

1

LINUX
CNO
Programming



LINUX
KERNEL
INTERNALS

KERNEL MODE DEVELOPMENT MODULE



LINUX KERNEL SERIES



- After the Linux Kernel Series, the student will understand the basic subsystems of the Linux Kernel, which will provide the necessary pre-requisites for successfully completing the Linux Kernel CNO section.
- Assessments:
 - Daily Quizzes
 - 15+ Labs
 - Final Assessment
 - Minimum score of 80%

Labs



- 15+ labs
 - Do not count towards pass/failure of course
 - All solutions will be posted
- We are here to help
- Lab Tips:
 - Beware of the invalid pointer
 - Use documentation resources
 - <https://elixir.bootlin.com>
 - <https://www.kernel.org/doc/html/latest>
 - kgdb is your friend



Quizzes and Final Assessment

- You are strongly encouraged to build, compile, and execute any code that might help answer questions
- Multiple choice questions...
- Quizzes do not count towards passing/failure of the course
 - But are recorded
- All conversations will be taken outside during assessments





Learning Objectives

Given a workstation, device, and/or technical documentation, the student will be able to:

- Describe the kernel's purpose and the responsibilities of its major subsystems
- Configure and compile the Linux kernel
- Cross compile the kernel for different architectures
- Dynamically debug the kernel using GDB and a virtual serial port via KGDB
- Develop a loadable kernel module to modify kernel functionality
- Create, register, unregister kernel devices
- Apply CNO Usermode practices to the Kernel

Series Agenda

Kernel Overview

Kernel Compiling and Debugging

Kernel Documentation

Kernel Modules

Kernel Components

Kernel Data Structures

Process Management

VII. Interrupt Handling

VIII. System Call Handling

IX. Driver Interaction

X. Memory Management

XI. Virtual Filesystem

XII. Sysfs

XIII. Network Subsystem

XIV. Security Modules

XV. CNO Kernel Practices



A Brief History of the Kernel

- “I’m doing a (free) operating system (just a hobby, won’t be big and professional like GNU) for 386 (486) AT clones. This has been brewing since April, and is starting to get ready. I’d like any feedback on things people like/dislike in MINIX, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things)...” Linus, 1991
- Initial version (0.01) had 10,239 lines of code
- Version 1.0 (1994) had 176,250 lines of code
- Version 2.4 (2001) had 2.4 million lines of code (by coincidence)
- Version 4.20 (2018) has around 20 million lines of code
 - A lot of recent versions have actually reduced the lines of code, dropping support for older hardware



Distributions

- The Linux kernel is bundled into an operating system package by various vendors
 - Window management engines, default configurations, application repositories, etc. may differ
- These bundles are referred to as Distributions (distros for short)
- Fedora is a community distribution sponsored by RedHat
 - We use it in this class
- While there are occasionally some slight differences in the base code or configuration of the kernel itself across different distributions, these differences are usually negligible
 - In fact, we'll be replacing the default kernel in Fedora with our own and everything else will work just fine



LINUX CNO Programming



Overview of the Kernel

What is a Kernel?

- A kernel is the memory-resident portion of the operating system
- The kernel is responsible for brokering access to various system resources
 - CPU time
 - Network devices
 - Storage devices
 - System memory
 - etc.



Responsibilities of the Kernel

- Architecture-dependent code
- Process scheduling
- Virtual memory management
- Handling hardware interrupts
- Implementing system calls
- File system support
- Networking support
- Interaction with any other hardware devices

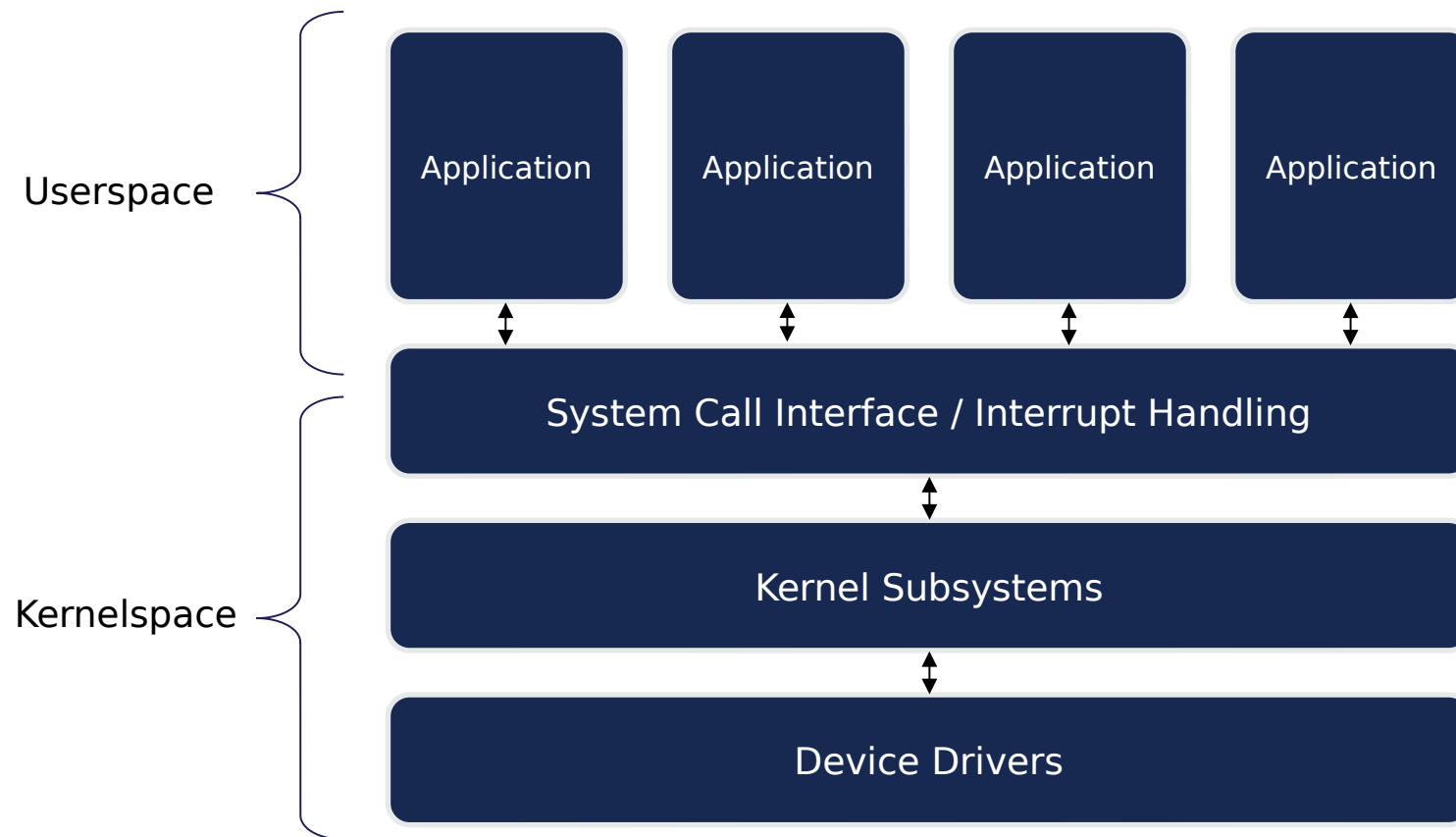


Without a Kernel

- System operation without an explicit kernel is possible, and historically some primitive systems did this
- Problems arise in this model
 - What happens if you need more than process running?
 - How do you make sure one process doesn't overwrite memory of another?
 - How do you make sure one process doesn't read the memory of another?
 - How do you make sure an application doesn't destroy your file system?
 - How do you make sure an application doesn't change the function pointers in the interrupt descriptor table?
 - Etc.

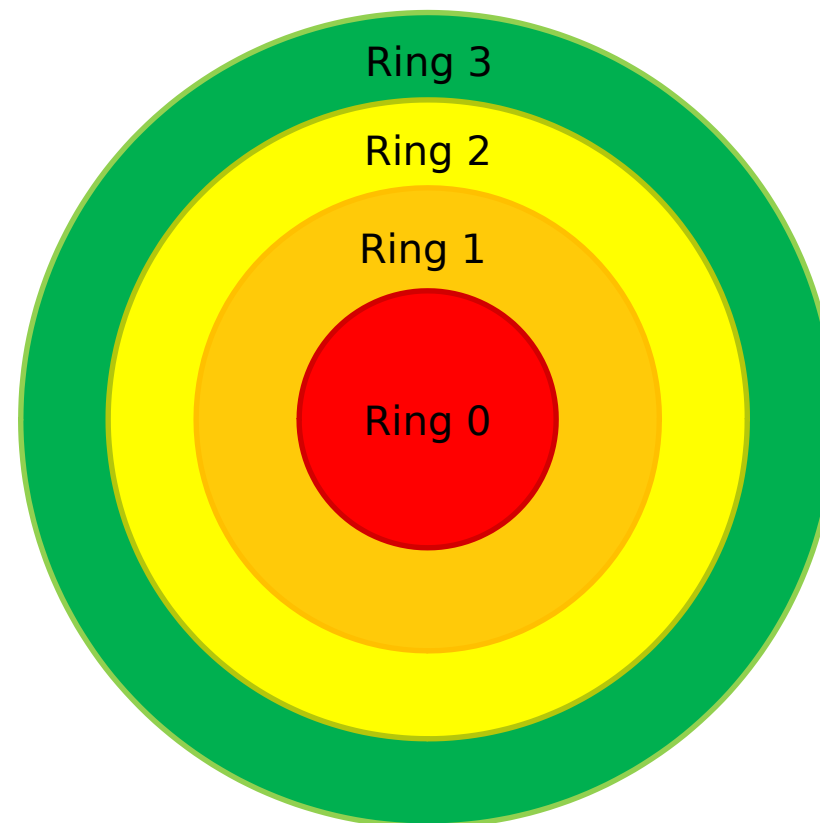


Userspace and Kernelpace



x86 Protection Rings

- Many CPU architectures support different modes
- The current mode of a CPU dictates which of the supported instructions are available for execution
- Under x86 (and x86_64) there are four modes, rings 0 through 3
- Ring 0 is the most privileged, and levels above that are progressively less privileged
- Linux only uses ring 0 (for kernel and drivers) and ring 3 (for applications)



x86 Protection Rings *(continued)*

- Each CPU (read: core) has its own ring level
- At ring 3, the CPU can
 - Use most instructions (mov, add, jmp, etc.)
 - Access unprivileged memory
- At ring 0, the CPU can
 - Do everything it can at ring 3, except in some special circumstances (like SGX)
 - Access privileged memory
 - Use special instructions



Switching Protection Levels (x86_64)



- Switching between ring 3 and 0 occurs when a program invokes a system call or when an interrupt occurs
- When a program executes a **syscall** instruction, the following occurs
 1. The address of the instruction following the **syscall** is placed into RCX (and various flags are saved)
 2. RIP is set to the address of the kernel's system call handler, provided by the OS during boot. This address is stored in the **LSTAR** register (a model-specific register)
 3. Ring level set to 0 (Current Protection Level (CPL) set to 0)
- When the system call is finished, the operating system can execute the **sysret** instruction
 1. RIP set to RCX, flags restored
 2. Ring level set to 3



Linux System Call Setup

- Linux uses the `wrmsrl` function (an architecture-dependent function which stands for write model specific register long) on boot to load the address of its system call handler into the LSTAR register
- Linux uses its `cpu_init` function to setup state for a CPU, defined in `arch/x86/kernel/cpu/common.c` (for x86)
- `cpu_init` calls `syscall_init`, which sets the LSTAR register with `wrmsrl(MSR_LSTAR, entry_SYSCALL_64)`
- For your continued reading: `linux/arch/x86/entry/entry_64.S` contains the definition of `entry_SYSCALL_64`



Switching Protection Levels

- Switching protection levels can also result through interrupts
- Interrupts can occur for a variety of reasons
 - User presses a key on the keyboard
 - Network device obtains data to parse
 - Program accesses memory that isn't paged in
 - Program performs a divide by zero
 - Etc.
- Unlike a **syscall** instruction, interrupts are not invoked directly by applications through a particular instruction
 - Instead, they occur indirectly through the use of instructions that cause exceptional conditions or completely asynchronously as the result of some external event



x86 Hardware Interrupt Handling



- When an interrupt occurs, it is identified by an interrupt vector number (of which there are 256, numbered 0 - 255)
- For hardware interrupts, the Advanced Programmable Interrupt Controller (APIC), part of the processor, receives notice from a hardware device needing service through a signal on a hardware line
- The APIC raises the interrupt line for a CPU that is not currently masking (ignoring) that interrupt.
- This causes the CPU to stop what is doing and handle the interrupt. When the CPU acknowledges the interrupt, the APIC will then send the interrupt vector number to the CPU
- The CPU checks the interrupt vector number to determine its next steps
- When it is done servicing an interrupt, the CPU informs the APIC that it has done so via the out instruction



x86 Interrupt Handling

- Every time an instruction finishes execution, the CPU checks to see if it has been notified of a hardware interrupt from the APIC
 - Unless the interrupt flag (IF) has been cleared with the **cli** instruction
 - If it has, the kernel can set it again with the **sti** instruction
- If an interrupt has occurred (and interrupts are not being ignored), the CPU saves some state on the stack and sets RIP to an address specified in the Interrupt Descriptor Table (IDT)
- The IDT is set by the kernel during boot, using the **lidt** instruction
 - The IDT is essentially a map from interrupt vectors to the addresses of functions that handle those interrupts (and other info like what ring level to use during handling)
 - A structure informing the CPU of the size and starting address of the IDT is the operand to the **lidt** instruction



Linux Interrupt Handling

- On boot, the kernel initializes a global variable called **idt_table** with the proper “gates” (we’ll get into this later) that specify the functions and settings for each interrupt handler
 - `idt_table` entries taken from **struct idt_data** inside `/arch/x86/kernel/idt.c`
 - See x86 interrupt enums in `/arch/x86/include/asm/traps.h`
- During **cpu_init**, the kernel calls **load_current_idt**, which calls **load_idt**, which in turn executes the **lidt** instruction
- When the kernel’s interrupt handlers are invoked they run in the ring level specified in the given interrupt’s entry in the IDT
- After an interrupt handler runs, it terminates in an **iret** instruction, which restores state for the code that was interrupted



Showing Your Current Ring Level

- The current protection level (CPL) is part of the CS (Code Segment) register
- You can see the value of this register in GDB (info regs)
- CPL = Lower 2 bits
- Write a C function that can return your current privilege/ring level



Privileged Instructions

- Code that executes within ring 0 is allowed to use special instructions that control how the system reacts to interrupts, exceptions
 - LIDT - Load Interrupt Descriptor Table Register
 - LLDT - Load local descriptor table
 - LGDT - Load Global Descriptor Table Register
 - LTR - Load Task Register
- And reading/writing machine specific registers: RDMSR, WRMSR
- And virtual machine opcodes: VMCALL, VMLAUNCH, VMRESUME, VMXON, VMXOFF
- And others
 - HLT - Halt,
 - INVLPG - Invalidate TBL entry,
 - SYSRET, SYSEXIT - Return from fast system call
 - WBINVD, and more



LINUX CNO Programming



Kernel Compiling and Debugging

Obtaining the Kernel Source

- You can obtain the kernel source in a variety of ways
 - (Ubuntu) `sudo apt-get install linux-source`
 - (Fedora) `dnf install kernel-devel`
 - **Get it from kernel.org**
 - Recode it from scratch, on the spot, blindfolded
- We have a tarball of the kernel version for this class located on your machines and the network share



Compiling the Kernel

- There are a variety of reasons to compile the kernel yourself
 - Enable debugging features
 - Add functionality
 - Change functionality
 - Building for a new architecture
- We will be replacing the default kernel in your virtual machine with one that we will be modifying
- Given how long it takes to compile the kernel, we will only do it twice during this course



Linus on Debugging



- “I happen to believe that not having a kernel debugger forces people to think about their problem on a different level than with a debugger. I think that without a debugger, you don't get into that mindset where you know how it behaves, and then you fix it from there. Without a debugger, you tend to think about problems another way. You want to understand things on a different *level*.”
- “It's partly ‘source vs binary’, but it's more than that. It's not that you have to look at the sources (of course you have to - and any good debugger will make that *easy*). It's that you have to look at the level *above* sources. At the meaning of things. Without a debugger, you basically have to go the next step: understand what the program does. Not just that particular line.”
- “I do realize that others disagree. And I'm not your Mom. You can use a kernel debugger if you want to, and I won't give you the cold shoulder because you have ‘sullied’ yourself. But I'm not going to help you use one, and I would frankly prefer people not to use kernel debuggers that much. So I don't make it part of the standard distribution, and if the existing debuggers aren't very well known, I won't shed a tear over it.”



Regardless...

- Linus makes some excellent points, from a *forward* engineering point of view
- While the kernel itself is open-source, some modules present on a target system may not be
- Linus merged in kgdb support to the mainline kernel in 2000 anyway:
 - “Another feature that is notable not for its size, but because people have tried to get me to merge it for some long is kgdb support. Which really turned out pretty small and clean, once people started putting their effort into making it so.”



Debug Environment

- You will be debugging your virtual machine's kernel from your physical machine
- To do this, we will be using VirtualBox, building a kernel with remote debugging support, and connecting the two machines using a virtual serial port
- By the end of the following demonstration and lab, you will be able to run GDB on your host machine to break within and remotely debug your VM's kernel



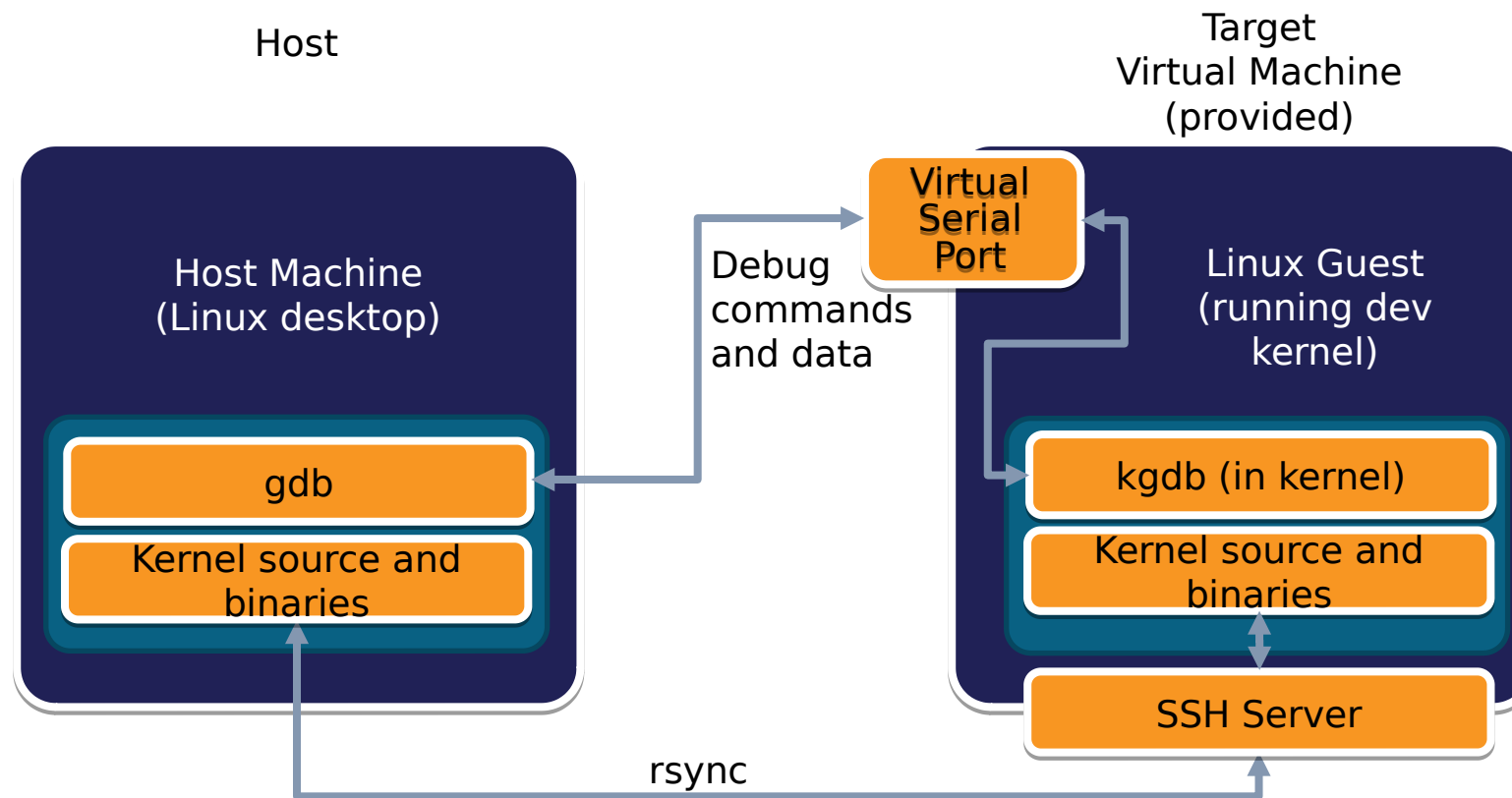
LINUX CNO Programming



Lab 1

Building the kernel and
Setting up Debugging

VM Debug environment



Symbols and Source

- The host needs to have access to the built kernel image (vmlinux) and source to provide accurate debug symbols and create breakpoints based on source code
 - The vmlinux kernel image being run on the target machine should be the same one that is loaded by gdb on the host machine
- You can use a variety of things to facilitate this
 - Shared folder functionality of your virtualization software
 - have the host and target share the same directory for these data
 - Note that on VirtualBox, vboxfs does not support symlinks, which the kernel build process makes use of
 - Use rsync to copy source and binaries between the host and target as needed
 - This is preferred for this class because vboxfs has numerous stability issues. However, feel free to use shared folders if you want



Adding a Virtual Serial Port

1. Shut down your target machine
2. Your host machine (running gdb) will communicate with your target machine using a virtual serial port connection
3. To enable this, add a serial port to your target machine
4. In VirtualBox Manager, click settings
5. Select Serial Ports and enable Port 1
6. Ensure that the Port Number is COM1, Port Mode is Host Pipe
7. Do **not** select the “Connect to existing pipe/socket” option
8. Make sure your target machine’s network connection is set to ‘bridged’



Adding a Virtual Serial Port

- Path/Address field:
 - For Linux hosts: /tmp/debug

Port 1 Port 2 Port 3 Port 4

☒ Enable Serial Port

Port Number: COM1 IRQ: 4 I/O Port: 0x3F8

Port Mode: Host Pipe

☐ Connect to existing pipe/socket

Path/Address: /tmp/debug

← Linux host



Get Kernel Source

1. Your host machine will need the following packages to build the Linux kernel (these are pre-installed):
 - Fedora: `sudo dnf install bison flex git fakeroot ncurses-devel openssl-devel elfutils elfutils-libelf-devel`
2. Download the kernel that you want to use (a tarball for the kernel used in the class will be provided)
 - or go to <https://www.kernel.org/> and download a compressed tarball of kernel source
3. Place the downloaded file into your kernel development directory (e.g., /home/student)



Configure the Kernel

1. From your **host machine**, extract your kernel tarball and change to the extracted directory
 - Example: `tar xvf linux-5.2.11.tar.xz`
`cd linux-5.2.11`
2. You can configure the kernel in one of two ways:
 1. Run `make config` to answer a series (an incredibly long one) of questions
 2. Copy your existing configuration and alter the configuration as you need, using `menuconfig`
 - `cp /boot/config-$(uname -r) .config`
`make menuconfig`
 - Alternatively, you may also use `xconfig` or `gconfig` instead of `menuconfig`



Enabling kgdb

- Some kernel versions or default configurations do not include support for a kernel debugger
- However, you can configure the kernel for dynamic debugging by enabling kgdb support in the configuration
- The succeeding slides show how to enable kgdb in the graphical menuconfig
- It is likely that these options are already enabled, but the following slides will assist you otherwise



Enabling kgdb (continued)



```
.config - Linux/x86 4.20.1 Kernel Configuration

Linux/x86 4.20.1 Kernel Configuration
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenu
----). Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes,
<M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for
Search. Legend: [*] built-in [ ] excluded <M> module < > module capable

*** Compiler: gcc (Ubuntu 4.8.4-2ubuntu1~14.04.4) 4.8.4 ***
General setup --->
[*] 64-bit kernel
Processor type and features --->
Power management and ACPI options --->
Bus options (PCI etc.) --->
Binary Emulations --->
Firmware Drivers --->
[*] Virtualization --->
General architecture-dependent options --->
[*] Enable loadable module support --->
[*] Enable the block layer --->
IO Schedulers --->
Executable file formats --->
Memory Management options --->
[*] Networking support --->
Device Drivers --->
File systems --->
Security options --->
-*- Cryptographic API --->
Library routines --->
Kernel hacking --->
```



Enabling kgdb (continued)



```
[ ] Warn for all uses of unseeded randomness
[ ] kobject debugging
[*] Verbose BUG() reporting (adds 70K)
[ ] Debug linked list manipulation
[ ] Debug priority linked list manipulation
[ ] Debug SG table operations
[ ] Debug notifier call chains
[ ] Debug credential management
    RCU Debugging --->
[ ] Force round-robin CPU selection for unbound work items
[ ] Force extended block device numbers and spread them
[ ] Enable CPU hotplug state control
<M> Notifier error injection
<M>   PM notifier error injection module
< >   Netdev notifier error injection module
[ ] Fault-injection framework
[*] Latency measuring infrastructure
[*] Tracers --->
[ ] Remote debugging over FireWire early on boot
[ ] Enable debugging of DMA-API usage
[*] Runtime Testing --->
[*] Memtest
[ ] Trigger a BUG when data corruption is detected
[ ] Sample kernel code ----
[*] KGDB: kernel debugger --->
[ ] Undefined behaviour sanity checker
[*] Filter access to /dev/mem
[ ]   Filter I/O access to /dev/mem
[ ] Enable verbose x86 bootup info messages
```

**You may
have to
scroll down a
bit to find
this**



Enabling kgdb (continued)



```
.config - Linux/x86 4.20.1 Kernel Configuration
> Kernel hacking > KGDB: kernel debugger
    KGDB: kernel debugger
    Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenu
    ----). Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes,
    <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for
    Search. Legend: [*] built-in [ ] excluded <M> module < > module capable

    --- KGDB: kernel debugger
    --> <*> KGDB: use kgdb over the serial console
    [ ] KGDB: internal test suite
    --> [*] KGDB: Allow debugging with traps in notifiers
    --> [*] KGDB_KDB: include kdb frontend for kgdb
    --> (0x1) KDB: Select kdb command functions to be enabled by default
    --> [*] KGDB_KDB: keyboard as input device
    (0) KDB: continue after catastrophic errors
```



Compiling the Kernel

- From your host machine, run make to compile. You can split this job across your available CPU cores with the -j option.
 - make -j <num threads>

Notes:

- there is a pre-built/pre-configured 'Kernel Class Backup' VM in Virtual Box
- the remaining slides for lab 1 describe the steps required to configure a base VM into what would mirror the 'Kernel Class Backup'
- to save a lot of time, instructors will ask you to cancel the make process, review the remaining slides for lab 1, and prepare to use the 'Kernel Class Backup' for the remainder of the module
- you will be given the opportunity to implement the remaining steps and rebuild your Kernel Class Backup during lab 7

ManTech



Compiling the Kernel

- Enable an SSH server on your target machine (maybe done already)
 - Fedora: `systemctl enable sshd && systemctl start sshd`
- Copy your `linux-5.2.11` directory from your host to your target using `rsync` (this will likely take ~15 minutes)
 - `rsync -av <path_to_linux-5.2.11_dir> student@<vm_ip>:~`
- To install the kernel and its modules (on target machine):
 - `cd <path_to_linux-5.2.11_dir>`
 - `sudo make modules_install -j <num threads>`
 - `sudo make install -j <num threads>`



Activating the new Kernel

1. Update the bootloader entries (on your target machine)
 - Ubuntu: `sudo update-grub`
 - Fedora:
 - BIOS: `sudo grub2-mkconfig -o /boot/grub2/grub.cfg`
 - **UEFI:** `sudo grub2-mkconfig -o /boot/efi/EFI/fedora/grub.cfg`
2. If you aren't planning on debugging, you're done!
3. Continue if you plan to use gdb on your host to debug the kernel on your target



Editing the bootloader (on your target machine)

- Some special parameters need to be passed to the Linux kernel to enable kgdb debugging
- Fedora and other modern distributions make this easy
 - Edit the `/etc/sysconfig/grub` file (as a sudoer)
 - Add the following parameters to the `GRUB_CMDLINE_LINUX` line:
`nokaslr nopti kgdboc=ttyS0,115200 kgdbwait`
- When you're done, update the GRUB bootloader using the appropriate command for your system setup:
 - **BIOS:** `sudo grub2-mkconfig -o /boot/grub2/grub.cfg`
 - **UEFI:** `sudo grub2-mkconfig -o /boot/efi/EFI/fedora/grub.cfg`



Editing the bootloader *(continued)*

- Your new menu entry should look similar to this:

```
GRUB_TIMEOUT=5
GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="resume=/dev/mapper/fedora-swap rd.lvm.lv=fedora/root rd.lvm.lv=fedora/swap rhgb
nokaslr nopti kgdboc=ttyS0,115200 kgdbwait"
GRUB_DISABLE_RECOVERY="true"
```

- nokaslr** disables kernel ASLR, to make our debug symbols accurate during kernel runtime
- nopti** disables page-table isolation
- kgdboc** tells kernel to communicate over ttyS0 (serial 0) for debug
- kgdbwait** tells the kernel to halt during boot and wait for the host system to attach a debugger



Testing the Debug Setup

- Reboot the target machine
- The kernel should pause before loading, waiting for you to connect the debugger from the host machine
- On the host machine cd to the Linux source directory
- Load the kernel image with GDB using **`gdb ./vmlinux`**



Testing the Debug Setup *(continued)*

- In another terminal on the host machine, run the following command to connect a PTY to the virtual serial port created earlier:
 - **socat -d -d /tmp/debug pty**
 - Let this run for as long as you are debugging
- The socat output should indicate which PTY is connected to the virtual serial port (e.g., "PTY is /dev/pts/2")
- In the terminal running gdb, connect to the target kernel
 - **target remote <pty path>**
 - Example: **target remote /dev/pts/2**



Testing the Debug Setup *(continued)*

- GDB should connect to the target machine and you should see something similar to the following

```
Source
1068 noline void kgdb_breakpoint(void)
1069 {
1070     atomic_inc(&kgdb_setting_breakpoint);
1071     wmb(); /* Sync point before breakpoint */
1072     arch_kgdb_breakpoint();
1073     wmb(); /* Sync point after breakpoint */
1074     atomic_dec(&kgdb_setting_breakpoint);
1075 }
1076 EXPORT_SYMBOL_GPL(kgdb_breakpoint);
1077
1078 static int __init opt_kgdb_wait(char *str)

Stack
[0] from 0xffffffff81165f64 in kgdb_breakpoint+20 at kernel/debug/debug_core.c:1073
(no arguments)
[1] from 0xffffffff81166f31 in kgdb_initial_breakpoint at kernel/debug/debug_core.c:974
(no arguments)
[+]

>>> □
```



Testing the Debug Setup *(continued)*

- Typing “c” or “continue” into the GDB terminal should allow the target’s kernel to boot
- If you don’t set any breakpoints before this point, you will not be able to regain control of the kernel via GDB until the target kernel breaks itself
- You can cause the kernel to break back to the debugger by issuing the following command in a terminal from the **target machine**
 - `echo g | sudo tee /proc/sysrq-trigger`
 - Continue again from gdb using c



LINUX CNO Programming



Kernel Documentation

Kernel Source Tree

- The Linux kernel source is divided into several directories for organizational purposes
- Each directory contains code for a particular portion of the kernel
- This structure has remained relatively unchanged since version 2.6 of the kernel
 - Reorganization of subdirectories within the top-level ones is more common



Notable Kernel Source Directories

| DIRECTORY | DESCRIPTION |
|-----------|---|
| arch | Architecture-specific code (assembly portions of the kernel). Contains a subdirectory for each supported CPU architecture |
| block | Code for managing block I/O devices |
| drivers | Code for other devices |
| fs | Virtual Filesystem (VFS) layer and native filesystem code |
| include | Header files |
| kernel | Critical kernel components (scheduling, locking, process management) |
| mm | Memory management code, virtual memory support |
| net | Network stack (sockets, routing, IP, TCP, UDP, etc.) |
| security | Linux Security Module (LSM) framework code (with native modules such as SELinux, AppArmor, Tomoyo, etc.) |



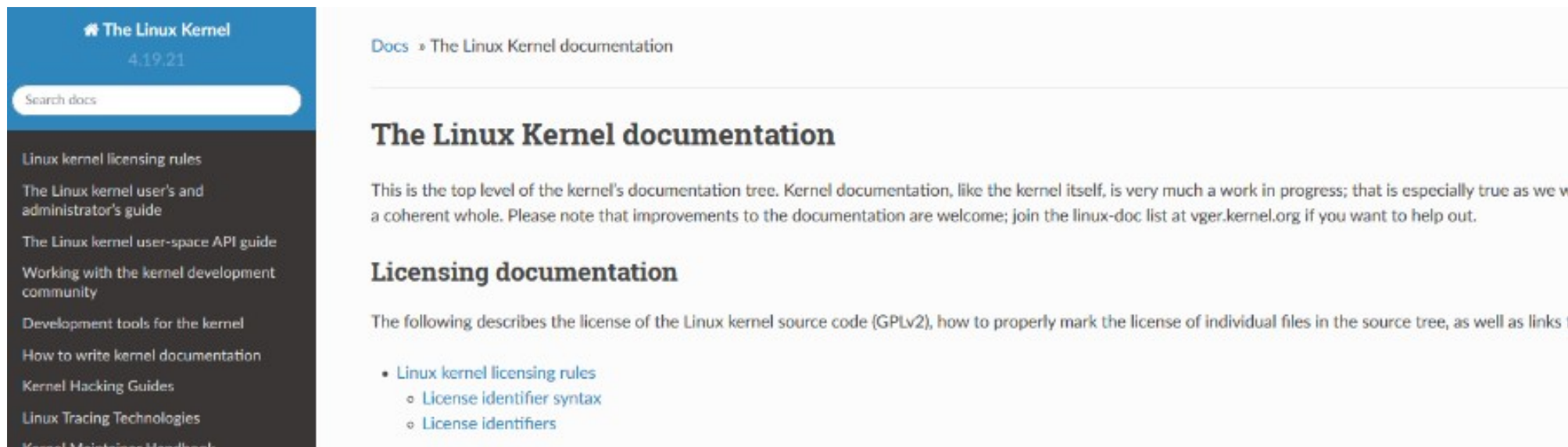
Sources of Documentation

- The Linux kernel is fairly complex
- Documentation is your best friend and comes in the following forms
 - The Documentation directory in the kernel source
 - ReStructuredText HTML docs (generated from source package)
 - Online: <https://www.kernel.org/doc/html/latest/>
 - LWN.net articles
 - The code itself
 - Visual cross references will likely be of more use than grep
 - But grep powers are not to be underestimated



Creating a local htmldoc site

- You can build the kernel's ResStreucturedText HTML documentation yourself
 - Dependencies: `sphinx (dnf install python2-sphinx)`
 - Create the docs (`make htmldocs`)
 - HTML files will be located in `Documentation/output` after creation
 - Open `Documentation/output/index.html` in your browser

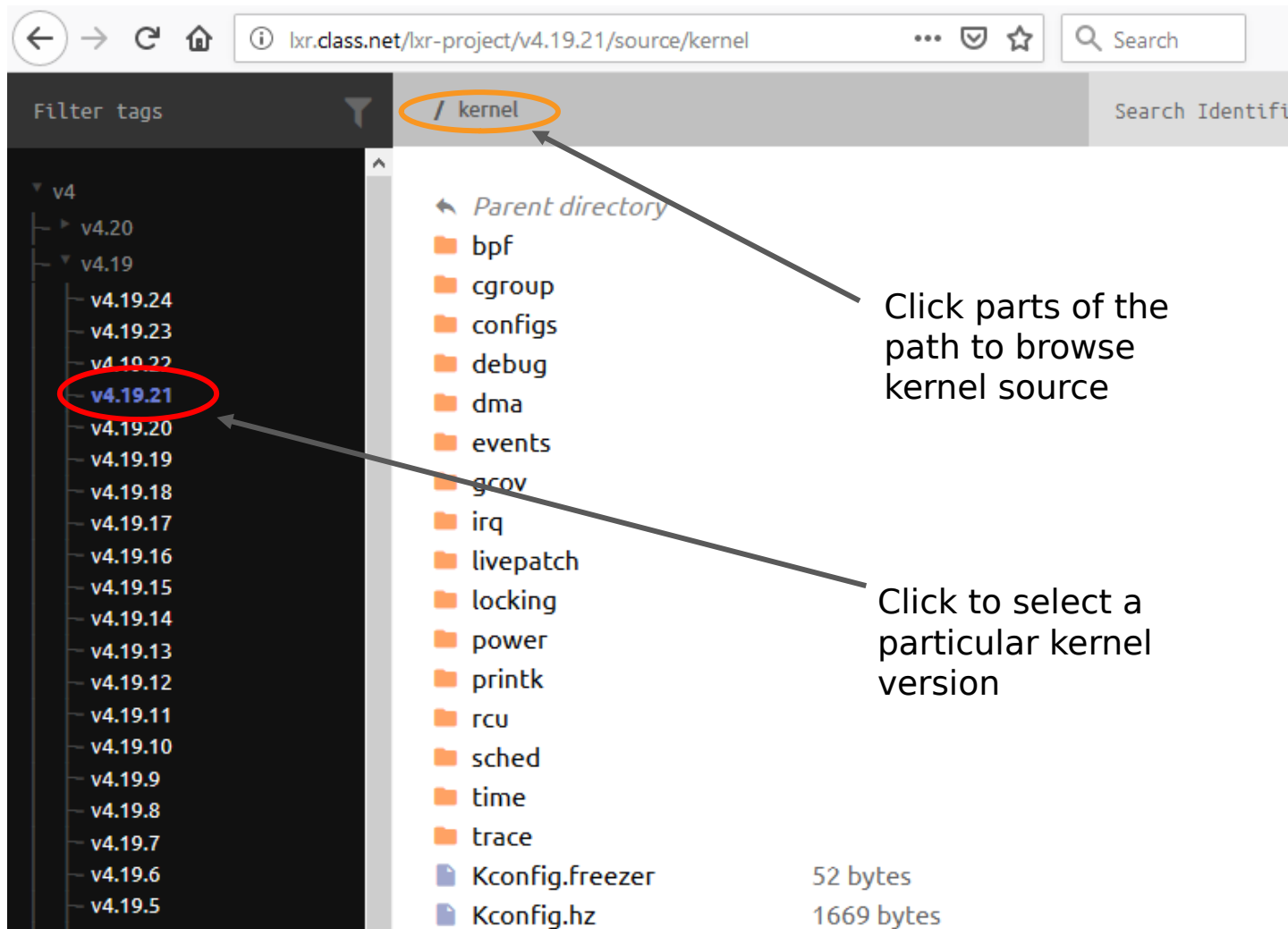


Linux Cross Referencer

- Use the power of open source!
- The Linux Cross Referencer (LXR) is freely available
 - <https://lxr.sourceforge.io/en/index.php>
- Elixir, inspired by LXR, is better and also free
 - <https://elixir.bootlin.com/>
- You are allowed (and encouraged!) to use Elixir during classroom assignments and tests



Using Elixir



lxr.class.net/lxr-project/v4.19.21/source/kernel

Filter tags

/ kernel

Search

Search Identifi

Parent directory

- bpf
- cgroup
- configs
- debug
- dma
- events
- gcov
- irq
- livepatch
- locking
- power
- printk
- rcu
- sched
- time
- trace
- Kconfig.freezer 52 bytes
- Kconfig.hz 1669 bytes

Click parts of the path to browse kernel source

Click to select a particular kernel version



Using Elixir (continued)



The screenshot shows the lxr.class.net website with the search bar at the top right containing 'tcp_recvmsg'. The search results are displayed in the main content area, showing files where the symbol is defined and referenced. The left sidebar shows a tree view of the kernel source code, with 'v4.19.21' selected. The search results are as follows:

Defined in 3 files:

- include/net/tcp.h, line 405 (as a prototype)
- net/ipv4/tcp.c, line 1915 (as a function)
- net/ipv4/tcp.c, line 2177 (as a variable)

Referenced in 5 files:

- include/net/tcp.h, line 405
- kernel/bpf/sockmap.c
 - line 937
 - line 1016
 - line 1031
- net/ipv4/tcp.c
 - line 1915
 - line 2177
- net/ipv4/tcp_ipv4.c, line 2462
- net/ipv6/tcp_ipv6.c, line 1962

Annotations on the screenshot:

- 1. Search for a symbol here (function or variable) name (points to the search bar)
- 2. Click one of these to see the definition of that symbol (any declaration gets listed) (points to the 'Defined in 3 files' section)
- 3. Click one of these to navigate to that particular file and line (points to the 'Referenced in 5 files' section)



Using Elixir (continued)



lrx.class.net/lxr-project/v4.19.21/source/net/ipv4/t

Filter tags

/ net / ipv4 / tcp.c

Search Identifier

1914
1915 `int tcp_recvmg(struct sock *sk, struct msghdr *msg, size_t len, int nonblock,`
1916 `int flags, int *addr_len)`
1917 `{`
1918 `struct tcp_sock *tp = tcp_sk(sk);`
1919 `int copied = 0;`
1920 `u32 peek_seq;`
1921 `u32 *seq;`
1922 `unsigned long used;`
1923 `int err, ing;`
1924 `int target;`
1925 `long timeo;`
1926 `struct sk_buff *skb, *last;`
1927 `u32 urg_hole = 0;`
1928 `struct scm_timestamping tss;`
1929 `bool has_tss = false;`
1930 `bool has_cmsg;`
1931
1932 `if (unlikely(flags & MSG_ERRQUEUE))`
1933 `return inet_rcv_error(sk, msg, len, addr_len);`
1934
1935 `if (sk_can_busy_loop(sk) && skb_queue_empty(&sk->sk_receive_queue) &&`
1936 `(sk->sk_state == TCP_ESTABLISHED))`
1937 `sk_busy_loop(sk, nonblock);`
1938
1939 `lock_sock(sk);`
1940

Clicking a symbol link will take you to the corresponding file and line

Other symbol names in the file can be clicked to perform a lookup on them



Elixir Limitations

- Some symbols will not appear in the search
 - Those defined/used in assembly (Elixir does not parse asm)
 - Those which are defined through macro expansion (such as system calls)
- Cross references will not work on certain symbol uses
 - Function pointer assignments (function symbol names will not be matched to the name of the pointer to which they are assigned)
- Use `grep -rn` and other tools



LINUX CNO Programming



Kernel Modules

Kernel Modules

- The kernel provides the ability for modules to be inserted into it during runtime, to add additional functionality
- Device drivers and other services are implemented as modules
- Some modules are part of the kernel mainline (see drivers directory)
- Others are third party (some proprietary, some not)
- Modules are object code with specific entry points for insertion and removal
 - Kernel Object (**.ko**) extension
- You can insert a module using **insmod** and remove it using **rmmod**
- View what modules are currently loaded on the system with **lsmod**
- View individual module information using **modinfo**



Types of Kernel Modules

- Kernel modules for device drivers are typically organized into one of three categories
 - These aren't enforced and the lines between them can sometimes be blurred
 - There is nothing stopping a kernel module from being all of these, or none of them
- Character devices
- Block devices
- Network interfaces



Character Devices

- Devices that can be represented as a stream of bytes to applications
- Reading and writing can be performed on a per-byte (per-character) basis
 - Sometimes called char devices
- Accessed through file system nodes (e.g., /dev/ttyX)
- Acts like a file, implements open, read, write for interaction with applications
- Example: Driver for a console



Block Devices

- Performs I/O operations on chunks of data at a time, called blocks
- Blocks are typically sets of bytes with a size of a power of two (512, 4096)
- Linux allows block devices to be accessed as a stream of bytes by applications, so they are functionally similar to character devices from the standpoint of an application
 - The interface to the kernel, however, must use I/O operations at the granularity of a full block
- Accessed through /dev (e.g., /dev/sda, /dev/nvme1p1)
- Examples: Driver for disk drive



Network & Other Devices

- Network devices
 - Not accessed via the file system
 - Provide network interfaces instead (eth0, enp2s0)
 - Facilitates the transmission and reception of data packets
 - Implement a backend for kernel requests for sending and receiving data
- Drivers don't have to fall within these categories
 - Any peripheral hardware module also needs a driver (USB, serial, SCSI, etc.)



Other Uses of Modules

- Kernel modules don't have to implement drivers
- Modules can extend kernel functionality
 - Provide support for a new type of filesystem
 - Provide support for a new network protocol
 - Provide a new algorithm for scheduling certain types of processes



Kernel Module Installation

- Kernel modules can be installed at runtime with the `init_module` system call (see `man 2 init_module`)
 - Use the `delete_module` system call to unload a module
- The command-line utilities, `insmod` (insert module) and `rmmod` (remove module) can perform these operations on behalf of a user
 - Example: `insmod my_module.ko`
- List currently loaded modules with `lsmod`
- The kernel will also automatically load any module placed in `/lib/modules/<version>/drivers` and referenced in a configuration file in `/lib/modules-load.d`



Creating a Kernel Module

- Kernel modules require the declaration of at least two functions: a load routine and an unload routine
- The **module_init** (called on load) and **module_exit** (called on unload) macros are used to identify these functions to the kernel
- The MODULE_LICENSE macro tells the kernel what type of license your module carries
 - Some kernels are configured to deny the insertion of any module not declared to carry a license of type “GPL”
 - Non-GPL modules only have access to a **limited** set of exported kernel symbols



Hello World Module

```
#include <linux/module.h>

static int __init mod_start(void) {
    printk(KERN_INFO "Hello, World\n");
    return 0;
}

static void __exit mod_stop(void) {
    printk(KERN_INFO "Goodbye, World\n");
    return;
}

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Mark O'Neill");
MODULE_DESCRIPTION("Hello World Demo");
module_init(mod_start);
module_exit(mod_stop);
```

The `__init` and `__exit` decorators tell the kernel that it can discard these functions if they won't be used (in the case of a built-in module) or after use (after init)

See other license options in `/include/linux/module.h`

Tell the kernel which functions are your init and exit



Module Parameters

- When inserting a kernel module, you can provide it command line arguments:
 - Example: `insmod mymodule.ko mystr="bob" myint=4`
- Parameters are simply placed into global static variables within your module
 - Declare a global static variable for each parameter you want
 - The default value for your parameter will be whatever you initialize it to in the declaration
- To allow your module to handle a parameter, use the **`module_param`** or **`module_param_named`** functions for each parameter you want to create
- You can also provide a help for each parameter using **`MODULE_PARM_DESC`**



Kernel Module Makefile

- The kernel build system makes your life easy
- If you're building within the kernel source you only need a few lines in your Makefile

- To compile hello.c into hello.ko:

```
obj-m := hello.o
```

- To compile multiple source files into my_module.ko:

```
obj-m := my_module.o
```

```
my_module-objs := source_1.c source_2.c
```



Kernel Module Makefile *(continued)*

- To compile outside of the kernel source directory, we need only to add a few more lines
- We define targets for clean and all, directing make to the current kernel's module build directory

```
obj-m := hello.o  
  
all:  
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules  
  
clean:  
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```



Module Load Process

- When a module is inserted into the kernel the following steps (among others) occur:
 - The system call **sys_init_module** is invoked
 - The module code is copied into memory
 - The module's license is checked
 - The kernel symbols used by the module are looked up in the kernel symbol table and resolved
 - The module's init function is invoked
- See `/kernel/module.c` for more details



Accessing Kernel Symbols

- Modules do not have (direct) access to every kernel symbol
- If your module needs to use symbols from the kernel or another module, that symbol must be exported
- The kernel headers have two macros for exporting symbols
 - EXPORT_SYMBOL - export this symbol to all modules
 - EXPORT_SYMBOL_GPL - export this symbol to only GPL modules
- An easy way to find exported symbols is to find them in /proc/kallsyms
 - Example: `cat /proc/kallsyms | grep tcp_recvmmsg`
- In the kernel source, exported functions typically locate their EXPORT_SYMBOL immediately following their definition



LINUX CNO Programming



Lab 2

Basic Kernel Module

Tasks

- On your target VM, create a Makefile for a module for use outside of the kernel source directory
- Create a C source file for your module
- Define init and exit functions
- Set the module author and description to values of your choosing
- Set the module license to “GPL”
- Define a string and integer parameter
- Your init function should use **printk** to print out the string passed as an argument, or the string “Hello World” if no string parameter is provided
- Your init function should also use `printk` to display the current ring level of the CPU, if the integer parameter is 1 (and this should default to 0)
- Your exit function should print out “Module Exiting”



Tasks *(continued)*

- Compile your module
- Insert your module into your kernel using **insmod**
- Use `journalctl -f` or **dmesg** to view printouts from your module
- Remove your module using **rmmod**
- Play with different levels for the `printk` calls. What do you notice about them in the log?



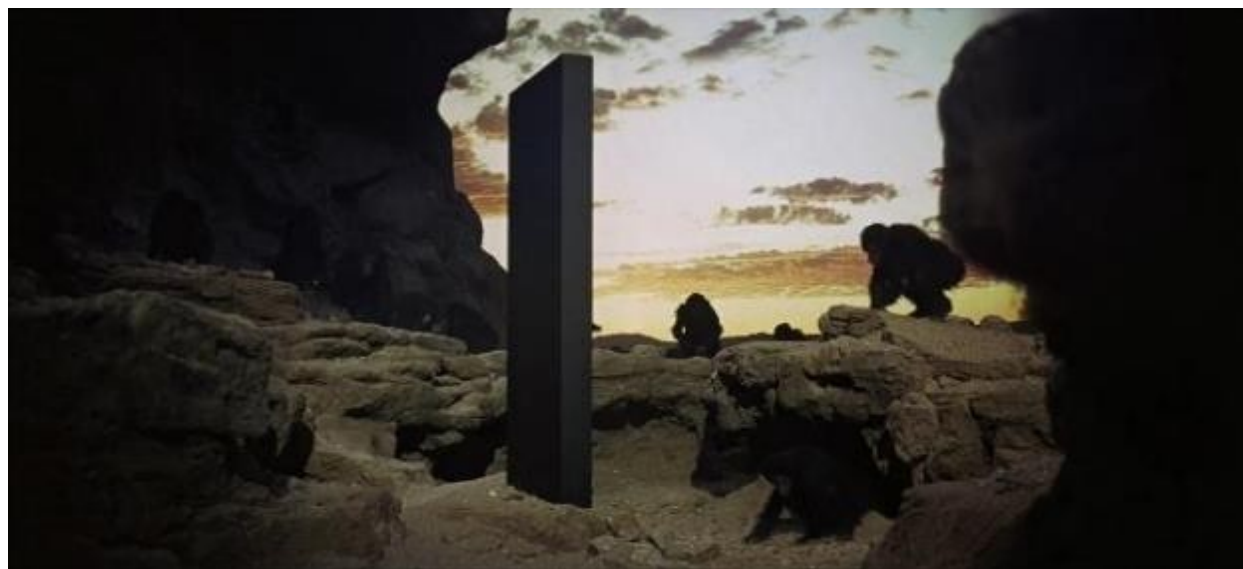
LINUX CNO Programming



Kernel Components

Monolithic Kernel

- Linux is a monolithic kernel, which means that all components of the kernel (including drivers) run in the same and highest privilege level of the CPU (ring 0)

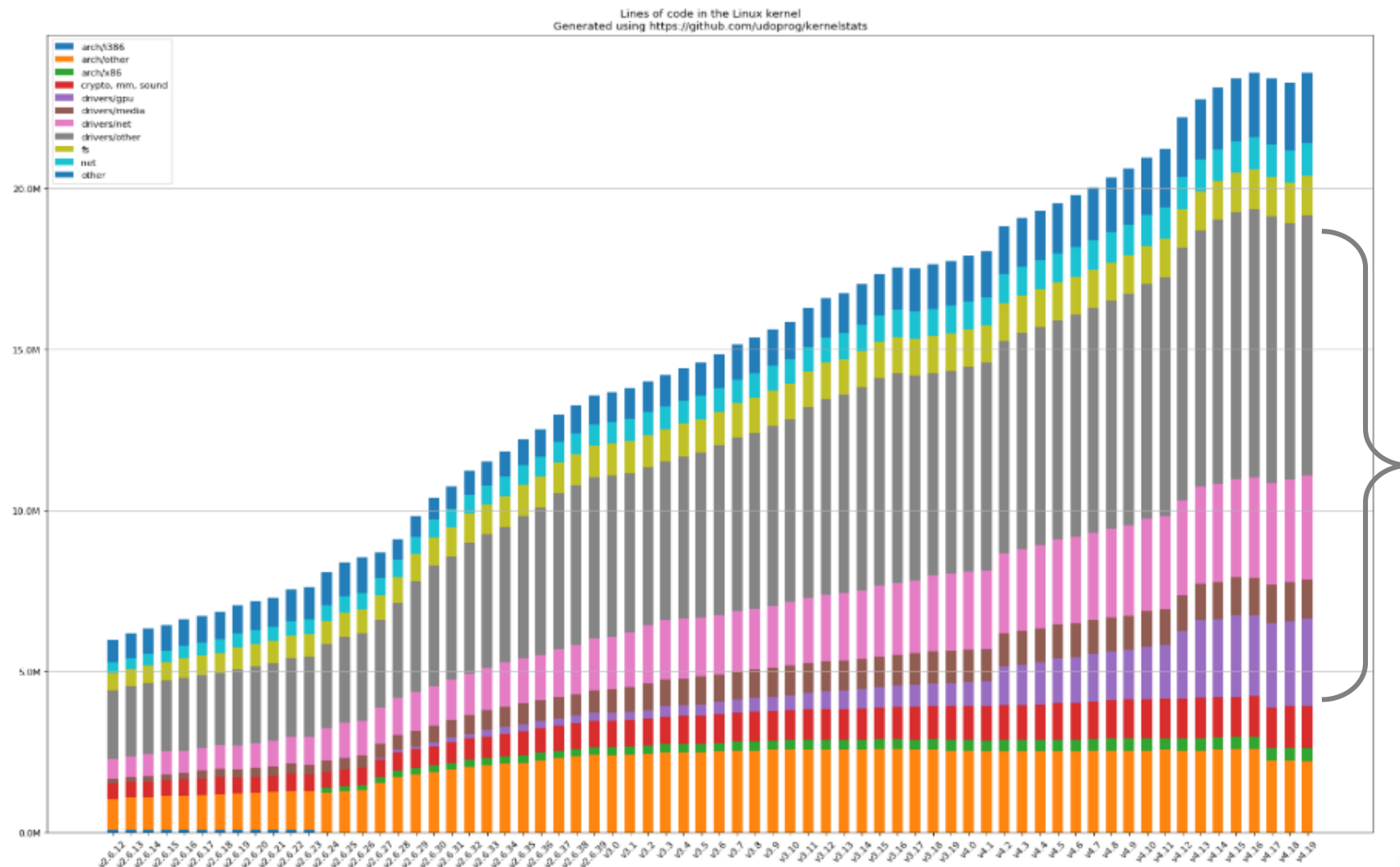


Major Kernel Components

- The kernel has over 28+ million lines of code, but the overwhelming majority are for driver code
- The primary components of the kernel make up the rest

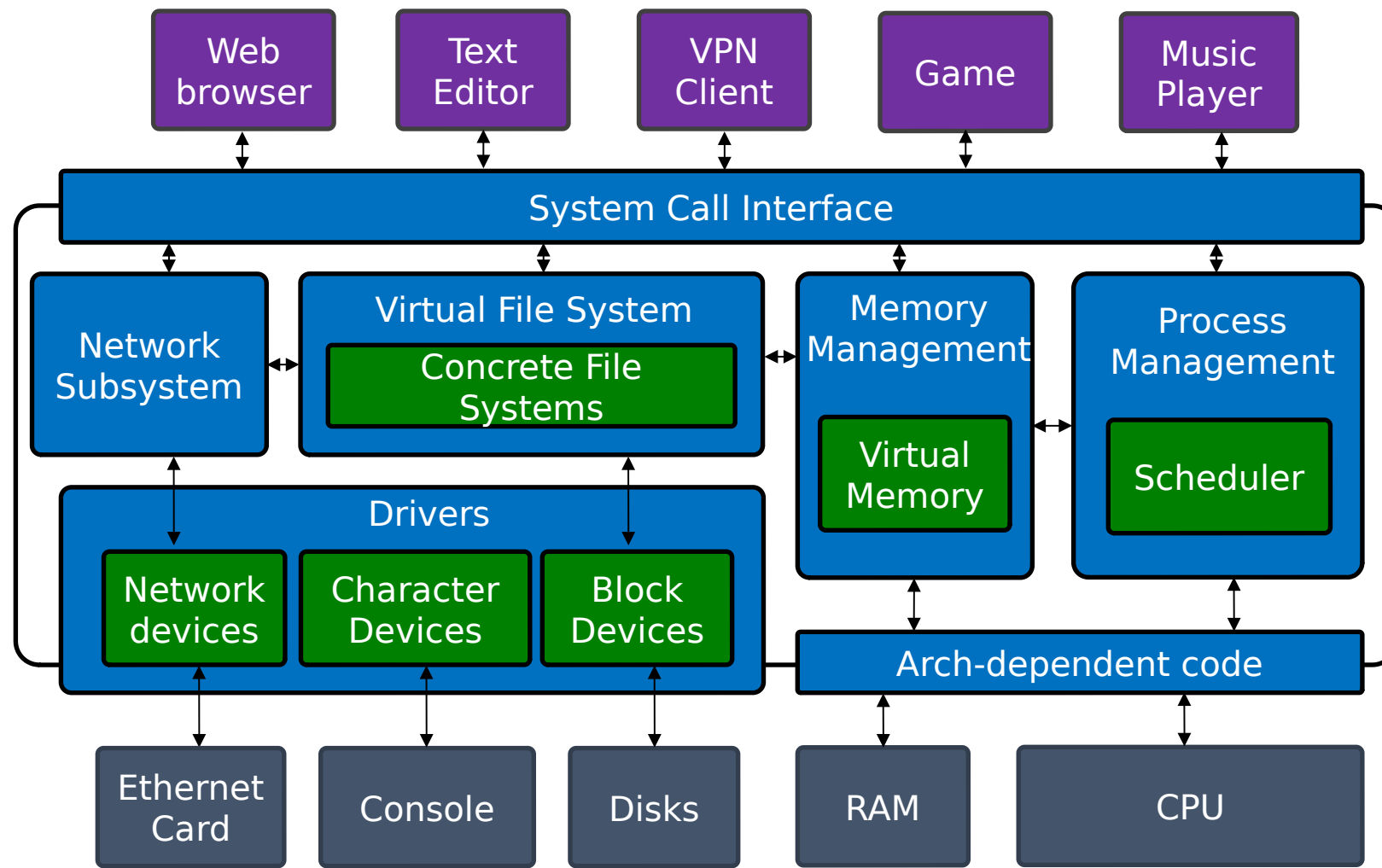


Kernel LOC Breakdown



ManTech

Major Components



Entry Points

- The kernel isn't a process – it's a collection of reactive services that work together
- To enter the kernel (ring 0) an interrupt or system call is required
 - Previously system calls used to use interrupts as well (int 0x80)
- The kernel invokes its services in response to these events
 - System calls are handled in the context of the process that invoked it
 - Interrupts literally interrupt the running process (or kernel code) to perform required processing
- The scheduler can get invoked explicitly by other kernel functions or invoked in response to timer interrupts
- See `/arch/x86/entry/` and `/arch/x86/kernel/idt.c`



Virtual File System

- The Virtual File System (VFS) layer provides a unified interface for file system access and manipulation by the kernel
- Structures and functions that create, open, read, and write dirents, inodes, and file data are contained within this subsystem
- File system drivers implement the VFS interface to allow the kernel to use foreign filesystems without making it aware of the specific details about how that filesystem works under the hood
- See the /fs directory in the kernel source



Network Subsystem

- The network subsystem is provided to allow application developers easy access to networking protocols
- The Linux kernel supports IPv4, IPv6, Bluetooth, NFC, TCP, UDP, and a host of other protocols by default
- The network subsystem provides the kernel handling of internal buffers, parsing, events, error checking, packet construction, etc. for all supported network protocols
- When a programmer uses the POSIX socket API, the system calls invoked as result all pass through this subsystem for handling
- See the /net directory in the kernel source



Memory Management

- The kernel manages memory for itself and all the applications and drivers that run on the system
- Working in tandem with the CPU, the kernel supports the virtual memory paradigm given to all applications
- The kernel is responsible for handling a variety of memory needs
 - Allocation of new memory
 - Intelligent sharing of memory
 - Swapping memory to disk and back
 - Providing fast and efficient memory allocation APIs
 - Keeping track of memory so it isn't leaked



Devices and Drivers

- Different physical systems all need a kernel, but they all have different hardware
- The kernel can run in a host of environments, from small embedded devices to home computers to massive supercomputers
 - Each of these have different peripherals and busses to which data needs to be sent and received
- The kernel supports the addition of modules (both statically and dynamically) to add support for different types of hardware and other functionality
- Most lines of code in the kernel mainline are actually for drivers



Ftrace

- Ftrace is a system that allows developers to view control flow through specific kernel functions
- It provides an interface in `/sys/kernel/debug/tracing` to perform various logging tasks
- View available tracers
 - `cat /sys/kernel/debug/tracing/available_tracers`
- View current tracer
 - `cat /sys/kernel/debug/tracing/current_tracer`
- Change current tracer
 - `echo function > /sys/kernel/debug/tracing/current_tracer`



Ftrace Support

- To use Ftrace, The following options should be set in your kernel configuration when you compile
 - CONFIG_FUNCTION_TRACER
 - CONFIG_FUNCTION_GRAPH_TRACER
 - CONFIG_STACK_TRACER
 - **CONFIG_DYNAMIC_FTRACE**
- During compilation the function calls for Ftrace have their locations recorded
- These locations are set to NOPs at boot time and converted to their call instructions when Ftrace is enabled during runtime
- This allows the kernel to dynamically enable and disable Ftrace, and incur no overhead when it is disabled



trace-cmd

- The utility **trace-cmd** provides an easier-to-use front-end for the sysfs interface
- Install it with `sudo dnf install trace-cmd`
 - Run it as a privileged user too
- Run `trace-cmd` as a privileged user (or use `sudo`)
- To monitor all invocations of function `myfunc`, run
 - `trace-cmd record -p function -l myfunc`
- Not all functions can be traced
 - `trace-cmd list` presents a list of available symbols



Ftrace Features

- To view the report generated by a **trace-cmd record**, use **trace-cmd report**
- The log file (trace.dat) created by trace-cmd record can get extremely large very quickly (popular kernel functions can get called thousands of times per second)
- When recording, you can constrain the tracing to function calls caused by a particular process ID using the -P option
 - trace-cmd record -p function -l myfunc -P 5213
- You can also see a full call-graph by using the -p function_graph tracer instead of function



Ftrace Features

- Some things worth tracking don't result from the invocation of system calls
- Some things worth tracking happen periodically, or are invoked by numerous execution paths
- Ftrace can also monitor for events to handle these cases
 - Example: Monitor context switches
- There are numerous, but finite events that you can track
 - See: `cat /sys/kernel/debug/tracing/available_events` or `trace-cmd list -e`



Ftrace Reports

- You can view Ftrace reports with trace-cmd using trace-cmd report
- Sample output from a few seconds of ncat:

```
CPU 0 is empty  
cpus=2
```

```
<...>-22424 [001] 102484.279433: function:      switch_mm_irqs_off  
<...>-22424 [001] 102484.279434: function:      load_new_mm_cr3  
ncat-22377 [001] 102484.279436: function:      finish_task_switch  
ncat-22377 [001] 102484.279437: function:      __fdget  
ncat-22377 [001] 102484.279437: function:      __fget_light  
ncat-22377 [001] 102484.279439: function:      sock_poll  
ncat-22377 [001] 102484.279439: function:      tcp_poll  
ncat-22377 [001] 102484.279439: function:      __fdget  
ncat-22377 [001] 102484.279439: function:      __fget_light  
ncat-22377 [001] 102484.279439: function:      sock_poll  
ncat-22377 [001] 102484.279440: function:      tcp_poll  
ncat-22377 [001] 102484.279440: function:      _cond_resched  
ncat-22377 [001] 102484.279440: function:      rcu_all_qs
```



LINUX CNO Programming



Lab 3

Going Deeper

Tasks

- Individually or in groups of two, use simple programs, trace-cmd, the Linux documentation, the Linux source code, and other resources to determine the name of the kernel functions that:
 - Handle the reception of TCP messages
 - Open a file for reading
 - Send a signal to a process
 - Load an ELF into memory for execution as a process



LINUX CNO Programming



Kernel Data Structures

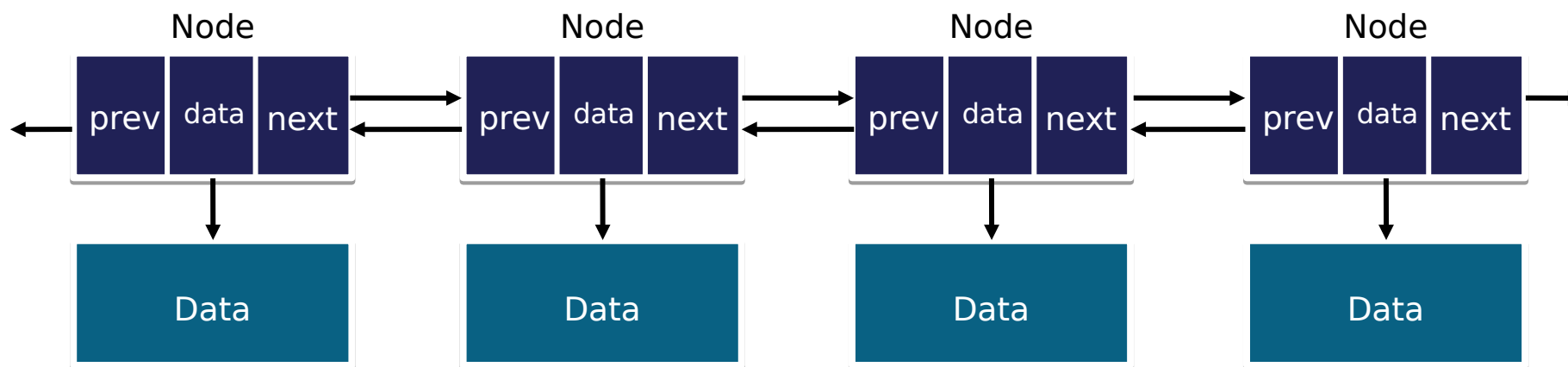
Abstract Data Types in the Kernel

- Nearly all subsystems in the kernel rely on traditional data structures such as
 - The kernel provides these in a special header-only manner
 - Makes the data structures general without sacrificing performance
 - Often faster due to less dereferencing and better cache usage



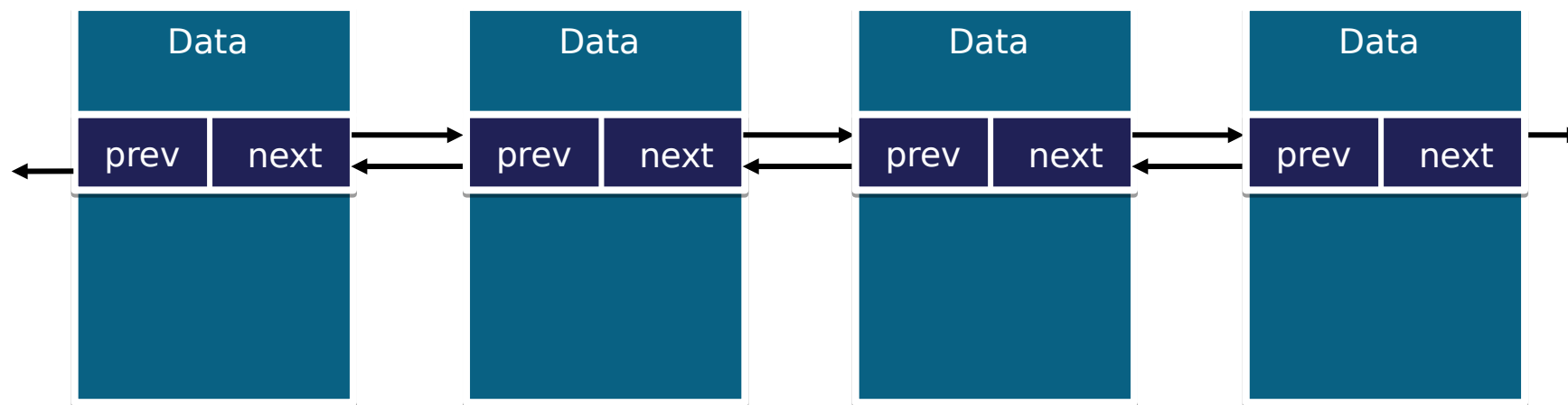
Linked Lists

- Traditional linked lists:
 - Have a node structure
 - Have functions that operate on node structures
 - Embed data or pointer to data in the node



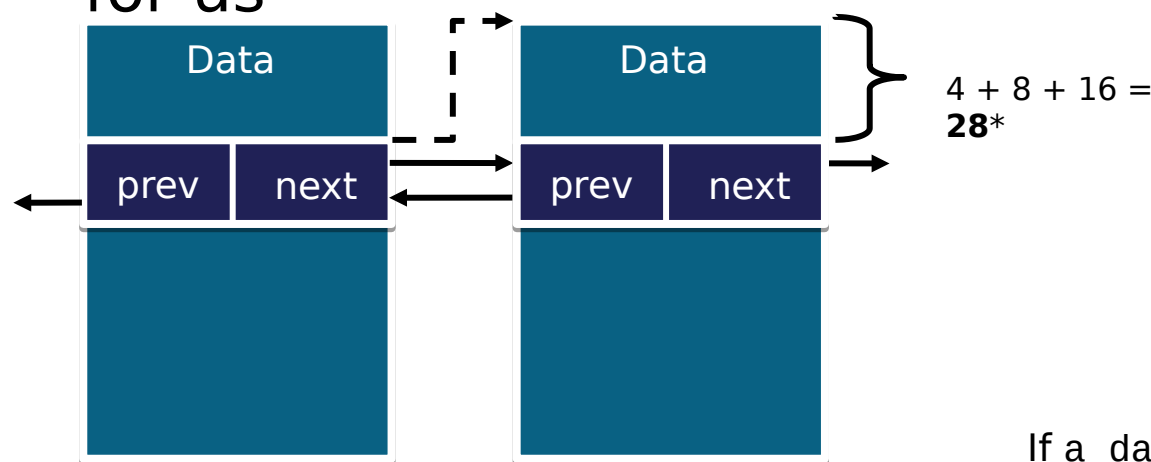
Linked Lists *(continued)*

- Linux linked lists
 - Data embeds a node structure within it
 - Links point to other links



Linked Lists *(continued)*

- A link to another link isn't very useful by itself
- How do you compute the pointer to the data?
- Macros compute the offset for us



*Assuming a packed data structure. Either way, the compiler knows how much space is used

Simple Example

```
struct list_head {
    struct list_head* next;
    struct list_head* prev;
}
```

```
struct my_data {
    uint32_t int_member;
    char* str_member;
    char[16] buf_member;
    struct list_head lnode;
    uint16_t port;
    ...
}
```

If a_data is a pointer to a struct my_data, we compute pointer to next item using
a_data->lnode.next - (28)



Behind the Scenes

- Embedding a structure inside another and doing offset calculations back to the parent structure is common in the kernel
- The **list_entry()** function can get a pointer to the structure that contains a list_head
 - Behind the scenes, it uses a generic **container_of()** function

```
#define container_of(ptr, type, member) ({  
    void *__mptr = (void *)(ptr);  
    BUILD_BUG_ON_MSG(!__same_type(*(ptr), ((type *)0)->member) &&  
        !__same_type(*(ptr), void),  
        "pointer type mismatch in container_of()");  
    ((type *)(__mptr - offsetof(type, member))); })
```



Linked Lists

- Macros and functions included with `/include/linux/list.h` allow kernel developers to create and manage kernel linked lists easily
 - Add a struct `list_head` to your data structure
 - Initialize the `list_head` member with `LIST_HEAD_INIT`
 - Use the functions provided to manage list functionality
- **`list_for_each`** - iterate through a linked list
- **`list_for_each_safe`** - Iterate through a linked list (safe with removal)
- **`list_entry`** - get a pointer to the data structure
- **`list_add`** - insert a new entry after a given one
- **`list_add_tail`** - insert a new entry at the the end of a list
- Also provided: `list_replace`, `list_swap`, `list_is_first`, `list_empty`, and many more



Additional Data Structures

- Other data structures have similarly-styled interfaces and are used throughout the kernel
 - Linked lists - `/include/linux/list.h`
 - Queues - `/include/linux/kfifo.h`
 - Hash maps - `/include/linux/hashtable.h`
 - Radix trees - `/include/linux/generic-radix-tree.h`
 - RB trees - `/include/linux/rbtree.h`
- Feel free to use these in your labs
 - For some, you'll have to



Synchronization Primitives

- The Linux kernel also supplies and makes use of various synchronization primitives
- Wait queues – Sleep in a FIFO until certain conditions are met
 - similar to POSIX condition variables
 - See `/include/linux/wait.h`
- Completion variables – Sleep until certain conditions are met
 - Also similar to POSIX condition variables, but simpler and more light-weight
 - See `/include/linux/completion`
- Spinlocks
 - Like POSIX spinlocks
 - See `/include/linux/spinlock.h`



Synchronization Primitives *(continued)*



- Semaphores
 - Like POSIX semaphores
 - See `/include/linux/semaphore.h`
- Atomics
 - `atomic_t` type has a few atomic operations you can perform on it
 - See `/include/linux/types.h` for type
 - See `/include/asm-generic/atomic-instrumented.h` for operations
- Mutexes
 - Like POSIX mutexes
 - See `/include/linux/mutex.h`



LINUX CNO Programming



Process Management

Processes

- From the view of a process, it has exclusive residence in the processor
 - No need to share registers or other program state with other processes
 - No need to even be aware of other processes at all
- The kernel is responsible for maintaining this abstraction (read: lie)
- Putting processes to sleep, saving and restoring register state, maintaining views of memory, open file descriptors, pending signals, etc. are all part of process management in the kernel



Tasks vs Threads/Processes

- Recall from Linux Internals that the relationship between threads and processes is how they were created
 - Do they share memory or not? Do they share file descriptors or not?
- Internally, the kernel considers all of these “tasks”
- Process ID can be shared among a group of threads, but internally each has its own task ID (remember `syscall(SYS_gettid)` vs `getpid()`)
 - We’ll see what else threads have in the next lab
- To avoid confusion, and be more accurate about how the kernel actually works, we simply use the word task to refer to processes/threads



Task Struct

- The kernel stores data about a task/process in the **task_struct**, defined in `/include/linux/sched.h`
- This is sometimes called the process descriptor
- It's pretty big – take a look
- Throughout the kernel, tasks are referenced by a pointer to their **task_struct**
- In the kernel, you can access the **task_struct** of the currently running process by using the **current** macro
 - This doesn't mean anything in interrupt context (more on this later)
- This is done in an architecture-specific way, but for the purpose of most modules/functions that need it, you can simply think of **current** as a global variable



Task Information

- Linux stores process descriptors (`task_structs`) in a doubly-linked circular list
- This list is encoded within the “task” member (a **`list_head`**)
- Each task structure also has pointers to a list of children and siblings, and a pointer to its parent
- Credentials are stored in the `task_struct` too (as a **`struct cred`**)
 - Capabilities
 - UID, EUID, etc.
 - Remember these from Linux Internals
- In short, all data needed to run, save, restore, and cleanup a process can be accessed through a `task_struct`



Scheduler Classes

- The Linux scheduler is modular, allowing different algorithms to operate within a single scheduler
- Different scheduler classes run different types of processes
- Each scheduler class has its own priority
- The base scheduler iterates through all the priorities from highest to lowest
- The next task that gets run is from the highest-priority scheduler class that has a runnable process
- Base scheduler defined in kernel/sched/core.c



Completely Fair Scheduler

- The Completely Fair Scheduler (CFS) is responsible for processes with normal priority
- Provides processes with a proportion of processor time
 - Processes run for a time slice inversely proportional to the number of other runnable processes on the system
 - This time slice is also weighted according to process priority
- As the number of runnable processes grows, each process gets less CPU time
- CFS is defined in kernel/sched/fair.c



"Completely Fair" Scheduling

Scenario: 4 tasks on the CPU

| Task | Time needed to run |
|------|--------------------|
| A | 4ms |
| B | 8ms |
| C | 4ms |
| D | 16ms |

What does it mean to be fair?

- For a machine with X running processes, each of them receives $1/X$ of the processor time

Ideal fair scheduler example:

| Time elapsed | A | B | C | D |
|--------------|---|---|---|----|
| 1ms | 1 | 1 | 1 | 1 |
| 1ms | 2 | 2 | 2 | 2 |
| 1ms | 3 | 3 | 3 | 3 |
| 1ms | 4 | 4 | 4 | 4 |
| 2ms | | 6 | | 6 |
| 2ms | | 8 | | 8 |
| 4ms | | | | 12 |
| 4ms | | | | 16 |

Reality:

- We can't exactly run for N milliseconds perfectly
- Scheduler gets invoked by all sorts of events too
- Not all processes have the same priority



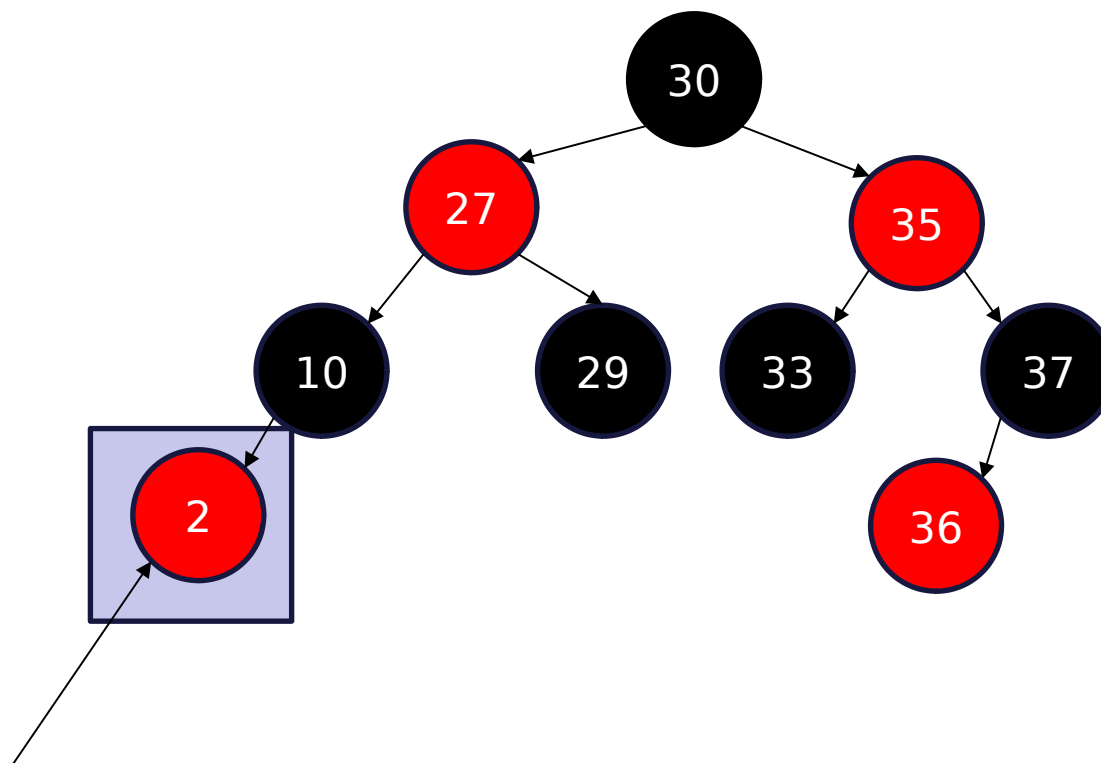
How to Do this?

- Simple:
 - Keep track of how long a program has been running
 - Each task has a vruntime, an increasing integer representing the number of nanoseconds that a task has run*
 - At each scheduler invocation, simply invoke the task with the smallest vruntime
- How do you pick the task with the smallest vruntime? Also simple:
 1. Use a balancing binary tree (rb-tree)
 2. Insert tasks into the tree based on their vruntime
 3. When scheduling, pick the one with the smallest vruntime
 4. When context switching, update the vruntime of the current task and insert it back into the tree

`vruntime += time_elapsed`



Scheduler Red-Black Tree



This task will be the next
to run



New Tasks and Priorities

What about new tasks?

new task vruntime = current minimum vruntime

Now a new task is treated like it's been there from the beginning, and all is "fair"

What about priorities?

vruntime += time_elapsed * *nice*

More nice = faster rate of accumulating vruntime = less actual time on the processor



IO-bound and CPU-bound



What about CPU-bound processes?

CPU-bound processes will use a ton of CPU and are less likely to block. Therefore they will often use up the time they are given by the scheduler. As a result, they'll naturally have more vruntime than other tasks when they are context switched out. They then get shoved more to the right of the rb-tree than an average task. This results in expected behavior: **long time slices, when run, but run less often**

What about IO-bound processes?

By definition, these block more often, and thus will have less vruntime when they are context switched out. As a result, they go more to the LEFT of the tree. This results in expected behavior: **short time slicers, but run more often**



ManTech

Scheduler Bookkeeping

- Each `task_struct` contains a member called **se**, a `sched_entity` structure that contains information used by the scheduler
 - Its **vruntime** variable (virtual runtime) stores the time the task has been running with respect to all the other runnable tasks, in nanoseconds
 - Updated by `update_curr`, which is called periodically by the system timer
- The run-queue (there is one of these for each CPU) is stored as a **struct rq**



Scheduler Invocation

- The main scheduler function is `schedule` in `kernel/core/sched.c`
 - This function chooses which task to run and performs context switches if needed
 - See notes in above `__schedule`
- The `schedule` function can be invoked through a variety of means
 - `update_process_times` is called on timer interrupt, and `schedule` may be called on interrupt return
 - Kernel threads and drivers sometimes `schedule`
 - Preemptive kernels (ones configured to have even kernel code be pre-emptable) can call `schedule` at any return
 - Processes that make explicit calls to functions which block may cause the kernel to put them to sleep and call `schedule`



Scheduling Symbols

- Common symbols relevant to the scheduler
 - `schedule` - main scheduler entry
 - `context_switch` - Switching tasks
 - `prepare_to_wait` - process being put to sleep (blocking)
 - `finish_wait` - process being woken up
- There are many more

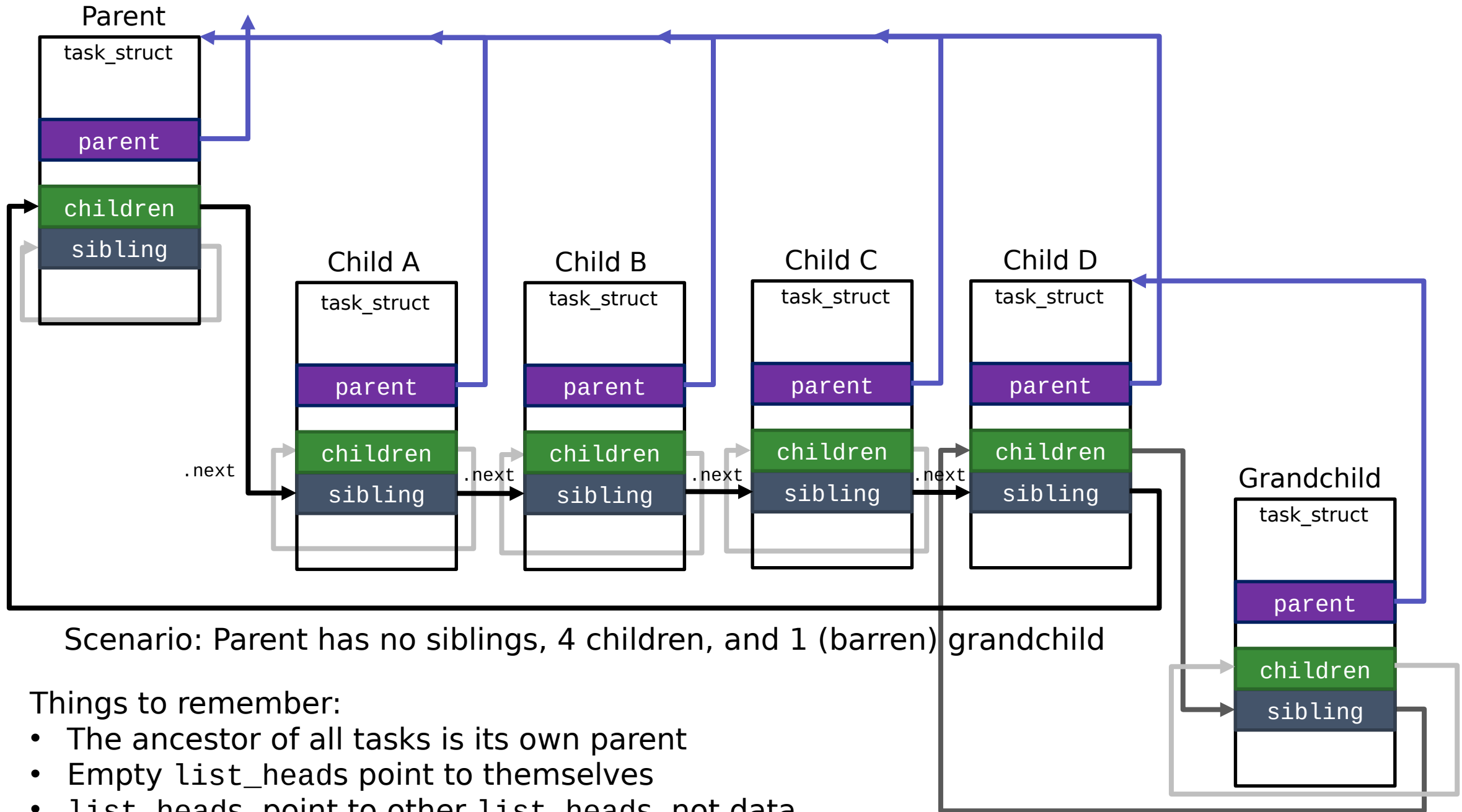


LINUX CNO Programming



Lab 4

Investigating Tasks



Tasks

- Find the `task_struct` that represents the init process
- Loop through the kernel's task family tree and print out each task's PID and then its children, in a depth-first manner
 - For each task you find, print the following
 - Its `mm` member
 - Its `pid`
 - Its `thread_pid` (address)
 - Its current state (`TASK_RUNNING`, `TASK_STOPPED`, etc.)
 - Its name (what member is this?)
- What is special about the tasks with a null `mm` member?



Tasks *(continued)*

- Now print out the last saved value of RSP (the stack pointer) of each task
 - You'll need to figure out where registers are stored
 - Hint: What API have you used that allowed you to read register state? Find it in the kernel
- Figure out how thread tasks are represented and add them to the task hierarchy
- Let's review now how the kernel handles threads and processes



Kernel Threads

- Not all tasks are userspace tasks – some are operated by the kernel
- Outside of kernel tasks, the kernel runs either in user context or interrupt context
 - A system call operates in the context of the user process that invoked it
 - An interrupt executes in interrupt context (sometimes called arbitrary context)
- Kernel tasks are pre-emptible and schedulable
- Kernel tasks are useful for
 - System daemons
 - Swapper process (idle process)
 - Background tasks for drivers



Kernel Threads *(continued)*

- Kernel threads are tasks, and as such they
 - Run in process context (their own)
 - Have a PID
 - Are scheduled by the scheduler
 - Can be preempted (if desired)
 - Can accept signals (if desired)
- A kernel API allows for the creation and management of kernel threads
 - API in `/include/linux/kthread.h`
 - `kthread_create` – Create a kthread
 - `wake_up_process` – Start a kthread (or other task)
 - `do_exit` – terminate a kthread



Kernel Threads *(continued)*

- Kernel threads can only ever be created by other kthreads (see implementation of **kthread_create**)
- **kthread_stop** does **NOT** stop a thread – it wakes it up if it's asleep, and sets a flag that allows the targeted kthread to see that it has been asked to stop
- If no other thread will be calling kthread_stop, a kthread should call **do_exit** when it knows its needs to terminate
- Otherwise it can check **kthread_should_stop** to see if it should stop (at which point it can simply return)
- A thread calling kthread_stop should be certain that the thread it intends to stop is active and not going to call do_exit itself – otherwise a system crash can occur (kthread_stop waits for the thread to finish and then operates on the target task struct)



Kernel Thread Gotchas

- The kthread API is slightly lower-level than the regular C/POSIX APIs
- Calling `allow_signal` will allow a kthread to receive that signal
- Use `signal_pending` to see if a signal has been sent (no handler gets automatically called)
- `set_current_state(TASK_INTERRUPTIBLE)` to make your kthread interruptible
- `schedule` – Give up the CPU
- `ssleep` – Give up the CPU for a given amount of seconds



LINUX CNO Programming



Lab 5

Kill me baby, one more time

Tasks

- Create a kernel module that spawns a kthread
- The kthread should
 - Put itself to sleep for 1 second
 - Wake up and iterate through the kernel task list
 - Check if any process contains the string “killme” in its name
 - Send a SIGKILL to the process if it does
 - Repeat
- When your kernel module is unloaded, it should terminate the kthread
- Remember to properly check kthread_should_stop in your thread!



LINUX CNO Programming



Interrupt Handling

Introduction to Interrupts

- Not all processing on a computer begins as a consequence of an executing program
- Various hardware devices connected to the machine periodically communicate with the processor to request various I/O handling
- These requests occur asynchronously with respect to the normal operation of the scheduler and the processes it runs
- We call them interrupts
- Interrupts are supported by a mixture of CPU hardware and kernel software



Types of Interrupts

- There are two types of interrupts
 - Asynchronous (also known as hardware interrupts)
 - Synchronous (also known as software interrupts)
- Asynchronous interrupts occur as the result of some external event
 - CPU timer expires
 - User presses key on keyboard
 - Network card receives data
- Synchronous interrupts occur as the result of code that was executed
 - Divide by zero
 - Page fault
 - `int` instruction



Types of Interrupts

- Synchronous interrupts are sometimes called exceptions, and are broken down into subcategories
 - Trap – Reported immediately after the instruction executes. Handler returns to the next instruction to be executed (preserves program continuity)
 - Example: software interrupt via `int`, `int3` (breakpoint)
 - Faults – Reported during an instruction's execution can possibly be corrected. State is saved and processor restores state to where it was before the faulting instruction executed after handler runs
 - Example: page fault
 - Aborts – Does not restart program at all after handler runs, used to report server errors
 - Example: Bad values in IDT, illegal instructions
- Chapter 6 in Intel manual
- tl;dr Traps increment RIP, Faults keep RIP, Aborts stop RIP



x86 Interrupt Descriptor Table

- As discussed in the Overview section, the CPU reads an interrupt descriptor table which is managed by the OS
- Let's take a look at `def_idts` and `apic_idts` in `/arch/x86/kernel/idt.c`
- And `idt_table`
- These specify the handlers and other settings for various interrupt vectors before the calls to `lidt` to load the table for the CPU



The Lifecycle of an Interrupt

1. When a device issues an asynchronous interrupt, it does so by sending a signal to the interrupt controller on the CPU (the APIC)
 - This is called an Interrupt Request (IRQ)
2. The interrupt controller monitors IRQ lines (different devices can be connected to different lines*)
3. The Interrupt controller sends a signal to the processor, which causes it to immediately stop executing the current process and transfer control to a kernel function
 - The current process has its context (registers, etc.) saved
4. The kernel function run is defined by the Interrupt Descriptor Table (IDT), which associates interrupts with addresses of interrupt handlers and is populated by the kernel on boot (using the `lidt` instruction on x86)

*different devices can be connected to the same line too, which we'll get to



The Lifecycle of an Interrupt *(continued)*

5. The interrupt handler routine runs

- It is possible to register multiple handlers with the same interrupt line (see `IRQF_SHARED`). In this case, the kernel will invoke all of the registered handlers in succession, and each handler is responsible for knowing whether the interrupt is actually intended for it or not

6. When the handler terminates, the kernel resumes normal execution, and executes `ret_from_intr`

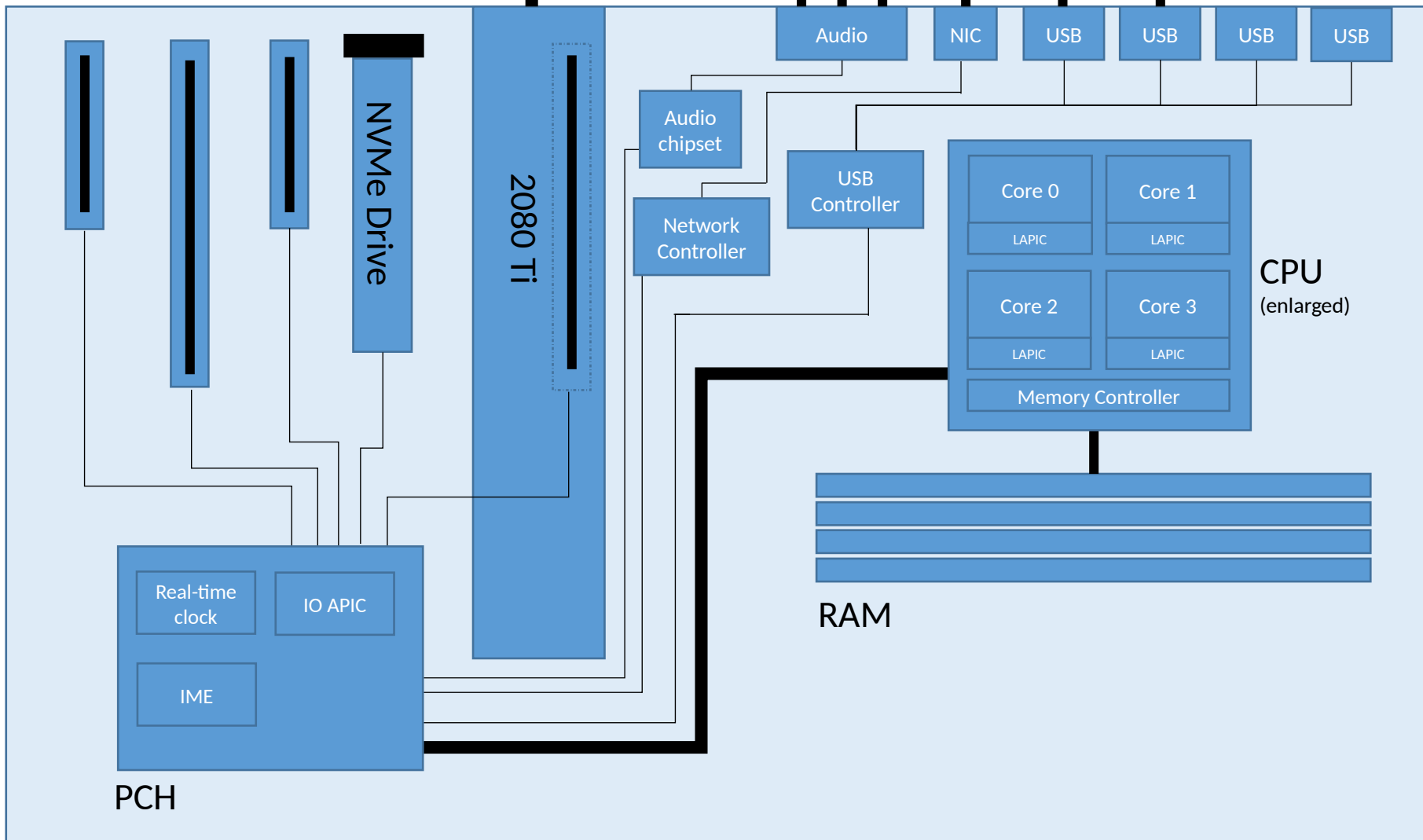
- This may invoke the scheduler again, depending on whether preemption should (or can) occur
- After this point, the initial kernel state is resumed
- `ret_from_intr` is an assembly function defined in `/arch/<arch>/entry/entry_64.S`



Interrupt Flow

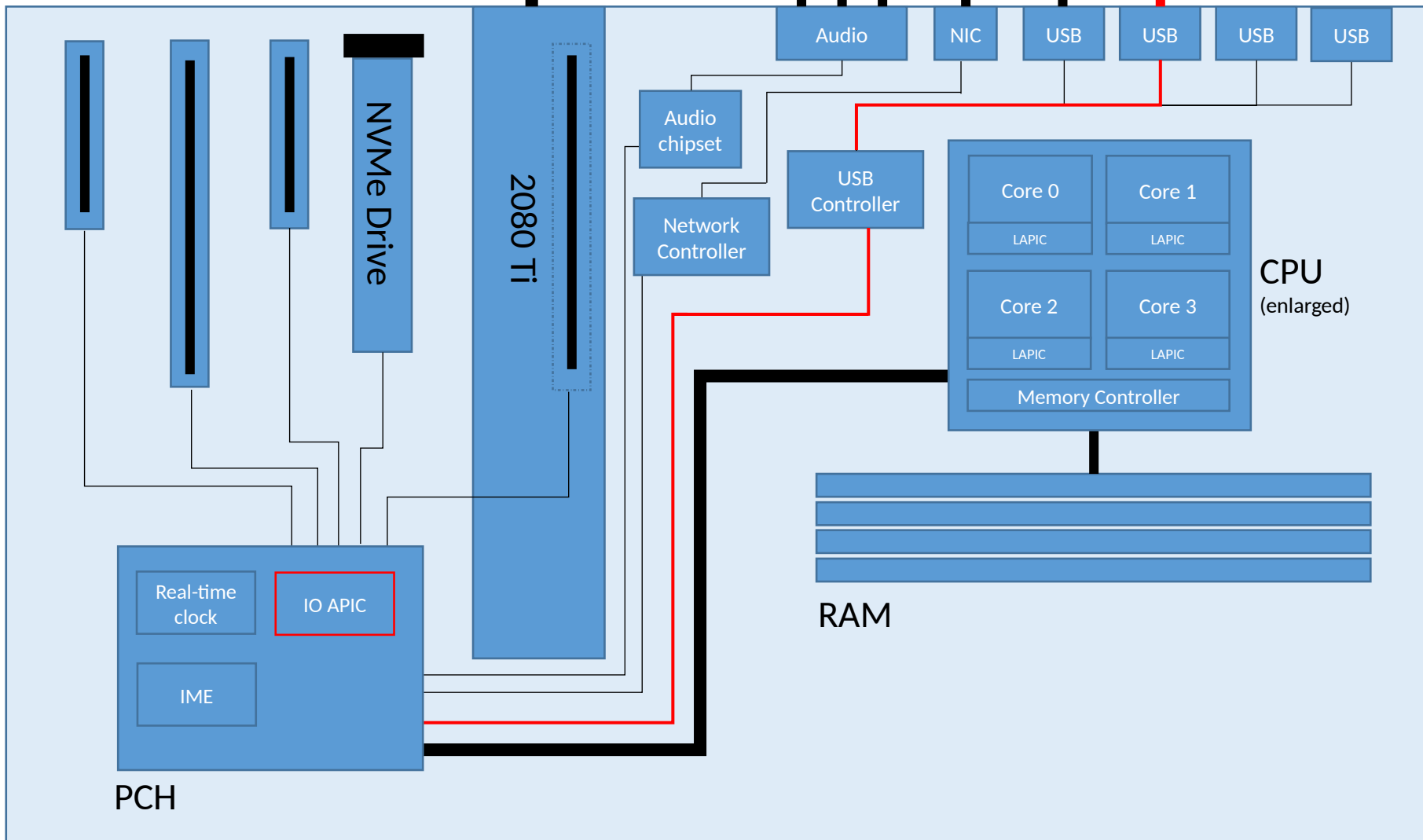


Motherboard



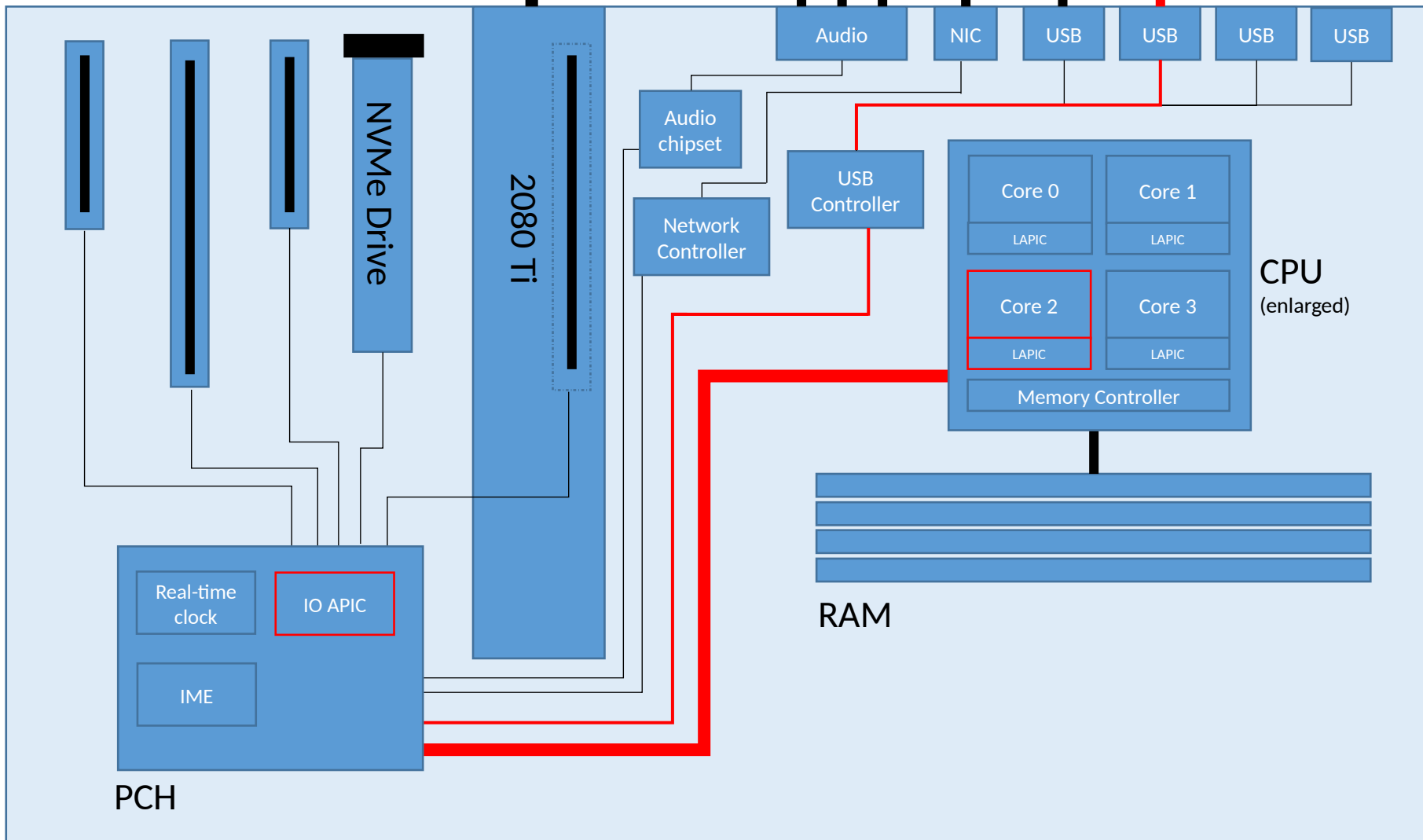
Interrupt Flow

Motherboard



Interrupt Flow

Motherboard



Programing the APIC

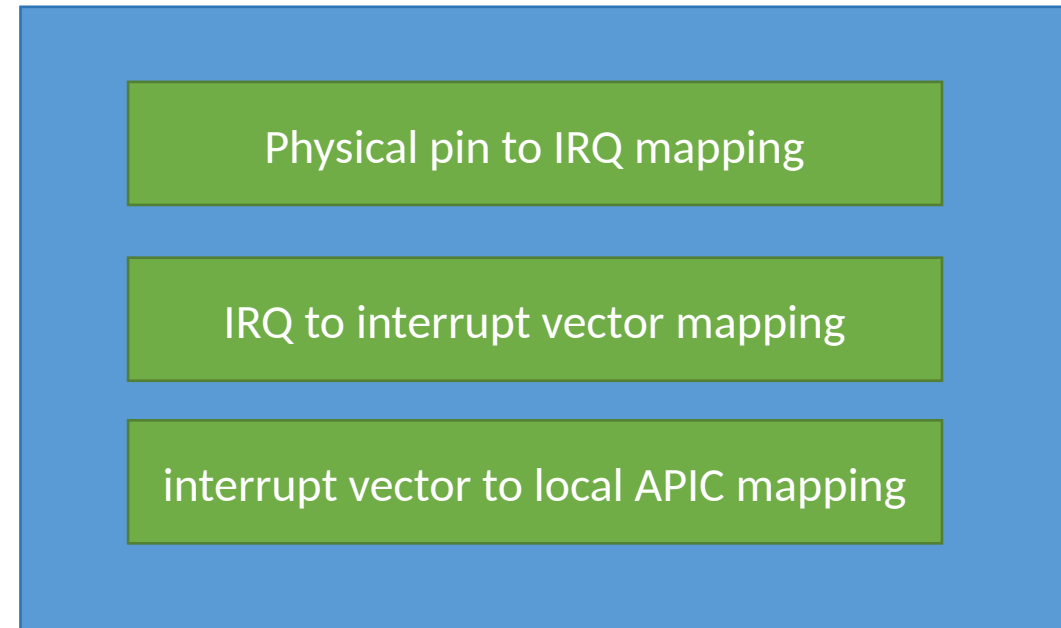
IO APIC

The APIC routes IRQs to vectors

- IRQ numbers are not interrupt vectors
- CPUs need an interrupt vector to know which interrupt handler to run
- The APIC needs to be told where it can route interrupt requests as well (to which CPUs)
- That's what the P in APIC is for

How?

- During boot the kernel will program the APIC
- This is done by writing and reading various memory-mapped registers from the APIC
- The physical address of these registers is 0xfef00000 on many systems, and can be determined via ACPI
- These settings can (and do) change at runtime



- See arch/x86/kernel/apic/io_apic.c
 - `__add_pin_to_irq_node`
 - `__iopaci_write_entry`
- structure sent to APIC for routing IRQs to interrupt vectors:
 - `arch/x86/include/asm/io_apic.h`
 - `struct IO_APIC_route_entry`

ManTech

Programming the APIC (continued)

IO APIC

Standards

- Some IRQ numbers are legacy or from standards
- For instance, the keyboard IRQ is often IRQ #1
 - Remember this is not interrupt vector #1
- IBM and intel have some standards for the first few IRQ numbers
- These can be ignored, but Linux conforms to the most common standards

Physical pin to IRQ mapping

IRQ to interrupt vector mapping

interrupt vector to local APIC mapping



IDT at a Glance

| vector | handler |
|--------|-------------|
| 0 | 0xffff..... |
| 1 | 0xffff..... |
| 2 | 0xffff..... |
| 3 | 0xffff.... |
| 4 | 0xffff.... |
| 5 | 0xffff.... |
| 6 | 0xffff.... |
| 7 | 0xffff.... |
| 8 | 0xffff.... |
| 9 | 0xffff.... |
| 10 | 0xffff.... |
| ... | 0xffff.... |
| 255 | 0xffff.... |

Each entry actually looks like this:

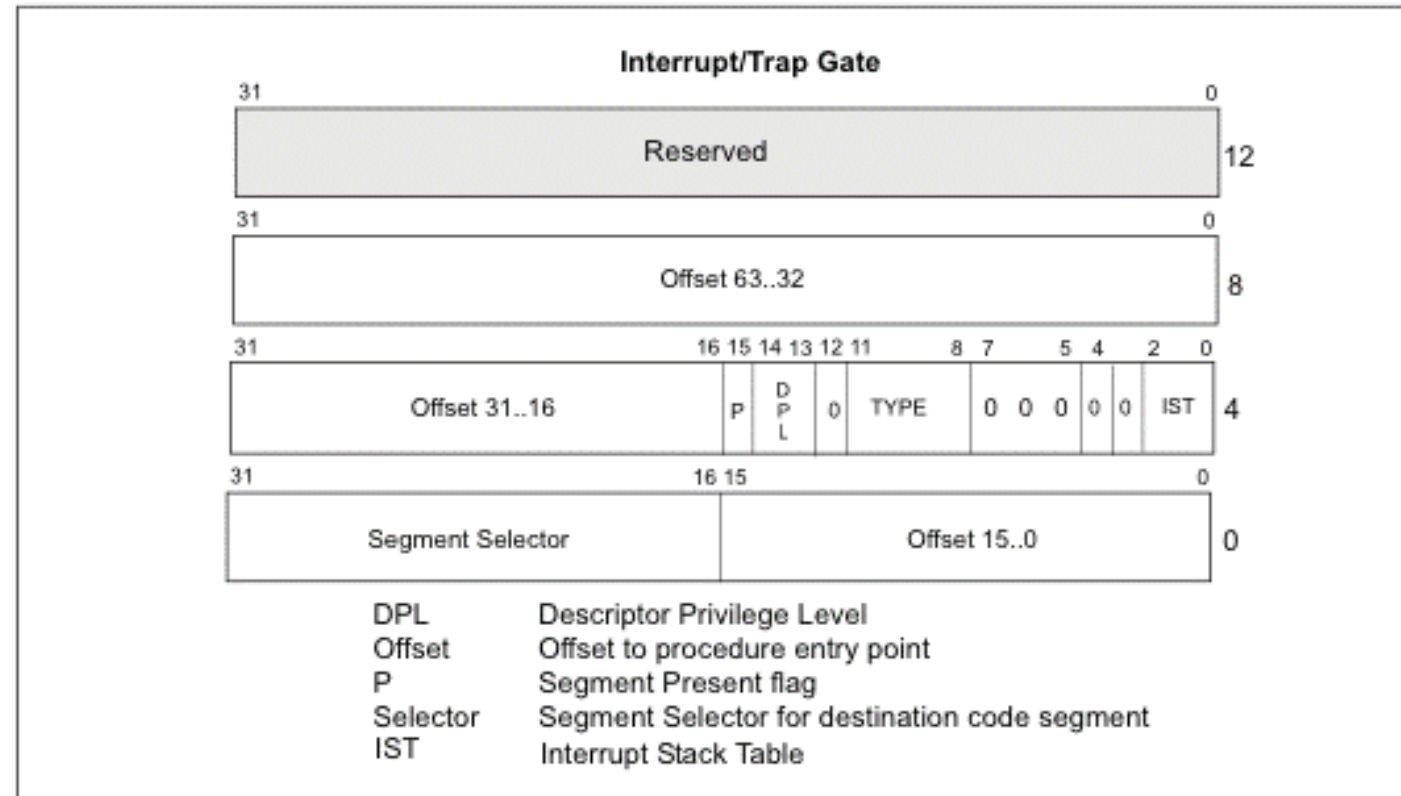


Figure 5-7. 64-Bit IDT Gate Descriptors

Interrupt Vector Assignment

- The first 32 (0 - 31) interrupt vectors are for exceptions
 - interrupts generated as a result of some internal condition
 - Examples: divide by zero, breakpoint, overflow, page fault
 - The latter-half of this range is reserved by intel
- The other interrupt vectors are usable by external IRQs
 - These can be mapped from any IRQ number to any interrupt vector > 31



Linux IDT Structs and Uses

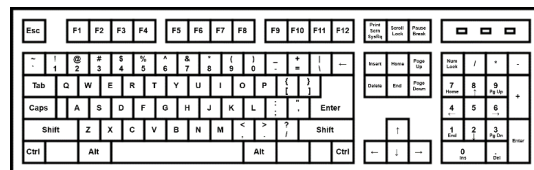
- Linux stores information that describes a IDT tables in memory
 - See `def_idts`, `apic_idts` in `arch/x86/kernel/idt.c`
 - The entries are of type `idt_data`
 - These are NOT what the CPU uses
- Linux converts entries here into the correct format for the CPU
 - See `idt_setup_from_table`
 - The `idt_init_desc` converts a single `idt_data` to a `gate_desc`
 - `gate_desc` is in the format the CPU wants (shown in previous slide)
- `load_idt` is then called to have the CPU use the table



Linux Interrupt Handlers

- The actual interrupt handlers referenced in the IDT are defined in `arch/x86/entry/entry_64.S`
- Interrupt vectors 0-31 share some macro code, but are distinct handlers
 - See `idtentry` assembly macro in the above file
- All IRQ interrupts share a single handler: `common_interrupt`
 - So the IDT is actually mostly the same entry, repeated over and over
 - This works out because the interrupt vector is put on the stack by the APIC
 - `common_interrupt` calls `do_IRQ`
 - `do_IRQ` can find the right IRQ handler based on the interrupt vector and call it!





Hardware pin

IO APIC

Interrupt vector

Local APIC

Interrupt vector

CPU Core

IDT

1) Press key

2) Raise IRQ line

3) Map pin to IRQ to interrupt vector

4) Send vector to proper local APIC

5) Interrupt execution, save state switch stacks, put interrupt vector on stack

6) Lookup handler (IDT[vector]), call handler

Everything above this line is done by hardware

Interrupted Code

Interrupt
received
here!

```
...
21: 48 8b 83 d8 06 00 00    mov    0x6d8(%rbx),%rax
28: 48 8d 8b 90 06 00 00    lea    0x690(%rbx),%rcx
2f: 89 ee                    mov    %ebp,%esi
31: 48 c7 c2 00 00 00 00    mov    $0x0,%rdx
38: 48 c7 c7 00 00 00 00    mov    $0x0,%rdi
3f: 8b 40 08                mov    0x8(%rax),%eax
...
```

Interrupt handler code

```
...
37: 50                      push   %rax
38: 68 3f fa 02 81          pushq  $0xfffffffff8102fa3f
3d: 48 cf                    iretq
```

7) Kernel tells APIC that interrupt is being handled
(EOI register written to)

Interrupted Code



Interrupt
received
here!

```

...
21:  48 8b 83 d8 06 00 00    mov    0x6d8(%rbx),%rax
28:  48 8d 8b 90 06 00 00    lea    0x690(%rbx),%rcx
2f:  89 ee                    mov    %ebp,%esi
31:  48 c7 c2 00 00 00 00    mov    $0x0,%rdx
38:  48 c7 c7 00 00 00 00    mov    $0x0,%rdi
3f:  8b 40 08                mov    0x8(%rax),%eax
...

```

Interrupt handler code

```

...
37:    50                    push   %rax
38:   68 3f fa 02 81        pushq  $0xffffffff8102fa3f
3d:   48 cf                    iretq

```

7) Kernel tells APIC that interrupt is being handled
(EOI register written to)

8) common_interrupt called, do_IRQ called

9) Registered ISR for IRQ runs

10) common_interrupt returns with iretq

11) CPU restores previous state, switches stack back

12) Execution continues

```

...
21:  48 8b 83 d8 06 00 00    mov    0x6d8(%rbx),%rax
28:  48 8d 8b 90 06 00 00    lea    0x690(%rbx),%rcx
2f:  89 ee                    mov    %ebp,%esi
31:  48 c7 c2 00 00 00 00    mov    $0x0,%rdx
38:  48 c7 c7 00 00 00 00    mov    $0x0,%rdi
3f:  8b 40 08                mov    0x8(%rax),%eax
...

```

Notes

- The previous example is for an external interrupt from a keyboard
- Internal interrupts behave a bit differently:
 - No APIC involvement (it's all local to the CPU core)
 - Interrupted code **may** execute same instruction (like for page fault)
 - Interrupted code **may** stop execution (like for divide by zero)
 - Interrupted code **may** execute from next instruction (like for overflow)



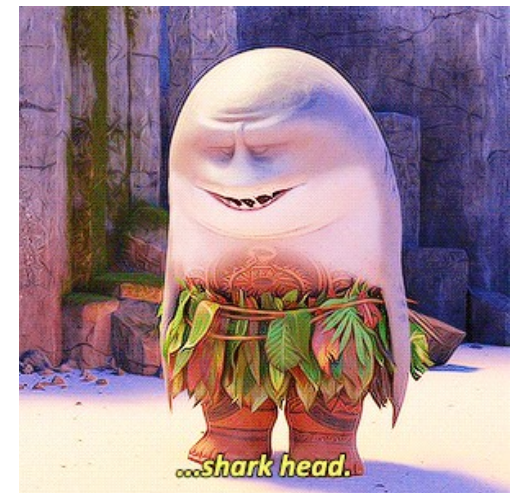
Interrupt Service Routines (ISRs)

- Interrupt Service Routines (ISRs), or interrupt handlers, are functions invoked by the kernel in response to the reception of an interrupt
- ISRs are responsible for performing any processing needed in response to the interrupt
 - Logging a keypress from a keyboard
 - Copying data from a network card
 - Acknowledging a timer expiration
- They can't (read: shouldn't) block
- They can't (read: shouldn't) do a lot of processing



Handling Interrupts

- Handling of interrupts occurs in two phases – the **top half** and **bottom half**
 - These terms are a bit dated, but they still apply to the semantics of interrupts
- The top half refers to the ISR, wherein code cannot block and must execute briefly to not stall the processor
- If more processing is to be done (such as the kind that potentially requires blocking), then the top half can queue up a bottom half to do it
- Bottom halves are run by the kernel through normal scheduling
- What mechanism in the kernel might be useful for executing the bottom half?



ManTech



Deferring Work

- The “bottom half” historically referred to the actual mechanism used for an interrupt handler to defer its work
 - Sometimes you’ll still see symbols with “bh” (bottom half) in their name
- In modern Linux, this system has been replaced with three alternatives:
 - Softirq
 - Tasklets
 - Workqueues
- Let’s talk about all these



Softirqs

- Softirqs are statically allocated during the kernel compilation process
 - `static struct softirq_action softirq_vec[NR_SOFTIRQS];`

| NAME | RELATIVE PRIORITY | DESCRIPTION |
|------------------|-------------------|--------------------------|
| HI_SOFTIRQ | 0 | High-priority tasklets |
| TIMER_SOFTIRQ | 1 | Timers |
| NET_TX_SOFTIRQ | 2 | Network send |
| NET_RX_SOFTIRQ | 3 | Network receive |
| BLOCK_SOFTIRQ | 4 | Block devices |
| IRQ_POLL_SOFTIRQ | 5 | IRQ polling |
| TASKLET_SOFTIRQ | 6 | Normal priority tasklets |
| SCHED_SOFTIRQ | 7 | Scheduler |
| HRTIMER_SOFTIRQ | 8 | Not used |
| RCU_SOFTIRQ | 9 | RCU locking |



Softirqs *(continued)*

- Softirqs are executed only if they are “raised” by some top-half interrupt handler
 - Example: `raise_softirq(NET_TX_SOFTIRQ)`
- Raised softirqs are executed when
 - Returning from an a hardware interrupt (`irq_exit`)
 - Called by a kernel thread (`ksoftirqd`)
 - There is one `ksoftirqd` per processor (these are started at boot)
 - Explicitly called by some subsystem
- See `/proc/softirq`
- When `__do_softirq` is invoked, the handler for each raised softirq is executed



Softirqs *(continued)*

- Softirqs are used for extremely time-sensitive processing
- Registering a handler requires an index into the global `softirq_vec` and a handler function
 - Example: `open_softirq(NET_TX_SOFTIRQ, net_tx_action);` in `/net/core/dev.c`
- When a softirq handler is running, other softirqs on the same processor cannot be run (but other processors can run other handlers – even the same handler)
 - As a result, handlers have to be very conscious of race conditions and locking



Tasklets

- Tasklets are implemented on top of softirqs (see softirq HI_SOFTIRQ and TASKLET_SOFTIRQ)
 - The handlers for these two softirqs operate on a list of tasklets when they are run
 - The softirq handlers run the tasklet handlers, and do some general locking
- They can be allocated and initialized at runtime
- Defined in linux/interrupt.h as tasklet_struct
- A particular tasklet can only be run on one CPU at a time, greatly simplifying design, but sacrificing scalability
 - Two different tasklets can be run on two CPUs simultaneously



Tasklets *(continued)*

- Tasklets are scheduled (equivalent to softirq raised) with the `tasklet_schedule` and `tasklet_hi_schedule` functions
- Scheduled tasklets are added to the list of tasklets that will be run by the tasklet softirq handlers
 - These in turn call `tasklet_action` and `tasklet_hi_action`, which will invoke the handlers for the scheduled tasklets
- `tasklet_init` will initialize a tasklet
- `tasklet_disable` disables a tasklet
- `tasklet_enable` enables a tasklet
- `tasklet_kill` removes a tasklet from a pending queue
- Each tasklet has a handler
 - Type `void tasklet_handler(unsigned long data)`



Work Queues

- Work queues defer interrupt work to a kernel thread
- Unlike softirqs and tasklets, work queues always operate in process context (the process of a kernel task)
 - This means they can sleep, perform blocking I/O, wait on a semaphore, etc.
 - If your bottom half doesn't need this, just use a tasklet
- A generic set of worker threads are created by the kernel (events) that handle items in work queues
 - See `ps -ef | grep events`
- When using work queues, you can spawn your own kernel thread to perform the work, or use one of the generic worker threads



Work Queues *(continued)*

- Kernel worker threads run the `worker_thread` function, which puts a thread to sleep until it is woken up to perform work
 - There is one of these threads per processor
- These threads operate on `work_structs` in a linked list
 - These contain work functions that are executed
- Work queue API is in `/include/linux/workqueue.h`
- `DECLARE_WORK` or `INIT_WORK` will initialize a `work_struct`
- `schedule_work` will....schedule work
- `flush_scheduled_work` - wait for work to be done
- See also `cancel_delayed_work`, `schedule_delayed_work`
- The API also supports creating your own worker threads



Interrupt Context

- ISRs run in interrupt context
- Normally the kernel executes in **process context**, wherein the kernel is performing operations on behalf of a given userspace process (the task to which `current` points)
 - Sleeping in process context is okay, as the process can be rescheduled
- When an interrupt occurs, it executes in interrupt context
 - There is no task associated with the operations of an interrupt handler (though `current` still points to some task)
 - Since there is no associated task, an interrupt handler cannot be rescheduled, and therefore it cannot sleep
- ISRs should therefore run very briefly, performing the minimal amount of work necessary, and then offload any other required work to a bottom half handler



Running an ISR

- ISRs have their own stack (one per processor/core)
- Historically ISRs shared either the stack of the process they interrupted or the kernel's stack
- ISR stacks are one page in size (very small)
- When an ISR runs, all interrupts (except the one with which it is associated) are still enabled
 - However, during ISR registration, programmers can specify that all interrupts should be disabled when the given ISR runs



Kernel Interrupt Handler Interface

- `request_irq` / `free_irq` – register and unregister an interrupt handler
 - Flags used to register a handler
 - `IRQF_DISABLED` – disable all interrupts when this handler executes
 - `IRQF_SAMPLE_RANDOM` – use this handler as an entropy source
 - `IRQF_TIMER` – tells the kernel that this handler processes system timer interrupts
 - `IRQF_SHARED` – Allows this interrupt line to be shared by multiple handlers
- `local_irq_disable` / `local_irq_enable` – disable or enable interrupts
- `in_interrupt` – used to determine if currently executing in interrupt context
- `in_irq` – used to determine if currently executing an interrupt handler
- `local_irq_save` / `local_irq_restore` – save and restore interrupt state



Registering an Interrupt Handler

- See request_irq in linux/interrupt.h

```
int request_irq(unsigned int  irq,
                irq_handler_t handler,
                unsigned long flags,
                const char* name,
                void* dev);
```

```
typedef irqreturn_t (*irq_handler_t)(int, void *);
```

From interrupt.h

```
enum irqreturn {
    IRQ_NONE           = (0 << 0),
    IRQ_HANDLED        = (1 << 0),
    IRQ_WAKE_THREAD    = (1 << 1),
};
```

```
typedef enum irqreturn irqreturn_t;
#define IRQ_RETVAL(x)  ((x) ? IRQ_HANDLED : IRQ_NONE)
```

See free_irq for
unregistering an
interrupt handler



LINUX CNO Programming



Lab 6

Creating an interrupt handler

Tasks

- Create an interrupt handler for an IRQ number, and share it with another handler
 - See /proc/interrupts to view those currently taken
 - IRQF_SHARED will need to be set on both!
- Your interrupt handler should count the number of times it has been invoked
 - What protections might be necessary?
 - Can you sleep in an interrupt handler?
- Print out the current value of the counter using a deferred work mechanism of your choice
 - Of the three, which ones are actual options for a loadable module?





System Call Interface

System Calls

- System calls allow applications to request that the operating system perform some operation on their behalf
 - Send network data
 - Read data from disk
 - Get PID
 - Etc.
- Sometimes called syscalls for short
- Essentially the API to the operating system



Defining a Syscall

- System calls always return a long, but can have a variable number of arguments (including zero)
- Syscalls are defined using the SYSCALL_DEFINEn macros, where the n is the number of arguments that the system call takes
- After preprocessing, the system call symbol func is named sys_func
 - e.g., SYSCALL_DEFINE3(func) \square `asm linkage long sys_func`



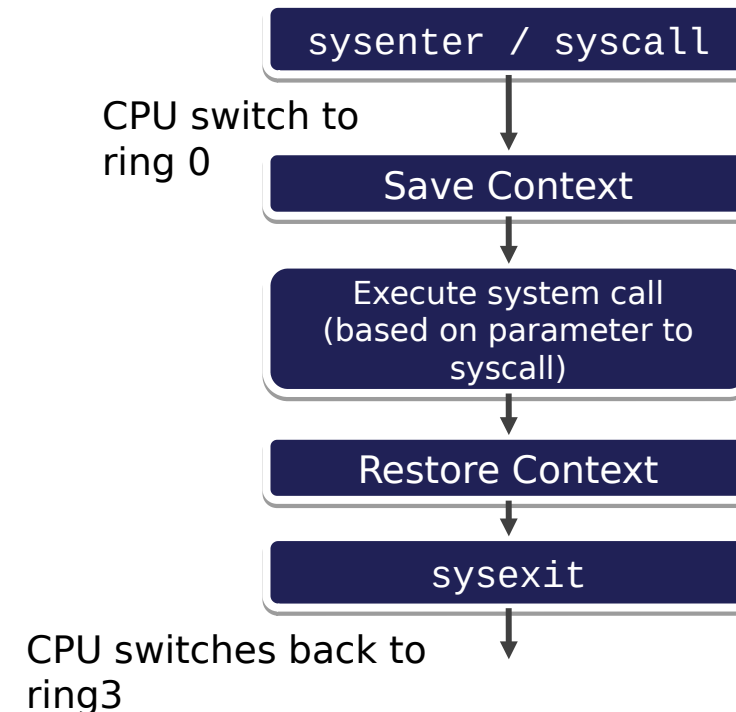
System Call Table

- Applications do not invoke a system call using its name (applications are not linked with the kernel)
- Instead they use a system call number, which is mapped to a particular system call
- In the kernel, the `sys_call_table` contains a mapping of numbers to system call function pointers
 - See `arch/x86/entry/syscall_64.c`
 - Also see `include/linux/syscalls.h`
- In Linux, these numbers do not change (as doing so would prevent compiled programs from running correctly)
- The `sys_ni_syscall` function is used to provide an implementation for unused system call numbers



System Call Execution

- System calls cause a context switch
- Privileged kernel code runs the requested system call on behalf of the calling process
- `sysenter/syscall` instructions (x86_64) are used for this
 - Previously system calls were implemented as software interrupts (`int 0x80` instruction)



Verification of Parameters

- The kernel can't trust anything coming from userspace
- Pointers coming from an application may be invalid
 - Pointer to unallocated memory
 - Pointer to memory not owned by the process
 - Pointer to memory whose contents change after validation
- File descriptors may be invalid
- Process IDs may be invalid
- To prevent these problems, the kernel must properly validate and copy any memory passed to it from userspace



Verification of Parameters *(continued)*

- The kernel provides an API to safely copy data coming from and going to user space in `linux/uaccess.h`
 - `access_ok` – Check to see if a user space pointer is valid
 - `copy_to_user` – Copy kernel memory to user memory
 - `copy_from_user` – Copy user memory to kernel memory
 - There are a variety of other helpers based on these that may save you some additional work
 - `strnlen_user`
 - `strncpy_from_user`
 - Etc.



Adding a System Call

1. Define the system call using the `SYSCALL_DEFINEn()` macro in a new file
2. Declare a system call prototype
 - Located at `/include/linux/syscalls.h`
3. For cross-architecture compatibility, add your system call to the generic shared system call table
 - Located at `include/uapi/asm-generic/unistd.h`
 - `#define __NR_mycall <number>`
 - `__SYSCALL(__NR_mycall, sys_mycall)`
4. For x86, update the master syscall table
 - Located at `arch/x86/entry/syscalls/syscall_64.tbl`
5. Provide a fallback implementation by adding an entry to the non-implemented set
 - Located at `kernel/sys_ni.c`
 - Add `cond_syscall(sys_mycall)`



Adding a System Call *(continued)*

6. Make any changes needed to kernel configuration options
7. Add a Makefile inside any new directories created for your system call to tell Make to build your files
 - Add any new objects to the `obj-y` list
 - e.g., `obj-y:=mysyscall.o` (for `mysyscall.c`'s object)
8. If you've added a new directory for your code, add this to the top-level kernel Makefile as well
 - Add new directory paths to the `core-y` list
 - e.g., `core-y += kernel/ certs/ mm/ ... mydir/`
9. Make and test your new kernel!
 - `make -j <cores>`



LINUX CNO Programming



Lab 7

Create a system call

Tasks

- Create a new system call in the kernel
 - Follow the steps on the previous slides
- The new system call will cause the kernel to set the canary value of a given PID task to a value specified by the user
 - See `stack_canary` member of `task_struct`
- It will also return the previous canary value in another parameter
- Your system call will require three parameters
 - The PID of the task to affect
 - The new canary value
 - A pointer to a buffer that will receive the old canary value
- Copy the parameters safely into and out of kernelspace as part of your system call handling



LINUX CNO Programming



Driver Interaction

What is a device?

- The term "device" is historically linked to actual physical devices
 - Mice, network cards, drives, etc.
- A device is an interface for interaction with kernel code
- A device driver can have multiple devices associated with it
- Usermode code can interact with kernel modules via a device
- Modules create devices
 - Kernel defines `THIS_MODULE` for all loadable modules
 - Modules may have zero or more associated devices



What is a device? *(continued)*

- Devices files, or special files, are inodes on the filesystem tree
 - Conventionally in the /dev directory
 - But don't have to be
 - Not a regular file
 - /dev/zero isn't anywhere on "disk"
- File operations are implemented by the Loadable module
 - Module decides what read and write means on it's device, and how they work
 - A read from /dev/random needs to work differently from a read from /dev/stdin
- Devices are identified by major/minor numbers



Maj / Min

- Device files are associated with the major/minor numbers of the device
- `ls -l /dev`
 - C for character devices
 - B for block devices
 - Major and Minor numbers associated with the device
 - `ls /sys/dev/char`
 - `ls /sys/dev/block`
 - `cat /proc/devices`
- Internally, the kernel uses `dev_t`
 - Unsigned 32 bit number
 - 12 bits for major, 20 bits for minor
 - `MAJOR(dev_t dev); MINOR(dev_t dev);`



Maj / Min *(continued)*

- Kernel modules should reserve major/minor number ranges with the kernel
 - `register_chrdev_region`
 - Registers a set number range with the kernel
 - Starting at (maj/min) and requesting a given number of device numbers
 - `alloc_chrdev_region`
 - Asks the kernel for a free region
 - Starting at (maj/min) and requesting a given number of device numbers
- Having a known major/minor number makes it easy to interact with your device
- Having a dynamic major/minor number prevents collisions with other modules



Mknod

- To create a special file, use mknod
 - Based on maj/min number
- There is a mknod system call
- /usr/bin/mknod is a command
 - Make a few nodes to the random device
 - `sudo mknod ./myrand c 1 8`
- Drivers can register with sysfs to have udev automatically do mknod for them
 - But more on that later



Character Devices

- Character devices are a type of device that the kernel allows us to make
 - Some other types are block devices and network devices
- Character devices work character by character
 - I can ask /dev/random for just 3 bytes
- Block devices work in blocks with buffered requests
 - Disks are usually block devices
 - Block devices need to work fast
- Usermode code doesn't need to worry about the type of a device
 - The kernel's API for the devices is very different though



Character Device

- `struct cdev`
 - Kernel's structure for a character device
 - Contains a pointer to owner (`struct module`)
 - Contains a `dev_t`
 - Contains a `file_operations` structure
- Allocated with `cdev_alloc`
 - Free with `kfree`
- Or can be embedded into another structure
 - Still needs to be initialized
 - `cdev_init`
- Register your character device with `cdev_add`



File Operations

- Character devices register their operations with struct `file_operations`
- This struct
 - is filled with important function pointers
 - can be unique per device
- Null entries are considered "unsupported"
- Syscalls working with special files go through these function pointers

```
struct file_operations my_fops = {  
    .owner =    THIS_MODULE,  
    .read =    my_read,  
    .write =    my_write,  
    .open =    my_open,  
    .release =  my_release,  
};
```



inode

- When using `mknod`, we create a new inode in the virtual file system (VFS)
 - We will cover this more later
- Inodes contain a `dev_t` to specify what device (if any) they are associated with
 - `i_rdev`
 - Use the `imajor` and `iminor` functions to get the numbers portably
- Inodes for character devices contain a pointer to a struct `cdev`
 - `i_cdev`
- Inodes also contain a pointer to the associated struct `file_operations`
 - `i_fop`
- Multiple different inodes can be associated with the same character device (appearing as different files on the system)



Open Operation

- The entry for open in the `file_operations` looks like this:
 - `int (*open) (struct inode*, struct file*);`
 - If set to NULL, opening a file handle to the device always succeeds, without notification
- Open is called with the backing inode, and a newly opened struct `file`
 - See `/fs/open.c` at `do_dentry_open`
 - Not called on a dup
 - But a dup will increase the reference count of the struct `file`
- When **all** references to the file are closed, the file operation `release` is called
 - This is not the same as each time `close` is called!
 - There is no `close file_operation` that can be implemented
 - True for sockets too



File Structure

- The struct `file` is associated with an open file, and goes away when all references to that file are gone
- Associated with file descriptors via the processes struct `files_struct`
 - `struct file* f = current->files->fd_array[fd];`
- The struct `file` contains a reference to its inode, file operations, mode, and more
 - `f_inode, f_op, f_mode`
- There is an entry that can be used by a driver to keep track of state
 - `void* private_data`
- All file operations take a struct `file` as a parameter, to know which file they operate on



Checking Access

- Remember to use:
 - `copy_to_user`
 - `copy_in_user`
 - `copy_from_user`
- Check user memory range is valid with: `access_ok`
- Check if a user has a capability with: `capable`
- Check file mode on the struct file
- Also see `<linux/cred.h>`
 - `current_uid`
 - `current_gid`
 - `current_euid`
 - `current->cred`



Other File Operations

- `ioctl`
 - The `ioctl` system call, from userspace has a declaration of:
 - `int ioctl(int fd, unsigned long request, ...);`
 - The "..." isn't actually a variable number of arguments, it just gets around C type checking
 - It's actually a `char* argp`
 - Lots of kernel developers don't like `ioctl`, because it is such an uncontrolled entry point
 - request can be anything, but it is nice to have it be unique and match a pattern
 - `_IOC(dir, type, nr, size)` in `<uapi/asm-generic/ioctl.h>`
 - `unlocked_ioctl`
 - There used to be a normal `ioctl`, that used a big kernel lock, but it doesn't exist anymore
 - `compat_ioctl`
 - Specify if you want to handle 32 bit `ioctl` requests differently, without relying on existing conversion mechanism
 - Is also unlocked



Other File Operations

- poll
 - Allows `epoll/poll/select` to all work with handles to your device
- mmap
 - Allows users to map memory into their address space
- `lseek`
 - Allows users to seek within the "file" to a new offset
- And more!



LINUX CNO Programming



Lab 8

Character Devices

Lab



- Tasks:

- Create a character device that will xor anything written into it
 - Xor on write, but spit out the xord bytes on read
- Implement at least the read, write, and unlocked_ioctl file operations
- Change the xor key via an ioctl
- The data should persist with the device

- Bonus:

- Don't have a maximum size
- Allow different buffers/keys for different nodes to your device





Virtual Memory

Kernel Space Memory

- Virtual memory is divided between userspace and kernelspace
 - Kernelspace addresses are called "supervisor-mode" addresses by Intel
 - Userspace addresses are called "user-mode" addresses by Intel
- Usually kernelspace memory is at high addresses
 - Intel x86_64 processors currently use 48 out of 64 bits for virtual addresses
 - Most-significant 16 bits should be the all 0 or all 1
- Usermode applications are not allowed to access kernel mode memory directly
- The kernel has an API for checking if an address is userspace:
 - `access_ok`
 - `user_addr_max`
- And an API for handling them:
 - `set_fs/get_fs` (don't use)
 - `iov_iter` with `copy_from_iter` / `copy_to_iter` (use)
 - `copy_from/in/to_user`



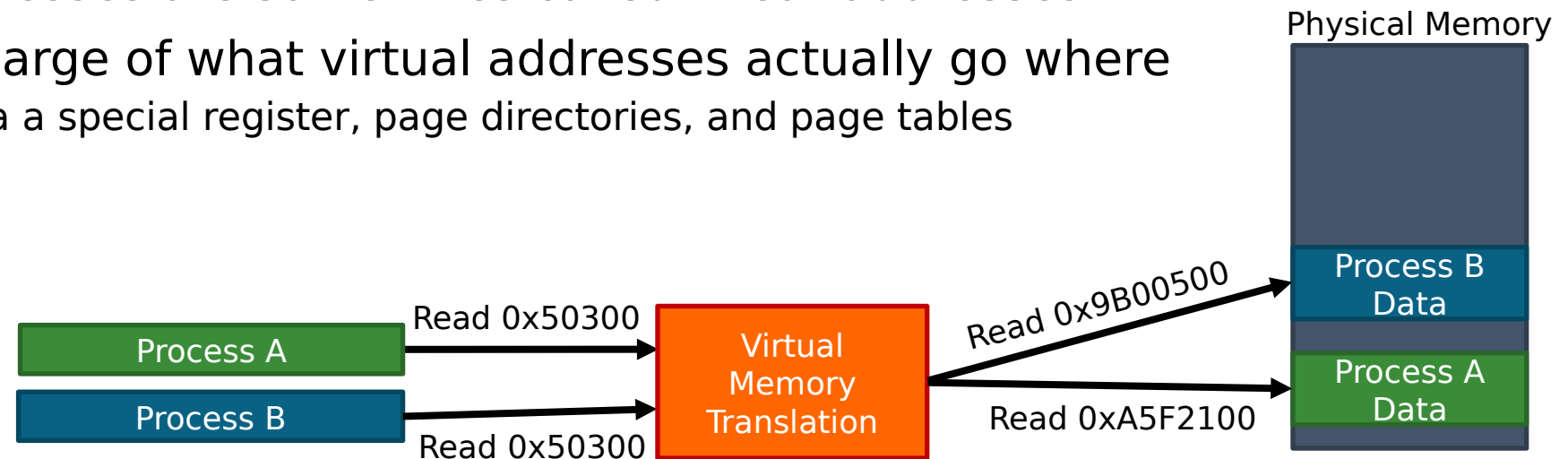
Virtual and Physical Memory

- Modern machines don't directly use the physical memory addresses
- An address in an instruction can point to a different part of physical memory than the same address used by another process
- Instructions use “virtual” addresses, which the CPU will automatically translate into a physical address
- This translation is performed by the CPU and managed by the kernel
- Virtual Memory gives us:
 - Memory permissions (RWX)
 - Process memory containment
 - Shared memory



Virtual and Physical Memory

- Physical addresses are the actual addresses of memory
 - On intel 64-bit machines they are constrained to 52 bits
 - Note that you can have more physical memory than you could actually map into the 48-bit virtual address space
 - These may point to RAM, ROM, devices on the bus, etc.
- Virtual addresses need to be translated to physical ones by the CPU
 - Virtual addresses are sometimes called linear addresses
- The OS is in charge of what virtual addresses actually go where
 - It does this via a special register, page directories, and page tables



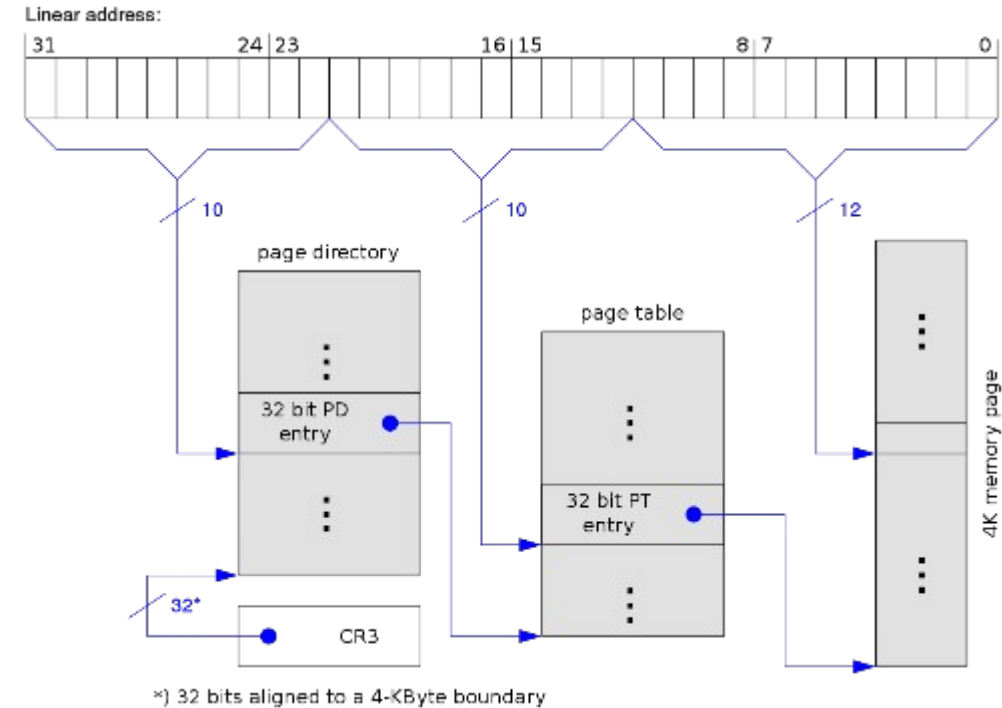
Page Tables and Directories

- Page table formats are specific to the hardware
 - The CPU has to be able to parse them
- Page tables contain entries used to:
 - Translate virtual addresses to physical
 - Set page permissions
 - Mark pages as dirty (modified)
- They use pieces of the virtual addresses to know what entry in the table to use
- Page tables can have multiple levels
 - Tree-like structure
 - x86_64 supports 4-tier and 5-tier page tables



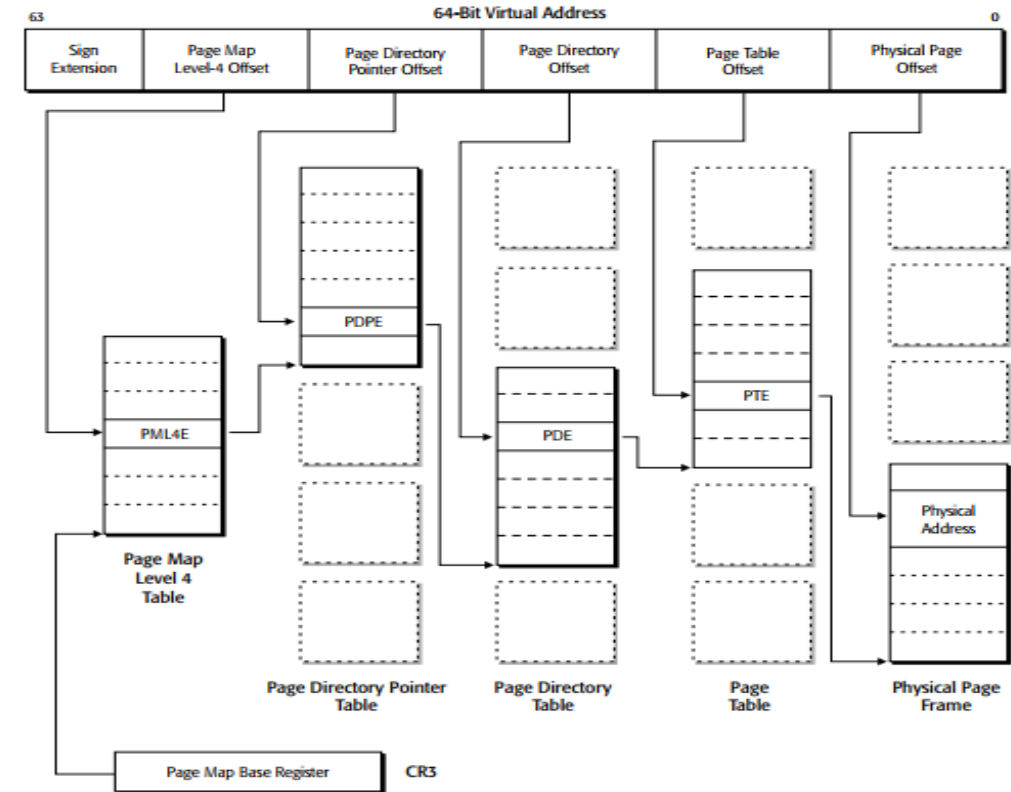
Page Tables

- Here we use a 32-bit address to traverse a 2-level page table structure
- Top level index:
 - $\text{addr} \gg 22$
- Next level index:
 - $(\text{addr} \gg 12) \& 0x3ff$
- Offset in physical page:
 - $\text{addr} \& 0xfff$



Modern 64-bit Address Translation

- On modern x86 processors, we use four levels of page tables
- Image from AMD's AMD64 Architecture Programming Manual
- Page Table Entries (PTEs) also contain information about the page they reference
 - Permissions bits
 - Present bit
 - Dirty bit
 - See `arch/x86/include/asm/pgtable_types.h`



Page Tables

- There are usually multiple page tables for a system
 - Each process may need its own unique page table
 - The CPU knows where to look for the page table via the CR3 register (physical address)
- The kernel keeps a pointer to the top level of the page table associated with a task
 - `current->mm->pgd`, which gets loaded into the CR3 register on context switch
- Levels all have different names
 - PGD: Page Global Directory
 - P4D: Page 4th-level Directory (only used in 5+ level page tables)
 - PUD: Page Upper Directory (only used in 4+ level page tables)
 - PMD: Page Mid-level Directory (only used in 3+ level page tables)
 - PTE: Page Table Entry
- Each level has SHIFT, SIZE, and MASK macros
 - e.g. `PAGE_SHIFT`, `PMD_SHFT`, `PGDIR_SHIFT`, etc.



Page Tables *(continued)*

- See example of page table traversal in the kernel:
 - `walk_page_range` in `/mm/pagewalk.c`
- The kernel has common functions to read entries
 - `/arch/x86/include/asm/pgtable.h`
 - `/arch/x86/include/asm/pgtable_64.h`
 - `pte_exec`
 - `pte_clear`
 - `pte_set_flags`
 - `pfn_pte`
 - Creates a page table entry (PTE) from a physical page number (page frame number)
 - `pte_offset`
 - Also exists for `pmd`, `pud`, ...



TLBs – The VM Cache

- Page table lookups can be expensive
 - The CPU has to access memory multiple times to translate a virtual address
- A Translation Lookaside buffer (TLB) caches page table entries
 - On x86_64, there are multiple of these, organized like a typical cache
 - One level 1 data-TLB and one level 1 instruction-TLB per core, as well as a unified level 2 TLB
 - This makes things go a lot quicker
- The TLB needs to be updated to match the current page table continually
 - During context switches, unmapping memory, etc
 - Can lead to some fun vulnerabilities
 - Also important in many speculative execution vulnerabilities
 - See `flush_tlb_all`, `flush_tlb_page`, `flush_tlb_range`



Paging

- Having virtual memory allows the kernel to optionally swap out currently idle pages to disk, in order to free up physical memory for other uses
- This is called swapping or paging
- The swap partition on your local disk is used for this purpose
- Swapping is managed by kswapd, a kernel thread
- Usermode programs do not have view into this
 - but should still be mindful of their memory use
- Kernelmode code has to be careful about this
 - Accessing swapped out memory will cause a page fault
 - Doing this at the wrong time/place can cause crashes



Task Memory

- `current->mm`
 - Structure of type `struct mm_struct`
 - Very struct
 - If the task is an anonymous process, then `current->mm` will be NULL
 - But `current->active_mm` will be the address space to which the anonymous process is currently attached
- The `struct mm_struct` contains
 - A pointer to the page global directory, `pgd`
 - A list of virtual memory regions for the process, `mmap`
 - Keeps virtual memory areas organized in an `rb_tree` and a linked list
 - A function for finding unused virtual addresses, `get_unmapped_area`
 - A pointer to the operations for virtual memory, `vm_ops`



A Note on mmap

- The file operations structure allows a device to implement mmap
 - When an mmap operation occurs, the kernel will create a `vm_area_struct` structure
 - This structure will dictate how operations on the new memory region are performed
- This can be used to map memory into userspace
 - `remap_pfn_range`
 - Also see `struct address_space` and `vm_operations_struct`



LINUX CNO Programming



Kernel Memory Management

Kernel Memory

- The kernel needs to run very quickly, but also needs to manage a large number of resources of various types
- Continually searching through one massive region of memory for usable free blocks for all possible kernel allocations is too costly
 - In other words, `malloc` wouldn't work well for the kernel
- The kernel uses a special meta-level allocator that dedicates memory regions to specific purposes (and element sizes)
 - Since the time to find a suitable memory region is reduced, the kernel can quickly allocate and free objects of specific purpose and size
- The kernel can be configured to use one of three main allocators
 - SLOB, SLAB, and SLUB
 - Let's talk about these



Kernel Memory Allocators

- Simple List of Blocks (SLOB)
 - Oldest
 - Compact memory usage
 - Fragments quickly
 - Requires traversing a list to find the correct size allocation
- Solaris Type Allocator, SLAB
 - Cache friendly
 - Complex structures
 - Lots of queues
 - Queues per CPU, and per node
 - Exponential growth of caches



SLUB

- SLUB, the Unqueued allocator
 - The newest allocator, and the default since Linux 2.6.23
 - Works with the existing SLAB API
 - No more complex queueing
 - Execution time friendly
 - Enables runtime debugging and inspection
 - The allocation/freeing fastpath doesn't disable interrupts



kmem_caches

- SLUB operates on caches, of type struct kmem_cache
- kmem_caches allow the allocation of only **one** size/type of object (read: struct)
 - They can have custom constructors and destructors
- Each cache has its own name, object alignment, and object size
 - A cache's name usually reflects the name of the structs it holds
- See `/include/linux/slab.h` for API
 - `kmem_cache_create`
 - `kmem_cache_destroy`
 - `kmem_cache_alloc_node`
 - `kmem_cache_free`



Slabs

- A `kmem_cache` internally uses slabs
- Slabs, the data containers, should not be confused with SLAB, the allocator
- A slab is a collection of one or more contiguous pages that contain objects of a certain size
 - Pages within a slab are organized into a linked list
- In the kernel, a slab and its pages are both defined by `struct page`
 - A slab's data is part of a union in the page struct, to reduce memory usage
 - A slab contains:
 - A pointer to the first free object, `freelist`
 - Meta data for bookkeeping (number of objects and pages, and how many are used)
 - A pointer to partially-full slabs, `next`
 - See `/include/linux/mm_types.h`

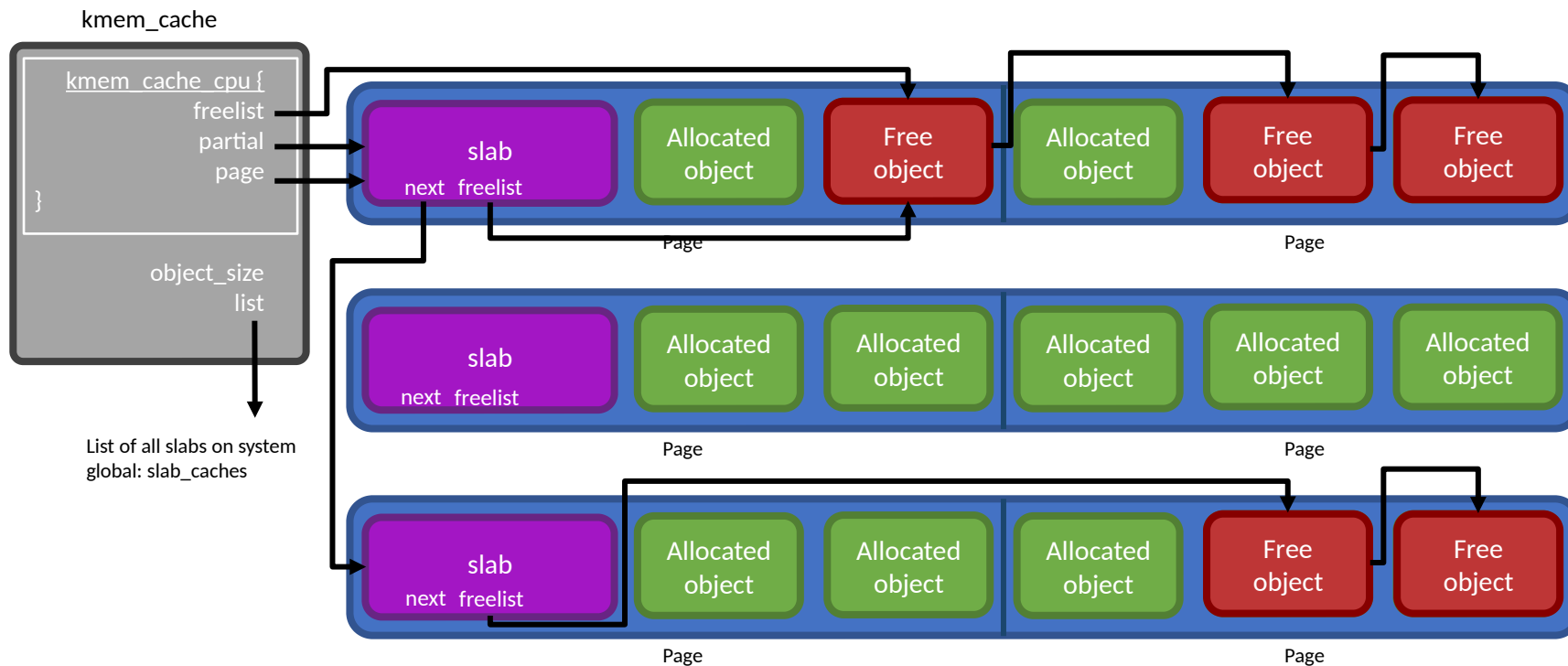


Slab Usage

- A `kmem_cache` has numerous associated slabs
- Slabs can be empty, partial, or full
- When a new object needs to be allocated in a `kmem_cache`, it scans lists of partial slabs to find a location for the object
- If no partial slabs exist, a new empty one is created, a new object is created within it, and the new slab is marked partial
 - The underlying low-level page allocator (`alloc_pages`) is used for this
- If full slabs become not full, they are added back to the partial list



Slab Memory Layout



SLUB Operation

- The SLUB allocator creates a `kmem_cache_cpu` per CPU
 - Object allocations are attempted from the local free list first
 - If that fails, allocations are attempted from other slabs on the current CPU
 - If that fails, allocations are attempted from the partial slabs of the cache
 - This is the slow path and requires a lock
 - If it fails to find any free object, SLUB will allocate another slab
 - Caches can be told to fail here, instead of growing
- The SLUB allocator can merge similar caches into the same cache
 - This can save space and time
 - It also makes exploitation easier
 - How?



Slab Usage

- See slab usage info in `/proc/slabinfo`
 - `slabtop` is an `htop`-like view into this info
 - See `man 7 slabinfo`
- In this list you will see lots of entries for `kmalloc`
 - `kmalloc` uses `kmem_caches` behind the scenes



kmalloc

- `kmalloc` keeps arrays of `kmem_caches`
 - `kmalloc_caches[kmalloc_type(flags)][kmalloc_slab(size)]`
 - Type boils down to:
 - `KMALLOC_NORMAL`
 - `KMALLOC_RECLAIM`
 - `KMALLOC_DMA`
 - `kmalloc` will inline itself, and takes a different path when a constant is used as the operand
 - Uses `__builtin_constant_p(size)` to know if something is a constant
- Very large allocations are handled by `kmalloc_large`
 - Which just calls `alloc_pages` behind the scenes
- Use `virt_to_phys` to get a physical address for a `kmalloc` allocation



kmalloc

- The flags to `kmalloc` help choose which `kmalloc` bin it will go in
 - Normal, reclaimable, and DMA
- The flags also tell `kmalloc` how to allocate the object
 - `GFP_NOWARN`
 - `GFP_ATOMIC` (opposite of `GFP_NOFAIL`)
 - `GFP_ZERO`
- Like the rest of kernel memory in Linux, this memory is not pageable



vma1loc

- There are other allocation APIs in addition to slab
 - vma1loc is used to allocate large buffers, larger than kma1loc can
 - But it can be slower, and the contents are not necessarily contiguous
- Use vfree to free
- Use vma1loc_to_pfn to get a physical page number for this address



Getting whole pages

- To allocate pages, use `alloc_pages`
 - Helper functions like `get_fre_pages` or `get_zeroed_page` exist
- To pin usermode pages, `get_user_pages`
 - This is not an allocation, but a way of locking
- To remap kernel pages to usermode, use `remap_pfn_range`
 - Make sure to `SetPageReserved`
- To make bus memory CPU accessible, use `ioremap`
- To map kernel pages into the kernels address space, use `kmap`



KASAN

- If the kernel is built with CONFIG_KASAN, the Kernel Address Sanitizer (KASAN) will be enabled
- KASAN is an error detector that can catch logs of dynamic memory bugs
 - This includes use-after-free and out-of-bounds uses of kernel memory
- KASAN will instrument code at compile time to check every memory access
- It can output a stack trace when it detects a problem
- See also: kmemleak
 - Finds memory leaks and reports in /sys/kernel/debug/kmemleak
 - CONFIG_DEBUG_KMEMLEAK has to be enabled at build time
- Another tool is UBSAN
 - Undefined Behavior Sanitizer
 - Watches for undefined behavior in the kernel



LINUX CNO Programming



Lab 9

Memory lab

Lab



- Create a driver that supports the mmap and the poll file_operations
- Map a buffer into the user's address space, and alert them when the buffer has changed
- Change the buffer to contain the name of any users with an open file to the device
- Alert the users via poll when changes are made



LINUX CNO Programming



The Virtual File System

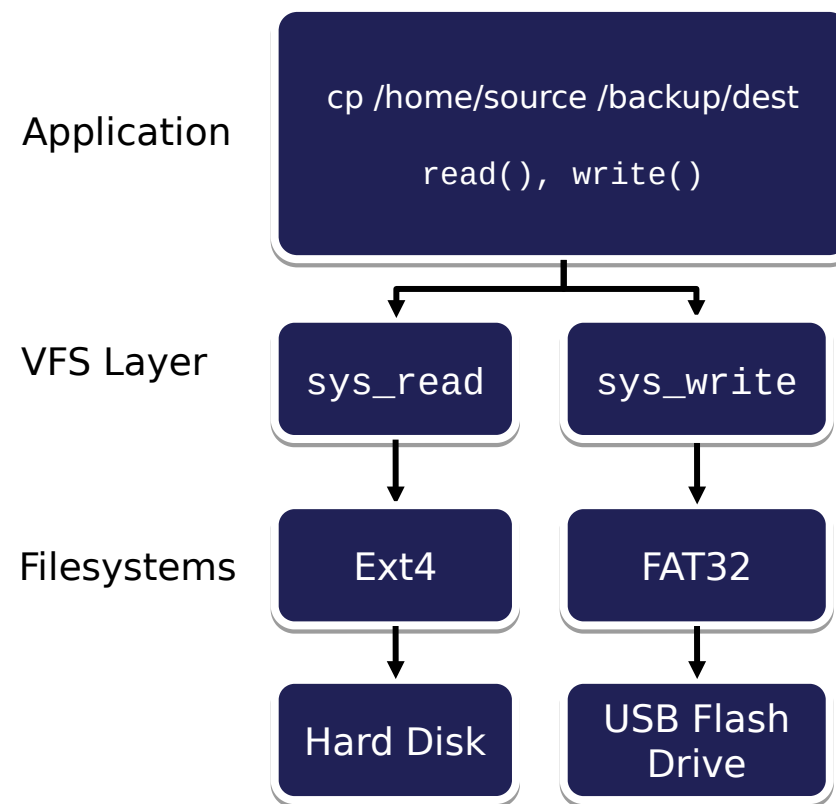
The Virtual Filesystem

- The Virtual File System (VFS) layer of the Linux kernel allows user applications to use a single interface for all file I/O operations, and the kernel to do the same internally
- To facilitate this functionality, file system drivers provide the kernel with their implementations of a set standard functions that the kernel uses to complete file-related tasks
- As a result, the kernel can support a variety of different file systems (Ext4, FAT32, NTFS, etc.) without having any knowledge of their internal workings
- This also allows easy interoperability between file systems of different types (think about cp between a USB drive and hard drive)



VFS Layer

- Under the VFS, applications can be written against the standard file API system calls
- Internally, the kernel can call functions with a standard signature, exported by the file system driver code
- In this way, kernel code and userspace code are filesystem agnostic



File System Support in the Kernel

- Support for a few file systems are included in the kernel mainline
 - EXT4, CIFS, XFS, FUSE, NFS, NTFS, etc.
- Others are implemented as external kernel modules which can be included at runtime
- Registering a new file system with the kernel is (relatively) easy
 - `register_filesystem` in `fs/filesystems.c` (include `linux/fs.h`)
 - Takes a pointer to a `file_system_type`
 - Adds the file system to the global `file_systems` array
 - Eventually calls `sget_userns`, which creates (or finds) a `super_block` structure
 - Iterates over `fs_supers` member of the `file_system_type`



Object Oriented Concepts

- Even though C does not have built-in support for object oriented programming (OOP), the kernel stills uses it internally
- One example of this is within the VFS layer
- The VFS layer defines various structs, and these structs contain
 - members (analogous to class variables)
 - function pointers (analogous to class methods)
- Each file system creates their own instance of these structs, defining their own values for members and functions
 - The kernel has some generic versions of these functions for file systems that do not require special behavior
 - In many cases, the first parameter to these functions is a pointer to the struct that holds them (analogous to a this pointer)
- The kernel operates on all these structs the same way, creating a form of polymorphism



VFS Objects

- The main structs (think objects) that the kernel uses for the VFS are declared in `linux/fs.h`:
 - `super_block` (file system meta data)
 - Primary function pointers within `s_op` (`super_operations`) member
 - `inode` (index node)
 - Primary function pointers within `i_op` (`inode_operations`) member
 - `dentry` (entries in the dirent cache)
 - Primary function pointers within `d_op` (`dentry_operations`) member
 - `file`
 - Primary function pointers within `f_op` (`file_operations`) member



Superblocks

- A struct `super_block` contains
 - The file system's block size
 - Methods for operating on the superblock (`s_op`)
 - A pointer to a dentry for the root directory (`s_root`)
 - And a bunch of other things
 - A UUID
 - A max file size



inodes

- inodes contain data and operations needed by the operating system to interact with all files and directories (directories are files)
 - Native file systems actually store inode data on disk where they can be directly read into memory and used
 - Other file systems must emulate inodes to provide a familiar API to the operating system
- All files, including special files such as pipes and devices in /dev have an inode when represented in memory
 - Everything in /proc and pseudo filesystems does too
- See struct `inode` in `include/linux/fs.h`



dentrys

- In the path, /home/project/file.txt, four components are represented (/ , home, project, and file.txt)
 - Each of these is represented in memory by the VFS as a struct dentry
 - Including the file.txt!
- Directory entries are not stored on disk, but are created and used by the VFS during runtime as needed
- These are stored in and managed by a cache named dcache
- This structure makes it easier to look up information and use paths, without having to do string manipulation
- See struct dentry in include/linux/dcache.h



dentry Internals

- A dentry contains
 - A pointer to its parent dentry (`d_parent`)
 - The kernel uses this to chain dentries together than constructing a path
 - A pointer to its corresponding inode (`d_inode`)
 - For both files and directories (directories are files, remember?)
 - A name (`d_name`)
 - This is not part of the inode – remember, multiple paths can refer to the same file
- Don't confuse dentry with `dirent`
 - dentries represent components of a path
 - `dirents` are contained by directory files on disk to represent their contents



Files

- A file object represent an opened file
 - Contains information regarding the current state of the open file (position, flags, etc.)
 - Contains function pointers (in `f_op`) for the kernel to invoke for I/O operations (read, write, open, close, etc.)
- See `struct file` in `include/linux/fs.h`



LINUX CNO Programming



Lab 10

Procfs

Tasks

- `readdir` allows userspace programs to enumerate directory contents
 - `ps` uses it to list processes from `/proc`
- Find out how `readdir` works (syscall is `getdents`)
 - Located in `/fs/readdir.c`
- Use your knowledge to remove a process of your choice from the listing provided by `ps`
- You will need to follow pointers in various VFS structures and overwrite certain function pointers to obtain your goal



LINUX CNO Programming



Sysfs

Sysfs



- A file system that has information about the kernel and it's systems and drivers to userspace via virtual files
- Works on kobjects and ksets
 - Kobjects are used for reference counting and heierarchy management
 - Character drivers have a kobject, so do lots of kernel components
 - Ksets are collections of kobjects, and are always associated with a sysfs directory
- Sysfs allows drivers and devices to create attribute files
 - Attribute files can be read and written to, as a way to configure kernel components
 - Binary attribute files exist as well



Sysfs *(continued)*



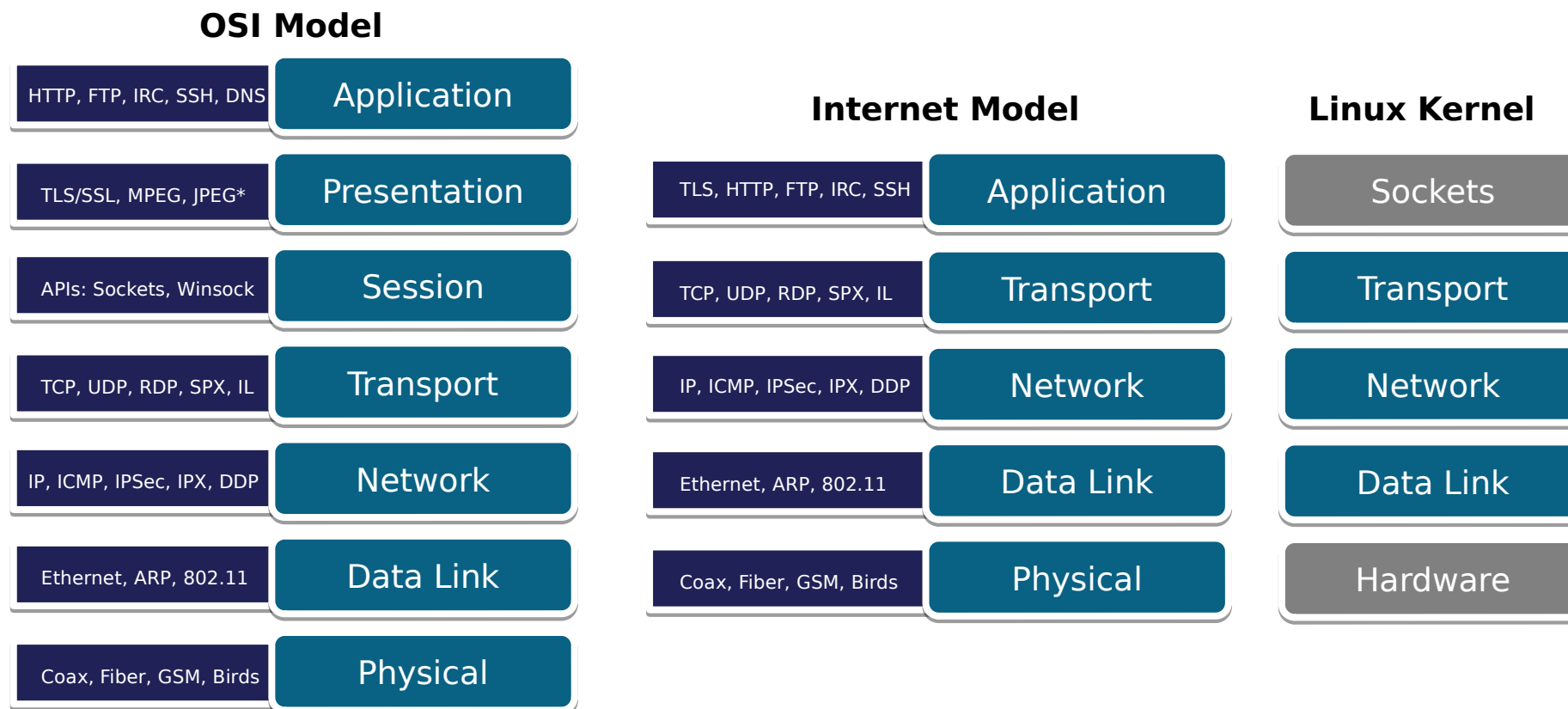
- There are helper function that will work with sysfs
 - `misc_register`
 - Creates a miscellaneous character device, registers it with sysfs
 - Will automatically have a device file created for it, and a maj/min number
 - `class_create`
 - `device_create`, `device_register`, `device_add`
 - `device_create_file`
 - Creates an attribute file
- Udev
 - Udev watches for changes in sysfs, and supplies the system with event info
 - This is how the file manager know when you plug in a usb device
 - Successor of hotplug and devfsd
 - Udev will also automatically create a device file in the `/dev` directory for new devices





Network Subsystem

OSI and Internet Models



*The presentation, session, and application layers are ill-defined at best, and this diagram is an amalgam of a few popular ones. The upper layers of the OSI model don't make much sense, and never really did



Network Subsystem

- As you know from the Linux System Programming and Linux Internals series, the POSIX network API is used to handle a variety of communication protocols
 - IP, TCP, UDP, Netlink, BlueTooth, raw packets, Unix domain sockets, etc.
- All of the POSIX socket APIs functions, with the exception of `getaddrinfo`, are implemented by system calls provided by Linux
 - Glibc provides thin wrappers for C programs and other languages that use it
- The kernel implements the polymorphic behavior of handling all the different protocols through the singular socket API



When you call socket

- When userspace calls the socket system call...
- The sys_socket system call handler in /net/socket.c is invoked
- A struct socket is created and sock_create is called
- The protocol family, as specified by the first parameter (e.g., PF_INET), is looked up in the net_families array
- The net_families array contains an entry for every registered protocol family
- And each of these has a function pointer that tells the kernel how to initialize sockets for that family

```
struct net_proto_family {  
    int family;  
    int (*create)(struct net *net, struct socket *sock, int protocol, int kern);  
    struct module *owner;  
};
```



Registering a Protocol Family

- `sock_register` is used to add support for a new protocol family
 - New protocols supply an identifier and implementation of the create function seen previously
- Example: during initialization, the kernel calls `init_inet` from `/net/ipv4/af_inet.c`, which registers protocols that run on IPv4 (TCP, UDP, Ping, and Raw) and then registers the `PF_INET` protocol family using `sock_register`
 - The struct `net_proto_family` used by `PF_INET` is `inet_family_ops`
 - The create function referenced in `inet_family_ops` is `inet_create`



Back to Calling socket

- After getting a pointer to the proper struct `net_proto_family`, the kernel calls the associated create function (`inet_create` for `PF_INET`)
- A pointer to the socket structure is passed as a parameter to the create function, and has had its type (e.g., `SOCK_STREAM`) set
- The protocol (e.g., `IPPROTO_TCP`) is also passed as a parameter to the create function
- At this point the create function for the protocol family knows both the type of socket to create and the desired protocol (if specified)



PF_INET's create

- `inet_create` initializes a struct `sock` named `sk`
 - struct `sock` is a network-layer socket representation
 - Try not to confuse this with the struct `socket` named `sock`
- `inet_create` looks up all the protocols it has for the sock's type (e.g., `SOCK_STREAM`) in `inetsw`
- `inetsw` is an array of lists of protocols, indexed by type
- Each entry is a struct `inet_protosw`

```
struct inet_protosw {
    struct list_head list;
    /* These two fields form the lookup key. */
    unsigned short    type;      /* This is the 2nd argument to socket(2). */
    unsigned short    protocol; /* This is the L4 protocol number. */

    struct proto *prot;
    const struct proto_ops *ops;
    unsigned char    flags;
};
```



Registering a Protocol

- During initialization, `inet_register_protosw` is called to register various protocols and associate them with `inet_sw`
 - See `/net/ipv4/af_inet.c` `inet_init`
- The protocols registered are contained statically, within the `inet_sw_array`
- Other protocol families (e.g., `PF_UNIX`, `PF_INET6`) have their own (similar) methods of registering their protocols, if needed
- Each `inet_protosw` entry in `inet_sw_array` contains



Back to PF_INET's create

- PF_INET's create will simply select the first item in the list for the given socket type if a protocol was not explicitly specified
- Otherwise it will make sure it has the desired protocol, and fail if not
- There is only a single entry in `inet sw` for the `SOCK_STREAM` type:

```
{  
    .type =      SOCK_STREAM,  
    .protocol =  IPPROTO_TCP,  
    .prot =      &tcp_prot,  
    .ops =       &inet_stream_ops,  
    .flags =     INET_PROTOSW_PERMANENT |  
                INET_PROTOSW_ICSK,  
},
```

- This means that `socket(PF_INET, SOCK_STREAM, 0)` will return TCP (as well as specifying `IPPROTO_TCP` directly)

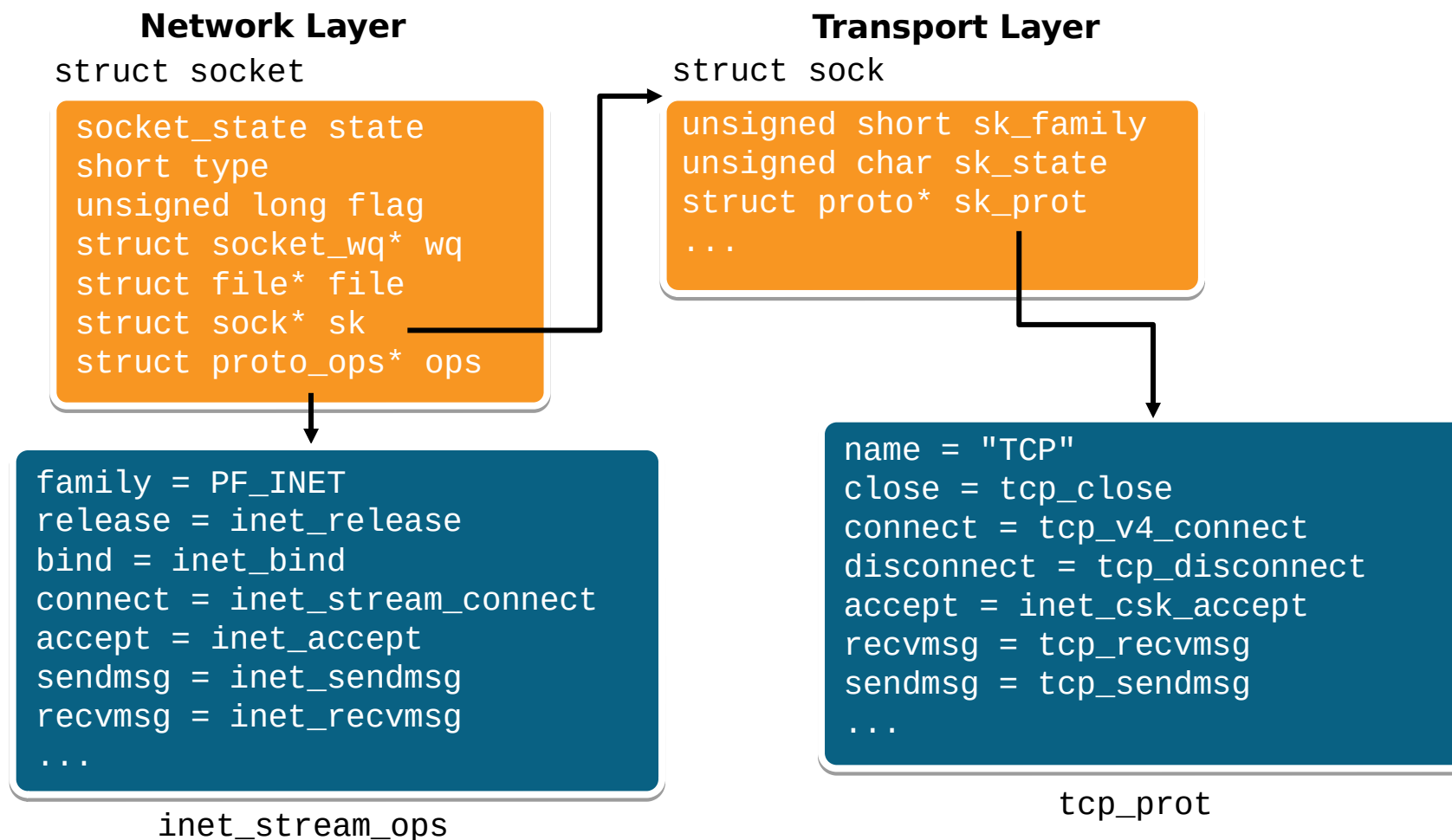


Finishing PF_INET's create

- `inet_create` sets the `ops` member of the struct `socket` to the one specified in the resolved protocol (`&inet_stream_ops` for `SOCK_STREAM`)
- It also sets the `sk_prot` member of its internal struct `sock`, `sk`, through a call to `sk_alloc`
 - This is `&tcp_prot` for PF_INET TCP sockets
- The `init` function specified in `tcp_prot` is then called to do any special protocol-specific initialization on the socket (if it exists)
 - `sk->sk_prot->init(sk)` in `/net/ipv4/af_inet.c`
- The `inet_stream_ops` and `tcp_prot` structures store function pointers to all the kernel functions that handle IP stream type and TCP functions, respectively
- Specifying `SOCK_DGRAM` and `IPPROTO_UDP` would have pointed the socket structures to `inet_dgram_ops` and `udp_prot`



What did this Accomplish?



What Happens when we call recv?

- When userspace calls the recv system call...
- The sys_recv system call handler in /net/socket.c is invoked
- Which then calls __sys_recvfrom
 - Which is also called by the recvfrom system call. Why?
- Which then calls sock_recvmsg on the socket
- Which simply calls sock->ops->recvmsg, and passes it the socket
- The specified operations that were set up during the socket system call take it from there
- Similar flows exist for send, close, accept, etc.



What Happens in those ops?

- `inet_recv_msg` is the inet stream operation for `recv`
- It calls the underlying `sk_prot->recvmsg` (`tcp_recvmsg` for `IPROTO_TCP`) almost instantly
- `tcp_recvmsg` does the majority of the work
 - Iterates through a list of buffers in its receive queue
 - Reads the data it can from each buffer
 - `skb_copy_datagram_msg` is called on each buffer to copy data from the network to the provided userspace buffer(s)
 - Continues until it reads a desired amount of data
- Socket network functions operate on a struct `sk_buff`



sk_buffs

- `struct sk_buff` (“Socket Buffer”) is the basic unit of operation within the network subsystem and is defined in `/include/linux/skbuff.h`
 - Commonly abbreviated `skb`, both in text and in code
- Within the kernel, the input and output buffers of any socket type are lists of `sk_buffs`
- `sk_buffs` are operated on from the highest layers of kernel socket functionality, through the individual protocol layers, and to the actual device drivers that communicate with hardware
- `skb->data` points to the data that will be sent (or that has been received)
- `skb->len` contains the data length



sk_buff Internals

- sk_buffs are used by every layer's processing
- Each has
 - a link to the next and previous sk_buff (top of structure)
 - inner_transport_header (encapsulated transport layer header)
 - inner_network_header (encapsulated network layer header)
 - inner_mac_header (encapsulated link layer header)
 - network_header (network layer header)
 - mac_header (link layer header)
 - a timestamp for when the packet was sent or received
 - a pointer to the net_device that will send or that received the packet
 - a pointer to the owning socket
 - and a ton more



sk_buff Processing

- During initialization, the kernel creates a series of queues for each CPU to store sk_buffs for processing
 - See `net_dev_init` in `/net/core/dev.c`
- It also registers two softirq (bottom half processing routines)
 - `open_softirq(NET_TX_SOFTIRQ, net_tx_action);`
 - `open_softirq(NET_RX_SOFTIRQ, net_rx_action);`
- These softirqs interact with queues of skbuffs to be processed (and will run the next time `do_softirq` is called)



sk_buff Processing *(continued)*

- Before passing on a received packet to the upper layers, drivers must specify the `dev` and `protocol` members of an `sk_buff`
 - Driver developers can use `eth_type_trans` function to assist in assigning the protocol
- Based on these values, the kernel's network reception function (`net_rx_action`) can decode packets and process them correctly
- `struct packet_types` define the proper function to invoke based on the protocol type
 - Types are registered with the `dev_add_pack` function during initialization
 - `ip_packet_type` in the kernel specifies the function `ip_rcv` for type `ETH_P_IP`
 - IP packets are sent to `ip_rcv`



sk_buff Processing *(continued)*

- `ip_rcv` can handle packets not destined for the local machine (and forward them on, as a router)
- If the IP packet is meant for the local machine, it will perform additional processing with `tcp_v4_rcv` (for a TCP connection)
- When TCP has finished its parsing of the packet (into TCP segment(s)), the `sk_buff` is added to the struct `sock` `sk_receive_queue` (See `/net/ipv4/tcp_input.c`)
- On the sending side, `tcp_transmit_skb`, will transmits packets enqueued by `tcp_sendmsg` (see `/net/ipv4/tcp_output.c`)
 - After processing This function will pass the `sk_buff` to the IP layer for further processing
 - `ip_queue_xmit` will be invoked, which will begin the IP-layer processing

Note that with TCP and some other connection-based protocols, receives can trigger sends (e.g., for ACKing data received)



Communicating with Network Drivers



- For sending, the kernel uses `dev_queue_xmit` to send an `skbuff` to a device, which calls the `hard_start_xmit` function defined by the device associated with the `sk_buff`
- When receiving, `netif_rx` is used by a driver to queue a received `sk_buff` to an upper-layer protocol for further processing
- `netif_receive_skb` can be called for polling new packets, in `softirq` context
- `net_devices` live in a linked list (`eth0`, `eth1`, etc.) that the kernel can iterate over to view hardware information, MAC information, and network layer



Network Drivers

- Network drivers are responsible for translating the raw packet data that comes from the hardware (such as an Ethernet interface) into `sk_buffs` and passing them to the higher layers
- Similarly, they must receive `sk_buffs` scheduled for transmission and send them to the hardware in a format that makes sense
- Throughout its lifetime, `sk_buff` data can go through the whole kernel network subsystem without ever being copied
- Drivers that support DMA allow an `sk_buff`'s data to be written to directly when receiving packets
 - Otherwise a single `memcpy` can be used to transfer the hardware's data to the `sk_buff`
- The only copy that has to occur is the one that happens when data is being copied to and from userspace for `send` and `recv` syscalls

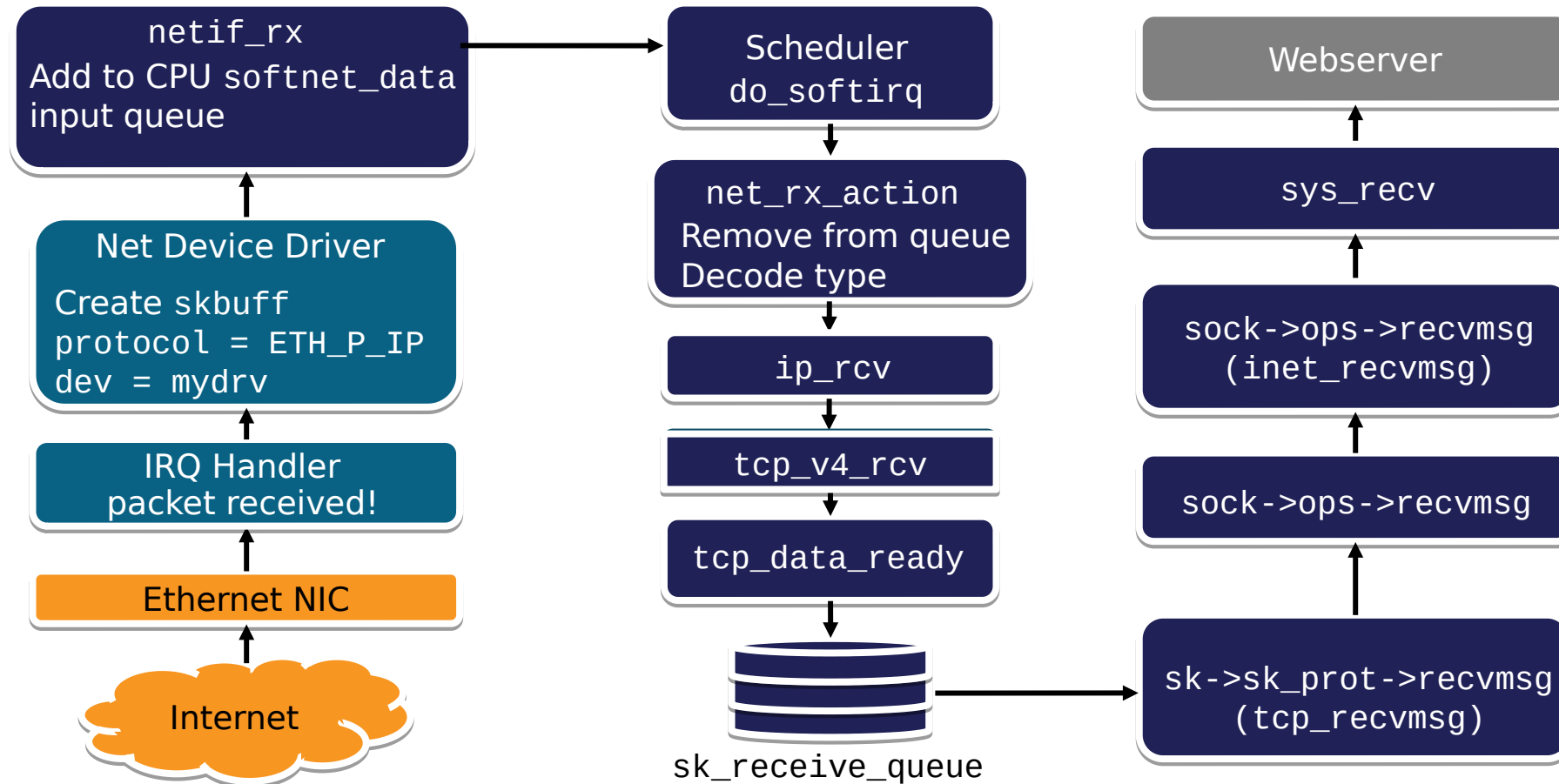


Network Drivers *(continued)*

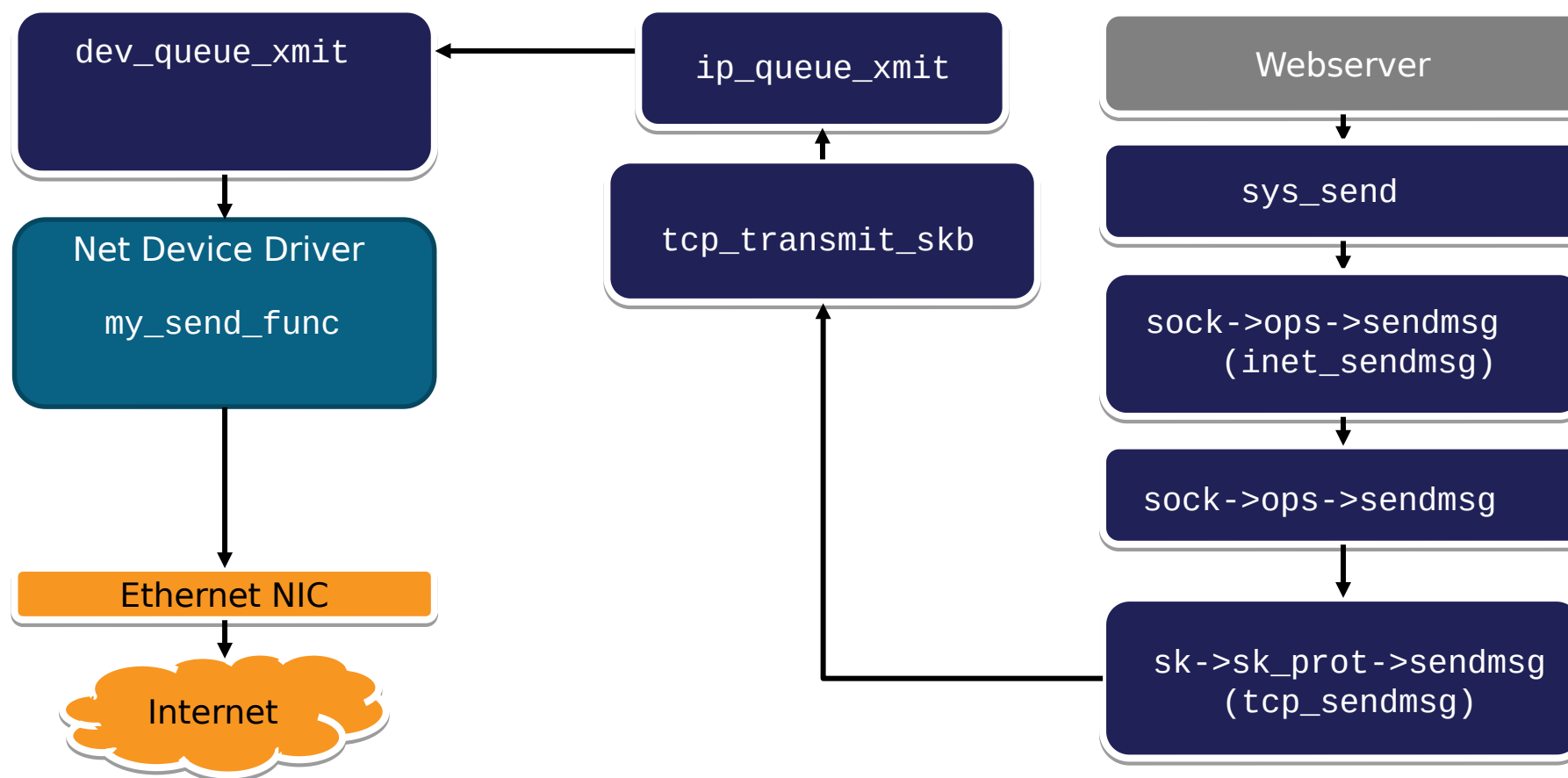
- Drivers must implement an interrupt handler that will be invoked when network data has been sent or received
- Drivers can also define a polling function to handle high volume traffic that will allow the kernel to poll for network data rather than receiving a horde of interrupts (see NAPI)
- They can control kernel behavior with an API that controls network packet queues
 - `netif_start_queue` to start handling packets
 - `netif_wake_queue` to resume handling packets
 - `netif_stopqueue` to stop handling packets



Putting it all Together – TCP RX



Putting it all Together – TCP TX



Caveats

- The previous slides show some possible (and fairly common) paths through kernel for the reception and transmission of TCP/IP packets
- Different paths are possible due to
 - Congestion (queuing)
 - Error cases (dropped packets)
 - Different routing
 - Different driver configurations (NAPI)
 - Etc.
- Different, but similar paths, are taken by other protocol families (PF_UNIX, PF_INET6, PF_BLUETOOTH, etc.)
- Different, but similar paths, are taken by other protocols (IPPROTO_UDP, IPPROTO_ICMP, etc.)



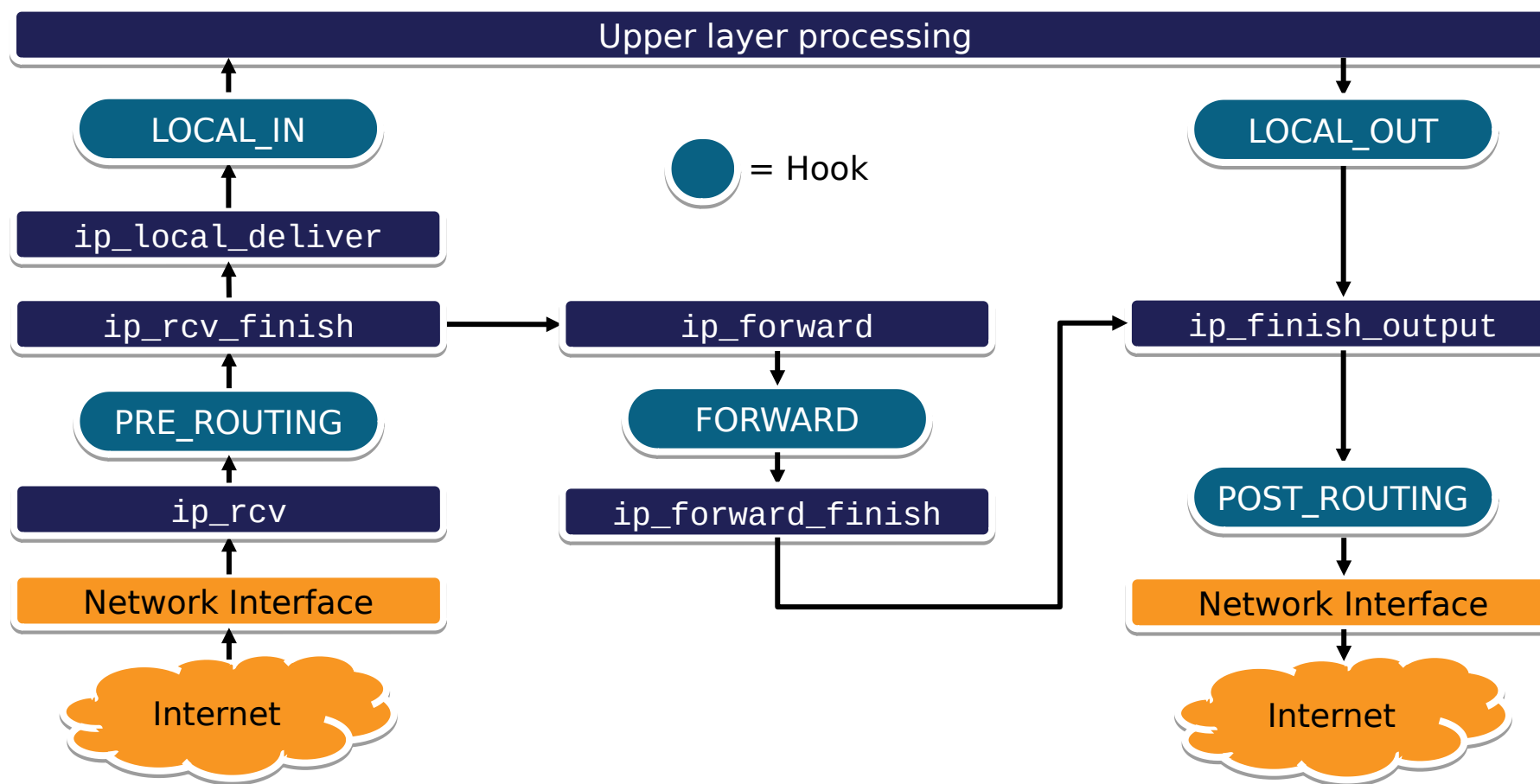
Netfilter

- Netfilter is a kernel API that allows the network subsystem to be hooked
- This allows security monitoring, packet mangling, selective packet forwarding, and other services
 - This is the API used by iptables
- Explicit hooks are placed in strategic points throughout the network subsystem
 - See NF_HOOK, NF_HOOK_LIST, and NF_HOOK_COND
- Each of these hook locations has a name:

```
enum nf_inet_hooks {  
    NF_INET_PRE_ROUTING,  
    NF_INET_LOCAL_IN,  
    NF_INET_FORWARD,  
    NF_INET_LOCAL_OUT,  
    NF_INET_POST_ROUTING,  
    NF_INET_NUMHOOKS  
};
```



Netfilter Hook Locations



Using Netfilter

- The API is defined in /linux/include/netfilter.h
- Also reference /include/uapi/linux/netfilter.h
- You define a hook using an instance of struct nf_hook_ops:

```
struct nf_hook_ops {  
    /* User fills in from here down. */  
    nf_hookfn      *hook;  
    struct net_device *dev;  
    void          *priv;  
    u_int8_t      pf;  
    unsigned int   hooknum;  
    int           priority;  
};  
  
typedef unsigned int nf_hookfn(void *priv,  
                                struct sk_buff *skb,  
                                const struct nf_hook_state *state);
```



Using Netfilter *(continued)*

- Register hooks

```
int nf_register_net_hook(struct net *net, const struct nf_hook_ops *ops);
```

```
int nf_register_net_hooks(struct net *net, const struct nf_hook_ops *reg, unsigned int n);
```

- Unregister hooks

```
void nf_unregister_net_hook(struct net *net, const struct nf_hook_ops *ops);
```

```
void nf_unregister_net_hooks(struct net *net, const struct nf_hook_ops *reg, unsigned int n);
```



LINUX CNO Programming



Lab 11

Netfilter

Tasks

- Create a kernel module that uses netfilter hooks
- Handle IPv4 and IPv6 packets
- Your hook should do the following for any data that goes to the local machine
 - If it's ICMP, drop it
 - If it's TCP, check the payload
 - If the start of the payload is "hello" change it to "HELLO"





Security Modules

Linux Security Modules (LSMs)

- At the 2001 Linux Kernel Summit it was proposed that SELinux be adopted into the Linux 2.5 Kernel
- Linus Torvalds initially rejected SELinux because the security community had not yet formed a consensus on which security model was best. He suggested making it a module to allow different security mechanisms to implement a single kernel interface, allowing for flexibility
- In 2002 at the USENIX Security Symposium, Morris et al. detailed the architecture and design philosophy of the LSM framework in the kernel
- This framework allowed numerous new types of access control and security policies to be deployed using the same interface in the kernel, providing flexibility for administrators and the community at large to pick and choose which security models best fit their needs



LSM Hooks

- The LSM framework provides hooks in the kernel that call out to an installed security module right before sensitive accesses are made
- Putting hooks close to the access allows the modules to obtain a maximal amount of context before making a decision
- The hooks that occur before a sensitive function call often have the same parameters as the function call itself
- If the hook returns a failure, the sensitive call is not made and an error is returned instead
- Thus hooks tend to be restrictive in nature, rather than permissive



LSM Hooks *(continued)*

- A typical LSM hook looks like the following

```
int sock_recvmsg(struct socket *sock, struct msghdr *msg, int flags)
{
    int err = security_socket_recvmsg(sock, msg, msg_data_left(msg), flags);

    return err ?: sock_recvmsg_nosec(sock, msg, flags);
}
EXPORT_SYMBOL(sock_recvmsg);
```

- Modules implement a function with the same prototype as the bolded function and register it with the LSM framework to provide a hook
- It is common to see the following function's name end in `_nosec`



Making a Module

- The LSM API is in `/include/linux/security.h`
- Also `/include/linux/lsm_hooks.h`
 - This contains a list of functions that an LSM can implement
 - And the `LSM_HOOK_INIT` macro that LSMs call to set a hook
 - `security_delete_hooks` - disable hooks
 - `security_add_hook` - enable hooks
- LSMs can define their own data types and associate them with various system structures
- The LSM framework allows modules to define allocation and free functions for their data
 - For example:
 - `inode_alloc_security`
 - `sb_alloc_security`
 - `sk_alloc_security`
 - `shm_alloc_security`



Capabilities

- LSMs can also implement POSIX1.e capabilities
 - Example: CAP_NET_ADMIN, CAP_SYS_ADMIN,
- Throughout the kernel you will see calls to `capable`, `has_capability` and others :

```
int mm_account_pinned_pages(struct mmpin *mmp, size_t size) {  
    unsigned long max_pg, num_pg, new_pg, old_pg;  
    struct user_struct *user;  
  
    if (capable(CAP_IPC_LOCK) || !size)  
        return 0;  
    ...  
}
```

- See `/include/uapi/linux/capability.h` for a list of supported capabilities
- These calls will return false if given capabilities are not held by the current process



Mainlined LSMs

- In the /security directory you will find a variety of LSMs that have been accepted into the mainline kernel
- These include
 - SELinux, AppArmor, Smack, and TOMOYO
- Take a look at their code to see the kinds of things that such systems monitor



SELinux

- Security-Enhanced Linux (SELinux)
- Many of the underlying concepts of SELinux were developed by the NSA Security-Enhanced Linux team
- Reduces privileges of programs to the minimum required to work by restricting access to system resources
- Enhances traditional Linux (discretionary) access control mechanisms
- Places emphasis on correctness of the kernel and its security-policy configuration, not on the correctness of all privileged applications and their configurations (DAC vs MAC)



AppArmor



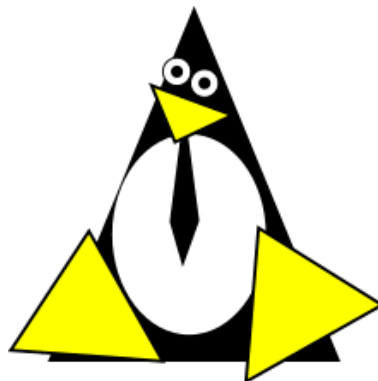
- Path name based access control scheme to restrict applications
- Per program restrictions that can be configured by a system administrator
- Allows the administrator to control access to things like networking, sockets, files, etc
- Provided as an alternative to SELinux, which has a bad reputation for being difficult to configure/manage
- Same general approach as SELinux of introducing MAC to a traditionally DAC model



SMACK



- SMACK lets the administrator define labels for kernel objects and tasks. If the labels do not match, that task is denied access to that object
- Uses filesystem extended attributes to store labels
- Simplified Mandatory Access Control Kernel (SMACK) tries to be a simpler SELinux by implementing MAC, but purposefully leaving out role based access and type enforcement



TOMOYO



- MAC implementation which allows behaviors and resources necessary for a program to run to be applied to a process
- Can monitor processes to see what “normal” behaviors are.
 - Useful even just as a system analysis tool
- Capable of automatic real-time policy generation
- When protection is enabled, processes are restricted to designated behaviors/resources/policies.



LINUX CNO Programming



CNO in the Kernel

Moving on...

- The previous sections we covered the major subsystems of the Linux kernel
- The usermode CNO class also covered some basic CNO techniques
 - Hooking
 - Injection
 - Hardening
 - Hiding
- In this section, you will be using your background in these other two classes
- There will be less time spent in lecture and much more time spent on labs
- The following labs will provide less hand-holding
- Use what you have learned!



CNO in the Kernel

- All of the concepts from usermode CNO apply to CNO in the kernel as well
- Hooking, injection, hiding, hardening, etc. are all feasible
 - Many of the same methods apply as well
- Since the kernel runs in ring 0, having our code run within it provides us with certain options not possible in userspace code
 - Directly interfacing with hardware
 - Directly influencing arbitrary processes
 - Easily reading page table data
 - Etc.



Pros of Kernel CNO

- Pretty much everything eventually goes through the kernel
 - Process creation
 - Memory accesses
 - Hardware interaction
 - File system data
 - Network data
- The kernel has full permission over everything on the system
 - Ring 0 means we can do anything we want (except in some cases like with SGX)
- Can manipulate other subsystems and hide from them
 - Can hide processes
 - Can hide files
 - Can hide from performance monitors



Cons of Kernel CNO

- Kernel code development is often harder than usermode development
 - Errors can cause catastrophic system failures and therefore be more noisy
- Sometimes hiding in plain sight is the best place
 - Making modifications to the kernel can set off a lot of red flags
 - Dodging these may be really hard against more hardened kernels (see first point)
- Some tasks don't require the kind of power the kernel gives you
 - So don't use it



LINUX CNO Programming



Using Events

Events

- Intercepting or registering for events in the kernel is a good way to gain information about a running system
- Subsystems routinely communicate with other subsystems through memory, explicit event mechanisms, and other methods during normal system operation
- Sometimes being a part of these can seem completely normal
- Some easy methods to place yourself in the middle of routine events are
 - Creating a shared handler for an IRQ
 - Registering for notifications using the notifier API



Shared IRQs

- Linux allows IRQs to be shared by multiple handlers
- In this case, `request_irq` must have the `IRQF_SHARED` flag set*
 - All other handlers registered for that IRQ number must also have this flag set
 - This is the case with interrupt #1, the keyboard interrupt
- When an interrupt with multiple handlers is invoked, the kernel iterates through each of them, passing each its own `dev_id` parameter that it used when registering with `request_irq`
- If the handler is going to service the interrupt, then it should return `IRQ_HANDLED`, otherwise it should return `IRQ_NONE`*

*but technically, it doesn't matter



Reading from Hardware

- Many times it's not sufficient to get an interrupt handler invoked to know what to do next
- Interrupt handlers typically have to obtain data from some device that their drivers registered with the system
- Along with devices, IO ports are accessible via the in and out instructions
 - (see `/arch/x86/include/asm/io.h` for some macros for `inb` (input byte) `outb` (output byte) functions)
- The keyboard device has mappings to ioports
 - see `sudo cat /proc/ioports`
- You can read and write data from and to these IO ports using the in and out instructions (or their Linux wrapper functions)
- Other devices may require more complicated interactions (like PCI ones)



IRQ Snooping

- If you find a handler that is registered as a shared handler for a given IRQ (or if you somehow modify it so that it is shared and won't break by becoming such) you can register your own handler for the same IRQ
- Return `IRQ_NONE` to not interfere with normal processing
- Inspect the correlated data to see what's going on
- Let's do this!
- Keylogger time



LINUX CNO Programming



Lab 12a

Keylogger - IRQ

Tasks

- Create a module that shares an IRQ line with the keyboard (IRQ 1)
- Figure out port numbers for keyboard IO data
- Write an interrupt handler that gets invoked on keystroke and reads the key data from the keyboard
 - Register it with your module
- For now, simply print out the key data (scancode) from the keyboard using `printk`



Tasks *(continued)*

- Once you have a working interrupt handler printing out the keyboard scancodes, modify your module to create a character device called “ghostboard”
- This character device only needs to specify a read file operation function, but feel free to add others
 - Use major ID 246 and minor ID 135
- Your keylogger module should insert scancodes into a ring buffer and your read function should remove from the ring buffer
- Create a userspace program that interacts with your “ghostboard” device by reading the scancodes and printing them as normal ASCII characters to standard output
 - You will need to use `mknod` to create a character device file with the major/minor IDs
 - You will need to do some sleuthing to figure out what the scancodes mean



Notifier Chains

- Linux provides a publish-subscribe model for event notifications across kernel subsystems called Notifier Chains
- This means that systems can register a callback with another system that will get invoked when the other system incurs some given event
- Implementations of the notifier chains are found in `/kernel/notifier.c`
- There are different types of notifier chains, each of which behave differently based on the context in which the callbacks should execute
 - Atomic – for execution within an atomic or interrupt context (no sleeping allowed)
 - Blocking – for execution within process context (sleeping allowed)
 - SRCU – for Sleepable RCU (read-copy-update) – similar to blocking but with different tradeoffs
 - Raw – for managing your own locking/blocking



Registering and Unregistering for Events

- The Notifier Chain API is defined in `include/linux/notifier.h`
 - `atomic_notifier_chain_register` / `atomic_notifier_chain_unregister`
 - `blocking_notifier_chain_register` /
 - `blocking_notifier_chain_unregister`
 - `raw_notifier_chain_register` /
 - `raw_notifier_chain_unregister`
 - `srcu_notifier_chain_register` /
 - `srcu_notifier_chain_unregister`
- All of these call `notifier_chain_register` or `notifier_chain_unregister` behind the scenes



Notification Callbacks

- When registering for events, a callback function is supplied within a `struct notifier_block`, defined by the caller

```
struct notifier_block {  
    notifier_fn_t notifier_call;  
    struct notifier_block __rcu *next;  
    int priority; /* goes unused by most of the time  
*/  
};
```

```
typedef int (*notifier_fn_t)(struct notifier_block *nb,  
    unsigned long action, void *data);
```

- When events are generated, the corresponding subsystem calls (a wrapper) for `notifier_call_chain`, which invokes all of the callbacks on a given list
- Calls to event registration are essentially requests to add the `notifier_call` function to a specific list of other `notifier_calls`



In the Source

- It is common for kernel subsystems to maintain their own list of notifiers and then supply a wrapper routine for registering callbacks on those lists
- For example, in `/net/core/netevent.c`:

```
static ATOMIC_NOTIFIER_HEAD(netevent_notif_chain);

int register_netevent_notifier(struct notifier_block *nb)
{
    return atomic_notifier_chain_register(&netevent_notif_chain, nb);
}
EXPORT_SYMBOL_GPL(register_netevent_notifier);
```



LINUX CNO Programming



Lab 12b (optional)

Keylogger – Notifier Chains

Tasks

- Remake your keylogger, this time using notifier chains
- Browse the kernel source to find a good spot to hook into keyboard notifications
- Don't use an interrupt handler
- Continue to use your ring buffer to queue up keyboard scancodes
 - This time around you may need to set up some filters for different types of notifications and figure out which ones are scancodes and which ones aren't
- Use your existing userspace program to read key data and print it out to standard output
- Bonus: Use a notifier chain to make your kernel module survive reboot



LINUX CNO Programming



Tracing

Tracing in the Kernel

- The Linux kernel supports a variety of tracing technologies
- In the context of CNO, tracing can be useful for
 - Figuring out how things work
 - Finding bugs
 - Using tracing APIs to hook/inject easily
 - Patterning your own hooking mechanisms after the ones used in tracing mechanisms
- Learning how each of these systems works and how to use them will help you perform both VR and CNO in the kernel



Kprobes

- Kprobes is an API that allows hooked tracing through the kernel
- Kprobes uses the hardware breakpoint exceptions to dynamically insert instructions into functions
- It inserts a breakpoint (`int3`) on a specified instruction address (or symbol)
- When the system hits the breakpoint, the `int3` exception handler in the kernel calls the handler functions defined by the user of Kprobes
- What does this remind you of?



Kprobe API

- The kprobes API is found in `/include/linux/kprobes.h`
- To create a kprobe, a struct `kprobe` must be created and registered with the kprobe system
 - Most of the members of this struct will be set by the `register_kprobe` function
 - The user specifies:
 - `symbol_name`, the symbol to hook
 - `pre_handler`, the function to run before the symbol is executed
 - `post_handler`, the function to run after the symbol is executed
 - `fault_handler`, the function to run if the symbol causes a fault
 - If this is invoked the `post_handler` will not be invoked
- Unregistering a probe is done with `unregister_kprobe`
- See also: a `kretprobe` will give you a callback for when a specified function returns and provide its return value



Tracepoints

- Tracepoints in the kernel are programmer-declared points that make callouts to registered probes (tracers)
- The `TRACE_EVENT` macro allows kernel programmers to insert tracepoints into their code
- These are more reliable than kprobes in the sense that their addresses don't change from one build to another
- However, if a tracepoint doesn't exist, you can't hook up a probe to it



Ftrace

- We've used Ftrace in the previous class
- Ftrace uses GCC's profiler (-pg) to insert a call to the mcount function (a trampoline) within each function call
- Simply keeping these calls results in too much overhead
- After compilation, the locations of these calls are recorded by a script (/scripts/recordmcount.c)
- Using these locations, the calls to mcount are overwritten with nops during kernel boot



Ftrace *(continued)*

- During boot, space is allocated to create tracepoints for all of the removed invocations of mcount
 - These correspond to those seen in trace-cmd list
- When you enable tracing of a particular function, the nop is changed to a call to `ftrace_caller`, which will save state, call a custom function, and then call a stub function to return to the original function
- What does this remind you of?



Gathering Data

- Another set of tools allow us to gather data from kprobes, tracepoints, and ftrace hooks
- The sys/kernel/debug/tracing pseudo file system allows us to control tracing
 - trace-cmd is a front-end for this
 - Let's you use data from events (tracepoints) and symbols (ftrace) and even kprobes
 - Difficult to parse, quite a bit of data
- perf_events (or sometimes just perf) is another method
 - Uses the perf_event_open system call, which returns tracepoint information that is stored in a ring buffer
 - See: `sudo perf trace` for a userspace tool that uses this



Extended BPF

- The most modern and advanced system for obtaining and working with tracing information is eBPF
- Originally the Berkeley Packet Filter (BPF) was a network packet filtering component in the kernel
- BPF allows programs to be inserted into and run by the kernel in a light-weight VM
- If you pass tcpdump the -d parameter it will display the bytecode used by the kernel to execute the running filter
- Programs are statically verified by the kernel before they are loaded and run to ensure that they will not cause system harm
 - What do you think?



Classic BPF Demo

- Run a local ncat server and connect to it with another ncat instance and send a bunch of packets running each of the following commands
 - `sudo tcpdump -i lo "ip6 and tcp"`
 - `sudo tcpdump -d -i lo "ip6 and tcp"`
- The code shown in the second example is classic BPF
- Can you guess what the code is doing? What do the constants represent?
- BPF is run by the kernel, and it allows the kernel to be programmed by user space tools (in a restricted fashion) without the use of a kernel module
- Classic BPF only had 2 registers, a handful of instructions, and no call to helpers
- Classic BPF isn't used anymore – instead its translated to eBPF before being run



eBPF



- The extended Berkeley Packet Filter (eBPF) extends BPF functionality to more than just network filtering to numerous other systems
 - This list is growing with each new kernel version: seccomp, socket filters, netfilter xt_bpf, cls_bpf, etc.
- This allows highly-specialized tracing and filtering to be inserted into the kernel at runtime without using a full loadable kernel module, in a restricted VM environment
- eBPF is supported for kernels running on x86_64, ARM64, and a variety of other architectures
- "One of the more interesting features in this cycle is the ability to attach eBPF programs (user-defined, sandboxed bytecode executed by the kernel) to kprobes. This allows user-defined instrumentation on a live kernel image that can never crash, hang or interfere with the kernel negatively."



eBPF bytecode

- Since eBPF requires byte code in a different language for a simple virtual machine, BPF programs look different from typical code
 - See `man 2 bpf`
- eBPF bytecode is intended to have a near one-to-one mapping to machine instructions for performance
 - The mapping for registers is one to one
- The kernel verifies this code and then translates the code into x86 (or ARM64)
- eBPF introduced the ability for BPF programs to call predefined helper functions in the kernel
 - These functions must be in the main kernel – helper functions cannot be introduced by loadable modules
 - The list of exported functions is expanding rapidly



eBPF bytecode *(continued)*

- The instruction set and verifier impose some limitations for security purposes
 - No exposed stack pointer
 - No loops allowed
- In eBPF there are 11 64-bit registers and a handful of instructions
 - R0 is return value
 - R1 – R5 are arguments
 - R6 - R9 general purpose callee saved
 - R10 is the frame pointer, and is read-only
- R1 contains the “context” for the program, which is similar to an argv, and varies depending on the type of program (socket BPF code has an skb as its context)
- These registers are translated to actual registers on the hardware by the JIT
- This calling convention allows a zero overhead when calling from and to other native kernel functions
 - No extra mov instructions required by the JIT to set up for a function call



Register Mapping



| BPF REGISTER | HW REGISTER |
|--------------|-------------|
| r0 | rax |
| r1 | rdi |
| r2 | rsi |
| r3 | rdx |
| r4 | rcx |
| r5 | r8 |
| r6 | rbx |
| r7 | r13 |
| r8 | r14 |
| r9 | r15 |
| r10 | rbp |



eBPF Instruction Set

- eBPF instructions are fixed-width: 64-bits
 - Opcode: 8 bits
 - Destination: 4 bits
 - Source: 4 bits
 - Offset: 16 bits
 - Immediate: 32 bits
 - Unused fields are zeroed
- There is only 16-byte instruction that performs a 64-bit immediate load
- Maximum program size is 4096 instructions
- No loops allowed, but you can tail-call to another function
 - Maximum tail call depth is 32



eBPF Instruction Set *(continued)*

- The instruction set itself can be used with the mnemonics from `tools/include/uapi/linux/bpf_common.h`
- Opcodes have multiple parts: class, fields, widths
- To create a full opcode you'll need to OR them together
 - Example: `BPF_JMP` | `BPF_JNE` | `BPF_X`
- See <https://github.com/iovisor/bpf-docs/blob/master/eBPF.md> for a breakdown of instruction opcodes with pseudocode



eBPF Verification

- The verifier first verifies that the program is a directed acyclical graph (DAG) to detect the presence of loops
- It then checks a variety of other factors
 - Can't use a register that you haven't written to (aside from R1, for context)
 - Can't read from a place in the stack value you didn't write to
- The verifier assigns a type to register values as it simulates all branches of execution for the program
 - NOT_INIT
 - SCALAR_VALUE
 - PTR_X, where X is a variety of pointer types (read: dangerous types)
 - e.g., PTR_TO_STACK, PTR_TO_PACKET, PTR_TO_MAP_VALUE, etc.
- Each of these types has rules about how it is allowed to be used



eBPF Maps

- eBPF allows data storage and data sharing between userspace and kernelspace using eBPF maps
- Maps are key-value stores, and are also created and managed by the bpf system call using the following as the cmd parameter:
 - BPF_MAP_CREATE
 - BPF_MAP_LOOKUP_ELEM
 - BPF_MAP_UPDATE_ELEM
 - BPF_MAP_DELETE_ELEM
 - And more
- Destroy a map by closing its file descriptor
- When creating a map, the programmer specifies the type, size of keys, size of values, and the maximum number of elements in the map
 - Type can be: hash, array, bloom filter, radix-tree, etc.



Using eBPF

- eBPF programs can call kernel helper functions that are exposed to them (like gaining access to socket information)
- eBPF programs can attach to sockets, kprobes, tracepoints, etc.
- See Documentation/networking/filter.txt
- You can attach a program to a socket using
 - `socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL))` and `setsockopt(s, SOL_SOCKET, SO_ATTACH_FILTER, ...)`



eBPF Helpers

- Helpers are (currently) limited to functions with 5 or less arguments
- They are defined in the kernel using a special macro similar to the one used for system calls
 - `BPF_CALL_[0-5]`
- When a `bpf_call` instruction is made to a helper function in the kernel, the JIT seamlessly integrates the call as a native one
- See `man 7 bpf-helpers` for a list of helper functions
- **You can also attach BPF programs to kprobes, perf, and tracepoints!**
 - Use `bpf` syscall in conjunction with `perf_event_open` syscall to do this
 - BPF program type should be set to `BPF_PROG_TYPE_KPROBE`, `BPF_PROG_TYPE_PERF_EVENT`, or `BPF_PROG_TYPE_TRACEPOINT`



Actually Using eBPF

- Writing the byte code by hand is impractical for all but simple examples
- Often BPF programs will call other BPF programs, each will do some special task
- Sometimes you'll want to make other code in C to attach BPF programs to kprobes or some other system
- Use the BCC (BPF compiler collection) instead, as it allows you to do all these things with a single suite
 - With BCC you can write C code that is compiled to BPF
 - Used by Netflix, Facebook, and has a large community
 - Has python bindings to make your life even easier
- bpfttrace is also available - a simple front-end that does not allow full BPF code but has some good flexibility



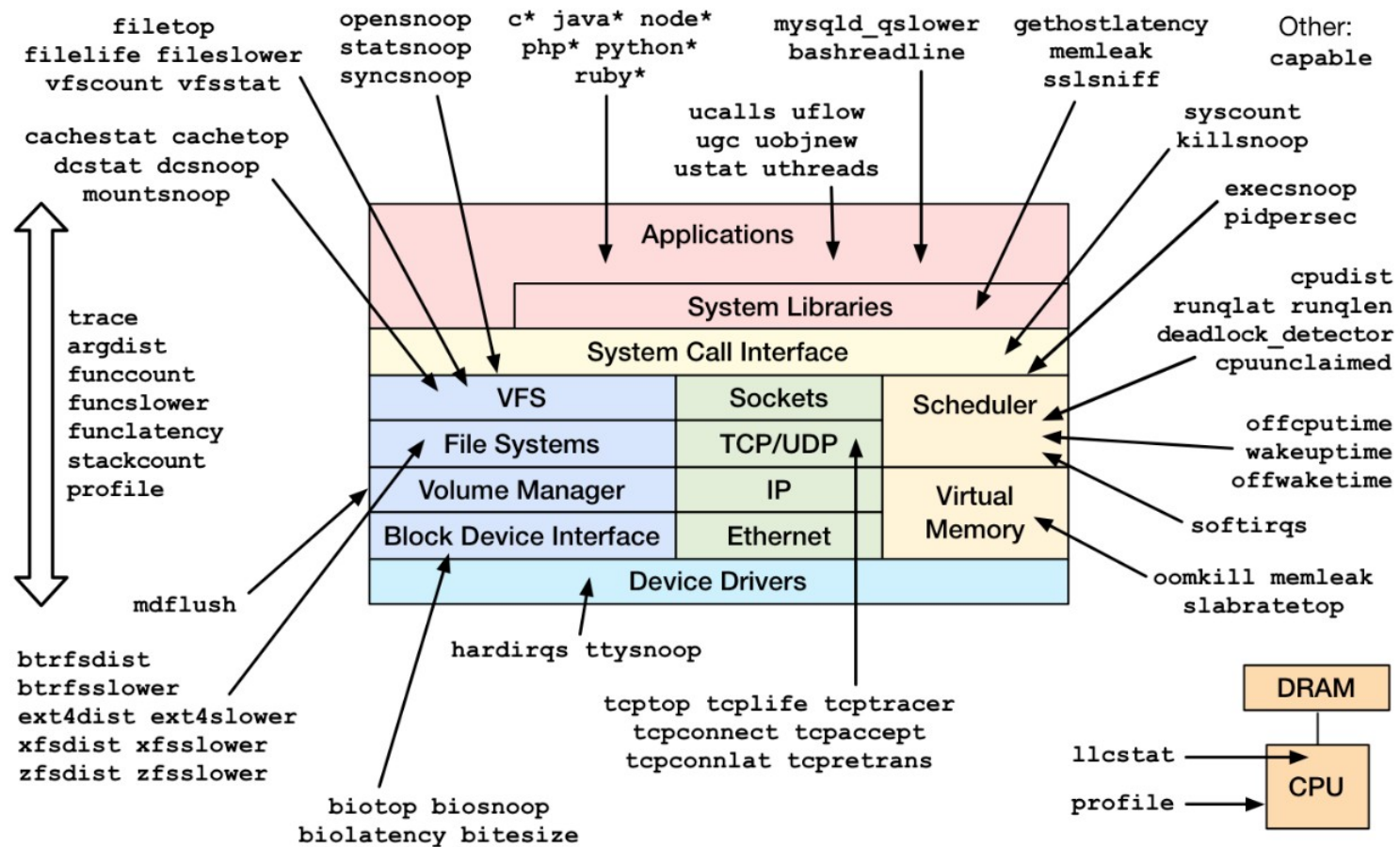
BCC Demo

- Let's make a small C program that prints out the filename of anything passed to the `execve` system call
- Let's compile it with BCC and load it into the kernel with a BCC python script



Actually Using eBPF

Linux bcc/BPF Tracing Tools



<https://github.com/iovisor/bcc#tools> 2018



See Also

- SystemTap
 - Another tracing framework
 - Loadable kernel module
- LTTng
 - Another tracing framework
 - Loadable kernel module
- Jprobes (deprecated)
 - Allowed a kprobe to access the runtime parameters of a given function
- If you're interested in tracing, Brendan Gregg of Netflix has numerous presentations and blog posts about tracing in the kernel
 - Especially eBPF



LINUX CNO Programming



Lab 13

Tracing

Tasks

- Build a kernel module that uses either
 - A tracing API of your choice
 - A method from one of the discussed tracing APIs
- Your module should insert a probe point to obtain interesting information from the kernel in some way
- The simplest option is to use the kprobe API to trace a function of interest
 - Feel free to do this, or to challenge yourself and do something else
 - eBPF by hand anyone?



LINUX CNO Programming



Rooting

got root?

What does it mean to have root?

- There are numerous facets of privileges on modern systems
 - UID, GID, and other DAC-oriented credentials
 - POSIX 1.e capabilities (e.g. CAP_NET_ADMIN, CAP_SYS_NICE, etc.)
 - MAC-oriented contexts (e.g. AppArmor policy, SELinux policy)
 - Sandboxing (seccomp, seccomp BPF, containerization, namespaces)
- Increasingly common are other facets of privilege/isolation:
 - Virtualization
 - Type 1 Hypervisors
 - Hardware secure enclaves (e.g. TrustZone, SGX)



What does it mean to have root? (continued)



- In the simplest sense, having root means that the UID of a process we control is zero
- In modern practice, we often need to have additional state to allow CNO tools to perform their various tasks.
- In the next lab, we will learn how to use exploit primitives to perform the following:
 - Become root for all ID types
 - Grant all capabilities
 - Break out of seccomp
 - Disable SELinux



First Things First

1. Wrap your exploit primitives
 - read, write, others
2. Find a task_struct you can control
 - What does this mean?
3. Identify the cred structure
 - Why might this be hard?
4. Change IDs
 - Where are these? Which one is which?
5. Change capsets
 - What capabilities do you need?
6. Break out of seccomp. How?
7. Disable SELinux. How?



Note: Depending on your exploit, your strategy may be very different,



LINUX CNO Programming



CNO Memory Operations

CNO Memory Management



- In the kernel procedures are usually event-driven, and there is not a clear way to swap contexts to operate in a different virtual memory space
 - There are, however, ways to perform operations on other tasks and their memory
 - If all else fails, the kernel provides a lot of helpful methods for working with page tables
- In the `task_struct`, there is a struct `mm_struct` entry called `mm`
 - This may be NULL if the process is an anonymous process, but in that case `active_mm` will be the memory context it is running in
- `mm_struct` contains an `mmap` member
 - This struct `vm_area_struct` is in a ordered doubly linked list, as well as an `rb_tree`
 - Each node in the data structure has a lot of information about the underlying memory
 - Much can be done to monitor or alter memory usage
 - See the `vm_ops` member



CNO Memory Management *(continued)*



- Look and understand how other components that do similar task work
 - /kernel/ptrace.c
 - /mm/memory.c
 - etc.
- Make sure you understand the complications of a real-world machine
 - Does your code work if the machine is under memory pressure?
 - Does your code work if interrupted?
 - CNO operations can be doubly difficult to make secure



Attacking Slub

- Exploits involving the kernel's dynamic memory require a deeper understanding of the SLUB allocator
- Typically, though, a good point of attack for an overflow is the freelist for the slab's objects
 - Control over the freelist can cause an "allocation" to happen in attacker controlled memory, which can give control over the contents to the attacker
 - If SMAP is not enabled, this can even be in userspace
 - Because similar caches can be combined into one, even systems using their own cache can be co-located with other objects, which can ease exploitation
- Dynamic overflow across boundaries of caches can also be massaged to gain some reliability, but overflowing / UAFing in the same cache is preferable



Bad mmap

- Bad implementations of operations can lead to some common mistakes
 - For file_operations, implementations of mmap can lead to bugs
- Some common bugs related to implementing the mmap file op include:
 - Not checking mapping size
 - Including integer overflows
 - Including signed integer types
 - Not checking mapping offset
 - Not checking protections
 - Timing attacks on shared memory
- If we can mess with the cred structure for our process, that is a privesc
 - Adding capabilities /changing user to root
 - Some hardening / security modules watch for this kind of escalation



LINUX CNO Programming



Lab 14

Steal Memory

Lab Tasks



- Tasks:
 - Create a loadable kernel module that can locate a loaded ELF file in a given process's memory
 - The PID and the ELF image to locate will be given
 - Read from, and write to, that memory
 - You are given an example program "target" that has a simple global buffer you can read from and write to
- Bonus:
 - Inject code into a remote process, to hook any calls to read
 - Find and replace words read in



Lab Tasks *(continued)*



- You will have to find the appropriate task for the target
- You will have to search the target memory structures to find the section associated with the image needed
- You will have to find a way to read to and write from the remote process's memory



LINUX CNO Programming



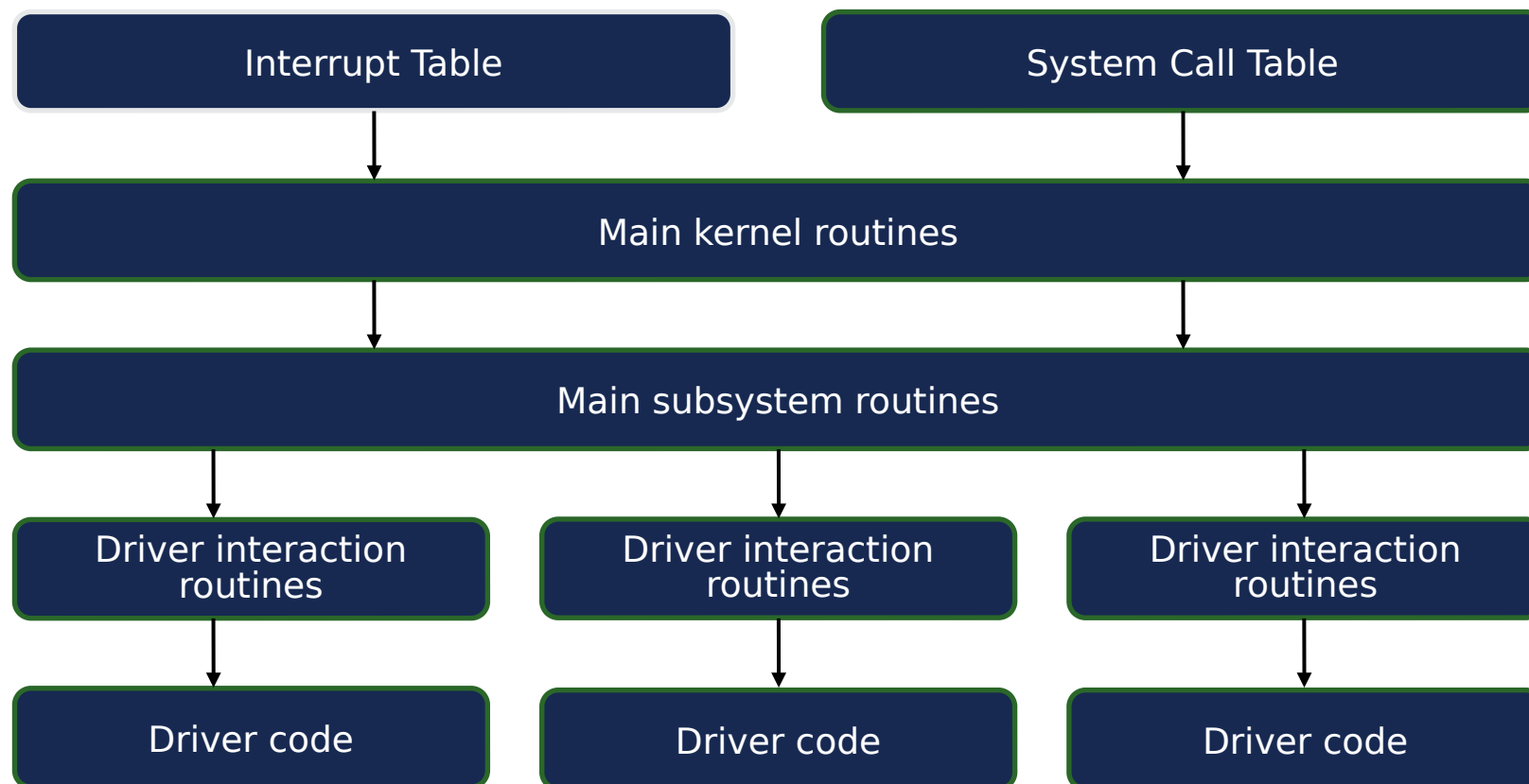
Hooking in the Kernel

Hooking in the Kernel

- Hooking can be performed in the kernel just like in userspace
 - Code patching
 - Pointer replacement
 - Etc.
- Note that some hooking points are monitored
 - E.g., Grsecurity will notice if you hook the system call table
- When in the kernel, you have access to the entire memory of the system – kernel and otherwise
 - As such, no function is impossible to hook (although some special cases may take extra effort)



Where should you hook?



System Call Table Hooking

- Hooking the system call table is an easy (and noisy) task in Linux
- The system call table itself is a global named `sys_call_table`
- Even though the symbol is not exported to modules, you can still get its address using a lookup with the `kallsyms_lookup_name` function
- Since the `sys_call_table` is write-protected, we also need to disable this (temporarily) before we swap a pointer contained within it with our own
 - Use `write_cr0` and `read_cr0` from `/arch/x86/include/asm/special_insns.h` to clear and set the write-protect bit (`X86_CR0_WP`) as needed
 - In an SMP system, disable pre-emption during this time as well using `preempt_disable` and `preempt_enable`
- Then simply replace the system call entry of your choice with your own hook function (remember to save the pointer to the old one)



Interrupt Gate Hooking

- Another place to hook the kernel is to hijack an entry in the interrupt descriptor table (IDT)
- Recall from the Kernel Internals series that the kernel uses the `load_idt` function (which invokes the `lidt` instruction)
- The interrupt descriptor table itself must be contained within a single page
- Each entry in the table is called a gate
- Each gate can one of three different types:
 - Interrupt gate: Cannot be accessed by a usermode process (DPL field is equal to 0). Used for interrupts.
 - System gate: Can be accessed by a usermode process (DPL field is equal to 3). The four Linux exception handlers associated with the vectors 3, 4, 5, and 128 are activated by means of system gates
 - Trap gate: Cannot be accessed by a usermode process (DPL field is equal to 0). Used by exception handlers



x86 Interrupt Gate Layout

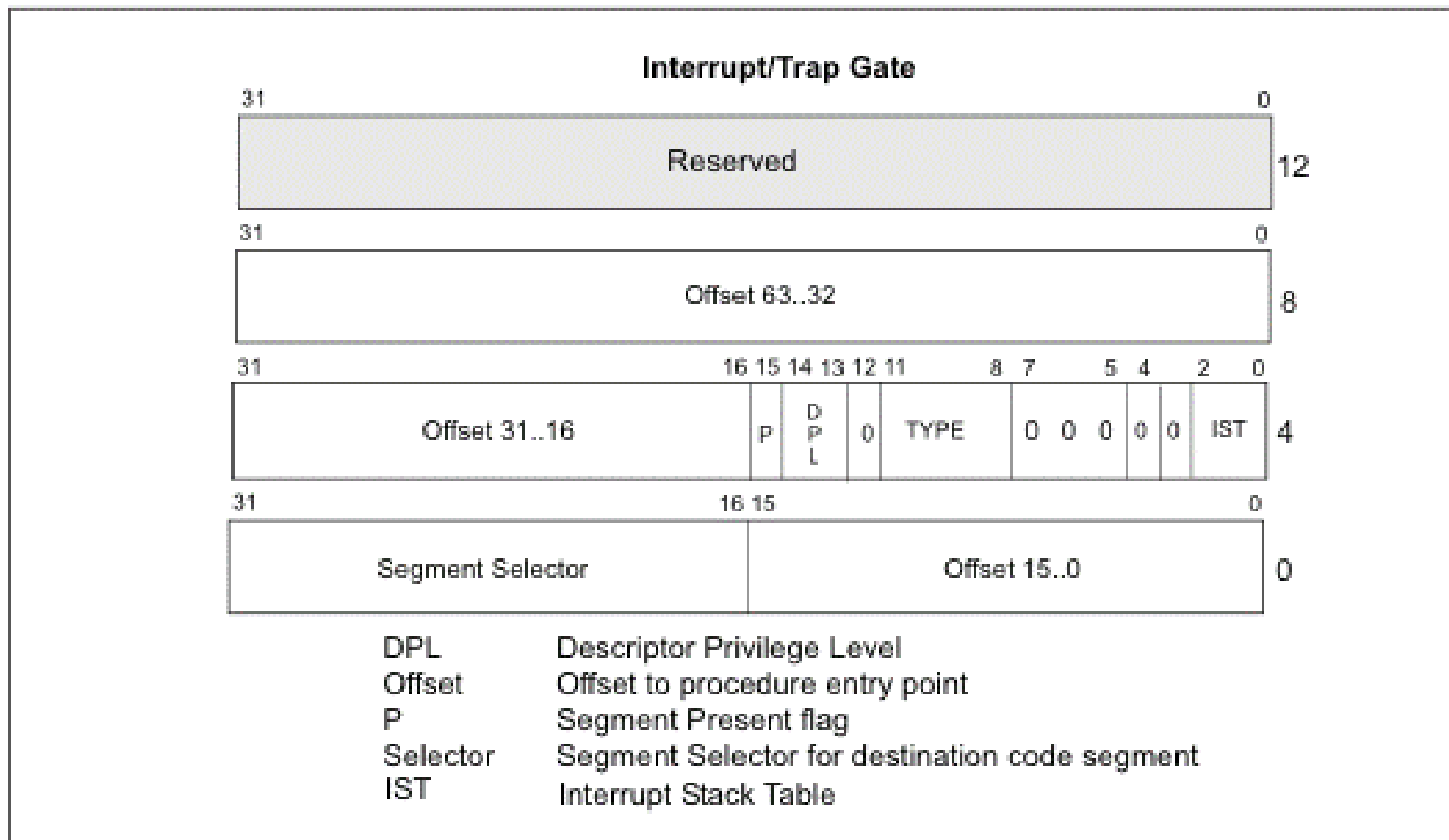


Figure 5-7. 64-Bit IDT Gate Descriptors



LINUX CNO Programming



Lab 15a

Hooking an Interrupt Gate

Tasks

- Figure out how Linux sets and/or modifies interrupt gates
- Figure out which gate is responsible for handling a divide by zero exception
- Use this knowledge to create a kernel module that hooks the interrupt handler for the divide by zero exception
- The new handler could count the number of times it has been invoked and then execute the original handler



Hooking Subsystem Routines

- Hooking inside kernel subsystems can often be performed using simple pointer replacement
- As always, more involved methods such as code patching can be used as well
- Subsystems like the VFS and networking make prolific use of function pointers in a way similar to object-oriented polymorphism
 - This allows us to replace important functions with our own without making much disturbance
 - Often you can selectively change these pointers so that your code only modifies specific object instances of interest



Hooking Driver Routines

- Hooking driver routines gathers information less generally than other approaches because the hooks only see data from the given driver
- Most of the kernel code is driver code, however, so it can be a bit easier to hide in the noise
- Since the kernel has access to all drivers and loaded modules, one option is to dynamically edit the code for a few specific drivers of interest
 - What specific drivers might be of use to hook?



LINUX CNO Programming



Lab 15b

Hijacking Network Traffic

Tasks

- Given what you learned from the Linux Kernel Internals series regarding the network subsystem, find a way to intercept and modify both incoming and outgoing TCP traffic on the local machine
- You should create a kernel module for this lab that is not “noisy” – do not use any APIs in the kernel that are too obvious, high-level, or easily monitored
 - In particular, you are **not** allowed to
 - use the Netfilter API
 - use iptables
 - hook the system call table
 - use Kprobes or Jprobes



Tasks *(continued)*

- When finished, your interceptor should print out all application-layer data that is sent and received by a given process (specified by PID)
- As a bonus, your interceptor should also be able to modify traffic
- As a bonus bonus, your interceptor should be able to lie to the application about how much data was sent and received as well as modify traffic
 - Things to consider:
 - What if your interceptor wants to modify 50 bytes but only 10 exist in the current buffer?
 - What if the userspace program is trying to receive 50 bytes but you only have 10 after your modifications?
 - What if the userspace program is trying to receive 50 bytes but you have 0 after your modifications?



Tasks *(continued)*

- Consider hiding your module from traditional lists using:
 - `list_del_init(&__this_module.list);`
 - `kobject_del(&THIS_MODULE->mkobj.kobj);`
- What does this code do?



LINUX CNO Programming



CNO Considerations

Randomized Struct Fields

- The Linux kernel makes judicious use of the `__randomize_layout` option supported by GCC
- This option randomly reorganizes the fields within a struct to change their offsets
 - This makes it harder for attackers to target multiple kernel builds for common Linux distributions
 - It makes it even harder for attackers to target builds of private in-house builds where the random seed won't even be accessible to them
- Many kernel structures use this macro to wrap the fields which should be randomized

```
#define randomized_struct_fields_start struct {  
#define randomized_struct_fields_end    }  
__randomize_layout;  
#endif
```



Randomized Struct Fields *(continued)*

- Sometimes the kernel relies on certain fields being at the beginning or end of a structure
- Still, they will often randomize the layout of the other fields
- Example: task_struct

```
struct task_struct {  
#ifdef CONFIG_THREAD_INFO_IN_TASK  
    struct thread_info    thread_info;  
#endif  
    /* -1 unrunnable, 0 runnable, >0 stopped: */  
    volatile long          state;  
    /*  
     * This begins the randomizable portion of task_struct. Only  
     * scheduling-critical items should be added above here.  
     */  
    randomized_struct_fields_start  
  
    void                    *stack;  
    refcount_t              usage;  
    ...  
}
```



Protection Mechanisms

- The kernel employs a variety of protections that make certain types of modifications more difficult
- KASLR – Kernel Address Space Layout Randomization
 - The kernel itself will be randomly relocated upon boot, making it more difficult for attackers to find kernel addresses
- SMEP – Supervisor Mode execution prevention
 - This is a CPU feature that allows the kernel to turn off its own ability to execute instructions in usermode memory
- SMAP – Supervisor Mode Access Prevention
 - This is a superset of SMEP which prevents any access of usermode memory by the kernel
 - This is turned off when the kernel needs to access memory (such as when reading in userspace buffers provided in system calls like `sendmsg`)
 - See the use and implementation of the `stac` and `clac` functions in the kernel



Secure Boot

- Secure Boot is a feature of UEFI systems that allows a system to perform some code integrity checks on an operating system before booting
- Keys (or certificates containing the public keys) of various software vendors can be loaded into firmware to be used in validation of binaries
 - Sometimes these come built-in, as is the case with many x86 vendors and Microsoft keys and certificates
- Each binary loaded during boot can be checked for legitimacy and consistency using PKI constructs
- How would you overcome or deal with this?



Class Conclusion

- Thanks for participating in this course!
- Thank you for your feedback – it will be used to improve this class in its future offerings

