CS161: Operating Systems

Matt Welsh
mdw@eecs.harvard.edu

Lecture 2: OS Structure and System Calls
February 6, 2007

Lecture Overview

Protection Boundaries and Privilege Levels

- What makes the kernel different from regular user programs

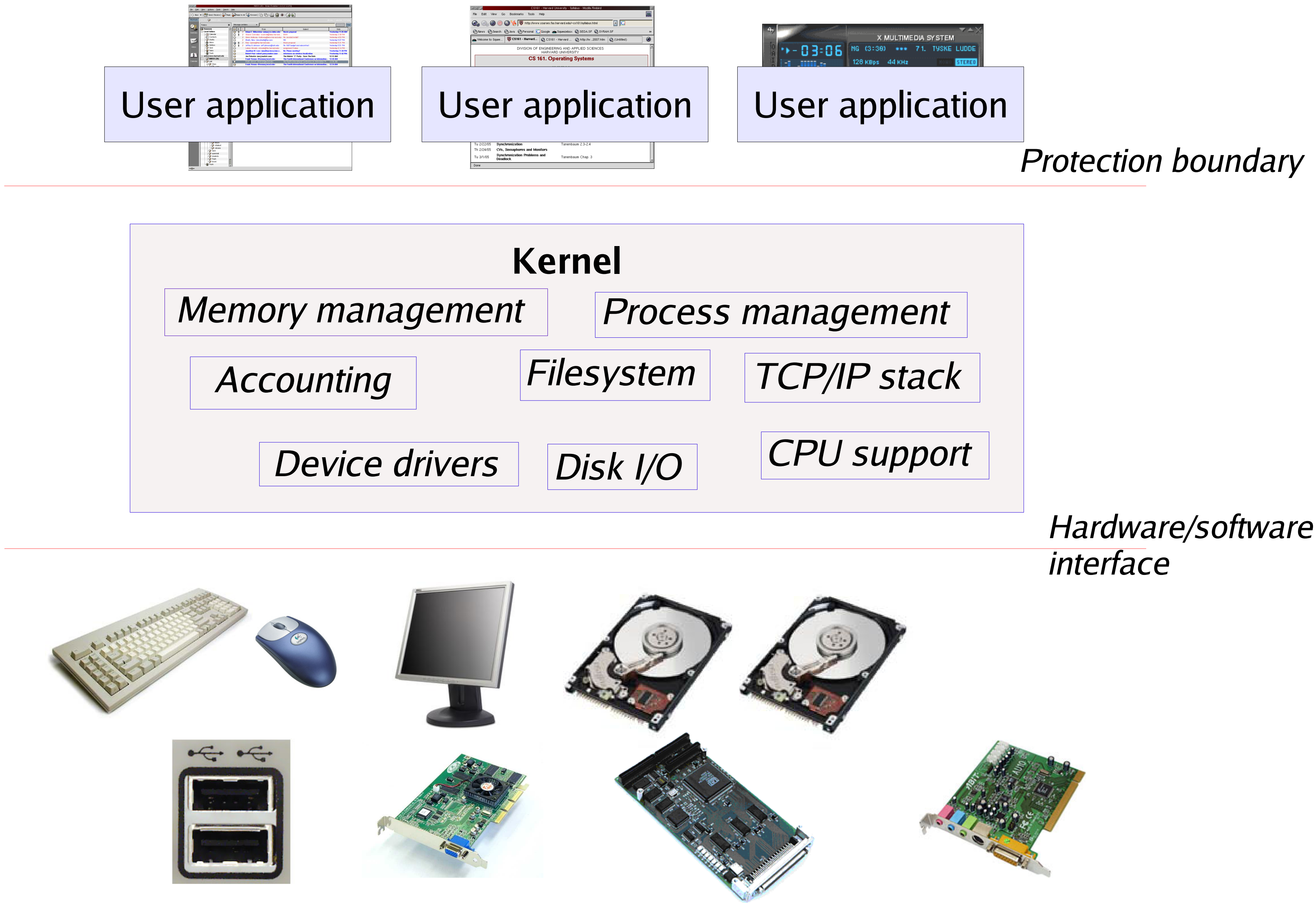
System calls and hardware interrupts

- How user applications and the kernel interact with each other

Operating System Structure

- What are the major functional components of an OS?
- How are those components interrelated?
- What are the interfaces to those components?

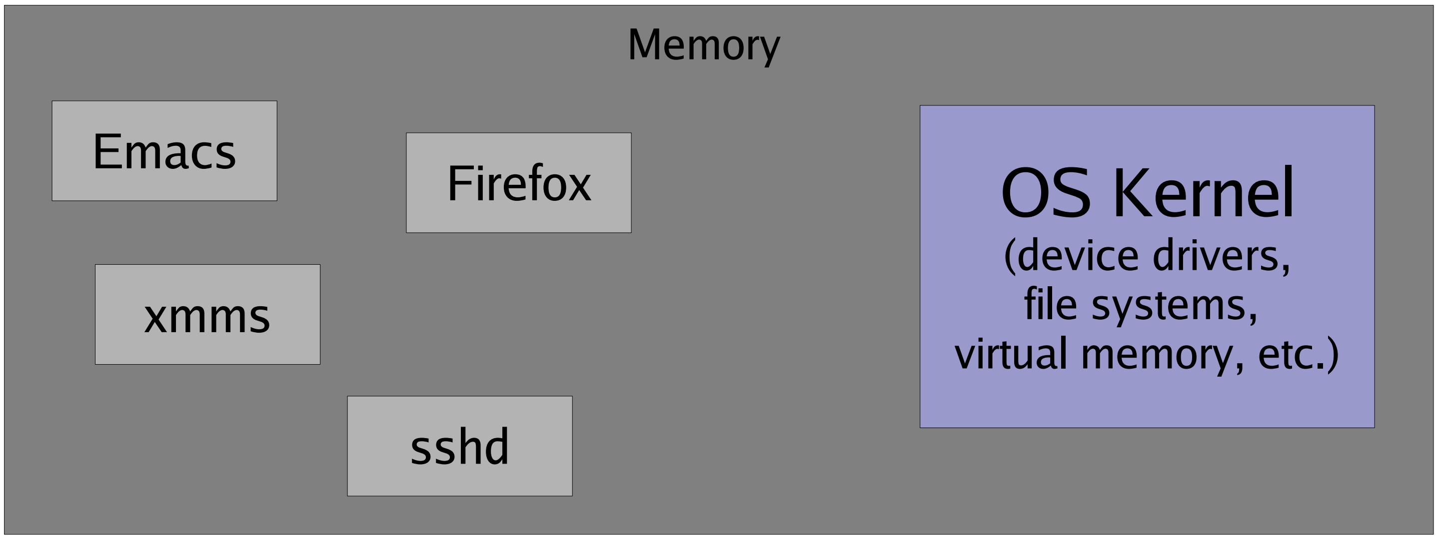
Operating System Overview



Gnarly world of hardware

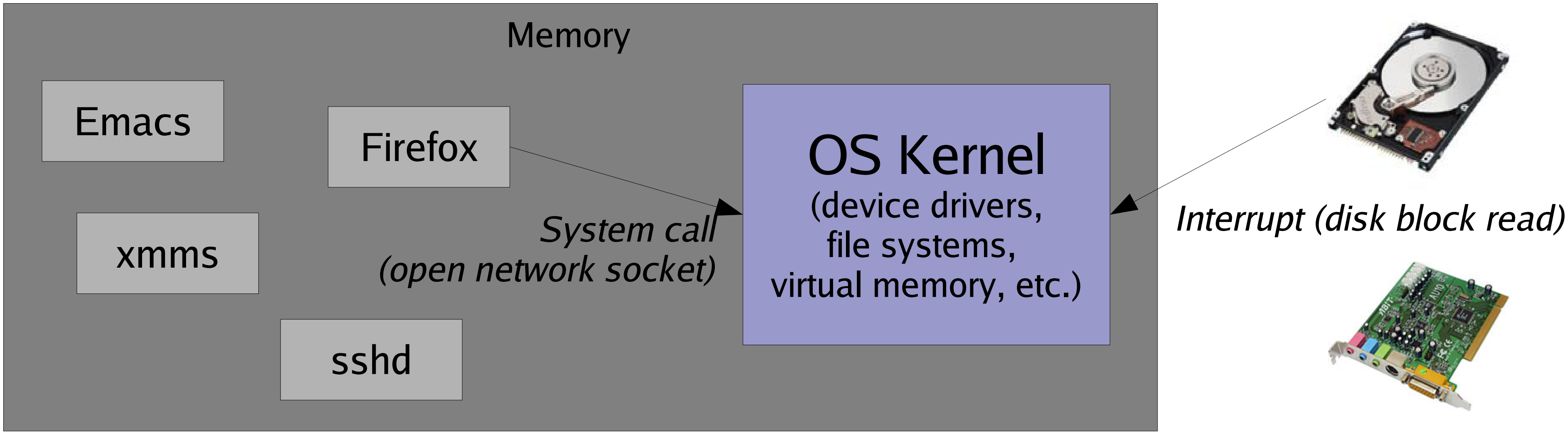
Operating System basics

The OS kernel is just a bunch of code that sits around in memory, waiting to be executed



Operating System basics

The OS kernel is just a bunch of code that sits around in memory, waiting to be executed



OS is triggered in two ways: *system calls* and *hardware interrupts*

System call: Direct “call” from a user program

- For example, open() to open a file, or exec() to run a new program

Hardware interrupt: Trigger from some hardware device

- For example, when a disk block has been read or written

How else might the kernel get control ???

Interrupts – a primer

An *interrupt* is a signal that causes the CPU to jump to a pre-defined instruction – called the *interrupt handler*

- Interrupt can be caused by hardware or software

Hardware interrupt examples

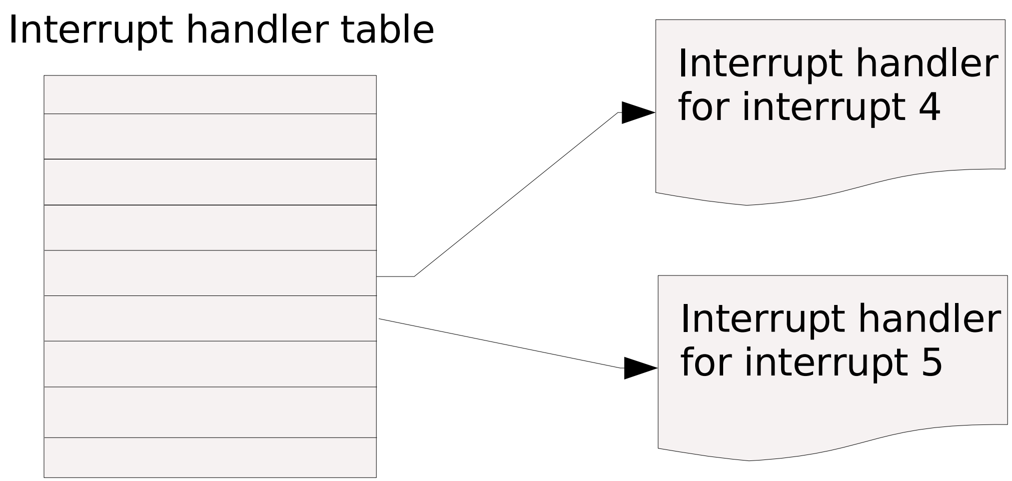
- Timer interrupt (periodic “tick” from a programmable timer)
- Device interrupts
  - *e.g., Disk will interrupt the CPU when an I/O operation has completed*

Software interrupt examples (also called *exceptions*)

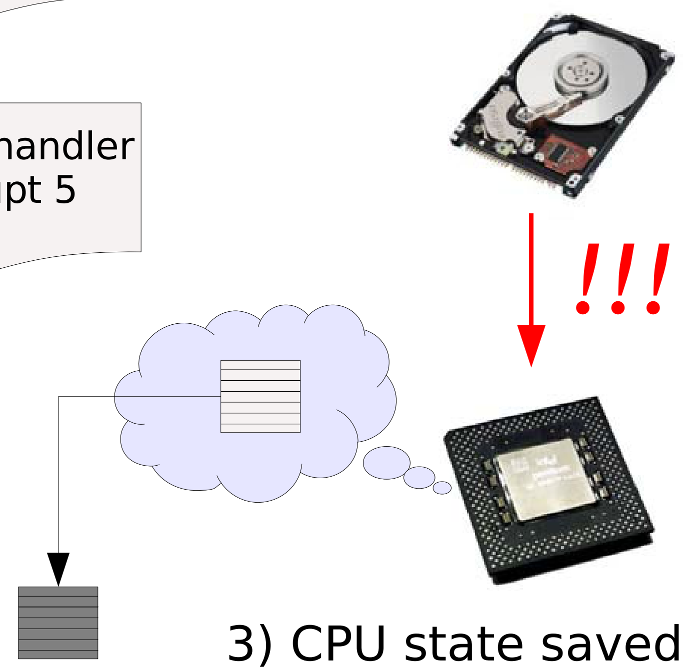
- Division by zero error
- Access to a bad memory address
- Intentional software interrupt – e.g., x86 “INT” instruction
  - *Can be used to trap from user program into the OS kernel!*
  - *Why might this be useful?*

Interrupt handler example

1) OS fills in interrupt handler table (usually at boot time)



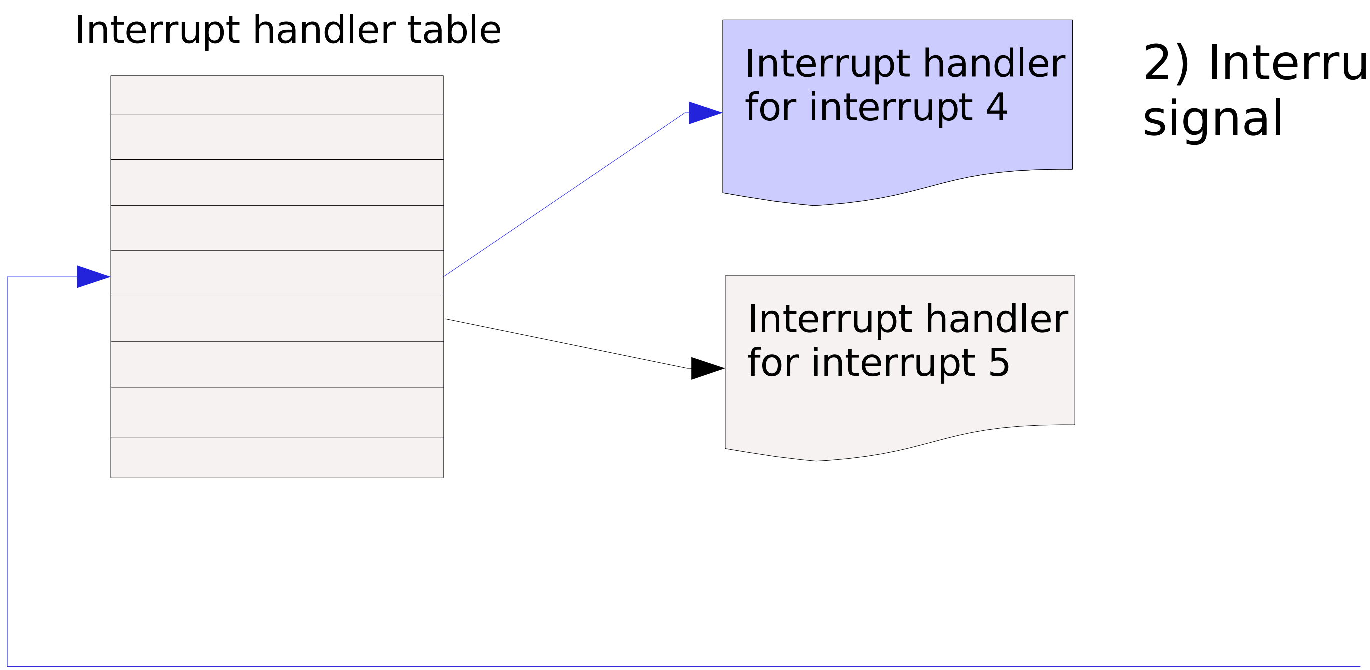
2) Interrupt occurs – e.g., hardware signal



3) CPU state saved to stack

Interrupt handler example

1) OS fills in interrupt handler table (usually at boot time)



2) Interrupt occurs – e.g., hardware signal

3) CPU state saved to stack

4) CPU consults interrupt table and invokes appropriate handler

Protection

A major job of the OS is to enforce *protection*

Prevent malicious or buggy programs from:

- Allocating too many resources (denial of service)
- Corrupting or overwriting shared resources (files, shared memory, etc.)

Prevent different users, groups, etc. from:

- Accessing or modifying private state (files, shared memory, etc.)
- Killing each other's processes

A lot of viruses, worms, etc. exploit security holes in the OS

- Overrunning a memory buffer in the kernel can give a non-root process root privileges
  - *Kernel code needs to be rock solid in order to be secure!!!*

How does the OS enforce protection boundaries?

User mode vs. kernel mode

What makes the kernel different from user programs?

- Kernel can execute special *privileged instructions*

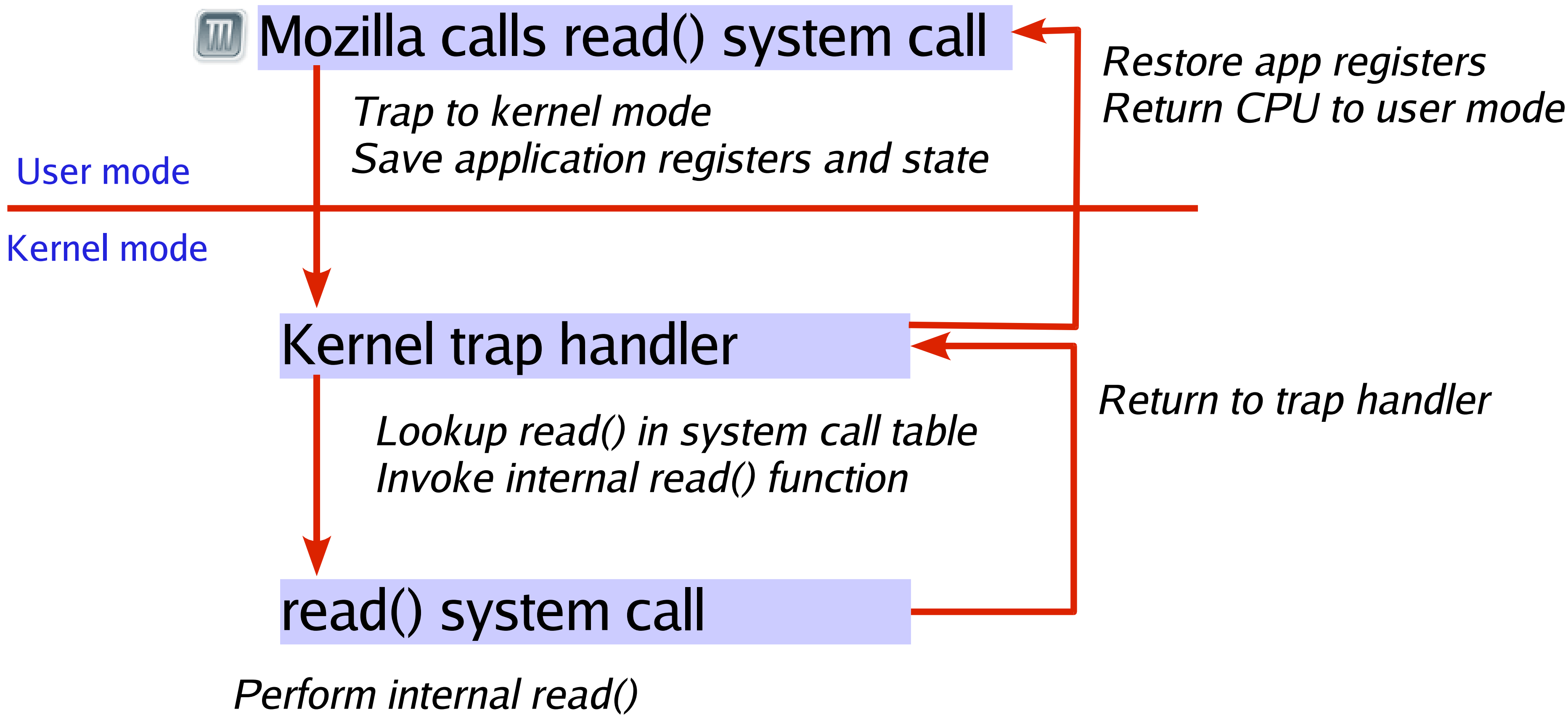
Examples of privileged instructions:

- Access I/O devices
  - *Poll for IO, perform DMA, catch hardware interrupt*
- Manipulate memory management
  - *Set up page tables, load/flush the TLB and CPU caches, etc.*
- Configure various “mode bits”
  - *Interrupt priority level, software trap vectors, etc.*
- Call halt instruction
  - *Put CPU into low-power or idle state until next interrupt*

These are enforced by the **CPU hardware itself**.

- CPU has at least two protection levels: *Kernel mode* and *user mode*
- CPU checks current protection level on each instruction
- What happens if user program tries to execute a privileged instruction?

Boundary Crossing



Protection Rings

On most CPU designs, there are just two protection levels:

- Kernel mode and user mode

Some CPUs, however, support multiple protection levels

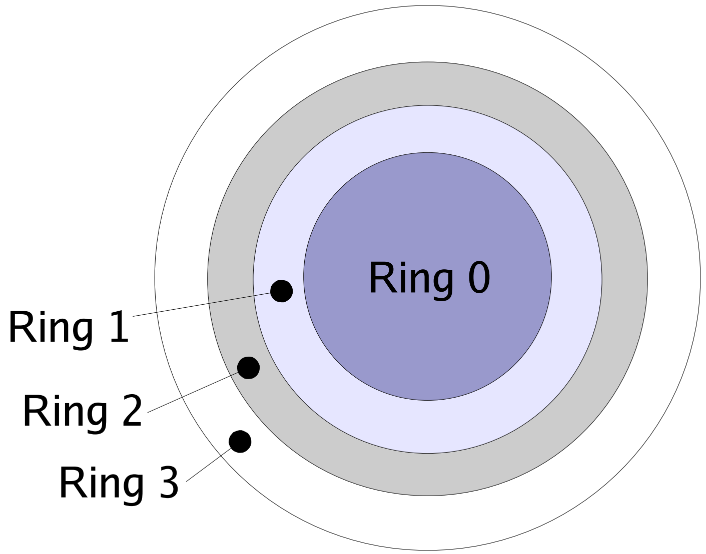
The Intel x86 family has four protection “rings”

Ring 0 (innermost ring) has the OS kernel

Ring 3 has application code

Rings 1 and 2 may be used for less-privileged OS code

- *e.g., Third-party drivers*



Intel x86 Protection Ring Rules

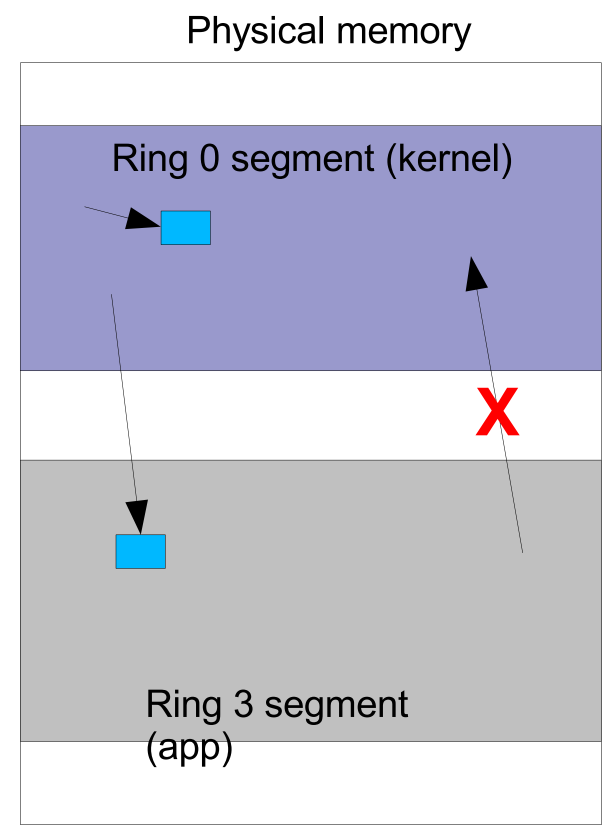
Each memory segment has an associated privilege level, 0 through 3

The CPU has a Current Protection Level (CPL)

- Usually the privilege level of the segment where the program's instructions are being read from

Program can read/write data in segments of *lower privilege* than CPL.

- e.g., Kernel can read/write user memory
- But, user cannot read/write kernel memory
  - *(Why?)*



Intel x86 Protection Ring Rules

Each memory segment has an associated privilege level, 0 through 3

The CPU has a Current Protection Level (CPL)

- Usually the privilege level of the segment where the program's instructions are being read from

Program can read/write data in segments of *lower privilege* than CPL.

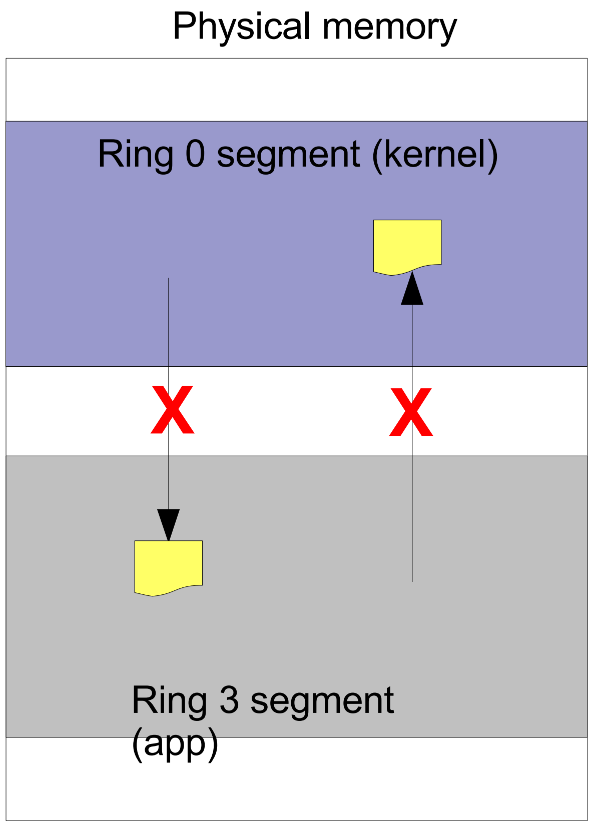
- e.g., Kernel can read/write user memory
- But, user cannot read/write kernel memory

Program cannot (directly) call code in *higher-privilege* segments

- *Why?*

Program cannot (directly) call code in *lower-privilege* segments

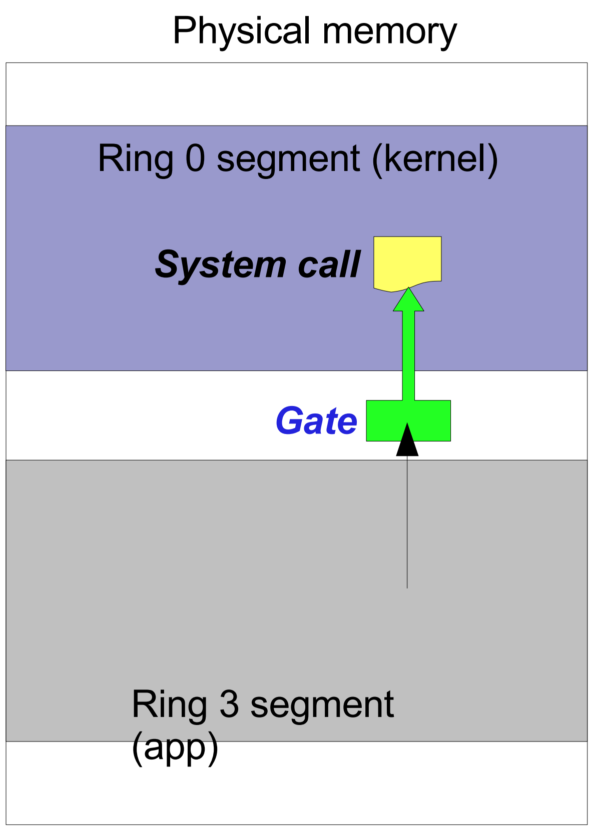
- *Why?*



Intel x86 Protection Ring Rules

A gate is a special memory address that "opens a door" from a low-privilege segment to a high-privilege one.

- When a low-privilege program calls a gate, it automatically raises its CPL to the higher level.
- When returning from a gate subroutine, CPL automatically dropped to original level.



Gates allow high-privilege program to control the entry points into its code.

- For example, OS system calls can be implemented using gates.

Gates do not allow higher-privileged code to call lower-privileged code.

- *Why?*

System Call Issues

Kernel must verify the arguments that it is passed – why?

How does the application pass in data to the kernel?

- Example: write() passes in a pointer to a buffer of data to write to the file

How does the OS return kernel state from a system call?

- Example: read() returns an int indicating the number of bytes read

Uniprogramming and Multiprogramming

Uniprogramming

- Only one program can run at a given time on the system
- Like old batch systems, MS-DOS, etc.

Multiprogramming (a.k.a. “multitasking”)

- Multiple programs can run simultaneously
- Although only one program running **at any given instant**
  - *(Unless you have multiple CPUs!!!!)*

Tradeoffs

- Writing a uniprogramming OS is simpler
  - *Why?*
- But, multitasking OSs use resources more efficiently
  - *Why?*

Note on terminology:
Multitasking/multiprogramming refer to the number of programs running
Multiprocessing refers to the number of CPUs in the system

Process Management

An OS executes many kinds of applications

- Regular user programs
  - *Emacs, Mozilla, this OpenOffice program, etc...*
- Administrative servers
  - *Crond: Runs jobs at pre-scheduled times*
  - *Sshd: Manages incoming ssh connections*
  - *Lpd: Queues up jobs for the printer*

Each of these activities is encapsulated in a *process*

- A process consists of three main parts:

Processor state

- registers, program counter

OS resources

- open files, network sockets, etc.

Address space:

- The memory that a process accesses – its code, variables, stack, etc.

Process Example

A **process** is an instance of a program being executed

- Use "ps" to list processes on UNIX systems

```
 PID TTY      STAT   TIME COMMAND
 842 tty1     S      0:00 -bash
 867 tty1     S      0:00 xinit
 873 tty1     S      0:00 fvwm2
 887 tty1     S      0:00 xload
 888 tty1     S      0:02 /usr/local/j2sdk1.4.0/bin/java ApmView 896 243
1881 tty1     S      0:00 rxvt -fn fixed -cr red -fg white -bg #586570 -geometr
1883 pts/2    S      0:00 bash
1910 pts/0    S      0:00 /bin/sh /home/mdw/bin/ooffice arch.sxi
1911 pts/0    S      1:20 /usr/local/OpenOffice.org1.1.0/program/soffice.bin ar
1937 tty1     S      0:00 /bin/sh /home/mdw/bin/set-wlan-OFF
2310 pts/2    R      0:00 ps -Umdw -x
```

terminal

Status

Command line

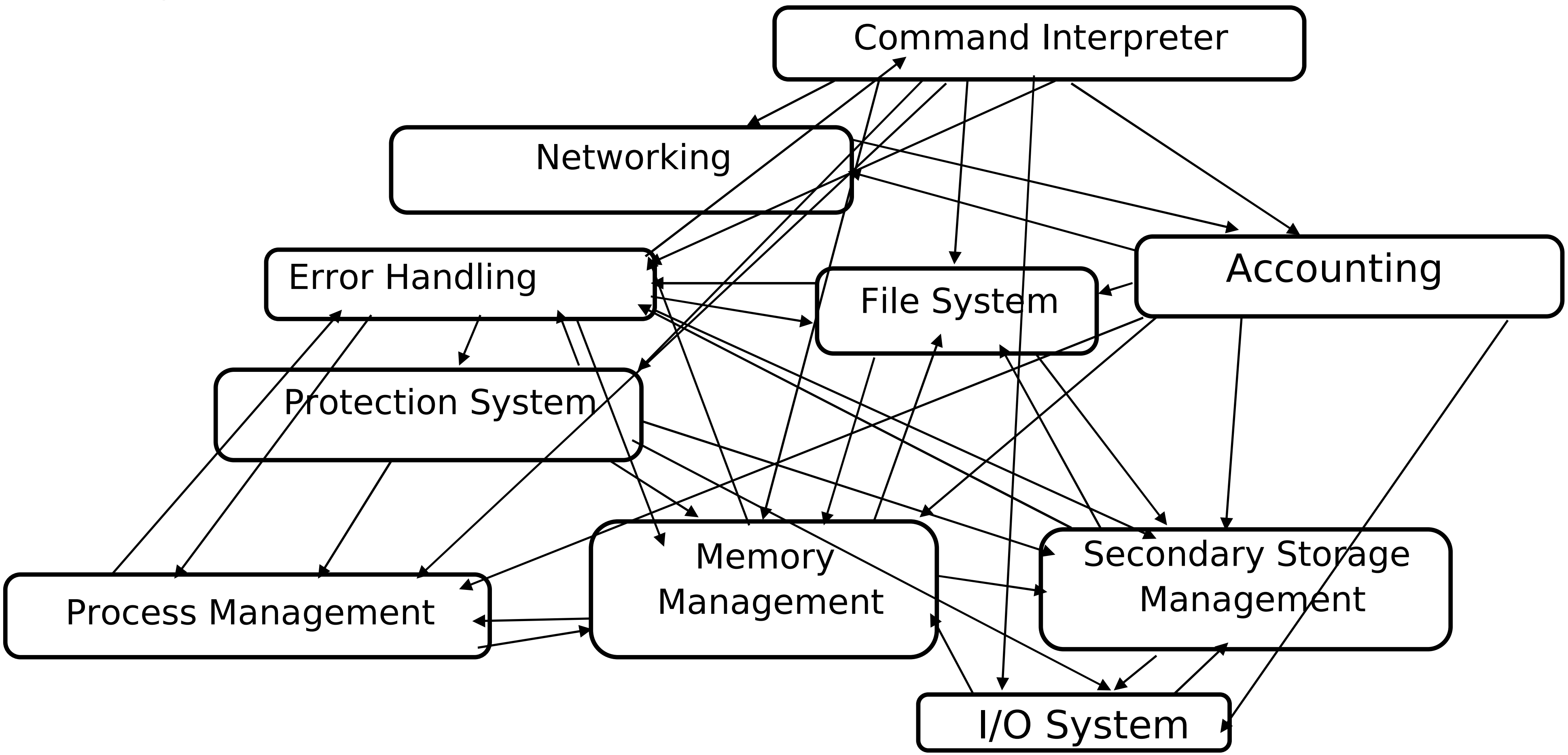
Process ID

Total CPU time

The next few lectures will deal with details of process creation and management...

OS Structure

It's not always clear how to tie these different OS components together...

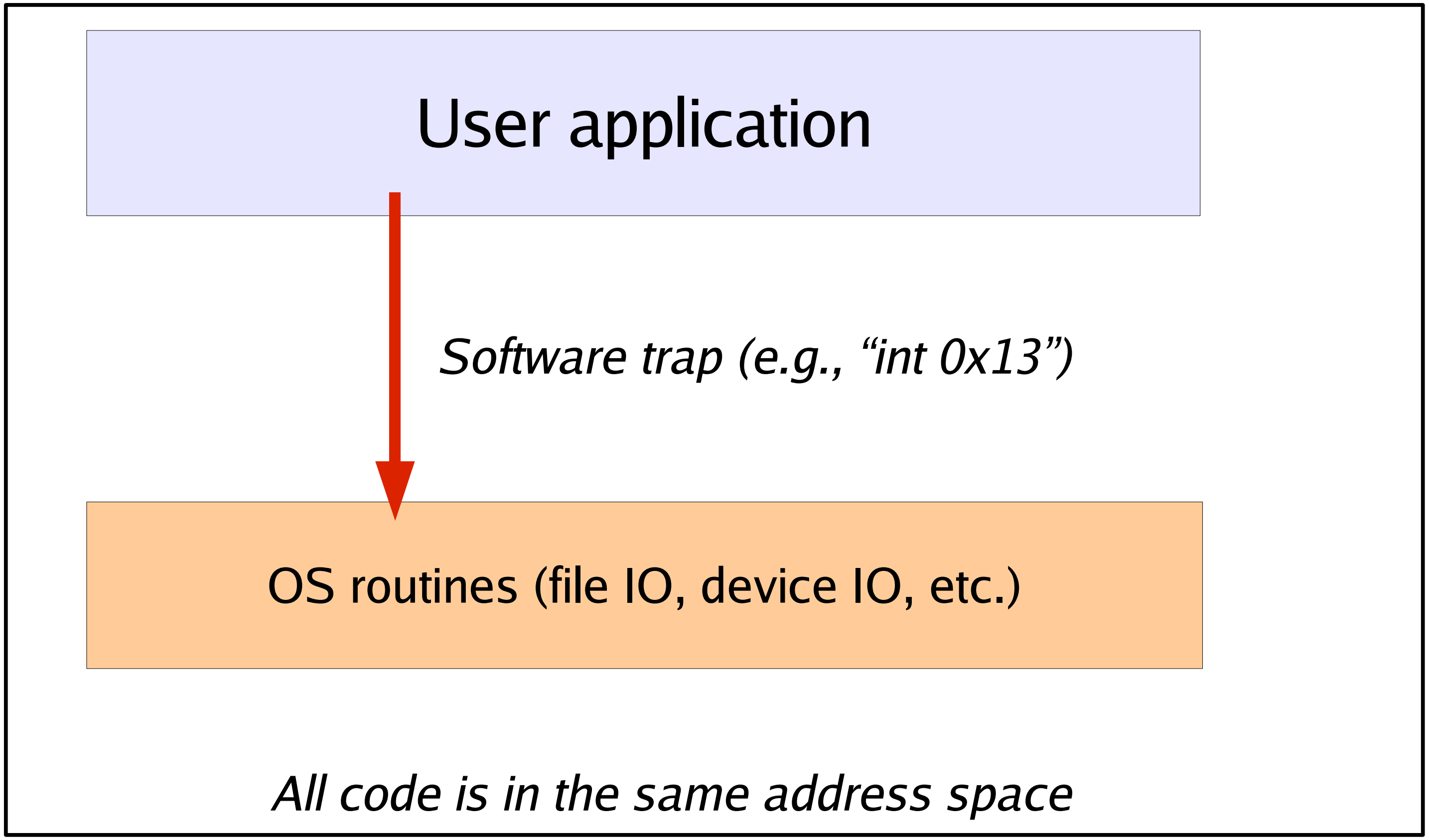


(For illustrative purposes only)

"Executive" model

Not really an OS "kernel" per se ...

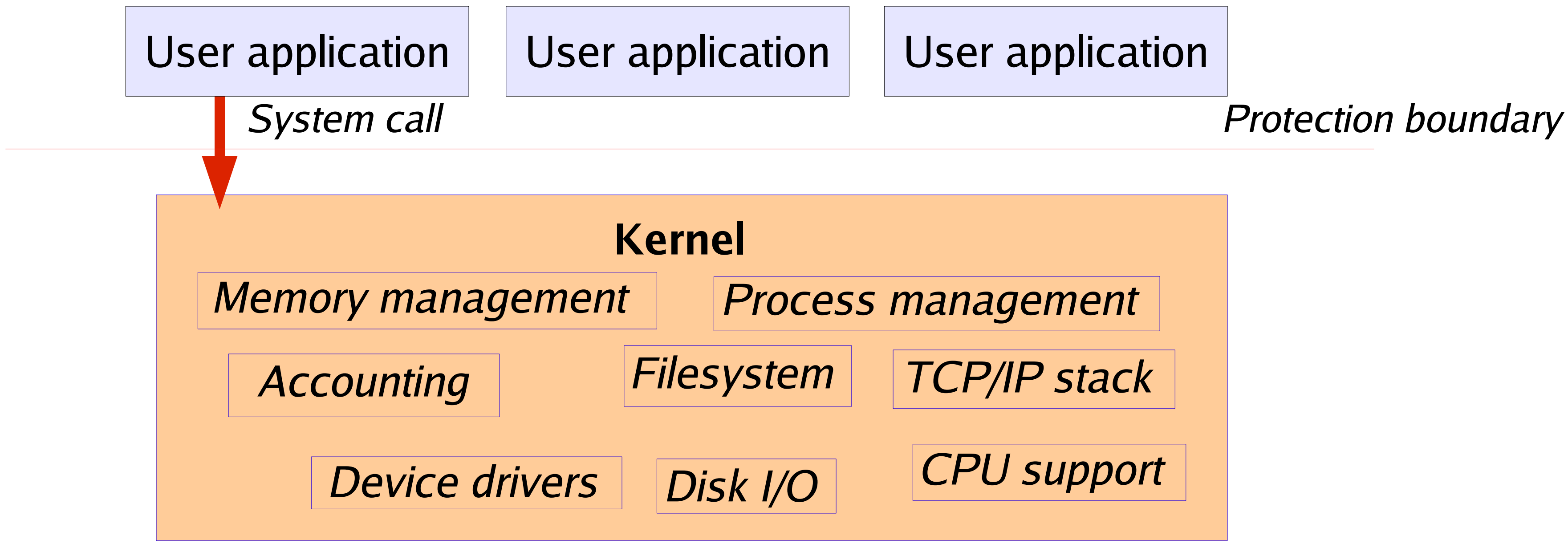
- But rather a collection of routines, resident in memory, that the application can invoke
  - *MS-DOS used this structure*
- Effectively a library, except the application is not explicitly linked to the OS routines
  - *Instead, uses software traps to invoke OS routines from user code*



Monolithic Kernels

Most common OS kernel design

- Kernel code is privileged and lives in its own address space
- User applications are unprivileged and live in their own separate address spaces
- All kernel functions loaded into memory as one large, messy program

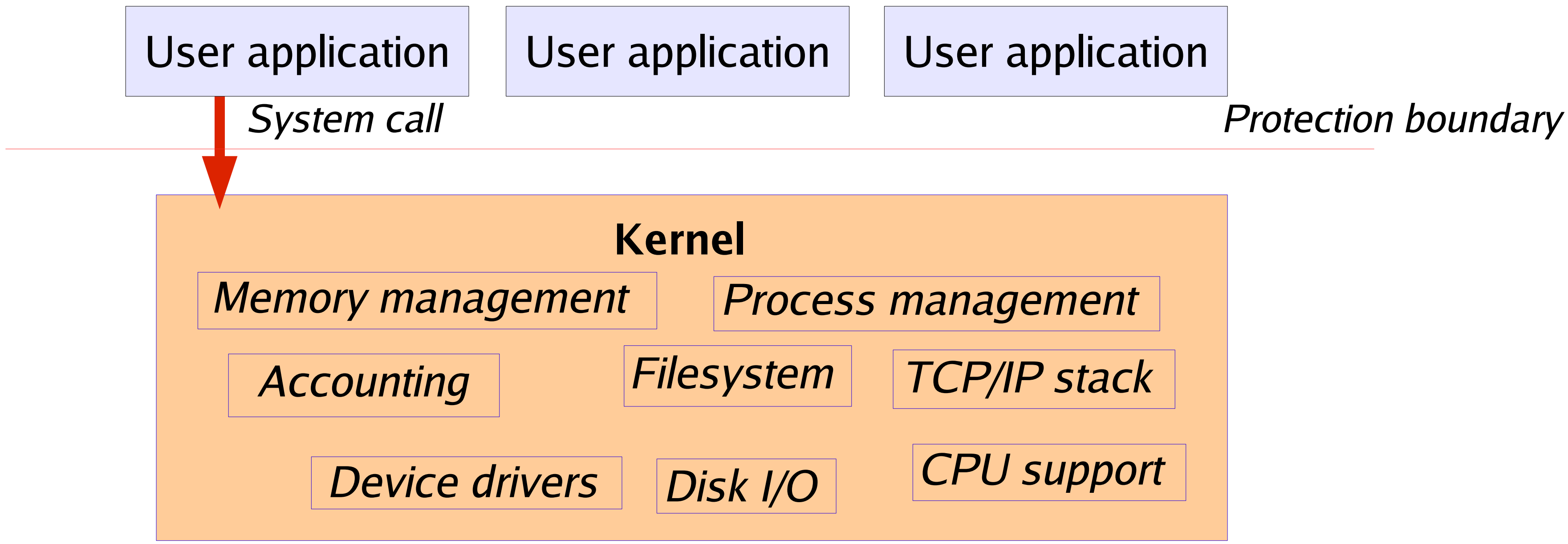


Pros and cons???

Monolithic Kernels

Most common OS kernel design

- Kernel code is privileged and lives in its own address space
- User applications are unprivileged and live in their own separate address spaces
- All kernel functions loaded into memory as one large, messy program



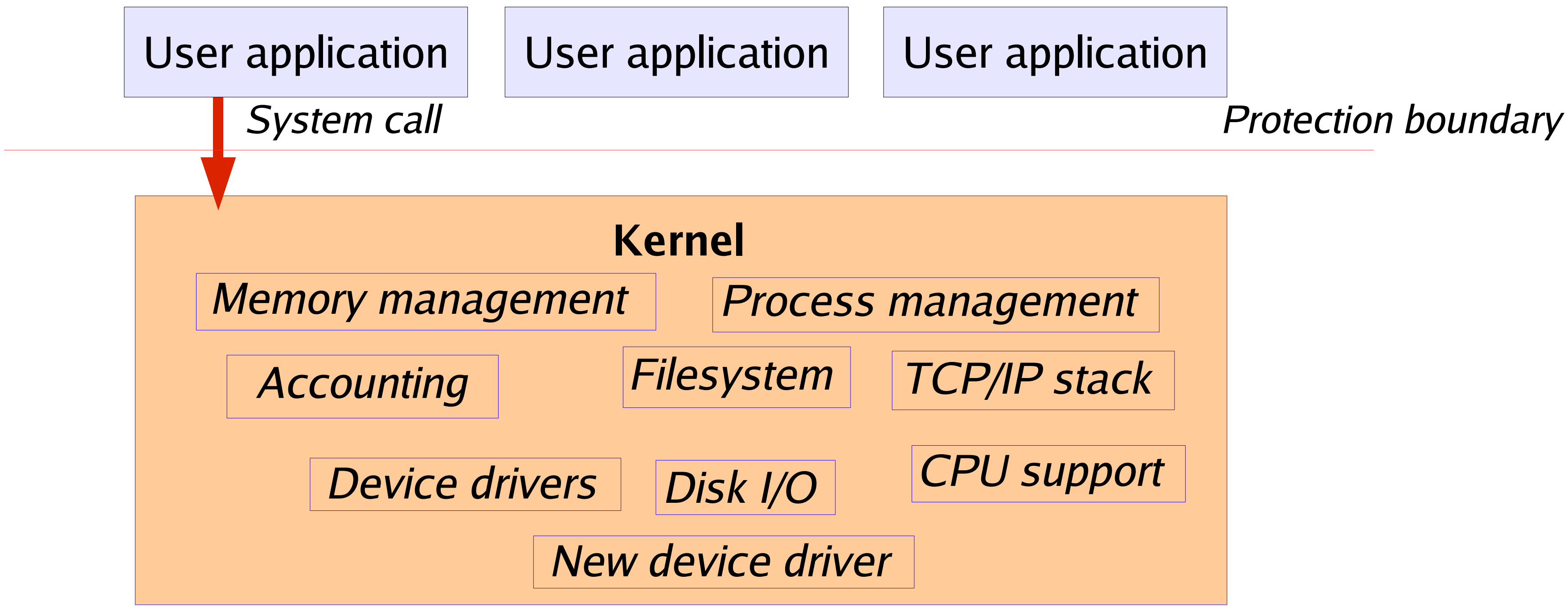
Pros and cons

- Pro: Overhead of module interactions *within* the kernel is low (function call)
- Pro: Kernel modules can directly share memory
- Con: Very complicated and difficult to organize
- Con: A bug in *any part* of the kernel can crash the whole system!

Loadable Modules

Allows new kernel code to be dynamically loaded and unloaded

- The kernel is otherwise monolithic, as before

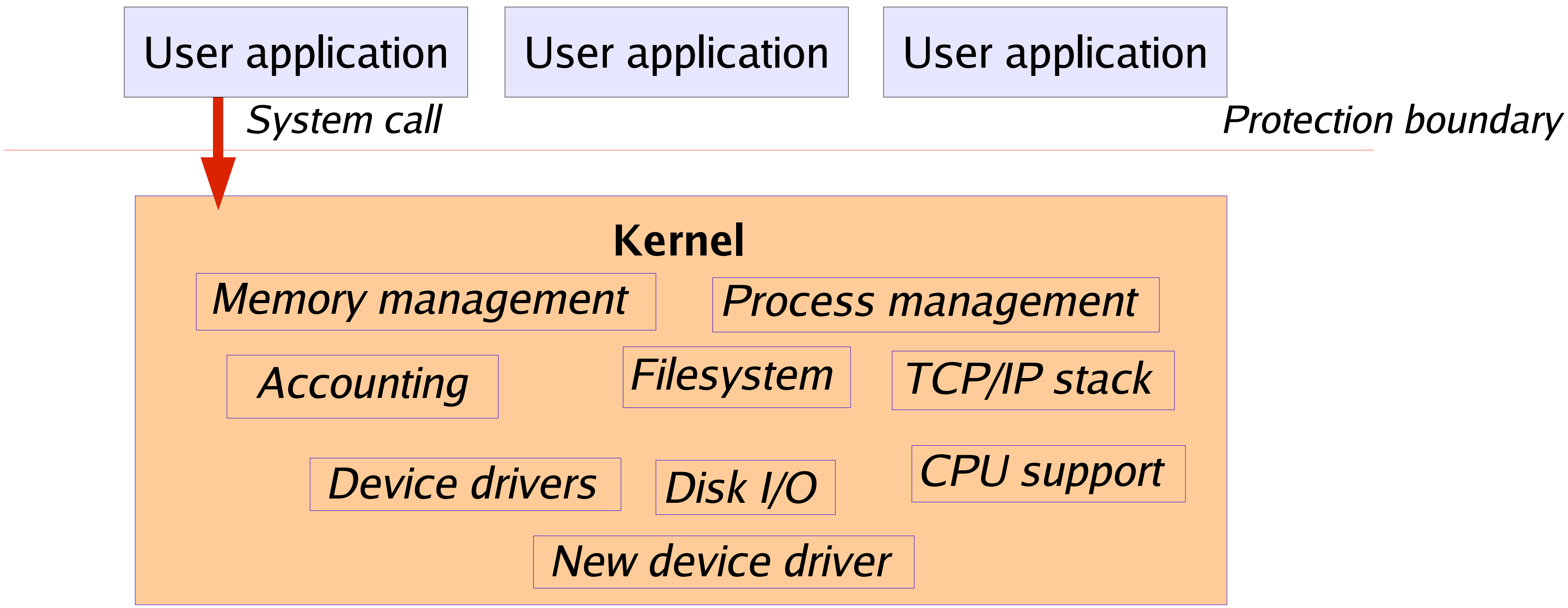


Pros and cons???

Loadable Modules

Allows new kernel code to be dynamically loaded and unloaded

- The kernel is otherwise monolithic, as before



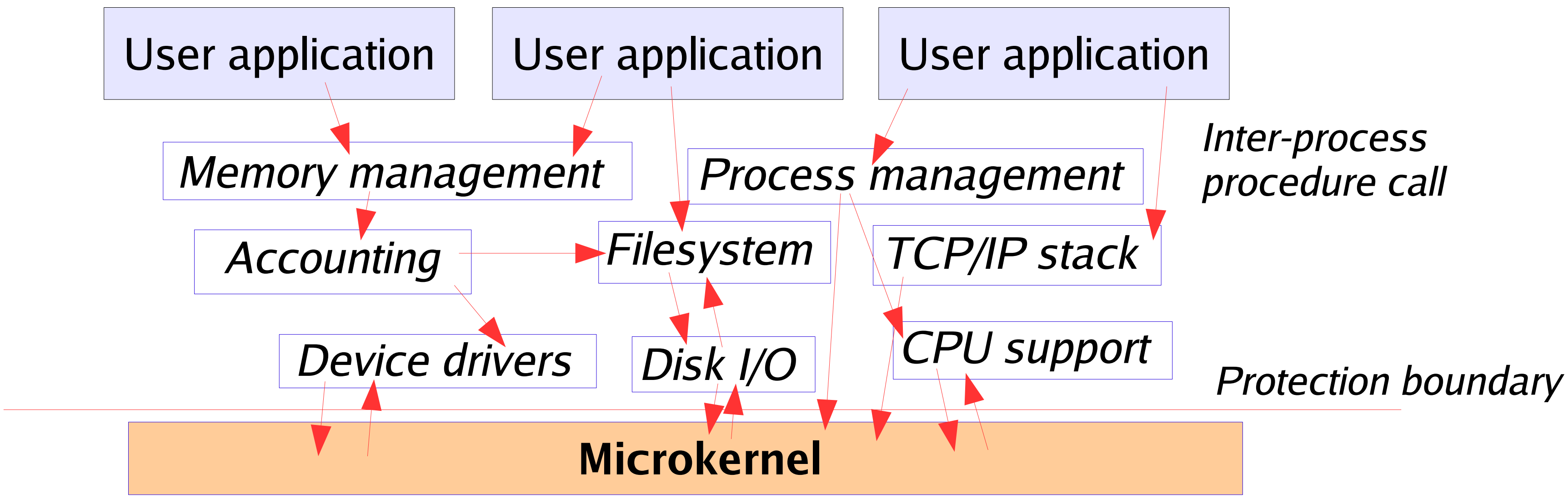
Pros and cons

- Pro: Save memory by only loading drivers, etc. that are needed
- Pro: Makes it easier to develop new kernel code outside of the main tree
- Con: Once loaded, module can wreak havoc on the running kernel

Microkernels

Use a very small, minimal kernel, and implement all other functionality as user level “servers”

- Kernel only knows how to perform lowest-level hardware operations
- Device drivers, filesystems, virtual memory, etc. all implemented on top
- Use inter-process procedure call (IPC) to communicate between applications and servers



Pros and Cons???

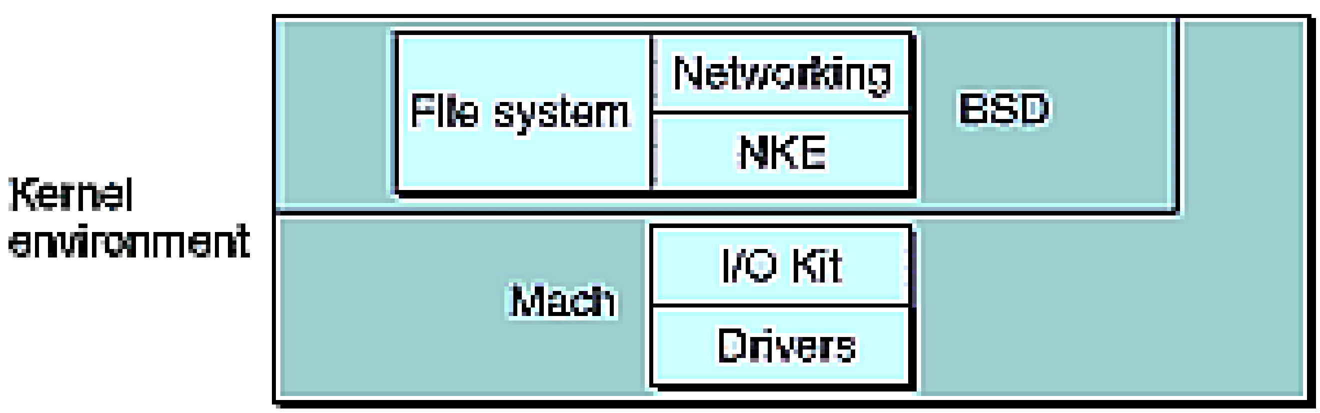
Microkernels 2

Pros and cons

- Pro: Kernel is small and simple, servers are protected from each other
- Con: Overhead of invoking servers may be very high
  - *e.g., A user process accessing a file may require inter-process communication through 2 or 3 servers!*

Microkernels today

- Lots of research in late 80's and early 90's
- Windows NT uses “modified microkernel”:
  - *Monolithic kernel for most things, OS APIs (DOS, Win3.1, Win32, POSIX) implemented as user-level services*
- Mac OS X has reincarnated the microkernel architecture as well:
  - *Gnarly hybrid of Mach (microkernel) and FreeBSD (monolithic)*



Administrative Stuff

Next Lecture: Dive into the process subssystem

- How the OS manages loading and running executables

Read Tanenbaum 2.1