System Calls +

The Plan Today...

- System Calls and API's
- Basics of OS design
- Virtual Machines

System Calls

- System programs interact with the OS (and ultimately hardware) through system calls.
- Called when a user level program needs a service from the OS.
  - Generally written in C/C++
  - Execute in kernel mode – code can access protected hardware.
  - Can't be called like a normal function (more soon...)

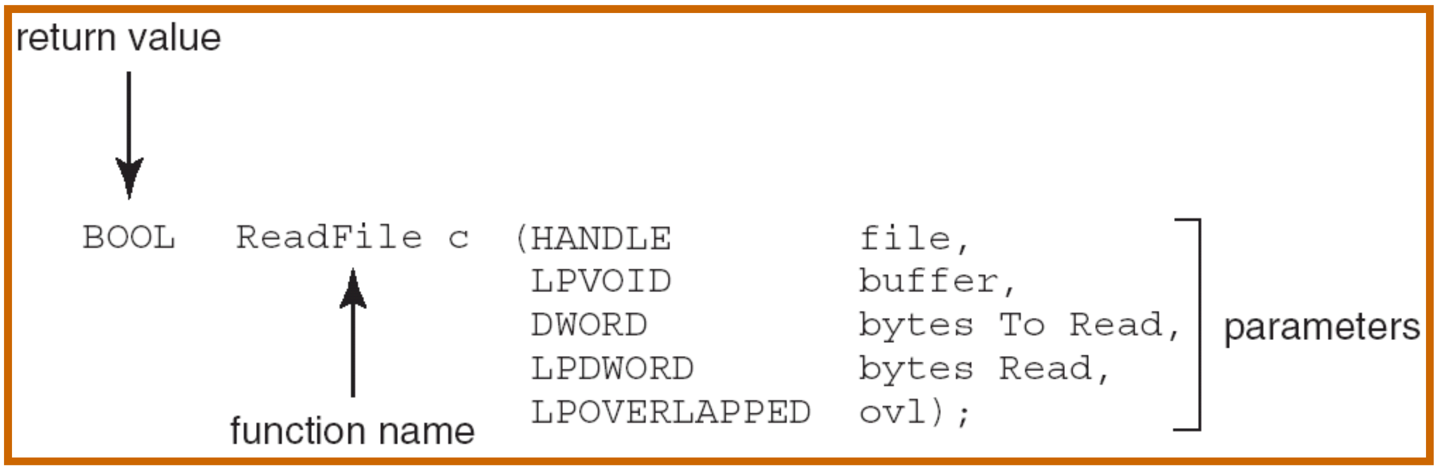
Types of System Calls

- Process control
- File management
- Device management
- Information maintenance
- Communications
  - Message passing
  - Shared memory
- Protection

Application Programming Interface

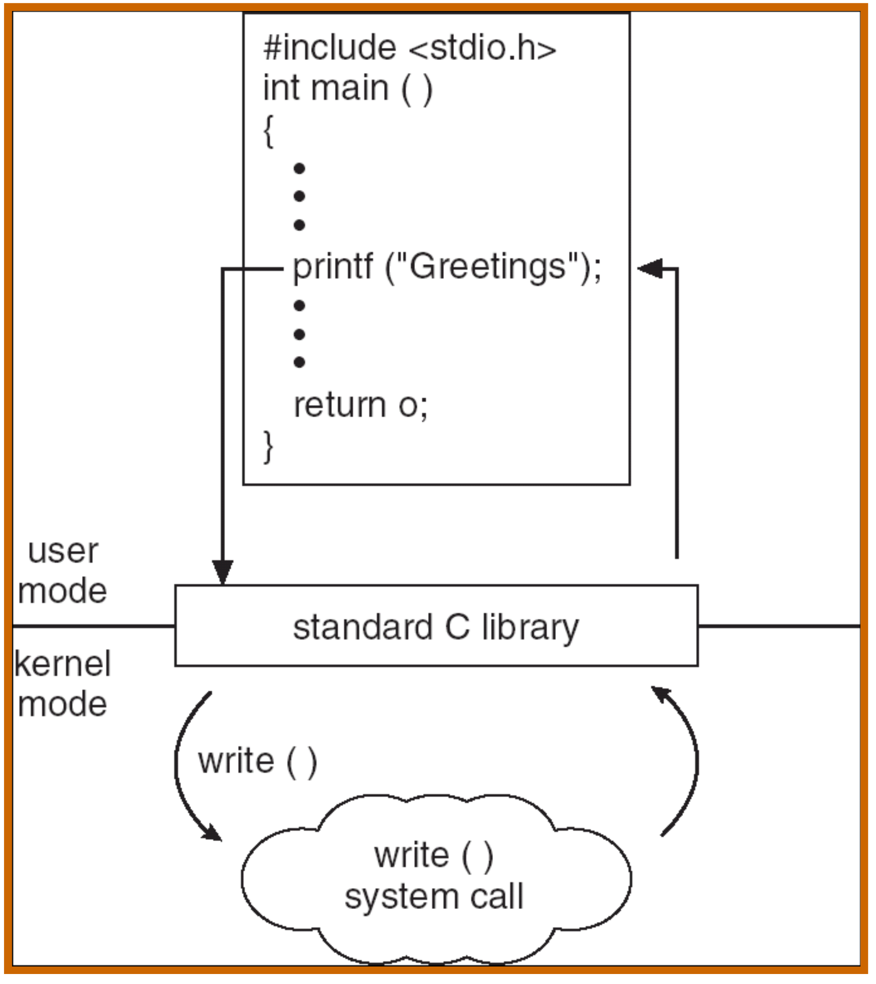
- Application code generally does not invoke system calls directly.
- Programmer calls functions defined by an API.
  - Win32 API (Windows OSs)
  - POSIX API (Most Unix-like OS's)
    - You can check it out at http://www.unix.org/single_unix_specification/
    - Basically a bunch of C header files along with precise, legalistic, descriptions of functionality.

Win32 API Example



- HANDLE file—the file to be read
- LPVOID buffer—a buffer where the data will be read into and written from
- DWORD bytesToRead—the number of bytes to be read into the buffer
- LPDWORD bytesRead—the number of bytes read during the last read
- LPOVERLAPPED ovl—indicates if overlapped I/O is being used

Unix API Invocation Example



More Linux Trivia

- In Linux API is provided by glibc: GNU libc.
- That's why you'll hear GNU/Linux OS.
- You still don't have a useful computer until you get some application programs.
- That's where distributions come in.
  - Debian, Fedora, Ubuntu etc.

Why Use an API?

- API tends to be more “programmer friendly” than direct system calls.
  - Designing an OS involves trade-offs between ease of use, and ease of implementation.
    - System calls – driven by ease of implementation
    - API – driven by ease of use.
  - Some API calls are basically wrappers for system calls.
  - Some are much more complex.
- Coding to an API results in more portable code.

How Do System Calls Work? (In Linux)

- Initiated by a software interrupt.
  - Architecture dependent.
- On x86 architectures:
  - Every interrupt has a unique number.
  - Copy appropriate number to register eax.
  - Copy syscall parameters to registers:
    - ebx, ecx, edx, esi , edi (for up to five parameters.)
    - Put an address in a register for more than five.
  - Execute software interrupt instruction:
    - `int $0x80`

API Example

```
#include <stdlib.h>

int main () {
  exit(0);
}
```

“Direct” syscall Example

```
#include <stdio.h>
#include <sys/syscall.h>

#define __NR_getppid 64



int main()
{
  printf("%d\n", syscall( __NR_getppid ));

}
```

strace

- Let's look at an example...

An Aside: Macros

- C preprocessor can define entities to be expanded in the code.

```
#define BIGNUM 999999
...
if (a > BIGNUM)
     printf(“a is huge.”);
```

- Macros can take parameters...

```
#define max(A,B) (A) > (B) ? (A) : (B)
...
printf(“max of c and d is %d\n”, max(c,d));
```

- It's just simple text substitution.

How Does Linux Handle the System Call?

- There is architecture dependent code in
  - `arch/`*`whatever_architecture.`*
- Assembly code for handling system calls is in:
  - `arch/x86/kernel/entry_32.S` (or _64. S)
  - (Until recently it was: `arch/i386/kernel/entry.S`)
- Other interesting locations:
  - `arch/x86/kernel/syscall_table_32.S`
  - `kernel/sys.c`

OS Design: Separate Policy and Mechanism

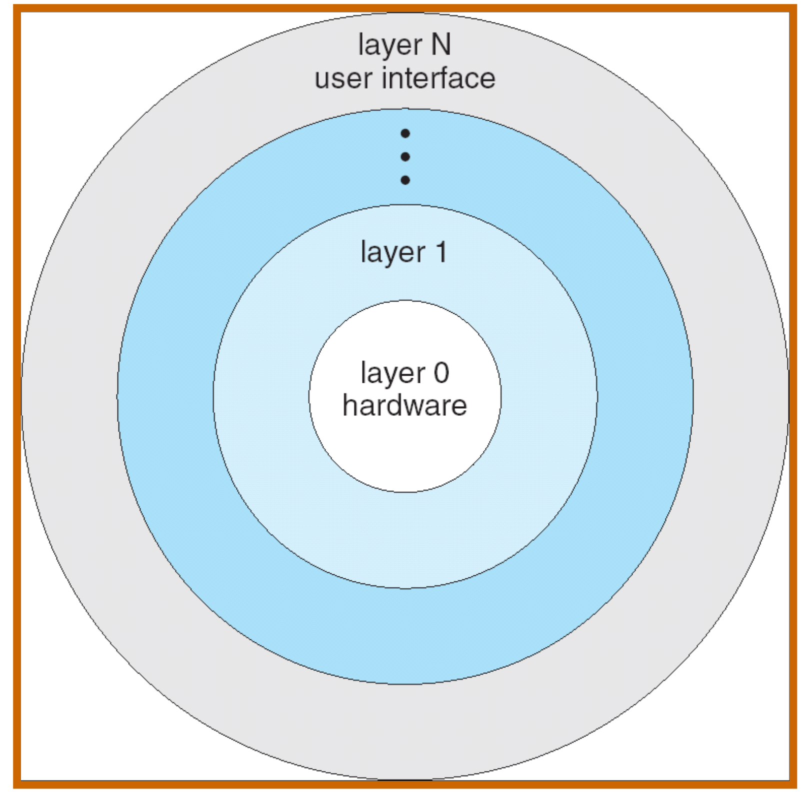
- How to do something (mechanism) vs. what should be done (policy).
- E.g. There needs to be a mechanism to swap out interactive process every N milliseconds.
- N should not be part of the implementation.

OS Design: Basic Organization

- Early OS designs were not particularly modular:
  - MS-Dos
- Some more principled approaches:
  - Layered OS
  - Micro-Kernel
  - Modular OSs

Layered Design

- Challenges:
  - Not always clear what should go in each layer
  - Overhead in moving from one layer to the next.



Micro-Kernels

- Kernel only provides some very basic functionality:
  - Process management.
  - Process communication via message passing.
- Everything else is handled by user level code.
- Advantages:
  - Easy to get a small Kernel right.
  - Easy to port a small Kernel.
  - Elegant design.
- Main disadvantage: slow.

Modular OS

- Most modern operating systems (including Linux) implement kernel modules.
  - Uses object-oriented approach.
  - Each core component is separate.
  - Each talks to the others over known interfaces.
  - Each is loadable as needed within the kernel.
- Modules interact through normal function calls.
  - Not much overhead at run time.

Virtual Machines

- Allow us to simultaneously run multiple OS's on a single computer.
- Many uses:
  - OS design and testing.
  - Maintaining legacy systems.
  - “Honeypots”

Implementing VMs

- You can emulate every hardware instruction in software, e.g. Bochs.
  - Performance is not good.
  - Relatively easy to get an OS running.
- You can depend on functionality from the host OS, e.g. User Mode Linux
  - Instructions run directly on hardware, system calls are captured and sent to host OS.
  - Easier if client OS is similar to host OS.
  - Better performance.
- Other VM systems: VMWare, Xen, VirtualBox

Acknowledgments

- Portions of these slides are taken from Power Point presentations made available along with:
  - Silberschatz, Galvin, and Gagne. Operating System Concepts, Seventh Edition.
- Original versions of those presentations can be found at:
  - http://os-book.com/