System Calls
A contract between processes and the operating system

Michael E. Locasto

Department of Computer Science
UofC
CPSC 457

September 24, 2014
Learning Objective (1 minute)
→ understand the organization and semantics of the system call layer
→ connect w/ last session: OS as an event-driven program
Review: Calling Functions on x86 (10 minutes)
The System Call API (30 minutes)
→ overview: roles, set of syscalls
→ using the API directly
Key Questions

We have been dealing with these three related questions:

How does a kernel enforce the division between userland and kernel space?
What is the necessary architectural support for basic operating system functionality?
How do processes ask the operating system to do stuff?

The basic architecture support made available by a CPU helps isolate the kernel from processes by (1) providing memory separation and (2) providing a mechanism for processes to invoke well-defined operating system functionality in a safe and supervised fashion.
Processes cannot do useful work without invoking a system call. In protected mode, only code marked at ring zero can access certain CPU functionality and state (e.g., I/O).

The OS implements common functionality and makes it available via the syscall interface (the application programs thus do not need to be privileged and they do not need to reimplement common functionality).

(1) An API: a stable set of well-known service endpoints
(2) As a “privilege transition” mechanism via trapping
There are about 300 system calls on Linux, but only a relatively small set are frequently used. We’ve seen some system calls already via strace(1) output. Where is the official list?

/usr/include/asm/unistd_32.h
This provides a comparison to making system calls: in source code, invoking a system call may *look* like making a function call, but it is actually an entirely different mechanism.

In order to more clearly understand the semantics of system calls, we will look at how *function calls* are accomplished on x86.
eax holds the syscall number, which we can get from unistd.h

ebx, ecx, edx, esi, edi hold the system call arguments (either the values or pointers to them, as appropriate – you can check the manual page for the specific system call to discover the argument types)

issue an INT 0x80 instruction; this causes the CPU to generate an interrupt and trap to the OS (via the IDT). The CPU then transitions to supervisor mode (CPL bits set to 00). This is one of the mechanisms that enforce a userspace / kernel split.
Making a system call via an INT instruction can be relatively slow; Intel introduced the SYSENTER / SYSEXIT instructions to make this transition (i.e., context switch) faster. Linux dynamically decides which mechanism is most suitable for the current hardware via the VDSO mechanism.

VDSO definition: http://kernelnewbies.org/KernelGlossary#V
Linux-gate vdso insight:
SYSENTER/SYSEXIT trivia:
http://lkml.org/lkml/2002/12/18/218
An explanation of various aspects of the Linux system call mechanism, including the VDSO:
MOS: 1.6: System Calls (you should have read this already)
LKD: Chapter 5