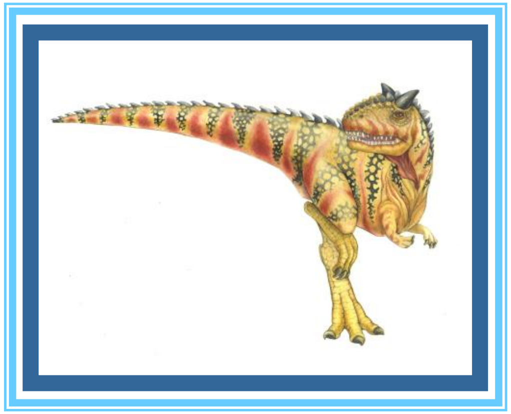
Chapter 2: Operating-System Structures



Chapter 2: Operating-System Structures

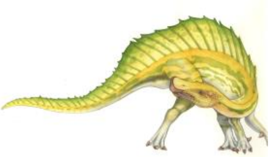
- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- Operating System Debugging
- Operating System Generation
- System Boot

Objectives

- To describe the services an operating system provides to users, processes, and other systems
- To discuss the various ways of structuring an operating system
- To explain how operating systems are installed and customized and how they boot

Operating System Services

- Operating systems provide an environment for execution of programs and services (helpful functions) to programs and users
- **User services:**
  - **User interface**
    - **No UI**, **Command-Line** (**CLI**), **Graphics User Interface** (**GUI**),   **Batch**
  - **Program execution** - Loading a program into memory and running it, end execution, either normally or abnormally (indicating error)
  - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device



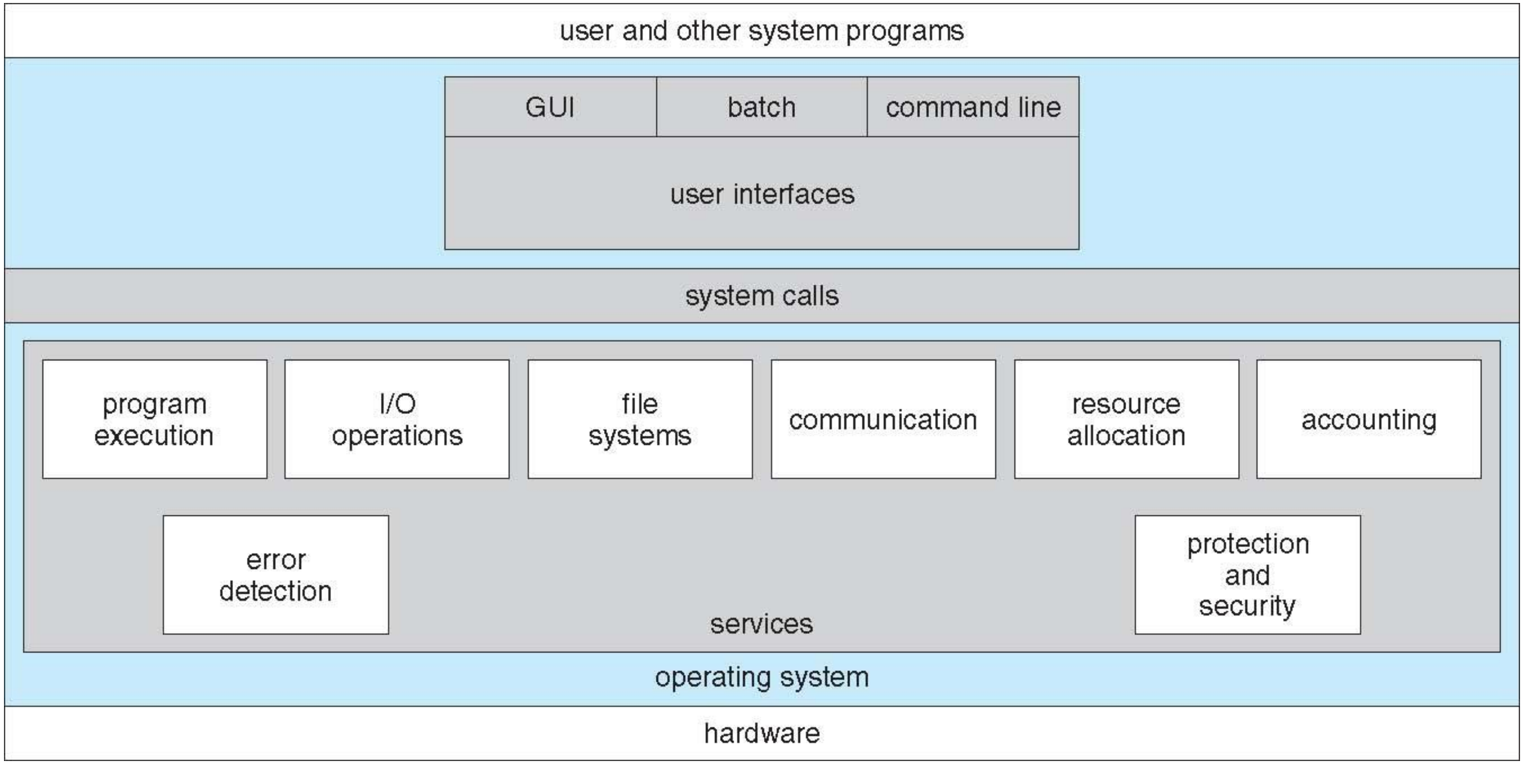
Operating System Services (Cont.)

- User services (Cont.):
  - **File-system manipulation** - Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
  - **Communications** – Processes may exchange information, on the same computer or between computers over a network
    - Communications may be via shared memory or through message passing (packets moved by the OS)
  - **Error detection** – OS needs to be constantly aware of possible errors
    - May occur in the CPU and memory hardware, in I/O devices, in user program
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
    - Debugging facilities can greatly enhance the user’s and programmer’s abilities to efficiently use the system

Operating System Services (Cont.)

- **System services:**
  - For ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - CPU cycles, main memory, file storage, I/O devices.
  - **Accounting -** To keep track of which users use how much and what kinds of computer resources
  - **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - **Protection** involves ensuring that all access to system resources is controlled
    - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

A View of Operating System Services



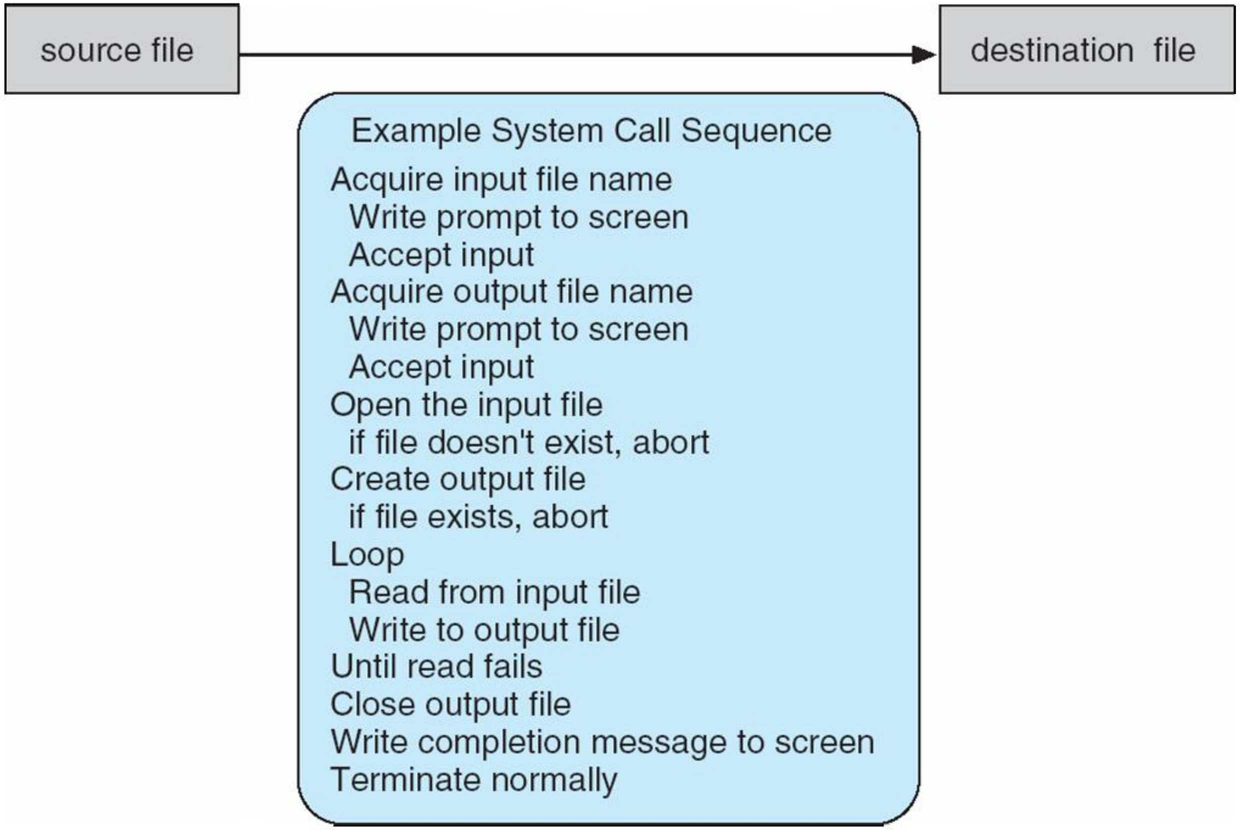
System Calls

- Systems 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,
  - Java API for the Java virtual machine (JVM)

Note that the system-call names used throughout this text are generic

Example of System Calls

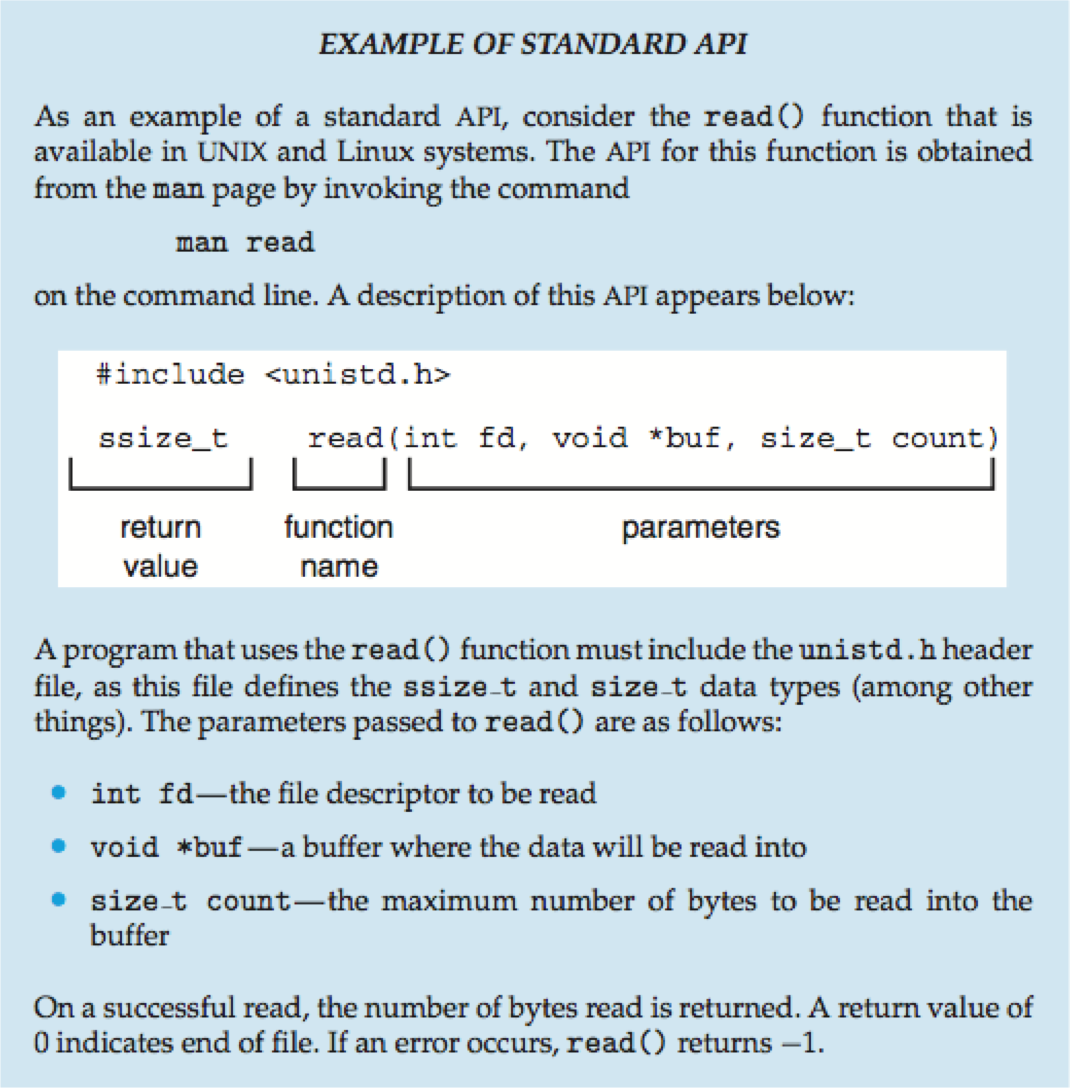
- System call sequence to copy the contents of one file to another file



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:

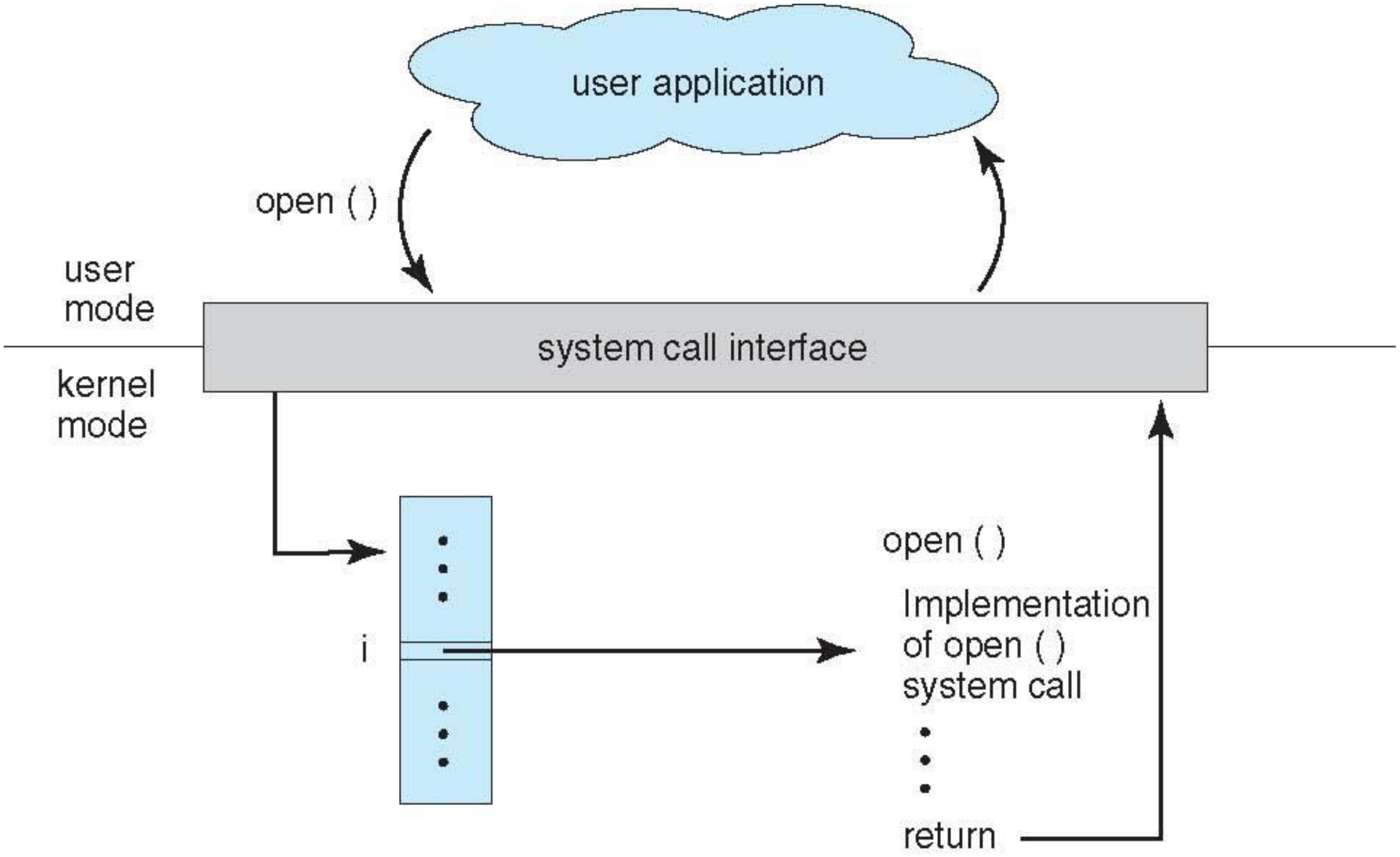
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

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)

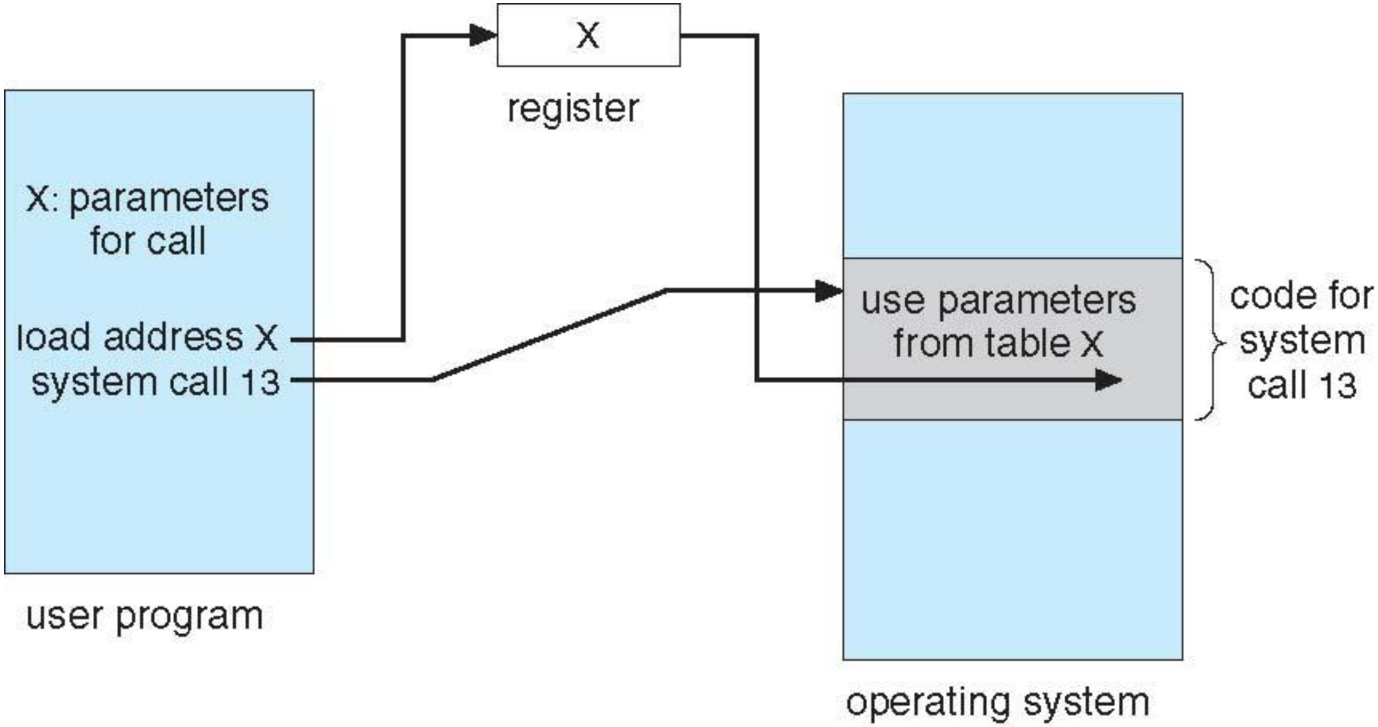
API – System Call – OS Relationship



System Call Parameter Passing

- Three general methods used to pass parameters to the OS in system calls
  - Simplest: 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



Types of System Calls

- Process control
  - create process, terminate process
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error
  - **Debugger** for determining **bugs, single step** execution
  - **Locks** for managing access to shared data between processes

Types of System Calls

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

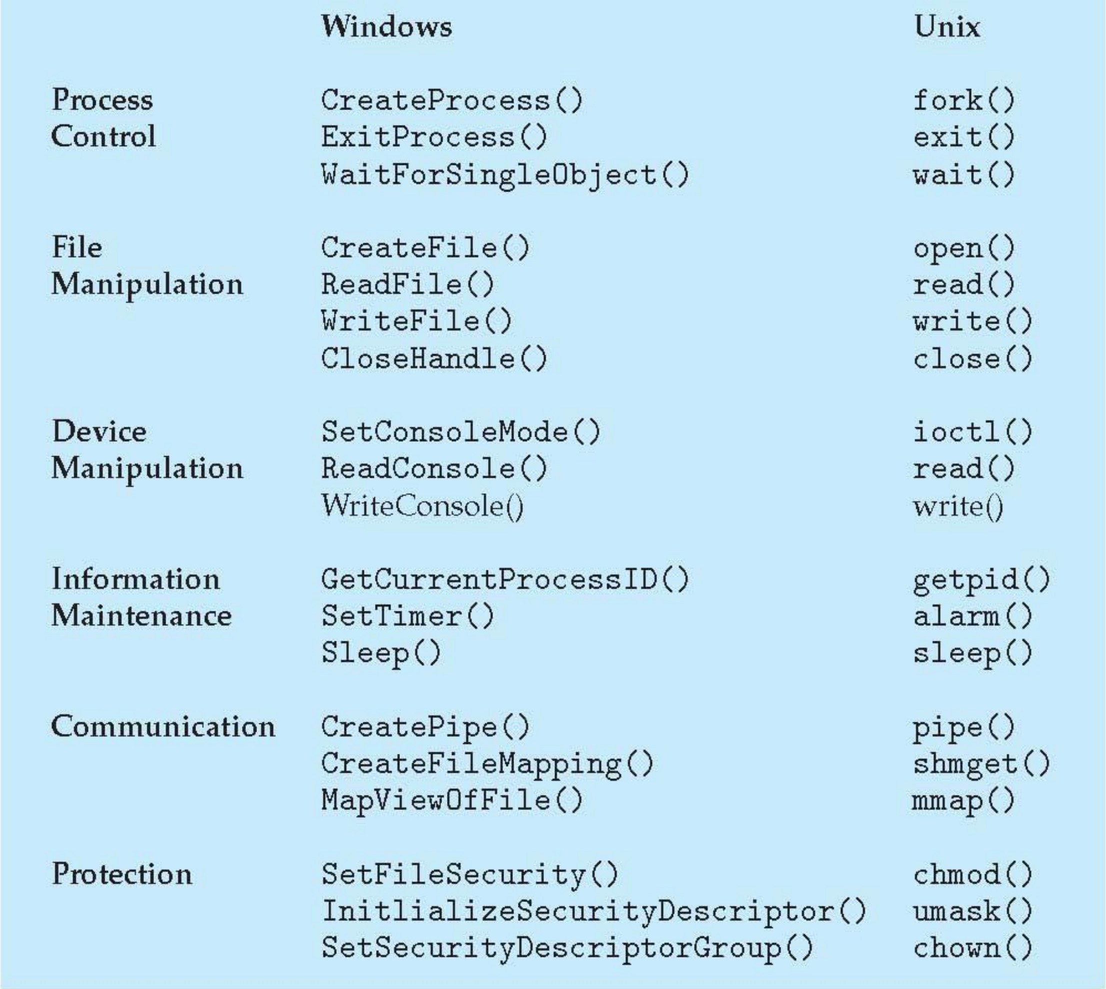
Types of System Calls (Cont.)

- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages if **message passing model** to **host name** or **process name**
    - From **client** to **server**
  - **Shared-memory model** create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices

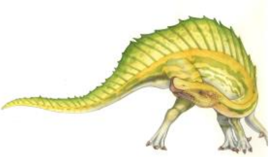
Types of System Calls (Cont.)

- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

Examples of Windows and Unix System Calls

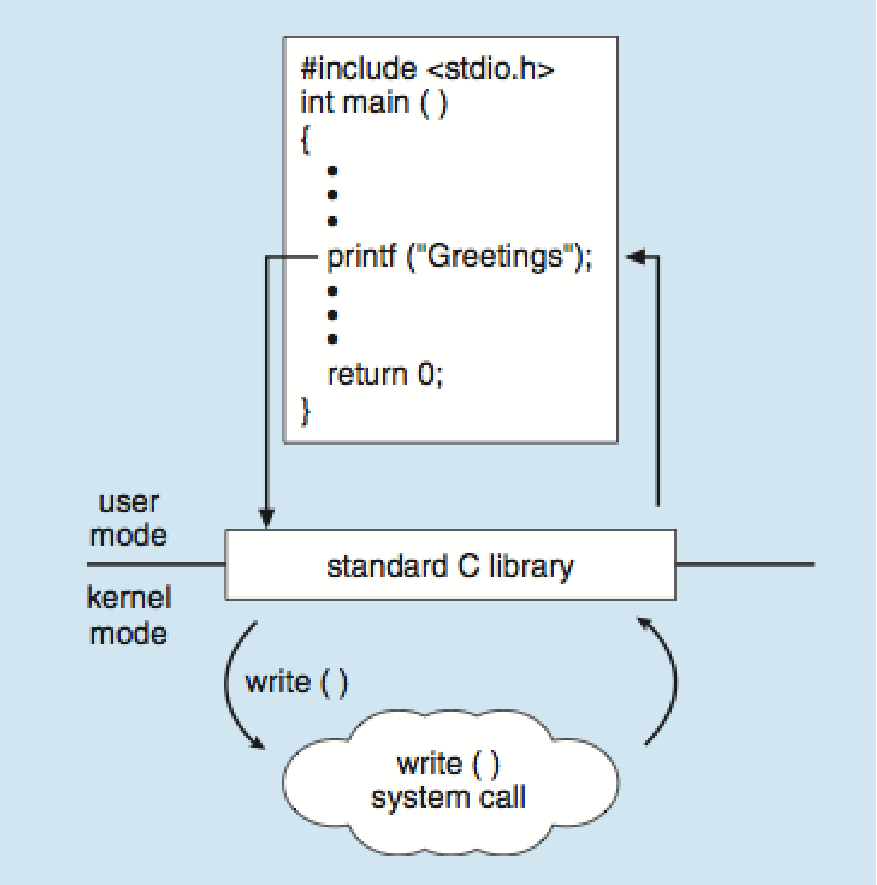


	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information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



Standard C Library Example

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



```
#include <stdio.h>
int main ( )
{
   •
   •
   •
   printf ("Greetings");
   •
   •
   •
   return 0;
}
```

System Programs

- System programs provide a convenient environment for program development and execution.
- Most users' view of the operation system is defined by system programs, not the actual system calls
- They can be divided into:
  - File manipulation
    - rm, ls, cp, mv, etc in Unix
  - Status information sometimes stored in a File modification
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
  - Application programs

Operating System Design and Implementation

- Design and Implementation of OS not “solvable”, but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start the design by defining goals and specifications
- Affected by choice of hardware, type of system
- **User** goals and **System** goals
  - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

Operating System Design and Implementation (Cont.)

- Important principle to separate

  Policy: ***What*** will be done?
  Mechanism: ***How*** to do it?

- Mechanisms determine how to do something, policies decide what will be done
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (example – timer)
- Specifying and designing an OS is highly creative task of **software engineering**

Implementation

- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to **port** to other hardware
  - But slower

Operating System Structure

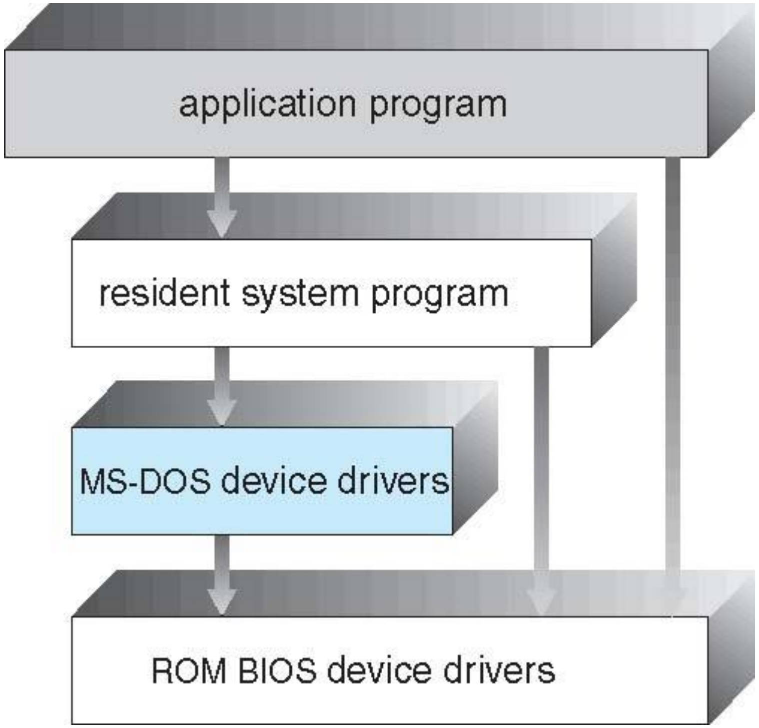
- General-purpose OS is a **very large program**
  - How to implement and structure it?
  - Can apply many ideas from software engineering
    - Software engineering - a separate area in CS
    - Studies design, development, and maintenance of software
- A common approach is to partition OS into modules/components
  - Each modules is responsible for one (or several) aspect of the desired functionality
  - Each module has carefully defined interfaces
  - **Advantages:**
    - Traditional advantages of modular programming:
      - Simplifying development and maintenance of computer programs, etc
      - Modules can be developed independently from each other
  - **Disadvantages:**
    - Efficiency can decrease (vs monolithic approach)

Operating System Structure (contd)

- In general, various ways are used to structure OSes
- Many OS’es don’t have well-defined structures
  - Not one pure model: Hybrid systems
  - Combine multiple approaches to address performance, security, usability needs
- Simple structure – MS-DOS
- More complex structure - UNIX
- Layered OSes
- Microkernel OSes

Simple Structure -- MS-DOS

- MS-DOS was created to provide the most functionality in the least space
- Not divided into modules
- MS-DOS has some structure
  - But its interfaces and levels of functionality are not well separated
- No dual mode existed for Intel 8088
  - MS-DOS was developed for 8088
  - Direct access to hardware is allowed
    - System crashes possible

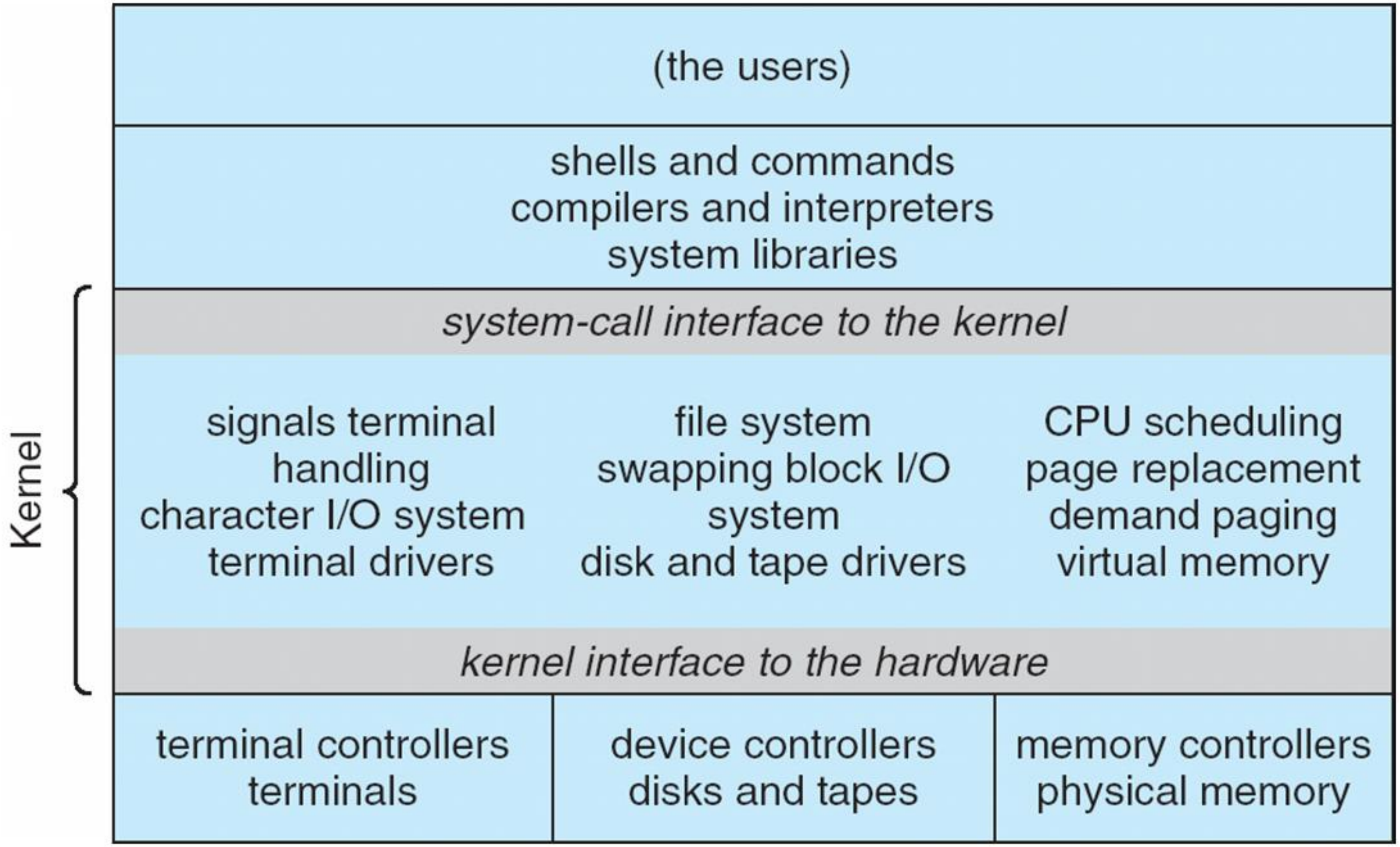


Non Simple Structure -- UNIX

- Traditional UNIX has limited structuring
- UNIX consists of 2 separable parts:
    1. Systems programs
    2. Kernel
- UNIX Kernel
    - Consists of everything that is
        - below the system-call interface and
        - above the physical hardware
    - Kernel provides
        - File system, CPU scheduling, memory management, and other operating-system functions
        - This is a lot of functionality for just 1 layer
        - Rather monolithic
            - But fast -- due to lack of overhead in communication inside kernel

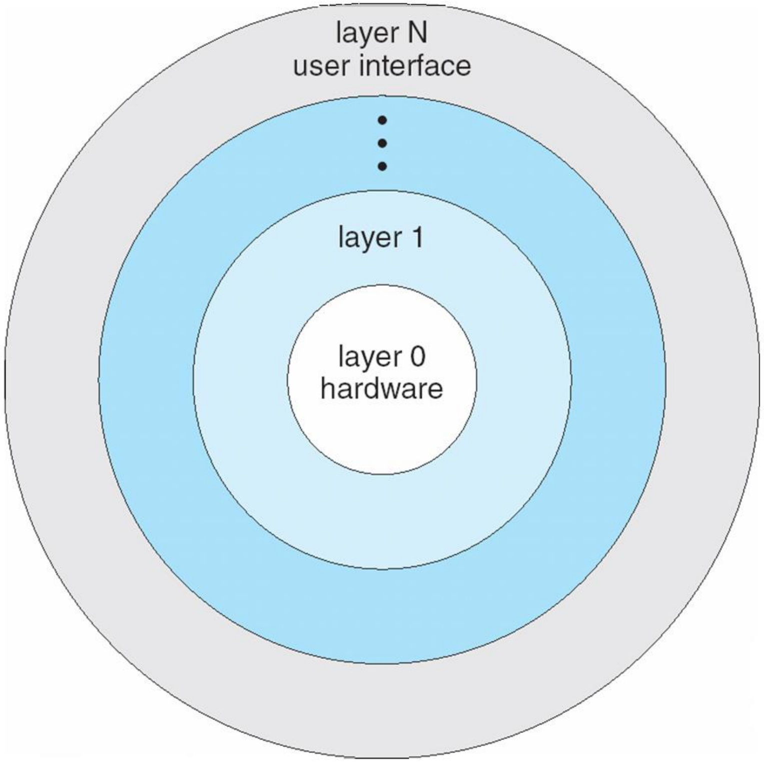
Traditional UNIX System Structure

Beyond simple but not fully layered



Layered Approach to structuring OS

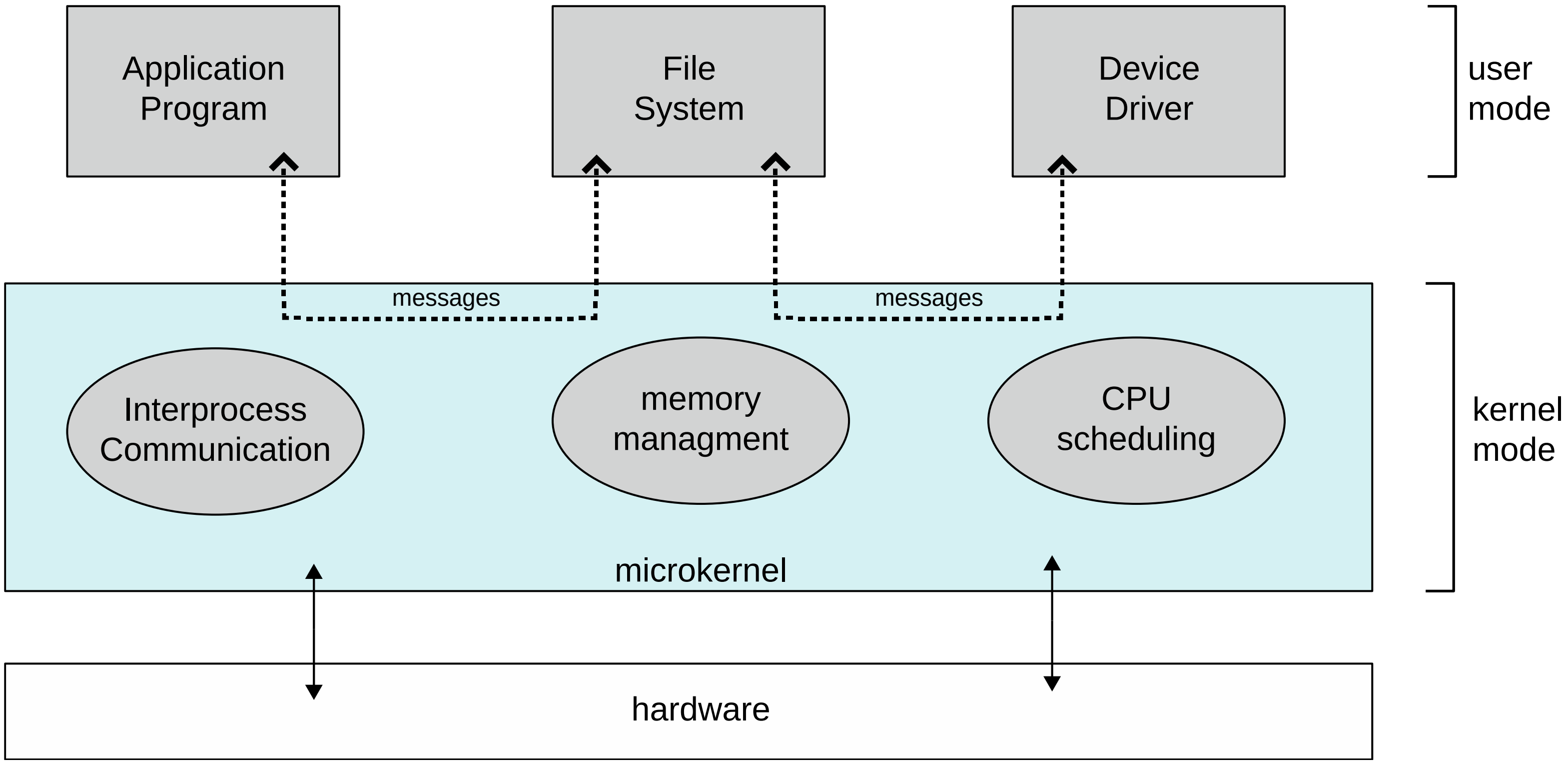
- One way to make OS modular – layered approach
- The OS is divided into a number of layers (levels)
- Each layer is built on top of lower layers
- The bottom layer (layer 0), is the hardware
- The highest (layer N) is the user interface
- Layers are selected such that each uses functions (operations) and services of only lower-level layers
- Advantages:
  - simplicity of construction and debugging
- Disadvantages:
  - can be hard to decide how to split functionality into layers
  - less efficient due to high overhead



Microkernel System Structure

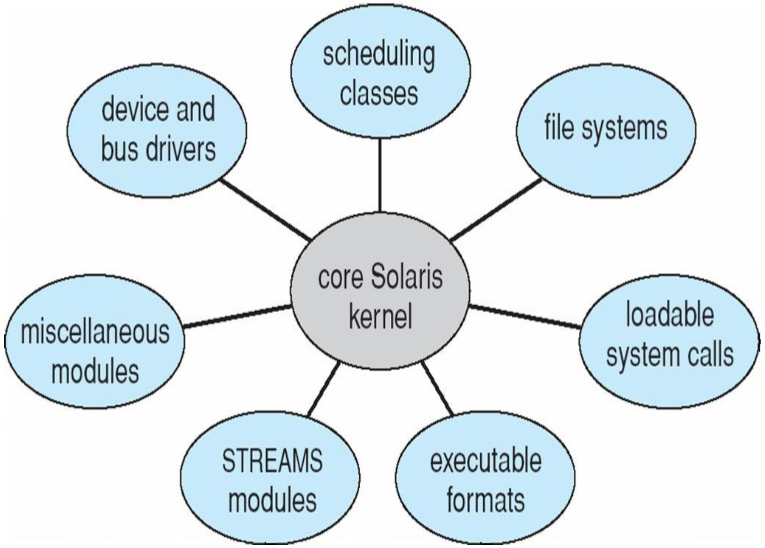
- Main idea:
  - Move as much from the kernel into the user space
  - Small core OS runs at the kernel level
  - OS services are built from many independent user-level processes
- Communication takes place between user modules using **message passing**
- Advantages:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Disadvantages:
  - Performance overhead of user space to kernel space communication

Microkernel System Structure



Modules

- Many modern operating systems implement **loadable kernel modules**
- Kernel provides only core services
  - The rest is via modules
  - Modules can be loaded as needed
    - Dynamic loading
    - Unloaded when not needed
  - Modules loaded into the kernel space
    - More efficient than microkernel solution
    - Does not use message passing
- More flexible than layered approach
  - Any module can call any other module
  - Calls are over known interfaces



Operating System Generation

- Operating systems are designed to run on any of a class of machines
- The system must be configured for each specific computer site
- The process of configuration is known as system generation **SYSGEN**
  - How to format partitions
  - Which hardware is present
  - Etc
- Used to build system-specific compiled kernel or system-tuned
- Can general more efficient code than one general kernel

System Boot

- How OS is loaded?
- When power is initialized on system, execution starts at a predefined memory location
  - Firmware ROM is used to hold initial bootstrap program (=bootstrap loader)
- Bootstrap loader
  - small piece of code
  - locates the kernel, loads it into memory, and starts it
- Sometimes 2-step process is used instead
  1. Simple bootstrap loader in ROM loads a more complex boot program from (a fixed location on) disk
  2. This more complex loader loads the kernel

End of Chapter 2