6.S081: Introduction

Adam Belay
abelay@mit.edu
6.S081 Objectives

• Understand how OSes are designed and implemented
• Hands-on experience building systems software
• Will extend a simple OS (xv6)
• Will learn about how hardware works (Risc-V)
Some things you’ll do in 6.S081

1. You will build a driver for a network stack that sends packets over the real Internet
2. You will redesign a memory allocator so that it can scale across multiple cores
3. You will implement fork and make it efficient through an optimization called copy-on-write
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 applications to share hardware
   - Isolation to contain bugs and provide security
   - Sharing to allow cooperation
OS Organization

Kernel

Userspace

vi
gcc
nginx
sshd

Kernelspace

CPU
RAM
Disk
Net

System calls
OS abstractions

- Process (a running program)
- Memory allocation
- File descriptors
- File names and directories
- Access control and quotas
- Many others: users, IPC, network sockets, time, etc.
User <-> kernel interface

• Primarily system calls
• Examples:
  
  ```
  fd = open("out", 1);
  len = write(fd, "hello\n", 6);
  pid = fork();
  ```
• Look and behave like function calls, but they aren’t
Why OSes are interesting

• Unforgiving to build: Debugging is hard, a single bug can take down the entire machine

• Design tensions:
  • Efficiency vs. Portability/Generality
  • Powerful vs. Simple
  • Flexible vs. Secure

• Challenge: good orthogonality, feature interactions

• Varied uses from smartbulbs to supercomputers

• Evolving HW: NVRAM, Multicore, 200Gbit networks
Take this course if you:

• Want to understand how computers really work from an engineering perspective
• Want to build future system infrastructure
• Want to solve bugs and security problems
• Care about performance
Logistics
Online resources

• Course website
  • Schedule, course policies, lab assignments, etc.
  • Videos and notes of 2020 lectures

• Piazza
  • https://piazza.com/mit/fall2021/6s081
  • Announcements and discussion
  • Ask questions about labs and lecture
Lectures

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

• Submit a question before each lecture
• Resource: xv6 book
Labs

• Goal: Hands-on experience
• Three types of labs:
  1. Systems programming: due next week
  2. OS primitives: e.g., thread scheduling
  3. OS extensions: e.g., networking driver
Collaboration

• Feel free to ask and discussion questions about lab assignments in class or on Piazza

• Discussion is great
  • But all solutions (code and written work) must be your own
  • Acknowledge ideas from others (e.g., classmates, open source software, stackoverflow, etc.)

• Do not post your solutions (including on github)
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
Grading

• 70% labs, based on the same tests you will run
• 20% lab check off meetings
  • We will ask questions about randomly selected labs during office hours
• 10% homework and class/piazza participation
Back to system calls

• I’ll show examples of using 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 thing.
• Why UNIX?
  • Clean design, widely used: Linux, OSx, Windows (mostly)
• xv6 runs on Risc-V, like 6.004
• You will use Qemu to run xv6 (emulation)
<table>
<thead>
<tr>
<th>System call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int fork()</td>
<td>Create a process, return child’s PID.</td>
</tr>
<tr>
<td>int exit(int status)</td>
<td>Terminate the current process; status reported to wait(). No return.</td>
</tr>
<tr>
<td>int wait(int *status)</td>
<td>Wait for a child to exit; exit status in *status; returns child PID.</td>
</tr>
<tr>
<td>int kill(int pid)</td>
<td>Terminate process PID. Returns 0, or -1 for error.</td>
</tr>
<tr>
<td>int getpid()</td>
<td>Return the process’s PID.</td>
</tr>
<tr>
<td>int sleep(int n)</td>
<td>Pause for n clock ticks.</td>
</tr>
<tr>
<td>int exec(char *file, char *argv[])</td>
<td>Load a file and execute it with arguments; only returns if error.</td>
</tr>
<tr>
<td>char *sbrk(int n)</td>
<td>Grow process’s memory by n bytes. Returns start of new memory.</td>
</tr>
<tr>
<td>int open(char *file, int flags)</td>
<td>Open a file; flags indicate read/write; returns an fd (file descriptor).</td>
</tr>
<tr>
<td>int write(int fd, char *buf, int n)</td>
<td>Write n bytes from buf to file descriptor fd; returns n.</td>
</tr>
<tr>
<td>int read(int fd, char *buf, int n)</td>
<td>Read n bytes into buf; returns number read; or 0 if end of file.</td>
</tr>
<tr>
<td>int close(int fd)</td>
<td>Release open file fd.</td>
</tr>
<tr>
<td>int dup(int fd)</td>
<td>Return a new file descriptor referring to the same file as fd.</td>
</tr>
<tr>
<td>int pipe(int p[])</td>
<td>Create a pipe, put read/write file descriptors in p[0] and p[1].</td>
</tr>
<tr>
<td>int chdir(char *dir)</td>
<td>Change the current directory.</td>
</tr>
<tr>
<td>int mkdir(char *dir)</td>
<td>Create a new directory.</td>
</tr>
<tr>
<td>int mknode(char *dir)</td>
<td>Create a device file.</td>
</tr>
<tr>
<td>int fstat(int fd, struct stat *st)</td>
<td>Place info about an open file into *st.</td>
</tr>
<tr>
<td>int stat(char *file, struct stat *st)</td>
<td>Place info about a named file into *st.</td>
</tr>
<tr>
<td>int link(char *file1, char *file2)</td>
<td>Create another name (file2) for the file file1.</td>
</tr>
<tr>
<td>int unlink(char *file)</td>
<td>Remove a file.</td>
</tr>
</tbody>
</table>