A Note on this Report

This report is a *starting point* for documenting project one. Proper communication of your work within a large project is important. In this class, you will practice writing project reports that follow a specific format, exemplified by this document and described in the course *Survival Guide*. At various points there will be asides with additional information.

The report structure is specific and formal. The principle goal of the project reports is to familiarize students with one way to communicate technical information to a technical audience. It is reasonable and prudent that students become familiar with the basics of such communication.

To obtain the LaTeX source for this document, change the ".pdf" in the URL to ".tex".

Description

For this assignment, I learned about the flow of control for system calls in xv6; how to add a new system call; how to access specific information for each active process; and how to use conditional compilation to enable and disable kernel features.

Aside: descriptions for all objectives for the assignment are given here in brief. DO NOT create a separate report for each objective or sub-project. A narrative or list style can be used. You can use first person or third person in the report, there is no preference.

Deliverables

The following features were added to xv6:

- A system call tracing facility that, when enabled, prints the following information to the console:

  ```
  <system call name> -> <system call return code>
  ```

- A new system call, `date()`, that returns the current UTC date and time.
- A new user command, `date`, that prints the current UTC date and time to standard output.
- Each process now records the value of the `ticks` global variable when that process is created. This value is used to calculate *elapsed time* for each process.
- A modification to the existing control–p mechanism, which displays debugging information, to include elapsed time for each process.

Aside: All deliverables are listed in this section. Note that the descriptions are concise and no source code is included. ***Important:*** *there is both a system call and a user command with the same name,* `date()` *and* `date`*; each is documented separately.*

Implementation

Aside: you will need to fill in the actual line numbers for your project.

System Call Tracing Facility

All the code for the system call tracing facility was conditionally compiled using the `PRINT_SYSCALLS` flag in the `Makefile` (line XX). The implementation modified `syscall.c` as follows:

- Lines XX – YY define an array of system call names, `syscallnames[]`, indexed by system call number as defined in `syscall.h`.
- Lines XX – YY prints the name of the system call and the corresponding return value.

Date System Call

The following files were modified to add the `date()` system call.

- `user.h`. The user-side function prototype for the `date()` system call was added (line XX). The system call takes a pointer to a user-defined `rtctime struct`. The prototype is:
  `int date(struct rtcdate*);`
  The file `date.h` contains the `rtcdate` definition.
- `syscall.h`. The `date()` system call number was created by appending to the existing list (line XX).
- `syscall.c`. Modified to include the kernel-side function prototype (line XX); an entry in the function dispatch table `syscalls[]` (line XX); and an entry into the `syscallnames[]` array to print the system call name when the `PRINT_SYSCALLS` flag is defined. All prototypes here are defined as taking a *void* parameter as the function call arguments are passed into the kernel on the stack. Each implementation (e.g., `sys_date()`) retrieves the arguments from the stack according to the syntax of the system call.
- `usys.S`. The user-side stub for the new system call was added (line XX). This stub uses a macro that essentially just traps into kernel-mode.
- `sysproc.c`. Contains the kernel-side implementation of the system call in `sys_date()` (lines XX – YY). This routine removes the pointer argument from the stack and passes it to the existing routine `cmostime()` in `lapic.c` (line XX). The pointer argument is expected to be a `struct rtctime*`. The routine `cmostime()` cannot fail so a success code is returned by `sys_date()`.

Date User Command

The `date` user command is implemented in the file `date.c`. This command invokes the new `date()` system call to fill in the supplied `rtcdate struct`; passed by reference. The command then displays the date and time information on standard output. The return code from the system call is checked and handled as a user program does not know if a system call will succeed or fail.

Aside: note that we specifically state that the command, not the system call, displays the information. A system call should not print any information except in the case of a catastrophic error or debugging information. The note on the return code is a hint to the student as all system calls (with the exception of `exec()`*) provide a return code that must be checked and properly handled.*

control–p Modifications and Elapsed Time

The control–p console command prints debugging information to the console. The following modifications were made to capture and display elapsed time as part of the existing control–p debugging information.

- `proc.h`. A new field was added to `struct proc` named `uint start_ticks` for storing the time of creation (in *ticks*) for each process (line XX).
- `proc.c`. The routines `userinit()` (line XX) and `fork()` (line XX) were modified to correctly set `start_ticks` on process creation.
- `procdump()`. This routine in `proc.c` was modified to:
  - Print a header (line XX) to the console.
  - Calculate the *elapsed time* since process creation (lines XX – YY). This section calculates elapsed time as seconds and hundredths of seconds as the granularity of the `ticks` variable is at hundredths of a second.
  - Include the elapsed time in the display of process information on the console (line XX).

Aside: each deliverable has a subsection in the implementation section. The information includes where in xv6 modifications were required and a short description.

Testing

Aside: Each student will need to provide their own screen shots or other test output as well as the test description.

System Call Tracing Facility

I tested this feature by modifying the `Makefile` to turn on `PRINT_SYSCALLS` flag, then booting the xv6 kernel, and observing the following output:

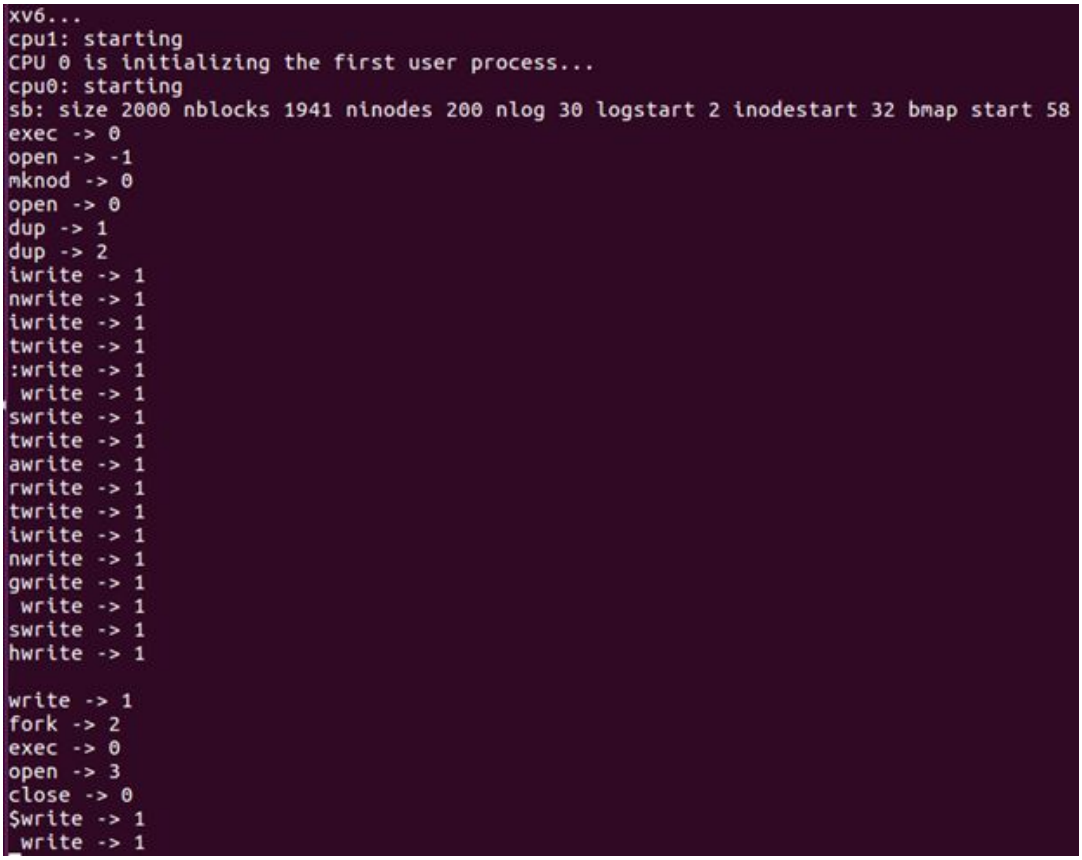

Figure 1: System Call Tracing Facility

The system call trace correctly displays invoked system calls. The standard output is interleaved with the trace output. The output "init: starting sh" is printed by the `init()` process (`init.c`) and the "$" is printed by the shell process (`sh.c`).

This test `PASSES`.

Date System Call and User Command

I am going to use the `date` command to test both the `date()` system call and date command, as I can't directly invoke a system call from the shell. My testing will invoke my `date` command in xv6 and then invoke the corresponding Linux `date` command and see if the former closely matches the latter.

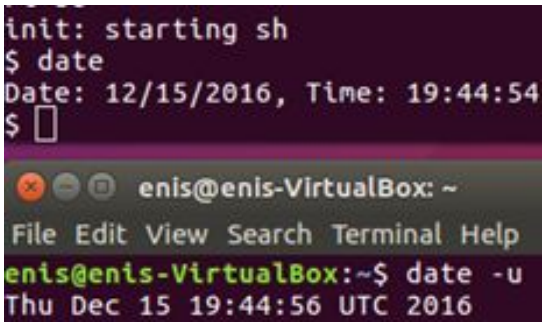

Figure 2: Date Test

The output from my `date` command closely matches the output of the Linux `date` command, except for a slight discrepancy in the number of seconds. This discrepancy is expected as it takes non-zero time to exit xv6. This test shows that the `date` command works correctly, along with the date system call, since the command prints out all of the information extracted by the system call.

This test `PASSES`.

control–p and Elapsed Time

The test for these will be split into two phases. My first test will show that control–p is outputting the correct information, while my second test will use the first test to show that the elapsed time is correct.

Here is the output of the first test:

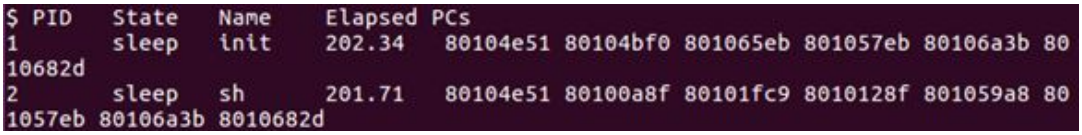

Figure 3: Control–p Test

The control–p output indicates that there are two processes running in xv6. This is correct. The first process is the initial process, here named "init", with a PID of 1. The second process is the shell, named "sh", with a PID of 2 (as it is the second process created). Note that the PCs appear to be correct, as they correspond to valid addresses in the kernel.asm file and the code for printing this information was not modified.

This sub-test `PASSES`.

For the second test, I will restart the kernel, and then press control–p several times, each press being within one second of the other. The results are shown below:

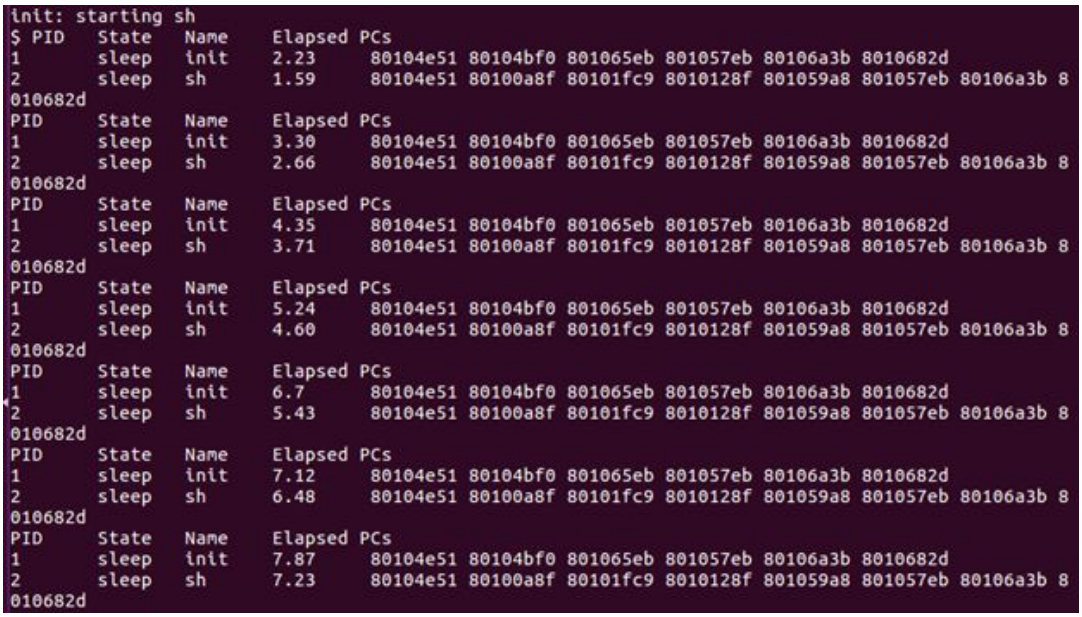

Figure 4: Elapsed Time Test

The elapsed time for the `init` process is 0.64 seconds higher than that of the `sh` process in all outputs. This makes sense as `init` starts before `sh`. Also, note that the elapsed times are steadily increasing by about one second with the same 0.64 s difference between `init` and `sh`.

This sub-test `PASSES`.

Because all sub-tests passed, this test `PASSES`.

Aside: note that each test explains the test, including expected output; then the test results are shown; then the results are discussed; and then an indication as to whether the test `PASSES` *or* `FAILS` *is given.*