O/S overview

Overview

- 6.828 goals:
  - Understand operating systems in detail by designing and implementing a small O/S
  - Hands-on experience with building systems ("Applying 6.033")
- What do applications want from an O/S?
  - Abstract the hardware for convenience and portability
  - Multiplex the hardware among multiple applications
  - Isolate applications to contain bugs
  - Allow sharing among applications
- What is an OS?
  - e.g. OSX, Windows, Linux
  - the small view: a h/w management library
  - the big view: physical machine -> abstract one w/ better properties
- Organization: layered picture
    - h/w: CPU, mem, disk
    - kernel: [various services]
    - user: applications, e.g. vi and gcc
  - we care a lot about the interfaces and internel kernel structure
- what services does an O/S kernel typically provide?
  - processes
  - memory
  - file contents
  - directories and file names
  - security
  - many others: users, IPC, network, time, terminals
- What does an O/S 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 O/S design/implementation hard/interesting?
  - the environment is unforgiving: weird h/w, 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: `fd = open(); ...; fork();`
  - behaviors interact: CPU priority vs memory allocator.
  - open problems: security, multi-core
- You'll be glad you learned about operating systems if you...
  - 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

Class structure

- http://pdos.lcs.mit.edu/6.828
- Lectures
  - basic O/S ideas
  - extended inspection of xv6, a traditional O/S
  - several more recent topics
  - in-class programming
- Lab: JOS, a small O/S for x86 in an exokernel style
  - you build it, 5 labs, final project of your choice
  - kernel interface: expose hardware, but protect -- no abstractions!
  - unprivileged library: fork, exec, pipe, ...
  - applications: file system, shell, ..
  - development environment: gcc, qemu
  - lab 1 is out
- Code review
- One quiz: in class
- No final, but project and project conferences

Case study: the shell (simplified)

- 6.828 is largely about design and implementation of system call interface. let's start by looking at how programs use that interface. example: the Unix shell.
- the shell is the Unix command UI
- the shell is also a programming/scripting language
- typically handles login session, runs other processes
- look at some simple examples of shell operations, how they use different O/S abstractions, and how those abstractions fit together. See The UNIX Time-Sharing System' by D. M. Ritchie and K. Thompson if you are unfamiliar with the shell.
- Basic structure: see sh.c
- Basic organization: parsing and executing commands (e.g., ls, ls | wc, ls > out)
  Shell implemented using system calls (e.g., read, write, fork, exec, wait) conventions: -1 return value signals
- error, error code stored in `errno`, `perror` prints out a descriptive error message based on `errno`.
- Many systems calls are encapsulated in libc calls (e.g., fgets vs read)
  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 wait() do?
  waits for any child to exit
- what if child exits before parent calls wait?
- What does ~~parsecmd()~~ do?
  Example: parsecmd()

```
$ ls
```

- The parse functions form a simple recursive-descent parser for the sh scripting language
- Do exercise 1 of Shell Exercises for Lecture 1
  You need a variant of exec() (type "man 3 exec"):
  replaces memory of current process with instrs/data from file
  i.e. runs a file created by compiler/linker
- still the same process, keeps most state (e.g. user id)
- the fork/exec split looks wasteful, but it turns out to be useful.
  how does "ls" know which directory to look at?
  cwd in kernel-maintained process state, copied during fork

- how does it know what to do with its output?
- I/O: process has file descriptors, numbered starting from 0.
  index into table in process's kernel state
- system calls: open, read, write, close
- numbering conventions:
  - file descriptor 0 for input (e.g., keyboard). fgets(stdin) invokes:

    ```
    read (0, buf, bufsize)
    ```

  - file descriptor 1 for output (e.g., terminal). fprintf(stdout) invokes:

    ```
    write (1, "hello\n", strlen("hello\n"))
    ```

  - 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 (show in process diagram).
- 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: Do exercise 2.
- Good illustration of why it's nice to have separate fork and exec.
- Many commands use 0/1 by default, so they work with redirection. For example, see xv6 cat.c.
- system call interface 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 FDs, 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:

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

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

  ```
  $ sort < in | uniq | wc
  ```

- A pipe is a one-way communication channel. Here is a simple example:

  ```
  int fds[2];
  char buf[512];
  int n;

  pipe(fds);
  write(fds[1], "hello", 5);
  n = read(fds[0], buf, sizeof(buf));
  // buf[] now contains 'h', 'e', 'l', 'l', 'o'
  ```

- file descriptors are inherited across `fork()`, so this also works:

  ```
  int fds[2];
  char buf[512];
  int n, pid;

  pipe(fds);
  pid = fork();
  if(pid > 0){
    write(fds[1], "hello", 5);
  } else {
    n = read(fds[0], buf, sizeof(buf));
    exit(0);
  }
  ```

- How does the shell implement pipelines (i.e., cmd 1 | cmd 2 |..)? We want to arrange that the output of cmd 1 is the input of cmd 2. The way to achieve this goal is to manipulate stdout and stdin.
- The shell creates processes for each command in the pipeline, hooks up their stdin and stdout, and waits for the last process of the pipeline to exit.
- Do exercise 3.
- Who waits for whom? (draw a tree of processes)
- Why close read-end and write-end? ensure that every process starts with 3 file descriptors, and that reading from the pipe returns end of file after the first command exits.
- You can run the shell, redirect its stdin/stdout, etc.
- I'll run this shell script with `sh < script`:

```
echo one
echo two
```

- Q: what will this shell command do?

```
$ sh < script > out
```

- the script itself didn't redirect the echo output, but it did inherit a fd 1 that was redirected to out.
- Q: I'll run the following directly; is it the same as above?

```
echo one > out
echo two > out
```

- Notes about the file descriptor design:
  - nice interaction with fork
  - FDs 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)
- How do you create a background job?

```
$ sleep 2 &
```

- Q: How does the shell implement "&"?
- Q: What if a background process exits while sh waits for a foreground process?
- Do challenge exercises

6.828 Operating System Engineering

Fall 2012