CSE 451: Lab 1 Section Notes
1/3/2018

## Implementing File Descriptors

**Note*: See `lab/overview.md` to see an overview of the kernel (this will give you information relevant to this lab and later labs). Also look at the design document, It is meant to guide you through your implementation.

**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. (See `lab1design.md`)

`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 errors on system calls.

**Design document:** This is a design document (you can view it on gitlab to see formatted `md`) we've written to give you an idea of the layouts you should be using. This will be a huge help in guiding how you implement your solution (we *highly* suggest you read it thoroughly and follow it)

**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! Read the design document and be clear about your API's functionality and supporting data structures before you begin coding. (Recall this is a read-only file system, handle accordingly)

**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?

**Recommended Reading:** OSPP (Chp 2)
**Extra Reading:** 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.

**n** (or **next**)
Execute one line of code.

**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.

**tui enable**
Splits your screen with the view of code. Very helpful for seeing where you are in the code while you step.

(Source: MIT 6.828/2012)

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

exiting qemu: ctrl-a x