Project 2
Linux Kernel Hacking

CS-3013 Operating Systems
Hugh C. Lauer

(Slides include materials from
Modern Operating Systems, 3rd ed., by Andrew Tanenbaum
and from Operating System Concepts, 7th ed., by Silbershatz, Galvin, & Gagne)
**Objective**

- To learn how to work inside an operating system kernel
- To understand some of the constraints and techniques of programming in a kernel (*versus* user space)
This Assignment

• Add a new system call to the Linux kernel

• Add a second system call to get useful information from the data structures of a Linux kernel
Background –
User mode vs. Kernel mode

- Hardware provides two or more modes
  - Indicated by bits in PSW
- Allows OS to protect itself & system components against
  - Faulty and malicious processes
- Some instructions and memory locations are designated as privileged
  - Only executable or accessible in kernel mode
- System call, all traps, & interrupts change mode from user to kernel
  - return from system call resets mode to user
Transition from User to Kernel Mode

- Note: each different system call has its own number or other identity.
- Kernel trap handler uses syscall number to index into table of syscall routines.
Inside Kernel, the OS can ...  

- Read and modify data structures not in user address space  
- Control devices and hardware settings forbidden to user processes  
- Invoke operating system functions not available to user processes  
- Access address of space of invoking process
Processes – Address Space

Virtual address space

Kernel Space

User Space

Kernel Code and Data

stack
(dynamically allocated)

heap
(dynamically allocated)

static data

code
(text)

0x00000000

0xFFFFFFFF

32-bit Linux & Win XP – 3G/1G user space/kernel space

PC

SP

Not readable or writeable in user mode
Accessing the Kernel via System Call

• Normally embedded within a library routine
  • User API *never* makes system calls directly
• System call mechanism is machine specific
  • Different CPU architectures make system calls in different ways
• System call *numbers* different for various architectures
  • Even for same operating system & version!
  • E.g., `poll` system call is #167 on PowerPC but #168 on Intel 386 platforms (in SUSE Linux 9.3)
Accessing Kernel via Library interface

user application

open ()

user mode

system call interface

kernel mode

Implementation of open ()

system call

return
Accessing Kernel via Library interface

```c
#include <stdio.h>
int main ()
{
    
    printf("Greetings");
    
    return 0;
}
```

Diagram:
- User mode
- Kernel mode
- Standard C library
- `write()` method
- System call

CS-3013, A-Term 2011
Project 2 Linux Kernel Hacking
In this project, we will ...

- Add a new system call to the Linux kernel
  - It does nothing except announce its presence

- Add a second system call to provide information about the calling process
  - Some of which is not readily available via existing system calls

- Follow Linux naming & numbering conventions
In this project, we won’t ...

• ... bother to make a library to encapsulate our systems calls
• ... try to support them on all machine architectures
Part 1: Adding a System Call

• See *Linux Kernel Development*, 3rd ed., Ch. 5
  • *System Calls*
  • Many how-to details, but some things have changed

• Clone a new kernel tree as in *Project 0*
  • `cp -al /usr/src/linux-2.6.37.6-0.5 kernelSrc`
  • Remember to build to a destination – `O=~/kernelDst`

• Note:— need to clean up disk space in virtual machine
  • Start with new clone; or
  • Remove boot files from previous projects & use **YaST** to clean up boot configuration
Linux Naming Convention (all versions)

• If your library routine is `alarm(int seconds)`
  
• … then the corresponding system call name is `sys_alarm`

• … and the corresponding function prototype for its kernel implementation is

  `SYSCALL_DEFINE1(alarm, unsigned int, seconds)`
Linux Naming Convention in Kernel (continued)

- `SYSCALL_DEFINE(name, type, parameter_name, type, parameter_name, ...)`

- ... is a macro that expands to `asmlinkage long sys_name(n params)`

- `asmlinkage` is a compiler directive that generates a special way of passing arguments across the privilege boundary
Robert Love says …

- To invoke `alarm` system call from a library routine in user space, use macro
  `__syscall1(unsigned long, alarm, unsigned int seconds)`
- `_syscalln` has \( n+2 \) arguments
  - Return type
  - Name of actual system call (in user space)
  - Arguments to system call function
- This macro defines the function
  `unsigned long alarm(unsigned int seconds)`
Linux Conventions (modified)

- _syscalln is “deprecated”
  - I.e., Linux/Unix speak for “don’t use this any more!”
  - It is officially on the way out (even if it still works)
- Instead, use
  - syscall(callNumber, ...), where ... are the arguments to the system call.
  - Result must be cast to appropriate type
- Example, for alarm system call, write
  ```c
  long alarm (unsigned int seconds) {
      return (long) syscall(__NR_alarm, seconds);
  }
  ```
Hello, World!

- First system call will be `helloworld`
  - No arguments
  - Return `long`
helloworld System Call

• /* This is the text of the helloworld system call implementation */
  
  SYSCALL_DEFINE0 (helloworld){
    printk(KERN_EMERG "Hello, world!\n");
    return 0;
  }

• Add to the file kernel/sys.c

Note: No comma here!

Don’t forget the newline character after the end!
printk() — the Kernel Debug Print Tool

- Very robust
  - May be called from (almost) anywhere in kernel
  - Same calling convention as printf()
  - Writes to system log
  - Output survives crashes (almost all of the time)

- To read output, see
  - /var/log/messages: Circular log, newest messages at end
  - Read with YaST > Miscellaneous > System Log
  - or /bin/dmesg

- See Linux Kernel Development, 3rd ed., Chapter 18
helloworld System Call

• /* This is the text of the helloworld system call implementation */
  SYSCALL_DEFINE0 (helloworld){
    printk(KERN_EMERG "Hello, world!\n");
    return 0;
  }

• Add to the file
  kernelSrc/kernel/sys.c
Registering your System Call

- **arch/x86/include/asm/unistd_32.h**
  - Add entry for your call number
  - *Increment total number of calls*

- **arch/x86/kernel/sysscall_table_32.S**
  - Lists entry points for system calls
  - Must be kept in numerical order!
  - Number must correspond to *unistd_32.h*

- Rebuild and install your kernel
Note #1

- The file organization in this part of the Linux kernel source tree seems to change from year to year.
- On x86 architecture (i.e., Pentium), the syscall table has moved since:
  - Robert Love’s book
  - Previous courses
- It used to be in:
  - arch/i386/kernel/entry.S
- But now it is in:
  - arch/x86/kernel/syscall_table-32.S
  - … which is included by entry.S
- Location of include files also changes.
Note #2

- The x86_64 architecture does it differently
  - Everything is in
    include/asm-x86_64/unistd.h
  - Add to the list
    ```
    #define 251 /*next number in list*/
    __SYSCALL(__NR_helloword, sys_helloword)
    ```
Note #3

• Remember: – to edit source file foo.h in your kernel tree
  – Move it to foo.h~
  – Make changes and save to foo.h
Testing your System Call

• In user space:

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

#define __NR_helloworld 333
    /* or whatever number you put in unistd-32.h */

long helloworld(void) {
    return (long) syscall(__NR_helloworld);
};

main () {
    printf("The return code from the helloworld "
            "system call is %d\n", helloworld());
}
```

• Run it in your new kernel; check log for `printk()` message!
Creating a Patch File

- One level above kernel source tree, do
  
  ```
  diff -urN /usr/src/linux-2.6.37.6-0.5 kernelSrc > patch1
  ```

- To recreate your directory from patch
  
  ```
  cp -al /usr/src/linux-2.6.37.6-0.5 newSrc
  cd newSrc
  patch -p1 < patch1
  ```

- Do not prefix name of kernelSrc directory or use fully qualified name
  
  - E.g, do not use ~/kernelSrc, ./kernelSrc
patch file (continued)

- Be sure to clean out extraneous files from your source tree
  - E.g., *sys.c~* (created when you edited *sys.c*)
  
- If not, your patch file will be too big to submit!
  - Also, graders will refuse to grade submission
**patch program**

- Official version in /usr/bin is v. 2.6.0
  - Does not automatically replace read-only files

- Old version in /usr/local/bin is v. 2.5.9
  - When patching a read-only file, it moves it aside and puts a new version in its place
Submission – Part 1

- Patch1
- Test program
- Makefile and write-up will be combined with Part 2

- Part 1 due next Friday!
- This is Project2_Part1
End of Part 1

Questions?
Part 2: Get Process Information

• Modify your kernel of Part 1 to add another system call to get information about process
  • Please leave **helloworld** system call in place!

• System call is
  − `long getprinfo(struct prinfo *info)`
  − `info` is pointer to caller area to receive results
    • In user-space!
  − Returns zero if successful, error code if not

• See handout for definition of **struct prinfo**
  − Download from [http://www.cs.wpi.edu/~cs3013/a11/Common/prinfo.h](http://www.cs.wpi.edu/~cs3013/a11/Common/prinfo.h)
Two-person Teams (optional)

- You *may* do Part 2 in two-person teams.
- Please register your team with the Tas
  - Submit only once for the team
  - Both members share the same grade for Part 2
- Academic honesty with respect to teams — *all members must carry roughly equal shares of the effort*
Information needed for `prinfo`

- See `task_struct` in `include/linux/sched.h`
- See `getuid` and `getpid` for examples of simple system calls
- See `include/asm/current.h` to find current process information
  - E.g., `current -> pid` is process ID of current process
- Use `copy_to_user` to safely copy data from kernel to user space *(next slide)*
- Return `EFAULT` error code if `info` argument is not valid pointer in user space
`copy_to_user` and `copy_from_user`

```c
#include <asm-generic/uaccess.h>
```

- Functions to safely copy data to/from user space
- Check validity of user-space pointer arguments
- Return zero if successful, number of bytes that fail if there is a problem
- Immune to page faults, pre-emption, null pointers, other errors, etc.
Implementing `getprinfo` System Call

- Add after `helloworld` system call from Part 1
- Copy `prinfo.h` to `include/linux` in kernel tree
- Implement `kernel/prinfo.c`
  - Edit `kernel/Makefile` to add `prinfo.o`
- Register in `unistd_32.h` & `syscall_table_32.S`
- Use `printk()` to print debugging statements to system log
  - For your debugging convenience
More on `getprinfo`

- Most of the information is easy
- Can be obtained from `task_struct` of process
- Siblings and children are harder —
  - Involve Linux kernel list traversal macros
  - See handout; also Ch 6 of *Linux Kernel Development*
- Suggestion:— get everything else working before attempting to master these
Testing `getprinfo()`

- Write test program in user space
  - Must have own user space version of `prinfo.h`
- Run multiple times from same shell, different shell, different processes
- Note differences in results
- Compare with what you can find about processes from `ps` command and from Project 1 program.
Testing `getprinfo()` (continued)

- For two points of extra credit:–
- Construct a test program that
  – forks multiple processes,
  – shows a hierarchy of children and grandchildren, and
  – lots of siblings
Submission – Part 2

- **Patch2**
  - Difference between *original* source tree and Part 2 kernel.
  - Includes patch lines from Part 1
- User-space test program
  - Include file(s)
  - Test program itself
  - **Makefile** for *both* Part 1 and Part 2
- Short writeup describing both parts
- Submit using *Turnin* — This is *Project2_Part2*

I.e., cumulative with Patch1
Warning!

• Check your patch files before submitting
  • i.e., open in a text editor
• Should be a few kilobytes
• Every line added by patch file should be something you wrote
  • Be sure no junk is lying around in your source tree
  • E.g., “~” files, old files, build files
• If your patch file is 100s of kilobytes or megabytes, it is wrong!
• Graders will refuse to grade your project!
Submission (continued)

• Put your name on all documents and at top of every edited file!
Due Dates

• Pace yourself:—
  – Part 1 due *next Friday*, September 9
  – Part 2 due following week, September 16

• Report to instructor or TAs any difficulties
Questions?