CSE 451
Section 3
Project 1 (and all its glory)

Reflections

- Project 0 is finished and the real fun starts.
- Homework grades should be posted.
- Shared space and SVN repos will be assigned today.
  - Sorry for the delay

- The speedometer seems to have fallen.
  - But, remember, there's also the anonymous feedback on the site.

Interrupts

- Interrupt
  - Hardware or software
  - Hardware interrupts caused by devices signalling 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

What happens in an interrupt?

- Execution halted
- CPU switched from user mode to kernel mode
- State saved
  - Registers, stack pointer, PC
- With interrupt number index interrupt descriptor table to find handler
- Run handler
  - Handler is (mostly) just a function pointer
- Restore state
- CPU switched from kernel mode to user mode
- Resume execution

Interrupts

- What happens if there's another interrupt during the handler?
- What happens if an interrupt fires when they are disabled?

System calls

- An invocation of an OS service
  - So the OS can manage and protect services
- Requires architectures support
  - But the most basic mechanism is just an interrupt

Syscall control flow

- User application calls a library
    - User-level library call
- Invoke system call through stub
    - _syscallN() sets up arguments for the OS
        - _syscallN() might be the out-of-date way...
- Software interrupt with syscall number.
- Syscall handler indexes system call table to find a vector to jump to
- OS performs operation
    - Must check arguments! Cannot allow user to trick the OS into performing an unsafe operation!
- OS prepares return
- OS returns from interrupt and resume user

How Linux does syscalls

- You can see what happens when the kernel invokes a syscall here:
  - Arch/x86/kernel/entry_32.S
    - If you don't mind reading assembly
- Doesn't really use “interrupts” anymore
  - “int 0x80” and “iret” replaced by “sysenter” and “sysexit”
  - Similar operations supported by architecture

Project 1

- Three main parts:
  - Write a simple shell in C
  - Add a simple system call to Linux kernel
  - Write a program using your system call
- Due: Fri., Oct 21, 11:59pm
  - Electronic turnin: code + writeup

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, getline, readline*
  - Error report: *perror*
  - Environment variables: *getenv*

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.

The question everyone asks...

- Your numbers will likely look like a high number of clone and exec calls, with very few or none to vfork and fork.

Writing a program using your system call

- Run a given program and get the fork/vfork/clone/exec call counts for it.
  - This is analogous to 'time' in that 'time' will run a program and report its running time.
- Write this program as one would if they any client of a system call (e.g. someone who didn't write the system call in the first place).
- Implement a signal handler to interrupt the running program and print its counts up to the given point.

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.

Programming in kernel mode

- 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
- Don't forget you're in kernel space
  - *You cannot trust user space*
  - E.g. unsafe to access a pointer from user space directly

Kernel development hints

- Best way to learn: read existing code
- Use grep –r *search_string* *
  - -I for case-insensitive
- Use LXR (Linux Cross Reference): *http://lxr.linux.no/*

Requirements and Caveats

Shell Requirements

- It's okay to only support a limited buffer or argument length.
  - Normally, this is bad. Setting hard limits.
  - There are some library calls to alleviate the problem (getline, readline, etc.).
  - Be careful with these though, they contain static state!
- The shell probably not be terribly long. Only ~100 or so lines.

System Call Requirements

- Implement your system call with the number 341.
- Follow the library interface.
- Don't worry about synchronization issues.
  - Imagine you're working on a uniprocessor and the kernel is not preemptable.
    - ...which is not really the case...
- This is also true when adding the signals in the final part.
  - There is a small race condition if the signal fires before the child execs.
- Again, it's unlikely you'll need to write much code.

Caveats

- Linux recently updated with the concept of PID namespaces. A virtual PID (vpid) is the PID of the currently used namespace and should be appropriate for this assignment.

Submission Requirements

- Your write-up is a major part of your grade and shouldn't be neglected.
- For changed Linux source files:
  - Give full path names in your modified files write-up
    - USE "./arch/i386/kernel/process.c"
    - NOT "process.c" – there are many of these
  - Maintain directories when submitting changed files:
    - When I extract your changed files, they should go to the right directory, so it is unambiguous which file you changed
    - This is easy to do with tar

Build Options

- Run your shell on forkbomb or your own machine.
  - Do not forkbomb attu.
  - You won't be able to kill a forkbomb individually. Use killall.

Watch out for...

- What architecture the code you're reading is for:
  - You'll want x86
  - And 32-bit!
- You're working on the latest stable...
  - ...but a lot of online resources are for older versions! ...even the previous slide...
- Your environment
  - VMWare is supported by us. Virtualbox is possible but you'll have to solve some problems yourself.
- Everything has an up-to-date way to do it and an obsolete way.
  - For the sake of this class, what works will work but its still something to look out for.

Linux directory structure

- mm → memory management
- ipc → interprocess communication
- fs → files system
- include → user exposed headers
- kernel → core OS
- arch → architecture specific code
  - Much of the lower level implementation is here