OS Architecture and System Calls

Instructor: William Zheng
Email:
william.zheng@digipen.edu
PHONE EXT: 1745

A word about the study of OS

- Generalist versus Specialist
- Organic and circularity

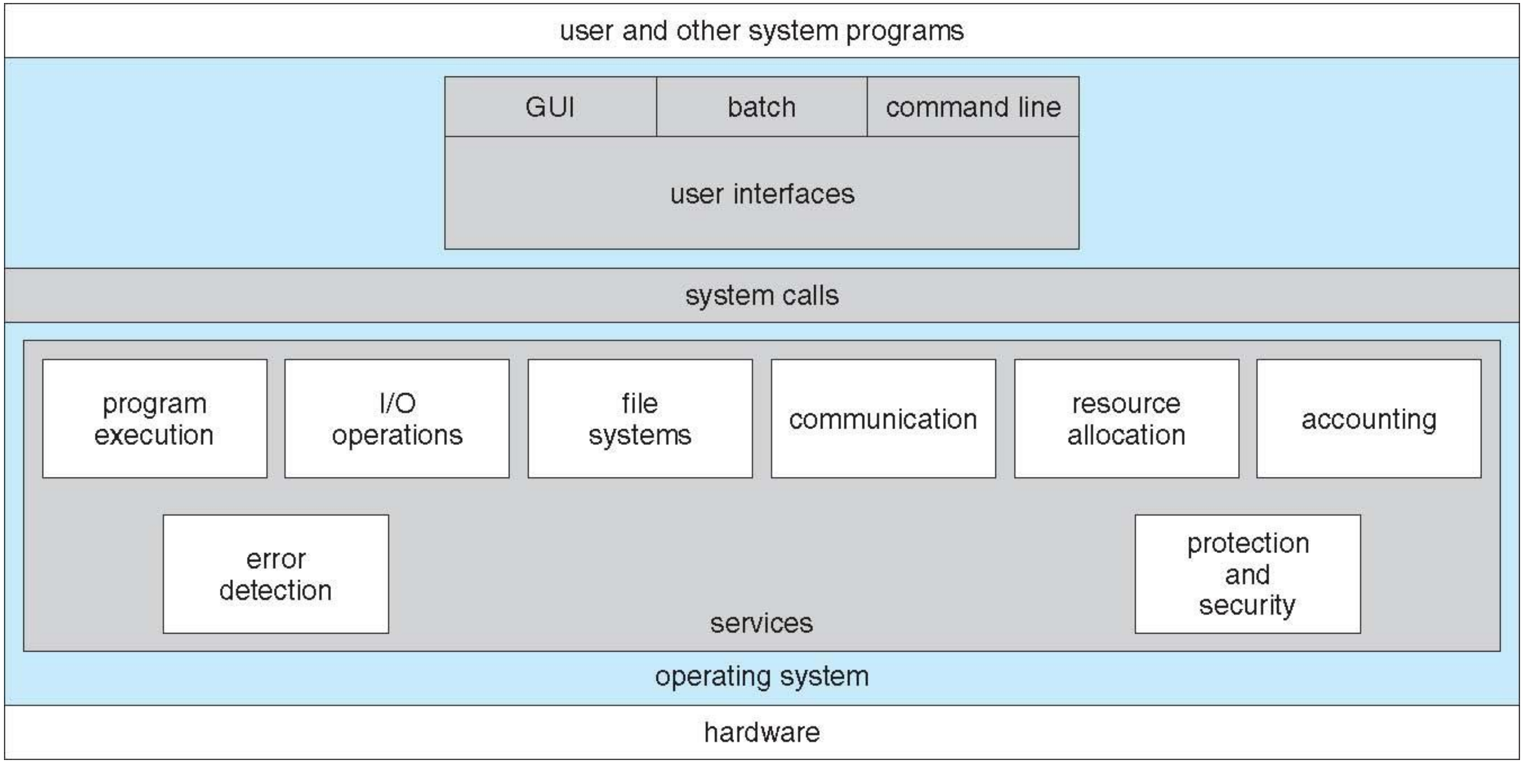
Goals of this lecture

- System Calls
  - Interface between user program and OS
- OS architecture
  - Organization and Design of OS code

What is an OS and what are the roles of OS?

- Process Management
  - A process a running program.
  - Starting, terminating processes
  - Coordinating processes (allowing processes to talk to one another, protecting them from one another)
- Memory Management
  - Allocation of memory for the processes
  - Protect memory from being written by another process
  - How to handle when we run out of RAM
- I/O Management
  - Device drivers
  - Providing API for the user programs (encapsulation principle)
- Storage Management
  - File systems and data

OS Services



user and other system programs
GUI
batch
command line
user interfaces
system calls
program execution
I/O operations
file systems
communication
resource allocation
accounting
error detection
protection and security
services
operating system
hardware

Invoking the OS services

- int instruction
  - Generate the software interrupt
    - int 80h
    - 80h: the interrupt vector for system call
  - Return: iret
- sysenter/sysexit instruction
- syscall instruction

```
Main:
        …
        call FUNC
        …
        ret

FUNC:
        …
        …call the OS. How?
        …
        ret
```

System calls

- API provided by OS
  - For access to services provided by OS
- How to access?
  - software-generated interrupt (usually)
  - call instruction (older days)

Why system calls necessary

Access to privileged operations:
Many operations, such as managing hardware devices or modifying system configurations, require higher privileges that are only accessible through system calls.
Resource management:
System calls provide a standardized interface for allocating and managing system resources like memory, files, and devices, ensuring fair and controlled access by different processes.

Why system calls necessary

Abstraction:

- *System calls abstract the underlying complexities of the operating system, allowing application developers to interact with the system in a higher-level, platform-independent manner.*

Security and protection:

- *System calls enforce access control and security policies, preventing unauthorized access to sensitive resources and protecting the integrity of the system.*

Types of system calls – Windows/Linux

Process	Windows	Linux
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	Fork() Exit() Wait()
File manipulation	CreateFile() ReadFile() WriteFile()	Open() Read() Write() Close()
Device Management	SetConsoleMode() ReadConsole() WriteConsole()	Ioctl() Read() Write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	Getpid() Alarm() Sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	Pipe() Shmget() Mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorgroup()	Chmod() Umask() Chown()

System call example

- Copy contents of file A to file B
  - How many system calls involved?

Difference between system call and function call

- System call: a call into kernel code, typically performed by executing an interrupt
- Function call, if calling a system call (via library), switches into kernel mode from user mode

Portability - I

Program X

Computer
System A

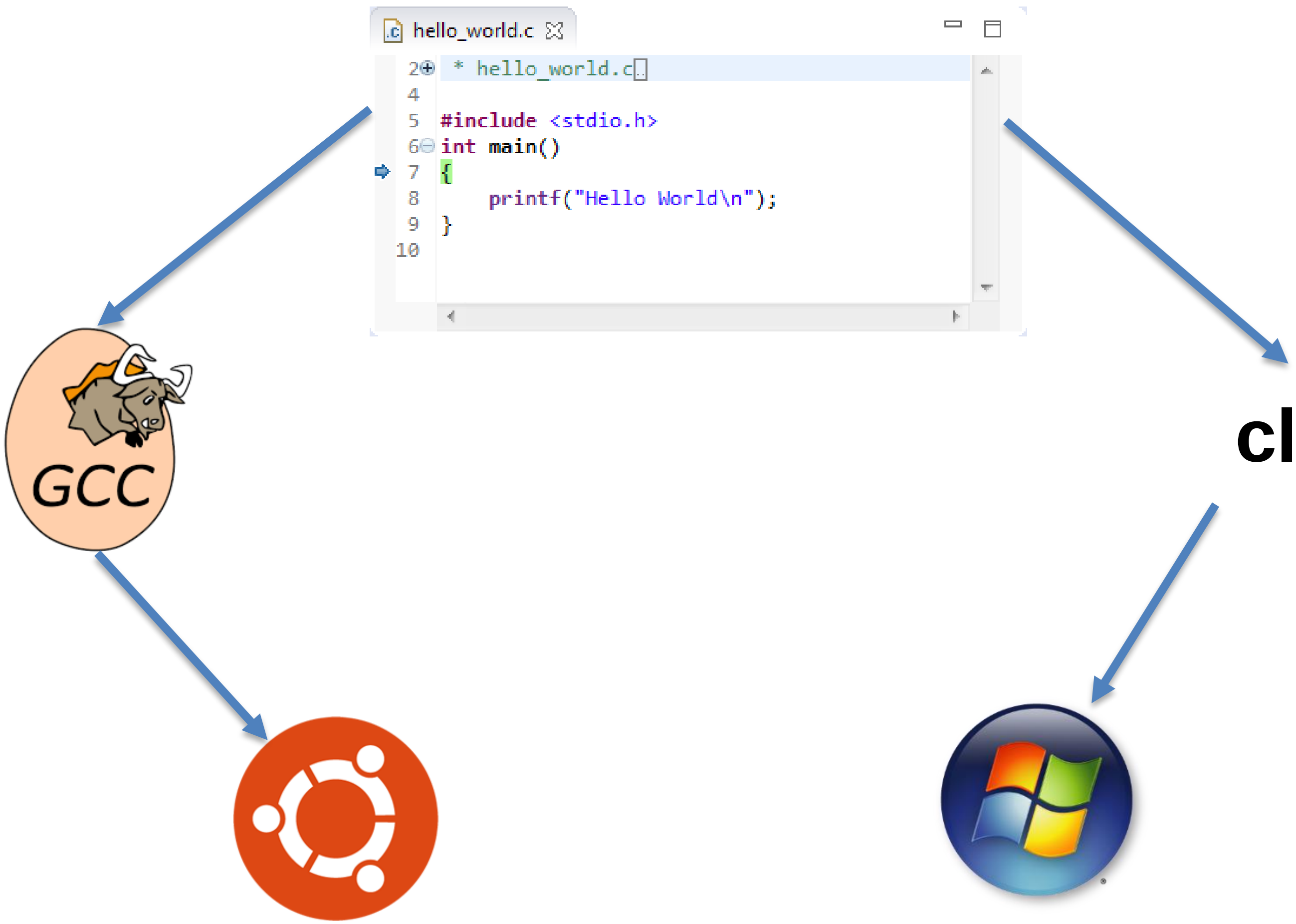
Program X

Computer
System B

Portability - II

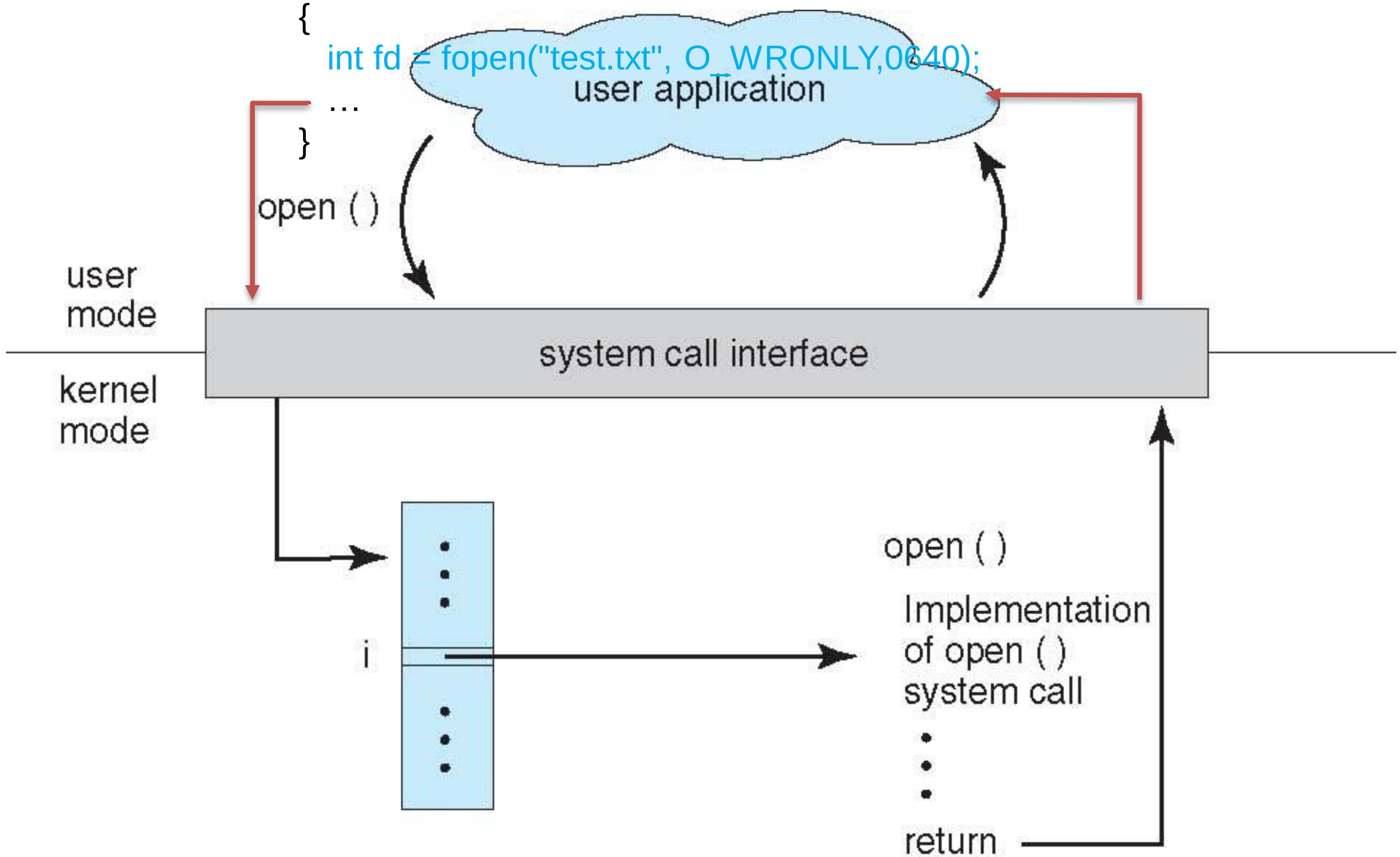
- Binary level
- Source code level

Source Code Portability - Demo



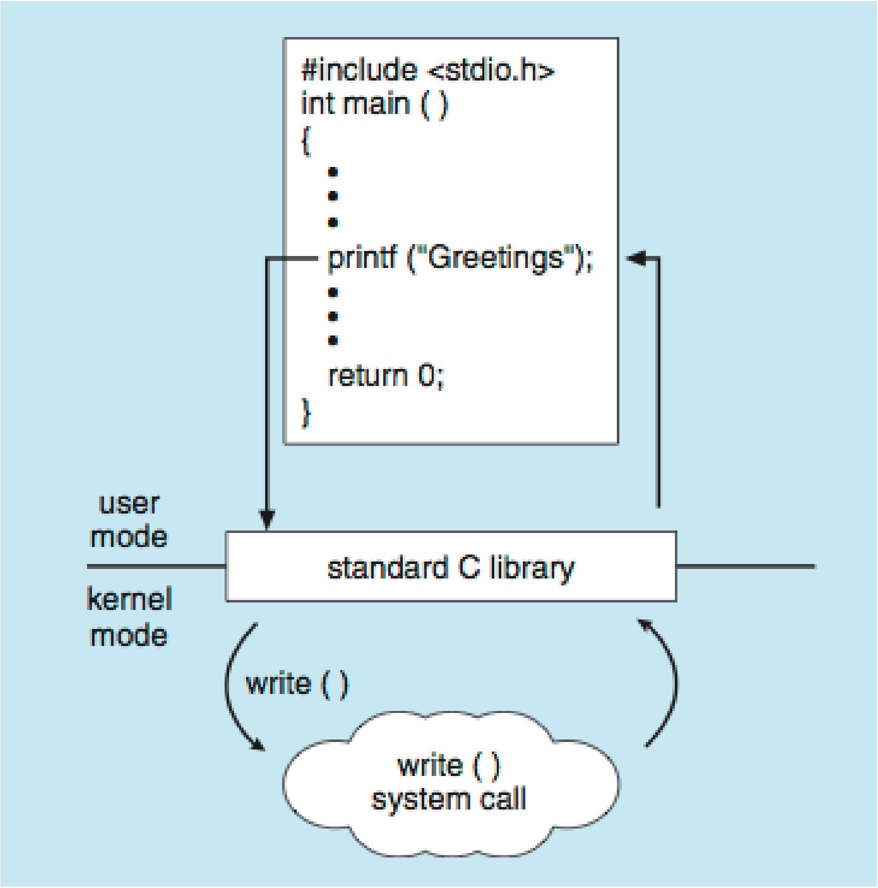
System calls, API and portability

```
#include <stdio.h>
int main ()
{
   int fd = fopen("test.txt", O_WRONLY,0640);
   …
}
```

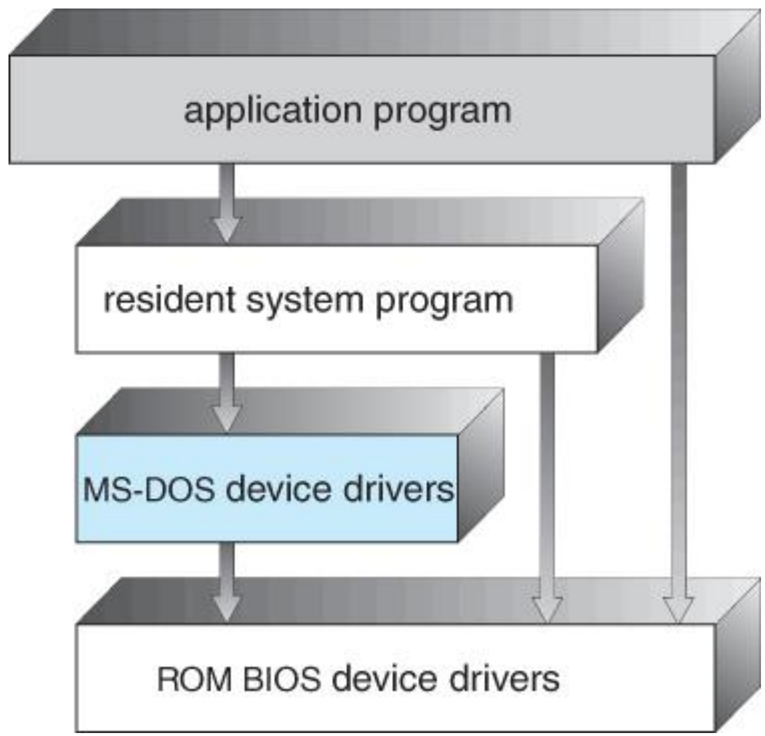


Standard C Library Example

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

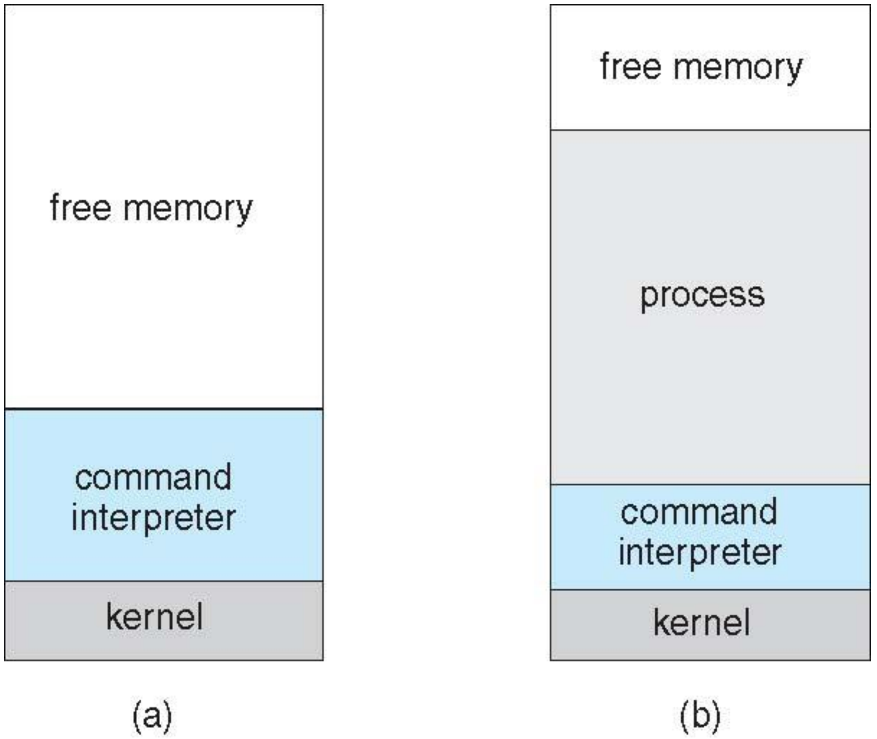


DOS OS architecture - I



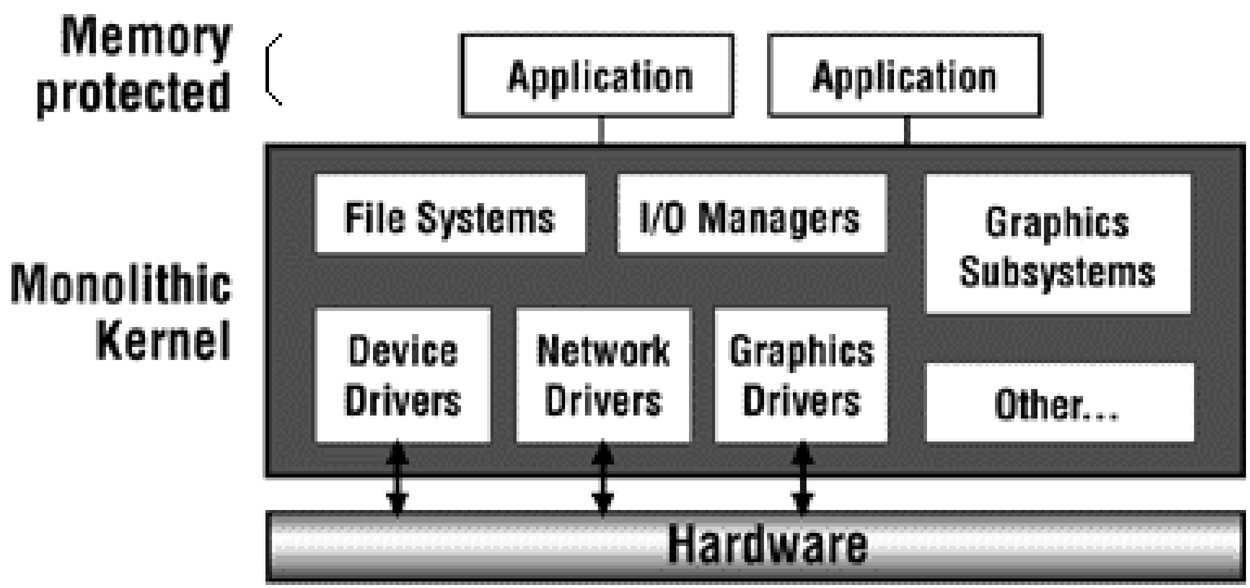
DOS OS architecture - II

- Single-tasking
- Shell invoked when system booted
- Simple method to run program
  - No process created
- Single memory space
- Loads program into memory, overwriting all but the kernel
- Program exit -> shell reloaded



OS architecture

- Everything *almost* runs in kernel model
- Example: Linux
- Pros and Cons



Monolithic OS architecture

Pros: Good performance (**S**peed) / (Shared kernel space) Simplicity of design

Cons:

No information hiding

Potential stability issues / Chaotic

Hard to understand

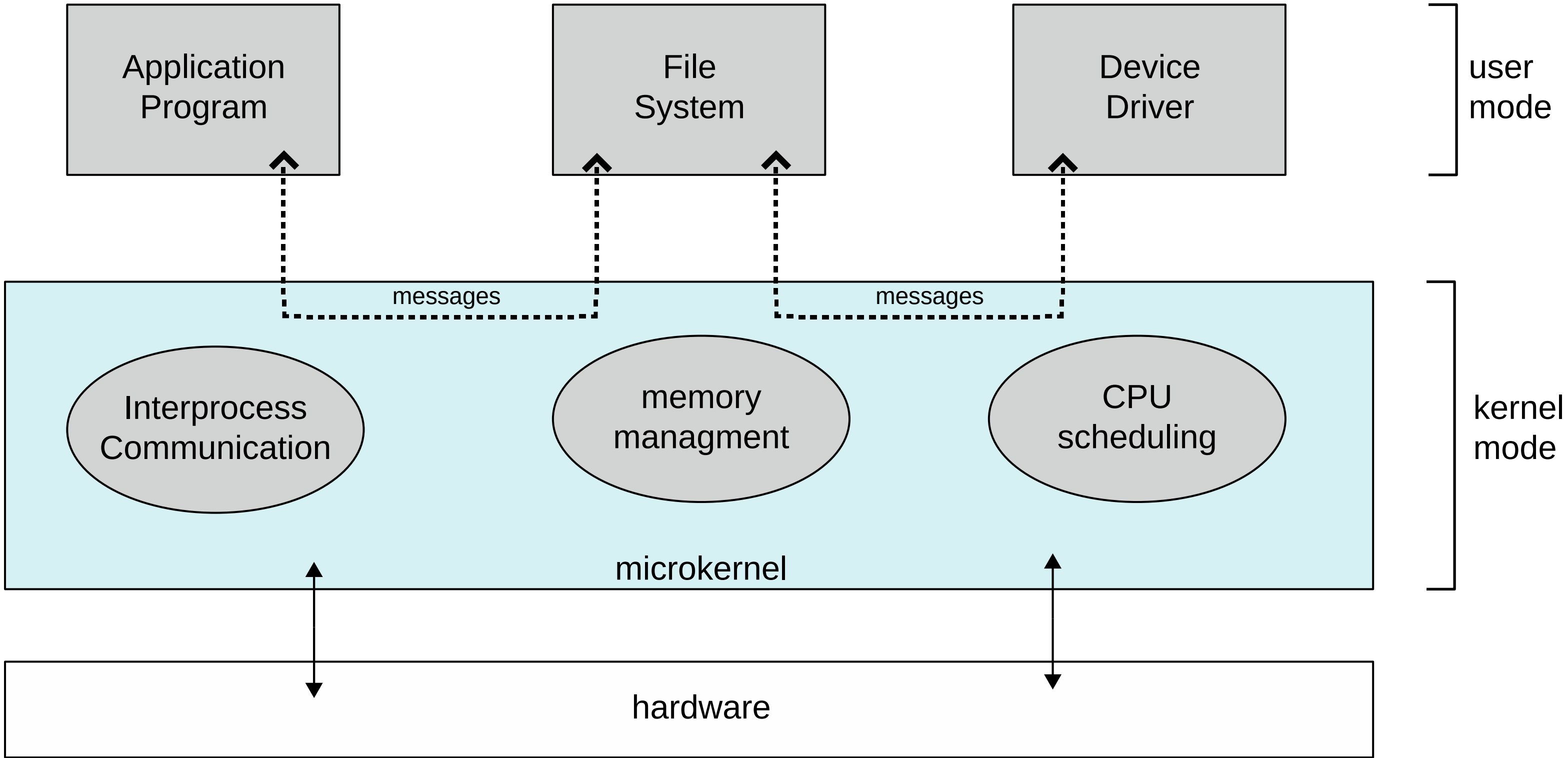
- Can become huge
- Linux 4.15 has 20 million lines of code
- Windows 10 contains over 40 million lines!
- Potentially difficult to maintain

Examples:

- Traditional Unix kernels (includes BSDs and Solaris)
- Linux
- MS-DOS, Windows 9x
- Mac OS versions below 8.6

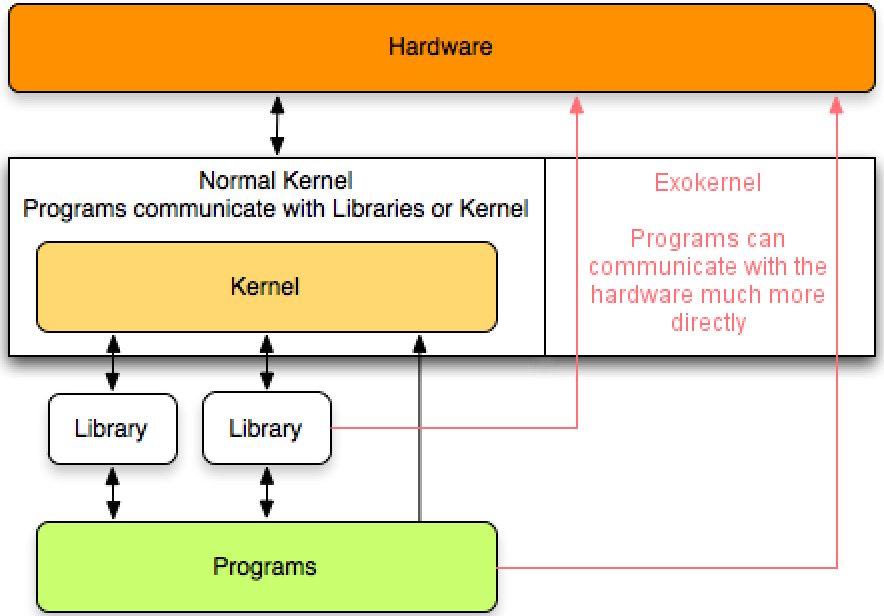
Microkernel architecture

- Push stuffs out of kernel
- Client-Server model
- Example: Mac OS, iPhone OS etc
- Pros : Modularity: easier management, Fault isolation and reliability
- Cons: Inefficient (boundary crossings); Insufficient protection; Inconvenient to share data between kernel and services

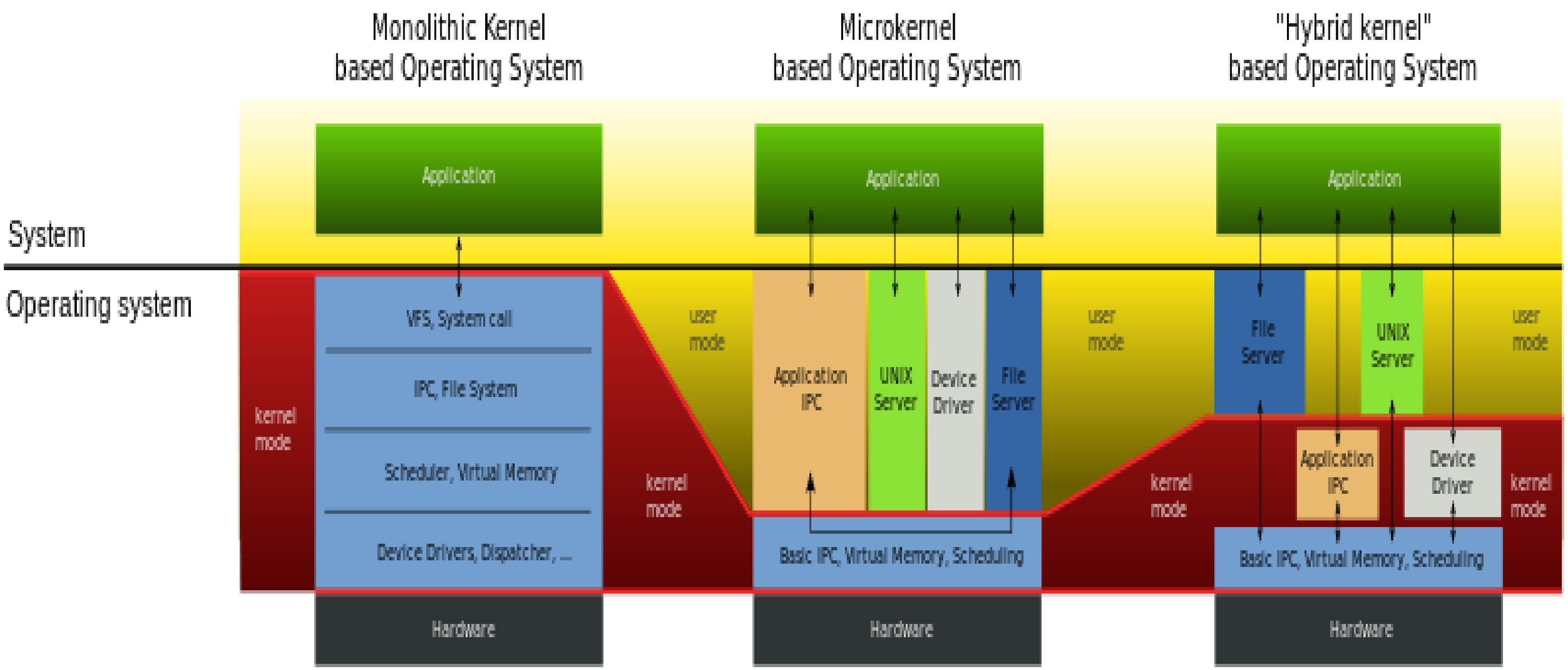


Exokernel architecture

- Proposed by MIT
- Application-oriented OS
  - Gives them all control
- Research



Comparison



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
Basic IPC, Virtual Memory, Scheduling
user mode
kernel mode
File Server
UNIX Server
Application IPC
Device Driver
Basic IPC, Virtual Memory, Scheduling
user mode
kernel mode
Hardware
Hardware
Hardware

Questions

- Is the operating system using the CPU at all times?
- After booting and initialization, what are the circumstances that would cause OS code to use the CPU?