Oview of Operating Systems (2)

Dr. Jun Zheng

CSE325 Principles of Operating Systems

8/26/2019

NEW MEXICO TECH

SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

File systems

- ❑ Secondary storage devices are crude and awkward
  - ❑ e.g., "write a 4096 byte block to sector 12"
- ❑ File system: a convenient abstraction
  - ❑ defines logical objects like files and directories
    - ❑ hides details about where on disk files live
  - ❑ as well as operations on objects like read and write
    - ❑ read/write byte ranges instead of blocks
- ❑ A file is the basic unit of long-term storage
  - ❑ file = named collection of persistent information
- ❑ A directory is just a special kind of file
  - ❑ directory = named file that contains names of other files and metadata about those files (e.g., file size)
- ❑ Note: Sequential byte stream is only one possibility!

File system operations

- The file system interface defines standard operations:
  - file (or directory) creation and deletion
  - manipulation of files and directories (read, write, extend, rename, protect)
  - copy
  - lock
- File systems also provide higher level services
  - accounting and quotas
  - backup (must be incremental and online!)
  - (sometimes) indexing or search
  - (sometimes) file versioning

Protection

- Protection is a general mechanism used throughout the OS
  - all resources needed to be protected
    - memory
    - processes
    - files
    - devices
    - CPU time
    - …
  - protection mechanisms help to detect and contain unintentional errors, as well as preventing malicious destruction

Command interpreter (shell)

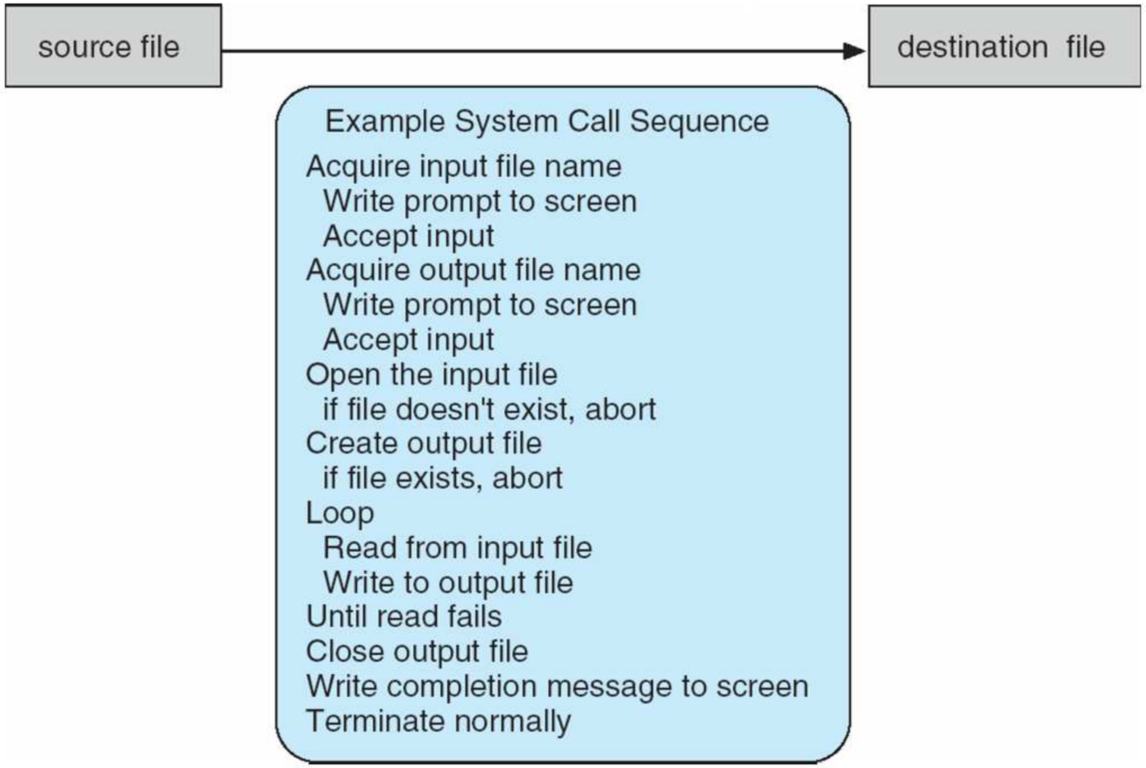
- A particular program that handles the interpretation of users' commands and helps to manage processes
  - user input may be from keyboard (command-line interface), from script files, or from the mouse (GUIs)
  - allows users to launch and control new programs
  - Sometimes multiple flavors implemented – **shells**
  - Sometimes commands built-in, sometimes just names of programs
    - If the latter, adding new features doesn't require shell modification

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface** (**API**) rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

Example of System Calls

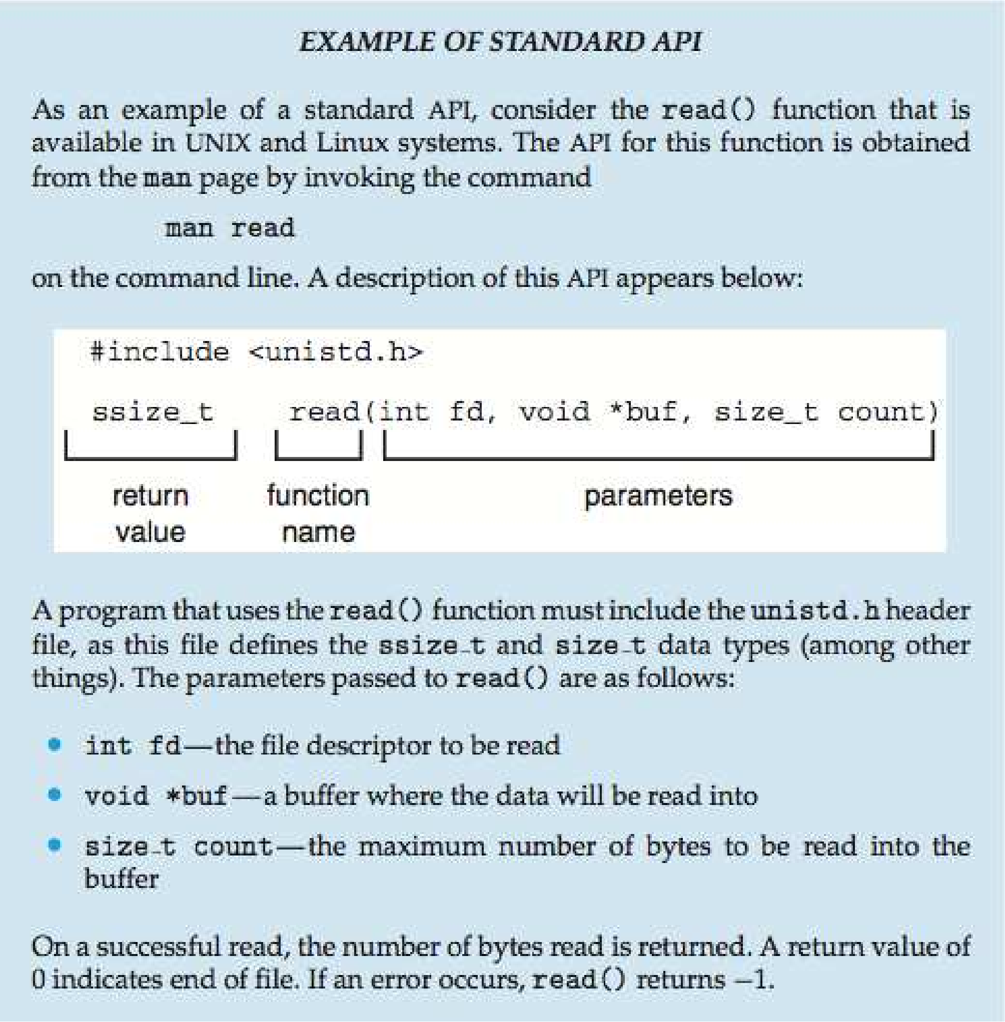
System call sequence to copy the contents of one file to another file



Example System Call Sequence

```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

Example of Standard API



EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t     read(int fd, void *buf, size_t count)
```

return value
function name
parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

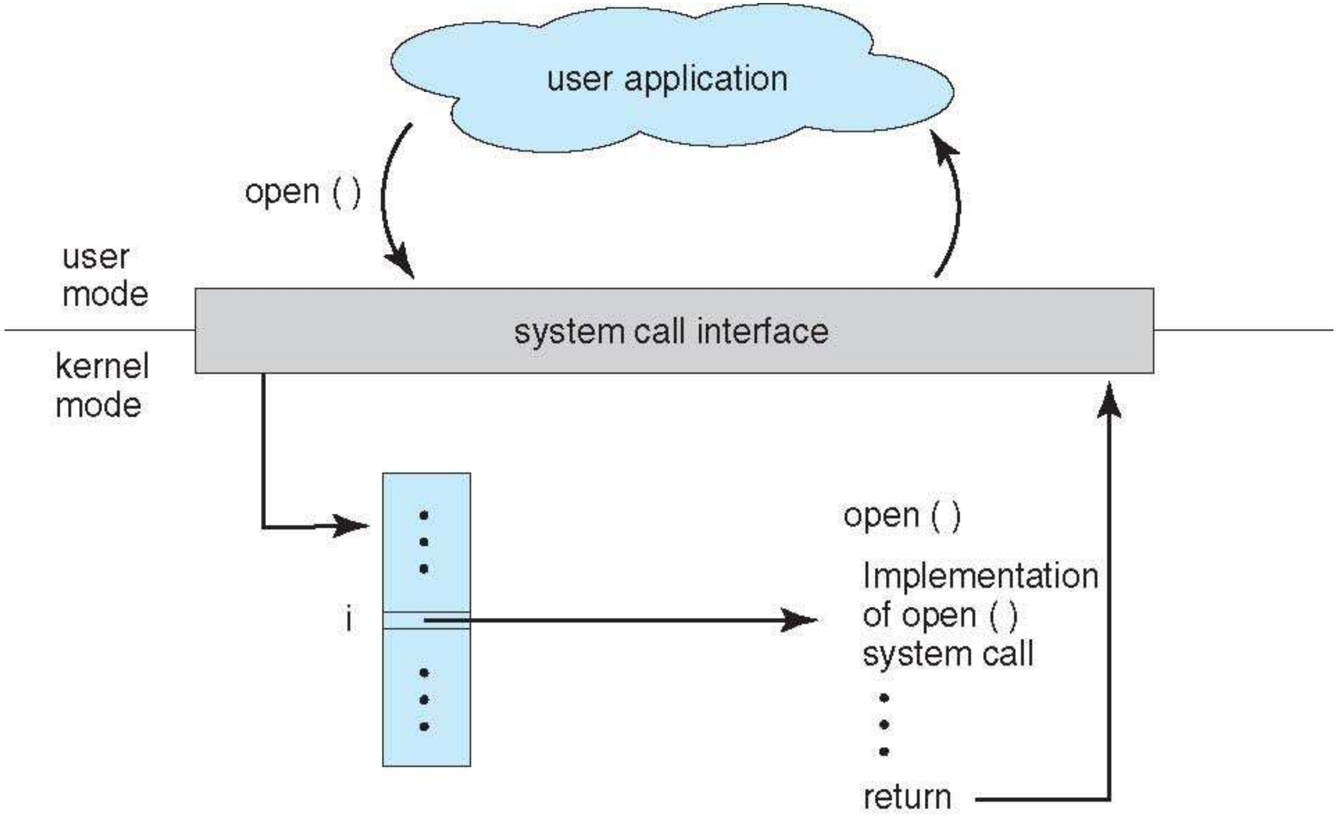
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns –1.

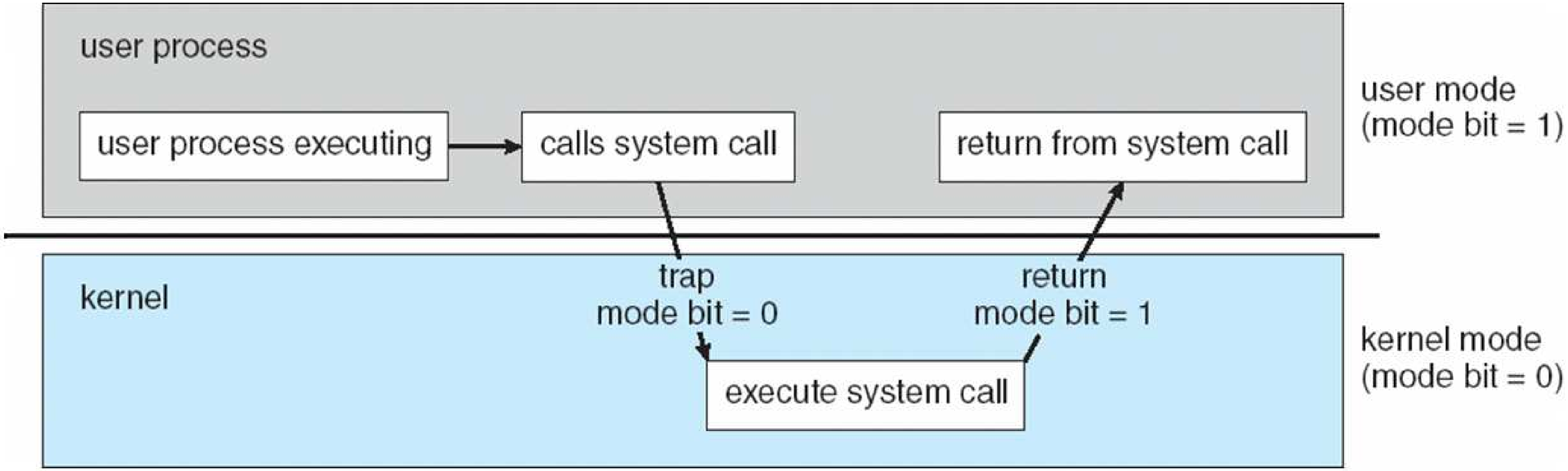
System Call Implementation

- Typically, a number associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship



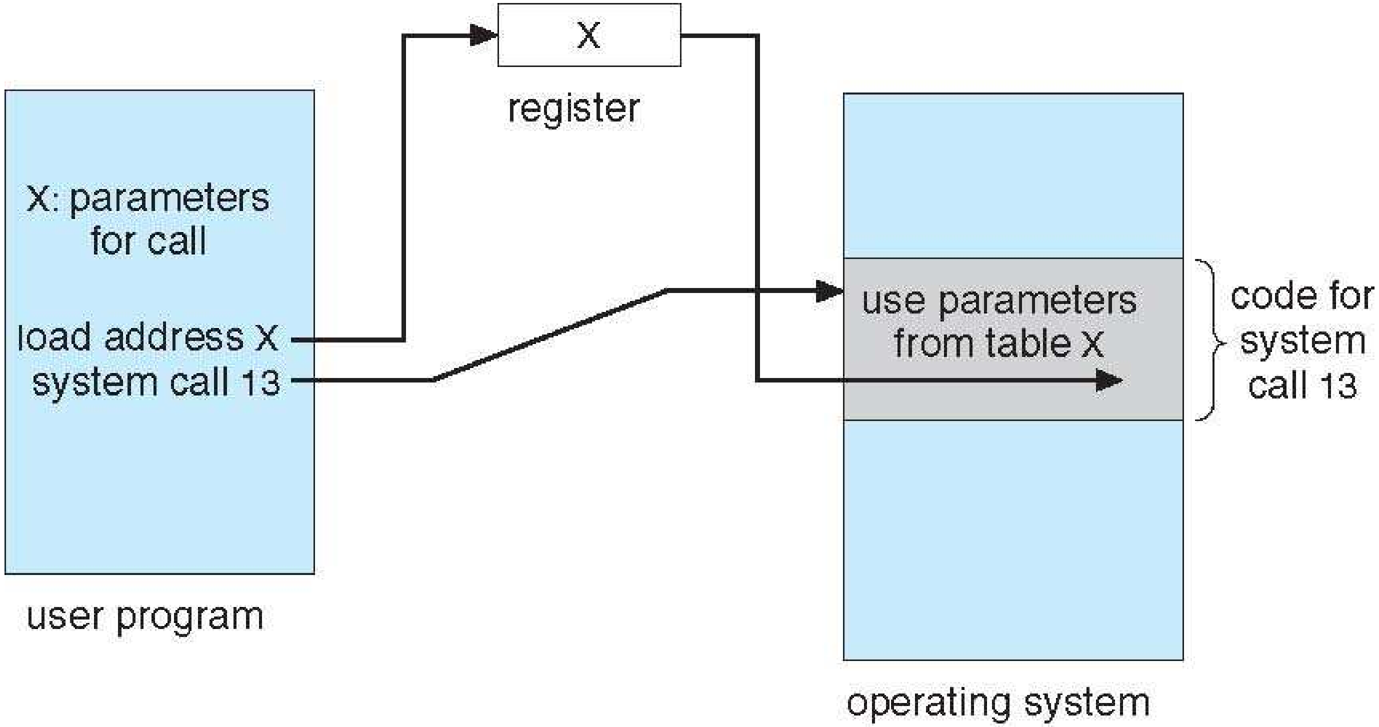
Mode Transfer



System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
    - In some cases, may be more parameters than registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table

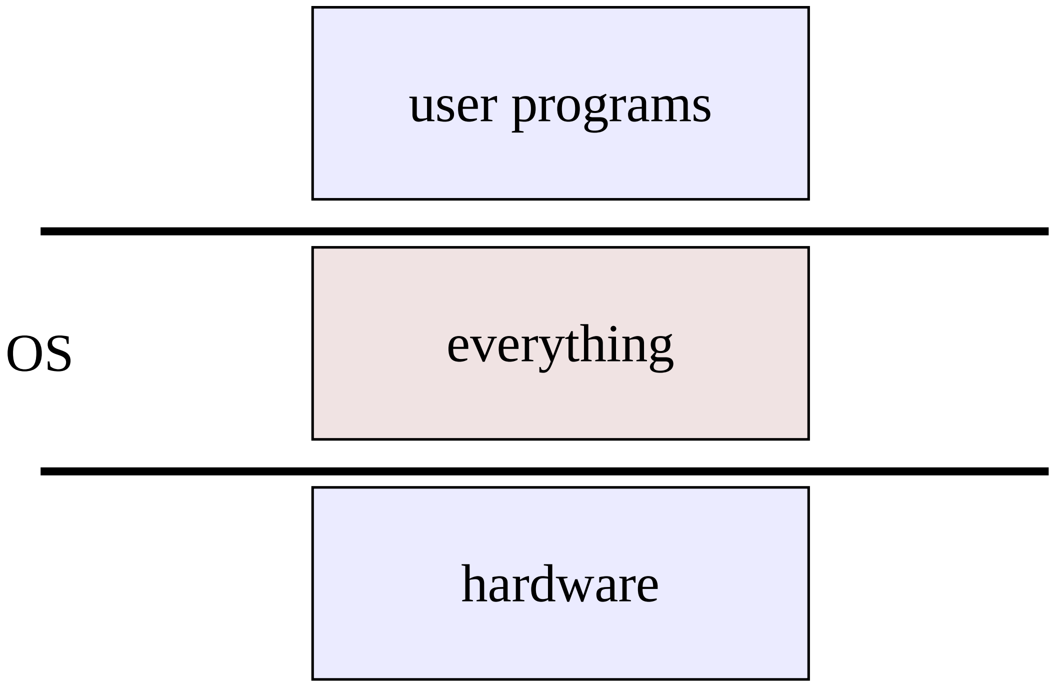


OS structure

- An OS consists of all of these components, plus:
  - many other components
  - system programs (privileged and non-privileged)
    - e.g., bootstrap code, the init program, …
- Major issue:
  - how do we organize all this?
  - what are all of the code modules, and where do they exist?
  - how do they cooperate?
- Massive software engineering and design problem
  - design a large, complex program that:
    - performs well, is reliable, is extensible, is backwards compatible, …

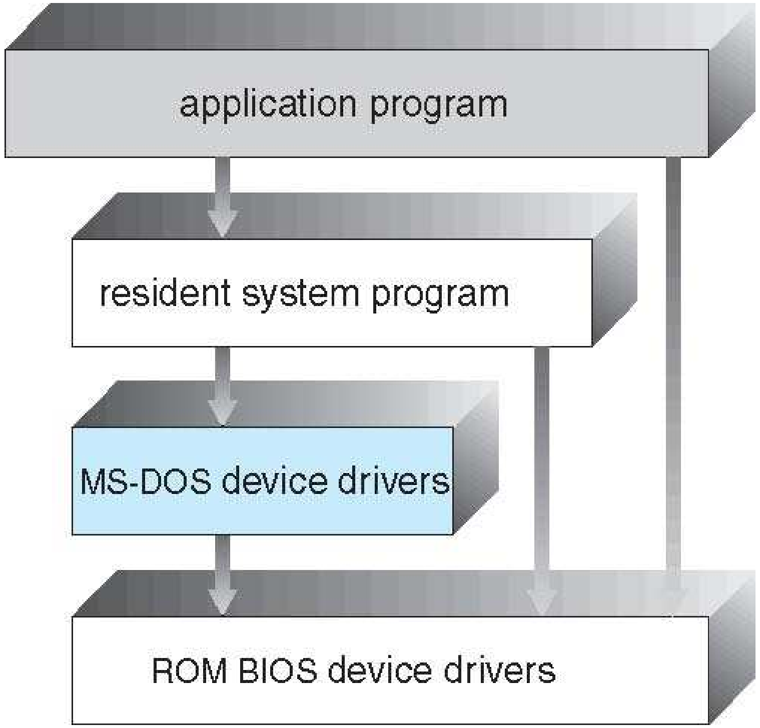
Eary Structure: Monolithic

- ❑ Traditionally, OS's are built as a monolithic entity:
  - ❑ Single linked binary
  - ❑ Any function can call any other function
  - ❑ Exmple: MS-DOS, xv6



MS-DOS System Structure

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



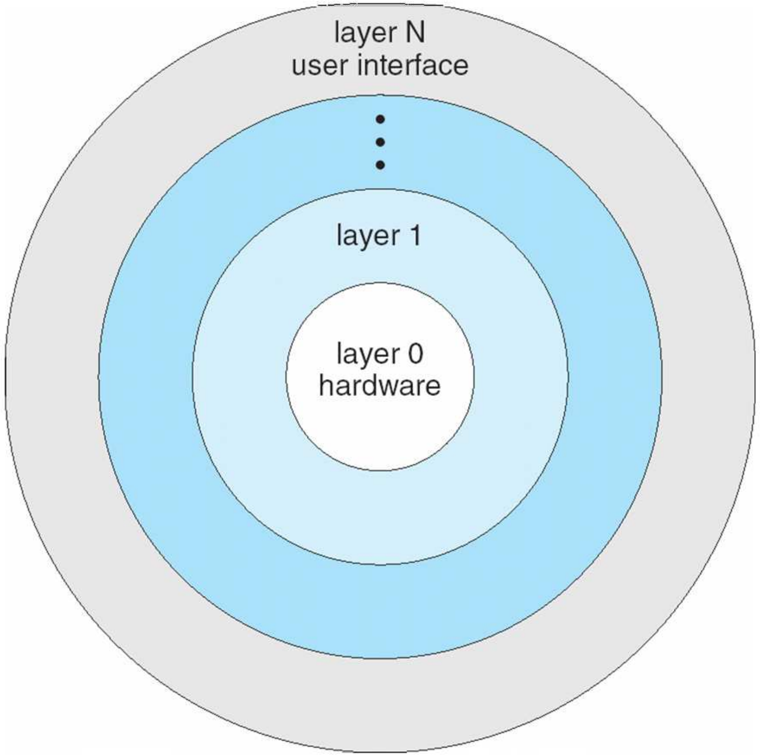
NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

Monolithic Structure

- ❑ Major advantage:
  - ❑ cost of module interactions is low (procedure call)
- ❑ Disadvantages:
  - ❑ As system scales, it becomes:
    - ❑ Hard to understand
    - ❑ Hard to modify
    - ❑ Hard to maintain
  - ❑ Unreliable (no isolation between system modules)
- ❑ What is the alternative?
  - ❑ Find a way to organize the OS in order to simplify its design and implementation

Layered Approach

- ❑ The operating system is divided into a number of layers (levels), each built on top of lower layers.
- ❑ The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- ❑ With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers
- ❑ Each layer can be tested and verified independently



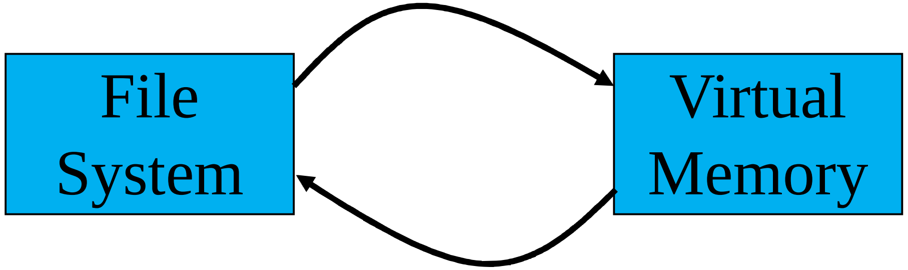
NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

THE System

- The first description of layered approach was Dijkstra’s THE system (1968!)
  - 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

Problems with Layering

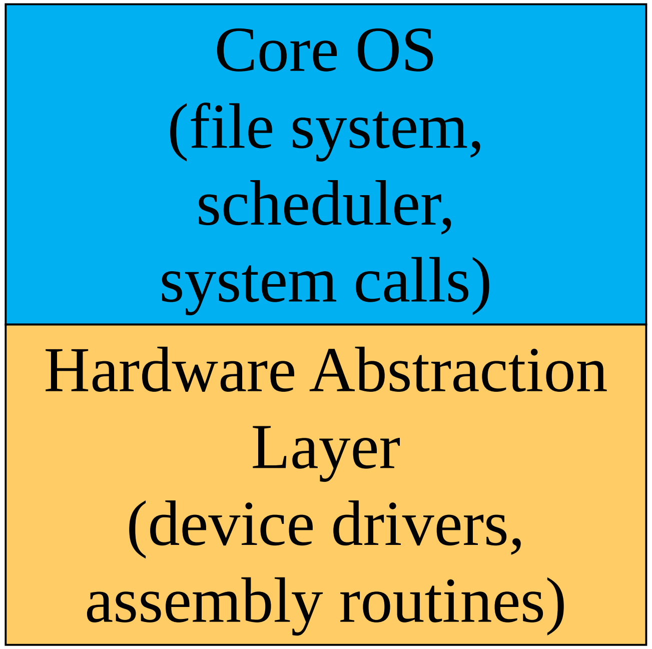
- ❑ Strict hierarchical structure is too inflexible
  - ❑ Real systems have “uses” cycles
    - ❑ File system requires virtual memory services (buffers)
    - ❑ Virtual memory would like to use files for its backing store



- ❑ Poor performance
  - ❑ Each layer crossing has overhead associated with it

Hardware Abstraction Layer

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

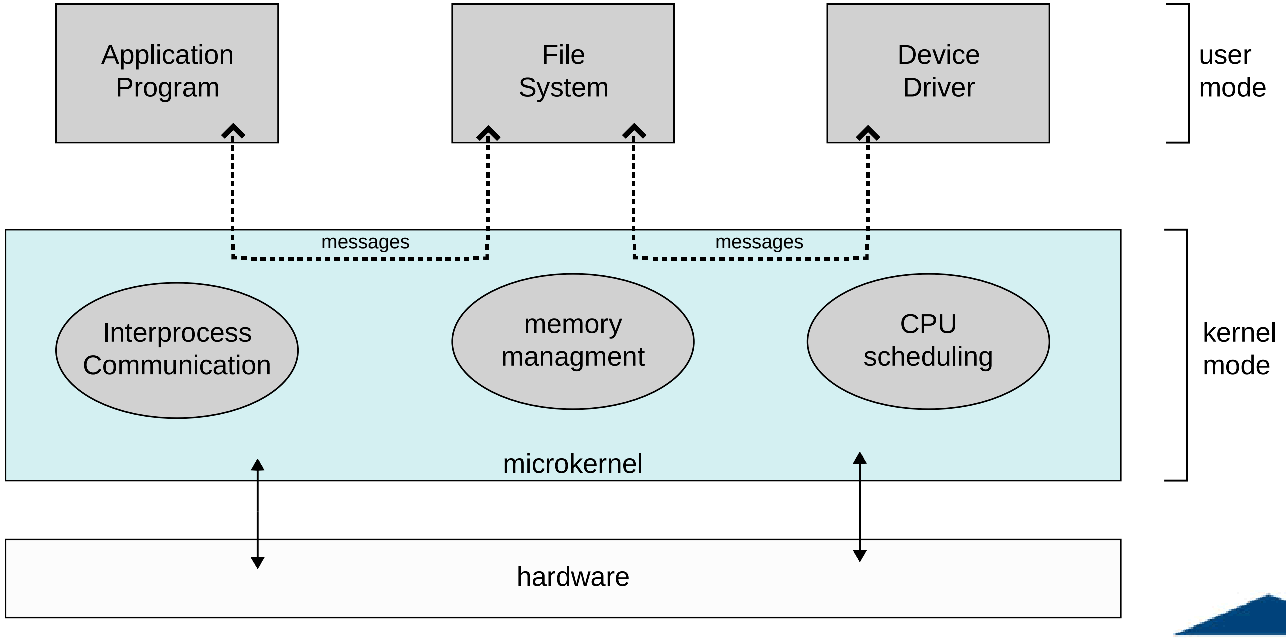


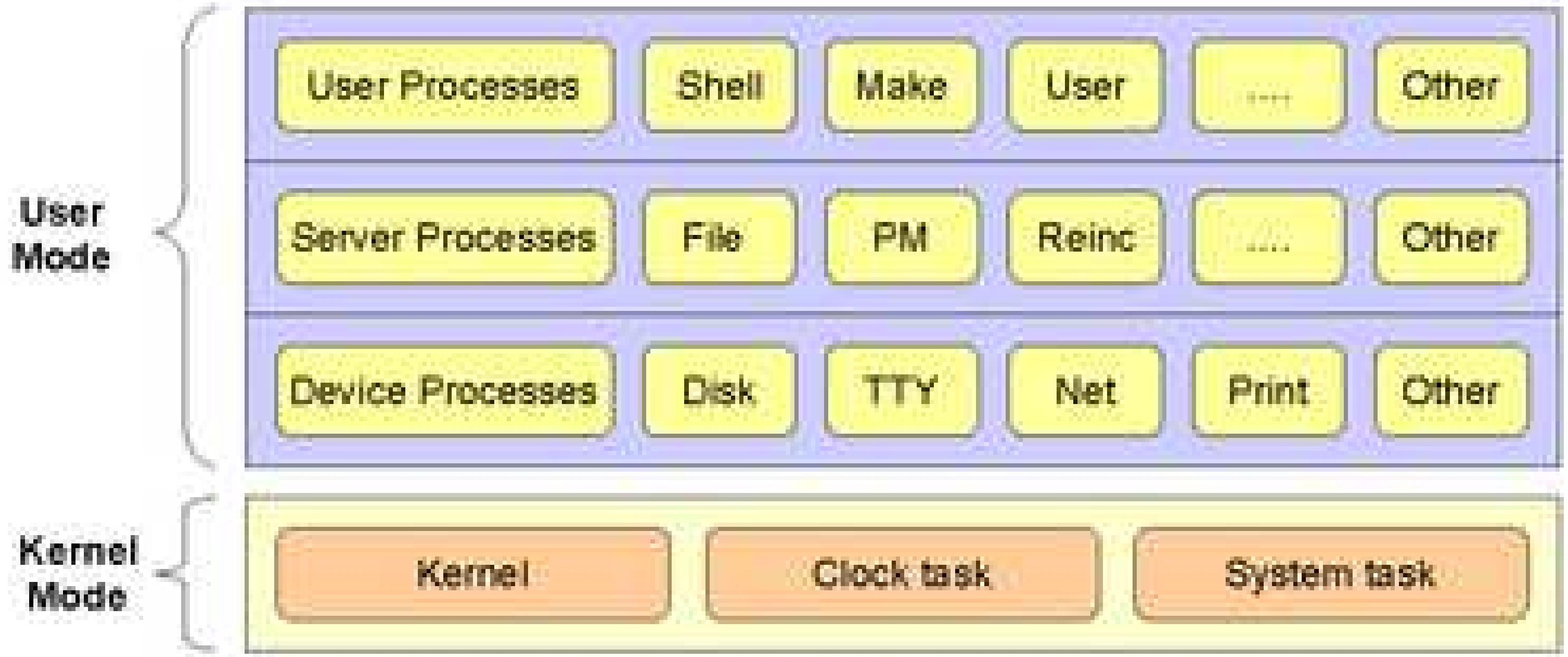
Core OS
(file system,
scheduler,
system calls)

Hardware Abstraction
Layer
(device drivers,
assembly routines)

Microkernel System Structure

- ❑ Moves as much from the kernel into user space
- ❑ **Mach:** example of **microkernel**
  - ❑ Mac OS X kernel (**Darwin**) partly based on Mach
- ❑ Communication takes place between user modules using **message passing**





User Mode

User Processes | Shell | Make | User | .... | Other

Server Processes | File | PM | Reinc | .... | Other

Device Processes | Disk | TTY | Net | Print | Other

Kernel Mode

Kernel | Clock task | System task

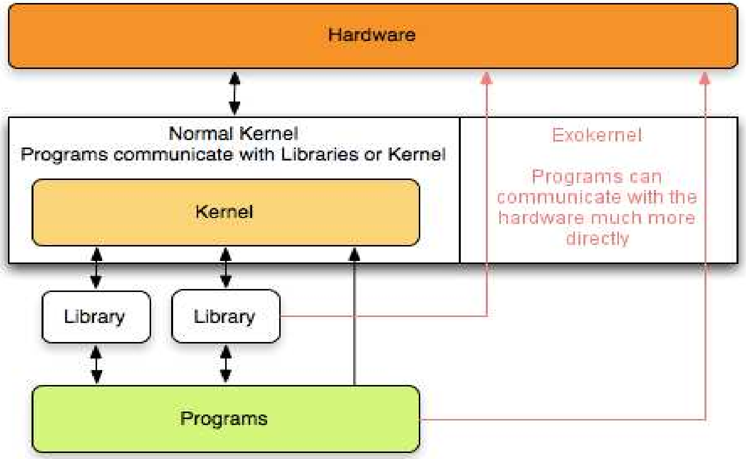
The MINIX 3 Microkernel Architecture

Microkernels: Pros and Cons

- Pros
  - Simplicity
    - Core kernel is very small
  - Extensibility
    - Can add new functionality in user-mode code
  - Reliability
    - OS services confined to user-mode programs
- Cons
  - Poor performance
    - Message transfer operations instead of system call
- Tanenbaum–Torvalds debate
  - https://en.wikipedia.org/wiki/Tanenbaum%E2%80%93Torvalds_debate

Exokernel

- ❑ Similar to microkernel in that only minimum functionalit y is in the kernel.
- ❑ Unlike the microkernel it exports hardware resources rat her than emulating them.
- ❑ Physical resources are safely allocated to the application, where it can be managed.
- ❑ All abstractions are implemented in application level or as part of a library OS that is part of the applicatio n address space.
- ❑ Example: JOS

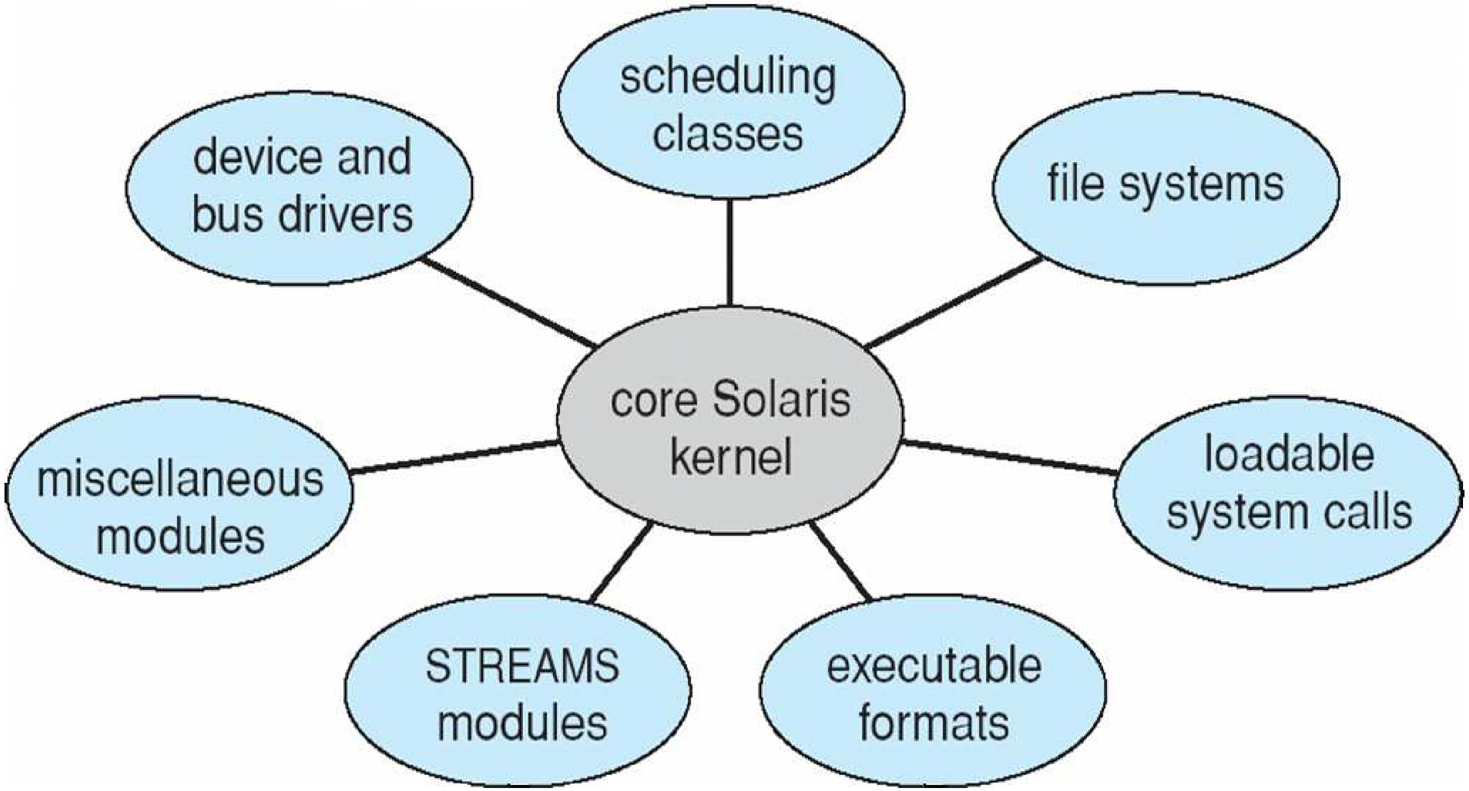

Hardware
Normal Kernel
Programs communicate with Libraries or Kernel
Kernel
Exokernel
Programs can communicate with the hardware much more directly
Library
Library
Programs

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

Modules

- ❑ Many modern operating systems implement **loadable kernel modules**
  - ❑ Uses object-oriented approach
  - ❑ Each core component is separate
  - ❑ Each talks to the others over known interfaces
  - ❑ Each is loadable as needed within the kernel
- ❑ Overall, similar to layers but with more flexible
  - ❑ Linux, Solaris, etc

Solaris Modular Approach



Hybrid Systems

- ❑ 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, **Aqua** UI plus **Cocoa** programming environment
  - ❑ Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)