Project One Report
Introduction to Operating Systems
Spring 2017

Andy Keene

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.

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.

Implementation

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 73). The implementation modified `syscall.c` as follows:

- Lines 131 – 157 define an array of system call names, `syscallnames[]`, indexed by system call number as defined in `syscall.h`. Here the table is conditionally compiled, to save space, when the `PRINT_SYSCALLS` flag is set, so the line numbering included the conditional compilation directives.
- Lines 169 – 171 prints the name of the system call and the corresponding return value (line numbering includes the conditional compilation directives).

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 27). 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 25).
- `syscall.c`. Modified to include the kernel-side function prototype (line 102); an entry in the function dispatch table `syscalls[]` (line 127); 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 33). 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 96-106). This routine removes the pointer argument from the stack and passes it to the existing routine `cmostime()` in `lapic.c` (line 205). 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.

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 69).

- `proc.c`. The routines `userinit()` (line 102) and `fork()` (line 161) were modified to correctly set `start_ticks` on process creation.

- `procdump()`. This routine in `proc.c` was modified to:

  - Print a header (line 519) to the console.
  - Calculate the *elapsed time* since process creation (lines 535-540). This section calculates elapsed time as seconds and hundredths of seconds as the granularity of the `ticks` variable is at hundredths of a second. Note that the display will be

    `<seconds>.<hundredths of second>`

    including leading zeroes, so that if a process has been running for 5 seconds and 9 one-hundredths of a second it will print *5.09*.
  - Include the elapsed time in the display of process information on the console (lines 538, 540). The calculation for whole seconds is done in-line - since unlike the hundredths of a second which is needed for the conditional statement *and* printing - its information needs only to be calculated when it is printed .

Testing

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:

```
xv6...
cpu1: starting
cpu0: starting
sb: size 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
exec -> 0
open -> -1
mknod -> 0
open -> 0
dup -> 1
dup -> 2
iwrite -> 1
nwrite -> 1
iwrite -> 1
twrite -> 1
:write -> 1
 write -> 1
swrite -> 1
twrite -> 1
awrite -> 1
rwrite -> 1
twrite -> 1
iwrite -> 1
nwrite -> 1
gwrite -> 1
 write -> 1
swrite -> 1
hwrite -> 1

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

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.

```
$ date
Sat Apr 8 19:8:46 UTC 2017
$
```

```
&e$date -u
Sat Apr  8 19:08:51 UTC 2017
&e$
```

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. Note also that the **date** command on xv6, unlike **date** on Linux, does *not* print leading zeroes (see line 32 in **date.c**); thus a further discrepancy in the output is due to **date.c**'s implementation, but not the system call itself.

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:

```
init: starting sh
$
PID     State     Name     Elapsed      PCs
1       sleep     init     0.84         80104db1 80104b3b 8010661d 8010581d 80106a11 8010680c
2       sleep     sh       0.79         80104db1 80100a05 80101f3f 80101205 801059da 8010581d 80106a11 8010680c
```

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:

```
$
PID     State   Name    Elapsed     PCs
1       sleep   init    1.01        80104db1 80104b3b 8010661d 8010581d 80106a11 8010680c
2       sleep   sh      0.96        80104db1 80100a05 80101f3f 80101205 801059da 8010581d 80106a11 8010680c

PID     State   Name    Elapsed     PCs
1       sleep   init    1.85        80104db1 80104b3b 8010661d 8010581d 80106a11 8010680c
2       sleep   sh      1.80        80104db1 80100a05 80101f3f 80101205 801059da 8010581d 80106a11 8010680c

PID     State   Name    Elapsed     PCs
1       sleep   init    2.82        80104db1 80104b3b 8010661d 8010581d 80106a11 8010680c
2       sleep   sh      2.77        80104db1 80100a05 80101f3f 80101205 801059da 8010581d 80106a11 8010680c

PID     State   Name    Elapsed     PCs
1       sleep   init    3.84        80104db1 80104b3b 8010661d 8010581d 80106a11 8010680c
2       sleep   sh      3.79        80104db1 80100a05 80101f3f 80101205 801059da 8010581d 80106a11 8010680c

PID     State   Name    Elapsed     PCs
1       sleep   init    4.74        80104db1 80104b3b 8010661d 8010581d 80106a11 8010680c
2       sleep   sh      4.69        80104db1 80100a05 80101f3f 80101205 801059da 8010581d 80106a11 8010680c

PID     State   Name    Elapsed     PCs
1       sleep   init    5.84        80104db1 80104b3b 8010661d 8010581d 80106a11 8010680c
2       sleep   sh      5.79        80104db1 80100a05 80101f3f 80101205 801059da 8010581d 80106a11 8010680c

PID     State   Name    Elapsed     PCs
1       sleep   init    7.21        80104db1 80104b3b 8010661d 8010581d 80106a11 8010680c
2       sleep   sh      7.16        80104db1 80100a05 80101f3f 80101205 801059da 8010581d 80106a11 8010680c

PID     State   Name    Elapsed     PCs
1       sleep   init    8.01        80104db1 80104b3b 8010661d 8010581d 80106a11 8010680c
2       sleep   sh      7.96        80104db1 80100a05 80101f3f 80101205 801059da 8010581d 80106a11 8010680c

PID     State   Name    Elapsed     PCs
1       sleep   init    8.84        80104db1 80104b3b 8010661d 8010581d 80106a11 8010680c
2       sleep   sh      8.79        80104db1 80100a05 80101f3f 80101205 801059da 8010581d 80106a11 8010680c
```

Figure 4: Elapsed Time Test

The elapsed time for the `init` process is 0.05 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 0.05 second difference between `init` and `sh` and that leading zeros are printed for the hundredths of a second component of the time.

This sub-test `PASSES`.

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