Operating System Structure

Heechul Yun

Disclaimer: some slides are adopted from the book authors' slides with permission

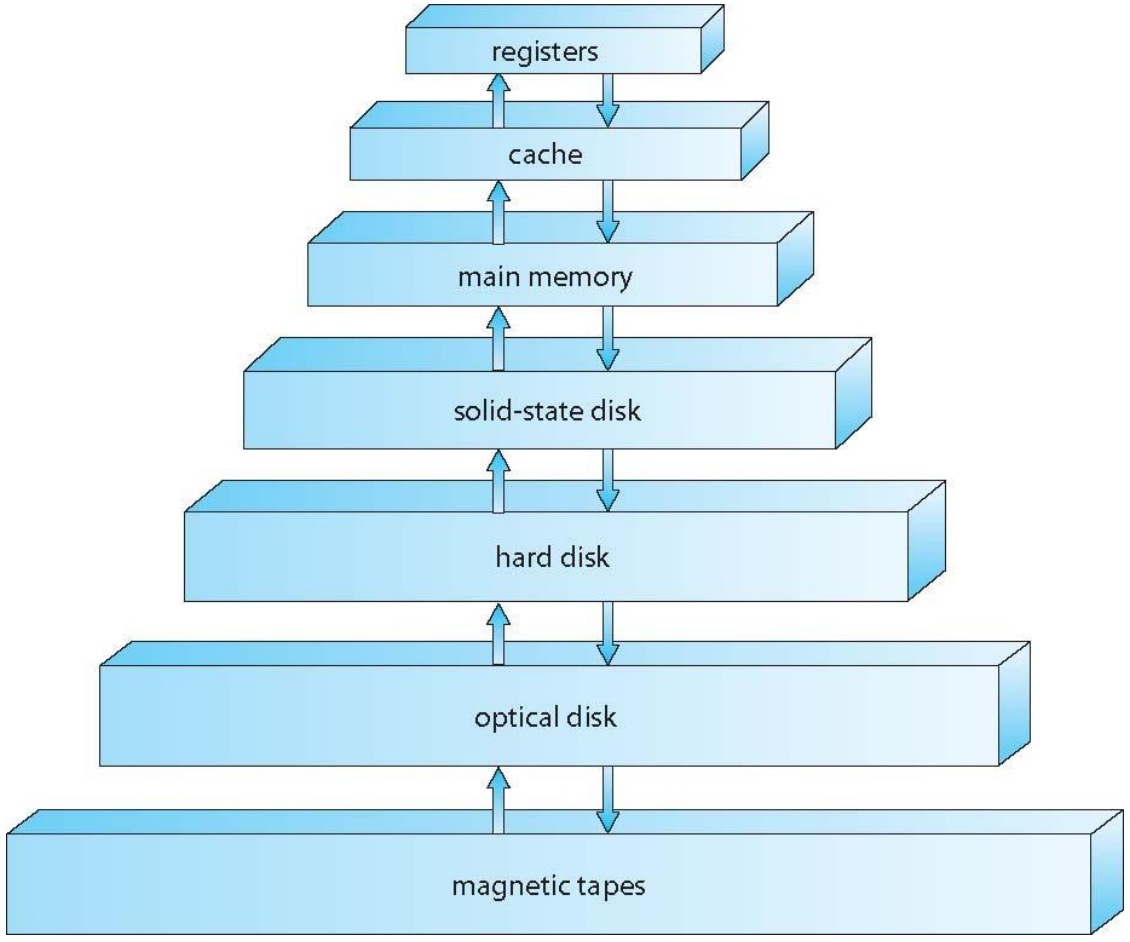
KU
THE UNIVERSITY OF
KANSAS

Recap

- OS needs to understand architecture
  - Hardware (CPU, memory, disk) trends and their implications in OS designs
- Architecture needs to support OS
  - Interrupts and timer
  - User/kernel mode and privileged instructions
  - MMU
  - Synchronization instructions

Recap: Memory Hierarchy

Fast,
Expensive



Slow,
Inexpensive

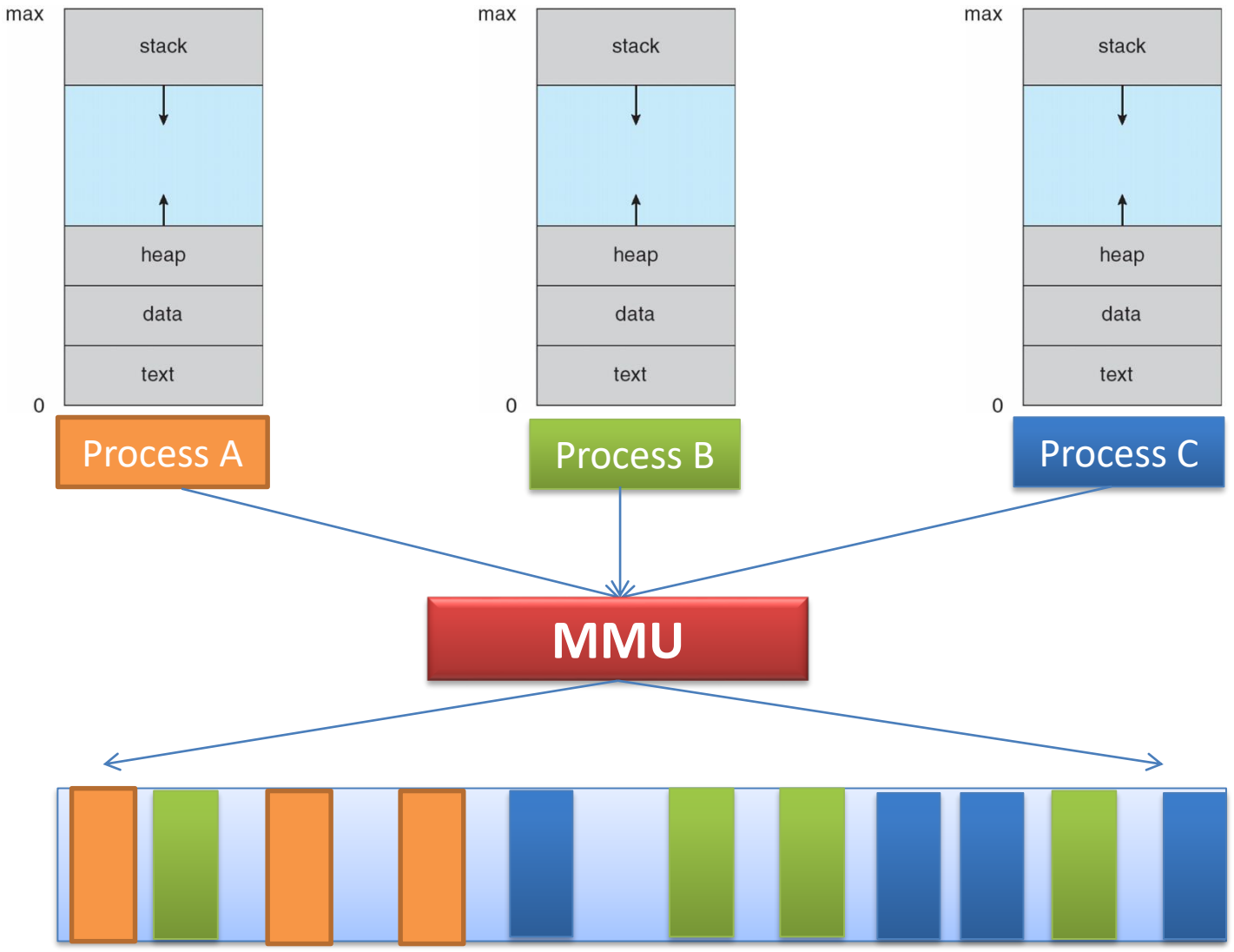
Memory Protection

- How to protect memory among apps/kernel?
  - Applications shouldn't be allowed to access kernel's memory
  - An app shouldn't be able to access another app's memory

Virtual Memory

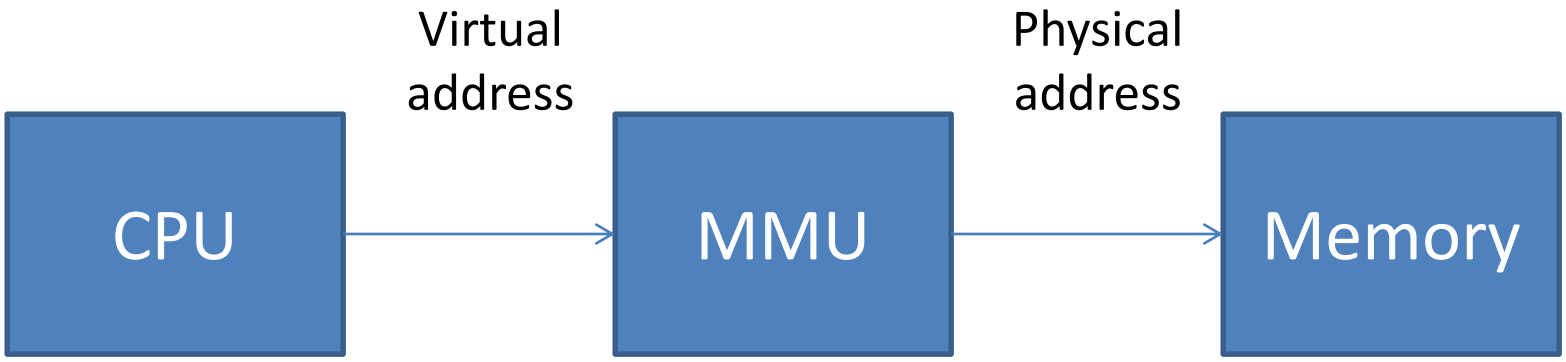
- How to overcome memory space limitation?
  - Multiple apps must share limited memory space
  - But they want to use memory as if each has dedicated and big memory space
  - E.g.,) 1GB physical memory and 10 programs, each of which wants to have a linear 4GB address space

Virtual Memory



MMU

- Hardware unit that translates *virtual address* to *physical address*
  - Defines the boundaries of kernel/apps
  - Enable efficient use of physical memory



Synchronization

- Synchronization problem with threads

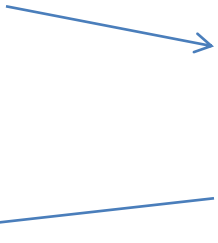
```
Deposit(account, amount) {
{
   account->balance += amount;
}
```

Thread 1: *Deposiit(acc, 10)*

```
LOAD R1, account->balance


ADD R1, amount
STORE R1, account->balance
```

Thread 2: : *Deposiit(acc, 10)*

```
LOAD R1, account->balance
ADD R1, amount
STORE R1, account->balance
```



Synchronization Instructions

- Hardware support for synchronization
  - *TestAndSet*, *CompareAndSwap* instructions
  - Atomic load and store
  - Used to implement **lock** primitives
  - New TSX instruction → hardware transaction

- Another methods to implement locks in single-core systems
  - Disabling interrupts

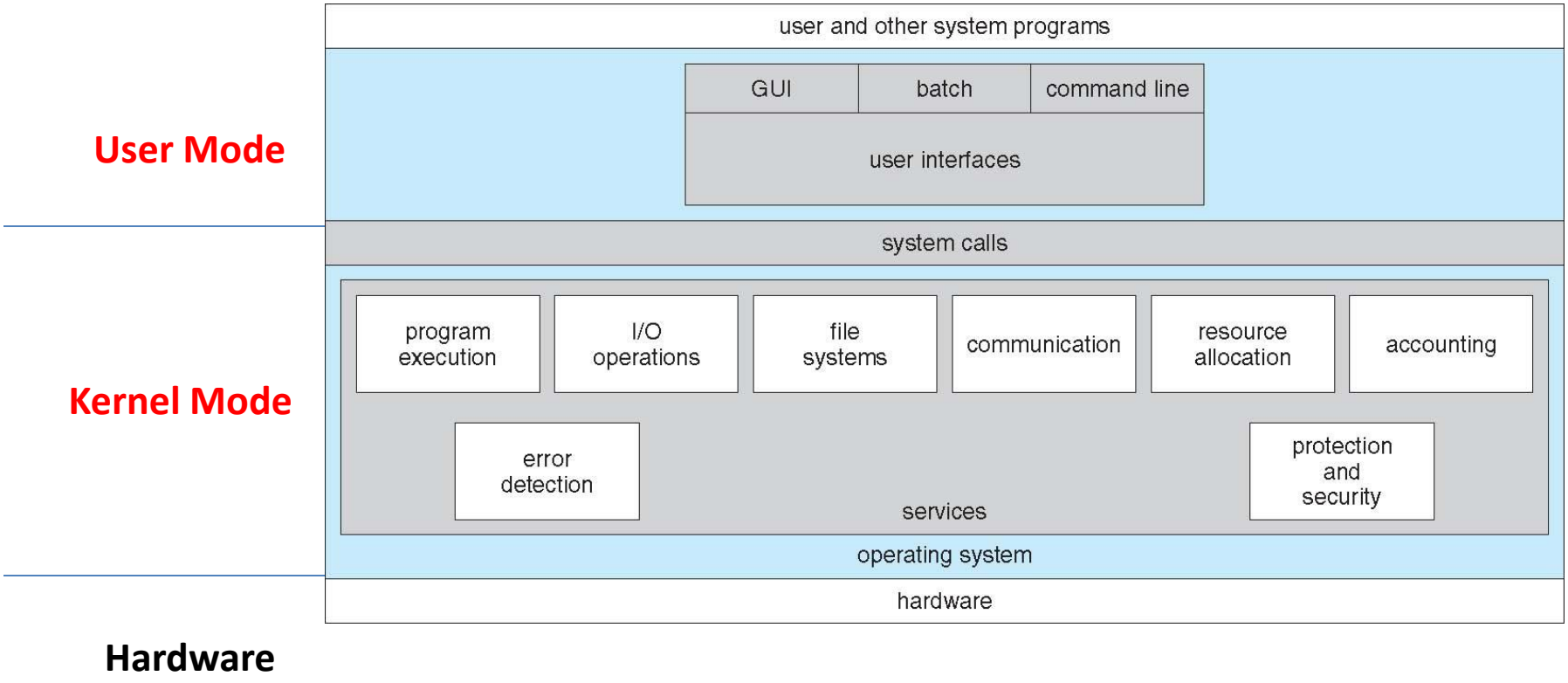
Summary

- OS needs to understand architecture
  - Hardware (CPU, memory, disk) trends and their implications in OS designs
- Architecture needs to support OS
  - Interrupts and timer
  - User/kernel mode and privileged instructions
  - MMU
  - Synchronization instructions

Today

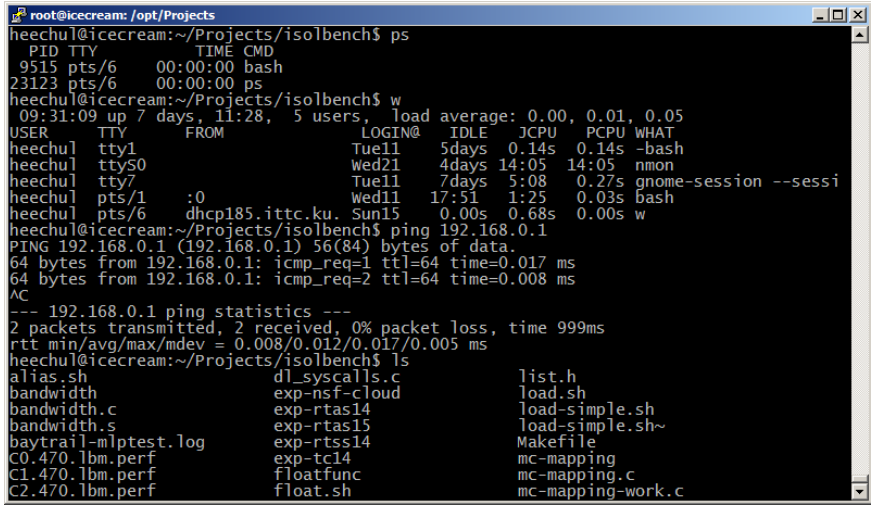
- OS services
  - User's perspective
  - What are the major functionalities of an OS?
- OS interface
  - How applications interact with the OS?
- OS structure
  - What are possible structures of an OS?

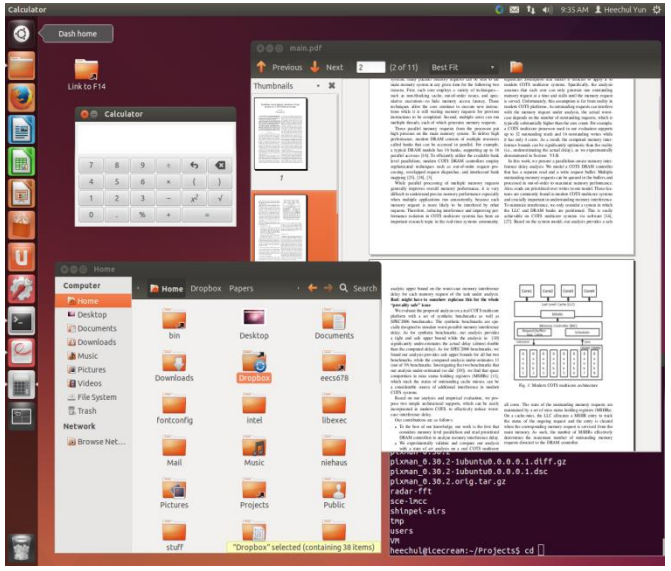
A View of Operating System Services



Operating System Services

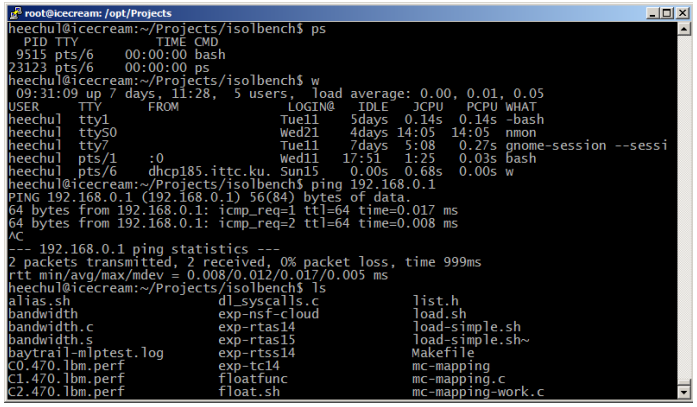
- User interface
  - Command-Line Interface (CLI) vs. Graphical User Interface (GUI)





Command-Line Interface (CLI)

- Command-line interpreter (shell)
  - Many flavors: bash, csh, ksh, tcsh, …
  - Usually not part of the kernel, but an essential system program
- Allow users to enter text commands
  - Some commands are built-in
    - E.g., alias, echo, read, source, …
  - Some are external programs
    - E.g., ls, find, grep, …



- Pros and Cons.

  \+ Easy to implement, use less resources, easy to access remotely

  \+ Easy to **automate**
    - E.g., $ grep bandwidth /tmp/test.txt | awk '{ print $2 }'

  – Difficult to learn

Graphic User Interface (GUI)

- GUI
  - Mouse, keyboard, monitor
  - Invented at Xerox PARC, then adopted to Mac, Window,…

The first commercial GUI from Xerox Star workstation. (source: Wikipedia)

- Pros and Cons
  + Easy to use
  - Use more h/w resources

- Many systems support both CLI and GUI

Operating System Services

- File-system service
  - Read/write /create/delete/search files and directories
  - See file information (e.g., file size, creation time, access time, …)
  - Permission management (read only, read/write, …)

- Communications
  - Share information between processes in the same computer (Inter-process communication - IPC) or between computers over a network (TCP/IP)

Operating System Services

- Resource allocation
  - CPU cycles, main memory space, file space, I/O devices
- Accounting
  - Keeping track of who uses what for how much
- Security
  - Login, administrators vs. normal users vs. guests

Operating System Services

- Protection
  - Prevent memory corruption between multiple user programs and between user programs and the kernel
  - Detect and report errors
    - Divide by zero, access violation, hardware faults, …

Recap

- OS services
  - CPU scheduling
  - Memory management
  - Filesystem
  - Communication
  - Protection & security
  - Device drivers

Exam

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C)
- Most programmers do not directly use system calls
  - They use more high level APIs (i.e., libraries)
  - But **system programmers** (or you) do use system calls
- Two most popular system call APIs
  - Win32 API for Windows
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)

Example

- Copy the contents of one file to another file

```
int main(int argc, char *argv[])
{
        int src_fd, dst_fd; char buf[80]; int len;

        src_fd = open(argv[1], O_RDONLY);
        dst_fd = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC);

        while ((len = read(src_fd, buf, 80)) > 0) {
                write(dst_fd, buf, len);
        }

        printf("Done\n");
        return 0;
}
```

Example

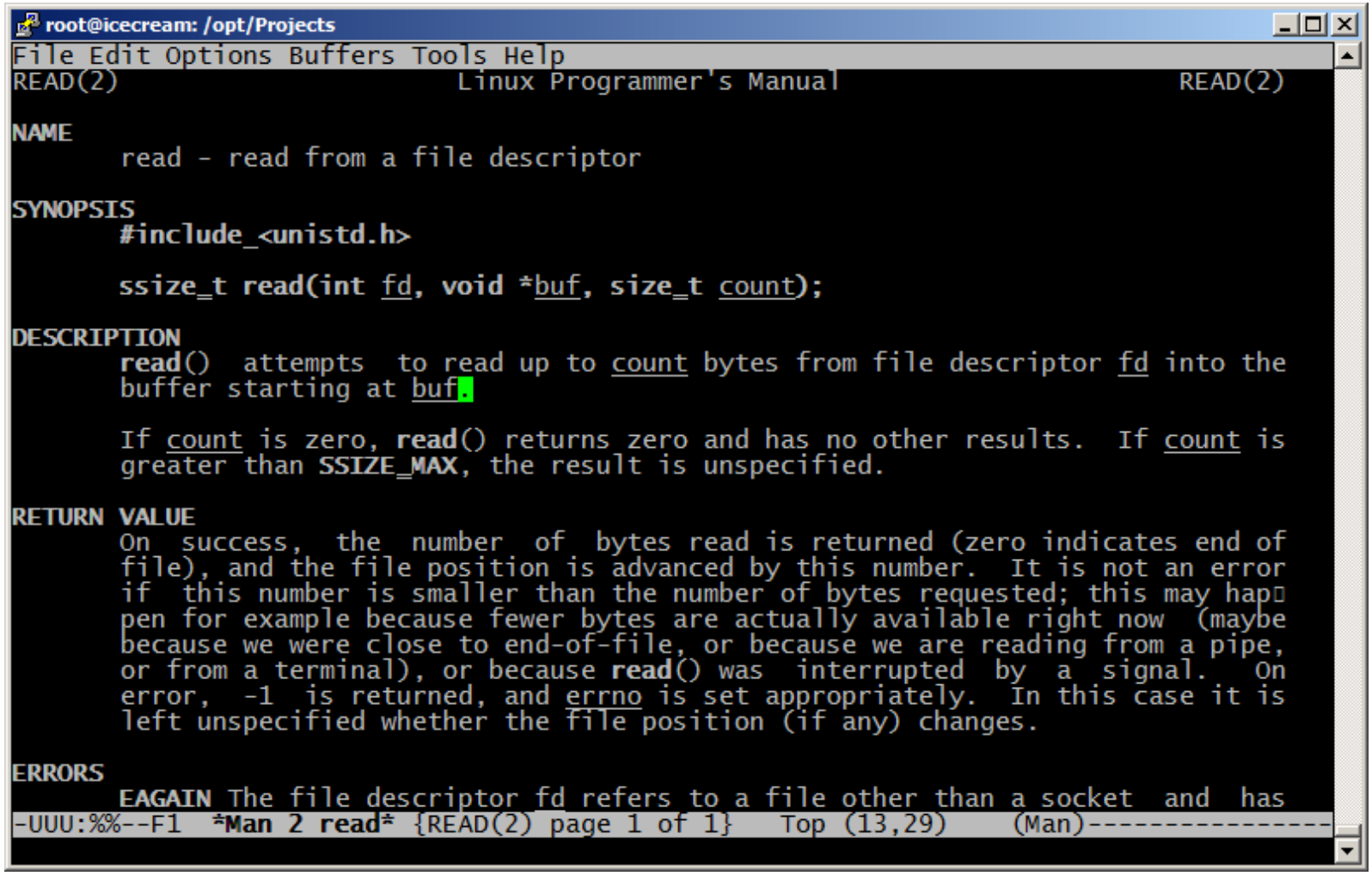
- Copy the contents of one file to another file

```
int main(int argc, char *argv[])
{
        int src_fd, dst_fd; char buf[80]; int len;

        src_fd = open(argv[1], O_RDONLY);
        dst_fd = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC);

        while ((len = read(src_fd, buf, 80)) > 0) {
                write(dst_fd, buf, len);
        }

        printf("Done\n");
        return 0;
}
```

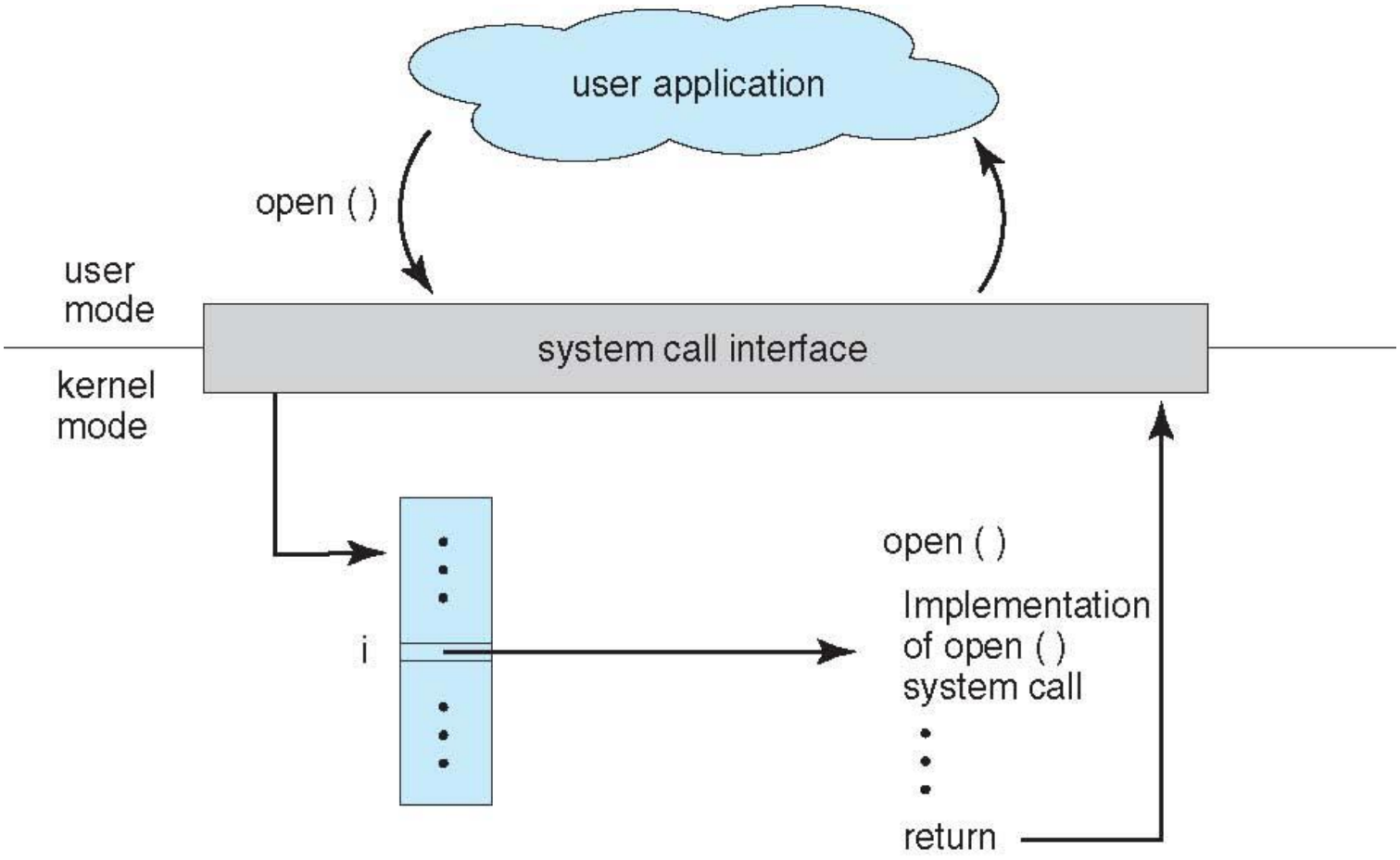
Syscalls: open, read, wrtie
Non-syscall: printf

System Call API Description

- $ man 2 read

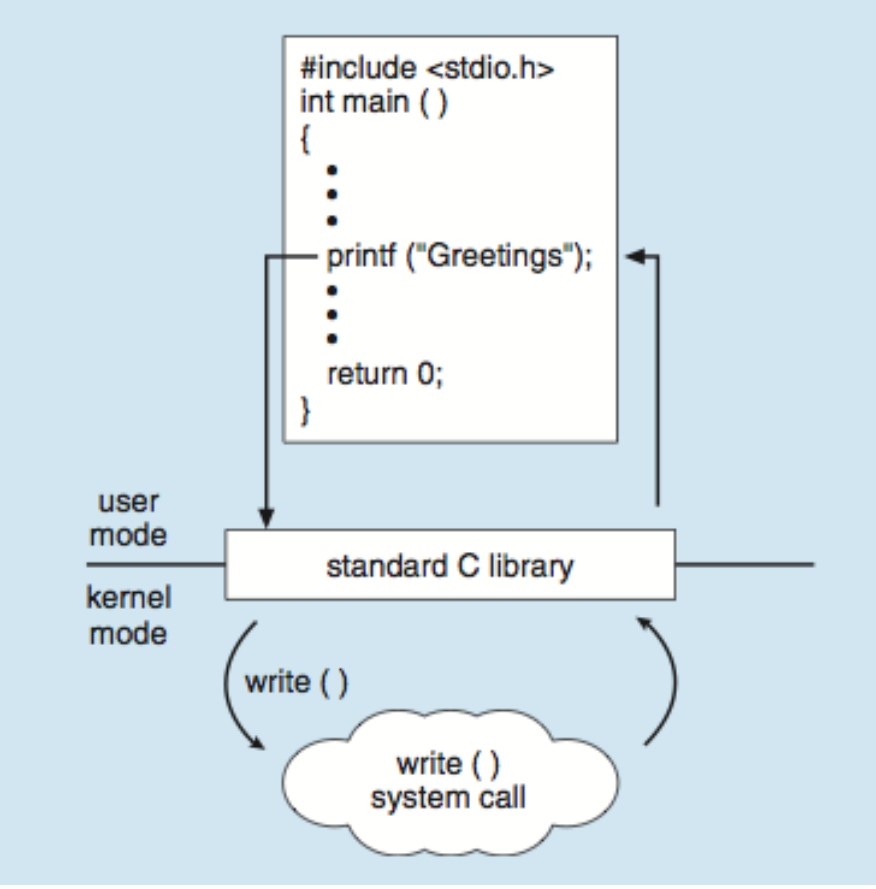


API - System Call - OS



Standard C Library Example

- C program invoking printf() library call, which calls write() system call



Types of System Calls

- Process control
  - Create/terminate process, get/set process attributes, wait for time/event, allocate and free memory
- File management
  - create, delete, open, close, read, write, reposition
  - get and set file attributes
- Device management
  - request device, release device, read, write, reposition, get device attributes, set device attributes
- Communications
  - create, delete communication, send, receive messages
- Protection
  - Control access to resources, get/set permissions, allow and deny user access

Examples of Windows and Unix System Calls

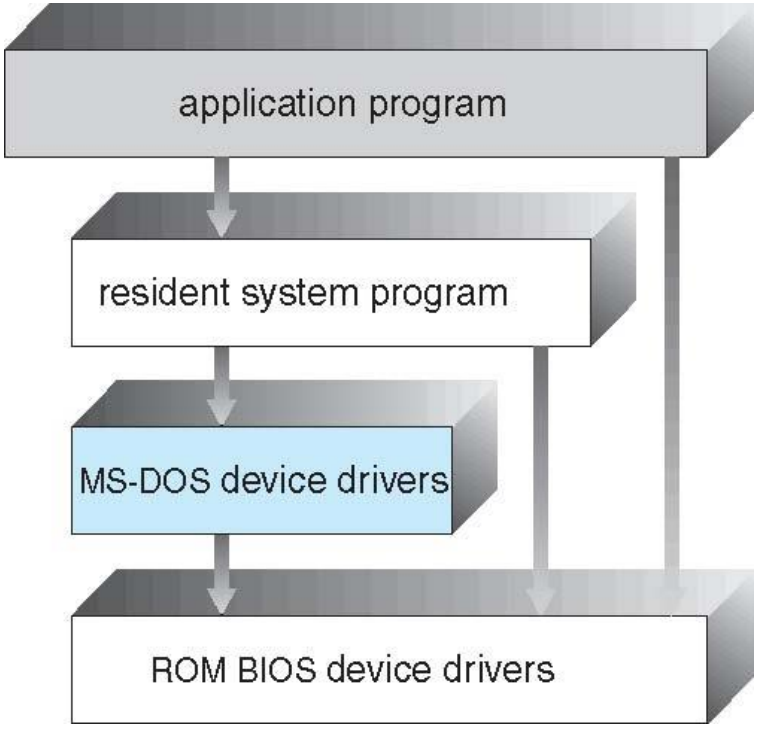
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Operating System Structure

- Simple structure – MS-DOS
- **Monolithic kernel – UNIX**
- Microkernel – Mach

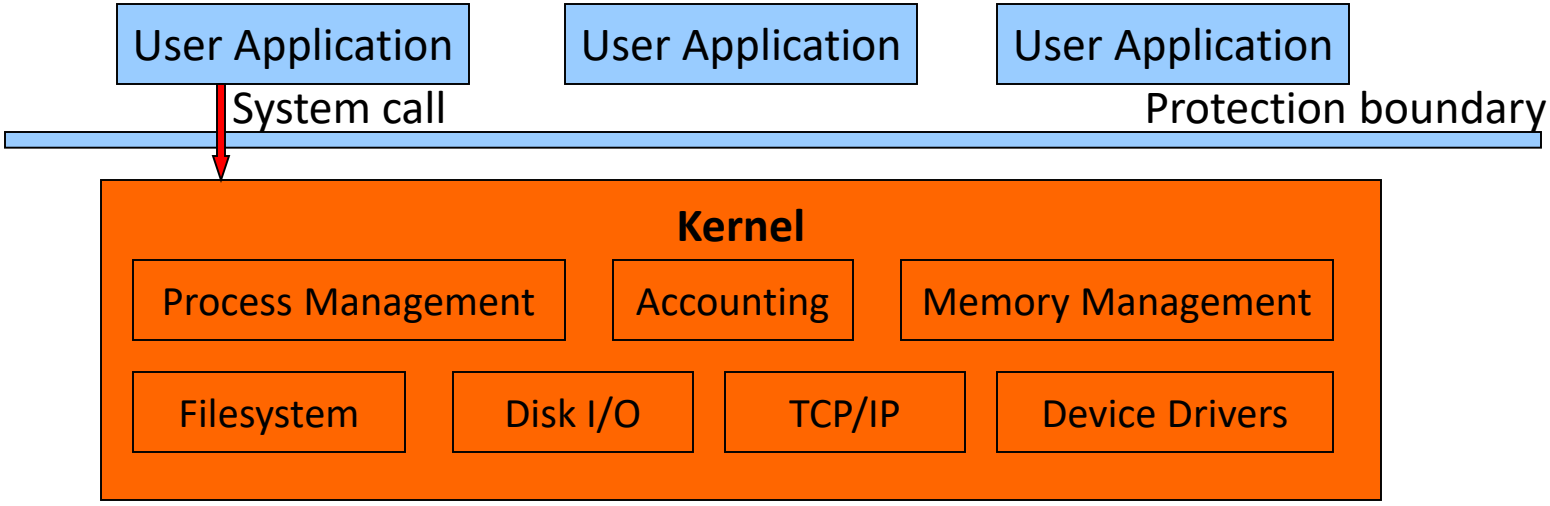
MS-DOS Structure

- Written to provide the most functionality in the **least space**
- Minimal functionalities
- Not divided into modules
- Although MS-DOS has some structure, its interfaces and levels of *functionality are not well separated*



UNIX: Monolithic Kernel

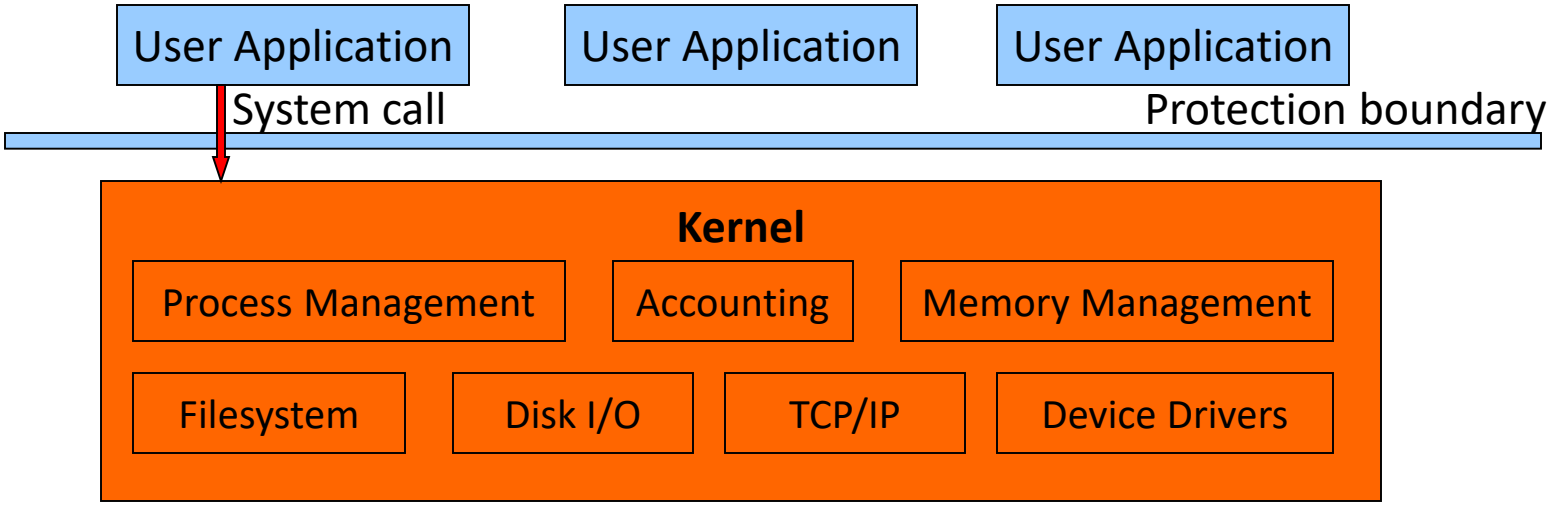
- Implements CPU scheduling, memory management, filesystems, and other OS modules all in a single big chunk



- Pros and Cons
  - + Overhead is low
  - + Data sharing among the modules is easy
  - – Too big. (device drivers!!!)
  - – A bug in one part of the kernel can crash the entire system

Loadable Kernel Module

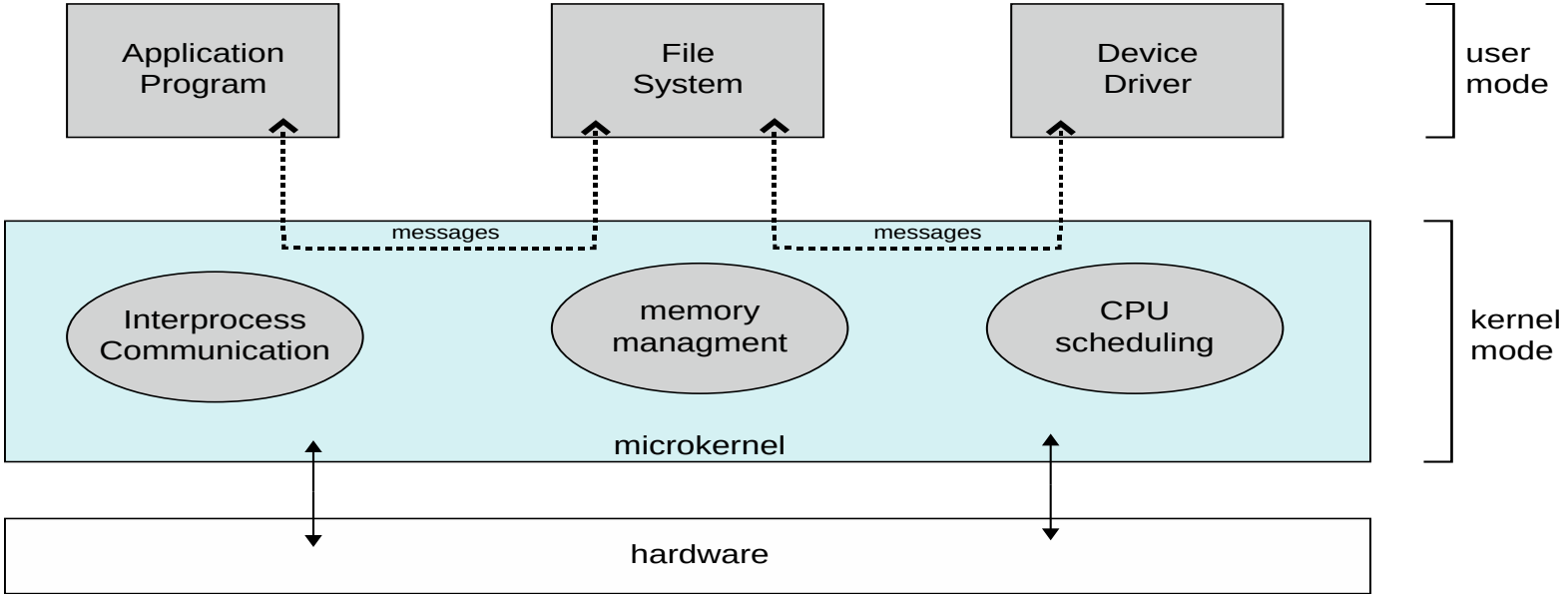
- Dynamically load/unload new kernel code
  - Linux and most today's OSes support this



- Pros and Cons
  - + Don't need to have every driver in the kernel.
  - + Easy to extend the kernel (just like micro kernel. See next)
  - – A bug in a module can crash the entire system

Microkernel

- Moves as much from the kernel into user space
- Communicate among kernels and user via **message passing**



- Pros and Cons

  \+ Easy to extend (user level driver)

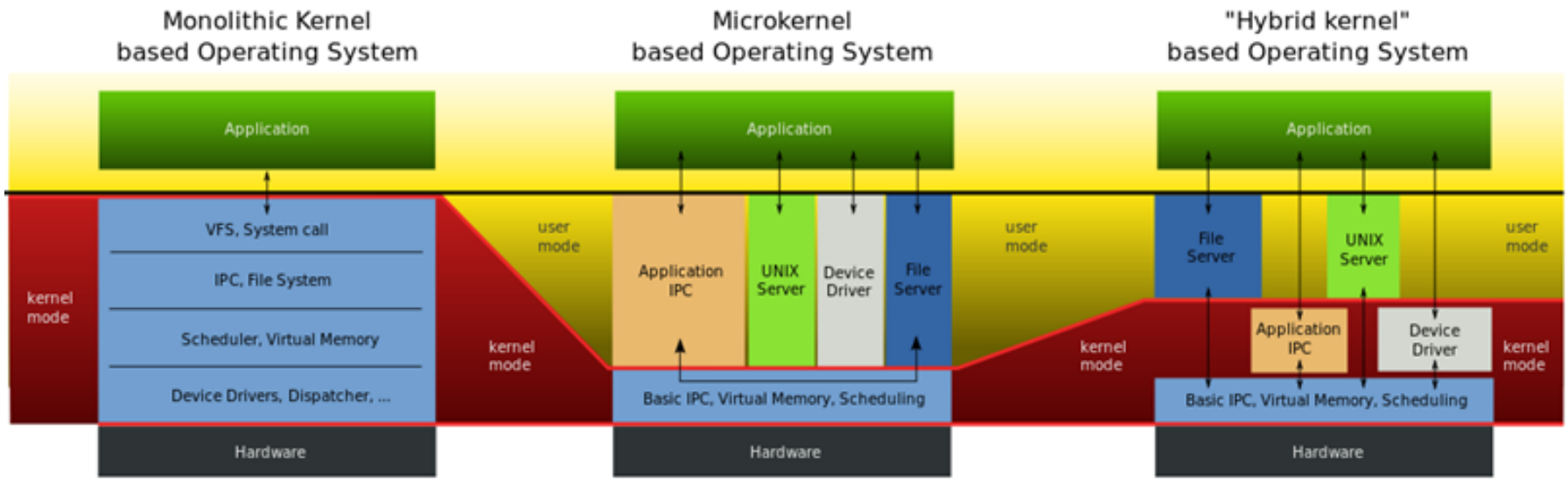
  \+ More reliable (**less code** is running in kernel mode)

  - Performance overhead of user space to kernel space communication

Hybrid Structure

- Most modern operating systems are actually not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
  - Windows mostly monolithic, plus microkernel for different subsystem ***personalities***
- Apple Mac OS X
  - hybrid, layered
  - Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called kernel extensions)

OS Structure Comparison



Source: http://en.wikipedia.org/wiki/Monolithic_kernel