PROJECT 2: SYSTEM CALLS

Ricardo Valencia

PORTLAND STATE UNIVERSITY - CS333: Intro to Operating Systems

Project Description

This project focuses on the invocation path for system calls in xv6, implementing a new system call that utilizes the hardware timer, implementing a new system call that passes a data structure into and out of the kernel, and implementing a new user-level program that utilizes our new system call to time system calls or programs.

Project Deliverables

System Call Tracing

- syscall.c

Date System Call

- usys.S
- user.h
- sysproc.c
- date.c
- syscall.h
- syscall.c

User-Level Time Command

- time.c
    - This goes under the assumption that the timing of an operation would never exceed an hour since the hour is not recorded. It also records the time it takes for the time command to know if a command is not actually a program.

Project Implementation

System Call Tracing

The first step in printing out the system calls was to locate the syscall.c and syscall.h files where the rest of the system calls are stored. First, I started out by creating a new char* array that stored the system call number which mapped to the corresponding system call name. This was done for every system call in order to print the correct system call that is called in my syscall()

function also located in syscall.c. It's important to note, however, that this new data structure was stored inside an #ifdef statement that would only compile if the appropriate CFLAG was included in my Makefile.

Then, after my new char* array was complete, I added the print statement after the system call in the syscall() function. This print statement was also wrapped in an #indef statement so that it will only compile if the proper CFLAG is included in the Makefile. This print statement utilized the new char* array to print the correct system call which used a local variable, num, that took the value of the %eax register from the current process's trap frame, then it used the return value of the system call – which was stored back into the %eax register – and displayed that as well. Doing all of this allowed me to boot up the xv6 kernel and print all the system calls as it booted up, and then any additional system calls done while the kernel is running.

Date System Call

The next step was to create a new system call that would utilize the hardware's timer. First I started out by making sure that this new system call would print correctly by altering the syscall.h and syscall.c files. In the syscall.h file, I added sys_date to the list of the other function calls and added the correct system call number along with it. Then I added sys_date to every system call list located in the syscall.c file, as well as the print list in order to be properly printed when it's called with my system tracer.

Next, I added the new system command into the usys.S file that created the connection between the call in the terminal to the actual system call. In order for it to show up in my xv6 terminal, however, I had to add it to my Makefile under UPROGS.

Then I had to implement the new system call, so first I started by adding the function prototype into the user.h file along with all the other system calls(user.h:26). This function prototype has a rtcdate data structure being passed in as an argument from the shell that will be used in the kernel. Then, in sysproc.c, I implemented the sys_date() function that is used in the kernel. There is no argument being passed in though, so in my implementation I utilized the argptr(..) function

located in syscall.c that retrieves the rtcdate object argument that was passed from the stack. Then it uses cmostime(..) and passes in the rtcdate object as an argument to retrieve the correct time and date form the hardware. Having done all this successfully, I return 0, but if sys_date() fails to retrieve the rtcdate object from the stack, then it returns -1.

Finally, I need to create the shell program that will call my new system call. This was implemented entirely in date.c. This calls the new date system call and passes in an rtc object as an argument, and returns if it failed to retrieve that object from the stack. If it was successful, it prints the date and time into xv6.

User-Level Time Command

My final task was to create a new command that allowed the user to time a program that was passed in from the shell done entirely in time.c. The command is called by typing "time [program]" into the bash, but in order for this to be callable I had to add _time to the UPROGS list in the Makefile.

The program starts by taking in the program command line argument and storing it into it's own array for further processing later in the program. Then it records the time before the program executes by utilizing my new date system call and stores it in a rtcdate variable. Then I created a new process by calling fork(), and in the parent process I called wait() so the child process could execute. In the child process, it executed the program by calling the exec(..) system call and passing in the program variable as the arguments and prints an error if it failed to execute the program. If it was successful, it would close out of the child process in which the parent process would take control again. Then the date system call is called again to record the new time after the timed-program is done running.

In order to find the time the program took to run, I use the last time recorded with the initial time and subtract the difference between the two to get the elapsed time. This is what is then printed to the xv6 console for the user to see how long a program took to run.

Testing Methodology

System Call Tracing

As mentioned previously in the report, I had to add a CFLAG into the Makefile that would allow the #indef structure to print out all the system calls. In order to test this, I included *CFLAGS += -DPRINT_SYSCALL* in my Makefile and booted up xv6 (Makefile:80). As it was booting, all the appropriate system calls were being printed along with their return values which can be seen here:

```
gwrite -> 1
 write -> 1
swrite -> 1
hwrite -> 1

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

This output matches the required output that the project 2 report displays, so the results seem to be correct. After proving it works, I comment out the new addition to the Makefile to remove the cluster from xv6 while I use it for other tasks and it stops printing.

Date System Call

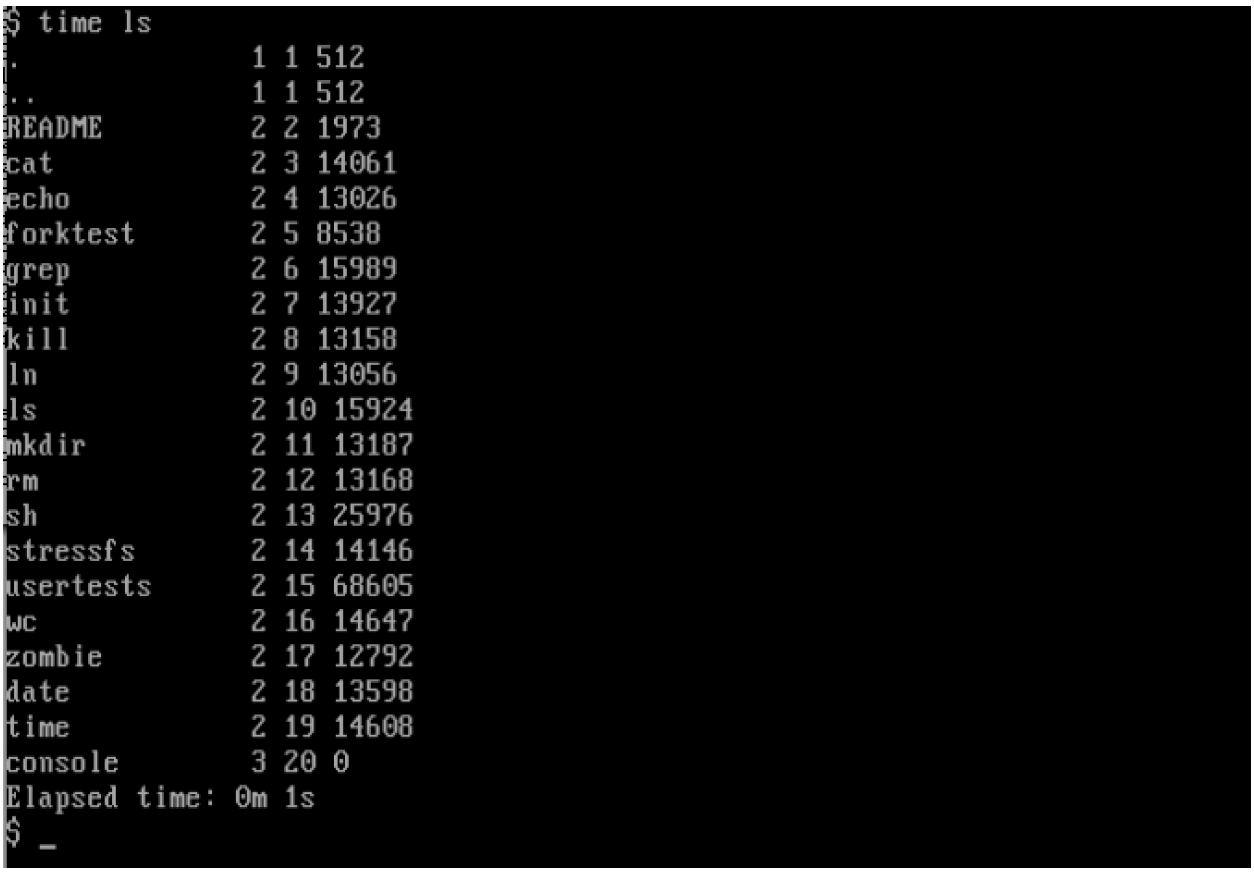
The new system call is a rather straight forward test – I boot up xv6 and type “date” right into the command line. Doing so prints out the current time in the UTC time zone, which can be seen as follows:

```
Booting from Hard Disk...

cpu0: starting xv6

cpu1: starting
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ date
Jan 20, 2016 9:42:2 UTC
$ _
```

This new system call is very simple, so there is not much to test besides this with this command.

User-Level Time Command

In order to test the functionality of this new command, I booted up xv6 and typed “time [program]” into the bash prompt. In place of [program], I used ‘ls’ to time how long listing my current directory would take and I was given the following:



I proceeded to run this on multiple different programs such as the date system call and usertests.c as well in order to see if the timing was working correctly. The usertests file took quite some time to run, and it is reflected by the time recorded and proves to be accurate. The date system call is almost as fast as the ‘ls’ call, so they matched up very closely too. Both of these results can be seen here:

```
rmdot test
rmdot ok
fourteen test
fourteen ok
bigfile test
bigfile test ok
subdir test
subdir ok
linktest
linktest ok
unlinkread test
unlinkread ok
dir vs file
dir vs file OK
empty file name
empty file name OK
fork test
fork test OK
bigdir test
bigdir ok
exec test
ALL TESTS PASSED
Elapsed time: 10m 9s
$
```

```
$ time date
Jan 20, 2016 10:30:10 UTC
Elapsed time: 0m 0s
$
```

These seemed to be sufficient in testing that the time command worked correctly for proper programs, but then I tested to see if it worked on improper programs. For example, in this test case, I used a non-existent program called garbage and ran it using my time command. As you can see, the time command didn't execute the command, but it did give the time it took for it to know that it was a bad command.

```
$ time garbage
Error: execution failed
Elapsed time: 0m 1s
$
```