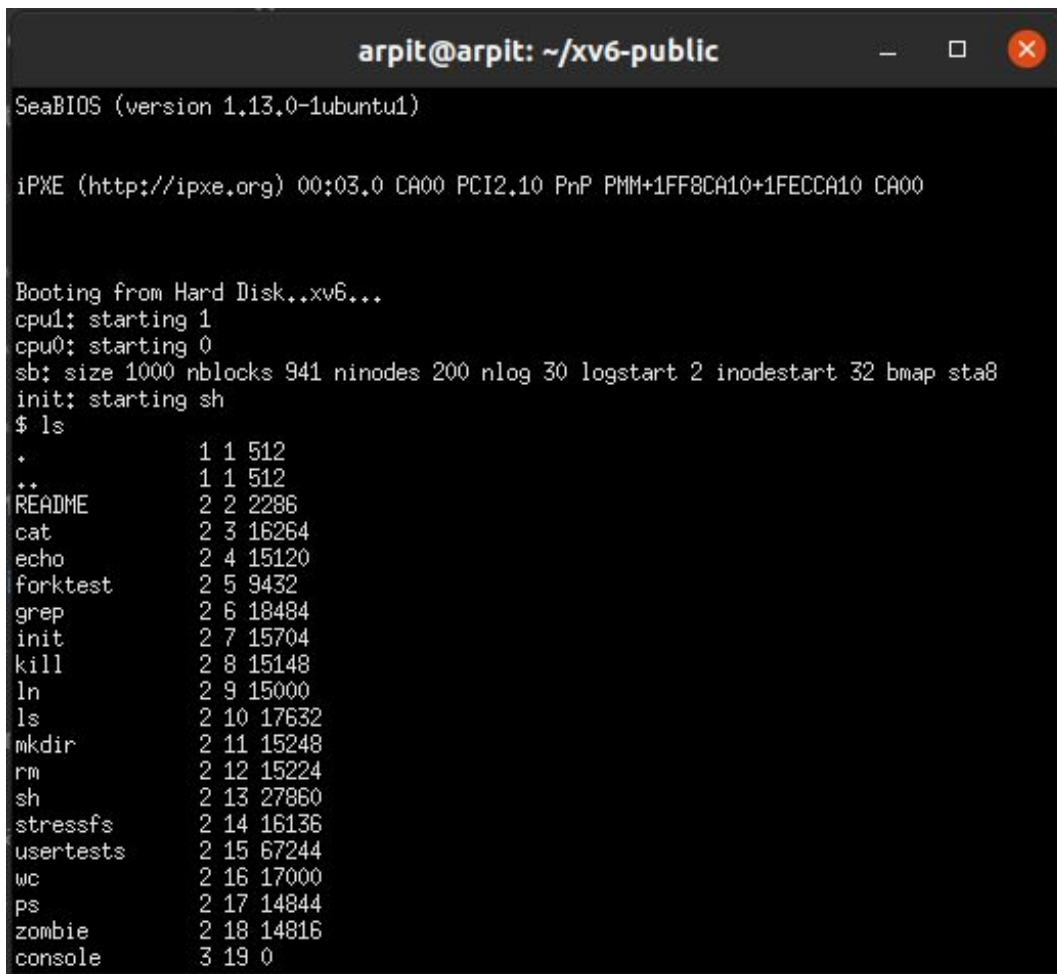
**// We have to make a new file named "ps.c"**

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

int
main(int argc, char *argv[])
{
    cps();

    exit();
}
```

**// Finally modify MAKEFILE to add ps there...**



```
arpit@arpit: ~/xv6-public
SeaBIOS (version 1.13.0-1ubuntu1)

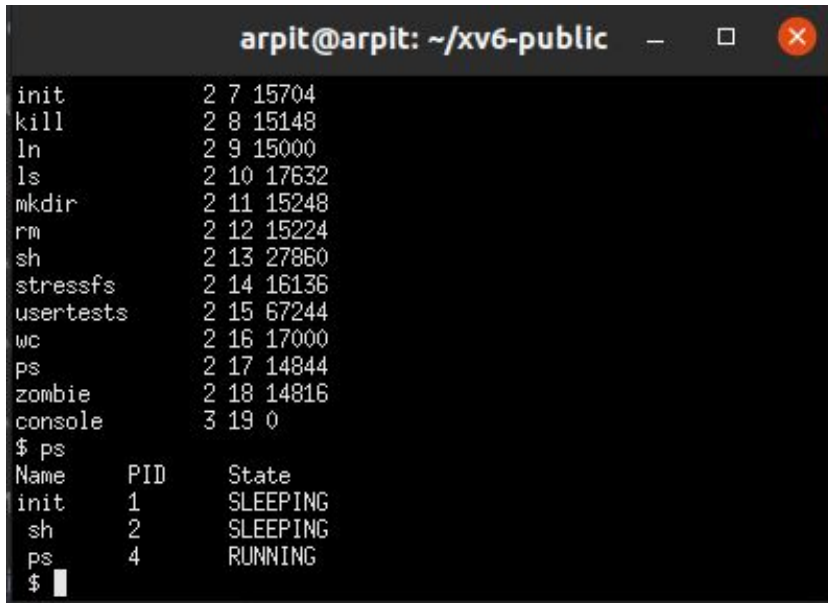
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00

Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta8
init: starting sh
$ ls
*          1 1 512
..         1 1 512
README    2 2 2286
cat       2 3 16264
echo      2 4 15120
forktest  2 5 9432
grep      2 6 18484
init      2 7 15704
kill      2 8 15148
ln        2 9 15000
ls        2 10 17632
mkdir     2 11 15248
rm        2 12 15224
sh        2 13 27860
stressfs  2 14 16136
usertests 2 15 67244
wc        2 16 17000
ps        2 17 14844
zombie    2 18 14816
console   3 19 0
```

**\*\*In the output we can see that ps syscall has been added\*\***



***\*\* Now executing ps we gather the number of processes running  
their current state along with their PID \*\****



A terminal window titled 'arpit@arpit: ~/xv6-public' showing a list of processes and the output of the 'ps' command. The process list includes init, kill, ln, ls, mkdir, rm, sh, stressfs, usertests, wc, ps, zombie, and console. The 'ps' command output shows the Name, PID, and State for the init, sh, and ps processes.

```
arpit@arpit: ~/xv6-public
init      2 7 15704
kill      2 8 15148
ln         2 9 15000
ls         2 10 17632
mkdir     2 11 15248
rm         2 12 15224
sh         2 13 27860
stressfs  2 14 16136
usertests 2 15 67244
wc         2 16 17000
ps         2 17 14844
zombie    2 18 14816
console   3 19 0
$ ps
Name      PID    State
init      1      SLEEPING
sh         2      SLEEPING
ps         4      RUNNING
$
```

---

## **ALITER:**

**getprocinfo** is defined as follows

```
static char *states[] = {
[UNUSED] "unused",
[EMBRYO] "embryo",
[SLEEPING] "sleep ",
[RUNNABLE] "runnable",
[RUNNING] "run ",
[ZOMBIE] "zombie"
};

int i;
struct proc *p;
char *state;
uint pc[10];
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

    if(p->state == UNUSED)
        continue;

    if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
        state = states[p->state];
}
```

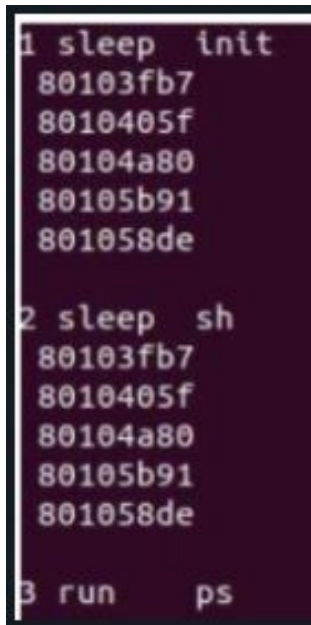
```
else
state = "N/A";

cprintf("%d %s %s \n", p->pid, state, p->name);

if(p->state == SLEEPING){
getcallerpcs((uint*)p->context->ebp+2, pc);

for(i=0; i<10 && pc[i] != 0; i++)
    cprintf(" %p \n", pc[i]);
}

cprintf("\n");
```

A terminal window with a dark purple background and light green text. It shows the output of a program. The first section is labeled '1 sleep init' and lists five memory addresses: 80103fb7, 8010405f, 80104a80, 80105b91, and 801058de. The second section is labeled '2 sleep sh' and lists the same five memory addresses. The third section is labeled '3 run ps' and is partially visible.

```
1 sleep init
80103fb7
8010405f
80104a80
80105b91
801058de

2 sleep sh
80103fb7
8010405f
80104a80
80105b91
801058de

3 run ps
```

---

**\*\* THE END \*\***

---