

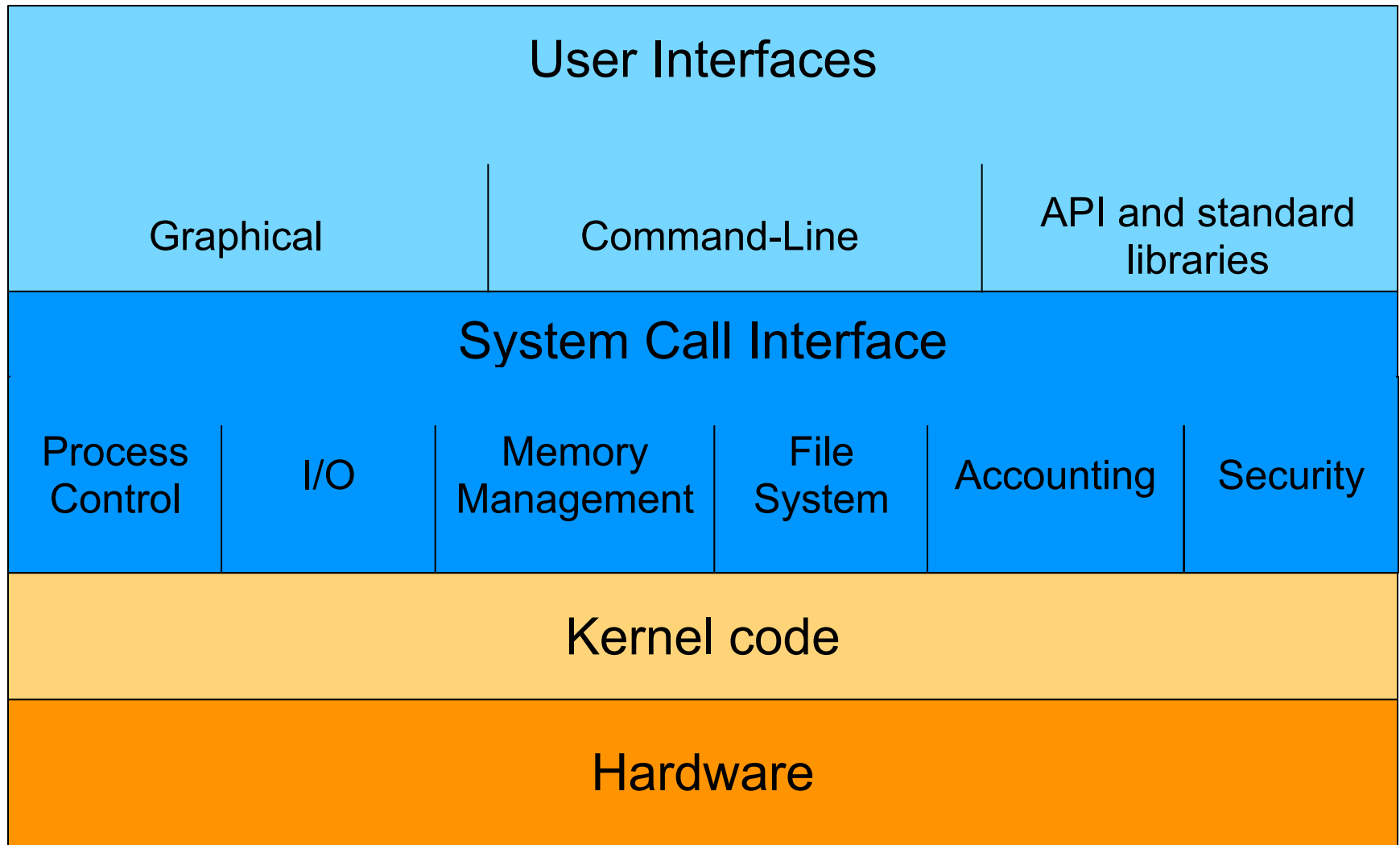


Operating System Interfaces

**ICS332
Operating Systems**

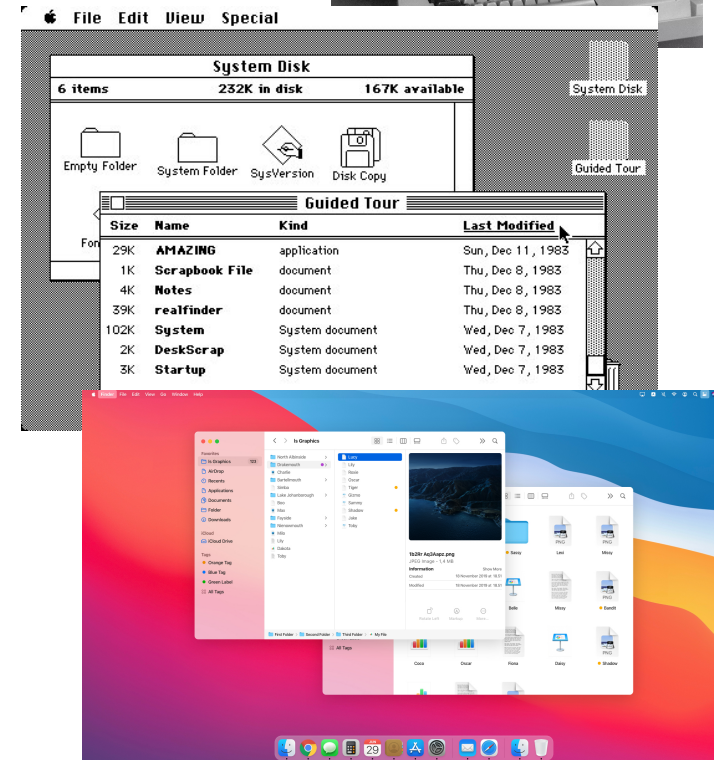
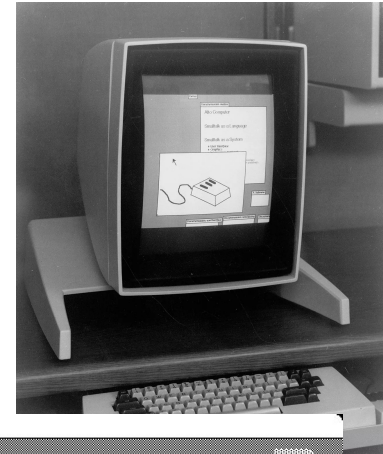
Henri Casanova (henric@hawaii.edu)

OS Interfaces



Graphical User Interfaces

- Early 1970s (Xerox PARC research)
- Popularized by Apple's Macintosh (1980s)
- Many UNIX users still use the command-line heavily, while Windows users usually prefer the GUI
- Mac OS < 10: no CLI, but Mac OS ≥ 10 does: Terminal
- Question: Is the GUI part of the OS or not?
 - Windows: YES
 - MacOS: YES
 - Linux: NO



Command-Line Interfaces (CLI)

- Also known as the Shell
- Provides many built-in **commands**
 - On my Mac: `man builtin`
- The Shell is often used to invoke low-level **system programs**
 - On UNIX-like systems, often brief one-word executables (`ls`, `ps`, `sed`, `grep`, ...)
 - Not part of the OS, but almost always installed with it
- It is often used to invoke **user programs**
- The distinction between system and user programs is vague at best and really not useful because it's a matter of perspective
 - What about `ls`? If you're a kernel developer, then it's a very high-level application. If you're a novice Linux user, then you probably think of it as some "OS thing"...

The System Call (Syscall) API

- System calls, or **syscalls**, provide the lowest-level interface to the OS
- GUIs and CLIs (and in fact all programs!) are built on top of the System Call API
- Often programs will use some library, that uses some library, that uses some “standard” library, that uses the system call API
 - It all boils down to system calls (unless your program does nothing but computation without anything else, which is never useful)
- You can think of your running program as doing one of two things:
 - Either fetch-decode-execute instructions that you wrote or that are in the libraries that you use
 - Or fetch-decode-execute instructions that are in the kernel because your program invoked a syscall
- We will see uses of the syscall API (or low-level standard libraries that use it)
- Interestingly, you can spy on system call usage...

Spying on System Calls

- There are tools to “spy on processes” and see what system calls they place as they happen!
 - **strace** in Linux
 - **dtruss** in macOS
 - **ProcMon** in Windows
- Why is this useful?
 - Find bugs, find performance bugs, detect malware, reverse-engineer code, and learning :)
- Let's look at **strace** in Linux...

strace Example Uses

-i option: shows the value of the Program Counter

```
strace -i sleep 1
```

-x option: shows non-ASCII characters in hex

```
strace -x touch /tmp/foo
```

-c option: obtain cumulative statistics

```
mkdir tmp; cd tmp
for a in `seq 0 9`; do
  for b in `seq 0 9`; do
    touch $a$b;
  done
done
strace -c rm *
```

-p option: attach to a running process (may require sudo)

```
strace -p <pid of process> # Let's demonstrate with sshd
```

strace in-class Demo (if time)

- One can do all kinds of things with **strace**
- Here is one that we could live-code right now
 - I have a machine on which I've many users with accounts, and I am a superuser
 - I am a malicious person, and I want to steal my user's passwords on other machines!
 - My users all use **ssh** from my machine to connect to those other machines
 - So I replace the **ssh** executable on my machine with a fake script that uses **strace** to steal passwords while invoking **ssh**
- Off we go....
 - Reminders for me as sometimes forget these
 - **\$@** : all the arguments passed to the original script
 - **paste -sd ""**: concatenates multiple lines into one

Syscall Table

- There are many syscalls in a typical OS (~300-400 in Linux)
- Each system call is identified by a unique number, stored in an internal table called the **syscall table**
- Let's look at the [ChromiumOS syscall table](#)
 - Linux kernel, open-source version of ChromeOS, developed by Google (support stopped in 2022...ChromeBooks weren't a commercial success)
- On the **henricasanova/ics332** Docker image, we can find the syscall table in: **/usr/src/linux-source-6.8.0/linux-source-6.8.0/arch/x86/entry/syscalls**
- There are syscalls for everything that you'd expect (to manage processes, memory, files, devices, communication, permissions, etc.)
- **Syscalls are the only way to access hardware resources virtualized by the OS**

Timing Programs and System Calls

- The UNIX `time` command can be used to see what time a program spends running user code and what time it spends running kernel code (i.e., system calls)
 - Does not have a great resolution, so results can be weird when timing lightning quick programs
- It reports:
 - **Real time:** The time you experience (also called wall-clock time, elapsed time, execution time, run time...)
 - **User time:** The time spent executing user code
 - **System time:** The time spent executing kernel code



Measuring System Time

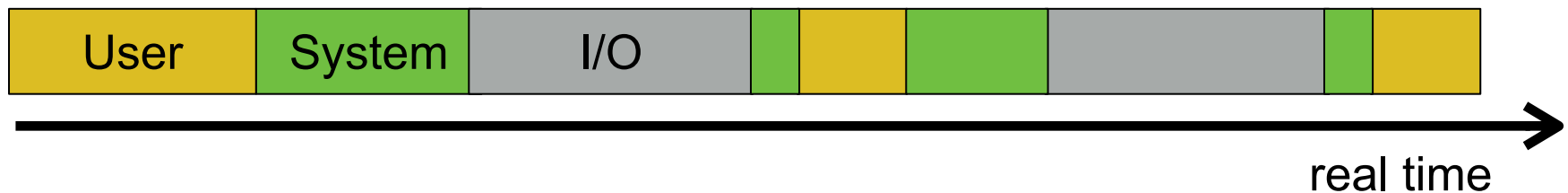
- Lets use the time command for
 - Archiving/Compressing some directory
 - Running `du` on a large and deep directory
 - Running `jekey11`

Measuring System Time

- Lets use the time command for
 - Archiving/Compressing some directory
 - Running `du` on a large and deep directory
 - Running `jekey11`
- We observe: real time \neq user time + system time
- What's the missing time?

Measuring System Time

- Lets use the time command for
 - Archiving/Compressing some directory
 - Running `du` on a large and deep directory
 - Running `jekey11`
- We observe: real time \neq user time + system time
- What's the missing time? **I/O!**



- I/O time could be waiting for the disk, network, keyboard, etc.
- $\text{real time} = \text{user time} + \text{system time} + \text{i/o time}$

System Calls are Expensive

- The OS tries to be fast
 - Kernel developers are good at writing lean/mean code
- Nevertheless, system calls can be expensive
 - Especially when they involve some hardware overhead (i.e., waiting for the disk)
- As a programmer you should use system calls wisely (if you care about speed at all)
- This can fly in the face of what you learn in the CS curriculum
- Well-known example
 - ICS111/211: Data structures are great, so use them
 - BUT, your code may end up calling malloc/free all the time!
 - So then you want to use arrays
 - But then everything's ugly/cumbersome because an array is such a weak/rigid data structure
- The life of the developer is about making difficult compromises

The System Call API

- System calls can be complicated to place
- Therefore, there is a **system call interface**, i.e., a set of useful functions, often provided in standard libraries, that are “easier-to-use wrappers” around the raw system calls e.g., the **fork()** “system call” is a simple interface to the **clone()** system call
 - e.g., When in C you open a file with **fopen()**, and **fopen()** calls the more complicated **open()** system call on your behalf
- Often one says “I am placing a system call” even when calling a higher-level library function
- If the API is standard then the code can be portable!
 - Windows: Windows 16, Windows 32, Windows 64 API
 - UNIX: POSIX (Portable Operating Systems Interface IEEE-IX)
 - Java API: The JVM has OS-like functionalities on top of the OS

Standard APIs: Writing a file

■ System Call in C (`man 2 write`)

- Really a low-level library that directly invokes the system call for you, since one doesn't simply place a system call from user code, as we'll see
- `ssize_t write(int fildes, const void *ptr, size_t nbyte);`

■ Higher level library in C (`man fwrite`)

- `size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream);`

■ Java: `OutputStream::write` (see JavaDoc)

- `public void write(byte[] b) throws IOException;`
- Most details are hidden thanks to OO approach

Standard APIs: Writing a file

■ System Call in C (`man write`)

Returns a possibly negative number (-1 means “failure”)

- Really a low-level library, that directly invokes the system call for you, since one doesn't simply place a system call from user code, as we'll see
- `ssize_t write(int fildes, const void *ptr, size_t nbyte);`

■ Higher level library in C (`man fwrite`)

- `size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream);`

■ Java: `OutputStream::write` (see JavaDoc)

- `public void write(int b);`
 - Most details are hidden
- Returns a ≥ 0 number ($< \text{size}$ means failure)

Standard APIs: Writing a file

■ System Call in C (man `write`) Takes in a number of bytes

- Really a low-level library that directly invokes the system call for you, since one doesn't simply place a system call from user code, as we'll see
- `ssize_t write(int fildes, const void *ptr, size_t nbyte);`

■ Higher level library in C (man `fwrite`)

- `size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream);`

■ Java: `OutputStream::write` (see Javadoc)

- `public void write(int b);`
 - Most details are hidden
- Takes in a number of elements and an element size

Standard APIs: Writing a file

■ System Call in C (`man write`)

Takes in a file descriptor number

- Really a low-level library that directly invokes the system call for you, since one doesn't simply place a system call from user code, as we'll see

- `ssize_t write(int fildes, const void *ptr, size_t nbyte);`

■ Higher level library in C (`man fwrite`)

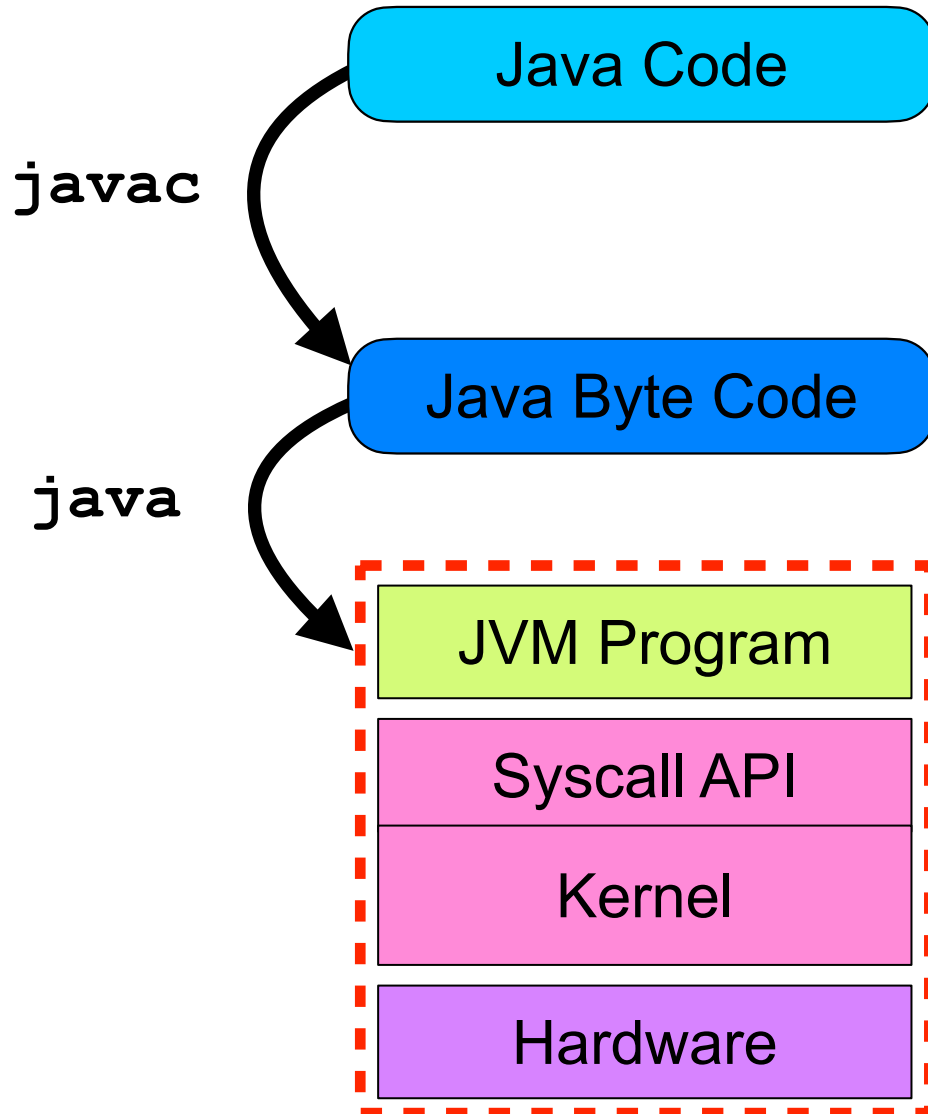
- `size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream);`

■ Java: `OutputStream::write` (see JavaDoc)

- `public void write(...);`
- Most details are hidden

Takes in a higher-level FILE "object"

A Word on the JVM



- The JVM is just a program
- It interacts with the OS using Syscalls, like any other program
- It knows how to interpret byte code that places calls to the Java API
- To implement some of these Java API calls, the JVM places System Calls



Main Takeaways

- OSes come with interactive interfaces
 - Shells, GUIs
- All are based on the Syscall API
 - All (useful) programs use this API
 - Directly or indirectly via standard library calls
- The Linux **strace** tool
- The UNIX **time** tool



Conclusion

- All programs use syscalls at some level
- This makes it possible to analyze, understand, reverse-engineering what programs actually do
- Let's look at optional Homework #1...