Operating Systems

Week 2 Recitation: The system call

Paul Krzyzanowski
Rutgers University
Spring 2015

System calls

- System calls are an operating system's API
  - The set of functions that the operating system exposes to processes
- If you want to the OS to do something, you tell it via a system call
- Examples

Windows	Linux
NtOpenFile	*open*
NtReadFile	*read*
NtCreateProcess	*fork*
NtGetCurrentProcessorNumber	*getpid*

See http://j00ru.vexillium.org/ntapi/ for a list of Windows system calls
See http://linux-documentation.com/en/man/man2/ for a list of Linux system calls

What are system calls used for?

- Anything to do with:
  - Accessing devices
  - Accessing files
  - Requesting memory
  - Setting/changing access permissions
  - Communicating with other processes
  - Stopping/starting processes
  - Setting a timer
- You need a system call to:
  - Open a file
  - Get data from the network
  - Kill a process
- You ***do not*** need a system call to:
  - Replace data in a string
  - Create an object (instance of a class)
  - Call a function

System calls are made via traps

- System calls request operating system services
- Operating system code executes with the processor running in *kernel* (also known as *supervisor* or *privileged*) mode
  - Privileged mode gives the CPU the rights to:
    - Execute special instructions (change interrupt masks, set hardware timers, halt the processor)
    - Access specific registers (e.g., private stack pointer)
    - Change the memory map
    - Access regions of memory that have been restricted for kernel access only
    - Access the processor's I/O ports (if the architecture has them)
- A trap takes has one parameter: index into an Interrupt Vector Table
  - The table is in memory that only the kernel can access
  - All addresses in the table go to well-defined entry points in the OS

Variations on software interrupts

- "Classic" system call mechanism in Intel's x86 architecture
  - Use INT 80h (software interrupt) instruction to invoke a system call
  - On Intel architectures, if the privilege level changed, the processor switches to a different stack
    - For security: don't leave kernel crud on a stack that the user might inspect
    - What happens:
      - Various registers are saved in temporary space in the processor (flags, instruction pointer, stack segment, etc.)
      - The new stack pointer is loaded
      - The saved registers are pushed on the stack
      - Any error code indicating the nature of the trap is pushed on the stack
      - Flags are adjusted
      - Execution continues

Variations on software interrupts

- Call gate (Intel x86 architecture)
  - Operating system sets up a "call gate"
  - The user program executes a "CALL FAR" instruction (essentially just a regular subroutine *call* instruction) with a specific segment number
  - The CPU checks if the segment number is a valid "gate"
  - If so, it loads the appropriate instruction pointer and elevates the privilege level
  - Unique to Intel architecture – nobody else used memory segments
    - Hence, portable operating systems avoided this

Variations on software interrupts

- SYSCALL/SYSRET (Intel) or SYSENTER/SYSEXIT (AMD) instructions
  - Faster mechanism than interrupts or call gates
  - Target address is in a CPU register
    ⇒ no need to access memory to do a table lookup
- Linux does a test to check which mechanisms exist before making a system call:
  - Check if *syscall* exists (Intel architecture)
  - Check if *sysenter* exists (AMD architecture)
  - Otherwise use *INT 80* (works on even the oldest processors)
- No matter what is used, the effect is the same:
  - Branch to a well-known location & run in privileged mode

System calls have parameters

- A software interrupt (trap) has one parameter: the trap #
- There are more system calls than interrupt vectors
  - All system calls share the same trap # (the same entry point)
  - Use one vector & have the system call number be a parameter
  - The operating system can jump to the right place based on sys call #
    - Dispatch table
- System calls need to pass multiple parameters
  - E.g., *read* needs to identify the open file, starting byte, number of bytes
  - There are three ways to pass these parameters
    1. In the processor's registers
    2. On the stack
    3. In some memory location whose address is passed to the kernel

Making system calls programmer-friendly

- System calls are made to look like function calls
- As a programmer, you do not want to
  - copy parameters into some special place
  - know the system call number
  - invoke a software interrupt
  - figure out how to copy any return data back
- System call library
  - A user-level library that is linked with your program
  - Provides a functional interface to system calls
  - Handles the work of passing parameters and getting results

System calls

Entry
Trap to system call handler

- Save state
- Verify parameters are in a valid address
- Copy them to kernel address space
- Call the function that implements the system call
  - If the function cannot be satisfied immediately then
    - Put process on a ***blocked*** list
    - ***Context switch*** to let another *ready* process run

Return from system call or interrupt

- Check for signals to the process
  - Call the appropriate handler if signal is not ignored
- Check if another process should run
  - Context switch to let the other process run
  - Put our process on a ***ready*** list
- Calculate time spent in the call for profiling/accounting
- Restore user process state
- Return from interrupt

System call walk-through

1. User calls a system call library function (e.g., *open*)
   - Compiler generates code to push parameters on the stack & call the function
2. The library function is run
   - Compiler generates code to save registers
   - System call number for the open system call (5) is placed in register %eax
   - Other parameters go in registers %ebx, %ecx, and %edx
   - Trap to the OS
3. The operating system kernel code is now run
   - Save registers
   - Look up the address of system call #5
   - Call the system call handler, which processes the request
   - Return the result of the system call in the %eax register
   - Restore other registers
   - Return from interrupt
4. Back to the library function
   - Copy results (if necessary)
   - Restore registers (except for return)
   - Return value to the caller

Note: This is an example using Linux and an x86 architecture. x86-64 uses the 64-bit version of the eax register: rax. Other processors will use totally different registers. Other operating systems may use a different entry point.

The End