```
6.S081 2020 Lecture 1: O/S overview

Overview

* 6.S081 goals
  * Understand operating system (O/S) design and implementation
  * Hands-on experience extending a small O/S
  * Hands-on experience writing systems software

* What is the purpose of an O/S?
  * Abstract the hardware for convenience and portability
  * Multiplex the hardware among many applications
  * Isolate applications in order to contain bugs
  * Allow sharing among cooperating applications
  * Control sharing for security
  * Don't get in the way of high performance
  * Support a wide range of applications

* Organization: layered picture
  [user/kernel diagram]
  - user applications: vi, gcc, DB, &c
  - kernel services
  - h/w: CPU, RAM, disk, net, &c
  * we care a lot about the interfaces and internal kernel structure

* What services does an O/S kernel typically provide?
  * process (a running program)
  * memory allocation
  * file contents
  * file names, directories
  * access control (security)
  * many others: users, IPC, network, time, terminals

* What's the application / kernel interface?
  * "System calls"
  * Examples, in C, from UNIX (e.g. Linux, macOS, FreeBSD):

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

  * These look like function calls but they aren't

* Why is O/S design+implementation hard and interesting?
  * unforgiving environment: quirky h/w, hard to debug
  * many design tensions:
    - efficient vs abstract/portable/general-purpose
    - powerful vs simple interfaces
    - flexible vs secure
  * features interact: `fd = open(); fork()`
  * uses are varied: laptops, smart-phones, cloud, virtual machines, embedded
  * evolving hardware: NVRAM, multi-core, fast networks

* You'll be glad you took this course if you...
  * care about what goes on under the hood
  * like infrastructure
  * need to track down bugs or security problems
  * care about high performance

Class structure

* Online course information:
  https://pdos.csail.mit.edu/6.S081/ -- schedule, assignments, labs
  Piazza -- announcements, discussion, lab help
```

```
* Lectures
  * O/S ideas
  * case study of xv6, a small O/S, via code and xv6 book
  * lab background
  * O/S papers
  * submit a question about each reading, before lecture.

* Labs:
  The point: hands-on experience
  Mostly one week each.
  Three kinds:
    Systems programming (due next week...)
    O/S primitives, e.g. thread switching.
    O/S kernel extensions to xv6, e.g. network.
  Use piazza to ask/answer lab questions.
  Discussion is great, but please do not look at others' solutions!

* Grading:
  70% labs, based on tests (the same tests you run).
  20% lab check-off meetings: we'll ask you about randomly-selected labs.
  10% home-work and class/piazza discussion.
  No exams, no quizzes.
  Note that most of the grade is from labs. Start them early!

Introduction to UNIX system calls

* Applications see the O/S via system calls; that interface will be a big focus.
  let's start by looking at how programs use system calls.
  you'll use these system calls in the first lab.
  and extend and improve them in subsequent labs.

* I'll show some examples, and run them on xv6.
  xv6 has similar structure to UNIX systems such as Linux.
  but much simpler -- you'll be able to digest all of xv6
    accompanying book explains how xv6 works, and why
  why UNIX?
    open source, well documented, clean design, widely used
    studying xv6 will help if you ever need to look inside Linux
  xv6 has two roles in 6.S081:
    example of core functions: virtual memory, multi-core, interrupts, &c
    starting point for most of the labs
  xv6 runs on RISC-V, as in current 6.004
  you'll run xv6 under the qemu machine emulator

* example: copy.c, copy input to output
  read bytes from input, write them to the output
  $ copy
  copy.c is written in C
    Kernighan and Ritchie (K&R) book is good for learning C
  you can find these example programs via the schedule on the web site
  read() and write() are system calls
  first read()/write() argument is a "file descriptor" (fd)
    passed to kernel to tell it which "open file" to read/write
    must previously have been opened
    an FD connects to a file/device/socket/&c
    a process can open many files, have many FDs
    UNIX convention: fd 0 is "standard input", 1 is "standard output"
  second read() argument is a pointer to some memory into which to read
  third argument is the maximum number of bytes to read
    read() may read less, but not more
  return value: number of bytes actually read, or -1 for error
  note: copy.c does not care about the format of the data
    UNIX I/O is 8-bit bytes
    interpretation is application-specific, e.g. database records, C source, &c
```

```
    where do file descriptors come from?

  * example: open.c, create a file
    $ open
    $ cat output.txt
    open() creates a file, returns a file descriptor (or -1 for error)
    FD is a small integer
    FD indexes into a per-process table maintained by kernel
    [user/kernel diagram]
    different processes have different FD name-spaces
      i.e. FD 1 often means different things to different processes
    these examples ignore errors -- don't be this sloppy!
    Figure 1.2 in the xv6 book lists system call arguments/return
      or look at UNIX man pages, e.g. "man 2 open"

  * what happens when a program calls a system call like open()?
    looks like a function call, but it's actually a special instruction
    hardware saves some user registers
    hardware increases privilege level
    hardware jumps to a known "entry point" in the kernel
    now running C code in the kernel
    kernel calls system call implementation
      open() looks up name in file system
      it might wait for the disk
      it updates kernel data structures (cache, FD table)
    restore user registers
    reduce privilege level
    jump back to calling point in the program, which resumes
    we'll see more detail later in the course

  * I've been typing to UNIX's command-line interface, the shell.
    the shell prints the "$" prompts.
    the shell lets you run UNIX command-line utilities
      useful for system management, messing with files, development, scripting
      $ ls
      $ ls > out
      $ grep x < out
    UNIX supports other styles of interaction too
      window systems, GUIs, servers, routers, &c.
    but time-sharing via the shell was the original focus of UNIX.
    we can exercise many system calls via the shell.

  * example: fork.c, create a new process
    the shell creates a new process for each command you type, e.g. for
      $ echo hello
    the fork() system call creates a new process
      $ fork
    the kernel makes a copy of the calling process
      instructions, data, registers, file descriptors, current directory
      "parent" and "child" processes
    only difference: fork() returns a pid in parent, 0 in child
    a pid (process ID) is an integer, kernel gives each process a different pid
    thus:
      fork.c's "fork() returned" executes in *both* processes
      the "if(pid == 0)" allows code to distinguish
    ok, fork lets us create a new process
      how can we run a program in that process?

  * example: exec.c, replace calling process with an executable file
    how does the shell run a program, e.g.
      $ echo a b c
    a program is stored in a file: instructions and initial memory
      created by the compiler and linker
    so there's a file called echo, containing instructions
    $ exec
```

```
    exec() replaces current process with an executable file
      discards instruction and data memory
      loads instructions and memory from the file
      preserves file descriptors
    exec(filename, argument-array)
      argument-array holds command-line arguments; exec passes to main()
      cat user/echo.c
      echo.c shows how a program looks at its command-line arguments

  * example: forkexec.c, fork() a new process, exec() a program
    $ forkexec
    forkexec.c contains a common UNIX idiom:
      fork() a child process
      exec() a command in the child
      parent wait()s for child to finish
    the shell does fork/exec/wait for every command you type
      after wait(), the shell prints the next prompt
      to run in the background -- & -- the shell skips the wait()
    exit(status) -> wait(&status)
      status convention: 0 = success, 1 = command encountered an error
    note: the fork() copies, but exec() discards the copied memory
      this may seem wasteful
      you'll transparently eliminate the copy in the "copy-on-write" lab

  * example: redirect.c, redirect the output of a command
    what does the shell do for this?
      $ echo hello > out
    answer: fork, change FD 1 in child, exec echo
    $ redirect
    $ cat output.txt
    note: open() always chooses lowest unused FD; 1 due to close(1).
    fork, FDs, and exec interact nicely to implement I/O redirection
      separate fork-then-exec give child a chance to change FDs before exec
      FDs provide indirection
        commands just use FDs 0 and 1, don't have to know where they go
      exec preserves the FDs that sh set up
    thus: only sh has to know about I/O redirection, not each program

  * It's worth asking "why" about design decisions:
    Why these I/O and process abstractions? Why not something else?
    Why provide a file system? Why not let programs use the disk their own way?
    Why FDs? Why not pass a filename to write()?
    Why are files streams of bytes, not disk blocks or formatted records?
    Why not combine fork() and exec()?
    The UNIX design works well, but we will see other designs!

  * example: pipe1.c, communicate through a pipe
    how does the shell implement
      $ ls | grep x
    $ pipe1
    an FD can refer to a "pipe", as well as a file
    the pipe() system call creates two FDs
      read from the first FD
      write to the second FD
    the kernel maintains a buffer for each pipe
      [u/k diagram]
      write() appends to the buffer
      read() waits until there is data

  * example: pipe2.c, communicate between processes
    pipes combine well with fork() to implement ls | grep x
      shell creates a pipe,
      then forks (twice),
      then connects ls's FD 1 to pipe's write FD,
      and grep's FD 0 to the pipe
```

```
    [diagram]
  $ pipe2 -- a simplified version
  pipes are a separate abstraction, but combine well w/ fork()

* example: list.c, list files in a directory
  how does ls get a list of the files in a directory?
  you can open a directory and read it -> file names
  "." is a pseudo-name for a process's current directory
  see ls.c for more details

* Summary
  * We've looked at UNIX's I/O, file system, and process abstractions.
  * The interfaces are simple -- just integers and I/O buffers.
  * The abstractions combine well, e.g. for I/O redirection.

You'll use these system calls in the first lab, due next week.
```