CSCI350 Project 1

Due: February 4th by 11:59pm

Objectives

In this first xv6 assignment you will be modifying and adding code to different files that implement the kernel. Along the way you'll learn how to build the kernel and test/debug your work, and will hopefully acquire a working knowledge of some essential kernel tasks and modules.

You won't be adding much source code; rather, the goal is to get you comfortable with the workflow so that future assignments requiring more extensive changes to the codebase won't be too intimidating!

Your assignment is to download and build the xv6 operating system and boot it on an x86 emulator QEMU. You will add a new system call called *trace*, which turns console-based system call tracing for the calling process on and off. The system call also reports the total number of system calls called by the process since it started. You will also create a date system call which will be called via date.c. This will print out the date and time in a recognizable format. Finally, you will answer a few questions about this assignment and the xv6 operating system and submit.

Summary:

- Install QEMU and other setup
- Create trace system call
- Create date system call
- FRQ and submission

We will use Ubuntu Linux in this class.

C language and gdb

All projects in this class will be done in the C programming language.

Here is a short, free, and incomplete overview of the C programming http://pages.cs.wisc.edu/~remzi/OSTEP/lab-tutorial.pdf.

There are many online tutorials for C, such as learn-c.

If you are interested in mastering the Unix programming environment: Advanced Programming in the UNIX Environment by W. Richard Stevens.

It will be useful to figure out how to use the debugger, gdb.

Here is a link to a short tutorial, there are others on the web too.

What is xv6?

xv6 is a slimmed-down, simplified operating system that is a loosely-based reimplementation of the UNIX sixth edition (v6) kernel, written in standard C and for the Intel x86 architecture.

It was designed at MIT as a small, easy-to-understand operating system that is suitable for teaching. For information about xv6, take a look at its home page at MIT. A detailed description of the structure of xv6, including its system calls, file system, and memory management can be found in:

- *Russ Cox, Frans Kaashoek, Robert Morris*, xv6: a simple, Unix-like teaching operating system. Sept 3, 2014.

What is QEMU?

QEMU is an open source machine emulator and virtualizer. It allows you to run software for one machine on a different machine architecture. It also makes it easy to run and debug operating systems since the complete operating system just runs as another process on your system.

Compiler Toolchain

You'll use two sets of tools in this class: an x86 emulator, QEMU, for running your kernel; and a compiler toolchain, including assembler, linker, C compiler, and debugger, for compiling and testing your kernel.

A "compiler toolchain" is the set of programs, including a C compiler, assemblers, and linkers, that turn code into executable binaries. You'll need a compiler toolchain that generates code for 32-bit Intel architectures ("x86" architectures) in the ELF binary format.

Test Your Compiler Toolchain

Modern Linux and BSD UNIX distributions already provide a toolchain suitable for 6.828. To test your distribution, try the following commands:

```
% objdump -i
```

The second line should say `elf32-i386`.

```
% gcc -m32 -print-libgcc-file-name
```

The command should print something like `/usr/lib/gcc/i486-linux-gnu/`*`version`*`/libgcc.a` or `/usr/lib/gcc/x86_64-linux-gnu/`*`version`*`/32/libgcc.a`

If both these commands succeed, you're all set, and don't need to compile your own toolchain.

If the gcc command fails, you may need to install a development environment. On Ubuntu Linux, try this:

```
% sudo apt-get install -y build-essential gdb
```

On 64-bit machines, you may need to install a 32-bit support library. The symptom is that linking fails with error messages like "`__udivdi3` not found" and "`__muldi3` not found". On Ubuntu Linux, try this to fix the problem:

```
% sudo apt-get install gcc-multilib
```

Using a Virtual Machine

The easiest way is to install a modern Linux distribution on your computer. With platform virtualization, Linux can cohabitate with your normal computing environment. Installing a Linux virtual machine is a two-step process. First, you download the virtualization platform.

- VirtualBox (free for Mac, Linux, Windows) — Download page
- VMware Player (free for Linux and Windows, registration required)
- VMware Fusion (Downloadable from IS&T for free).

VirtualBox is a little slower and less flexible, but free!

Once the virtualization platform is installed, download a boot disk image for the Linux distribution of your choice.

- Ubuntu Desktop is what we use.

This will download a file named something like `ubuntu-10.04.1-desktop-i386.iso`. Start up your virtualization platform and create a new (32-bit) virtual machine. Use the downloaded Ubuntu image as a boot disk; the procedure differs among VMs but is pretty simple. Type `objdump -i`, as above, to verify that your toolchain is now set up. You will do your work inside the VM.

If you experience dependency problems with the 64 bit virtual machines, 32 bit VM was created that you can download from Piazza (csci350_32bitvm.ova in Resources).

Installation Instructions

Step 1 – Install qemu:

```
$ sudo apt-get install qemu
```

If you have 64 bit OS there is a chance Makefile will not be able to find qemu. In that case you should edit the Makefile at line 54 and add the following code:

```
QEMU = qemu-system-x86_64
```

Step 2 – Install xv6

Create a directory, and clone xv6 to that directory:

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

Step 3 – Compile xv6

```
$ make
```

Step 4 – Compile and run the emulator qemu:

```
$ make qemu
```

The emulator will start but it will echo on the terminal too. Use `make qemu-nox`. Doing so avoids the use of X windows and is generally fast and easy. However, quitting is not so easy; to quit, you have to know the shortcuts provided by the machine emulator, qemu. Type `control-a` followed by `x` to exit the emulation. There are a few other commands like this available; to see them, type `control-a` followed by an `h`.

```
$ make qemu-nox
```

Also, for gdb enabled execution run:

```
$ make qemu-nox-gdb
```

To enter the qemu console at any time, press ctrl+a, then c. If you want to exit, simply type "quit" to exit the emulator.

Running xv6 under QEMU

Runxv6 on top of QEMU by calling `make qemu`

QEMU's virtual BIOS will load xv6's boot loader from a virtual hard drive image contained in the file `xv6.img`, and the boot loader will in turn load and run the xv6 kernel. After everything is loaded, you should get a '`$`' prompt in the xv6 display window and be able to enter commands into the rudimentary but functional xv6 shell. For example, try:

```
$ ls
.                 1 1 512
..                1 1 512
README            2 2 1844
cat               2 3 12129
...
$ echo Hello!
Hello!
$ cat README
```

```
xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix
Version 6 (v6).  xv6 loosely follows the structure and style of v6,
...
```

Here is a Youtube video that demonstrates xv6 installation procedure.

Assignment Part 1 of 2

Chapter 3 of the xv6 book contains details on traps and system calls (though most of the low level details won't be necessary for you to complete this exercise).

Your assignment is to add a new system call called *trace*. Its 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 system call?

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 contains 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 function 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.

When you implement your *trace* call, you'll need to retrieve the incoming parameter. The file syscall.c defines a few helper functions to do this. The functions *argint*, *argptr*, and *argstr* retrieve the nth 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.

proc.h

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 metrics required by this assignment. You'll also need to find where the proc structure is allocated so that you can ensure that the elements are initialized appropriately.

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 takes a single integer parameter.

Testing your code for trace system call

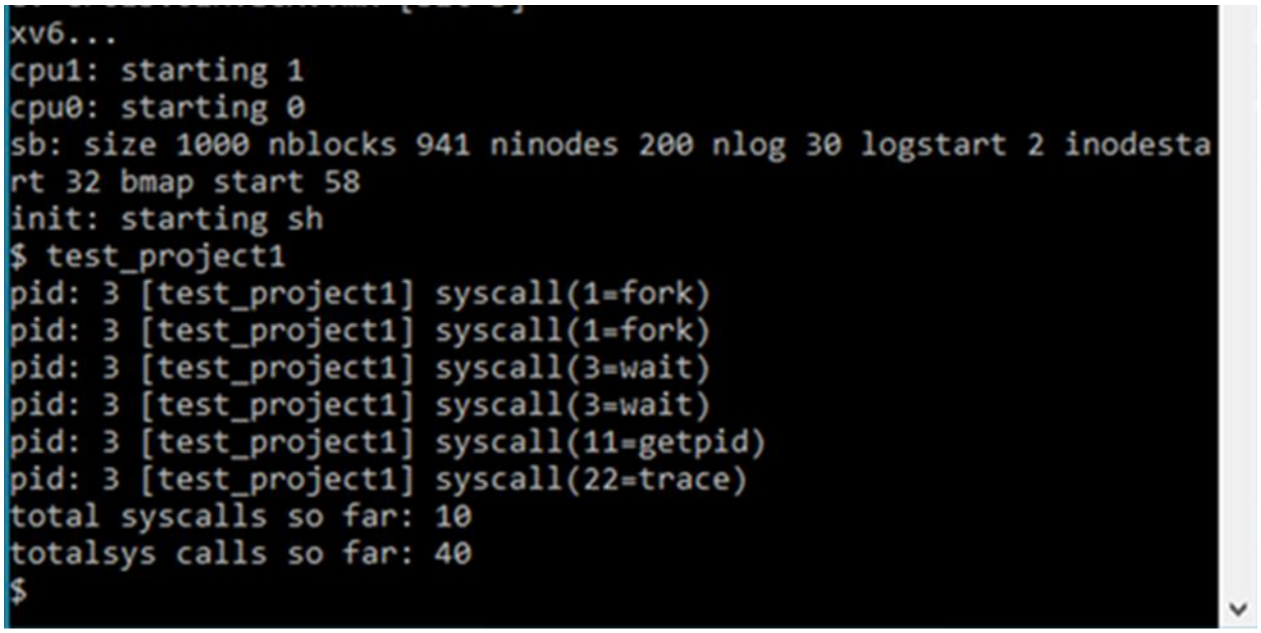
Steps:

1. Download test_project1.c program from Resources on Piazza.
2. Place it in your xv6 directory.

3. Modify your Makefile file to include a new user level program:
   UPROGS=\
   ….
   _test_project1\

   EXTRA=\
   …
   test_project1.c\

   4. Compile and run the program
   5. Expected output:



```
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodesta
rt 32 bmap start 58
init: starting sh
$ test_project1
pid: 3 [test_project1] syscall(1=fork)
pid: 3 [test_project1] syscall(1=fork)
pid: 3 [test_project1] syscall(3=wait)
pid: 3 [test_project1] syscall(3=wait)
pid: 3 [test_project1] syscall(11=getpid)
pid: 3 [test_project1] syscall(22=trace)
total syscalls so far: 10
totalsys calls so far: 40
$
```

Assignment Part 2 of 2: Date system call

Your second task is to add a new system call to xv6. The main point of the exercise is for you to see some of the different pieces of the system call machinery. Your new system call will get the current UTC time and return it to the user program. You may want to use the helper function, cmostime() (defined in lapic.c), to read the real time clock. date.h contains the definition of the struct rtcdate struct, which you will provide as an argument to cmostime() as a pointer.

You should create a user-level program, date.c that calls your new date system call; here's some source code you should put in date.c:

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

```
int
main(int argc, char *argv[])
{
  struct rtcdate r;

  if (date(&r)) {
    printf(2, "date failed\n");
    exit();
  }
  // your code to print the date-time
  exit();
}
```

In order to make your new date program available to run from the xv6 shell, add _date to the UPROGS definition in Makefile.

Your strategy for making a date system call should be to clone all of the pieces of code that are specific to some existing system call, for example the "uptime" system call. You should grep for uptime in all the source files, using grep -n uptime *.[chS].

When you're done, typing date to an xv6 shell prompt should print the current UTC time by running your date.c program. There is no specific format required for printing the date and time, just as long it is clear (e.g. 5:30 pm January 1, 2020).

Free Response Questions

1 - A system call is always triggered by hardware interrupts (T/F) (1 point)

2 - _____ provide(s) some interface to the services provided by the OS. (1 point)
A) Fork
B) System calls
C) Tests
D) Trace

3 - In Windows, CreateProcess() sys call that creates a new process. What is its equivalent in xv6? (1 point)
A) NTCreateProcess()
B) process()
C) fork()
D) getpid()

4 - Describe the relationship between a system-call interface and the operating system. Also, describe the relationship of them with API. (4 points)

5 - Describe either one aspect you think was well done about this first project or give constructive feedback about how you think it could be improved. (3 points)

Grading

Total number of the points for this project is 100.

Report (20 points):

See Submission Instructions document in Resources for details.

- DESIGNDOC.txt/pdf (10 points) (Describe your changes to the xv6 code)
    - Name the files your modified and briefly describe the changes
    - Please don't modify any files you don't need to! You'll make grading harder for us if you do.
- README.md (5 points)
- Your code must match the general code style of xv6 (5 points)

Trace System Call (50 points):

Incorrect number of total syscalls output by the test will result in a 10-point deduction

Missing any one of the following items will results in a 10-point deduction (40 points total).

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

Date System Call (20 points):

Typing date into the xv6 shell prints out the date and time in a recognizable format (e.g. 5:30 pm January 1, 2020).

Free Response Questions (10 points):

Submit FRQs on BLACKBOARD.

Project Submission

See Submission Instructions document in Resources