# 6.s081
# Intro to OS
# Lecture Notes

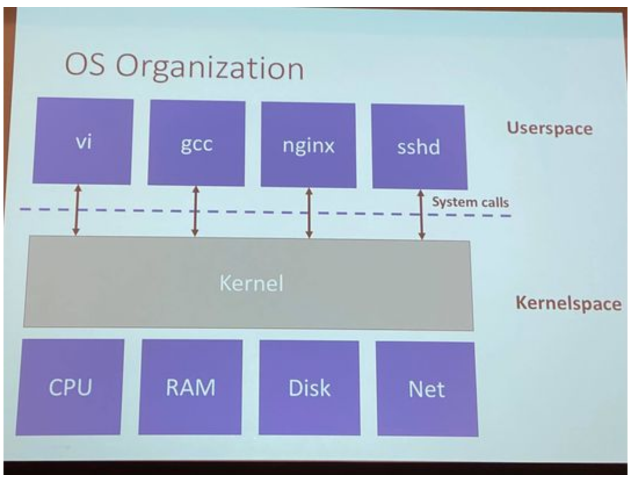
Kevin Yang

9/8/21-?/?/??

# 1 Lecture 1

## 1.1 What is the purpose of an OS?

1. Abstraction
   - Hides hardware details for portability and convenience
   - Must not get in the way of high performance
   - Must support a wide range of applications
2. Multiplexing
   - Allows multiple application to share hardware
   - Isolation to contain bugs and provide security
   - Sharing to allow cooperation




## 1.2 OS abstractions

- Process (a running program)
  - Instructions
  - Memory Storage/Allocation
- Memory allocation

- File descriptor
- File names and directories
  - Namespaces
- Access control and quotas
- Many others: users, IPC, network sockets, time, etc.

### 1.3 User $\leftrightarrow$ Kernel Interface

- Primarily system calls
- Examples:
  - fd = open("out", 1)
  - len = write(fd, "hello how\n",6)
  - pid = fork();
- Look and behave like function calls but they aren't. They are switching between user and kernel space and directly call things in the hardware

### 1.4 Why OSes are interesting

- Unforgiving to build: debugging is hard
- Design tensions:
  - Efficiency sv Portability/Generality
  - Powerful vs Simple
  - Flexible vs Secure
- Challenge: good orthogonality, feature interactions
- Varied uses from smartbulbs to supercomputers
- Evolving HW: NVRAM, multcore, 200Gbit networks

### 1.5 Take this course if you:

- What to understand how computers reallyh work from an engineering perspective
- Want to build future system infrastructure
- Want to solve bugs and security problems
- Care about performance

### 1.6 Logistics

- Course Website
  - https://pdos.csail.mit.edu/6.s081
  - Schedule, course policies, lab assignments, etc
  - Videos and notes of 2020 lectures
- Piazza
  - https://piazza.com/mit/fall2021/6s081
  - Announcements and discussions
  - Ask questions about labs and lecture

## 1.7 Lectures

1. OS concepts
2. Case studies of xv6 — a small simple OS
3. Lab background and solutions
4. OS papers

- Submit a question before each lecture
- Resource: x6 book

## 1.8 Labs

- Goal: Hands-on experience
- Three types of labs:
  1. Systems programming: due next wek
  2. OS primitives: e.g. thread scheduler
  3. OS extensions: e.g. networking driver

## 1.9 Collaboration

- Feel free to ask and discuss questions about lab assignments in class or Piazza
- Discussion is great
  - But all solutions(code and written work) must be your own
  - Acknowledge ideas from others
- Do not post solutions on Github etc

## 1.10 Covid-19 and in-person learning

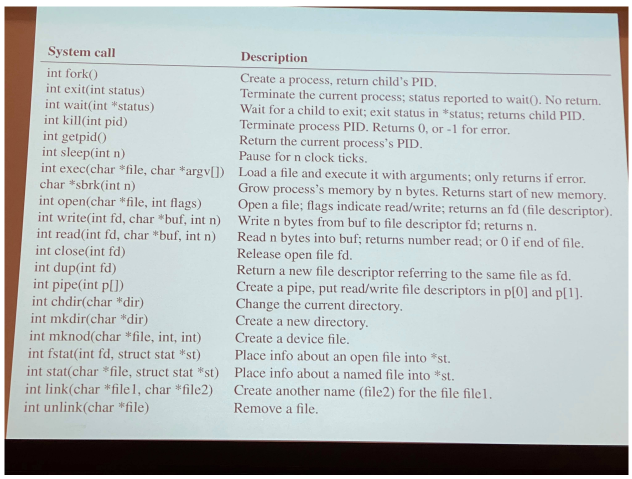
- Masks are **required**; must be worn correctly
- If you have symptoms or test positive...
  - Don't attend class, contact us right away
  - We will work with you to provide course materials

## 1.11 Grading

- 70% labs, based on the same tests you will run
- 20% lab checkoff meetings
  - We will ask questions about randomly selected labs during office hours
- 10% homework and class/piazza participation

## 1.12 Back to system calls

- Will use xv6, the same OS you'll build labs on
- xv6 is similar to UNIX or Linux, but way simpler
  - Why? So you can understand the entire thin
- Why UNIX?
  1. Clean design, widely used: Linux, OSx, Windows(mostly)
- xv6 runs on Risc-V, like 6.004
- You will use Qemu to run xv6 (emulation)



| System call | Description |
|---|---|
| int fork() | Create a process, return child's PID. |
| int exit(int status) | Terminate the current process; status reported to wait(). No return. |
| int wait(int *status) | Wait for a child to exit; exit status in *status; returns child PID. |
| int kill(int pid) | Terminate process PID. Returns 0, or -1 for error. |
| int getpid() | Return the current process's PID. |
| int sleep(int n) | Pause for n clock ticks. |
| int exec(char *file, char *argv[]) | Load a file and execute it with arguments; only returns if error. |
| char *sbrk(int n) | Grow process's memory by n bytes. Returns start of new memory. |
| int open(char *file, int flags) | Open a file; flags indicate read/write; returns an fd (file descriptor). |
| int write(int fd, char *buf, int n) | Write n bytes from buf to file descriptor fd; returns n. |
| int read(int fd, char *buf, int n) | Read n bytes into buf; returns number read; or 0 if end of file. |
| int close(int fd) | Release open file fd. |
| int dup(int fd) | Return a new file descriptor referring to the same file as fd. |
| int pipe(int p[]) | Create a pipe, put read/write file descriptors in p[0] and p[1]. |
| int chdir(char *dir) | Change the current directory. |
| int mkdir(char *dir) | Create a new directory. |
| int mknod(char *file, int, int) | Create a device file. |
| int fstat(int fd, struct stat *st) | Place info about an open file into *st. |
| int stat(char *file, struct stat *st) | Place info about a named file into *st. |
| int link(char *file1, char *file2) | Create another name (file2) for the file file1. |
| int unlink(char *file) | Remove a file. |

In UNIX, for std: Use `make qemu` to run xv6 emulation. `-smp` tag controls number of multiprocessors.

0 input

1 output

2 errors

`read` loads a keyboard text buffer in the kernel space which is then sent into the user space's program when enter is pressed. A program like `copy` will then write to the kernel using the user's input.

`open` will open a file based on the path provided. It takes flags such as `O_WRONLY` or `O_CREATE`. `write` is used to write to a certain file by sending in a string and the number of chars in the string.

The shell like a very simple programming language that helps you chain together other instructions and programs using things like pipes and other commands. Command shells are bash, etc.

`fork` creates a completely identical process with copied over memory and instructions. It uses the return code `pid`, a unique number (process identifier), to differentiate between the parent and the child. It is a single system call that is called once but is returned twice. If the `pid` is 0 then it is a child. Can cause a race condition since both processes output to the same console.

`exec` tells the kernel to run another program/instruction by loading another binary code into the console. This replaces the existing program binary code so a new fork is able to run something new.
The program runs `wait` really fast when `exec` is called on the child so there are much less race issues. It provides a status of whether the process succeeded or failed. exec jumps to a new instruction and clears away everything else in the forked program.

The `exit` system call takes the child status and delivers it to the parent as it waits. 0 is a success and 1 is a failure. If you have multiple forks that execs, it will return the first status return rather than use unique pids.

`fds[2]` is used to set up two of file descriptors. These FDs are used in `pipe` that is used to read/write from/to. The text written into the pipe is stored in a buffer that the kernel maintains for each pipe. Using pids, you are able to use pipes to communicate between two processes.