Home

HW3: System Calls

This homework asks you to first extend the xv6 kernel with several simple system calls.

You will program the xv6 operating system, so you should use the same setup as for the HW2: Xv6 boot.

Part 1: Backtrace system call

In this part of the homework you will add a new system call to the xv6 kernel. The main point of the exercise is for you to see some of the different pieces of the system call machinery.

Your new system call will print the backtrace of the program, i.e., it's invocation history on the console. Specifically, if the user program invokes this new system call the system call prints all registers of the user program, and then walks the stack frame by frame printing the return addresses saved on the stack on the console.

Specifically, your new system call will have the following interface:

```
int backtrace();
```

And when you invoke it from your test program `bt` should print something like this on the console:

```
$ bt
eax:0x16
ebx:0xbfa8
...
esp:0x2f9c
eip:0x369
ebp:0x2fa8
#0  0x4c
```

```
#1  0x6c
#2  0x16
#3  0xffffffff
```

In order to test your system call you should create a user-level program `bt` that calls your new system call. In order to make your new `bt` program available to run from the xv6 shell, look at how other programs are implemented, e.g., `ls` and `wc` and make appropriate modifications to the Makefile such that the new application gets compiled, linked, and added to the xv6 filesystem.

When you're done, you should be able to invoke your `bt` program from the shell. You can follow the following example template for `bt.c`:

```
#include "types.h"
#include "stat.h"
#include "user.h"

int baz() __attribute__((noinline));
int baz() {
   int a;
   a = backtrace();
   return a + uptime();
}

int bar() __attribute__((noinline));
int bar() {
  int b;
  b = baz();
  return b + uptime();
}

int foo() __attribute__((noinline));
int foo() {
  int c;
  c = bar();
  return c + uptime();
}
```

```
int
main(int argc, char *argv[])
{
  foo();
  exit();
}
```

In order to make your new `bt` program available to run from the xv6 shell, add `_bt` to the `UPROGS` definition in `Makefile`.

Your strategy for making the `backtrace` system call should be to clone all of the pieces of code that are specific to some existing system call, for example the "uptime" system call or "read". You should grep for uptime in all the source files, using `grep -n uptime *.[chS]`.

Some hints

It's convenient to implement the body of the `backtrace()` system call in the syscall.c file. Remember that xv6 saves user registers in the trap frame data structure during the system call invocation. Xv6 user programs are compiled with the stack frame, and hence if you review the calling convention lecture, you'll be able to implement the logic of unwinding the stack.

Extra credit (10%): Unwind the stack of the program inside the kernel

Change your `backtrace()` system call to unwind the kernel stack as well.

Extra credit (5%): Maybe you can come up with a way of unwinding the stack of a program that does not maintain the stack frames?

Hint: your implementation does not have to be completely accurate, but should be useful for debugging.

Part 2: ps tool

Implement the `ps` tool that lists all processes running on the system. For that you should implement yet another system call `getprocinfo()` that returns information for a process. Specifically, your new system call will have the following interface:

```
int getprocinfo(int proc_num, struct uproc *up);
```

Where proc_num is the process number in the `ptable.proc[NPROC]` array of all possible processes and `struct uproc` is a structure that describes the process, i.e., contains the following information about the process: process name, process id, parent process id, size of process memory, process state, whether process is waiting on a channel, and whether it's been killed.

You will have to define the `struct uproc` and implement the ps utility by querying the system repeatedly about all possible processes in the system, i.e., all elements of the `ptable.proc[NPROC]` array. You should create a user-level program that calls your new `getprocinfo()` system call.

When you're done, typing `ps` to an xv6 shell prompt should print all processes running in the system and information about them.

Submit

Submit your answers on Canvas HW3 System calls as a compressed tar file of your xv6 source tree (after running make clean). You can use the following command to create a compressed tar file (if you submit extra credit assignments, put a short hw3.txt readme file describing what you've done inside your archive).

```
openlab$ cd xv6-public
openlab$ make clean
openlab$ cd ..
openlab$ tar -czvf hw3.tgz xv6-public
```

Updated: November, 2018