```
6.828 2017 Lecture 1: O/S overview

Overview

* 6.828 goals
  * Understand operating system design and implementation
  * Hands-on experience by building small O/S

* What is the purpose of an O/S?
  * Support applications
  * Abstract the hardware for convenience and portability
  * Multiplex the hardware among multiple applications
  * Isolate applications in order to contain bugs
  * Allow sharing among applications
  * Provide high performance

* What is the O/S design approach?
  * 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, &c
   kernel services
   user applications: vi, gcc, &c
  * we care a lot about the interfaces and internal kernel structure

* What services does an O/S kernel typically provide?
  * processes
  * memory allocation
  * file contents
  * directories and file names
  * security
  * many others: users, IPC, network, time, terminals

* What does an O/S abstraction look like?
  * Applications see them only via system calls
  * Examples, from UNIX (e.g. Linux, OSX, FreeBSD):

            fd = open("out", 1);
            write(fd, "hello\n", 6);
            pid = fork();

* Why is O/S design/implementation hard/interesting?
  * the environment is unforgiving: quirky h/w, weak 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; performance

* 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

* See web site: https://pdos.csail.mit.edu/6.828

* Lectures
  * O/S ideas
```

```
    * detailed inspection of xv6, a traditional O/S
    * xv6 programming homework to motivate lectures
    * papers on some recent topics

  * Labs: JOS, a small O/S for x86 in an exokernel style
    * you build it, 5 labs + final lab of your choice
    * kernel interface: expose hardware, but protect -- few abstractions!
    * unprivileged user-level library: fork, exec, pipe, ...
    * applications: file system, shell, ..
    * development environment: gcc, qemu
    * lab 1 is out

  * Two exams: midterm during class meeting, final in finals week

  Introduction to system calls

  * 6.828 is largely about design and implementation of system call
  interface. let's look at how programs use that interface.
  we'll focus on UNIX (Linux, Mac, POSIX, &c).

  * a simple example: what system calls does "ls" call?
    * Trace system calls:
      * On OSX: sudo dtruss /bin/ls
      * On Linux: strace /bin/ls
    * so many system calls!

  * example: copy input to output
    cat copy.c
    cc -o copy copy.c
    ./copy
    read a line, then write a line
    note: written in C, the traditional O/S language

    * first read/write argument is a "file descriptor" (fd)
      passed to kernel to tell it what "open file" to read/write
      must previously have been opened, connects to file/device/socket/&c
      UNIX convention: fd 0 is "standard input", 1 is "standard output"

    * sudo dtruss ./copy
      read(0x0, "123\n\0", 0x80)           = 4 0
      write(0x1, "123\n@\213\002\0", 0x4)          = 4 0

  * example: creating a file
    cat open.c
    cc -o open open.c
    ./open
    cat output.txt
    note: creat() turned into open()
    note: can see actual FD with dtruss
    note: this code ignores errors -- don't be this sloppy!

  * example: redirecting standard output
    cat redirect.c
    cc -o redirect redirect.c
    ./redirect
    cat output.txt
    man dup2
    sudo dtruss ./redirect
    note: writes output.txt via fd 1
    note: stderr (standard error) is fd 2 -- that's why creat() yields FD 3

  * a more interesting program: the Unix shell.
    * it's the Unix command-line user interface
    * it's a good illustration of the UNIX system call API
    * some example commands:
```

```
      ls
      ls > junk
      ls | wc -l
      ls | wc -l > junk
    * the shell is also a programming/scripting language
      cat > script
        echo one
        echo two
      sh < script
    * the shell uses system calls to set up redirection, pipes, waiting
      programs like wc are ignorant of input/output setup

  * Let's look at source for a simple shell, sh.c

    * main()
      basic organization: parse into tree, then run
      main process: getcmd, fork, wait
      child process: parsecmd, runcmd
      why the fork()?
        we need a new process for the command
      what does fork() do?
        copies user memory
        copies kernel state e.g. file descriptors
        so "child" is almost identical to "parent"
        child has different "process ID"
        both processes now run, in parallel
        fork returns twice, once in parent, once in child
        fork returns child PID to parent
        fork returns 0 to child
        so sh calls runcmd() in the child process
      why the wait()?
      what if child exits before parent calls wait()?

    * runcmd()
      executes parse tree generated by parsecmd()
      distinct cmd types for simple command, redirection, pipe

    * runcmd() for simple command with arguments
      execvp(cmd, args)
      man execvp
      ls command &c exist as executable files, e.g. /bin/ls
      execvp loads executable file over memory of current process
      jumps to start of executable -- main()
      note: execvp doesn't return if all goes well
      note: execvp() only returns if it can't find the executable file
      note: it's the shell child that's replaced with execvp()
      note: the main shell process is still wait()ing for the child

    * how does runcmd() handle I/O redirection?
      e.g. echo hello > junk
      parsecmd() produces tree with two nodes
        cmd->type='>', cmd->file="junk", cmd->cmd=...
        cmd->type=' ', cmd->argv=["echo", "hello"]
      the open(); dup2() causes FD 1 to be replaced with FD to output file
      it's the shell child process that changes its FD 1
      execvp preserves the FD setup
      so echo runs with FD 1 connected to file junk
      again, very nice that echo is oblivious, just writes FD 1

    * why are fork and exec separate?
      perhaps wasteful that fork copies shell memory, only
        to have it thrown away by exec
      the point: the child gets a chance to change FD setup
        before calling exec
      and the parent's FD set is not disturbed
```

```
      you'll implement tricks to avoid fork() copy cost in the labs

    * how does the shell implement pipelines?
      $ ls | wc -l

    * the kernel provides a pipe abstraction
      int fds[2]
      pipe(fds)
      a pair of file descriptors: a write FD, and a read FD
      data written to the write FD appears on the read FD

    * example: pipe1.c
      read() blocks until data is available
      write() blocks if pipe buffer is full

    * pipe file descriptors are inherited across fork
      so pipes can be used to communicate between processes
      example: pipe2.c
      for many programs, just like file I/O, so pipes work for stdin/stdout

    * for ls | wc -l, shell must:
      - create a pipe
      - fork
      - set up fd 1 to be the pipe write FD
      - exec ls
      - set up wc's fd 0 to be pipe read FD
      - exec wc
      - wait for wc
      [diagram: sh parent, ls child, wc child, stdin/out for each]
      case '|' in sh.c
      note: sh close()es unused FDs
            so exit of writer produces EOF at reader

    * you'll implement pieces of a shell in an upcoming homework
```