Lab: Counting System Calls

Peter Chapin
Vermont State University
CIS-4020, Operating Systems

Objective

- Modify the kernel so that it counts the number of times each system call is executed.
  - Each system call is counted separately.
  - Counts will start at zero when the kernel boots.
  - No ability to reset; counts always increasing.
    - A reset feature might be a nice enhancement.
  - Use 64 bit unsigned integer counters.
    - Limited to 2^{64} - 1 = 18,446,744,073,709,551,615
    - Wrap around very unlikely!
    - Compare with 32 bit counters.

Two Parts

- Part 1: Modify kernel to count system calls.
  1. Create an array with one counter for each system call, initialized to zero.
  2. Modify the system call dispatch code to increment the appropriate counter before invoking the system call.
  3. Rebuild and install the modified kernel.
  4. Verify that the modified kernel can still run the system!

Two Parts

- Part 2: Provide a way for users to view the counts.
  1. Write a module that creates a `/proc` file named `syscalls.txt`.
  2. When the file is read it displays the table of counters in a nice way.
     - This turns out to be the more complicated task.
  3. Other methods to consider...
     - Device driver that returns count information when device is read or as an `ioctl` operation.
     - A new system call that returns the information in a buffer.

Counting System Calls

- How many system calls are there?
  - Consult the dispatch table in syscalls_64.h in `arch/x86/include/generated/asm`
  - For our purposes it is okay to hard code the size.
    - *Note:* the 64 bit kernel may or may not have 32 bit ABI system calls configured into it.
- Where should the counter arrays be declared?
  - Put it in an existing file.
    - This avoids adding a new file to the kernel build system.
    - How about: `arch/x86/kernel/sys_x86_64.c`

Counter Array

- Needs to be an array of unsigned long.
  - `unsigned long syscall_counts[512];`
    - *Replace 512 with a more appropriate value*.
    - Maintenance problem: When new system calls added, developers must remember to change array size.
    - *C will automatically initialize the global array to zeros*.
- *Must export the array to modules*.
  - Modules do not know about symbols in the kernel by default. Use the EXPORT_SYMBOL macro.
    - Look it up in cscope.
    - See how it is used elsewhere

Increment the Counts

- Modify the assembly language file
  `arch/x86/kernel/entry_64.S`
  - Look for the system call entry point:
    - `ENTRY(system_call)`

```
...
cmpq __NR_syscall_max, %rax
ja ret_from_syscall
movq %r10, %rcx
# Add increment operation here!
call *sys_call_table(,%rax,8)
```

  - Increment appropriate counter after %rax verified, but before the call is actually made.

Rebuild

- Recompile the kernel...
  - ... and install your new kernel.
    - Be sure to back up the old kernel!
- Verify that the system still works.
  - If it boots at all, there is most likely no problem.
    - If you broke system call dispatching, no applications will be able to execute successfully.

Performance

- Consider the performance cost of system call counting.
  - Memory...
    - The counter table requires extra kernel memory. How much memory? Is it a problem?
  - Execution time...
    - Incrementing the counters requires extra time. How much time? Is it a problem?

Kernel Module Programming

- Big topic...
  - Modern Linux supports many kinds of modules.
    - Device drivers
    - Network protocol drivers
    - File system drivers
  - Kernel exports functions that modules can use.
  - BUT... not everything can be done in a module.
    - Hooking system calls can not.

`syscall_counters.c`

- Lab comes with a skeleton module.
  - Creates a `/proc` file.
  - You must flesh out the skeleton so that it returns a formatted list of the counters when that file is read.
- Problem: The formatted list is large (kinda).
  - 41 characters per line.
  - 350 system calls (for example) * 41 characters per call = 14350 bytes.
  - Normally kernel only provides a single memory page. (4096 bytes)

seq_file

- To make it easier to return large amounts of `/proc` file data, Linux provides the seq_file API.
  - When user tries to read the file.
    - Kernel calls your "start" function.
    - Then calls your "next" function repeatedly.
    - Finally calls your "stop" function.
  - Each call to your "next" function returns one record of data.
    - You define what that means (in our case, a line of text for one counter seems natural).

Easier?

- This is easier because...
  - Kernel deals with the application:

```
read(fd, buffer, 64)
```

    - Application reads a chunk of data without any knowledge of how many "natural" records it might contain.
    - The old `/proc` file methods required you (the module author) to coordinate this.
    - The seq_file handling in the kernel takes care of that for you.
    - Plus it allows you to return a lot of data without worrying about memory page sizes, etc, etc.