Outline

1. Adminstrivia
2. Very brief introduction to operating systems
3. Case study: the shell
4. First lab: Booting a PC
Welcome to OSE!

closely follows the MIT 6.828 “Operating Systems Engineering” course

lecturer: Prof. Dan Tsafrir

TA: Igor Smolyar

HW checker: Marina Minkin

4 credit points & counts as a project.

http://webcourse.cs.technion.ac.il/236376/
What to expect

- (a little) blood, (some) sweat, (hopefully not many) tears
- our goal is to get you to \textit{understand} operating systems
- only way to do that is by building an operating system
- you will design and implement a tiny operating system (at least, the interesting parts)
- sense of accomplishment: \textit{guaranteed}
Structure

- lectures on Tuesdays 12:30 (Igor), on Wednesdays 16:30 (Dan)
- different material covered in each!
- lectures include a mix of:
  - basic OS ideas
  - extended inspection of xv6, a traditional OS
  - some recent topics
- the heart of the course is the homework assignments (labs). You build it, five labs, final project of your choice.
- first lab is out or will be out soon—we’ll talk about it later
- one simple quiz, date TBD
Grading (from the last year)

- labs are worth 30%
- quiz is worth 30%
- final project is worth 40%
- labs and final project are mandatory and build on previous labs
- labs and final project are in pairs
- labs and final project require serious effort... plan ahead!
Outline

1. Adminstrivia
2. Very brief introduction to operating systems
3. Case study: the shell
4. First lab: Booting a PC
What do applications want from an OS?

- abstract the hardware for convenience and portability
- multiplex the hardware among multiple applications
- isolate applications to contain bugs
- allow sharing among applications
How is an OS built?

- examples: OSX, Windows, Linux
- the small view: a hardware management library
- the big view: abstract physical machine into a one that is “better”
- system layers: hardware, kernel, user applications
- in OSE we care a lot about the interfaces and internal kernel structure
what services does an OS kernel typically provide?

- processes
- memory
- file contents
- directories and file names
- security
- many others: users, IPC, network, time, terminals
What does an OS abstraction look like?

- applications only see them via system calls
- examples, from UNIX / Linux:
  - `fd = open("out", 1)`
  - `write(fd, "hello\n", 6)`
  - `pid = fork()`
Why is OS design/implementation hard/interesting?

- the environment is unforgiving: weird hardware, no debugger
- it must be efficient (thus low-level?)
- ... but abstract/portable (thus high-level?)
- powerful (thus many features?)
- ... but simple (thus a few composable building blocks?)
- features interact: \texttt{fd = open(); ... ; fork();}
- behaviors interact: CPU priority vs memory allocator.
- open problems: security, multi-core
You’ll be glad you learned about OS if...

- want to work on the above problems
- care about what’s going on under the hood
- have to build high-performance systems
- need to diagnose bugs or security problems
- want to get a high-paying job at Intel, IBM, Google (not a lot of grads will have the knowledge you’ll gain, and those companies spend effort in seeking grads that do)
- want to do systems research in graduate school
1. Adminstrivia
2. Very brief introduction to operating systems
3. Case study: the shell
4. First lab: Booting a PC
The shell

- the shell is the Unix command user interface
- the shell is also a programming/scripting language
- typically handles login session, runs other processes
- the shell uses different O/S abstractions; let’s see how those abstractions fit together
while (1) {
    write (1, "$ ", 2);
    readcmd (cmd, args); // parse user input
    if ((pid = fork ()) == 0) { // child?
        exec (cmd, args, 0);
    } else if (pid > 0) { // parent?
        wait (0); // wait for child to terminate
    } else {
        perror ("fork");
    }
}
System calls

- system calls: read, write, fork, exec, wait
- what does fork do?
  - copies user memory
  - copies process kernel state (e.g. user id)
  - child gets a different PID
  - child state contains parent PID
  - returns twice, with different values
what does exec do?
- replaces memory of current process with instructions/data from file
  i.e., runs a file created by compiler/linker
- still the same process, keeps most state (e.g. user id)

what does wait do?
- waits for any child to exit
- what if child exits before parent calls wait?

the fork/exec split looks wasteful, but it turns out to be useful.
Alternatives?
example:

```
# ls
```

Q: how does ls know which directory to look at?
A: cwd in kernel-maintained process state, copied during fork
Q: how does it know what to do with its output?
A: I/O: process has file descriptors, numbered starting from 0, which index into table in process’s kernel state

file related system calls: open, read, write, close
numbering conventions:
- file descriptor 0 for input (e.g., keyboard).
- file descriptor 1 for output (e.g., terminal)
- file descriptor 2 for error (e.g., terminal)

so `ls` writes output to file descriptor 1

on fork, child inherits open file descriptors from parent

on exec, process retains file descriptors.
this shell command sends ls’s output to the file out:

```
# ls > out
```

Q: how could our simple shell implement output redirection?
A: just before exec insert:

```
close(1);
creat("out", 0666); // fd will be 1
```

Q: can you think of an alternative?

good illustration of why it’s nice to have separate fork and exec.
Q: system call interface is very simple, just ints and char buffers. why not have open() return a pointer reference to a kernel file object?
Linux has a nice representation of a process and its file descriptors, under /proc/PID/

- maps: VA range, perms (p=private, s=shared), offset, dev, inode, pathname
- fd: symlinks to files pointed to by each fd. (what’s missing in this representation?)
- try:
  
  $ exec 3>/tmp/xx ; ls -l /proc/$$/fd
one often wants to run a series of programs on some data:

```bash
$ sort < in > out
$ uniq < out > out2
$ wc < out2
$ rm out out2
```

the shell supports this more concisely with "piping" of "filters":

```bash
$ sort < in | uniq | wc
```
How does this work?

```c
int fds[2];

if (pipe(fds) < 0) panic ("error");
if ((pid = fork ()) == 0) {  // child (left end of pipe)
    close (1);
    dup (fds[1]);  // fds[1] is the write end, ret will be 1
    close (fds[0]);  // close read end
    close (fds[1]);  // close fds[1]
    exec (command1, args1, 0);
} else if (pid > 0) {  // parent (right end of pipe)
    close (0);
    dup (fds[0]);  // fds[0] is the read end, ret will be 0
    close (fds[0]);
    close (fds[1]);  // close write end
    exec (command2, args2, 0);
} else {
    printf ("Unable to fork\n");
}
```
Notes about file descriptors

- nice interaction with fork
- file descriptors help make programs more general purpose: don’t need special cases for files vs console vs pipe
- shell pipelines only work for programs w/ common formats (lines of text)
Background jobs

- how do you create a background job?
  
  `$ sleep 2 &`

- Q: how does the shell implement "&"?
- Q: what if a background process exits while the shell waits for a foreground process?
- more details in the shell lecture later in the term.
1. Adminstrivia

2. Very brief introduction to operating systems

3. Case study: the shell

4. First lab: Booting a PC
Labs in general

- labs are programming ("wet") assignments
- we provide some code, you need to (a) read, (b) think, and (c) implement the rest
- usually we will provide an automatic tester that will help you test code before you submit it
- environment: Linux, git, QEMU, gcc (C and assembly)
Lab 1: Booting a PC

- get the know the environment: Linux, git, QEMU, gcc
- familiarity with x86 assembly, C, the PC boot process
- boot loader implementation
- initial bring up of the JOS kernel
How to approach a lab

- read it through
- read the references
- experiment
- think...
- implementation: usually short and to the point
- exercises, challenges, and submissions
What we look for in a solution?

- does it work?
- does it pass the tests?
- is it efficient? elegant? understandable?
- “perfection is reached not when there is nothing left to add, but when there is nothing left to take away”