CSE 451: Operating Systems

Section 2

Interrupts, system calls, 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

Interrupt handling

- Execution of current process halts
- CPU switches from user mode to kernel mode, saving process state (registers, stack pointer, program counter)
- CPU looks up interrupt handler in table and executes it
- When the interrupt handler finishes, the CPU restores the process state, switches back to user mode, and resumes execution

Interrupt handling

- What happens if there is another interrupt during the handler?
  - The kernel disables interrupts before entering a handler routine?
  - Preemption
- 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
- x86 uses the INT X instruction, X in [0,255]

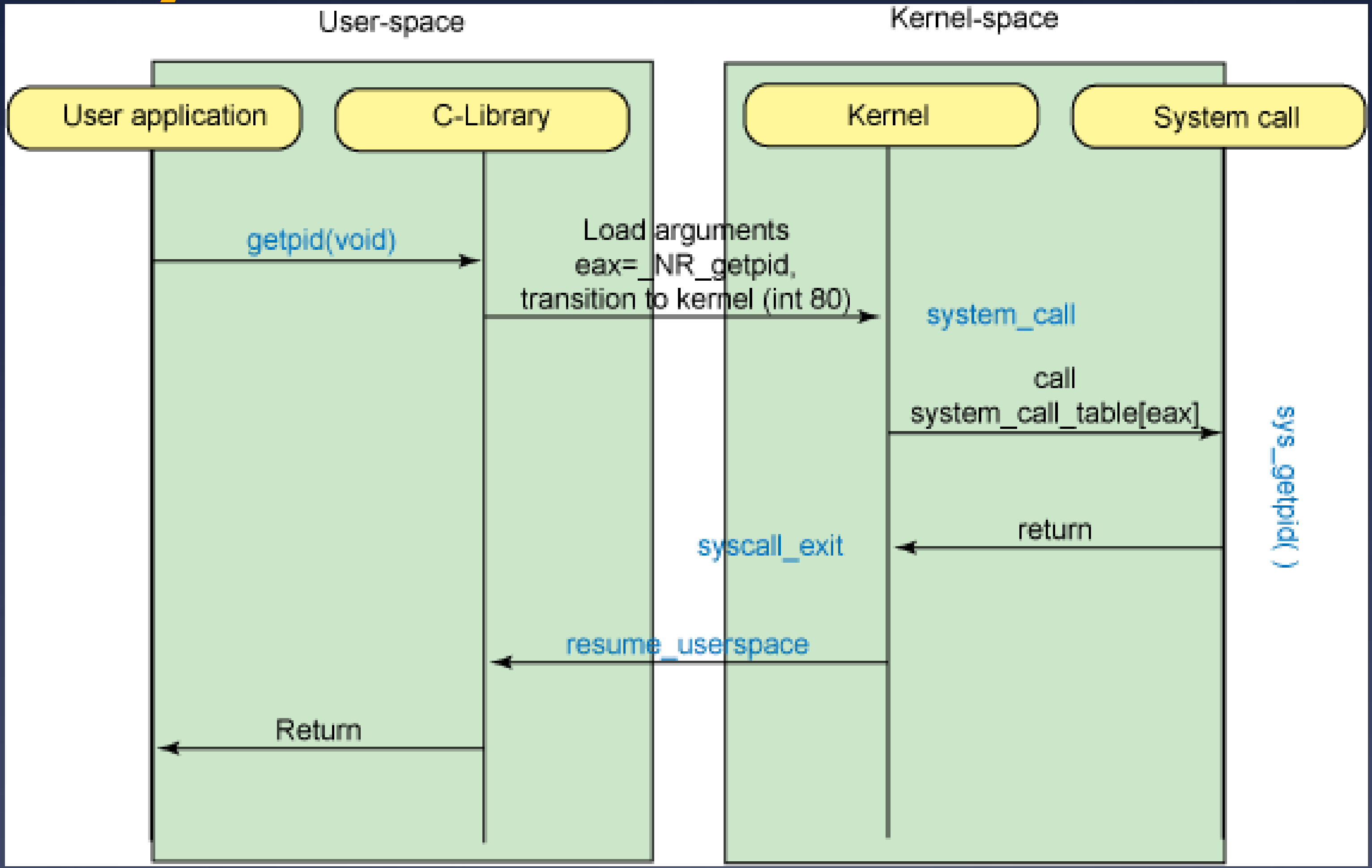
System call control flow

* 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

System call control flow



* Specifics have changed since this diagram was created, but the idea is still the same

Linux Syscall Specifics

- The syscall handler is generally defined in `arch/x86/kernel/entry_[32|64].S`
- In the Ubuntu kernel I am running, `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- There used to be "`int`" and "`iret`" instructions, but those have been replaced by "`sysenter`" and "`sysexit`", which provide similar functionality.

Project 1

* Due: **April 24** at 11:59 PM.
* 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
* Turn in code plus a write-up 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
Wed Apr 31 21:58:55 PDT 2013
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` or (preferrably) `readline`
  - Error reporting: `perror`
  - Environment variables: `getenv`

CSE451 shell hints

* 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 last 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
  * Each partner in your group should contribute some work to each piece 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 a process *and all of its descendents* call the `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

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.

Kernel programming

* Can’t use application libraries (e.g. libc)
  * No printf—use prink instead
* Use only headers/functions exposed by 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 find + grep as a starting point to find interesting code

```
find . -type f -name "*.h" -exec grep -n \
  gettimeofday {} +
```

- Pete Hornyack (a previous TA) put together a tutorial on using ctags and cscope to cross-reference type definitions: http://www.cs.washington.edu/education/courses/cse451/13sp/tutorials/tutorial_ctags.html

Kernel development hints

- Use Git to collaborate with your project partners
  - There is a guide to getting Git set up for use with project 1 on the website:
    - http://www.cs.washington.edu/education/courses/cse451/13sp/tutorials/tutorial_git.html
  - Overview of use:
    - Create a shared repository in /projects/instr/13sp/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
- Git makes it easy to find any files you've changed.

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/git push 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: http://www.cs.washington.edu/education/courses/cse451/13sp/vminfo.html

Project 1 development

* Once you have built the kernel, copy the resulting bzImage file to your VM and overwrite `/boot/vmlinuz-3.8.3-201.cse451custom`
* Reboot with `sudo 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 `printk`s will show up here—use `grep` to find the ones you care about)

Project 1 development

- Instructions will be coming out soon for using Qemu to test the kernels
  - Much more convenient than Vmware
  - It will run in a terminal window
  - You can debug the kernel from your host machine using GDB
  - It's a bit trickier to set up … but good stuff to know if you plan to get into backend dev
  - Forkbomb is a Qemu virtual machine!

Time Left?

- We could chat about
  - Linux kernel basics – modules, compiling, configuring
  - Some nice features the Linux kernel provides
  - The weather
  - Workflow tricks (automation is your friend)