CSE 451: Lab 1 Section Notes
1/9/2020

Lecture

Debugging:
gdb, printf, kassert statements, unit tests, system/integration tests

Interrupts, system calls, exceptions:
Note: an instruction cannot be interrupted/half executed. A line of C != a CPU instruction.

- see overview.md traps section for how they are handled in osv
- see osdev for more examples of exceptions

Process and threads:

- see overview.md Processes, threads and synchronization

What are the ways to go from user mode to kernel mode? Give an example of each.

How does a process go back to user mode?

Implementing File Descriptors

**Note*: See `labs/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 to allow users to utilize storage on the computer with provisioned access and management. The OS provides *structures* (better abstractions than pure disk sectors) and *protection* (so people can't delete all your work) for storage through a file system. File system functionalities are exposed through system calls.

How should users refer to a file?
Suppose a coworker suggests using the file path to address a file in all syscalls, what are the pros and cons of this decision?
What other options are there?
Modern day systems use file descriptor (a number) to refer to open files. Why file descriptors?

Overview: These are the main components you'll be developing and working with for the file portion of Lab 1. (More details see `labs/lab1design.md`)

`include/kernel/fs.h` -- File system interface you will be using

`include/kernel/proc.h` -- You will need to extend proc to hold open files.

`kernel/syscall.c` -- We don't trust the arguments that users give to us, so we check input before doing any logic on the file system. All arguments fetched are typed *sysarg_t* and every function returns a *sysret_t* as result. These types are used to encapsulate variations in argument size for different architectures. When you pass these arguments to kernel internal functions, you need to cast the arguments to their appropriate types. Remember to return proper error codes(lab1.md) for 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)

File system layer - This is what you will be interacting with! Read the design document and be clear about the file system API's functionality and data structures before you begin coding.

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.

Throwbacks to C:

```
int *x = set_x();
printf("%d\n", *x);

int *set_x(void) {
      int x = 4;
      return &x;
}
```

```
char *buf;
set_buf(&buf);
printf("%s\n", buf);

void set_buf(char **buf) {
     *buf = malloc(sizeof(char) * 6);
     buf[0] = 'h';
     buf[1] = 'e';
     buf[2] = 'l';
     buf[3] = 'l';
     buf[4] = 'o';
     buf[5] = '\0';
}
```

Are the C programs above correct? If not, how would you correct them?

```
struct Point {
      int x;
      int y;
};

struct Point p;
p.x = 1;
p.y = 2;

change_point(p);
print(p.x);
print(p.y);

void change_point(struct Point p) {
      p.x = 10;
      p.y = 20;
}
```

What is printed out above? What needs to be changed to get the desired output?

More C examples:

multiple return values

```
int x,y;
err_t err;
err = fill_xy(&x, &y);
if (err == ERR_OK) {
    printf("%d, %d\n", *x, *y);
}

err_t fill_xy(int *x, int *y) {
      if (x && y) {
          *x = 1;
          *y = 2;
          return ERR_OK;
      }
      return ERR_NOMEM;
}
```

limited stack space
large items should be on heap

```
void parse_string(char* str) {
     char buf1[1024]; // :(
     char *buf2 = malloc(1024); // :)

     strncpy(buf1, str, 1024);
     if (buf2) {
         strncpy(buf2, str, 1024);
     }
}
```

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:

add-symbol-file filepath(ex. build/user/sh)
load symbol table from an executable
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 (i r)
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