Alex Ferguson
CS333
4/8/17

Project 1 Write-Up

Description:

The goal of this assignment was to introduce us to some of the basic concepts of an operating system, to reintroduce us to the C programming language, and to solidify some of the concepts that we learned in CS201. During this assignment I learned about system calls and how programs (processes) request functionality from the kernel, I got the opportunity to learn how to add a new system call to the kernel (the date system call), and I learned more about the life cycle of a process by outputting the running time of the process. I also learned about conditional compilation and how it can be an important feature when building software across multiple different operating systems.

Deliverables:

The following functionality was added to the xv6 operating system:

- System call tracing, implementing conditional compilation, to output the following information when the proper flag is enabled:
  <system call name> → <system call return value>
- Added the date() system call that returns the current date and time information in UTC format.
- Added the date command that, when typed into the shell, outputs the date information in UTC format that was returned by the date() system call.
- Added functionality for capturing the start time of a process upon its creation, utilizing the global ticks variable, in order to capture the total running time of the process upon its destruction.
- Modified the output for the control sequence, Ctrl-P, to include the total running time of each process as well as the traditional PID, state, name, and process codes.

Implementation:

System Call Tracing:
Conditional compilation was used to ensure that all system call tracing code only gets executed if the PRINT_SYSCALLS flag in the Makefile (line 73) is turned on (uncommented).

- **Syscall.c lines 131-157** defined a static array, named syscallnames[], that follows the same conventions as the syscall[] array directly above it to map system calls to their appropriate name in string format.
- **Syscall.c lines 168-170** prints the system call name as well as the return value of the system call to the screen.

Date System Call:
Code was added to the following files in order to implement the date() system call in xv6:

- **user.h** The function prototype for the user version of the date() system call was added (line 27). The prototype for this function has an int return value and takes in a pointer to a user-defined rtcdate variable, defined in date.h, and looks like this:
  ```
  int date(struct rtcdate*);
  ```

- **usys.S** A user-side stub for the date system call was added (line 33) in order to gain access to kernel-mode.
- **syscall.h** The date system call number (line 25) was added to the list of already existing system call numbers following the same system of adding one to the prior system call number in the list. This maps the system call name SYS_date to its appropriate system call number.
- **syscall.c** The entry point for the sys_date routine (line 102), implemented in sysproc.c, was added to the end of the list of other extern routines already declared. An entry for date was added to the syscall[] function dispatch table (line 127) as well as an entry to the syscallnames[] array table (line 155) to ensure that the system call name is printed when the PRINT_SYSCALLS flag is enabled. Because the sys_date function retrieves its arguments from the stack using argptr() there is no need to pass any arguments into the function. For this reason the function is given a void parameter.
- **sysproc.c** The kernel-side implementation of the sys_date function was done in this file (lines 97-107). This function uses argptr() to grab the pointer to an rtctime argument from the stack. It then passes that argument to the function cmostime() (line 105) along with a pointer to an rtcdate variable (line 100). cmostime() fills that variable with the appropriate time information. sys_date returns 0 as a sign of success.

Date User Command:

The user-side date command, in the date.c file, uses the date() system call (kernel-side) to populate an rtcdate variable that is passed into the user-side date command. The user-side date command is where the date and time information is then displayed since system calls do not display any non-error/non-debugging related information. The kernel-side date system call functions as specified above.

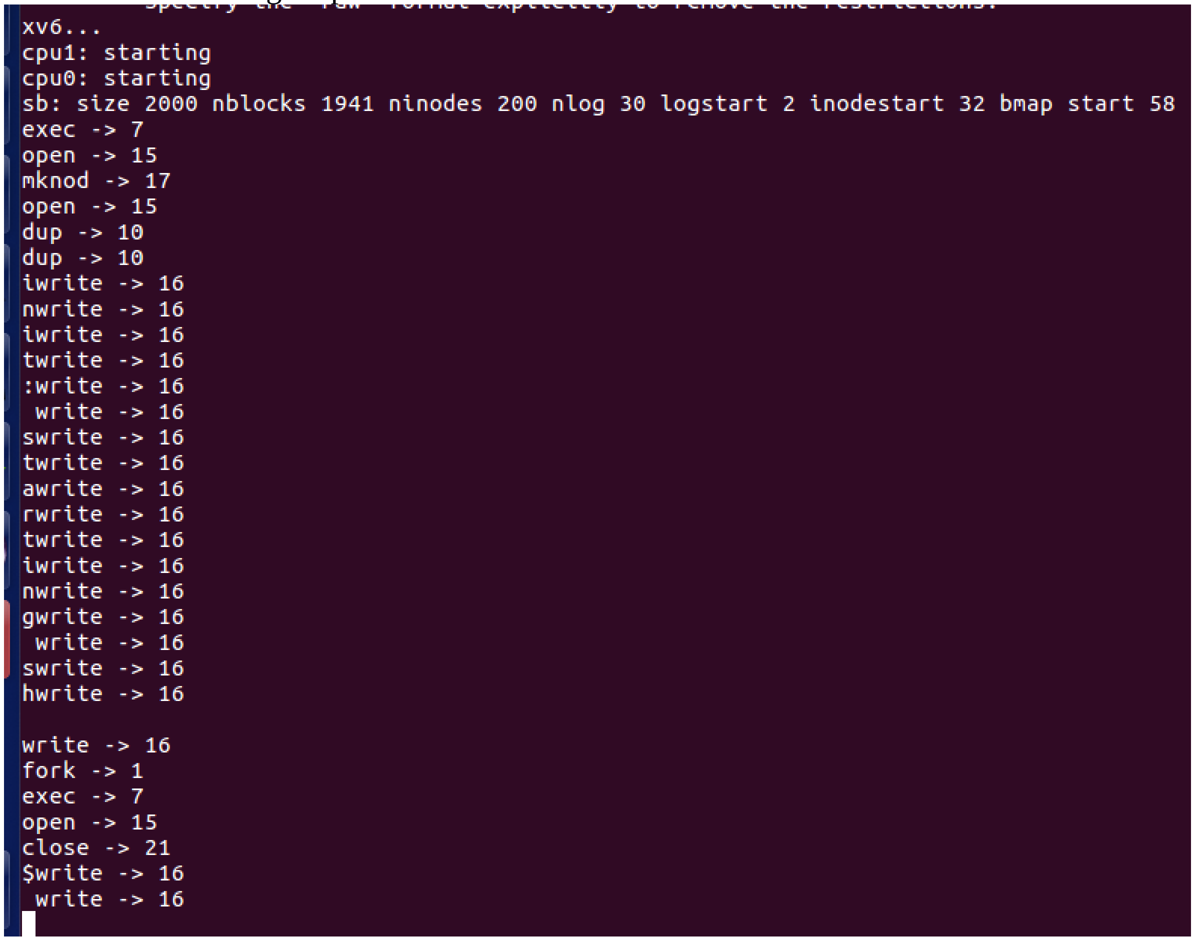
Control-P and Elapsed time Modifications

In order to get useful debugging information the Ctrl-P sequence is used to print the PID, the process state, process name, and process codes. I modified the program to also print the process' running time when Ctrl-P is pressed.

- **proc.h** A uint variable named start_ticks was added to the proc.h file (line 69). This variable acts as a way to measure the start time of a process.
- **proc.c** In allocproc() the start_ticks variable is initialized (line 73) to the global ticks variable as the process is created. In order to calculate the total running time of the process I added a few lines of code to procdump() (lines 518-520). I created two variables, uint seconds and uint partial_seconds (lines 518 and 519), to store the elapsed time. Seconds captures the full second value by first subtracting start_ticks from the global ticks variable and then integer dividing that value by 100 to get the seconds. Partial_seconds captures the partial second value by subtracting start_ticks from the global ticks variable and then modding by 100 to get the remainder of that value. These two values are then used in the print statement (line 520) for the overall running time of each individual process. A bug fix was added (lines 521-523) for printing partial_seconds when the value of partial seconds is less than 10. Originally, when the value was less than 10, 08 for example, it would print .8 instead of .08 for the partial second value. It has been fixed to function properly, printing the correct value of .08 for example. I also added a print statement (line 510) to print out the name of each column of output for Ctrl-P in order to label the output so that it is meaningful.

Testing:

System Call Tracing Facility:
To test the system call tracing the PRINT_SYSCALLS flag was uncommented (turned on) in the Makefile and the following output was observed:



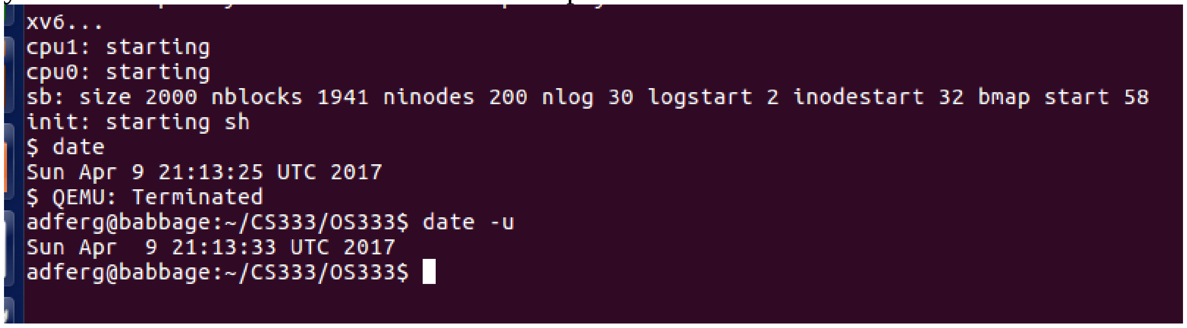
```
         Specify the 'raw' format explicitly to remove the restrictions.
xv6...
cpu1: starting
cpu0: starting
sb: size 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
exec -> 7
open -> 15
mknod -> 17
open -> 15
dup -> 10
dup -> 10
iwrite -> 16
nwrite -> 16
iwrite -> 16
twrite -> 16
:write -> 16
 write -> 16
swrite -> 16
twrite -> 16
awrite -> 16
rwrite -> 16
twrite -> 16
iwrite -> 16
nwrite -> 16
gwrite -> 16
 write -> 16
swrite -> 16
hwrite -> 16

write -> 16
fork -> 1
exec -> 7
open -> 15
close -> 21
$write -> 16
 write -> 16
```

Figure 1: System Call Tracing

As you can see in Figure 1 the following output verifies that the system call tracing portion of the assignment correctly displays the system calls that get invoked and, in the scope of this assignment, standard output is appropriately mixed with trace output, displaying the string “init: starting sh”. This test PASSES.

Date System Call and User Command:
In order to test the date system call, since we cant invoke the system call from the shell, I used the date user command along with the date -u linux command to check and see that the xv6 date command closely matched the linux date command. The output is as follows:

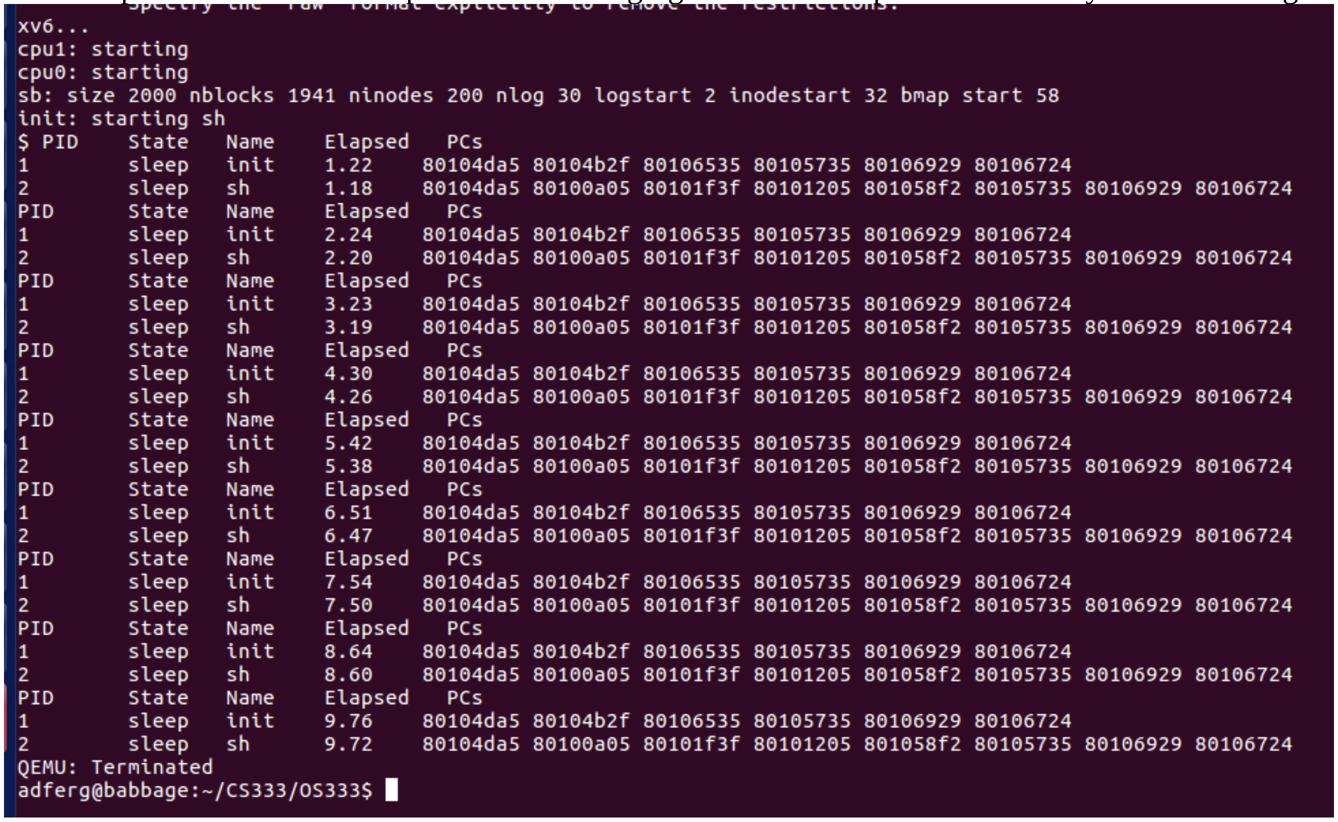


```
xv6...
cpu1: starting
cpu0: starting
sb: size 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ date
Sun Apr 9 21:13:25 UTC 2017
$ QEMU: Terminated
adferg@babbage:~/CS333/OS333$ date -u
Sun Apr  9 21:13:33 UTC 2017
adferg@babbage:~/CS333/OS333$
```

Figure 2: Date System Call and User Command

As you can see in Figure 2 the xv6 date command invokes the date() system call properly and differs from the linux date -u command only by the amount of time it took to exit the xv6 shell and invoke the linux date command.
This test PASSES.

Ctrl-P and Elapsed Time:
To test the Ctrl-P sequence and the elapsed time functionality I booted qemu and pressed Ctrl-P multiple times at an interval of approximately one second between each press. Pressing Ctrl-P multiple times helps to show that the elapsed time is changing between each press while the system is running.



```
xv6...
cpu1: starting
cpu0: starting
sb: size 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ PID    State   Name    Elapsed   PCs
1        sleep   init    1.22     80104da5 80104b2f 80106535 80105735 80106929 80106724
2        sleep   sh      1.18     80104da5 80100a05 80101f3f 80101205 801058f2 80105735 80106929 80106724
PID      State   Name    Elapsed   PCs
1        sleep   init    2.24     80104da5 80104b2f 80106535 80105735 80106929 80106724
2        sleep   sh      2.20     80104da5 80100a05 80101f3f 80101205 801058f2 80105735 80106929 80106724
PID      State   Name    Elapsed   PCs
1        sleep   init    3.23     80104da5 80104b2f 80106535 80105735 80106929 80106724
2        sleep   sh      3.19     80104da5 80100a05 80101f3f 80101205 801058f2 80105735 80106929 80106724
PID      State   Name    Elapsed   PCs
1        sleep   init    4.30     80104da5 80104b2f 80106535 80105735 80106929 80106724
2        sleep   sh      4.26     80104da5 80100a05 80101f3f 80101205 801058f2 80105735 80106929 80106724
PID      State   Name    Elapsed   PCs
1        sleep   init    5.42     80104da5 80104b2f 80106535 80105735 80106929 80106724
2        sleep   sh      5.38     80104da5 80100a05 80101f3f 80101205 801058f2 80105735 80106929 80106724
PID      State   Name    Elapsed   PCs
1        sleep   init    6.51     80104da5 80104b2f 80106535 80105735 80106929 80106724
2        sleep   sh      6.47     80104da5 80100a05 80101f3f 80101205 801058f2 80105735 80106929 80106724
PID      State   Name    Elapsed   PCs
1        sleep   init    7.54     80104da5 80104b2f 80106535 80105735 80106929 80106724
2        sleep   sh      7.50     80104da5 80100a05 80101f3f 80101205 801058f2 80105735 80106929 80106724
PID      State   Name    Elapsed   PCs
1        sleep   init    8.64     80104da5 80104b2f 80106535 80105735 80106929 80106724
2        sleep   sh      8.60     80104da5 80100a05 80101f3f 80101205 801058f2 80105735 80106929 80106724
PID      State   Name    Elapsed   PCs
1        sleep   init    9.76     80104da5 80104b2f 80106535 80105735 80106929 80106724
2        sleep   sh      9.72     80104da5 80100a05 80101f3f 80101205 801058f2 80105735 80106929 80106724
QEMU: Terminated
adferg@babbage:~/CS333/OS333$
```

Figure 3: Ctrl-P and Elapsed Time

As you can see in Figure 3 both the appropriate information (PID, state, name, elapsed time, and PCs) gets printed every time and the elapsed time is being calculated appropriately with a difference of about one second between each press. Additionally, the two processes are off by a constant of .04 across all presses. This makes sense because the starting process init should have a slightly higher running time than sh in each scenario.

Because the appropriate information is being printed, the elapsed time during runtime is consistent across all presses of Ctrl-P, and a consistent runtime difference between the init and sh processes exists I can confidently say this test PASSES.