COMP8051 – Operating System Engineering

Completion Date: 12th March 2021

Value: 40 marks

On completion please zip up your files and upload to Canvas.

Install qemu on your Ubuntu vm

sudo apt-get update && sudo apt-get install git nasm build-essential qemu gdb

Download xv6 using git.

Open a terminal window

git clone git://github.com/mit-pdos/xv6-public.git

cd xv6-public

run **make** to build xv6

and

make qemu

to run xv6

Execute some shell commands to get familiar with the user part of the OS. The shell commands are separate programs e.g. ls.c, cat.c.

To get a feel for how programs look in xv6, and how various APIs should be called, you can look at the source code for other utilities: echo.c, cat.c, wc.c, ls.c.

Hints:

In places where something asks for a file descriptor, you can use either an actual file descriptor (i.e., the return value of the open function), or one of the standard I/O descriptors: 0 is "standard input", 1 is "standard output", and 2 is "standard error". Writing to either 1 or 2 will result in something being printed to the screen.

The standard header files used by xv6 programs are "types.h" (to define some standard data types) and "user.h" (to declare some common functions). You can look at these files to see what code they contain and what functions they define.

Q1 hello world in xv6

Write a program for xv6 that, prints "Hello world" to the xv6 console. This can be broken up into a few steps:

1. Create the file hello.c in the xv6 directory

2. Edit the file Makefile, find the section UPROGS (which contains a list of programs to be built), and add a line to tell it to build your hello.c code. When you're done that portion of the Makefile should look like:

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _hello\
```

3. Run **make** to build xv6, including your new program

4. Run **make qemu** to launch xv6, and then type hello in the QEMU window. You should see "Hello world" being printed out.

(1 mark)

Question 2 User programs

a) Write a program that prints the first 10 lines of its input for the xv6 operating system. If a filename is provided on the command line (i.e., ***head*** FILE) then ***head*** should open it, read and print the first 10 lines, and then close it. If no filename is provided, ***head*** should read from standard input.

See how the program cat.c works e.g

cat README

grep the README | head

should show ten lines each line has the word the in it.

Hints:

Many aspects of this are similar to the wc program: both can read from standard input if no arguments are passed or read from a file if one is given on the command line. Reading its code will help you if you get stuck.

b) What is a user program for the xv6 operating system? Explain how a user program links to library functions such as printf and how does it access the operating system?

c) Explain how the xv6 shell works – refer to the xv6 source code for your answer.

d) Explain how xv6 implements the ***ls*** program – refer to the xv6 source code for your answer.

(10 marks)

Question 3 - cp and mv commands

Add basic versions of the commands cp, mv, to xv6. The cp and mv commands

are to work on files only (no dirs). System calls to be used are reported in brackets. Please see user.h (on xv6) for a complete list of syscalls and library functions available.

These commands are available on Linux.

* cp src dst (open, read, write)

* mv oldname newname (link, unlink)

The build procedure can be broken up into a few steps:

1. Create the file cp.c in the xv6 directory

2. Edit the file Makefile, find the section UPROGS (which contains a list of programs to be built), and add a line to tell it to build your cp.c code.
3. Run make to build xv6, including your new program

4. Run make qemu to launch xv6, and then type execute cp in the QEMU window.

(4 marks)

Question 4 – interrupts and system calls

a) Read chapter 3 of the XV6 book and the xv6 source and describe how xv6 dispatches interrupts and system calls – refer to the xv6 source code for your answer.

b) Explain how the keyboard driver buffers keystrokes for the xv6 operating system.

(12 marks)

Question 5 - Add a new system call called trace.

The trace syntax is

int trace(int)

When called with a non-zero parameter, e.g., trace(1), system call tracing is turned on for that process. Each system call from that process will be printed to the console in a user-friendly format showing:

- the process ID
- the process name
- the system call number
- the system call name

Any other processes will not have their system calls printed unless they also call trace(1).

Calling trace(0) turns tracing off for that process. System calls will no longer be printed to the console

In all cases, the trace system call also returns the total number of system calls that the process has made since it started. Hence, you can write code such as:
printf("total system calls so far = %d\n", trace(0));

How to add a new system call to XV6

You need to touch several files to add a system call in xv6. Look at the implementation of existing system calls for guidance on how to add a new one. The files that you need to edit to add a new system call include:
user.h
This contains the user-side function prototypes of system calls as well as utility library functions (stat, strcpy, printf, etc.).

syscall.h
This file ontains symbolic definitions of system call numbers. You need to define a unique number for your system call. Be sure that the numbers are consecutive. That is, there are no missing number in the sequence. These numbers are indices into a table of pointers defined in syscall.c (see next item).

syscall.c

This file contains entry code for system call processing. The syscall(void) function is the entry function for all system calls. Each system call is identified by a unique integer, which is placed in the processor's eax register. The syscall function checks the integer to ensure that it is in the appropriate range and then calls the corresponding function that implements that call by making an indirect funciton call to a function in the syscalls[] table. You need to ensure that the kernel function that implements your system call is in the proper sequence in the syscalls array.

usys.S

This file contains macros for the assembler code for each system call. This is user code (it will be part of a user-level program) that is used to make a system call. The macro simply places the system call number into the eax register and then invokes the system call. You need to add a macro entry for your system call here.

sysproc.c

This is a collection of process-related system calls. The functions in this file are called from syscall. You can add your new function to this file.
Per-process state is stored in a proc structure: struct proc in proc.h. You'll need to extend that structure to keep track of the process related metrics. You'll also need to find where the proc structure is allocated so that you can ensure that the elements are initialized appropriately.

When you implement your trace call, you'll need to retrieve the incoming parameter. The file sysproc.c defines a few helper functions to do this. The functions argint, argptr, and argstr retrieve the n th system call argument, as either an integer, pointer, or a string. argint uses the esp register to locate the argument: esp points at the return address for the system call stub.

Implementation steps

1. Write a test program.
Add the test program to the Makefile so that it will be compiled and built whenever you run make.

In the Makefile, add your program (e.g., try.c) to the list of user commands in the UPROGS= section. That should be all you need to do to that file.

Beware that programs don’t have access to the typical stdio library that you expect to find on most systems. You’ll have many of the functions you expect but some of the behavior might be different. For example, printf accepts an initial parameter that is the output stream: 1 represents the standard output and 2 represents the standard error stream. There is no FILE* type and no fopen, fclose, fgets, etc. calls. Look through usertests.c for examples on how all of the system calls provided with xv6 are used.

2. Add system call tracing to the kernel.

Print a message identifying everysystem call that is requested by any process as well as the process ID and process name. You do not need to print the arguments to the system calls. When you run any program, including the shell, you will see output similar to this:

```
...
pid: 2 [sh] syscall(5=read)
pid: 2 [sh] syscall(5=read)
pid: 2 [sh] syscall(1=fork)
pid: 2 [sh] syscall(3=wait)
pid: 3 [sh] syscall(12=sbrk)
pid: 3 [sh] syscall(7=exec)
pid: 3 [try] syscall(20=mkdir)
pid: 3 [try] syscall(15=open)
pid: 3 [try] syscall(16=write)
pid: 3 [try] syscall(21=close)
pid: 3 [try] syscall(2=exit)
pid: 2 [sh] syscall(16=write)
pid: 2 [sh] syscall(16=write)
...
```

Use the cprintf function in the kernel, which prints using direct access to the vga controller. It works just like the normal Linux printf function. For example:

```
cprintf("hello, I'm number %d\n", num);
```

3. Restrict this output to a single process.

Create a new system call called trace(int). This turns console-based system call logging on and off for only the calling process.
Extend the proc structure (the process control block) in proc.h to keep track of whether tracing for the process is on or off. Be sure that the elements are cleared (initialized) whenever a new process is created.

In implementing your system call, you'll need to access the single parameter passed by trace. Use the helper functions defined in syscall.c (argint, argptr, and argstr). Take a look at how other system calls are implemented in xv6. For example, getpid is a simple system call that takes no arguments and returns the current process' process ID; kill and sleep are examples of system calls that take a single integer parameter.

4. Add system call counting.

Be sure to count calls on a per-process basis. You

will need to keep track of this in the process control block, the proc structure.

(7 marks)

Question 6 – ps command in XV6

The process table and struct proc in proc.h are used to maintain information on the current processes that are running in the XV6 kernel. Since ps is a user space program, it cannot access the process table in the kernel. So we'll add a new system call. The ps command should print:

- process id
- parent process id
- state
- size
- name

The system call you need to add to xv6 has the following interface:

int getprocs(int max, struct uproc table[]);

struct uproc is defined as (add it to a new file uproc.h):

```
struct uproc {
int pid;
int ppid;
int state;
uint sz;
char name[16];
};
```

Your ps program calls getprocs with an array of struct proc objects and sets max to the size of that array (measured in struct uproc objects). Your kernel code copies up to max entries into your array, starting at the first slot of the array and filling it consecutively. The kernel returns the actual number of processes in existence at that point in time, or -1 if there was an error.

(6 marks)