CSc 360
Operating Systems
OS Interfaces+Structures

Wenjun Yang

Fall 2025

OS services

- User/programmer interfaces
  - command line, GUI, API, system calls
- Program execution
- I/O operation
- File manipulation
- Process communication
- Error handling: software/hardware error

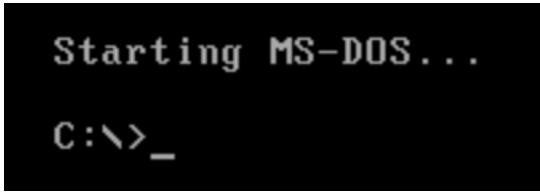
* OS in a (web) browser?

More OS services

- Resource allocation and arbitration
  - CPU, memory, storage, I/O
- Resource sharing and protection
  - among processes, users, computers
  - authentication, authorization, accounting
- Different interfaces to these services
  - regular user, application programmer, system programmer, system designer

* OS virtualization

Command line interface



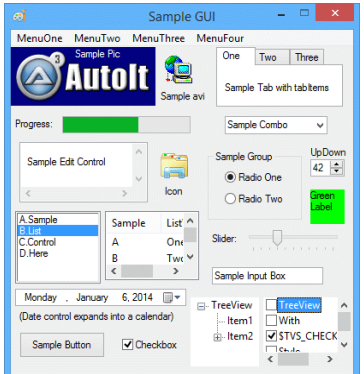
- E.g.
  - Microsoft DOS: \command.com
  - Linux: /bin/bash
- Interactivity: interpreter
- Implementation
  - internal: dir (DOS), cd (DOS/Unix)
  - external: ls (Unix)
- Programmability: shell script
  - run ./hello.sh

hello.sh

```
#!/bin/bash
echo "Welcome to CSC 360!"
```

Graphics user interface

- E.g.
  - Microsoft Windows
  - K Desktop Environment (KDE)
- Interactivity: point-and-click, drag-and-drop
- Implementation
  - integrated with OS (e.g., start menu in Windows)
  - or OS front-end (e.g., KDE)
- Programmability: e.g., AutoIt



System calls

- Primitive interfaces to OS services
- System call categories
  - process control
    - fork, exec*, wait, kill, signal, exit, etc
  - file/device manipulation
    - creat[e], open, read, write, lseek, close, etc
    - socket, bind, listen, accept, connect, etc
  - information manipulation
    - time, getpid, getgid, gethostname, etc

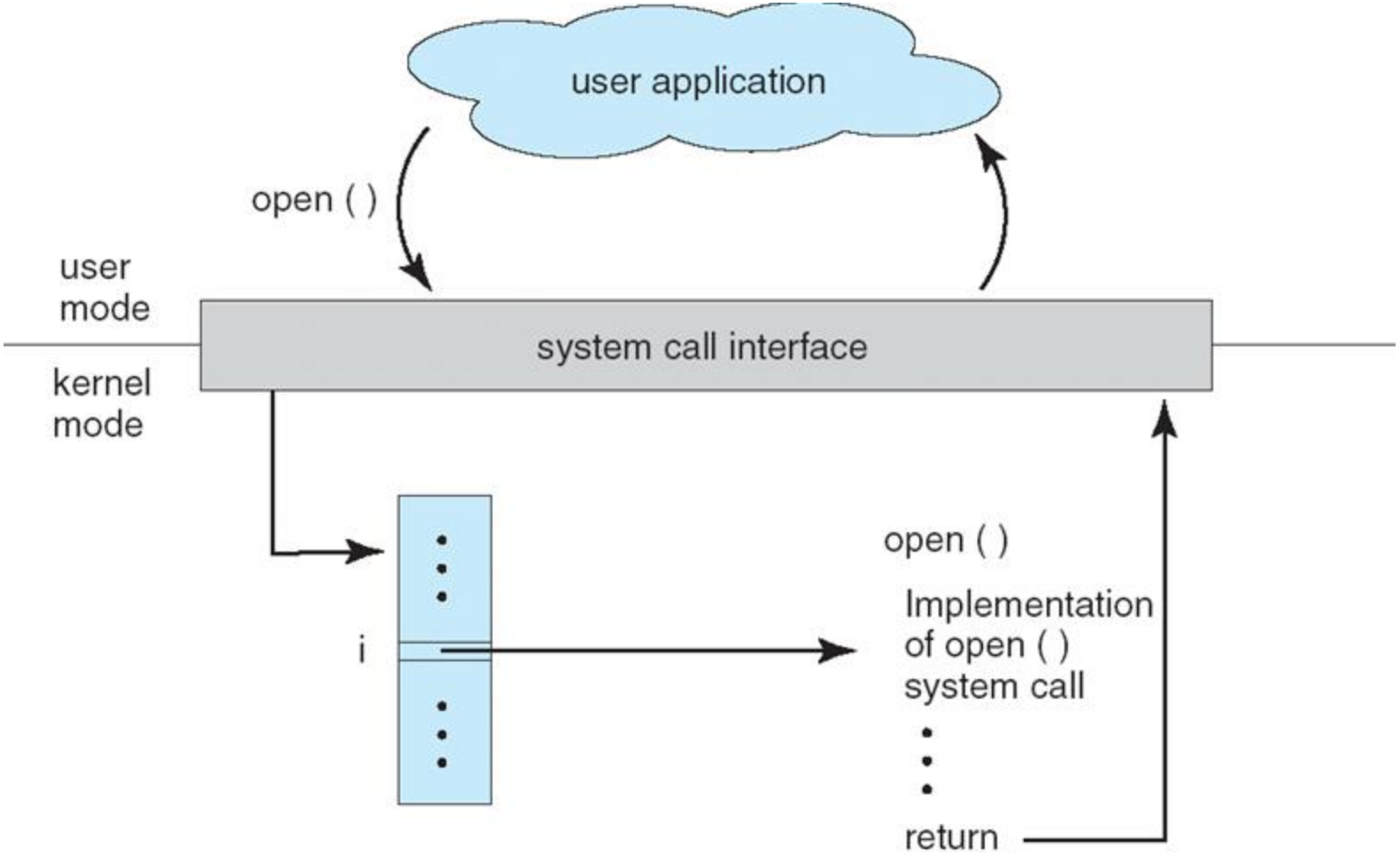
System call examples

- Copy (the content of) file A to file B
  - in CLI: cp /path/to/a /path/to/b
  - in GUI: Ctrl-C and Ctrl-V, Drag-and-Drop
- With system calls
  - open(“/path/to/a”, O_RDONLY);
  - creat(“/path/to/b”, S_IRWXU);
    - open() with O_CREAT|O_WRONLY|O_TRUNC
  - read() and write()
  - close()

System call implementation

- Software interrupt
  - e.g., INT21H in DOS
  - command: AH (e.g.,2A/2B: get/set system date)
  - parameters
    - in registers
    - on system stack
    - in memory (pointed by registers)
  - return status: in specific registers
  - return data

System call flows



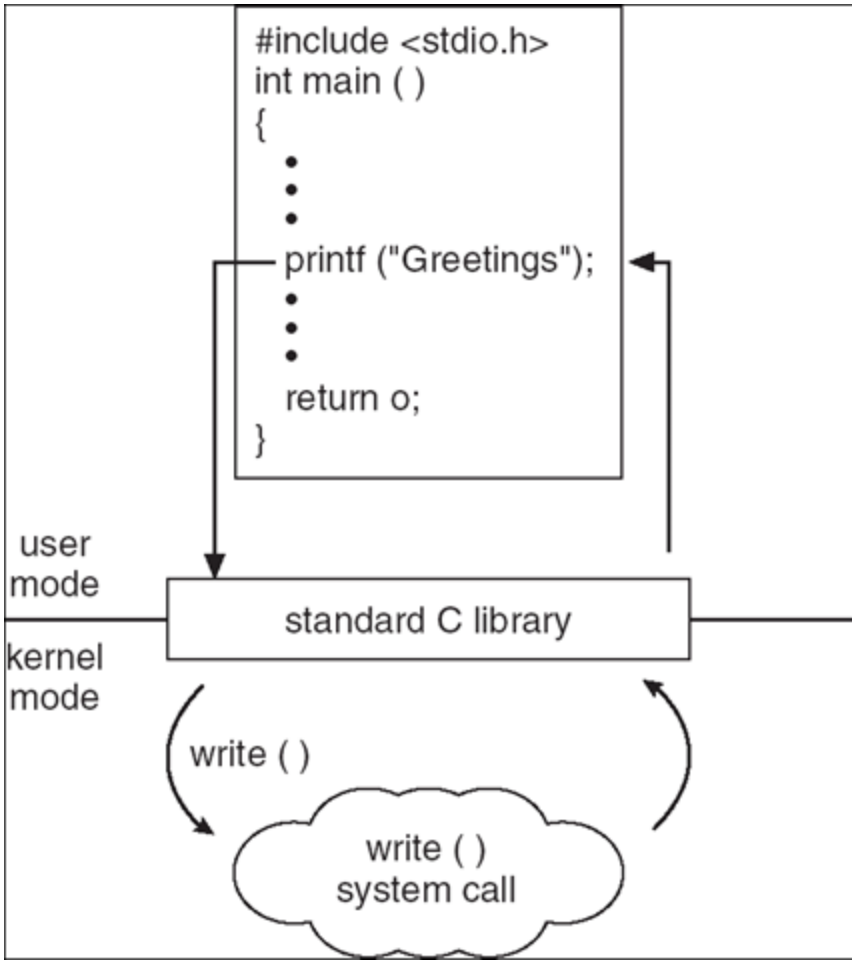
App programming interface

- E.g.
  - Win32 API: Windows
  - POSIX API: Unix, Linux, OSX, (Windows)
  - Java API: Java JVM
- API: another layer of abstraction
  - mostly OS-independent
  - higher level of functionality
    - implemented by a series of system calls and more

API examples

- Copy (the content of) file A to file B
- With C library
  - fopen(“/path/to/a”, “r”);
  - fopen(“/path/to/b”, “w”);
  - fread() and fwrite()
    - formatted I/O: element size, # of elements
    - buffered I/O: streams
  - fclose()

API flows


#include <stdio.h>
int main ( )
{
printf ("Greetings");
return o;
}
user mode
kernel mode
standard C library
write ( )
write ( )
system call

This lecture so far

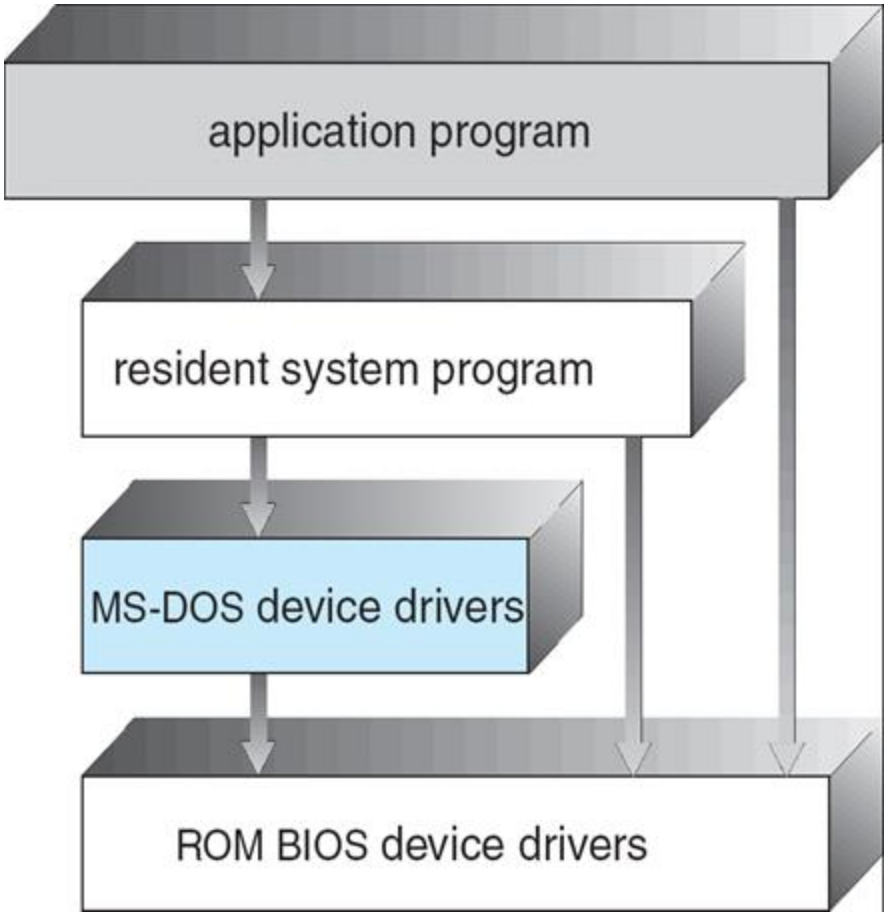
- Interfaces to OS services
  - CLI, GUI
  - system calls
  - API
- Explore further
  - compare different OS interfaces for one of your favorite tasks using lab computer
  - how to copy file attributes?

OS design and implementation

- An art of balance
  - hardware vs software
    - efficiency vs flexibility
  - user vs system
    - convenience vs effectiveness
- General design guidelines
  - separation of mechanisms and policies
- Best current practices

Simple (***all***-in-one) structure

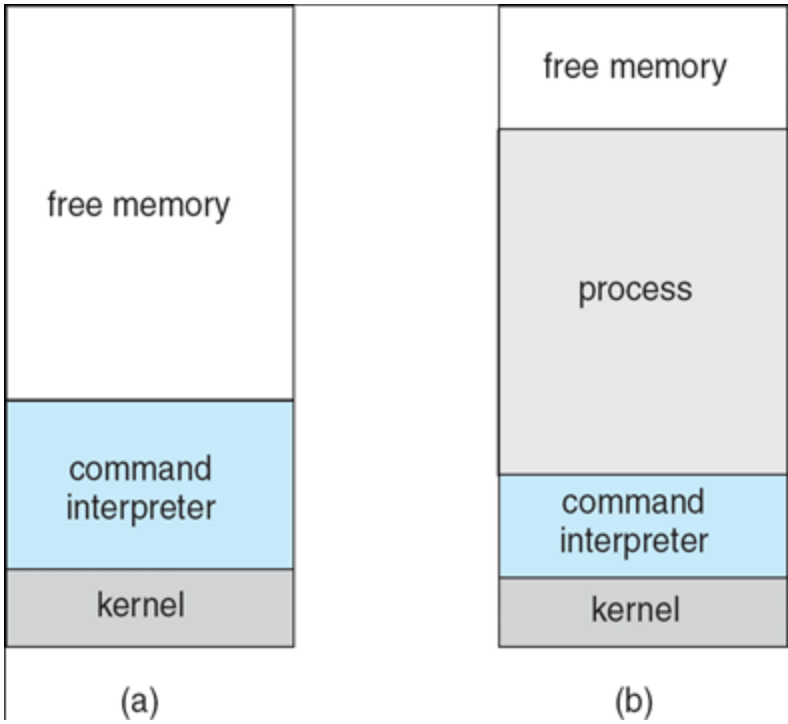
- E.g., MS-DOS
  - single user
  - almost single process
    - direct access
  - almost flat memory
    - MZ linked list
  - executables
    - .COM: segment limit
    - .EXE: MZ file magic



https://en.wikipedia.org/wiki/Mark_Zbikowski

MS-DOS

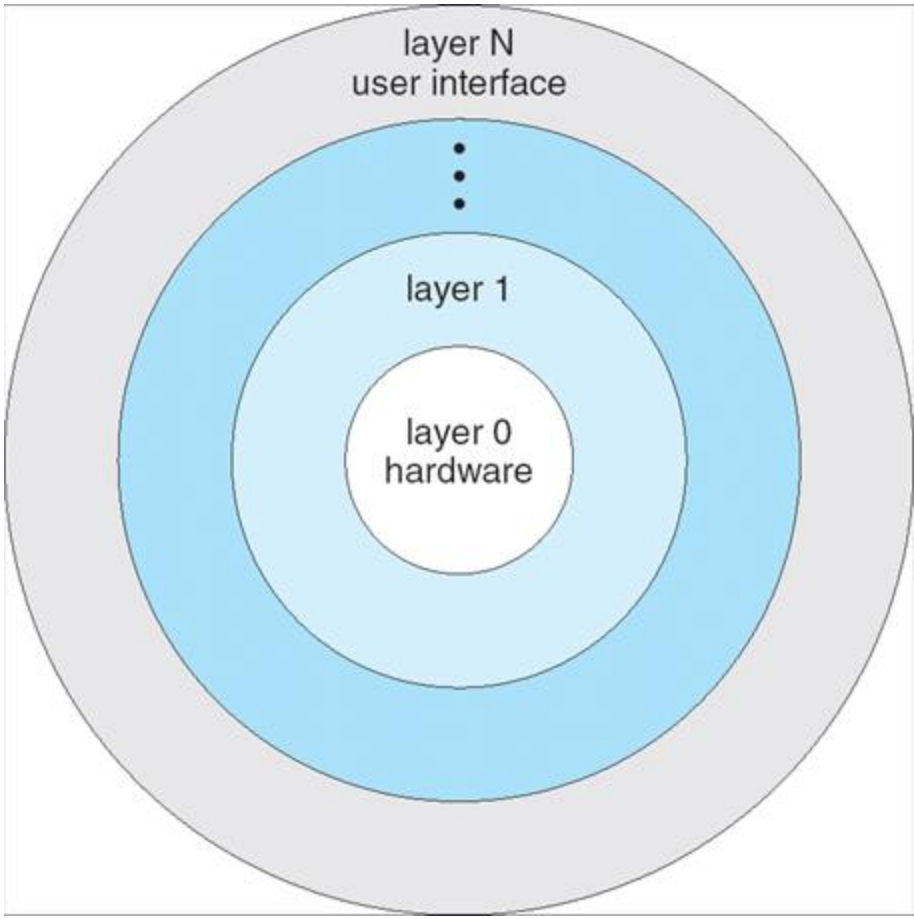
- Load program
  - "shrink" interpreter
  - make room for program
- Execute program
  - access to everywhere
  - even "kernel"/interpreter
- Reload interpreter back
  - otherwise, "cannot find command.com…"



* more details at memory management

Layered structure

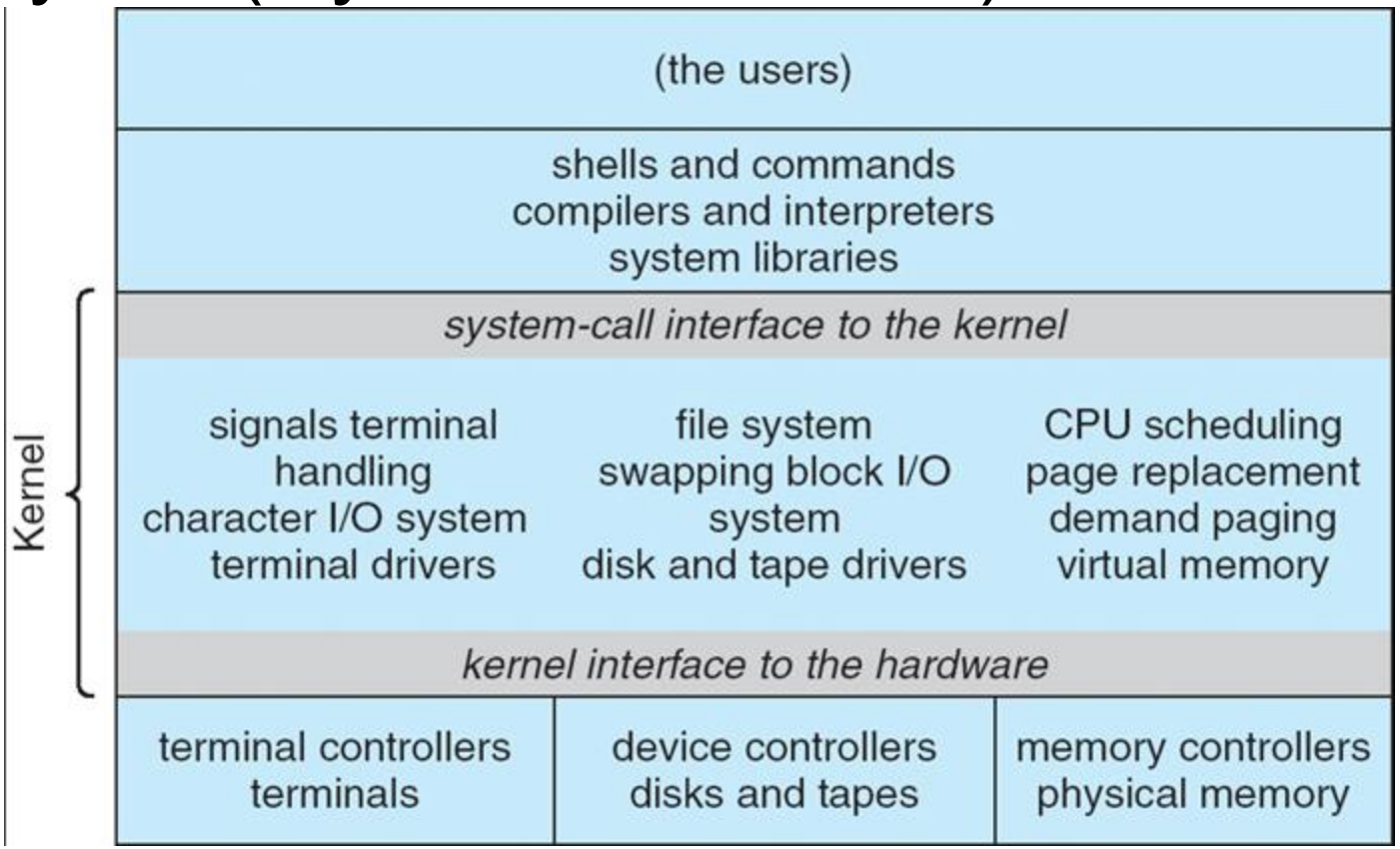
- OS Layers
  - L_0: hardware
  - L_N: user interface
  - L_i: anything in btw
    - use L_{i-1} service
    - offer service to L_{i+1}
- Divide & conquer
- Cross-layer issues



* x86 CPU ring 0: kernel mode; ring 1/2: device drivers; 3: apps

Unix

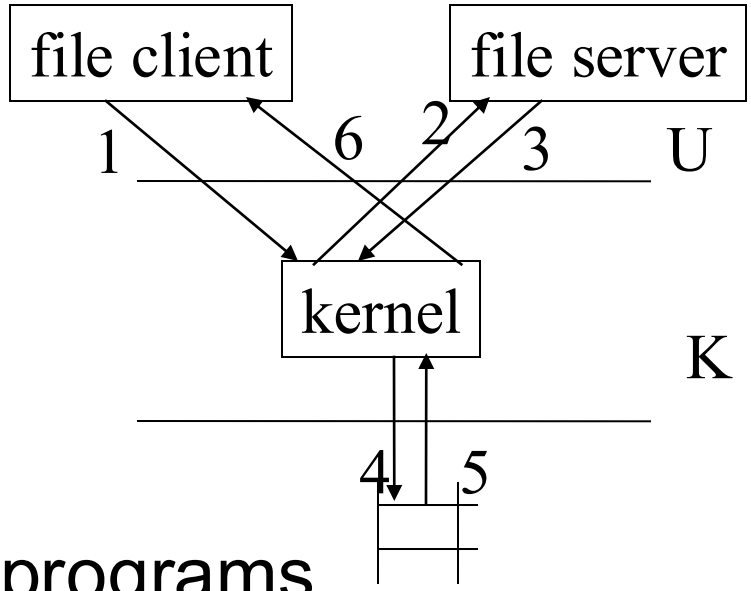
- Hybrid (layered+monolithic) structure



Q: pros and cons

Microkernel structure

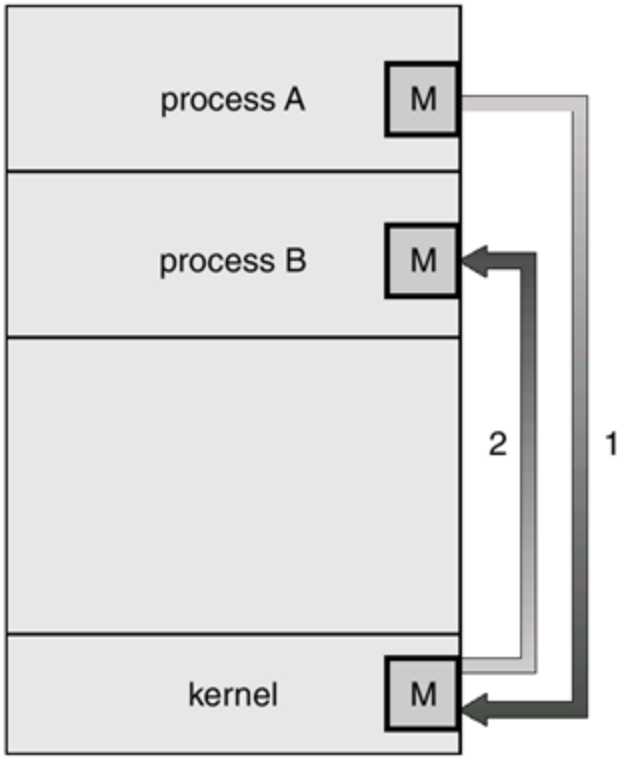
- E.g.
  - CMU Mach
- Smaller kernel
  - only those “essentials”
  - e.g., handle hardware
- More by system/application programs
  - message passing
- Overhead between kernel and user spaces

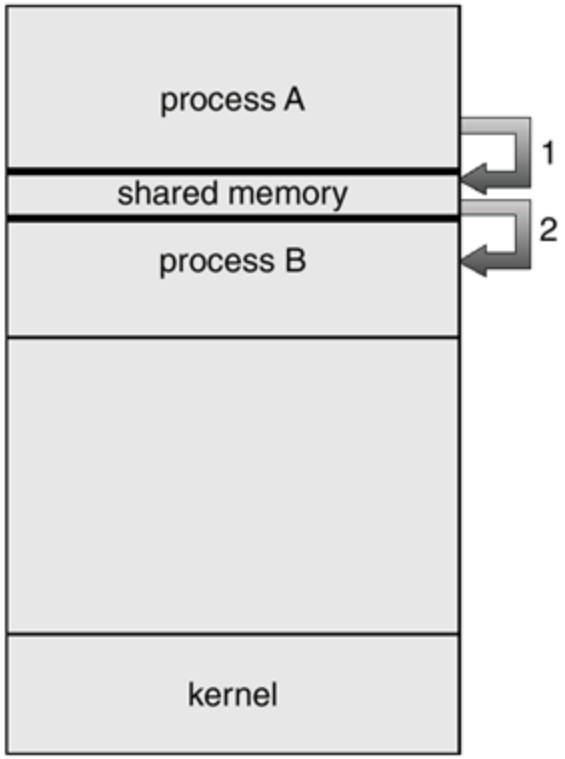


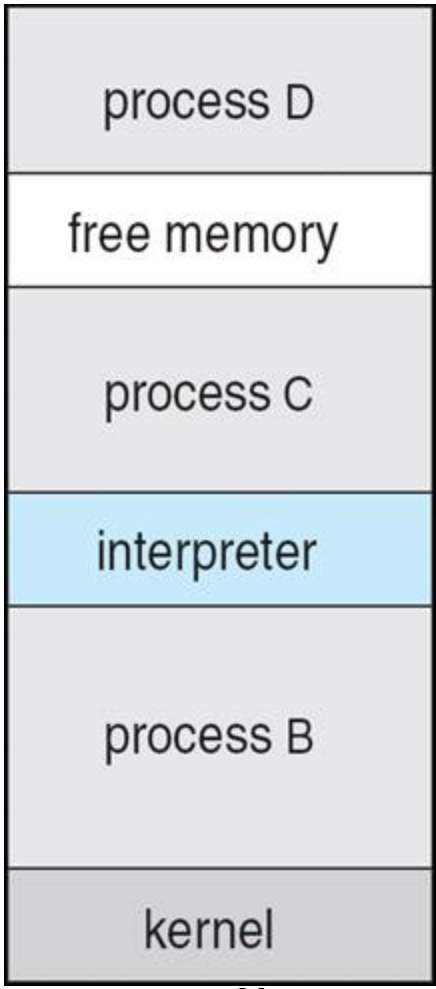
Q: pros and cons

Process communication

- Message passing vs shared memory



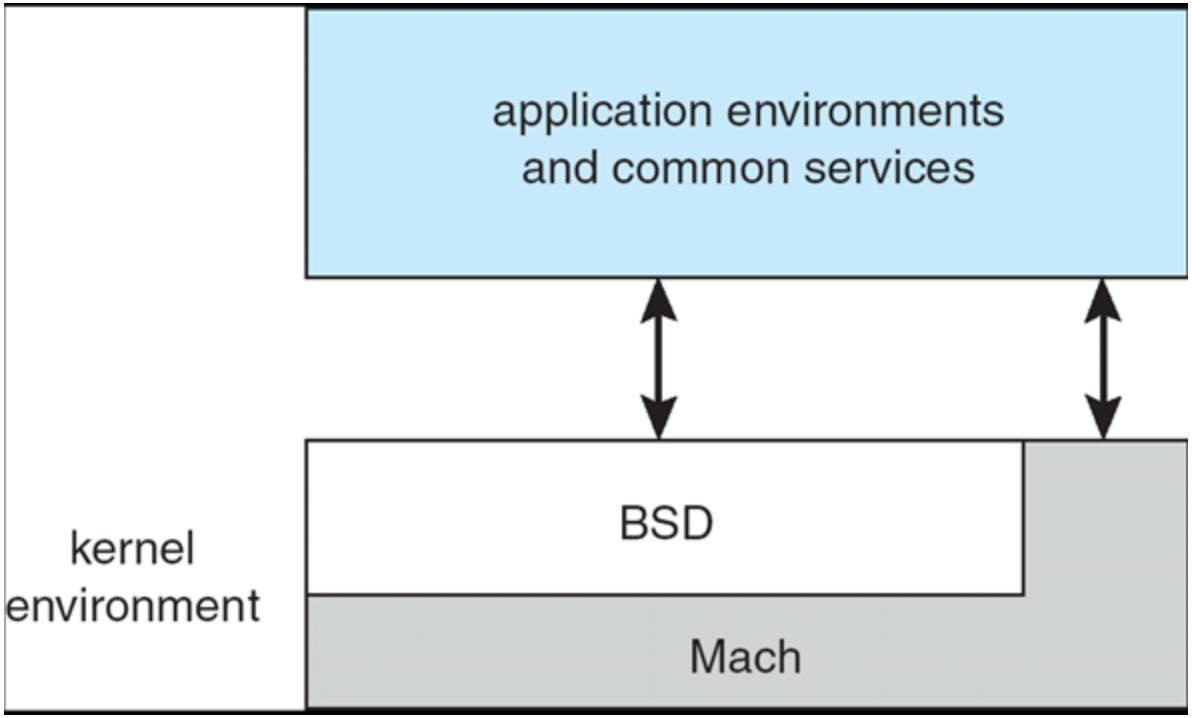




Q: pros and cons

Mac OS X

- Mach (CPU,memory) + BSD (file,network)



harmonyOS?

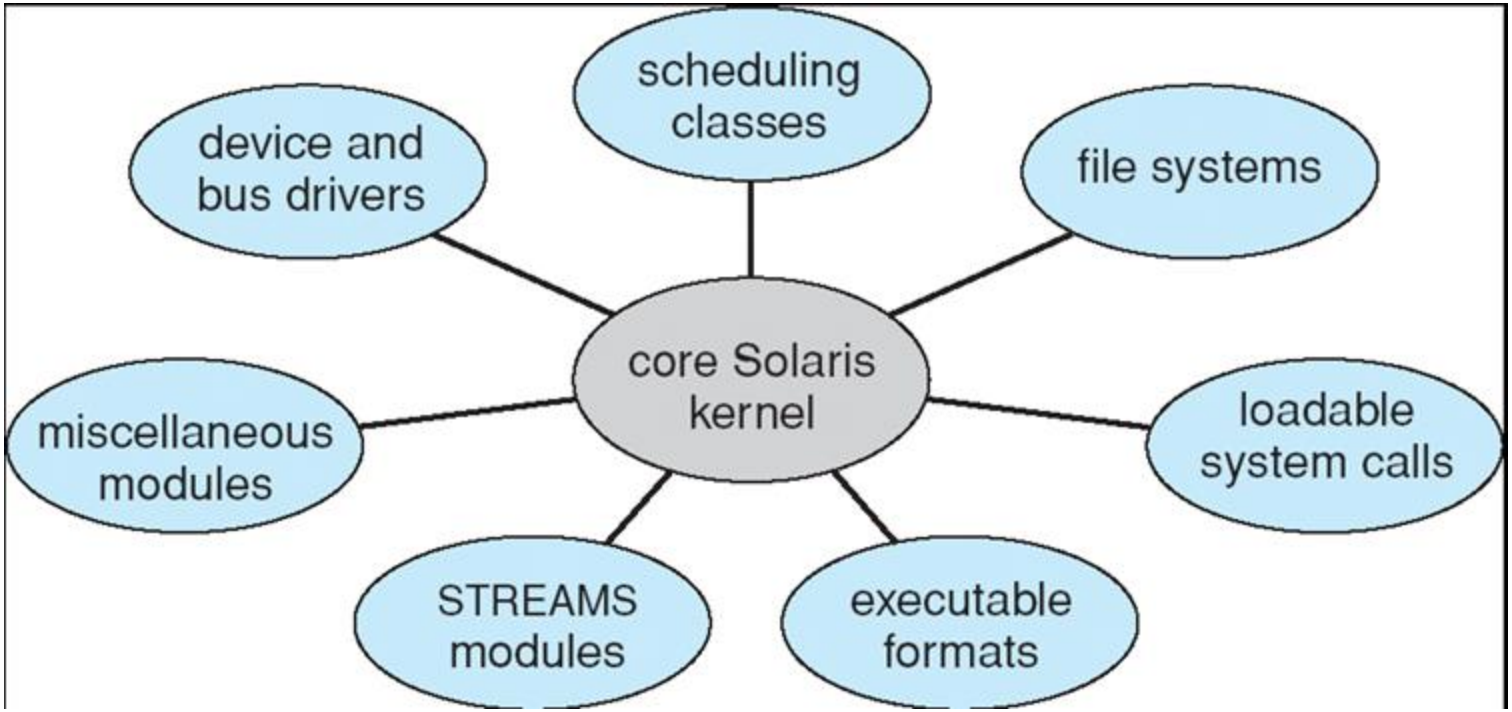
Modular structure

- Object-oriented methodology
  - not necessarily implemented in OO languages
  - popular choices for modern OS, e.g., Linux
  - e.g., insmod fat|vfat|msdos
- On-demand, loadable kernel modules
  - each module is a separate function/support
  - communicate through the known kernel interface
  - module dependency

Q: pros and cons

SunOS Solaris

- Modular design (high-level diagram)



* Linux loadable modules

The 2nd half of this lecture

- OS structures
  - design and implementation tradeoffs
    - user requirement
    - hardware support
  - layered, micro-kernel, modular
    - pros and cons
- Explore further
  - which OS structures are good for embedded system, I/O or computation-intensive system?
  - from power-on boot-up to login:

Next lecture

- Process management
  - Process: concepts
  - read OSC7/8/9/10 Chapter 3 (Processes)
  - (or OSC6 Chapter 4)