CSE 511: Operating Systems Design

Lectures 1,2

Course Logistics

OS Kernel Designs

OS Boot Process

Course Logistics

- **Instructor:** Dr. Ruslan Nikolaev
  - Assistant Professor, has prior industry background
  - Westgate Bldg W331, rnikola@psu.edu
  - Office hours (tentative): Wednesday, 10:00 AM-12:00 PM
- **TA:** Doug Rumbaugh
  - drumbaugh@psu.edu
- Try getting in touch with the TA first, then contact the instructor (but do contact directly if there is a ***real*** urgency)!
- **Lectures:** Tuesday, Thursday, 3:05-4:20 PM
  - Sackett Bldg 108

Prerequisites and Grading

- Graduate standing and/or CMPSC 473 or equivalent (ask the instructor if you are not sure), proficiency with the C language
- Tentative grading
  - 60% programming assignments (4+ assignments)
  - 15% midterm
  - 25% final exam
  - We may add one course project which will be performed by several students (most likely 2)
    - Then grading will be reconsidered - some above item(s) will be readjusted and/or completely substituted
  - We may also add in-class quizzes **(one quiz is certainly coming up this Thursday!)**

Technology Requirements

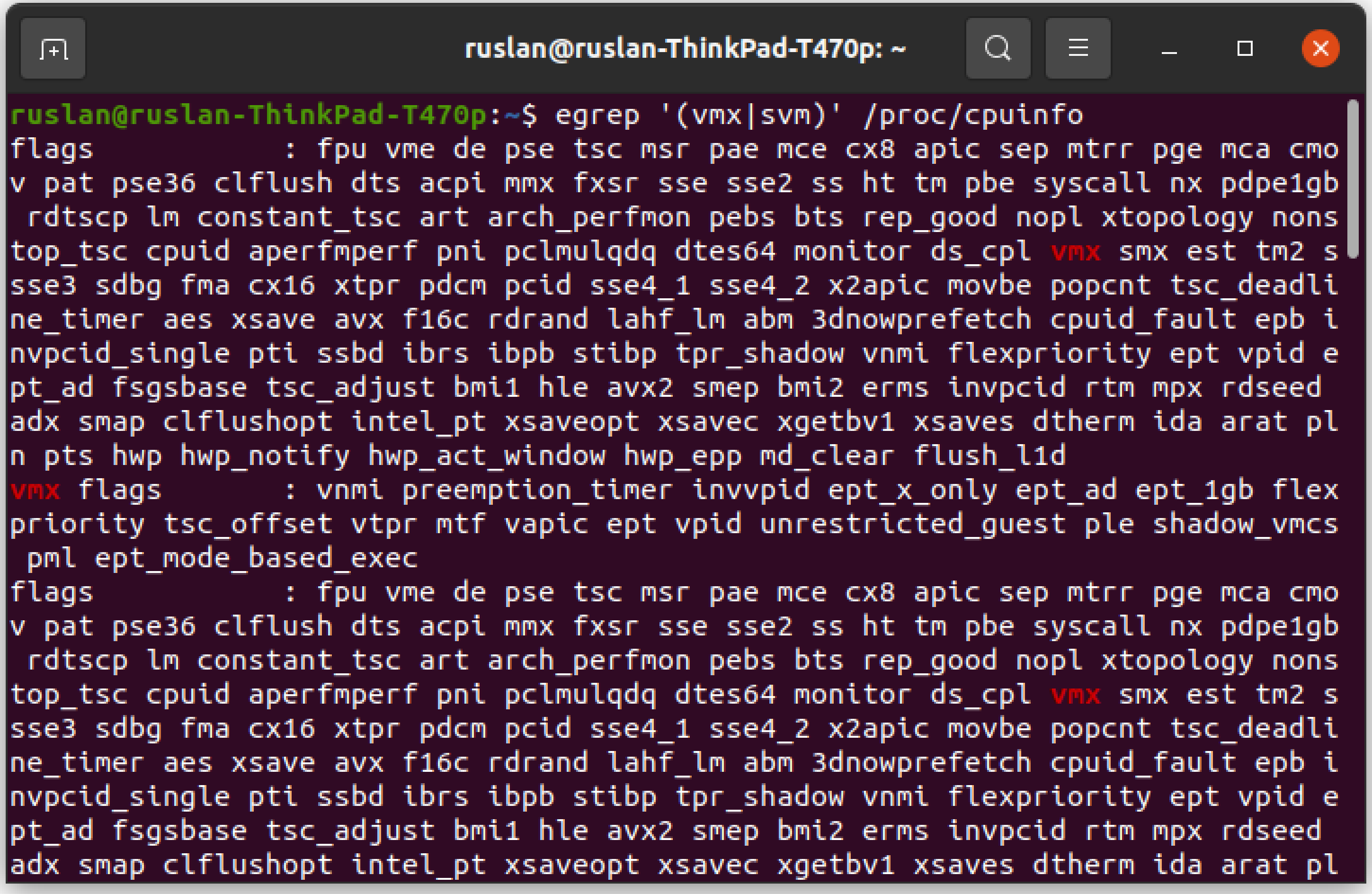
- For programming assignments, you need to install Linux
  - Check if your machine is x86-64 (not recent Apple M1!)
  - Check if your machine supports virtualization (most likely)
  - We will use VirtualBox and/or qemu (possibly WSL for compilation only)
    - I recommend using your own local machine, but we will explore if/how lab machine can be used
  - Ubuntu Desktop 22.04.1 LTS is recommended
    - https://releases.ubuntu.com/22.04/ubuntu-22.04.1-desktop-amd64.iso

Technology Requirements

- Ideally, install Linux **natively**, i.e., directly without virtualization
  - Then you can run VirtualBox and/or qemu directly from Linux
  - **Be careful not to erase everything from your system while installing Linux! Make sure you fully understand what you are doing. Fully backup your data!**
    - If you are not sure, do not install Linux like that. Use virtualization instead (e.g., run Linux via VirtualBox and/or use WSL on Windows)
    - Please see the WSL Tutorial under Module->Tutorials

Technology Requirements

- For virtualization, look for the ‘vmx’ or ‘svm’ CPU flags



What Would You Get Out of This Course?

- Basic knowledge of low-level OS parts
  - The boot strapping process, UEFI firmware
  - OS Design Concepts: kernel, processes, threads
  - APIC interrupt controller, exceptions
  - Symmetric Multi-Processing (SMP)
- Virtualization
- File Systems
- Concurrency (locks, lock- and wait-free algorithms)
- Distributed systems (as time allows)

Why That Matters?

- For successful systems research (or more specifically – OS research), you need to have both
  - Required theoretical background
  - Deep understanding of certain OS low-level pieces, so that you can easily work with and modify them, thereby advance state-of-the-art

Why That Matters?

- OS and virtualization techniques are experiencing a renewed interest
- In cloud computing, OS and virtualization approaches are being revisited to understand how
  - They can support application software systems
  - Satisfy increasing demands on security, performance, failure resilience, and compatibility

Intended Learning Objectives

- Learn various OS kernel, virtualization and concurrency approaches
- Hands-on experience with real systems
  - Programming assignments
- Building expertise in systems, which can help in research
  - Feel more confident when modifying low-level pieces of existing systems or building new ones

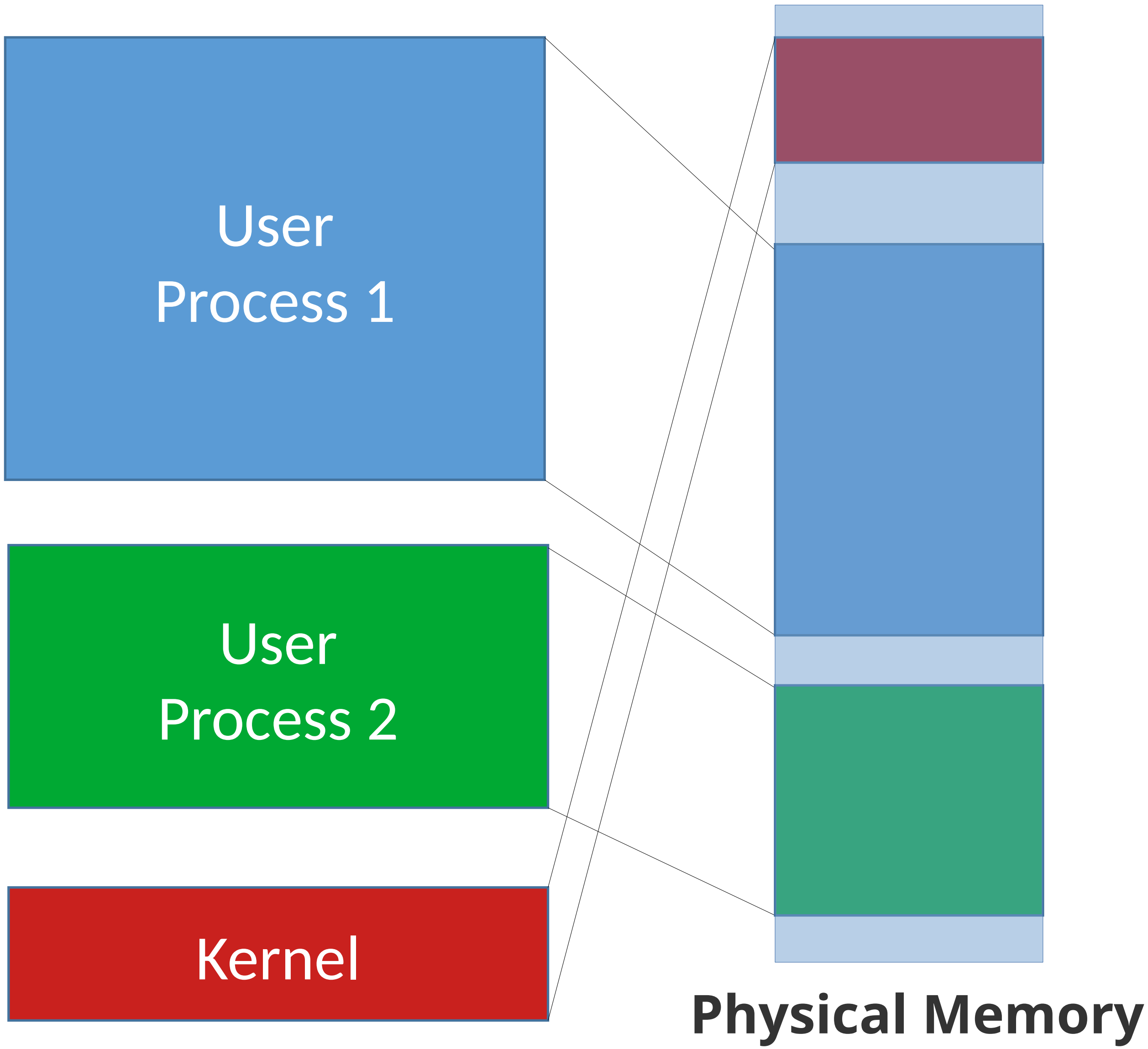
What Is an Operating System (OS)?

- An OS is an *essential* part of modern computer systems
  - It manages system resources such as CPU, memory, and peripheral devices
  - It provides standard interfaces for user programs to access these resources safely and concurrently
- OS kernels can be organized differently based on the privilege separation and code sharing
  - Monolithic kernels
  - Microkernels
  - Library OS

Recap: Virtual Address Space

- To isolate programs, OSs use the concept of a *process*
- Each process has its own *virtual address space*, which maps a *slice* of physical memory to virtual memory
- Each process has its own virtual address space
  - One process cannot adversely affect another process (at least not directly)
  - Virtual address space is programmed through a *page table*
    - *The page table contains physical-to-virtual mappings*

Recap: Virtual Address Space



Recap: Privilege Separation

- Modern CPUs run program code in two modes
  - *Privileged mode*, which provides unrestricted access to all CPU instructions
  - *Unprivileged mode*, which restricts access to certain CPU instructions, e.g., page table manipulation
- A page table entry indicates if a given page is
  - *A kernel page* (aka ‘ring 0’ in x86-64), accessible only in the privileged mode
  - *A user page* (aka ‘ring 3’ in x86-64), accessible in *both* modes

Recap: Privilege Separation

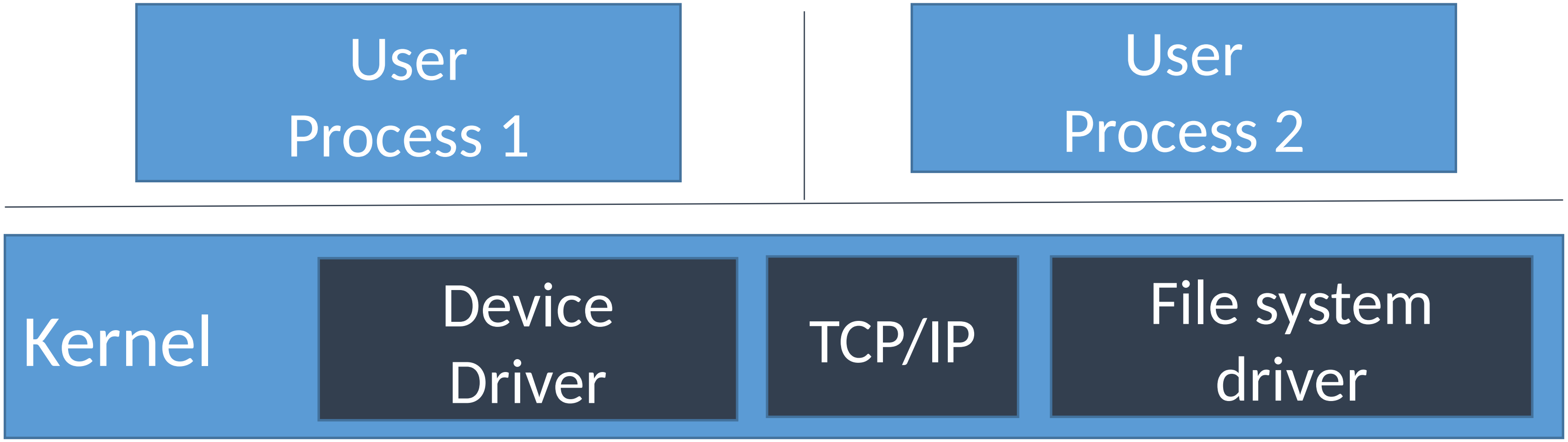
- Switching from the *privileged* to *unprivileged* mode can happen anywhere in the program code (e.g., ***sysret*** in x86-64)
- Switching from the *unprivileged* to *privileged* mode can only happen through special CPU-controlled *gates* (e.g., ***syscall*** in x86-64)
  - They are also known as '*system calls*'
  - Each system call has a special entry point (an address programmed in memory, CPU exception, etc)
  - The handler typically does not trust any input parameters and subject them to additional verification

OS Kernels

- Monolithic kernels
- Microkernels
- Library OS designs
  - Kernel-bypass libraries

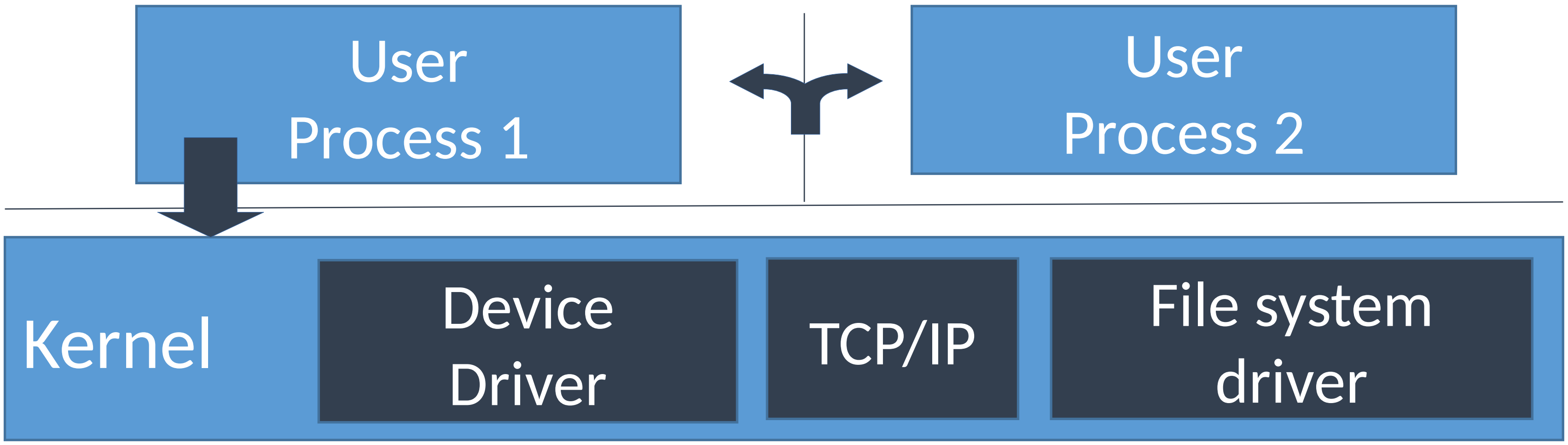
Monolithic Kernels

- Core OS code (e.g., drivers) runs in the *privileged* mode
- Only ordinary user processes run in the *unprivileged* mode



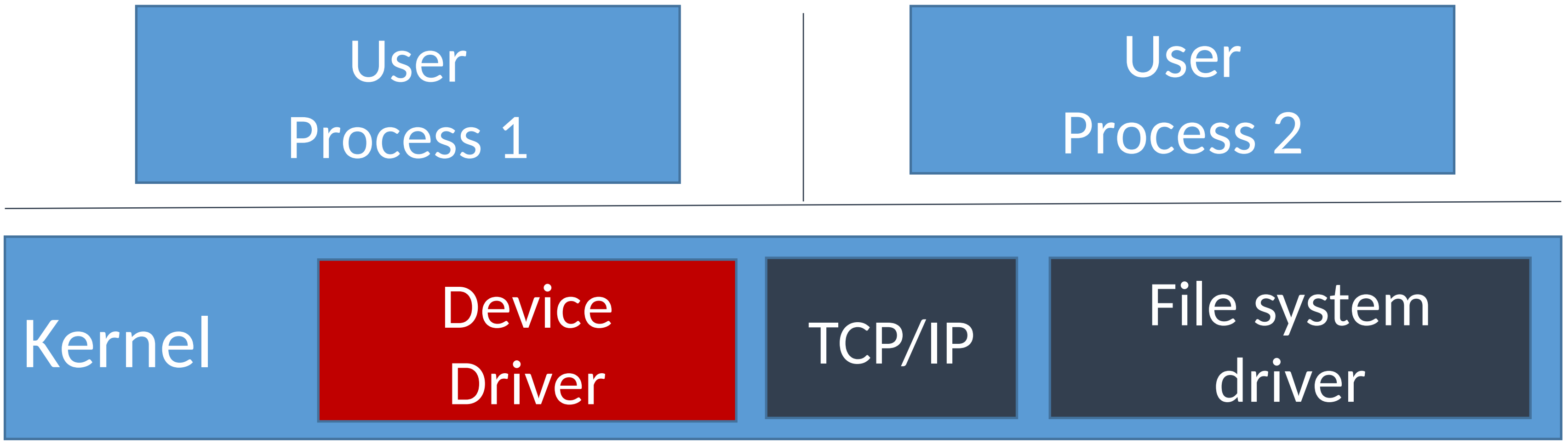
Monolithic Kernels

- Processes are fully isolated from each other
- OS API: system calls from user processes to the kernel



Monolithic Kernels

- A buggy or malicious driver can corrupt the OS memory and bring down the entire system

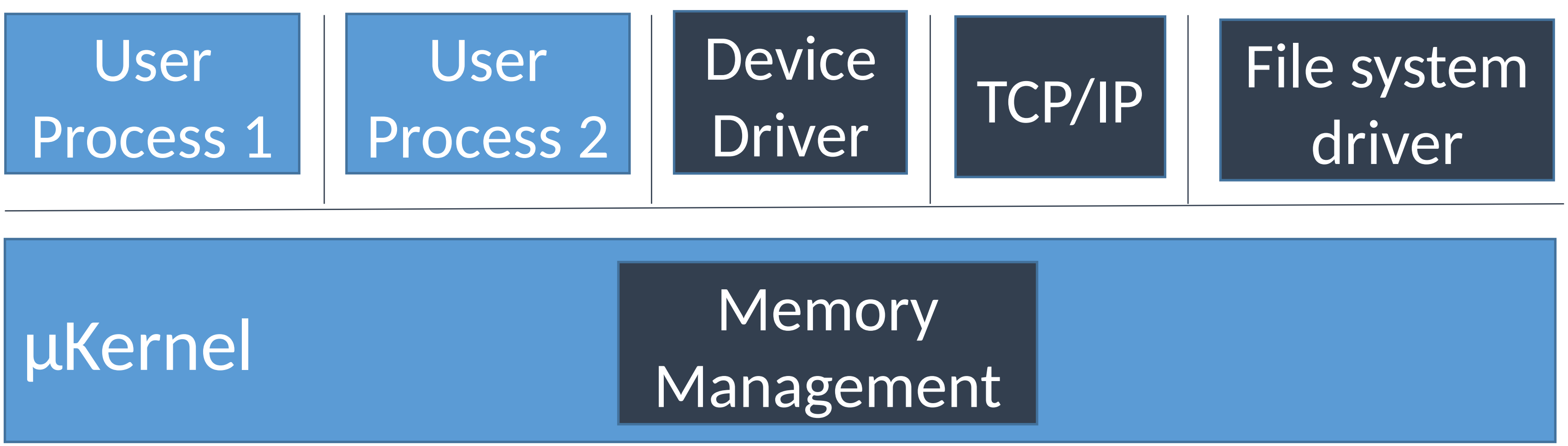


Monolithic Kernels: Trade-offs

- Preferred by most general-purpose OSs (mac, Windows, Linux, etc)
- Good performance
  - Unless dealing with extremely fast I/O
- Lack of isolation, limited recoverability
- Larger trusted code base
  - Potential security problems

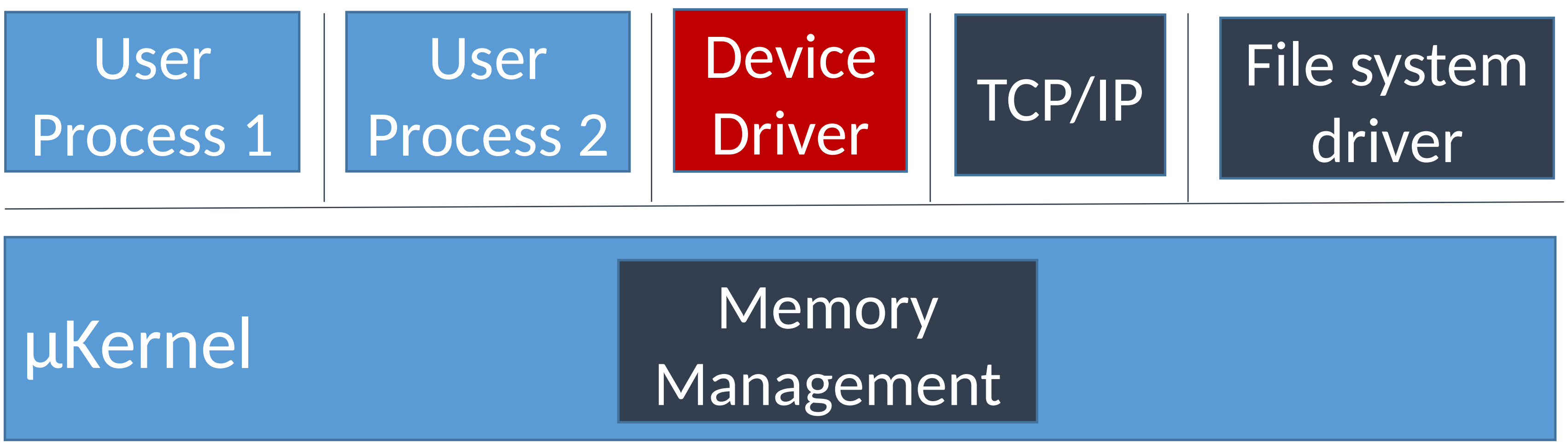
Microkernels

- Only the most fundamental parts (e.g., memory management) run in the *privileged* mode
- Many core components run in the *unprivileged* mode along with ordinary user processes



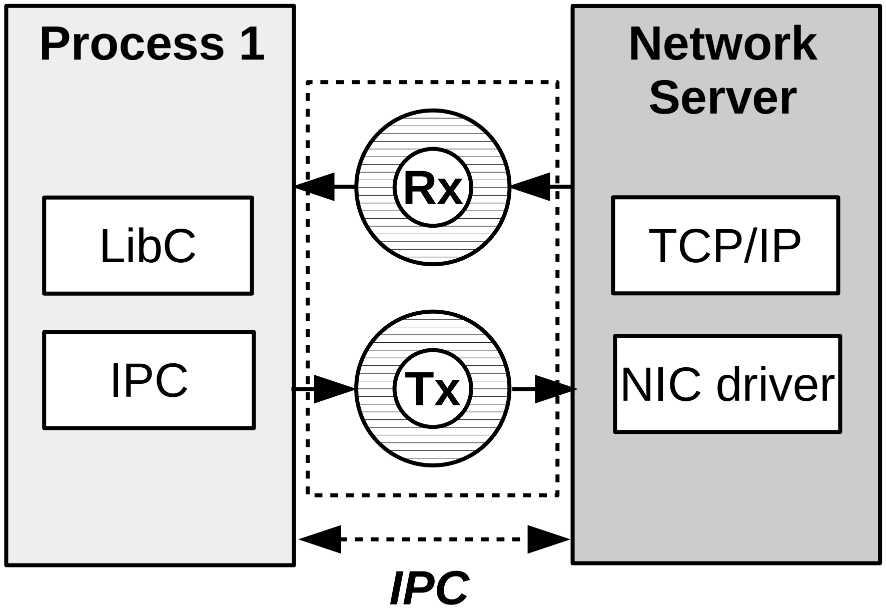
Microkernels

- A buggy or malicious driver **cannot** corrupt the OS memory, and the system can still (potentially) be recovered



Microkernels: Multiserver OS

- Using servers to run core components
- API: IPC (inter-process communication) between two processes
  - Can be some sort of “shared memory” between two processes

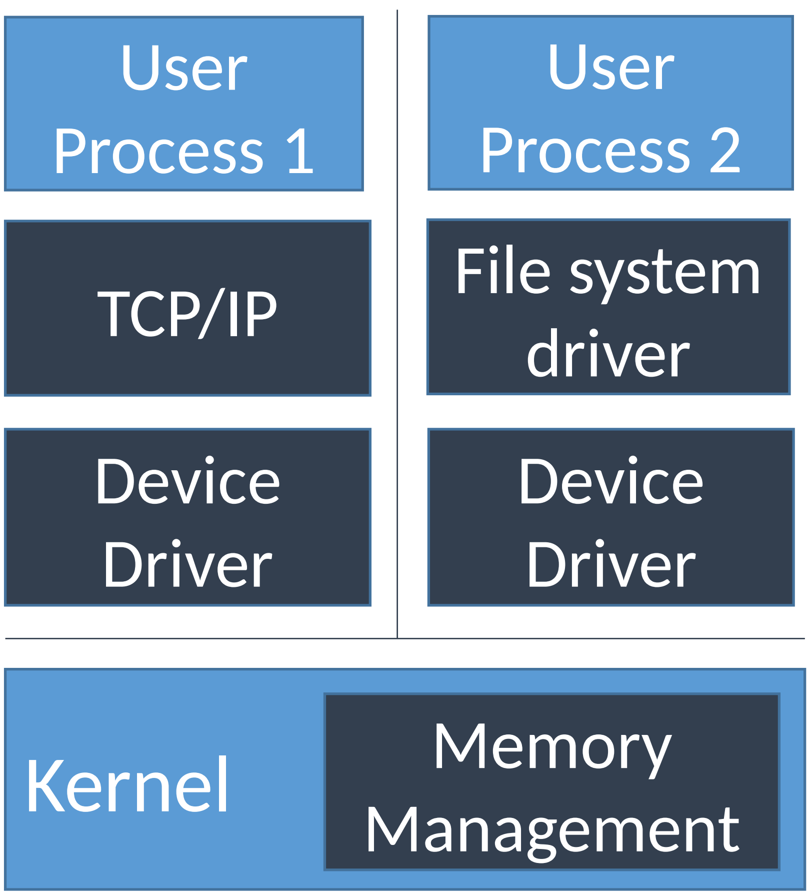


Microkernels: Trade-offs

- Not very popular, many implementations are academia-driven (e.g., MINIX 3)
- Performance can vary
  - IPCs are more expensive than system calls
  - But multi-core systems can avoid context switches and have better TLB and cache locality
- Better isolation, fault-tolerance, and recoverability
- Reduced trusted code base
  - Can be beneficial for security

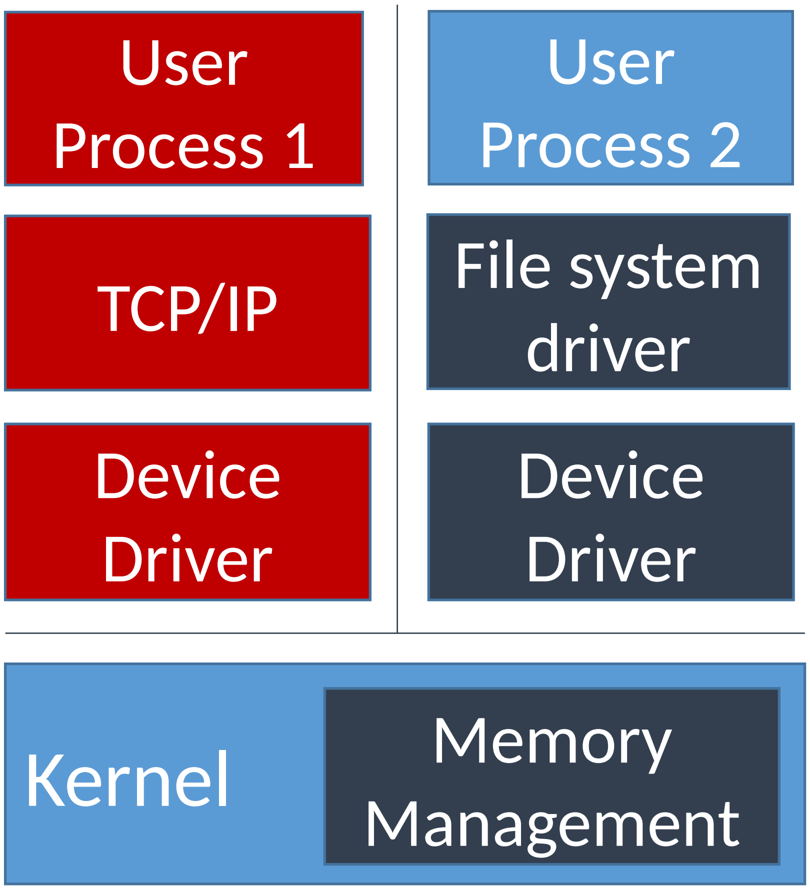
Library OS

- We can also have an application-specific OS design; e.g., run device drivers directly in the address space of programs
- API: through regular (function) calls



Library OS

- A buggy or malicious driver **cannot** corrupt the OS memory, only a specific application is affected



Library OS

- Gaining some popularity but mostly academia-driven (e.g., LibOS/exokernel)
- Great performance
  - More direct access to hardware, avoiding system calls
- Resource sharing can be more challenging
  - Depends on the actual resource that needs to be shared
- Reduced trusted code base
  - Can be beneficial for security

Example: Linux

- The Linux kernel
  - vmlinuz is the kernel image

```
ruslan@ruslan-ThinkPad-T470p:/boot$ ls -l *5.8.0-38*
-rw-r--r-- 1 root root   248291 Jan 12 10:15 config-5.8.0-38-generic
-rw-r--r-- 1 root root 52039171 Jan 15 10:28 initrd.img-5.8.0-38-generic
-rw------- 1 root root  5515823 Jan 12 10:15 System.map-5.8.0-38-generic
-rw------- 1 root root  9716608 Jan 12 10:39 vmlinuz-5.8.0-38-generic
ruslan@ruslan-ThinkPad-T470p:/boot$
```

Example: Linux

- Linux modules

ruslan@ruslan-ThinkPad-T470p: /lib/modules/5.8.0-38-generic

```
ruslan@ruslan-ThinkPad-T470p:/lib/modules/5.8.0-38-generic$ ls -l
total 6012
lrwxrwxrwx  1 root root      39 Jan 12 10:15 build -> /usr/src/linux-headers-5.8
.0-38-generic
drwxr-xr-x  2 root root    4096 Jan 12 10:15 initrd
drwxr-xr-x 16 root root    4096 Jan 15 10:28 kernel
-rw-r--r--  1 root root 1423433 Jan 15 10:28 modules.alias
-rw-r--r--  1 root root 1402618 Jan 15 10:28 modules.alias.bin
-rw-r--r--  1 root root    9975 Jan 12 10:15 modules.builtin
-rw-r--r--  1 root root   25773 Jan 15 10:28 modules.builtin.alias.bin
-rw-r--r--  1 root root   12521 Jan 15 10:28 modules.builtin.bin
-rw-r--r--  1 root root   77263 Jan 12 10:15 modules.builtin.modinfo
-rw-r--r--  1 root root  649563 Jan 15 10:28 modules.dep
-rw-r--r--  1 root root  900743 Jan 15 10:28 modules.dep.bin
-rw-r--r--  1 root root     330 Jan 15 10:28 modules.devname
-rw-r--r--  1 root root  229525 Jan 12 10:15 modules.order
-rw-r--r--  1 root root     885 Jan 15 10:28 modules.softdep
-rw-r--r--  1 root root  620584 Jan 15 10:28 modules.symbols
-rw-r--r--  1 root root  757020 Jan 15 10:28 modules.symbols.bin
drwxr-xr-x  3 root root    4096 Jan 15 10:27 updates
drwxr-xr-x  3 root root    4096 Jan 15 10:27 vdso
ruslan@ruslan-ThinkPad-T470p:/lib/modules/5.8.0-38-generic$
```

Example: Linux

- Linux Drivers

```
ruslan@ruslan-ThinkPad-T470p: /lib/modules/5.8.0-38-generic
ruslan@ruslan-ThinkPad-T470p:/lib/modules/5.8.0-38-generic$ ls kernel/drivers/
acpi        dax       hwtracing   media     pci        siox         vfio
android     dca       i2c         memstick  pcmcia     slimbus      vhost
ata         dma       i3c         message   phy        soc          video
atm         edac      iio         mfd       pinctrl    soundwire    virt
auxdisplay  extcon    infiniband  misc      platform   spi          virtio
base        firewire  input       mmc       power      spmi         visorbus
bcma        firmware  iommu       most      powercap   ssb          vme
block       fpga      ipack       mtd       pps        staging      w1
bluetooth   gnss      irqchip     mux       ptp        target       watchdog
bus         gpio      isdn        net       pwm        tee          xen
char        gpu       leds        nfc       rapidio    thermal
clk         greybus   lightnvm    ntb       regulator  thunderbolt
counter     hid       macintosh   nvdimm    reset      tty
cpufreq     hsi       mailbox     nvme      rpmsg      uio
cpuidle     hv        mcb         nvmem     rtc        usb
crypto      hwmon     md          parport   scsi       vdpa
ruslan@ruslan-ThinkPad-T470p:/lib/modules/5.8.0-38-generic$
```

Linux module

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/page.h>

static int __init ruslan_module_init(void) {
    printk(KERN_WARNING "Hello, I am a Linux module\n");
    printk(KERN_WARNING "Hurrah! I have unlimited access to the physical
memory\n");
    printk(KERN_WARNING "For a legacy BIOS system, we can even read last
BIOS timer value...\n");
    printk(KERN_WARNING "0x%x\n", *(short *) __va(0x046C));
    printk(KERN_WARNING "... but it is stale since BIOS is only used to
boot an OS!\n");
    return 0;
}

static void __exit ruslan_module_exit(void) {
    printk(KERN_WARNING "Bye!\n");
}
```

Linux module

```
module_init(ruslan_module_init);
module_exit(ruslan_module_exit);

MODULE_AUTHOR("Ruslan Nikolaev");
MODULE_DESCRIPTION("An example module");
MODULE_LICENSE("GPL");
MODULE_VERSION("1.0");
```

Linux module: Makefile

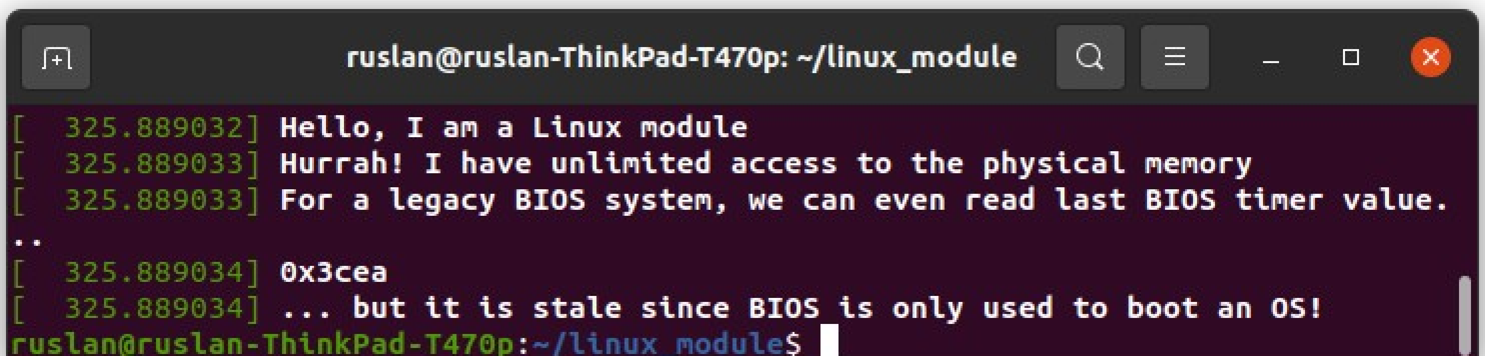
```
obj-m += ruslan_module.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Linux module: Running

```
sudo insmod ./ruslan_module.ko
sudo rmmod ruslan_module
```



OS Boot Process

- How do we boot an OS from USB, network card, etc?
  - Would not we need to have some drivers installed for that just like in an OS?
- There are two sets of device drivers
  - Basic I/O drivers in the system firmware
  - OS drivers

What is the System Firmware?

- Generally speaking, firmware is an “embedded program” for any piece of hardware
- The system firmware is the first program to run when the power is on
- There are three common types of the system firmware
  - BIOS (Basic Input/Output System), used in legacy x86 systems, very old, heavily relied on 16-bit (!) code
  - Open Firmware, common in SPARC and PowerPC systems
  - UEFI (Unified Extensible Firmware Interface)
    - EFI initially replaced BIOS for Itanium, later adopted by Apple for x86, and recently adopted for many platforms

BIOS (Basic Input/Output System)

- Was initially introduced as a system-specific part of DOS and was widely used in a “pure form” until around 2010
  - MS-DOS could really use it directly for input/output (disk access, mouse, keyboard, video graphics, etc)
  - Outlived DOS by more than 15 years! No one wants to deal with legacy 16-bit mode after an OS is booted
  - Consequently, only used for system initialization and by an OS boot loader
- This acronym is still loosely used to denote UEFI-powered systems even though there is no “real” BIOS anymore
  - UEFI (pre-2020) often has a legacy BIOS boot module (CSM) for x86-64

UEFI Firmware

- Used by all newer hardware but legacy (BIOS) boot is still supported
  - Legacy BIOS boot was supposed to be phased out completely in 2020. Cannot boot MS-DOS anymore.
  - Your machine may still use legacy boot!
- **Since all machines vary, and BIOS legacy boot is still widely used, we will use VirtualBox and qemu for our assignments/projects to make things simpler**
  - UEFI boot will work even for legacy machines
  - Ubuntu: `sudo apt-get install virtualbox`