System Calls

Module 03

Question:

While a user program is executing, what is the operating system doing?

Interrupts - Review

I/O Device when it's done or a timer expires

INTERRUPT

CPU running user program

The CPU is hard-wired to respond to the interrupt (electrical signal) by setting the PC to a predefined interrupt handler location in memory.

The OS begins running (CPU never stops!)

Protected Instructions

Only the OS is allowed to talk to devices… how is this enforced?

- CPU contains a **mode bit** (part of PSW)
  - 0 is kernel mode, 1 is user mode
- In hardware, there is logic that TRAPS if a protected instruction is executed while mode bit is 1.
  - The TRAP is basically a software generated *interrupt* - the OS will be run and things will be dealt with accordingly

System Calls

The Operating System allows user programs to *request* execution of privileged instructions via function calls

- They aren't standard functions though - so we call the ***system calls***
- Generally, the set of system calls will be provided as a set of **C library calls**
- What about other languages? More on this later...

POSIX Example

To read from disk, the C POSIX API provides the following function:

```
int read(FILE *fd, char *buffer, in bytes);
```

You'd `#include <unistd.h>` to use this on linux

This is no ordinary function though!

An ordinary function

```
#include <cmath>


int main() {
    cout << pow(4, 5) << endl;
}
```

Let’s discussed the compilation and execution of such a program...

What happens when you call `read`?

Initially, it's a lot like a normal call

- Call stack is manipulated (parameters pushed onto call stack)
- However - the read function doesn’t read - it sets a register with the request code, and issues a TRAP

TRAP - what happens?

The CPU instruction TRAP does two things:

1) Flips the **mode bit** to kernel mode!
2) Sets the **PC** to the interrupt vector location

Now the OS will begin running!

In kernel mode...

After the TRAP

The OS will now look at the registers and figure out what to do:

- In this case, it will issue a read request to disk, and then start up another user program
  - Why *another* user program?
  - What's going on in the original user program?
  - What's going on with the **mode bit**?

On `read` completion

1. The Disc controller fires an interrupt
2. The user program running is suspended
3. The OS examines the interrupt information
4. Copies data into original user program’s buffers
5. Schedules the original user program to run
6. Runs scheduling algorithm*
7. Flips to user mode
8. Sets to PC

Dual-Mode Execution

This process is called Dual Mode Execution

- The idea is that user programs cannot execute protected instructions unless the mode bit flips to kernel
- While in user mode, you cannot switch to kernel mode unless you've issued a TRAP
- A TRAP always runs the OS immediately
- The OS can flip the mode bit back to user mode when it's done.

Unlike normal functions - calling a system call results in the calling program being suspended, and another program (the OS) running!

POSIX C API

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

POSIX C API

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

What's an API?

- All flavors of UNIX support POSIX
  - Mac OS X, BSD, Linux, Solaris, etc.
  - However - these are **completely** different operating systems, they simply provide C-libraries with standardized function **prototypes**.
- The underlying implementation are quite different!

What about Windows

- Microsoft Windows supports a very small subset of the POSIX API (and its redundant)
- Windows provides a different API - called **Win32**
- Much like POSIX - its simple a set of standard functions - it's supported by many operating systems… Windows 95-ME, Windows NT, XP, Vista, 7, 8, 10!
- They are all different, but the Win32 API provides a stable API

Compatibility

The whole thing is unfortunate!

- Its generally hard to write C/C++ programs that compile/run on all platforms because you need to target different API's
- Some of this has changed with C++ x11
  - Older C/C++ does this for some common things too, like printing to the console (`cout`)

How come other languages (Java, Python, etc.) don't have this problem?

Other languages

- Java code doesn’t compile to binary - it compiles to an intermediate (byte code)
  - Byte-code is run by another program - the JVM
  - JVM’s are OS specific!

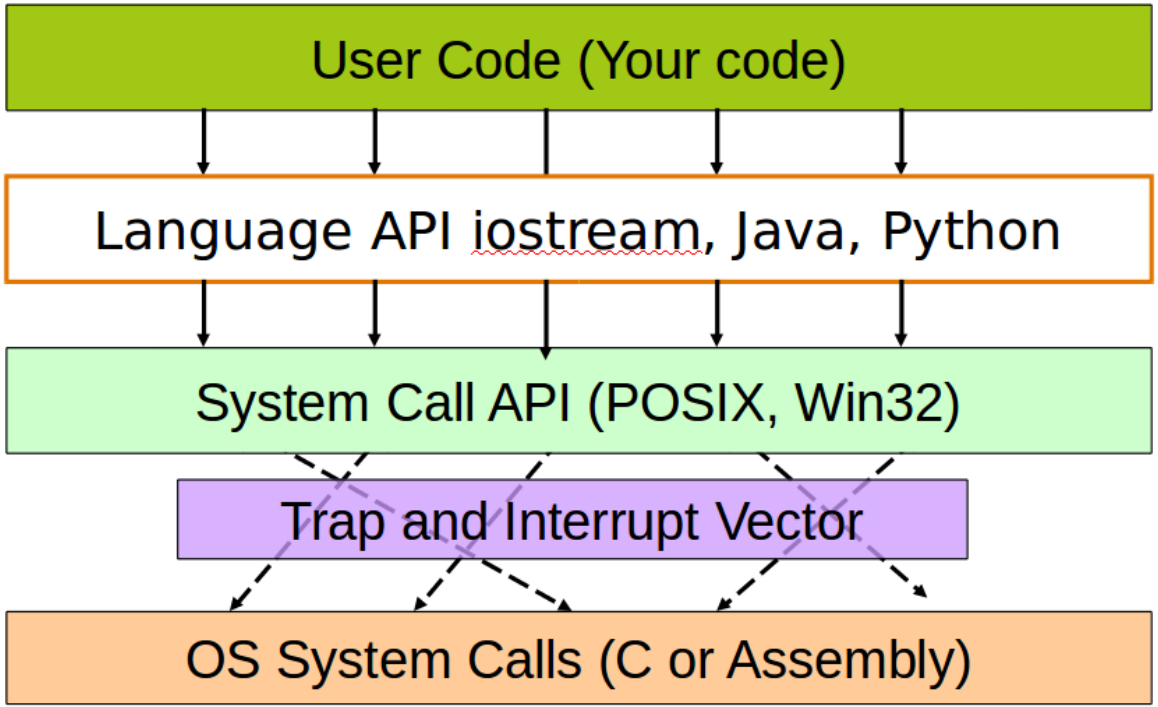
- Python is never compiled, it is interpreted by the Python interpreter
  - When you download the Python interpreter, you need to choose one for your Operating System!

Other Languages

Why are the JVM and Python interpreters platform specific?

Because they either call the C libraries directly (Python) or invoke the (correct) TRAPS themselves (some JVMs)

API Layers


User Code (Your code)
Language API iostream, Java, Python
System Call API (POSIX, Win32)
Trap and Interrupt Vector
OS System Calls (C or Assembly)