Linux Device Drivers

On Desktops

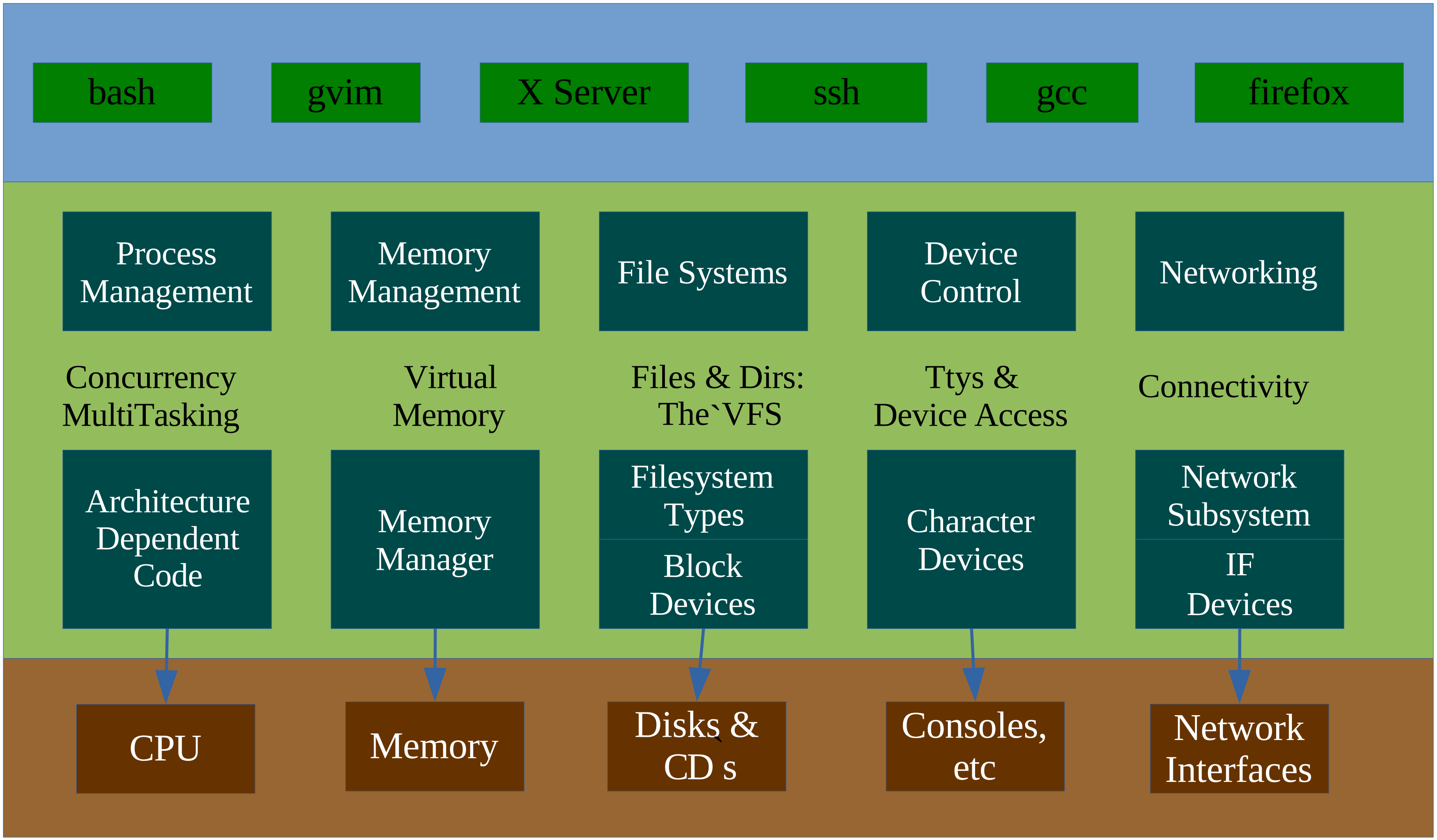
Team Emertxe

Introduction

Familiarity Check

- Good C & Programming Skills
- Linux & the Filesytem
  - Root, User Space Headers & Libraries
- Files
  - Regular, Special, Device
- Toolchain
  - gcc & friends
- Make & Makefiles
- Kernel Sources (Location & Building)

Linux Driver Ecosystem



The Flow

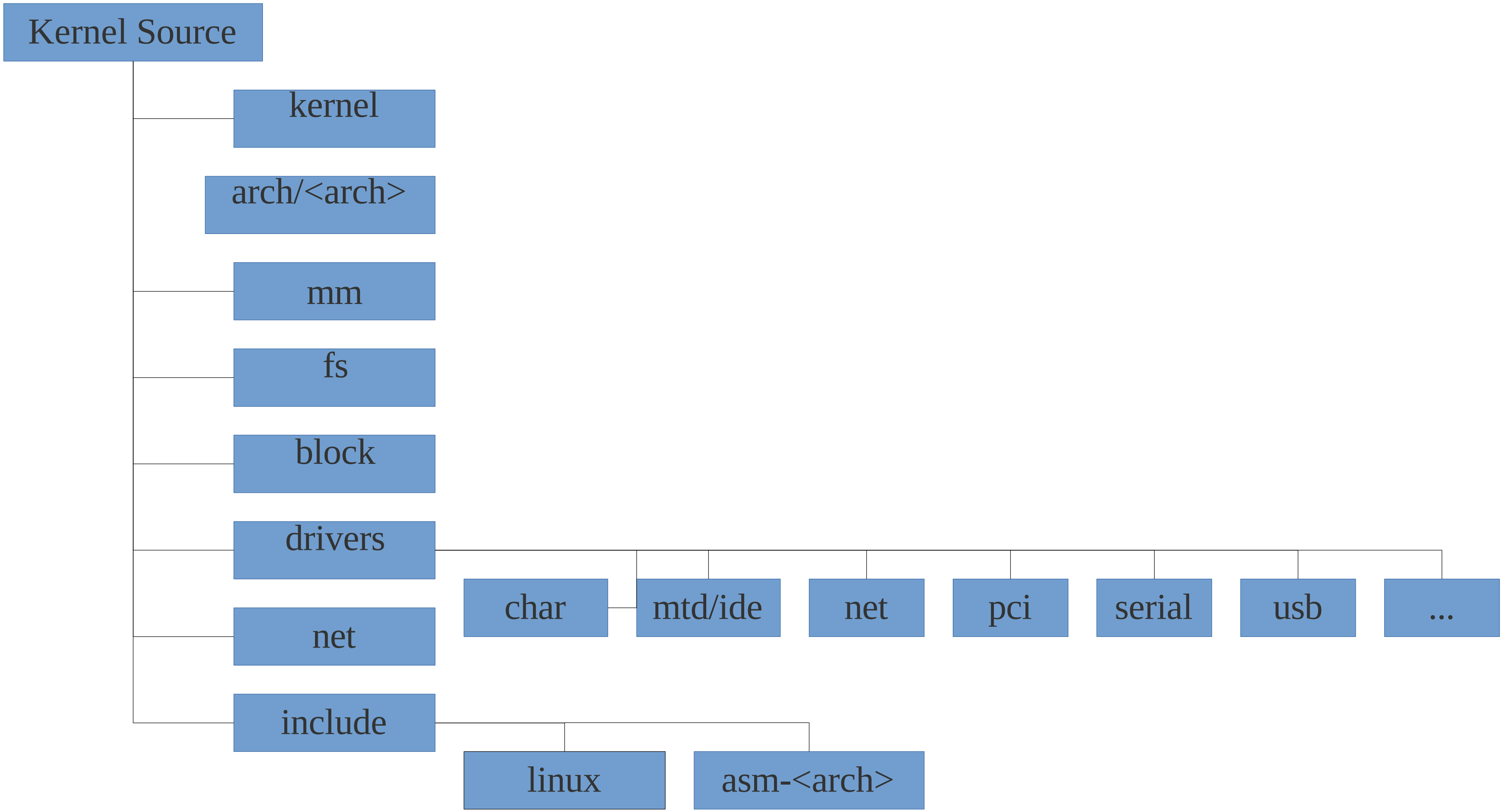
- Introduction
- Character Drivers
- Memory & Hardware
- Time & Timings
- USB Drivers
- Interrupt Handling
- Block Drivers
- PCI Drivers

ΣMERTXE

Hands-On

- Your First Driver
- Character Drivers
  - Null Driver
  - Memory Driver
  - UART Driver for Customized Hardware
- USB Drivers
  - USB Device Hot-plug-ability
  - USB to Serial Hardware Driver
- Filesystem Modules
  - VFS Interfacing
  - “Pseudo” File System with Memory Files

Kernel Source Organization



The Locations & Config Files

- Kernel Source Path: /usr/src/linux
- Std Modules Path:
  - /lib/modules/<kernel version>/kernel/...
- Module Configuration: /etc/modprobe.conf
- Kernel Windows:
  - /proc
  - /sys
- System Logs: /var/log/messages

The Commands

- lsmod
- insmod
- modprobe
- rmmod
- dmesg
- objdump
- nm
- cat /proc/<file>

The Kernel's C

- ctor & dtor
  - init_module, cleanup_module
- printf
  - printk
- Libraries
  - <kernel src>/kernel
- Headers
  - <kernel src>/include

The Init Code

```
static int __init mfd_init(void)
{
  printk(KERN_INFO "mfd
   registered");
  ...
  return 0;
}
module_init(mfd_init);
```

The Cleanup Code

```
static void __exit mfd_exit(void)
{
  printk(KERN_INFO "mfd
    deregistered");

  ...

}
module_exit(mfd_exit);
```

Usage of printk

- <linux/kernel.h>
- Constant String for Log Level
  - KERN_EMERG "<0>" /* system is unusable */
  - KERN_ALERT "<1>" /* action must be taken immediately */
  - KERN_CRIT "<2>" /* critical conditions */
  - KERN_ERR "<3>" /* error conditions */
  - KERN_WARNING "<4>" /* warning conditions */
  - KERN_NOTICE "<5>" /* normal but significant condition */
  - KERN_INFO "<6>" /* informational */
  - KERN_DEBUG "<7>" /* debug-level messages */
- printf like arguments

The Other Basics & Ornaments

- Headers
  - #include <linux/module.h>
  - #include <linux/version.h>

  –#include <linux/kernel.h>
- MODULE_LICENSE("GPL");
- MODULE_AUTHOR("Emertxe");
- MODULE_DESCRIPTION("First Device Driver");

Building the Module

- Our driver needs
  - The Kernel Headers for Prototypes
  - The Kernel Functions for Functionality
  - The Kernel Build System & the Makefile for Building
- Two options
  - Building under Kernel Source Tree
    - Put our driver under drivers folder
    - Edit Kconfig(s) & Makefile to include our driver
  - Create our own Makefile to do the right invocation

Our Makefile

```
ifneq (${KERNELRELEASE},)
   objm += <module>.o
else
   KERNEL_SOURCE := <kernel source directory path>
   PWD := $(shell pwd)
default:
   $(MAKE) C ${KERNEL_SOURCE} M=$(PWD)
     modules
clean:
   $(MAKE) C ${KERNEL_SOURCE} M=$(PWD)
     clean
endif
```

Try Out your First Driver

Character Drivers

Major & Minor Number

- ls -l /dev
- Major is to Driver; Minor is to Device
- <linux/types.h> (>= 2.6.0)
  - dev_t: 12 & 20 bits for major & minor
- <linux/kdev_t.h>
  - MAJOR(dev_t dev)
  - MINOR(dev_t dev)
  - MKDEV(int major, int minor)

Registering & Unregistering

- Registering the Device Driver
  - int register_chrdev_region(dev_t first, unsigned int count, char *name);
  - int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int cnt, char *name);
- Unregistering the Device Driver
  - void unregister_chrdev_region(dev_t first, unsigned int count);
- Header: <linux/fs.h>

The file operations

- #include <linux/fs.h>
- struct file_operations
  - int (*open)(struct inode *, struct file *);
  - int (*release)(struct inode *, struct file *);
  - ssize_t (*read)(struct file *, char __user *, size_t, loff_t *);
  - ssize_t (*write)(struct file *, const char __user *, size_t, loff_t *);
  - struct module owner = THIS_MODULE; / linux/module.h> */
  - loff_t (*llseek)(struct file *, loff_t, int);
  - int (*ioctl)(struct inode *, struct file *, unsigned int, unsigned long);

EMERTXE

User level I/O

- int open(const char *path, int oflag, ...)
- int close(int fd);
- ssize_t write(int fd, const void *buf, size_t nbyte)
- ssize_t read(int fd, void *buf, size_t nbyte)
- int ioctl(int d, int request, ...)
  - The ioctl() function manipulates the underlying device parameters of special files.
  - The argument d must be an open file descriptor.
  - The second argument is a device-dependent request code.

The file & inode structures

- struct file
  - mode_t f_mode
  - loff_t f_pos
  - unsigned int f_flags
  - struct file_operations *f_op
  - void * private_data
- struct inode
  - unsigned int iminor(struct inode *);
  - unsigned int imajor(struct inode *);

Registering the file operations

- #include <linux/cdev.h>
- 1st way initialization:
  - struct cdev *my_cdev = cdev_alloc();
  - my_cdev->owner = THIS_MODULE;
  - my_cdev->ops = &my_fops;
- 2nd way initialization:
  - struct cdev my_cdev;
  - cdev_init(&my_cdev, &my_fops);
  - my_cdev.owner = THIS_MODULE;
  - my_cdev.ops = &my_fops;

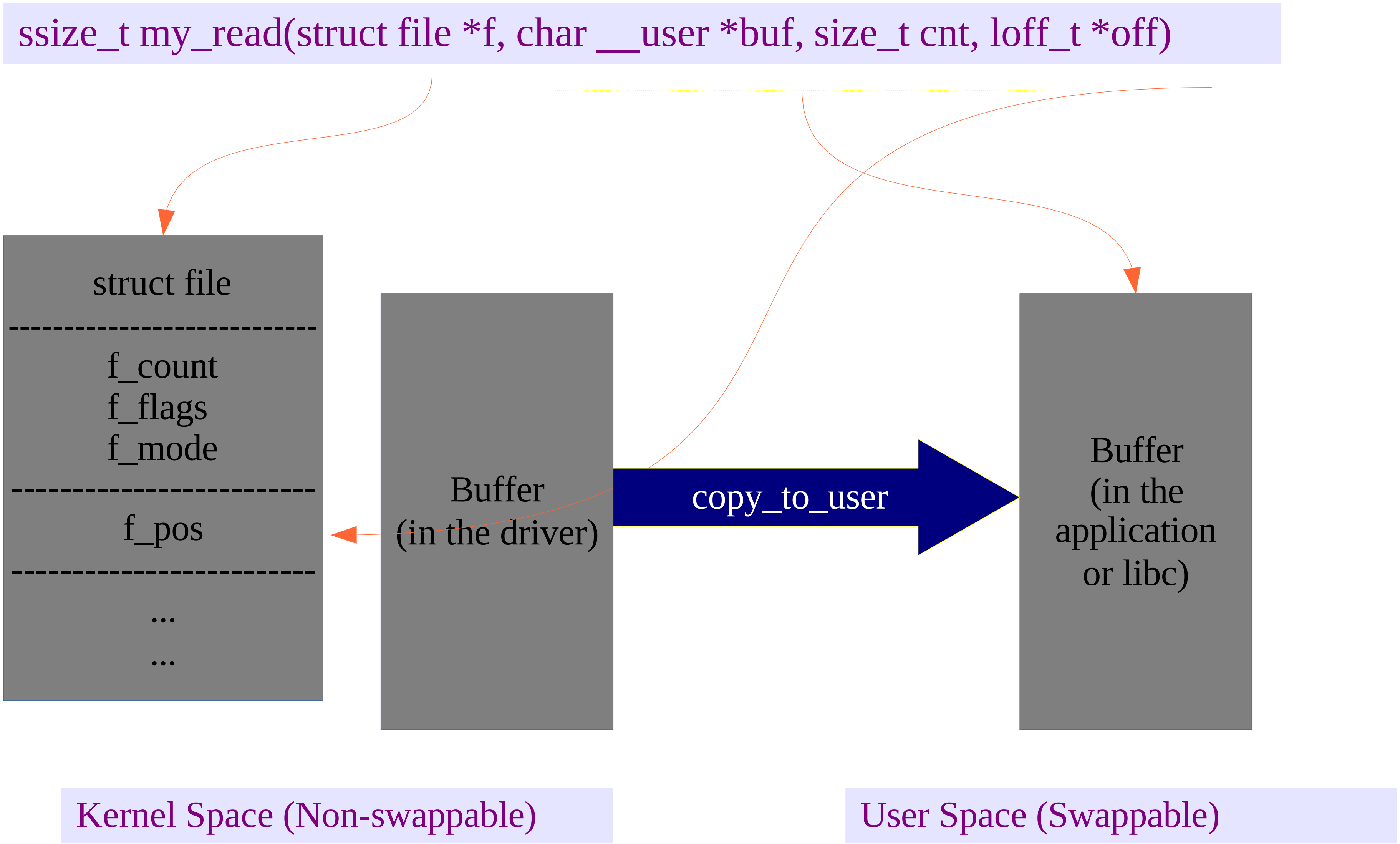
Registering the file operations...

- The Registration
  - int cdev_add(struct cdev *cdev, dev_t num, unsigned int count);
- The Unregistration
  - void cdev_del(struct cdev *cdev);

Registering/Unregistering Old Way

- Registering the Device Driver
  - int register_chrdev(undigned int major, const char *name, struct file_operations *fops);
- Unregistering the Device Driver
  - int unregister_chrdev(undigned int major, const char *name);

The read flow



The /dev/null read & write

```
ssize_t my_read(struct file *f, char __user
  *buf, size_t cnt, loff_t *off)
{
   ...
   return read_cnt;
}
ssize_t my_write(struct file *f, char __user
  *buf, size_t cnt, loff_t *off)
{
   ...
   return wrote_cnt;
}
```

The mem device read

```
ssize_t my_read(struct file *f, char __user
  *buf, size_t cnt, loff_t *off)
{
   ...
   if (copy_to_user(buf, from, cnt) != 0)
   {
      return EFAULT;
   }
   ...
   return read_cnt;
}
```

The mem device write

```
ssize_t my_write(struct file *f, char __user
  *buf, size_t cnt, loff_t *off)
{
   ...
   if (copy_from_user(to, buf, cnt) != 0)
   {
      return EFAULT;
   }
   ...
   return wrote_cnt;
}
```

Dynamic Device Node & Classes

- Class Operations
  - struct class *class_create(struct module *owner, char *name);
  - void class_destroy(struct class *cl);
- Device into & Out of Class
  - struct class_device *device_create(struct class *cl, NULL, dev_t devnum, NULL, const char *fmt, ...);
  - void device_destroy(struct class *cl, dev_t devnum);

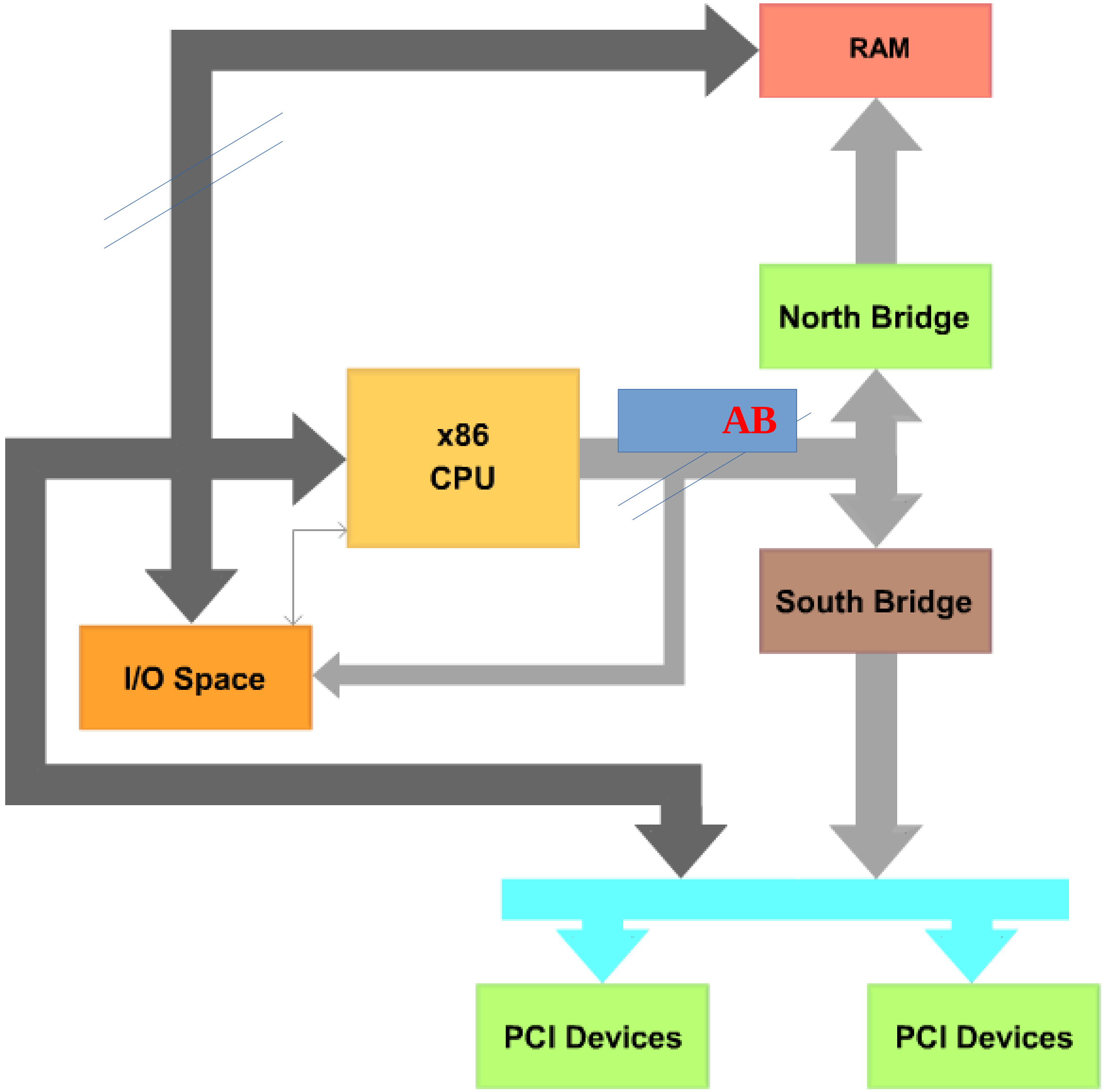
The I/O Control API

- int (*ioctl)(struct inode *, struct file *, unsigned int cmd, unsigned long arg)
- Command
  - <linux/ioctl.h> -> ... -> <asm-generic/ioctl.h>
  - Macros
    - _IO, _IOR, _IOW, _IOWR
  - Parameters
    - type (character) [15:8]
    - number (index) [7:0]
    - size (param type) [29:16]

Module Parameters

- <linux/moduleparam.h>
  - Macros
    - module_param(name, type, perm)
    - module_param_array(name, type, num, perm)
    - Perm (is a bitmask)
      - 0
      - S_IRUGO
      - S_IWUSR | S_IRUGO
  - Loading
    - insmod driver.ko name=10

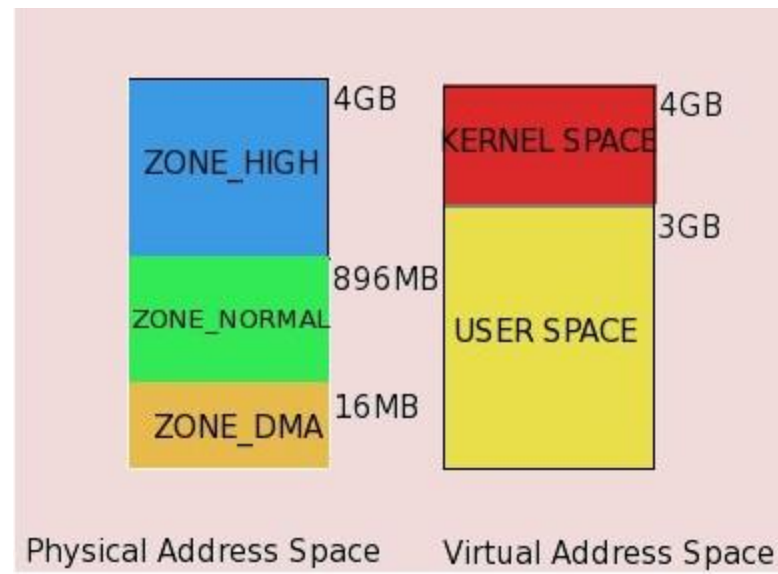
x86 Architecture



Memory Access

Physical Vs Virtual Memory

- The kernel Organizes Physical memory in to pages
  - Page size Depends on Arch
    - X86-based 4096 bytes
- On 32-bit X86 system Kernel total Virtual address space
  - Total 4GB (pointer size)
  - Kernel Configuration Splits 4GB in to
    - 3BG Virtual Sp for US
    - 1GB Virtual Sp for Kernel
      - 128MB KDS
  - Virtual Address also called Address”



ΣMERTXE

Memory Access from Kernel Space

- Virtual Address on Physical Address
  - #include <linux/gfp.h>
    - unsigned long __get_free_pages(flags, order); etc
    - void free_pages(addr, order); etc
  - #include <linux/slab.h>
    - void *kmalloc(size_t size, gfp_t flags);
      - GFP_ATOMIC, GFP_KERNEL, GFP_DMA
    - void kfree(void *obj);
  - #include <linux/vmalloc.h>
    - void *vmalloc(unsigned long size);
    - void vfree(void *addr);

Memory Access from Kernel Space...

- Virtual Address for Bus/IO Address
  - #include <asm/io.h>
    - void *ioremap(unsigned long offset, unsigned long size);
    - void iounmap(void *addr);
- I/O Memory Access
  - #include <asm/io.h>
    - unsigned int ioread[8|16|32](void *addr);
    - unsigned int iowrite[8|16|32](u[8|16|32] value, void *addr);
- Barriers
  - #include <linux/kernel.h>: void barrier(void);
  - #include <asm/system.h>: void [r|w|]mb(void);

Debugging

Debugging Options

- Printing & syslogd
- Querying
  - /proc
  - /sysfs
- Watching
  - strace
  - Oops

Introduction to Procfs

Objective:

- Understand the purpose of procfs in Linux.
- Learn how to create read/write interfaces using procfs.
- Explore practical examples for device drivers.

What is Procfs?

- **Definition:**
  - Procfs is a virtual file system in Linux that provides
  - information about processes and kernel data.
- **Mounted Location:**
  - Usually mounted at /proc.
- **Purpose:**
  - Exposes process and system information.
  - Provides an interface for user-space programs
  - to interact with kernel data.

Why Use Procfs?

- **Advantages:**
  - Provides system information in a human-readable format.
  - Allows users to configure kernel parameters.
- **Common Use Cases:**
  - Viewing process information.
  - Exposing driver-specific information.
  - Tuning kernel parameters.

Creating a Procfs

Steps to Create a Procfs Entry :

1. Use proc_create() to create an entry in /proc.
2. Define read/write operations using proc_read and proc_write functions.

R/W Operations

Code Summary:

Create /proc/simple_proc_entry.

- **Read Operation:**

  Use cat to read the value.

  cat /proc/simple_proc_entry.

- **Write Operation:**

  Use echo to write a value.

  echo 123 > /proc/simple_proc_entry.

Procfs Subdirectory

Steps to Create a Subdirectory in Procfs:

1. Use proc_mkdir() to create a directory.
2. Use proc_create() to create a file inside the directory.

Use Cases of Procfs

- **Monitoring Process Information:**
  - Exposing CPU usage, memory usage, etc.
- **Driver-Specific Information:**
  - Exposing driver states and statistics.
- **Tuning Kernel Parameters:**
  - Adjusting kernel behavior at runtime.

Best Practices

- Use procfs for process-related or kernel information.
- Avoid creating too many proc entries to prevent clutter.
- Ensure appropriate permissions to prevent unauthorized access.
- Validate user inputs to avoid kernel crashes.

Introduction to Sysfs

- **Objective:**
- Understand the purpose of sysfs in Linux.
- Learn how to create read/write interfaces using sysfs.
- Explore practical examples for device drivers.

What is Sysfs?

Definition:

- Sysfs is a virtual file system in Linux that exposes kernel objects (devices, drivers, etc.) to user space.

Mounted Location:

- Usually mounted at /sys.

Purpose:

- Provides a structured way to expose kernel attributes.
- Allows user-space programs to interact with kernel objects.

Why Use Sysfs?

- **Advantages:**
  - Human-readable interface.
  - Organized hierarchy.
  - Easily customizable.
- **Common Use Cases:**
  - Exposing device attributes.
  - Interacting with hardware parameters.
  - Controlling kernel modules.

Creating a Sysfs

Steps to Create a Sysfs Entry:

1. Use kobject_create_and_add() to create a directory in /sys.

2. Use sysfs_create_file() to create attributes.
3. Define read/write operations using show and store functions.

R/W Operations

Code Summary:

Create /sys/kernel/simple_sysfs/my_value.

Read Operation:

Use cat to read the value.

cat /sys/kernel/simple_sysfs/my_value.

Write Operation:

Use echo to write a value.

echo 42 > /sys/kernel/simple_sysfs/my_value.

Practical Use Cases

- **Exposing Device Attributes:**
  - Device temperature, voltage levels, etc.
- **Hardware Control:**
  - Enable/disable hardware features.
- **Driver Debugging:**
  - Exposing internal driver states for debugging.

Best Practices

- Use sysfs for hardware-related attributes.
- Organize entries in a hierarchical structure.
- Ensure appropriate permissions to prevent unauthorized access.
- Validate user inputs to avoid kernel crashes.

Feedback Time

Thank You