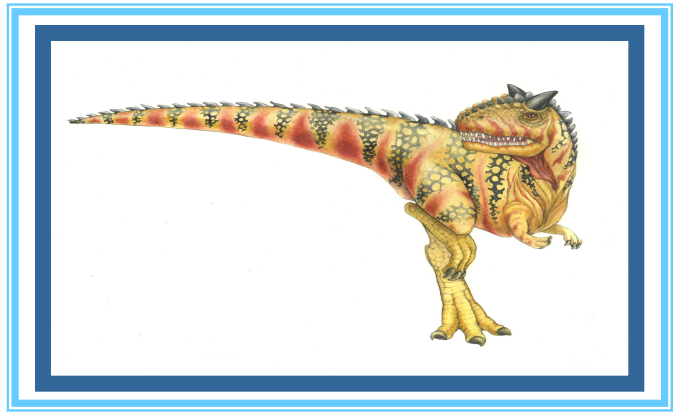
Chapter 2: System Structures



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 to programs and users
- One set of operating-system services provides functions that are helpful to the user:
  - **User interface** - Almost all operating systems have a user interface (**UI**).
    - Varies between **Command-Line (CLI)**, **Graphics User Interface (GUI)**, **Batch**
  - **Program execution** - The system must be able to load a program into memory and to run that program, 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
  - **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

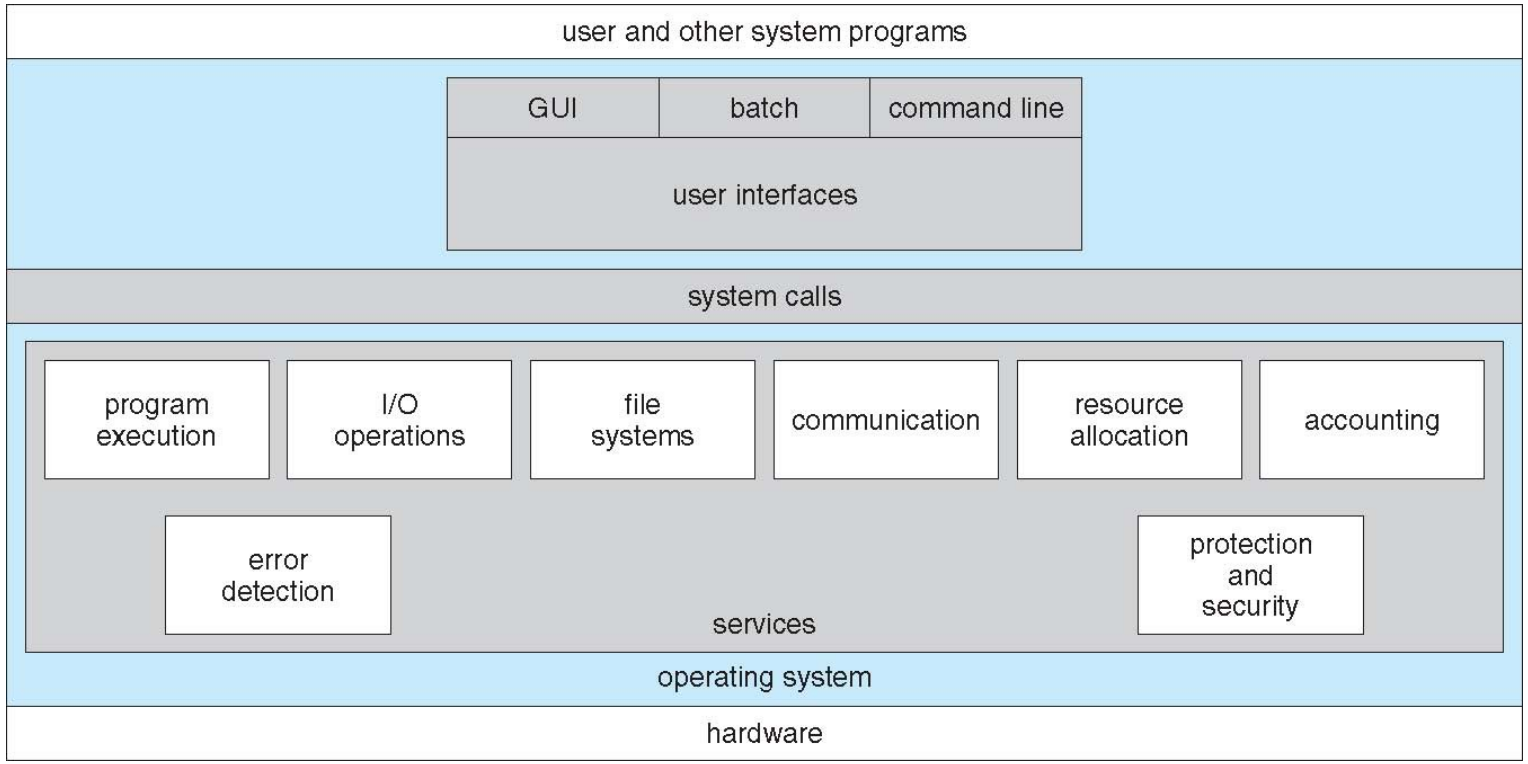
Operating System Services (Cont.)

- **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.)

- Another set of OS functions exists 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 - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code
  - **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
    - If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

A View of Operating System Services



User Operating System Interface - CLI

- CLI or **command interpreter** allows direct command entry
  - Sometimes implemented in kernel, sometimes by systems program
  - Sometimes multiple flavors implemented – **shells**
  - Primarily fetches a command from user and executes it
    - Sometimes commands built-in, sometimes just names of programs
      - If the latter, adding new features doesn't require shell modification

UNIX Shell

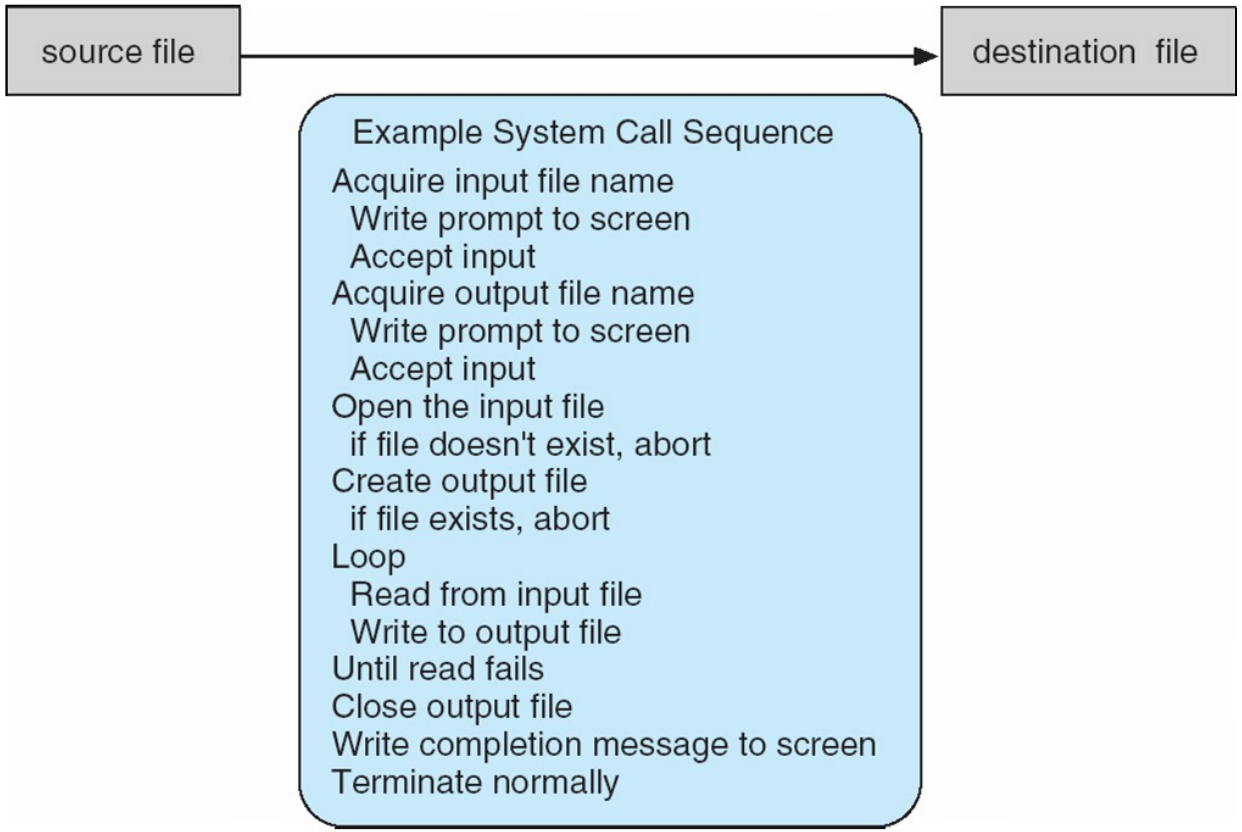
System Calls

- 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 Program 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)
- Why use APIs rather than system calls?

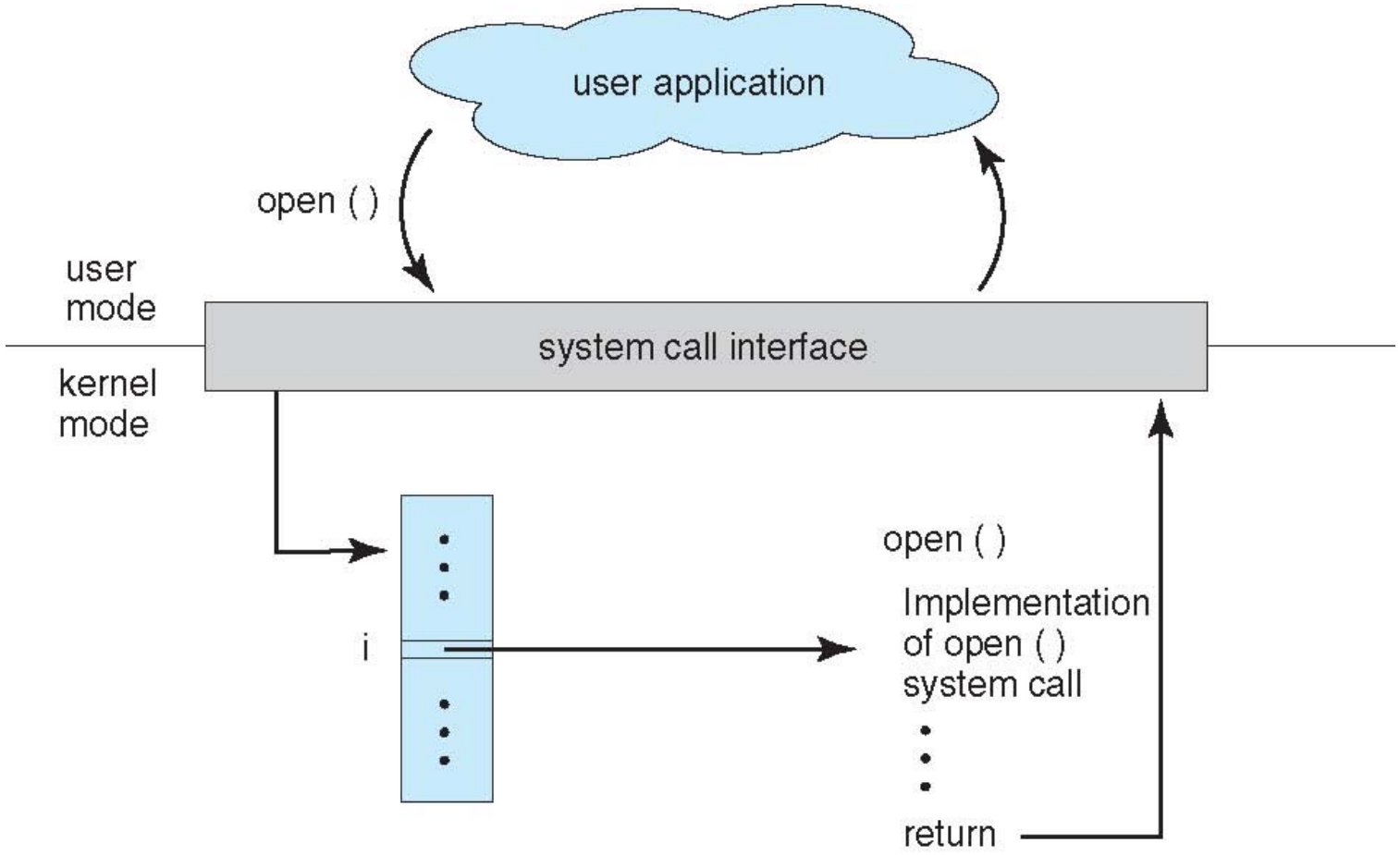
  (Note that the system-call names used throughout this text are generic)

Example of System Calls

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



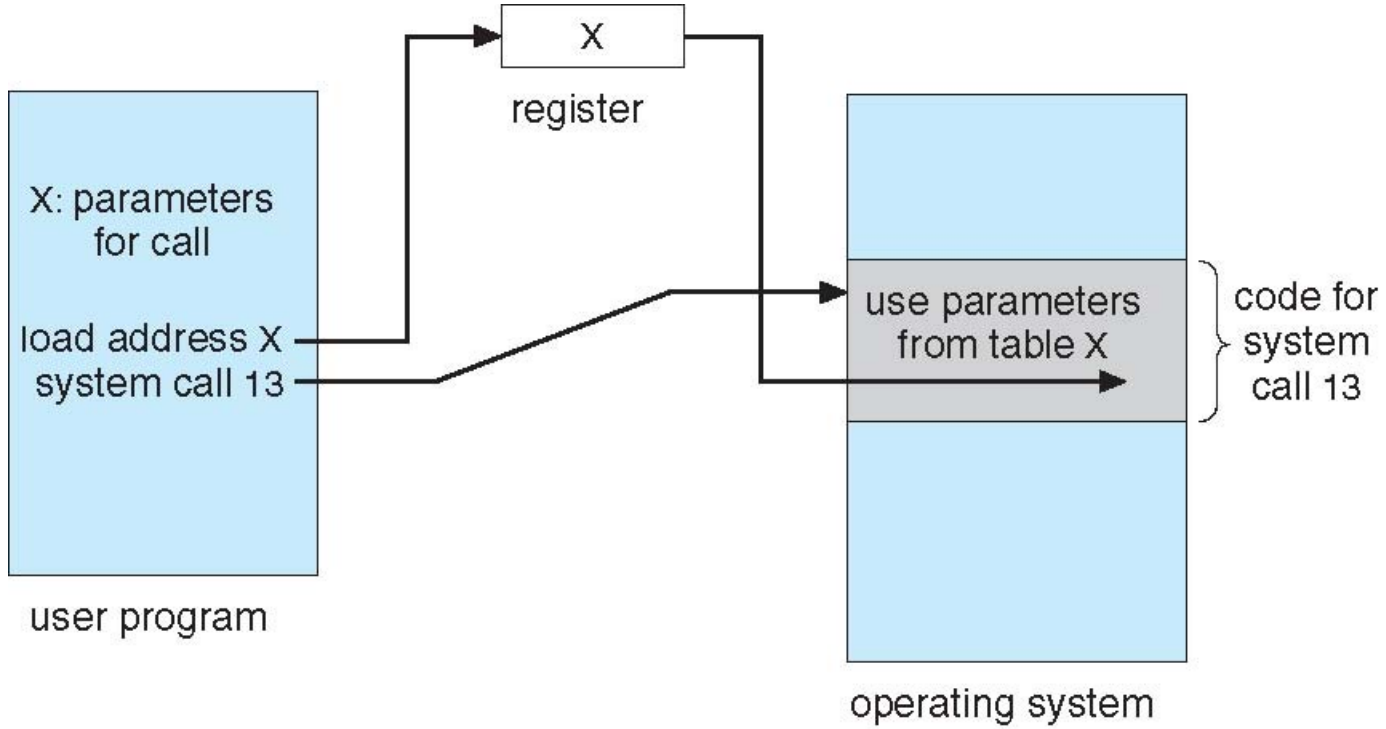
API – System Call – OS Relationship



System Call Parameter Passing

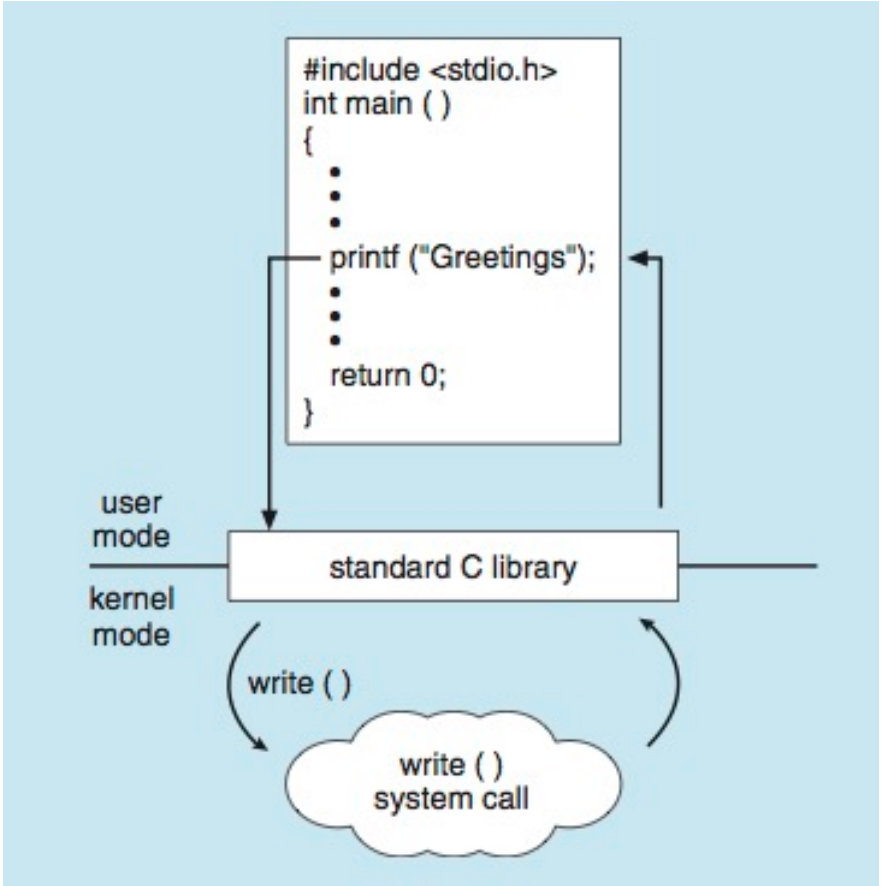
- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
    - In some cases, may be more parameters 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



Standard C Library Example

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



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 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

- Specifying and designing OS is highly creative task of **software engineering**

Implementation

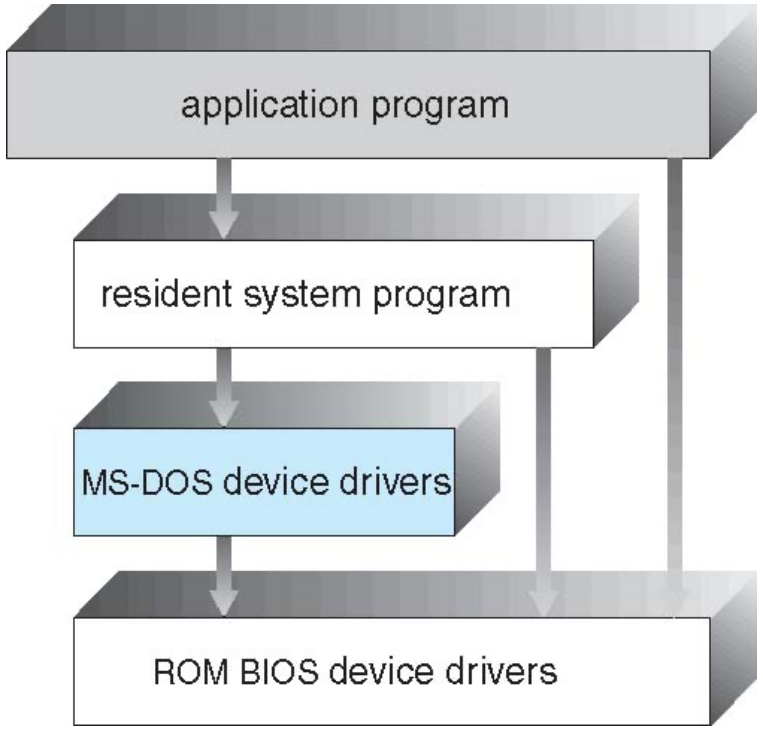
- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More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

Operating System Structure

- General-purpose OS is very large program
- Various ways to structure one as follows

Simple Structure

- I.e. 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

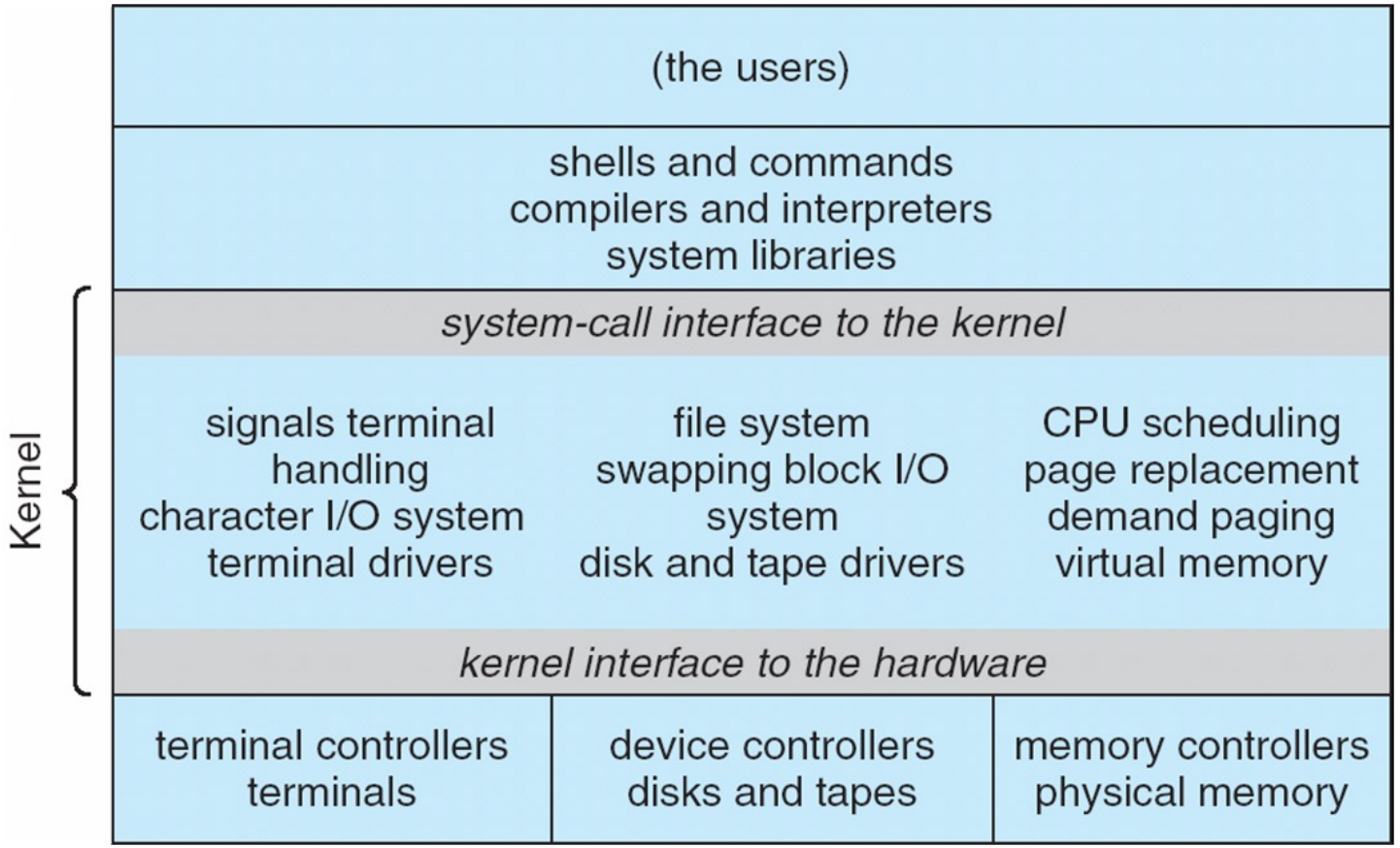


UNIX

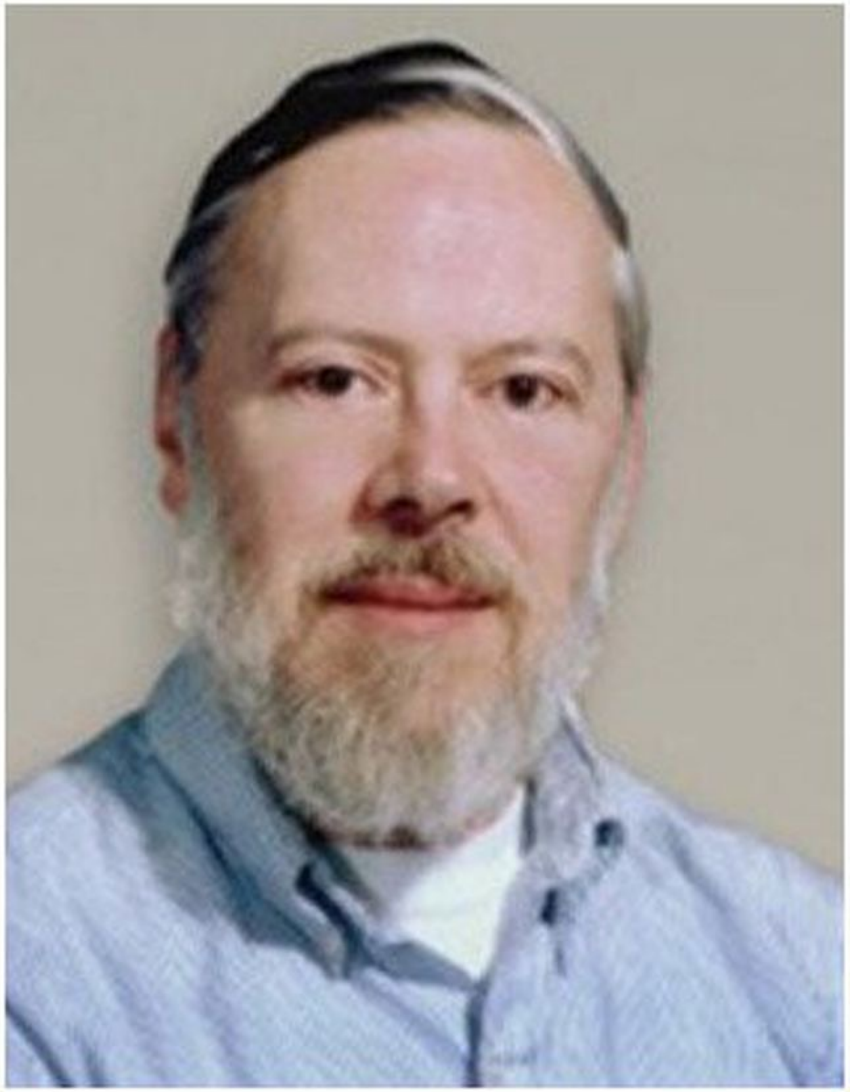
- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

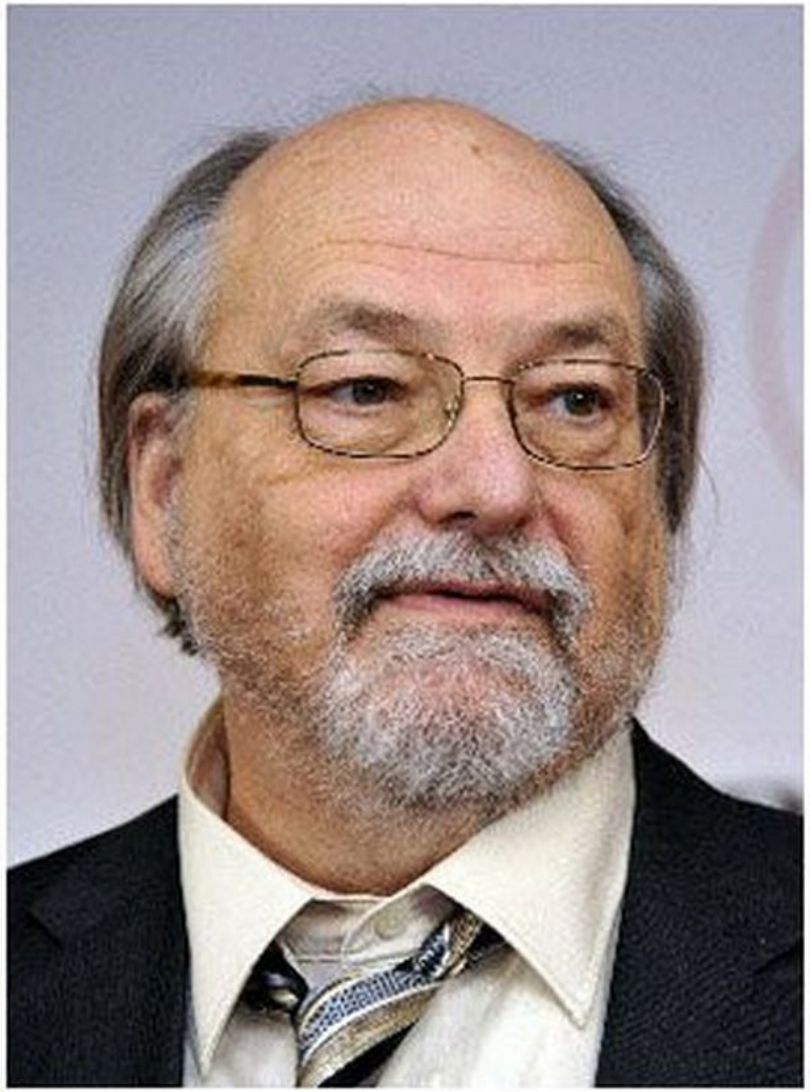
Traditional UNIX System Structure

Beyond simple but not fully layered



ผู้สร้าง UNIX : Dennis Ritche and Ken Thompson





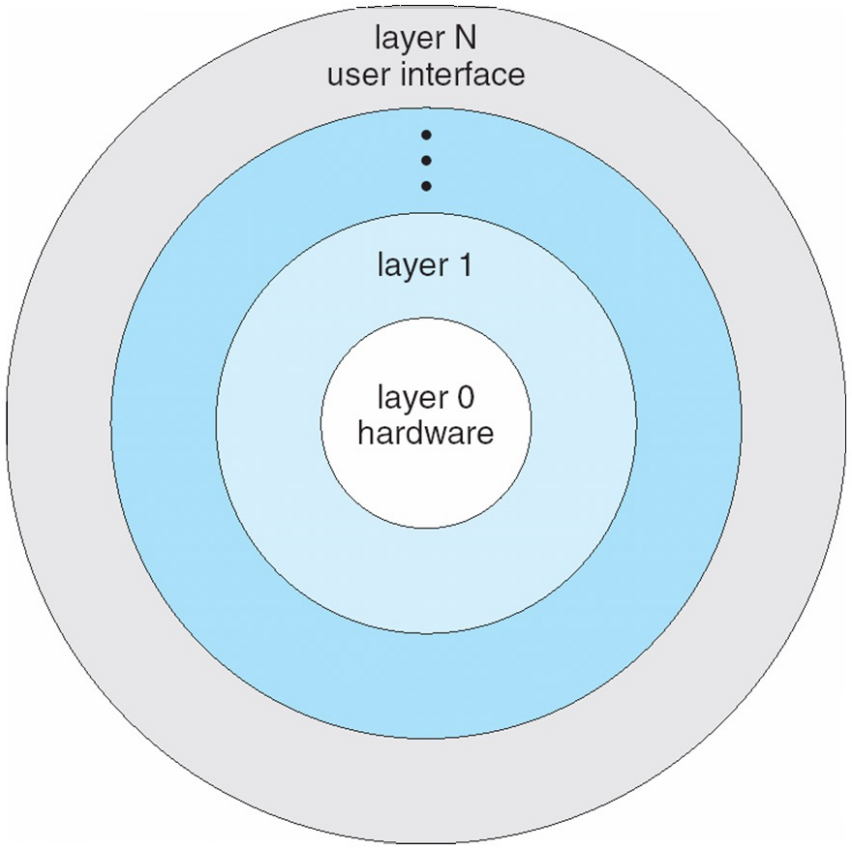
Bill Joy ผู้บุกเบิก UNIX BSD และอีกมาก


Sun
HP
IBM
VMWare
Apple
Oracle
OS X
Windows
Linux
Xen
Red Hat
Fedora
CentOS
Debian
Ubuntu
Mint
SUSE
Mageia
Arch Linux
Slackware
Mandriva
Gentoo
FreeBSD
OpenBSD
NetBSD
DragonFly BSD
Darwin

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.

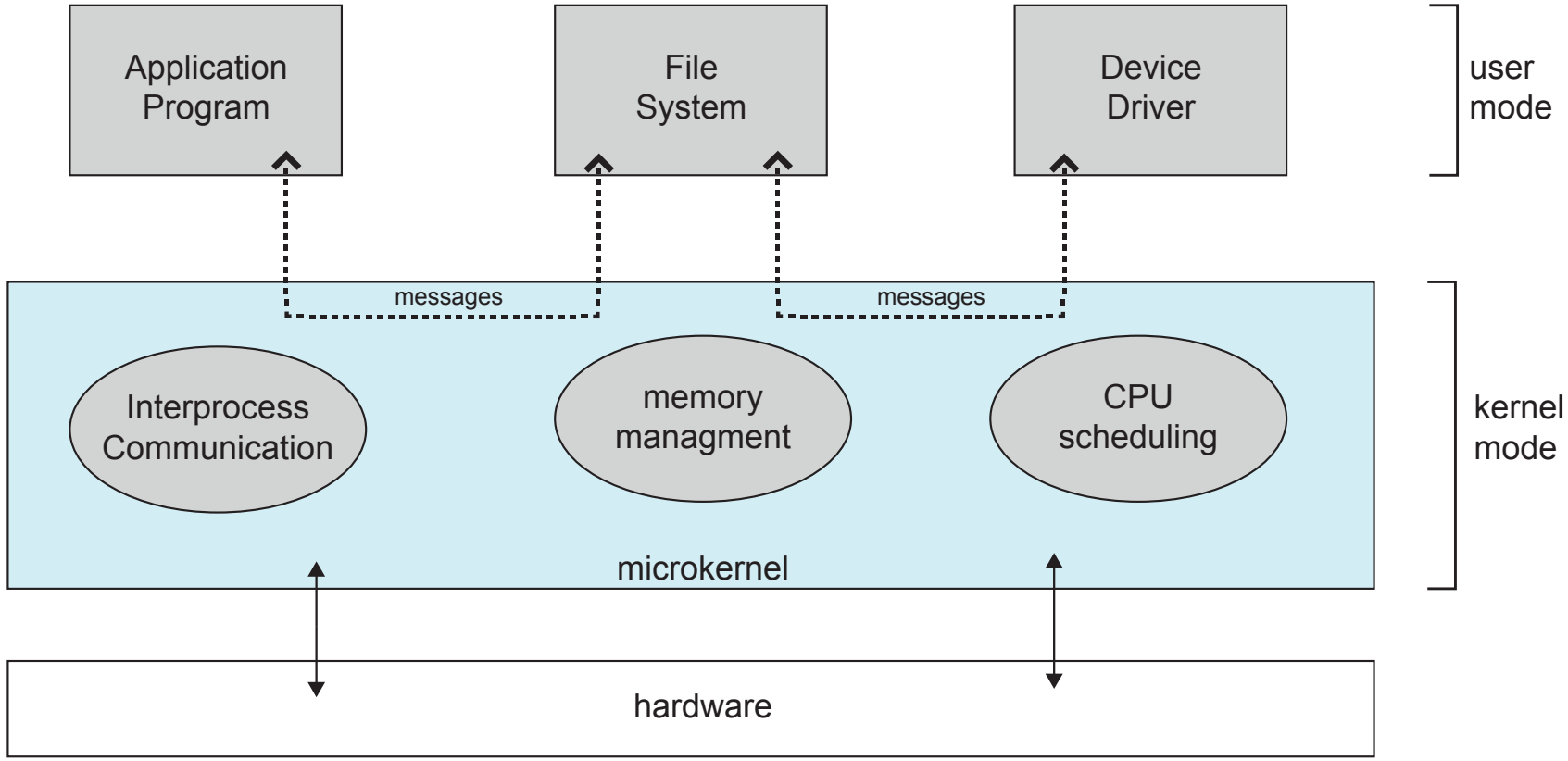
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers



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**
- Benefits:
  - 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
- Detriments:
  - Performance overhead of user space to kernel space communication

Microkernel System Structure



MINIX

Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate

Contribute

Help
Learn to edit
Community portal
Recent changes
Upload file

MINIX

From Wikipedia, the free encyclopedia

Minix (from "mini-Unix") is a POSIX-compliant (since version 2.0),[4][5] Unix-like operating system based on a microkernel architecture.

Early versions of MINIX were created by Andrew S. Tanenbaum for educational purposes. Starting with MINIX 3, the primary aim of development shifted from education to the creation of a highly reliable and self-healing microkernel OS. MINIX is now developed as open-source software.

MINIX was first released in 1987, with its complete source code made available to universities for study in courses and research. It has been free and open-source software since it was re-licensed under the BSD license in April 2000.[6]

Relationship with Linux [edit]

Early influence [edit]

Linus Torvalds used and appreciated Minix,[19] but his design deviated from the Minix architecture in significant ways, most notably by employing a monolithic kernel instead of a microkernel. This was disapproved of by Tanenbaum in the Tanenbaum–Torvalds debate. Tanenbaum explained again his rationale for using a microkernel in May 2006.[20]

Early Linux kernel development was done on a Minix host system, which led to Linux inheriting various features from Minix, such as the Minix file system.

Tanenbaum – Torvalds Debate

Tanenbaum–Torvalds debate

From Wikipedia, the free encyclopedia

This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed.
Find sources: "Tanenbaum–Torvalds debate" – news · newspapers · books · scholar · JSTOR (July 2013) (Learn how and when to remove this template message)

The **Tanenbaum–Torvalds debate** was a debate between Andrew S. Tanenbaum and Linus Torvalds, regarding the Linux kernel and kernel architecture in general. Tanenbaum began the debate in 1992 on the Usenet discussion group `comp.os.minix`, arguing that microkernels are superior to monolithic kernels and therefore Linux was, even in 1992, obsolete.[1] Linux kernel developers Peter MacDonald, David S. Miller and Theodore Ts'o also joined the debate.

The debate has sometimes been considered a flame war.[2]


Andrew S. Tanenbaum (called `ast`, in `comp.os.minix`)


Linus Torvalds

The debate [edit]

While the debate initially started out as relatively moderate, with both parties involved making only banal statements about kernel design, it grew progressively more detailed and sophisticated with every round of posts. Besides just kernel design, the debate branched into several other topics, such as which microprocessor architecture would win out over others in the future. Besides Tanenbaum and Torvalds, several other people joined the debate, including Peter MacDonald, an early Linux kernel developer and creator of one of the first distributions, Softlanding Linux System; David S. Miller, one of the core developers of the Linux kernel; and Theodore Ts'o, the first North American Linux kernel developer.[*citation needed*]

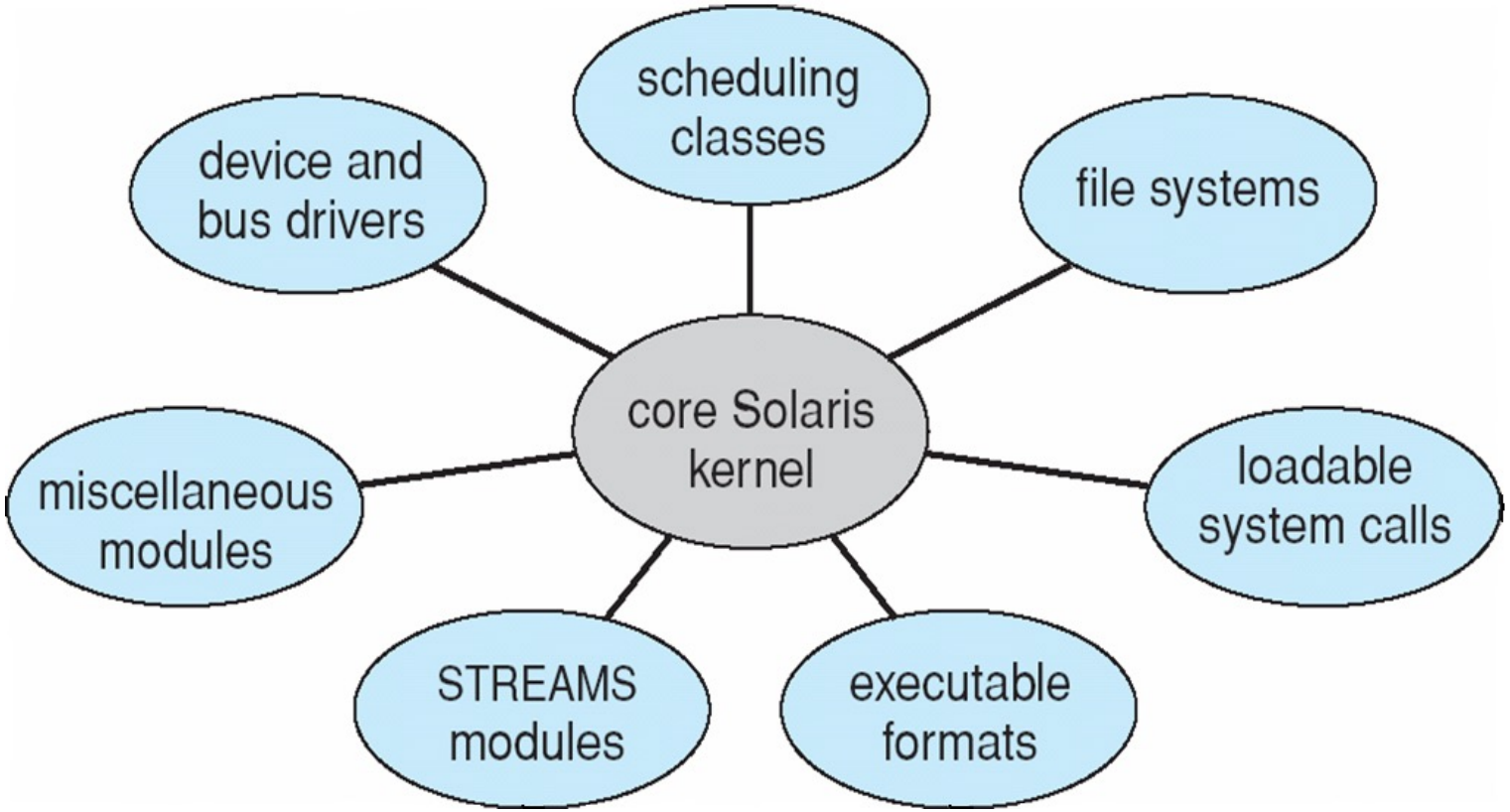
The debate opened on January 29, 1992, when Tanenbaum first posted his criticism on the Linux kernel to `comp.os.minix`, noting how the monolithic design was detrimental to its abilities, in a post titled "LINUX is obsolete".[1] While he initially did not go into great technical detail to explain why he felt that the microkernel design was better, he did suggest that it was mostly related to portability, arguing that the Linux kernel was too closely tied to the x86 line of processors to be of any use in the future, as this architecture would be superseded by then. To put things into perspective, he mentioned how writing a monolithic kernel in 1991 is "a giant step back into the 1970s".

Since the criticism was posted in a public newsgroup, Torvalds was able to respond to it directly. He did so a day later, arguing that MINIX has inherent design flaws (naming the lack of multithreading as a specific example), while acknowledging that he finds the microkernel kernel design to be superior "from a theoretical and aesthetical" point of view.[3] He also claimed that since he was developing the Linux kernel in his spare time and giving it away for free (Tanenbaum's MINIX was not free at that time), Tanenbaum should not object to his efforts. Furthermore, he mentioned how he developed Linux specifically for the Intel 80386 because it was partly intended as a learning exercise for Torvalds himself; while he conceded that this made the kernel itself less portable than MINIX, he asserted that this was an acceptable design principle, as it made the application programming interface simpler and more portable. For this reason, he stated, "linux is more portable than minix."

Modules

- Most 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



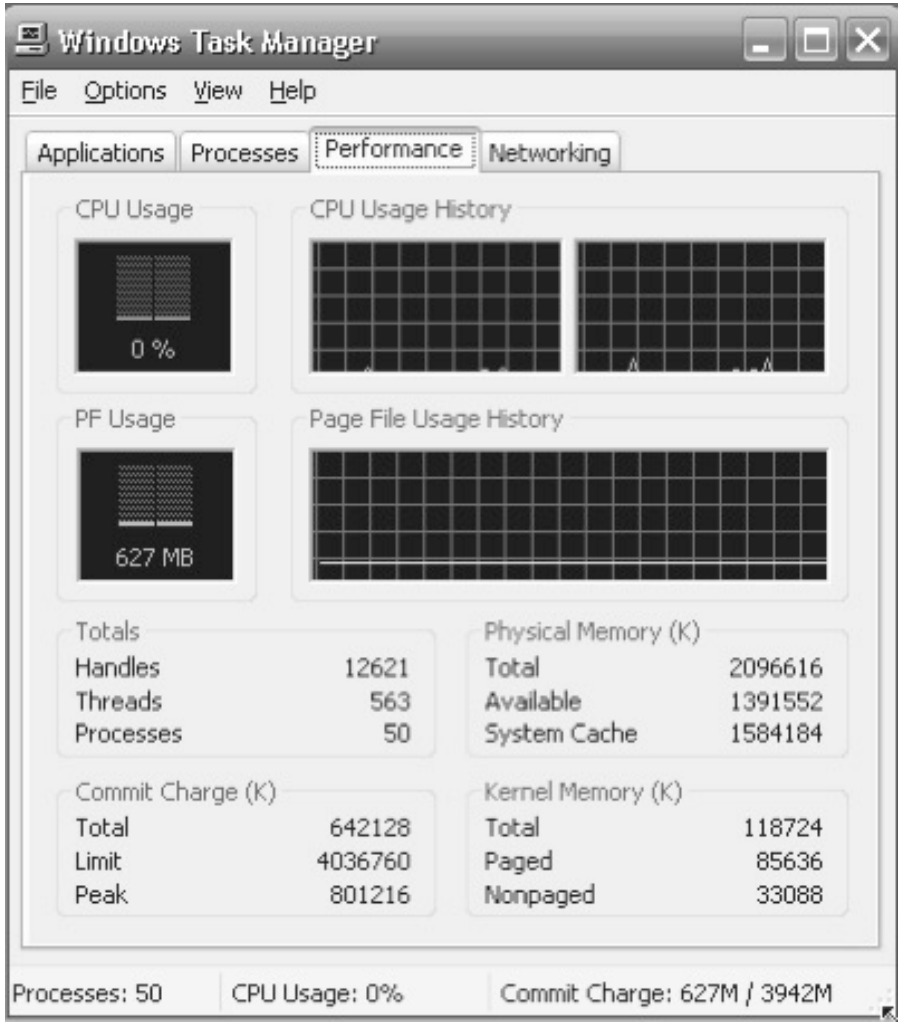
Operating-System Debugging

- **Debugging** is finding and fixing errors, or **bugs**
- OSes generate **log files** containing error information
- Failure of an application can generate **core dump** file capturing memory of the process
- Operating system failure can generate **crash dump** file containing kernel memory
- Beyond crashes, performance tuning can optimize system performance
  - Sometimes using ***trace listings*** of activities, recorded for analysis
  - **Profiling** is periodic sampling of instruction pointer to look for statistical trends

Kernighan's Law: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

Performance Tuning

- Improve performance by removing bottlenecks
- OS must provide means of computing and displaying measures of system behavior
- For example, “top” program or Windows Task Manager



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

- **SYSGEN** program obtains information concerning the specific configuration of the hardware system
  - Used to build system-specific compiled kernel or system-tuned
  - Can general more efficient code than one general kernel

System Boot

- When power initialized on system, execution starts at a fixed memory location
  - Firmware ROM used to hold initial boot code
- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – **bootstrap loader**, stored in **ROM** or **EEPROM** locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where **boot block** at fixed location loaded by ROM code, which loads bootstrap loader from disk
- Common bootstrap loader, **GRUB**, allows selection of kernel from multiple disks, versions, kernel options
- Kernel loads and system is then **running**