CSE 451: Operating Systems

Section 2
Interrupts, Syscalls, Virtual Machines, and Project 1
Interrupts

- Interrupt
  - Hardware or software
  - Hardware interrupts caused by devices signaling CPU
  - Software interrupts caused by code
- Exception
  - Unintentional software interrupt
  - E.g. errors, divide-by-zero, general protection fault
- Trap
  - Intentional software interrupt
  - Controlled method of entering kernel mode
  - System calls
Interrupts (continued)

- Execution halted
- CPU switched from user mode to kernel mode
- State saved
  - Registers, stack pointer, PC
- Look up interrupt handler in table
- Run handler
  - Handler is (mostly) just a function pointer
- Restore state
- CPU switched from kernel mode to user mode
- Resume execution
Interrupts (continued)

• What happens if there is another interrupt during the handler?
  • The kernel disables interrupts before entering a handler routine

• What happens if an interrupt fires while they are disabled?
  • The kernel queues interrupts for later processing
System calls

- Provide userspace applications with controlled access to OS services
- Requires special hardware support on the CPU to detect a certain system call instruction and trap to the kernel
System call control flow

- User application calls a user-level library routine (gettimeofday(), read(), exec(), etc.)
- Invokes system call through stub, which specifies the system call number. From unistd.h:
  
  ```
  #define __NR_getpid 172
  __SYSCALL(__NR_getpid, sys_getpid)
  ```
- This generally causes an interrupt, trapping to kernel
- Kernel looks up system call number in syscall table, calls appropriate function
- Function executes and returns to interrupt handler, which returns the result to the userspace process
Specifics have changed since this diagram was made, but idea is still the same.
How Linux does system calls

- The syscall handler is generally defined in arch/x86/kernel/entry_[32|64].S
- In the Ubuntu kernel I am running (2.6.38), entry_64.S contains ENTRY(system_call), which is where the syscall logic starts.
- There used to be “int” and “iret” instructions, but those have been replaced by “sysenter” and “sysexit”, which provide similar functionality.
Syscalls in a virtual machine

- For software VMMs (e.g. VMWare Player, VirtualBox, Microsoft Virtual PC), there are a couple options:
  - Install hardware interrupt handler for each VM (requires CPU support, such as with Core 2 Duo and up)
  - Use dynamic rewriting to avoid hardware trap entirely
- For paravirtualized VMMs (e.g. Xen) parts of the OS are actually rewritten to avoid hardware traps
- For hardware VMMs a.k.a embedded hypervisors (e.g. VMWare ESX), sandboxing requirements are smaller, as the only user-mode entities are VMs
- Is one approach “best”?
Project 1

- Three parts of varying difficulty:
  - Write a simple shell in C
  - Add a new system call and track state in kernel structures to make it work
  - Write a library through which the system call can be invoked
- Due: April 18 at 11:59 PM.
  - Turn in code plus a writeup related to what you learned/should have learned
The CSE451 shell

• Print out prompt
• Accept input
• Parse input
• If built-in command
  • Do it directly
• Else spawn new process
  • Launch specified program
  • Wait for it to finish
• Repeat

CSE451Shell% /bin/date
Sat Mar 31 21:58:55 PDT 2012
CSE451Shell% pwd
/root
CSE451Shell% cd /
CSE451Shell% pwd
/
CSE451Shell% exit
CSE451 shell hints

- In your shell:
  - Use *fork* to create a child process
  - Use *execvp* to execute a specified program
  - Use *wait* to wait until child process terminates

- Useful library functions (see man pages):
  - Strings: *strcmp*, *strncpy*, *strtok*, *atoi*
  - I/O: *fgets*
  - Error report: *perror*
  - Environment variables: *getenv*
• Advice from a previous TA:
  • Try running a few commands in your completed shell and then type exit. If it doesn’t exit the first time, you’re doing something wrong.
  • `echo $?` prints the exit code, so you can check your exit code against what is expected.
  • Check the return values of all library/system calls. They might not be working as you expect.
  • Don’t split the project along the three parts among group members. Each one should contribute some work to each part or you won’t end up understanding the big picture.
Adding a system call

- Add `execcounts` system call to Linux:
  - Purpose: collect statistics
  - Count number of times you call fork, vfork, clone, and exec system calls.

- Steps:
  - Modify kernel to keep track of this information
  - Add `execcounts` to return the counts to the user
  - Use `execcounts` in your shell to get this data from kernel and print it out.
  - Simple, right? ;)

4/5/2012
Programming in kernel mode

• Your shell will operate in user mode
• Your system call code will be in the Linux kernel, which operates in kernel mode
• Be careful - different programming rules, conventions, etc.
Userspace vs. kernel mode conventions

- Can’t use application libraries (e.g. libc)
  - E.g. can’t use printf
- Use only functions defined by the kernel
  - E.g. use printk instead
- Include files are different in the kernel
- Don’t forget you’re in kernel space
  - You cannot trust user space
  - For example, you should validate user buffers (look in kernel source for what other syscalls, e.g. gettimeofday() do)
Kernel development hints

- Use grep as a starting point to find code
  - For example:
  - `find . -name *.c | xargs grep -n gettimeofday`
  - This will search all c files below your current directory for `gettimeofday` and print out the line numbers where it occurs
- Pete has an awesome tutorial on the website about using ctags and cscope to cross-reference variable, struct, and function definitions:
Kernel development hints (continued)

• Use git to collaborate with your project partners
  • Pete has a guide to getting git set up for use with project 1 on the website:
  • Overview of use:
    • Create a shared repository in /projects/instr/12sp/cse451/X, where X is your group’s letter
    • Check the project’s kernel source into the repository
    • Have each group member check out the kernel source, make modifications to it as necessary, and check in their changes
    • See the web page for more information
Project 1 development

- Option 1: Use VMWare on a Windows lab machine
  - Can use forkbomb for kernel compilation (fast)
  - ...or use the VM itself for kernel compilation (slow)
  - The VM files are not preserved once you log out of the Windows machine, so copy your work to attu, your shared repository, or some other “safe” place

- Option 2: Use your own machine
  - Can use VMWare, VirtualBox, or your VMM of choice
  - See the “VM information” page on the website for getting this set up
Project 1 development (continued)

- If you build the kernel on forkbomb, copy the resulting bzImage file to your VM and overwrite /boot/vmlinux-2.6.38.2-CSE451
- If you build the kernel inside the VM, run `sudo make install` from inside the kernel directory to install it
- Reboot with `shutdown -r now`
- If your kernel fails to boot, pick a different kernel from the menu to get back into the VM
- While inside the running VM, use the `dmesg` command to print out the kernel log (your `printks` will show up here—use `grep` to find the ones you care about)