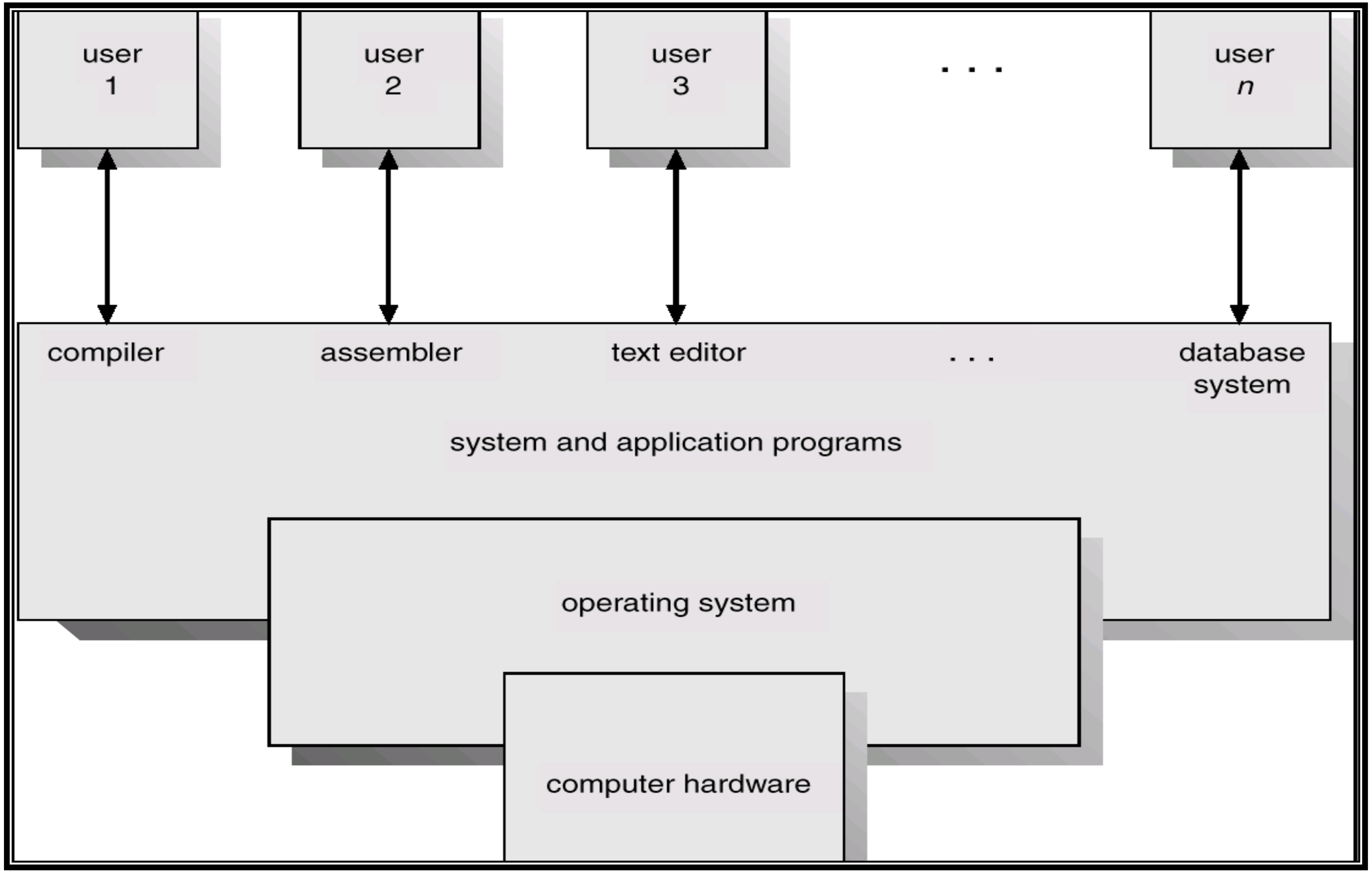
Advanced OS - Kernels

Manish Shrivastava

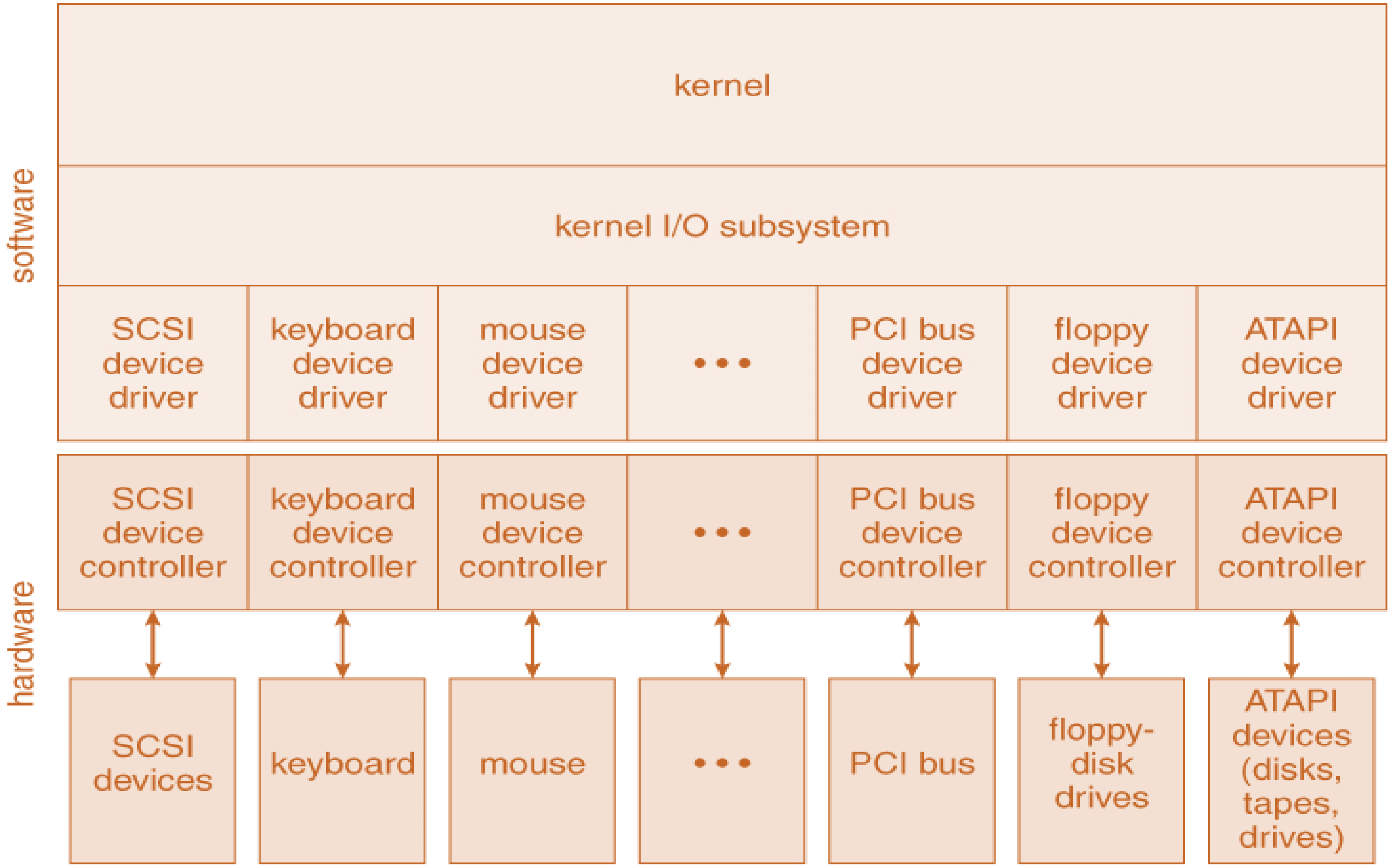
Computer OS components

- **Hardware** – provides basic computing resources (CPU, memory, I/O devices).
- **Operating system** – controls and coordinates the use of the hardware among the various application programs for the various users.
- **Applications programs** – define the ways in which the system resources are used to solve the computing problems of the users (compilers, database systems, video games, business programs).
- **Users** (people, machines, other computers).

Computer OS components



Kernel

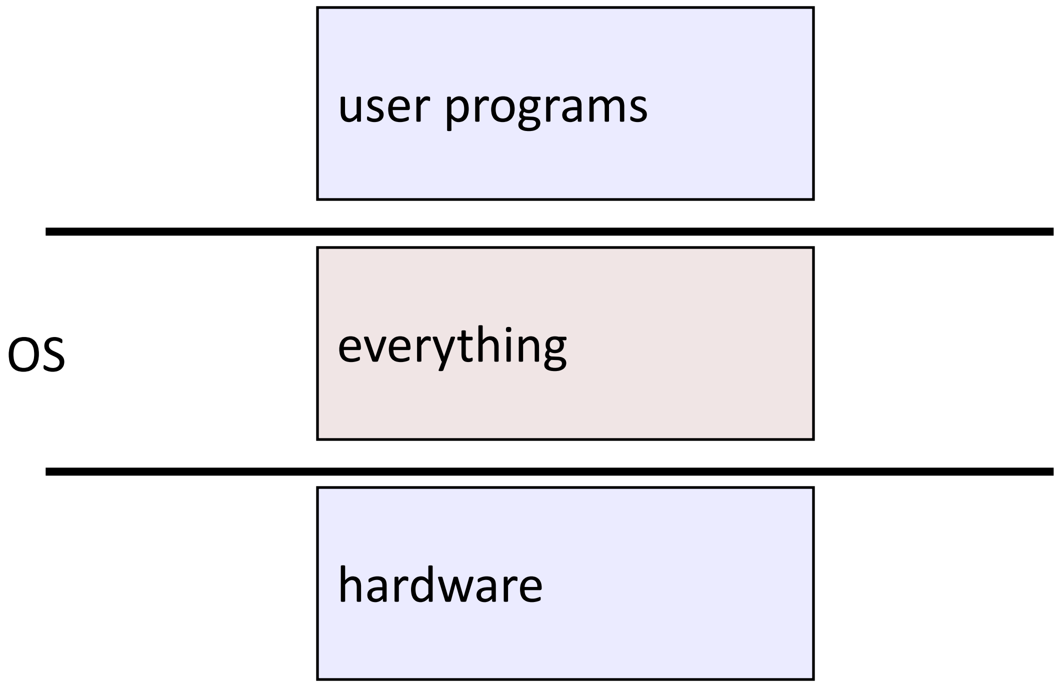

kernel
kernel I/O subsystem
software
SCSI device driver
keyboard device driver
mouse device driver
• • •
PCI bus device driver
floppy device driver
ATAPI device driver
hardware
SCSI device controller
keyboard device controller
mouse device controller
• • •
PCI bus device controller
floppy device controller
ATAPI device controller
SCSI devices
keyboard
mouse
• • •
PCI bus
floppy-disk drives
ATAPI devices (disks, tapes, drives)

Types of Kernels

1. Monolithic Kernel: Kernels where the user services and the kernel services are implemented in the same memory space
2. Microkernel: the user services and kernel services are implemented into different spaces
3. Hybrid Kernel: Duh!!

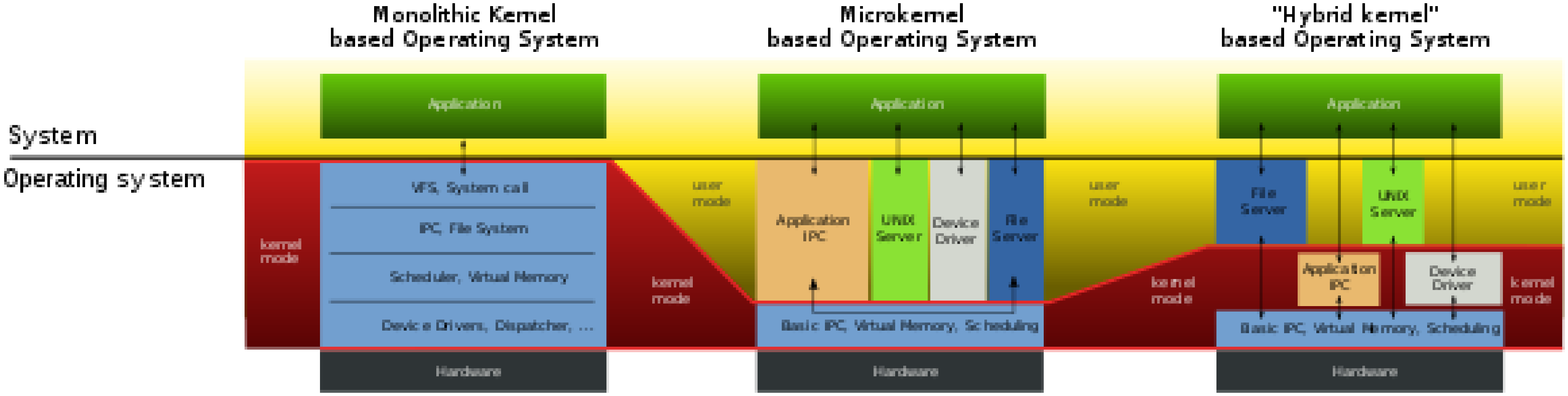
Early structure: Monolithic

- Traditionally, OS's (like UNIX) were built as a monolithic entity:

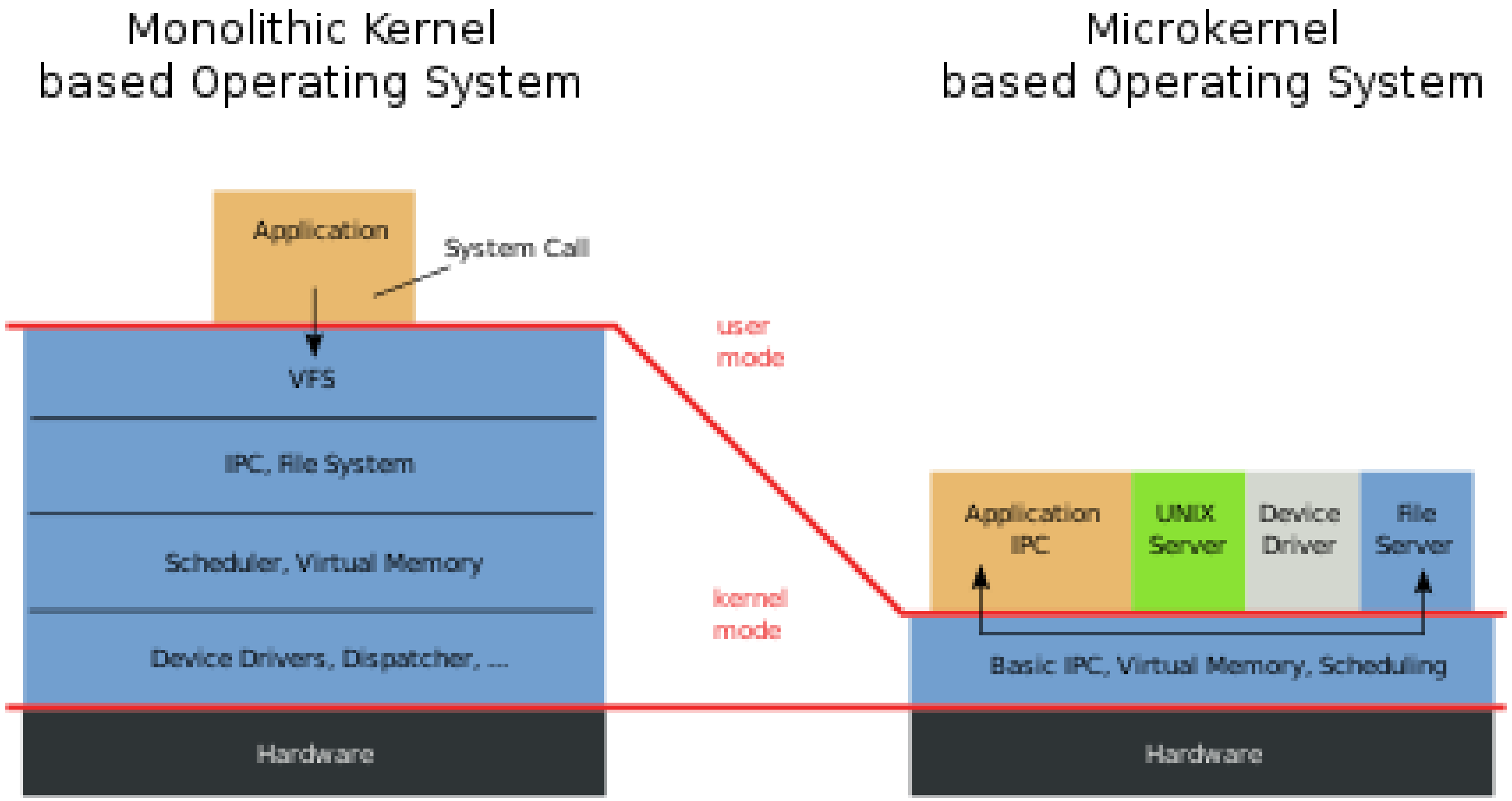


Monolithic design

- Major advantage:
  - cost of module interactions is low (procedure call)
- Disadvantages:
  - hard to understand
  - hard to modify
  - unreliable (no isolation between system modules)
  - hard to maintain
- What is the alternative?
  - find a way to organize the OS in order to simplify its design and implementation



Monolithic Kernel
based Operating System
Microkernel
based Operating System
"Hybrid kernel"
based Operating System
System
Operating system
Application
Application
Application
VFS, System call
IPC, File System
Scheduler, Virtual Memory
Device Drivers, Dispatcher, ...
kernel mode
user mode
kernel mode
Application IPC
UNIX Server
Device Driver
File Server
user mode
File Server
UNIX Server
user mode
Basic IPC, Virtual Memory, Scheduling
kernel mode
Application IPC
Device Driver
kernel mode
Basic IPC, Virtual Memory, Scheduling
Hardware
Hardware
Hardware


Monolithic Kernel
based Operating System
Microkernel
based Operating System
Application
System Call
VFS
IPC, File System
Scheduler, Virtual Memory
Device Drivers, Dispatcher, ...
Hardware
user
mode
kernel
mode
Application
IPC
UNIX
Server
Device
Driver
File
Server
Basic IPC, Virtual Memory, Scheduling
Hardware

Layering

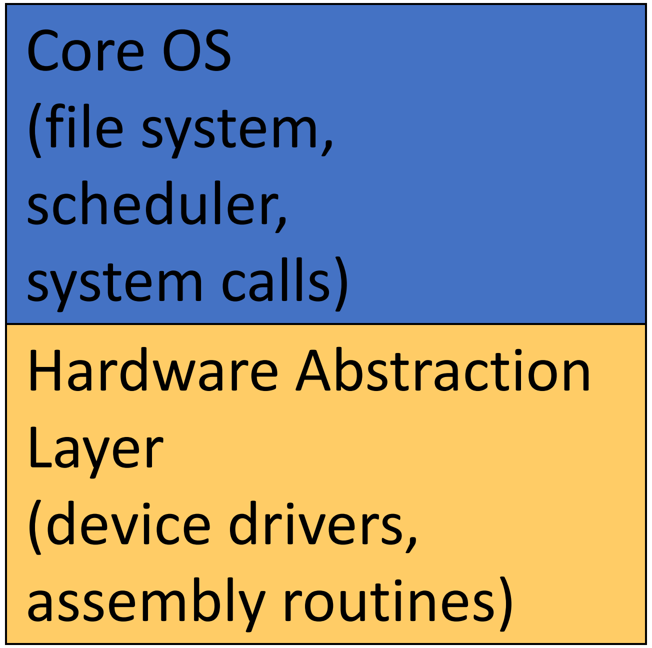
- The traditional approach is layering
  - implement OS as a set of layers
  - each layer presents an enhanced ‘virtual machine’ to the layer above
- The first description of this approach was Dijkstra’s THE system
  - Layer 5: Job Managers
    - Execute users’ programs
  - Layer 4: Device Managers
    - Handle devices and provide buffering
  - Layer 3: Console Manager
    - Implements virtual consoles
  - Layer 2: Page Manager
    - Implements virtual memories for each process
  - Layer 1: Kernel
    - Implements a virtual processor for each process
  - Layer 0: Hardware
- Each layer can be tested and verified independently

Problems with layering

- Imposes hierarchical structure
  - but real systems are more complex:
    - file system requires VM services (buffers)
    - VM would like to use files for its backing store
  - strict layering isn't flexible enough
- Poor performance
  - each layer crossing has overhead associated with it
- Disjunction between model and reality
  - systems modeled as layers, but not really built that way

Hardware Abstraction Layer

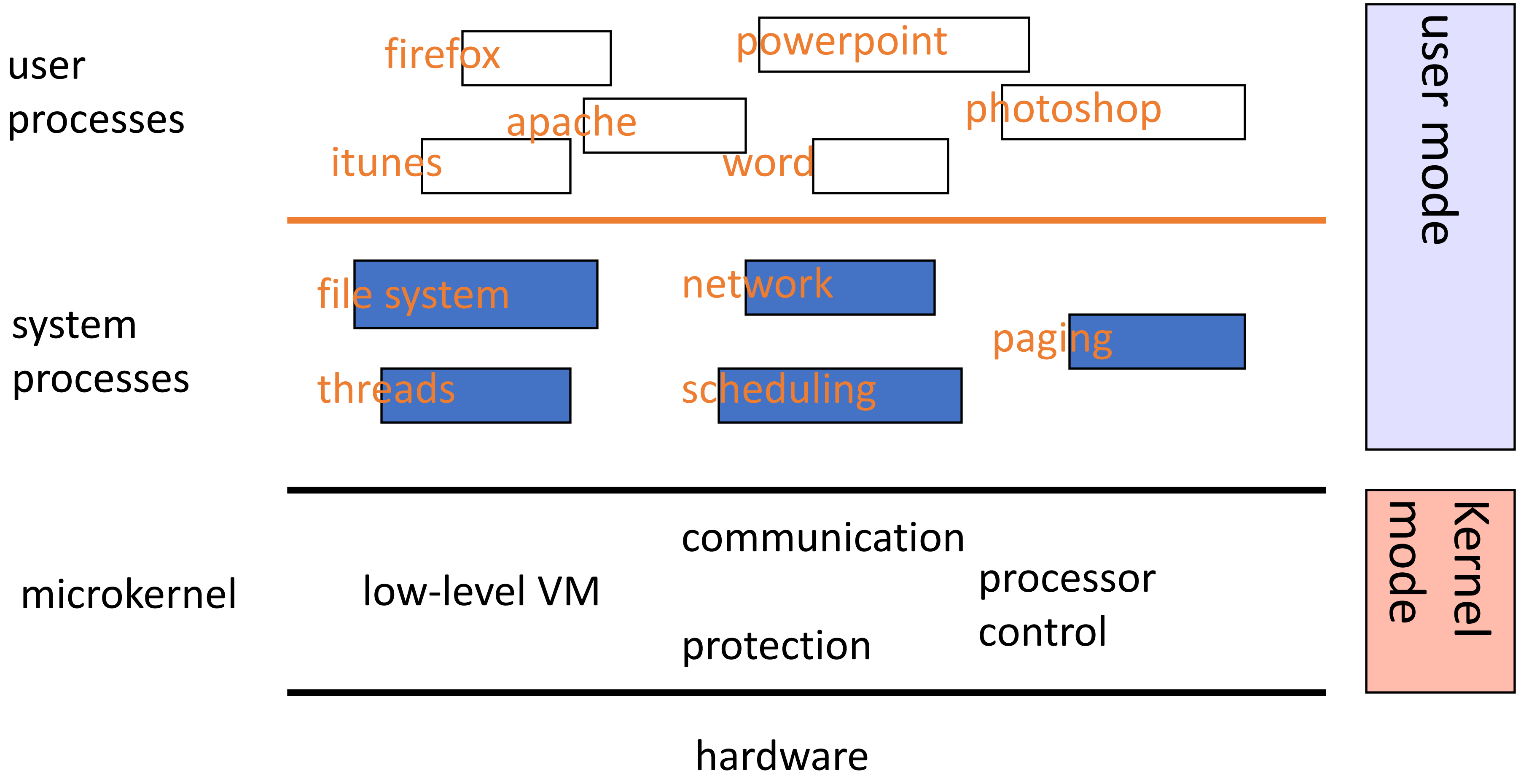
- An example of layering in modern operating systems
- Goal: separates hardware-specific routines from the “core” OS
  - Provides portability
  - Improves readability



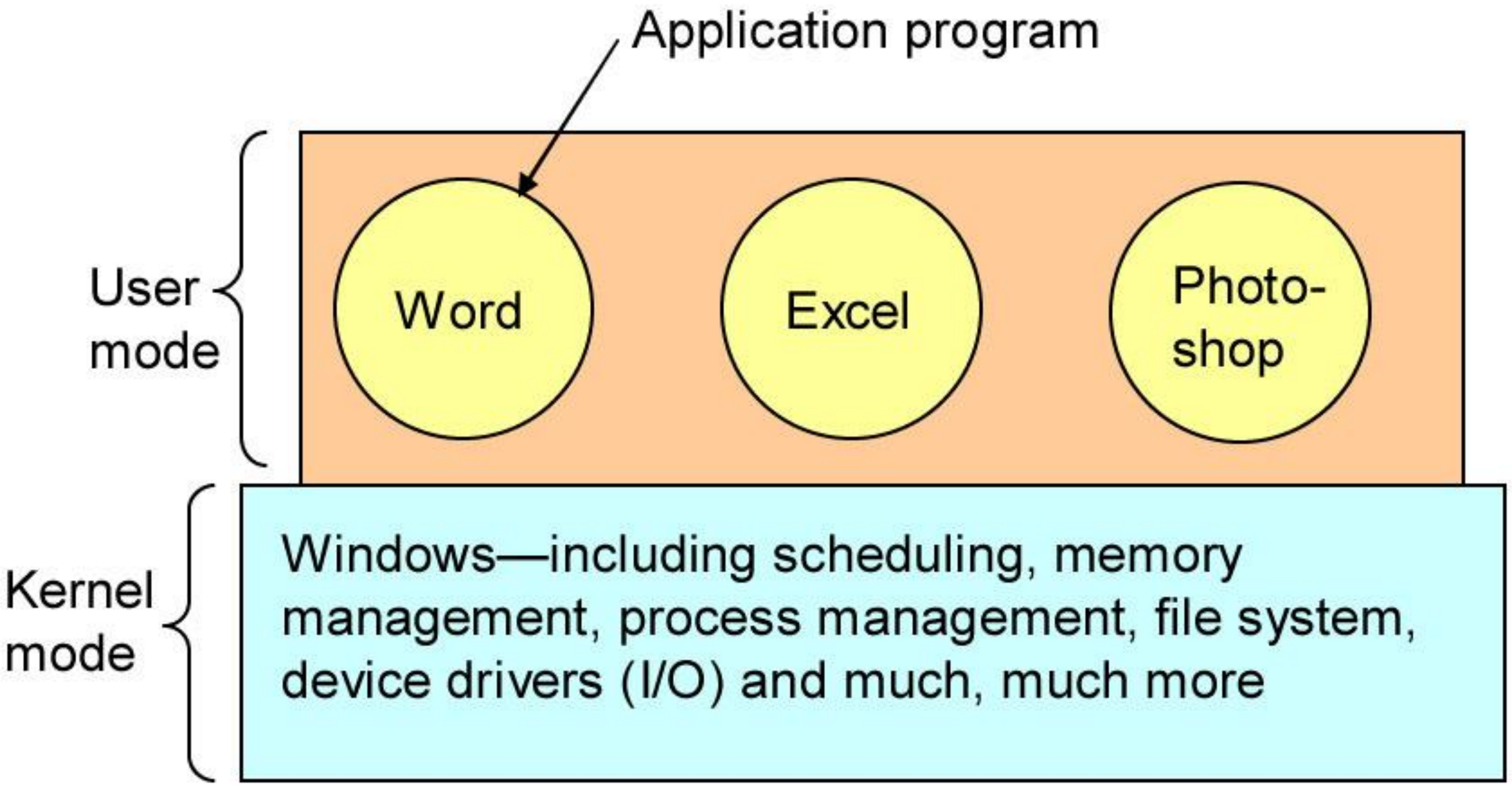
Microkernels

- Popular in the late 80's, early 90's
  - recent resurgence of popularity
- Goal:
  - minimize what goes in kernel
  - organize rest of OS as user-level processes
- This results in:
  - better reliability (isolation between components)
  - ease of extension and customization
  - poor performance (user/kernel boundary crossings)
- First microkernel system was Hydra (CMU, 1970)
  - Follow-ons: Mach (CMU), Chorus (French UNIX-like OS), OS X (Apple), in some ways NT (Microsoft)

Microkernel structure illustrated

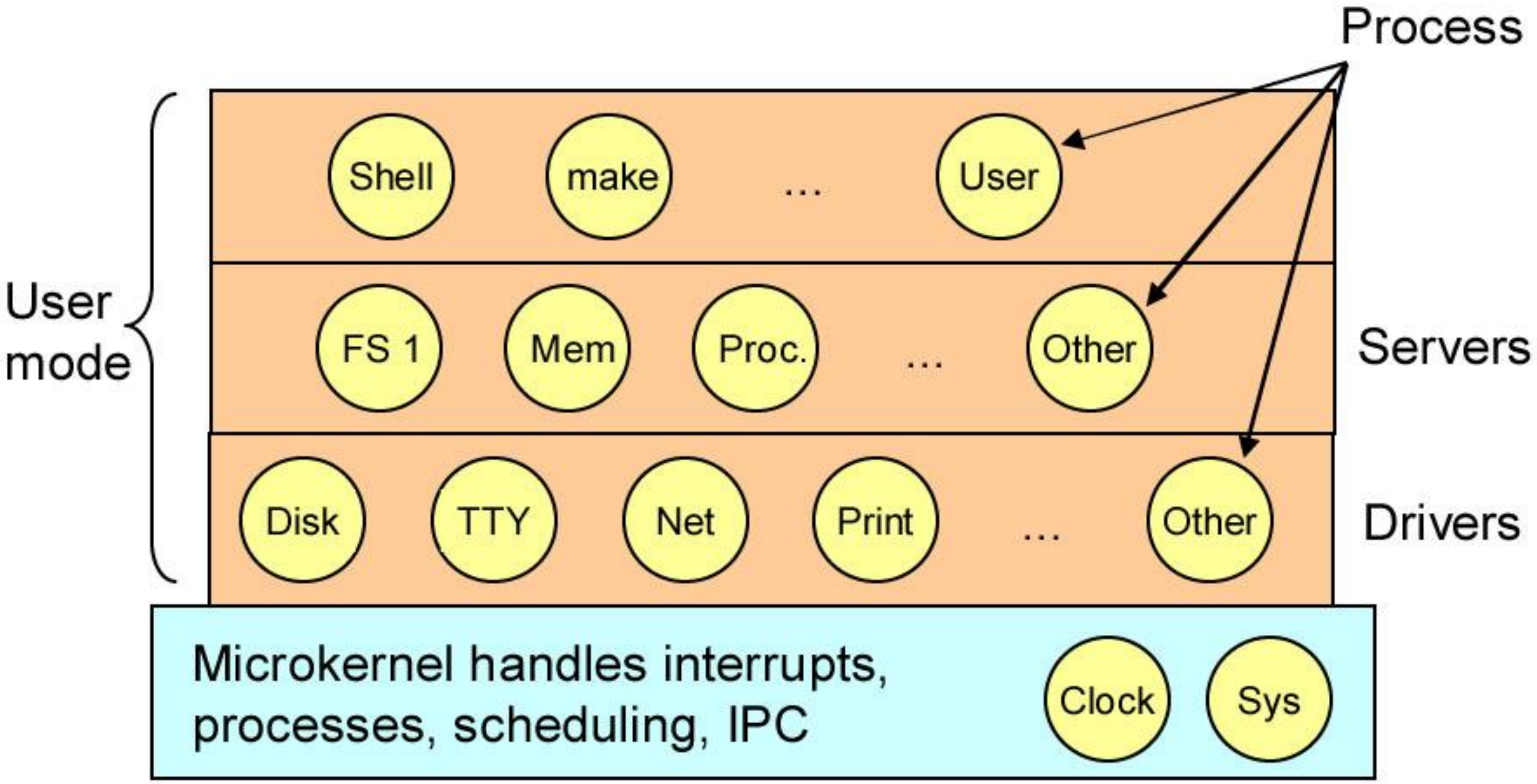


EXAMPLE: WINDOWS



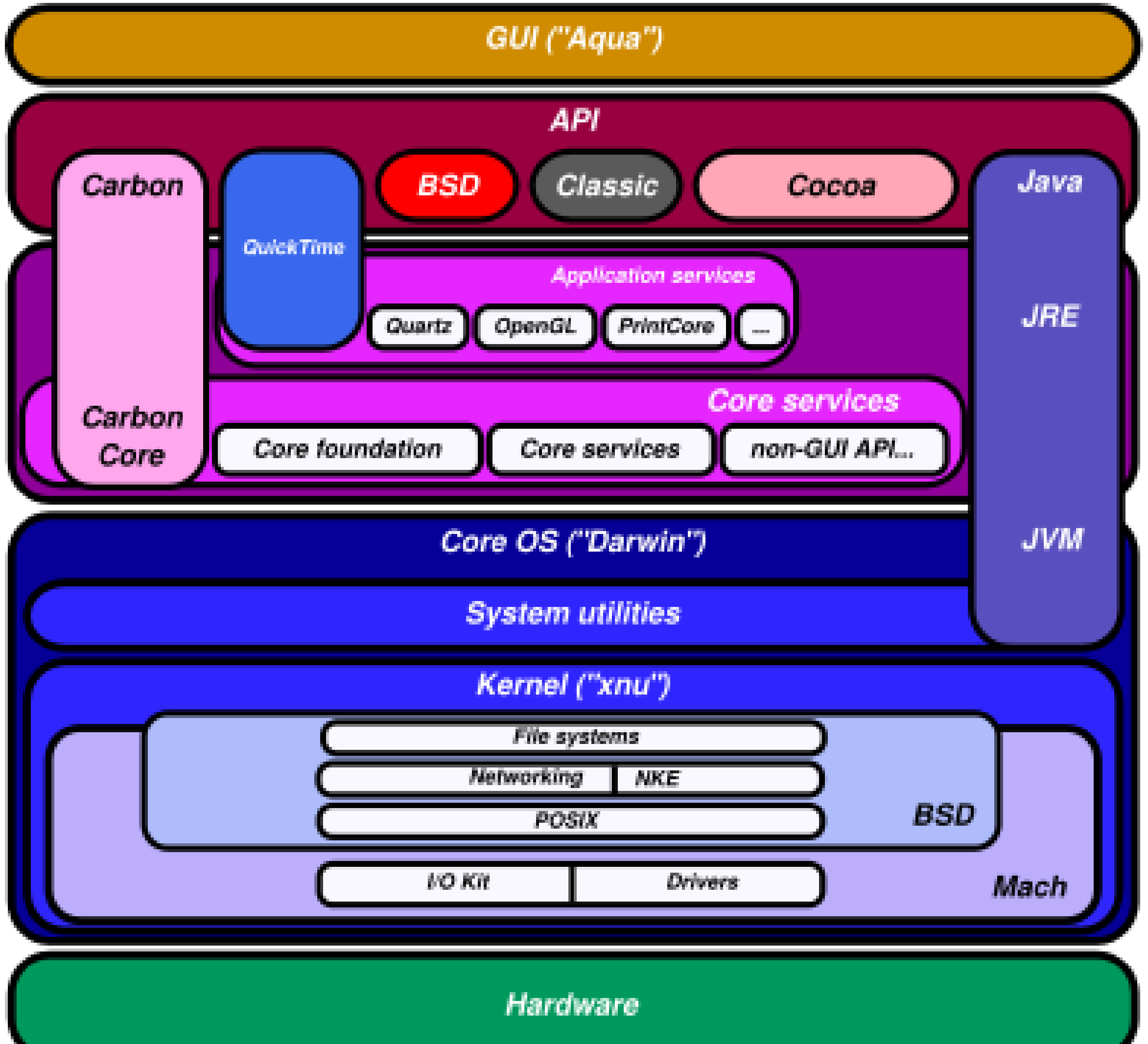
From Andy Tanenbaum

ARCHITECTURE OF MINIX 3



From Andy Tanenbaum

Mac OS X



GUI ("Aqua")
API
Carbon
QuickTime
BSD
Classic
Cocoa
Java
Application services
Quartz
OpenGL
PrintCore
...
JRE
Carbon Core
Core services
Core foundation
Core services
non-GUI API...
Core OS ("Darwin")
JVM
System utilities
Kernel ("xnu")
File systems
Networking
NKE
POSIX
BSD
I/O Kit
Drivers
Mach
Hardware

The Linux Kernel

Manish Shrivastava

Monolithic?

- What is a monolithic program?
- What is a Monolithic Kernel?

Process?

- What is a process?
- Is kernel a process?

Process?

- What is a process?
- Is kernel a process?
- No!
- Init is just the first process; it does not manage any processes or threads. It does create some, using the kernel syscalls fork() and exec.
- It is a Process Manager without being a process

Kernel vs. Application Coding

Two major differences:

- The core kernel has no standard libraries
- The core kernel must be a monolithic, statically linked library

No standard libraries:

- No libc (malloc, pthreads, string handling, etc.)
- Partly because of chicken/egg situation
- Also, standard libraries can be too slow

No dynamic loading of shared libraries for the core kernel

Two Big Problems

1. A totally static kernel would be enormous
   - About 20 million lines of code in 2015
   - Most of this is hardware drivers that are never used on any given platform

2. Programmers rely on “library” functions for efficiency and correctness
   - E.g., things like data structures, sleeping routines, etc.
   - What libraries to use? (hint: not glibc)
     - Kernel code has to be entirely self-contained
     - Also, chicken+egg problem, as glibc functions may in turn invoke the kernel via system calls

First Solution:
Loadable Kernel Modules

Kernel modules are kernel code that can be loaded dynamically:

- Can be loaded/unloaded whenever
- Runs in kernel mode
- Can access *exported* kernel variables and functions
- Can export variables and functions to kernel or other modules

Kernel and Kernel Modules

- Often, if you want to add something to the kernel you need to rebuild the kernel and reboot.
  - A "loadable kernel module" (LKM) is an object file that extends the base kernel.
  - Exist in most OSes
    - Including Windows, FreeBSD, Mac OS X, etc.
  - Modules get added and removed as needed
    - To save memory, add functionality, etc.

Why Kernel Modules

Linux is a monolithic kernel. All functionality is compiled into the same static binary that is loaded into memory on boot.

Without modules, ***the entire kernel*** would need to be loaded into memory to boot a node. Problems?

- ➢Waste of memory (embedded systems)
- ➢Slower boot time
- ➢Larger trusted computing base (TCB), more room for bugs

What are Kernel Modules used for?

- What pieces of code do we think might not be needed on every system that the kernel boots on?
  1. Device Drivers
  2. Architecture-specific code

- Lines of code (just `.c` files) for:
  1. Device Drivers: `1,583,159`
  2. Everything else: `1,003,777`
     - (Includes all of the architectures that we're not using!)

What are Kernel Modules used for?

- Beyond space savings, what else might modules be useful for?
  1. “Out-of-tree” functionality that is not accepted into “mainline” kernel
  2. Allowing users to load custom functionality at runtime
  3. Configuring/patching a running system without requiring a reboot

Modular?

- What does it mean to be modular (exactly)?
- How is that beneficial?
- How would a Modular kernel differ from a Monolithic one?

Linux Kernel Modules

- In general must be licensed under a free license.
  - Doing otherwise will taint the whole kernel.
    - A tainted kernel sees little support.
    - Might be a copyright problem if you redistribute.
- The Linux kernel changes pretty rapidly, including APIs etc.
  - This can make it a real chore to keep LKMs up to date.
  - Also makes a tutorial a bit of a pain.

Module Implementation

Must define:

- An initialization function called on load
- An exit function called on unload

The init function must be self contained!

- Must unwind actions if initialization cannot complete successfully
- E.g. if you `kmalloc()` space but don't free it, that physical memory is now lost (until system restart)

You can also pass parameters to modules at load time.

Creating a module

- All modules need to define functions that are to be run when:
  - The module is loaded into the kernel
  - The module is removed from the kernel
- We just write C code (see next slide)
- We need to compile it as a kernel module.
  - We invoke the kernel's makefile.
  - `sudo make –C /lib/modules/xxx/build M=$PWD modules`
    - This makes (as root) using the makefile in the path specified.
    - I think it makes all C files in the directory you started in
    - Creates .ko (rather than .o) file
    - Xxx is some kernel version/directory

Simple module

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

MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void) {
    printk("<1> Hello World!\n");
    return 0;
}

static void hello_exit(void) {
    printk("<1> Bye world!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

- `MODULE_LICENSE`
  - Required.
  - Short list of allowed licenses.
- Printk()
  - Kernel print.
    - Prints message to console and to log.
    - <1> indicates high priority message, so it gets logged.
- Module_init()
  - Tells system what module to call when we first load the module.
  - TIMTOWTDI
- Module_exit()
  - Same but called when module released.

Modules:
Listing, loading and removing

- From the command line:
  - lsmod
    - List modules.
  - insmod
    - Insert module into kernel
      - Adds to list of available modules
    - Causes function specified by module_init() to be called.
  - rmmod
    - Removes module from kernel

lsmod

```
Module                          Size  Used by
memory                         10888  0
hello                           9600  0
binfmt_misc                    18572  1
bridge                         63776  0
stp                            11140  1 bridge
bnep                           22912  2
video                          29844  0
```

insmod

- Very (very) simple
  - **`insmod xxxxx.ko`**
    - Says to insert the module into the kernel

Other (better) way to load a module

- **`Modprobe`** is a smarter version of insmod.
    - Actually it's a smarter version of insmod, lsmod and rmmod…
        - It can use short names/aliases for modules
        - It will first install any dependent modules
- We'll use insmod for the most part
    - But be aware of modprobe

[3]

So?

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

MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void) {
    printk("<1> Hello World!\n");
    return 0;
}

static void hello_exit(void) {
    printk("<1> Bye world!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

- When insmod, log file gets a “Hello World!”
- When rmmod, that message prints to log (and console…)
- It’s not the name, it’s the module_init().

Modules?

- There are a number of different reasons one might have a module
  - But the main one is to create a device driver
  - It’s not realistic for Linux to have a device driver for all possible hardware in memory all at once.
    - Would be too much code, requiring too much memory.
  - So we have devices as modules
    - Loaded as needed.

Device driver
(Thanks Wikipedia!)

- A device driver is a computer program allowing higher-level computer programs to interact with a hardware device.
    - A driver typically communicates with the device through the computer bus or communications subsystem to which the hardware connects.
    - When a calling program invokes a routine in the driver, the driver issues commands to the device.
    - Drivers are hardware-dependent and operating-system-specific.

Devices in Linux (1/2)

- There are special files called “device files” in Linux.
  - A user can interact with it much like a normal file.
  - But they *generally* provide access to a physical device.
  - They are generally found in /dev and /sys
    - /dev/fb is the frame buffer
    - /dev/ttyS0 is one of the serial ports

```
crw-rw---- 1 root dialout   4,  64 Jun 20 13:01 ttyS0
```

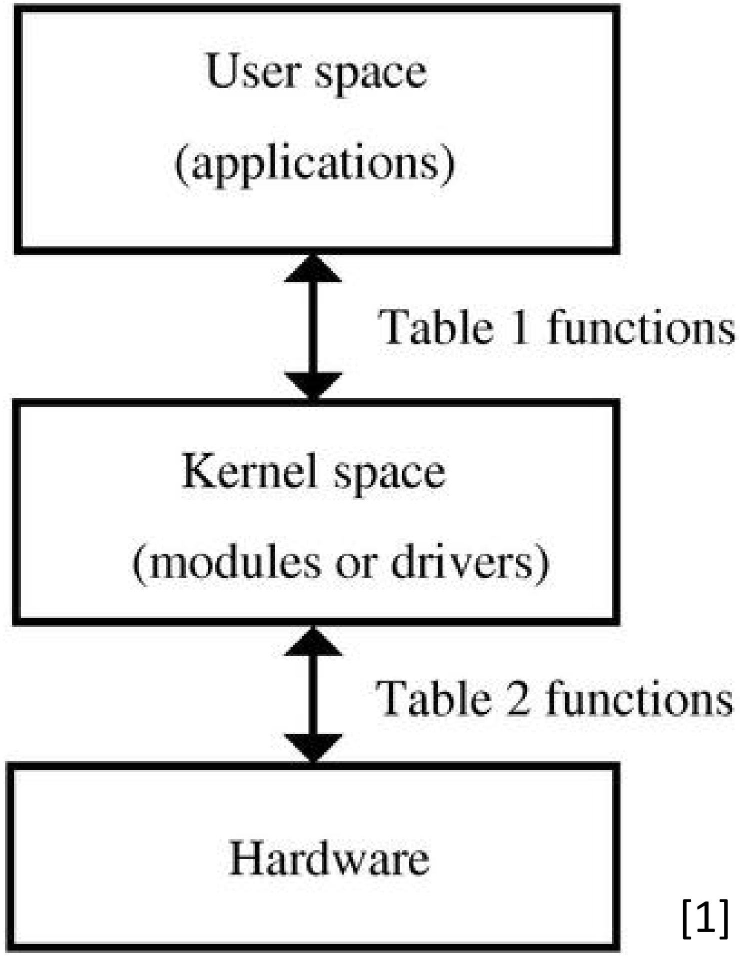
- Not all devices files correspond to physical devices.
  - Pseudo-devices.
    - Provide various functions to the programmer
    - /dev/null
      - Accepts and discards all input; produces no output.
    - /dev/zero
      - Produces a continuous stream of NULL (zero value) bytes.
    - etc.

Devices in Linux (2/2)

- Pretty clearly you need a way to connect the device file to the actual device
  - Or pseudo device for that matter
- We want to be able to “fake” this by writing functions that handle the file I/O.
  - So we need to associate functions with all the things we can do with a file.
    - Open, close.
    - Read, write.
- Today we’ll talk about all that…

Kernel vs. User space

- User Space
  - End-user programs. They use the kernel to interface to the hardware.
- Kernel Space
  - Provides a standard (and hopefully multi-user secure) method of using and sharing the hardware.
    - Private function member might be a good analogy.



[1]

What is a “device”?

- Linux devices are accessed from user space in exactly the same way files are accessed.
  - They are generally found in /dev and /sys
- To link normal files with a kernel module, each device has a “major number”
  - Each device also has a “minor number” which can be used by the device to distinguish what job it is doing.

```
% ls -l /dev/fd0 /dev/fd0u1680
brwxrwxrwx   1 root  floppy   2,  0 Jul  5  2000 /dev/fd0
brw-rw----   1 root  floppy   2, 44 Jul  5  2000
/dev/fd0u1680
```

Two floppy devices. They are actually both the same bit of hardware using the same driver (major number is 2), but one is 1.68MB the other 1.44.

Creating a device

- **`mknod /dev/memory c 60 0`**
  - Creates a device named /dev/memory
  - Major number 60
  - Minor number 0
- Minor numbers are passed to the driver to distinguish different hardware with the same driver.
  - Or, potentially, the same hardware with different parameters (as the floppy example)

Second Solution: Kernel “Libraries”

Kernel libraries re-implement a lot of the functionality programmers expect in user space

- Are statically compiled into the kernel
- Automatically available just by including relevant header
- Built to be kernel-safe (sleeping, waiting, locking, etc. is done properly)

Features:

- Utilities: kmalloc, kthreads, string parsing, etc.
- Containers: hash tables, binary trees etc.
- Algorithms: sorting, compression

Mostly found under `/lib` and `/include/linux`

Kernel “Libraries”

Mostly found under `/lib` and `/include/linux`

Many kernel “libraries” have clear analogues to user space libraries:

- malloc/free vs kmalloc/kfree
- pthread_create vs kthread_create
- sleep/usleep/nanosleep (user) vs msleep (kernel)

Some, however, are a bit different:

- e.g., linked list implementation

Example: Linked Lists

See also:

- /include/linux/list.h
- /include/linux/types.h

Moral of the story:

Always search for functionality before writing it yourself.

Notes:

- One important note is that module stuff is written in kernel space.
  - That means you can’t do a lot of things you might want to!
    - File I/O is a really bad idea
      - See next slide.
    - Talking to memory-mapped I/O devices requires effort
      - Still have virtual memory
    - Things like malloc don’t quite work
      - Thus kmalloc, kprint, etc.
  - Can be an unpleasant place to live…