CS 451 – Operating Systems

Lab 03 Assignment

Due: 11:59 PM on Tuesday, 11 August 2020

You are to submit your files, to the Lab 3 folder on Blackboard by the due date and time.

Objective

- To familiarize yourself with the xv6 environment
- To understand the default scheduling algorithm (round-robin)
- To add a user program to xv6
- To add a system call to xv6

Note: Strongly recommend a VM with 64-bit Ubuntu >= 16.04

Introduction to xv6

Xv6 is a reimplementation of the Unix sixth edition in order to use as a learning tool. xv6 was developed by MIT as a teaching operating system. A vital fact about xv6 is that it contains all the core Unix concepts and has a similar structure to Unix even though it lacks some functionality that you would expect from a modern operating system. This is a lightweight operating system where the time to compile is very low and it also allows remote debugging.

Setting up the Environment

Step 1: Dependencies

First of all, we have to install some tools and packages on our Ubuntu Linux system as follows.

```
sudo apt-get update
sudo apt-get install build-essential
sudo apt-get install gcc-multilib
sudo apt-get install qemu
sudo apt-get install git
```

Step 2: Download Source Code

The source code of xv6 can be cloned to your machine as follows.

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

Step 3: Compile xv6 and run the emulator qemu

Navigate to the directory to which you cloned the source code.

```
$ cd xv6/
```

Now compile the program

```
$ make
```

The emulator will start but it will echo on the terminal too. Better to run it with the following frag (preferred):

```
$ make qemu-nox
```

Note: To enter the qemu console at any time, press `ctrl+a`, then c. Then, type "quit" to exit the emulator

If you have a 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
```

Task 1: Add a user program to xv6

Once you have setup xv6 on your machine, you could have a look at how to add a new user program to xv6. A user program could be a simple and straightforward C program. However, just adding a file to the xv6 folder would not be sufficient as we need to make it available to the user at the shell prompt.

Step 1: Write a simple C program

Create a C program as follows and save it inside the xv6 operating system folder. This can be named **spin.c.** This program should accept an argument (integer) and loop over the number.

```
int
main(int argc, char *argv[])
{
     for(i=0; i< atoi(argv[1]); i++ )
          sum = sum + i ;

     exit();
}
```

Also, make sure that your user program includes the following header files instead of the regular std library and i/o files

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

Remember to have **`exit();`** (instead of **return**) at the end of the program.

Step 2: Edit the Makefile

The Makefile needs to be edited to make our program available for the xv6 source code for compilation.

Make changes to the Makefile as shown below:

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

Look for EXTRA=\ and add spin.c.

```
EXTRA=\
	mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
	ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
	spin.c\
	printf.c umalloc.c\
	README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
	.gdbinit.tmpl gdbutil\
```

Once the change is made, the user program is ready to be tested. Run the following commands to compile the source code of xv6. This would compile the whole system.

```
$ make clean

$ make

$ make qemu-nox

$ spin 1000000
```

Now add a print statement (in the appropriate location) inside the scheduler function in **proc.c**

```
cprintf("Process chosen: %s\t PID: %d\n",p->name, p->pid);
```

Run multiple spin commands to check for the switching (round robin) mechanism

```
$ spin 10000000 &; spin 10000000 &; spin 10000000
```

Task 2: Add a system call to xv6

What is a system call?

As you all know, an operating system supports two modes; the kernel mode and the user mode. When a program in user mode requires access to RAM or a hardware resource, it must ask the kernel to provide access to that particular resource. This is done via a system call. When a program makes a system call, the mode is switched from user mode to kernel mode. There are many system calls in an operating system which executes different types of tasks when they are called.

Let's write a system call to display the process state and it's Process ID

To add a custom system call that can be called in xv6's shell, you should so something with the following files:

- **syscall.h:** define the position of the system call vector that connect to your implementation
- **syscall.c**: define the function that connect the shell and the kernel, use the position defined in syscall.h to add the function to the system call vector
- **sysproc.c:** the real implementation of your method goes in here
- **user.h:** define the function that can be called through the shell
- **usys.S:** use the macro to define connect the call of user to the system call function
- **defs.h:** add a forward declaration for your new system call

Following are the steps to add a system call:

1. Start the procedure by editing ***syscall.h*** in which a number is given to every system call. This file already contains 21 system calls. In order to add the custom system call, the following line needs to be added to this file.

```
#define SYS_procStat 22
```

2. Next, we need to add a pointer to the system call in the ***syscall.c*** file. This file contains an array of function pointers which uses the above-defined numbers (indexes) as pointers to system calls which are defined in a different location. In order to add our custom system call, add the following line to this file.

```
[SYS_procStat] sys_procStat,
```

Note: When the system call with number 22 is called by a user program, the function pointer sys_procStat which has the index SYS_procStat or 22 will call the system call function.

3. The function prototype which needs to be added to the ***syscall.c*** file is as follows

```
extern int sys_procStat (void);
```

4. Next, we will implement the system call function. In order to do this, open the ***sysproc.c*** file where system call functions are defined. Add the following function at the end of the file.

```
int
sys_procStat(void)
{
     return procStat();
}
```

5. Download the function skeleton provided (**procStat**). Next, open proc.c and fill it referring to the scheduler() function in **proc.c**. Add your code to ***proc.c*** at the end**.**
6. **In defs.h** add a forward declaration for your new system call under the section //proc.c

```
int  procStat(void);
```

7. In **user.h** define the function that can be called through the shell

```
int procStat(void);
```

8. In **usys.S** use the macro to define connect the call of user to the system call function

```
SYSCALL(procStat)
```

9. Finally, add the user file **ps.c** (**provided**) into the xv6 folder. Remember this is a user file, hence you will be required to modify the Makefile. (**Refer to the previous task**)

Compile the source code. Make sure you are not getting any errors!

Once your code compiles without any errors run using

```
$ make qemu-nox
```

Now you are ready to run your user program and see how your system call runs.

What to turn in

- Compress the parent folder (xv6) along with your created files, to zip and upload the zipped folder.
- A README file (outside the parent folder) specifying the changes you made including a brief description of the user command and the system call