Kathleen Tran
Winter 2016

Project 2 | Project Report

1 | Assignment Description

This project consists of three implementations: the first is a system call using a hardware timer, the second is a system call to support the passing of a data structure into and out of the kernel, and the third is a user-level program to utilize the aforementioned system calls.

For the first of the three core tasks, *System Call Tracing*, the xv6 kernel needs to be modified such that it will print out one line (the system call name and its return value) per system call invocation. In the second task, *Date System Call*, a new system call must be added to xv6 that will get and return the current UTC time to the user program. The last of the three tasks, *Time Command*, will use the date system call to implement an xv6 version of the standard Unix "time" command.

2 | Assignment Deliverables

- System Call Tracing: `syscall.c`
- Date System Call: `date.c`
- Time Command user-level program: `simpletime.c`

Assumptions made whilst producing the deliverables consist of the following: (a) the `date` system call and its utilization, at least in terms of where code needs to be added, are similar to that of `uptime`'s, and (b) whatever arguments that are to be called with the system call `simpletime` will result in a value with a lower bound duration of `0m0s` to execute.

3 | Implementation

SYSTEM CALL TRACING. The first out of the three main deliverables was addressed and achieved through the modification of the `syscall.c` file. In order to trace each system call during the booting of xv6, certain additions were made to the `syscall()` function to print out the name of the system call alongside its return value. To obtain the system call names, a new character array, `syscallnames`, was created and maps existing system calls to their names represented via strings. Indexing into said array retrieves the name, while the corresponding return value is retrieved through an updated call to `proc->tf->eax`, where it was stored. These additions are wrapped in a new compiler flag such that it could be disabled by commenting out the flag in the Makefile (e.g., the added line: `CFLAGS += -DPRINT_SYSCALLS`).

Kathleen Tran
Winter 2016

DATE SYSTEM CALL. The second portion of the project established a new system call that displays the current date in UTC time. The implementation of this system call spans across various files, with the key components residing in the `sysproc.c` and the `date.c` files. Prototypes and the like are placed into the same additional files as where the `uptime` system call can be found (e.g., `syscall.c`, `user.h`, and `usys.S`). The key implementation of the date function is found in `sysproc.c`, where it examines whether or not the pointer for the argument, `struct rtcdate`, lies within the process address space prior to passing it into the helper `cmostime()` function, where its data members get populated. Then in `date.c`, aside from the provided template listed in the assignment PDF, it is just a matter of formatting the data members to showcase the date. With the exception of the day of the week, the `date` system call has been formatted like such:

```
Mmm d h:m:s UTC yyyy
```

where `Mmm` is the first three letters of the Month's name in English, in order to resemble that of Unix's `date -u` display as closely as possible.

TIME COMMAND. The last deliverable within this project entailed writing a user-level program that replicates Unix's `time` command, making use of the previously implemented `date` system call. This user program can be found in the `simpletime.c` file. The example code that is listed in the assignment PDF was closely referenced for the creation of the code for xv6. Valid functions were found within the `sysproc.c` file and were substituted into the general structure. Two `rtcdate` structs are used to hold the two values necessary for the duration calculation. After checking for a valid argument count, the arguments are [shallow] copied into a new character array called `p`. Then, the first struct gets passed into the `date` system call to obtain the date at which the process began. We then fork* and execute on the child process whilst calling `wait()` on the parent process. Once the child process finishes, the parent process resumes and the second struct is used to capture the end date. Finally, simple math is applied to the data members of the two `rtcdate` structs in one of two logical cases to obtain the elapsed time, measured in minutes and seconds. The duration is simply formatted as `0m0s`, where `0` is a placeholder for some integer, `m` is minutes, and `s` is seconds.

**In invalid instances, like a failed fork or an invalid command, an error message is printed to standard error. The printing of some error message also applies to the unsuccessful path of any sections and/or points of checking for valid data.*

4 | Testing Methodology

DATE_SYSCALL_TEST

In order to test the correctness of the implementation of the new date system call, various consecutive calls to `date` and `date -u` were made from QEMU and Linux, respectively, to verify that the times were within the general vicinity of each other (e.g., within a few seconds of each other, factoring in human delay in switching between the terminals to enter in the system call).

_DATE_SYSCALL_TEST OUTPUT:

```
$ date
Jan 19 7:37:10 UTC 2016
$ date
Jan 19 7:37:22 UTC 2016
$ date
Jan 19 7:37:26 UTC 2016
```

```
kathtran@ada:~/cs333/xv6-psu-kernel$ date -u
Tue Jan 19 07:37:11 UTC 2016
kathtran@ada:~/cs333/xv6-psu-kernel$ date -u
Tue Jan 19 07:37:23 UTC 2016
kathtran@ada:~/cs333/xv6-psu-kernel$ date -u
Tue Jan 19 07:37:25 UTC 2016
```

Above left: output from QEMU.
Above right: output from Linux.

In the screen captures above, a total of three pairs of calls to the `date` system call were made between QEMU and Linux. In the top two calls across the two systems, the `date` system call was first entered into QEMU, and then into the Linux terminal. However, in the last call, the `date` system call was first called in the Linux terminal, and then in QEMU. It just so happens here that none of them were delayed by more than one second from each other.

TIME_CMD_TEST_DATE

The time command was tested in a similar manner to the `DATE_SYSCALL_TEST` in regards to running the command various times and then comparing and examining the relation between the outputs. In this case, the time command, run by typing `simpletime` followed by the desired arguments, was coupled with the `date` system call.

_TIME_CMD_TEST_DATE OUTPUT:

```
$ simpletime date
Jan 19 8:10:9 UTC 2016

0m0s
$ simpletime date
Jan 19 8:10:15 UTC 2016

0m0s
$ simpletime date
Jan 19 8:10:24 UTC 2016

0m1s
```

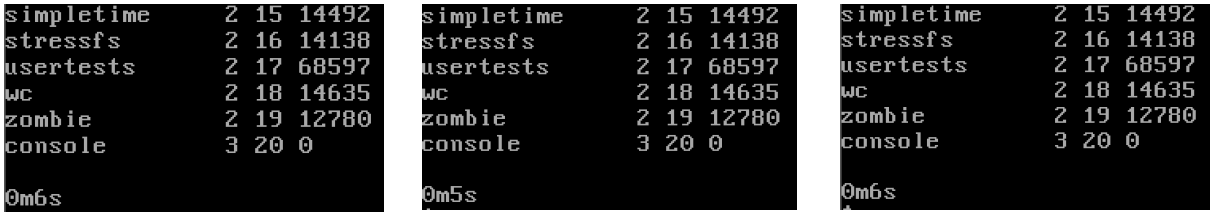
Left: three consecutive calls to simpletime using date as an argument.

From the screen capture to the left, it may be noted that the `date` system call does not take very long to execute. Many of the times in which it was executed during the testing process, the time duration came out to be equivalent to the lower bound: `0m0s`.

TIME_CMD_TEST_LS

In this test case for the time command, `ls` was used as an argument in hopes of testing against a command that is more probable to take more than zero or one second to finish execution.

_TIME_CMD_TEST_LS OUTPUT:



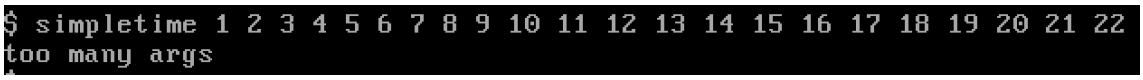
Above: three consecutive calls to simpletime using ls as an argument.

In the screen captures above, the `simpletime` command was run three different times with `ls` supplied as the singular argument. In comparison to the TIME_CMD_TEST_DATE outputs, it is apparent that `ls` takes a longer time, on average, to execute. Furthermore, this supports demonstrating the success of the time command implementation since we may reasonably extrapolate as to way it might take longer for a `ls` command to execute than that of *date*, where a single line of data is returned via the system call.

TIME_CMD_TEST_TOOMANYARGS

In the implemented program for `simpletime`, there exists a limit on the number of arguments that may follow the command; the upper bound was arbitrarily set to 20. In this particular test case, the checkpoint is put to the test by providing a list of more than 20 arguments alongside the `simpletime` command. The expected message to be printed to standard error in such an instance is "*too many args*."

_TIME_CMD_TEST_TOOMANYARGS OUTPUT:



Above: a call to simpletime with 22 arguments (the program takes a maximum of 20 arguments).

TIME_CMD_TEST_INVALIDARGS

In addition to the error check noted in TIME_CMD_TEST_TOOMANYARGS, there is also a check for valid argument(s). In the case where the argument is either an invalid call or an invalid command, the program is expected to print "`Error: exec failed. [Arg] probably needs full path`" to standard error.

_TIME_CMD_TEST_INVALIDARGS OUTPUT:

```
$ simpletime test
Error: exec failed. test probably needs full path

0m0s
```

Above: one of many invalid cases: calling simpletime with an invalid command.

5 | Bugs and Bug Fixes

BUG. Albeit an issue with little-to-no urgency, the `strcpy()` function found in `ulib.c` was not cooperating with the code written for the `simpletime` command user-level program. The intention was to use said function to perform a deep copy of the command line arguments into a new array that is later accessed for execution as well as for further specification in regards to the source of error in one of the error checks (e.g., populating the corresponding message that is printed to standard error). Although the program was still able to execute, printing out the contents of the array resulted in `null` each time the error check was encountered.

BUG FIX. As a possibly temporary fix, the deep copy was swapped for a shallow copy. In swapping out the line of code for a mere assignment statement, the print out has been able to display the correct argument via the new array, and the program continues to run successfully.

6 | Project Feedback and Suggestions for Improvement

Although it has already been noted throughout the various threads in the class mailing list, I would just like to concur with the addition of the very helpful information that was mailed out to the class every now and then within these past few days (all pertaining to this assignment) to the future class documents. At times, the initial Project 2 description felt too vague and lacked clear direction. I just hope that with future project descriptions, there will be a more focused target stated and no crucial bits of code to be appended to existing files left out. On a positive note, the recommended xv6 chapters to be read/referenced in conjunction with the assignments have been a helpful resource.