System Calls and Using the OS

Welcome back colloquium on Friday!
Shell (programming) due on Monday!

Outline

- What is an Operating System
- Introducing Systems Calls
- OS Kernel API and ABI
- System Call Table and Errno
- Implementing System Calls

(Formally) Defining an Operating System

- An *operating system* is the software that's between the application software layer and hardware
  - This means the OS *kernel* (an overloaded term, but generally referring to the privileged parts of the OS) must manage the processors, memory, disk, and I/O devices (mouse, keyboard, display, etc…)
- An operating system's role can be thought of as facilitating three primary goals between application software: (1) the referee, (2) the illusionist, and (3) the glue

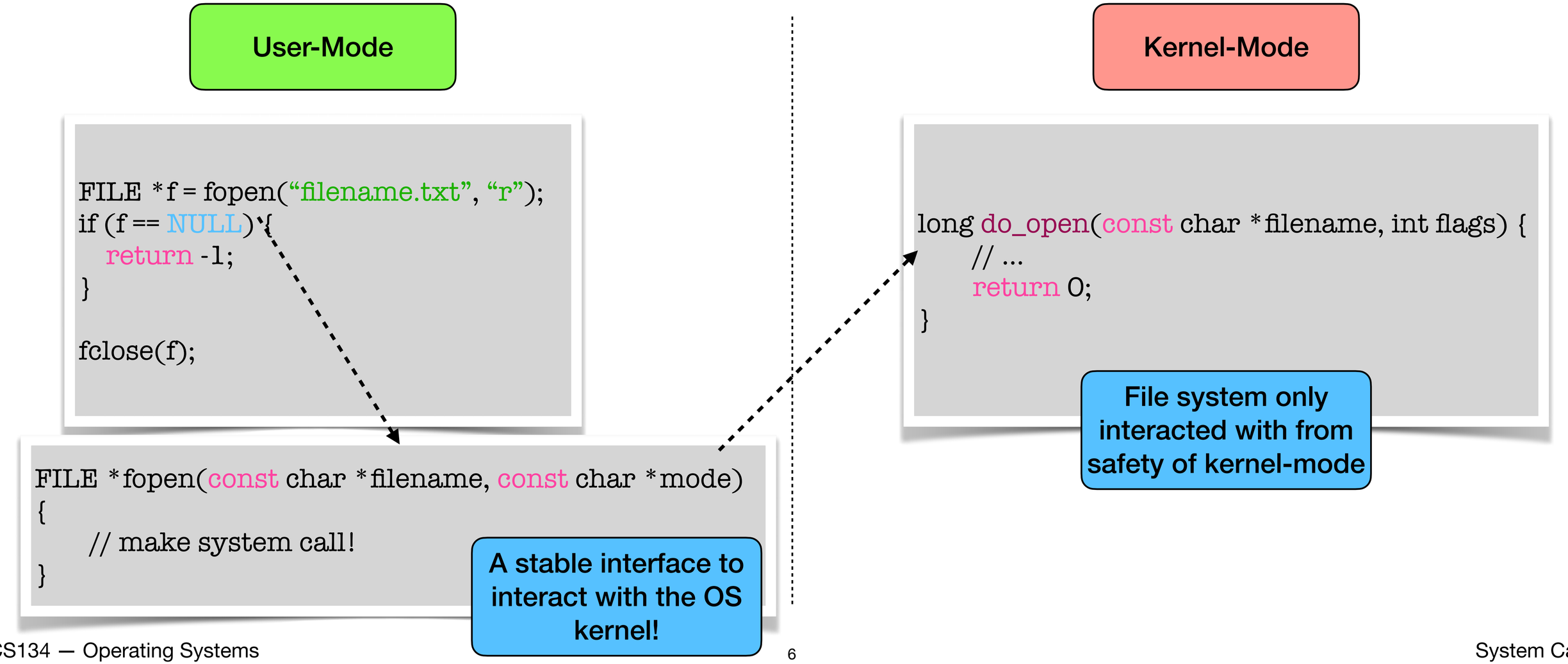
Demo!

You can access the demo at https://cs.pomona.edu/class/cs134/demos/lec02.tar

Chat with your neighbor(s)!

When thinking about how the OS kernel implements a system call, what are some of the desirable properties for the user and for the system? It may help to think of the OS in terms of its roles as a referee, illusionist, and glue.

The System Call Interface



OS Kernel API

- Application Program Interface (API): describes the names and arguments of procedures, functions, methods, etc… that application programs call to access the functionality of a system
  - In general, the interface that applications use to access the OS kernel is to make a *system call* (e.g., `open`, `read`, `write`, `close`, etc…)
  - Eventually, system calls become wrapped by Standard Library calls to have a clean interface when interacting with the OS kernel
- In Linux, the API is intended to be: (1) platform-agnostic, (2) highly stable, (3) follows well defined table

OS Kernel ABI

Nowhere in hardware does it say that function parameters need to be in certain registers! Nowhere in hardware does it say that data is big endian or little endian! These are phenomena defined in the Operating System ABI!

- Application Binary Interface (ABI): describes the machine language instructions necessary for accessing a systems functionality
  - Which registers are used for which function argument is a feature defined in the OS kernel ABI

	SysV x64 (64-bit x86 Linux)	Win x64 (64-bit x86 Windows)	SysV x32 (32-bit x86 Linux)
arg 0	%rdi	%rcx	stack
arg 1	%rsi	%rdx	stack
arg 2	%rdx	%r8	stack
arg 3	%rcx	%r9	stack
arg 4	%r8	stack	stack
arg 5	%r9	stack	stack

  - Other ABI tasks include system call interface, binary format specification, library linking rules, virtual file system interfaces, data representations
- Linux proposes a *highly stable* ABI so that legacy binaries are runnable on modern platforms

System Call Table

- Each system call has a *system call number* that is used to indicate to the operating system which call to execute
- When making a system call, different ABIs define different conventions to set the state before switching to kernel mode
- Different system calls interact with various parts of the operating system!

ChromiumOS System Call Table

Chat with your neighbor(s)!

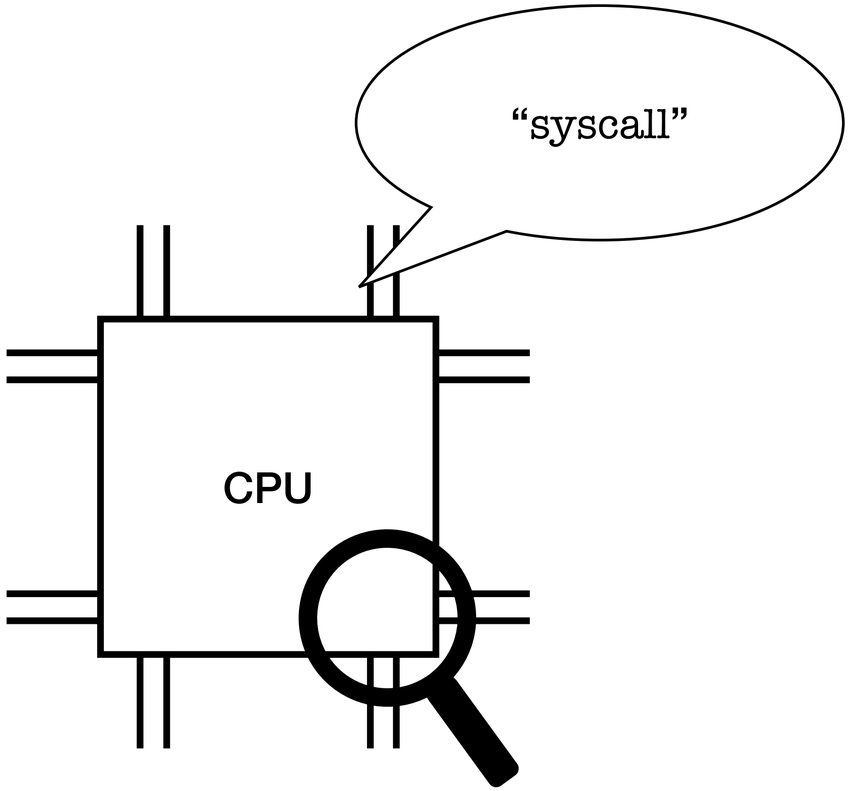
Consider the disassembled libc code around the “kill” function call. How is the OS kernel being invoked? What is the register/stack state in advance of the OS kernel invocation?

```
00000000000235a0 <kill>:
  235a0:    f3 0f 1e fa              endbr64
  235a4:    b8 3e 00 00 00           mov    $0x3e,%eax
  235a9:    0f 05                    syscall
  235ab:    48 3d 01 f0 ff ff        cmp    $0xfffffffffffff001,%rax
  235b1:    73 01                    jae    235b4 <kill+0x14>
  235b3:    c3                       retq
  235b4:    48 8b 0d ad c8 3a 00     mov    0x3ac8ad(%rip),%rcx
  235bb:    f7 d8                    neg    %eax
  235bd:    64 89 01                 mov    %eax,%fs:(%rcx)
  235c0:    48 83 c8 ff              or     $0xffffffffffffffff,%rax
  235c4:    c3                       retq
```

“syscall” Instruction in x86

- The `syscall` instruction is the primary mechanism to transition from user-mode to kernel-mode using the Linux x64 ABI
  - To do so, this instruction performs a “trap” ➡️ a mechanism that elicits an operating system response
  - The hardware sets the instruction pointer to a location defined by the *Interrupt Descriptor Table*, which has entries for all types of interrupts
  - The interrupt handler for the system call looks at the `%rax` register for which routine to perform (e.g., `do_open`)

“syscall” Instruction in x86



Interrupt	Handler
0	System traps and exceptions
1	…
…	…
32-127	Device interrupt handlers
128	**syscall interface**
…	…
255	…

1. Save the information needed to return from the interrupt onto the stack
2. Execute kernel interrupt handler (in this case, the syscall interface)
   - In practice, this will be a table of functions to potentially execute depending on the value of `%rax`
3. Return from the interrupt handler

Takeaways

- When writing C programs, tion system by making *system calls* (the OS kernel
- APIs and ABIs describe *int* isms to execute OS-specific tasks
- To implement system calls, system need to coordinate the facilitation of moving fr
  - The program does this b ➡️ this performs a *trap* which cause the hardwa errupt handler
  - The interrupt handler is i ystem and executes code in kernel-mode