Operating Systems II

System Calls

Motivation of System Calls

- System calls describe the interface that exists between user programs and the kernel.
- This interface has to be flexible & powerful
  - Allows for easy development of applications.
- The interface also has to be clear and controlled
  - Necessary for *any* security support the kernel wants to provide.

System Calls

- Define a transition from user mode to system mode.
- How the transition is made can vary dramatically.
- Key factors include:
  - Kernel design – Kernel thread to offer all services.
  - Processor architecture – Processor modes and addressing support.

Simple Design

- The simplest approach to system calls is to provide interrupt “wrapping”.
- That is, each of your system calls starts with disabling interrupts and ends with enabling interrupts.
- This results in the system call taking control of the system without any possibility of pre-emption.

Problems

- No support for kernel developer.
  - Careful to ensure all system calls deal with interrupts properly.
- The approach works but makes some vital assumptions!
  - Disabling interrupts will not cause application problems.
  - Kernel trusts the user!

Alternative Approach

- An alternative approach to system call is to invoke ALL system calls through a common interface.
- This interface will perform the transition between system and user modes.
- A common interface is through an interrupt!
  - Linux uses interrupt 0x80!

What System Call?

- Each system call has a registered number.
- When the system call is made, the *user mode* process will
  - put the system call id into a register
  - push parameters into registers (or onto a stack)
  - then generate the interrupt!
- The kernel interrupt service routine then takes over.

`system_call` ISR

- The global system call handler.
  - saves the registers
  - restricts memory accessing to kernel space.
  - grabs system call id and verifies it is legit.
  - checks if task is being traced/profiled.
  - invokes the system call – with parameters.

`system_call` ISR (cont.)

- After the specific system call function the handler will...
  - Place result of system call in register.
  - Perform system maintenance (scheduling of tasks, check for pending signals, ...)
    - Why not save time since we are in the kernel anyway?
  - Return from interrupt.

System Call Development

- Writing a system is simpler for the kernel developer
  - Write your system call function
  - Pick a system call id
  - Add function pointer and system call id to kernel table.
- Some limitations:
  - Finite number of system calls
  - Up to a fixed number of parameters
- Remember as well functionality in the system call is limited!
  - No access to shared libraries

System Call Stubs

- A system call stub is generated to aid the user application developer.
  - They have to load registers with parameters and system call id then generate the system call interrupt!
- The stub can be generated by the kernel during compilation.
  - `myKernel_foo()`

Where's the Beef in Linux?

- sys_call – routine to handle interrupt 0x80
  - arch/i386/kernel/entry.s
- sys_call_table – table of registered system calls.
  - arch/i386/kernel/entry.s
- NR_syscalls – number of system calls registered (hard coded)
  - arch/i386/kernel/entry.s

Creating a Linux System Call

- Write your system call function (kernel/sys.c)
  - Need to use `asmlinkage` in declaration
- Give your system call an id number
  - Register system call in (order matters) `arch/i386/kernel/entry.s`
  - Make sure to increment `NR_syscalls` (same file)
- Add your system call to the sys_call_table (include/asm/unistd.h)
  - #define __NR_my_sys_call 243

Creating a Linux System Call

- Specify the new system call in the user application.
- `_syscall#` facility to generate the stub.
- Remember, up to 6 arguments supported!

- Or you can invoke the system call without a stub – must `#include <sys/syscall.h>`
  - `syscall(194, &baz, bar);`