CSE 451: Lab 1 Section Notes
9/28/2017

Implementing File Descriptors

Motivation: We want an abstraction to describe the inputs and outputs of our single process. We should be able to interact with devices, other programs, data files, and any other data streams all the same. We also want a level of indirection between user code and data so we (the kernel) can ensure only valid data is accessed.

Overview: These are the main components you'll be developing and working with for the file portion of Lab 1.

`file.c` -- Should contain all logic related to the file-layer abstraction. Think carefully about what "public" (though only to the kernel) interface you want to provide for this ADT.

`file.h` -- Put your structs here, you're *probably* going to need a file struct.

`sysfile.c` -- We don't trust the arguments that users give to us, so we check input before doing any logic on the file system. Remember to return -1 for error.

`pipe.c` -- Just as "`file.c`" contains the logic corresponding to files, we suggest you similarly separate your pipe logic into its own file.

You'll need a way to keep track of which files are open. There are a number of implementations for this, but we would suggest you use a per-process open file table (put this in `inc/proc.h`).

inode layer - Inodes are the resources managed by the file system. They are not limited to data files, but also represent devices and named pipes (FIFO queues). Read up on what they are, and what they do. We interact with the inode layer through calls to functions in `kernel/fs.c`.

file layer - This is where you live! Design your interface wisely and be clear about its functionality and supporting data structures before you begin coding. (Recall this is a read-only file system.)

syscall layer - Set of kernel functions exposed to *user space*. These functions must ensure valid input from the user, protecting the kernel from errors and mischief. Understand the syscall mechanism. How do syscalls get arguments from the user? What is the difference between a syscall and a hardware-based interrupt?

Suggested Timeline: Ideally, have files finished by next section. We'll talk about creating pipes and how they interact with file descriptors. It's important you start to learn about inodes and files immediately, but we suggest you don't begin coding until you fully understand each concept outlined in the spec (such open file table, file descriptor, inode, syscalls).

Recommended Reading: OSPP, XV6 Book, OSDevWiki, Intel Developer Manuals.

GDB Cheat Sheet

See the GDB manual for a full guide to GDB commands. Here are an assortment of GDB commands your TAs and former 451 students have found to be useful:

Ctrl-c
Halt the machine and break in to GDB at the current instruction. If QEMU has multiple virtual CPUs, this halts all of them.

c (or **continue**)
Continue execution until the next breakpoint or Ctrl-c.

si (or **stepi**)
Execute one machine instruction.

b function or **b file:line** (or **breakpoint**)
Set a breakpoint at the given function or line.

b *_addr_ (or **breakpoint**)
Set a breakpoint at the EIP *addr*.

set print pretty
Enable pretty-printing of arrays and structs.

info registers
Print the general purpose registers, eip, eflags, and the segment selectors. For a much more thorough dump of the machine register state, see QEMU's own info registers command.

x/*N*x *addr*
Display a hex dump of *N* words starting at virtual address *addr*. If *N* is omitted, it defaults to 1. *addr* can be any expression.

x/*N*i *addr*
Display the *N* assembly instructions starting at *addr*. Using $eip as *addr* will display the instructions at the current instruction pointer.

(Source: MIT 6.828/2012)

Another useful GDB Cheat Sheet: http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf

exiting qemu: `ctrl-a x`