How do Users and Processes interact with the Operating System?

- **Users** interact indirectly through a collection of system programs that make up the operating system interface. The interface could be:
  - A GUI, with icons and windows, *etc.*
  - A command-line interface for running processes and scripts, browsing files in directories, *etc.*
  - Or, back in the olden days, a non-interactive batch system that takes a collection of jobs, which it proceeds to churn through (*e.g.* payroll calculations, market predictions, *etc.*)
- **Processes** interact by making *system calls* into the operating system proper (*i.e.* the *kernel*).
  - Though we will see that, for stability, such calls are not direct calls to kernel functions.

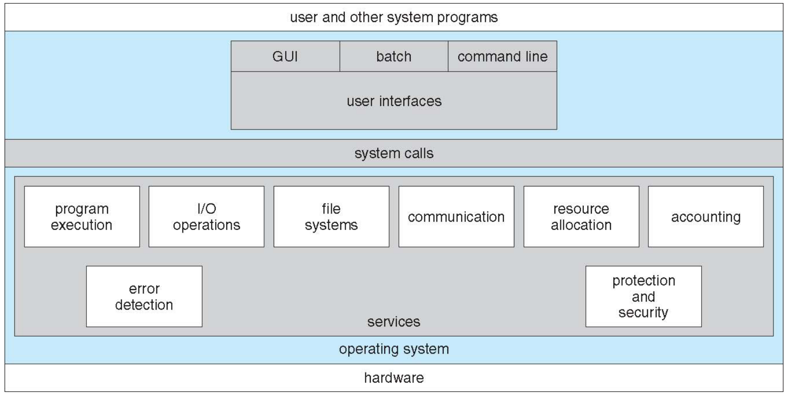
Services for Processes

- Typically, operating systems will offer the following services to processes:
  - **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**: Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
  - **Interprocess Communication (IPC)**: Allowing processes to share data through message passing or shared memory

Services for the OS Itself

- Typically, operating systems will offer the following internal services:
  - **Error handling**: what if our process attempts a divide by zero or tries to access a protected region of memory, or if a device fails?
  - **Resource allocation**: Processes may compete for resources such as the CPU, memory, and I/O devices.
  - **Accounting**: *e.g.* how much disk space is this or that user using? how much network bandwidth are we using?
  - **Protection and Security**: The owners of information stored in a multi-user or networked computer system may want to control use of that information, and concurrent processes should not interfere with each other

OS Structure with 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

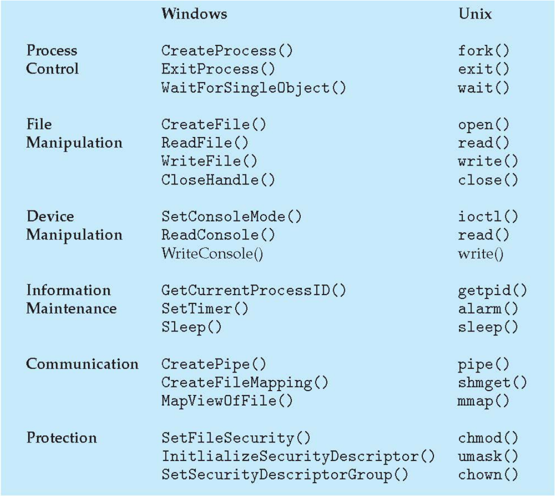
operating system

hardware

System Calls

- Programming interface to the services provided by the OS (*e.g.* open file, read file, *etc.*)
- 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.
- Three most common APIs are Win32 API for Windows, POSIX API for UNIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
- So why use APIs in user processes rather than system calls directly?
  - Since system calls result in execution of privileged kernel code, and since it would be crazy to let the user process switch the CPU to privileged mode, we must make use of the low-level hardware trap instruction, which is cumbersome for user-land programmers.
  - The user process runs the trap instruction, which will switch CPU to privileged mode and jump to a kernel pre-defined address of a generic system call function, hence the transition is controlled by the kernel.
  - Also, APIs can allow for backward compatibility if system calls change with the release of a OSS

System calls provided by Windows and Linux



	Windows	Unix

Process Control	CreateProcess()
	ExitProcess()
	WaitForSingleObject()
		fork()
		exit()
		wait()

File Manipulation	CreateFile()
	ReadFile()
	WriteFile()
	CloseHandle()
		open()
		read()
		write()
		close()

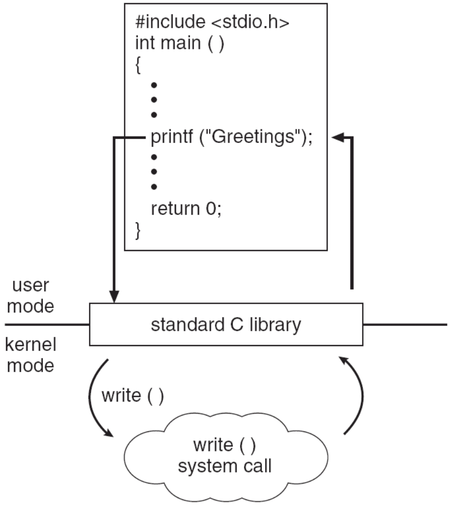
Device Manipulation	SetConsoleMode()
	ReadConsole()
	WriteConsole()
		ioctl()
		read()
		write()

Information Maintenance	GetCurrentProcessID()
	SetTimer()
	Sleep()
		getpid()
		alarm()
		sleep()

Communication	CreatePipe()
	CreateFileMapping()
	MapViewOfFile()
		pipe()
		shmget()
		mmap()

Protection	SetFileSecurity()
	InitlializeSecurityDescriptor()
	SetSecurityDescriptorGroup()
		chmod()
		umask()
		chown()

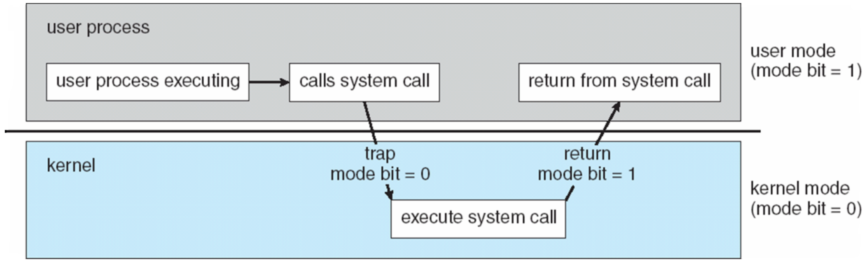
An example of a System Call



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

Trapping to the Kernel

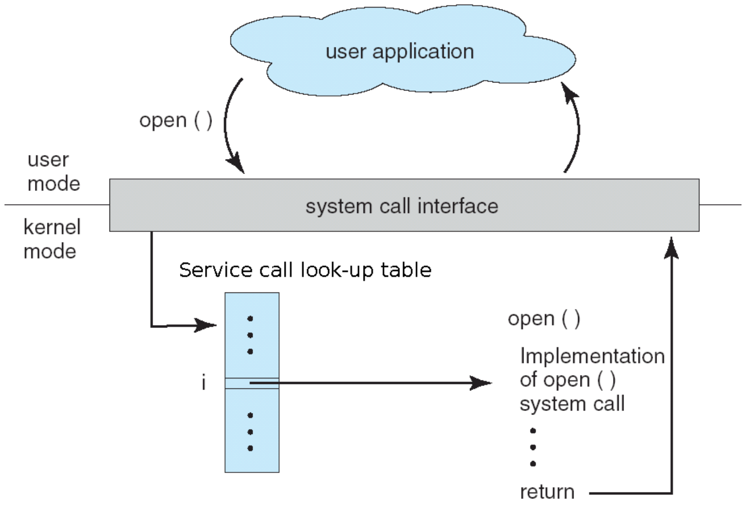
- The user process calls the system call wrapper function from the standard C library
- The wrapper function issues a low-level *trap* instruction (in assembly) to switch from user mode to kernel mode



Trapping to the Kernel

- To get around the problem that no call can directly be made from user space to a specific function in kernel space:
  - Before issuing the trap instruction, an index is stored in a well known location (*e.g.* CPU register, the stack, *etc.*).
  - Then, once switched into kernel space, the index is used to look up the desired kernel service function, which is then called.
- Some function calls may take arguments, which may be passed as pointers to structures via registers.

Trapping to the Kernel



OS Design

- An OS is possibly the most complex system that a computer will run, and it not yet clear (nor may it ever be) how to design an operating system to best meet the many and varied requirements placed on it.
- The internal structure of OSes can vary widely
- We can 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, easy to extend, and efficient
- OS architectures have evolved over the years, generally trying to better balance efficiency and stability

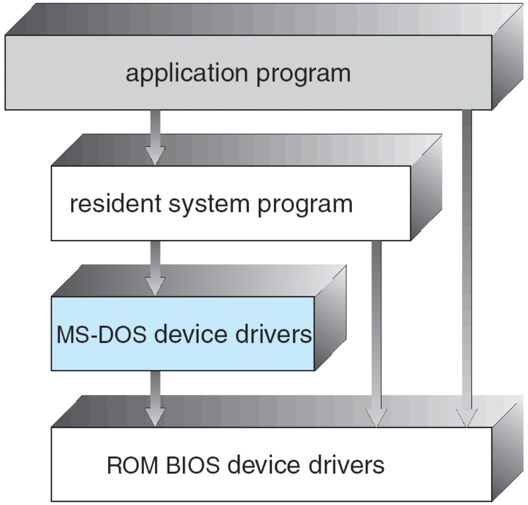
Separation of Policies and Mechanisms

- **Policy**: What will be done?
- **Mechanism**: How to do it?
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later
- An architecture that supports extendible file systems is a good example
  - Rather than hard code a particular file system into the kernel code, create an abstract file-system interface with sufficient flexibility that it can accommodate many file system implementations, eg: NTFS, EXT, FAT.

MS-DOS

- 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 - the highest layer is allowed to call the lowest layer.
  - So, it was easy for a user process (accidentally or on purpose) to de-stabilise the whole system, which is often what happened, even up until MS-DOS based Windows ME.

MS-DOS



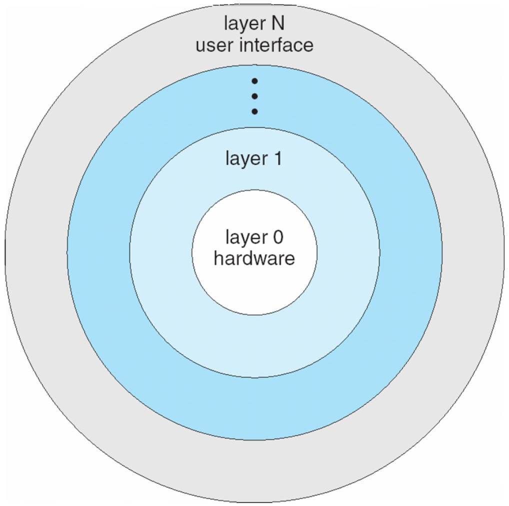
Strict Layered

- 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
  - Importantly for stability, modern CPUs offer protected mode, which means transition between layers is controlled by hardware
  - Attempts of software to run instructions or access memory regions that are higher privilege will result in the CPU raising a hardware exception (*e.g.* divide by zero, attempt to access hardware directly, *etc.*)
- Lends itself to simpler construction, since layers have well-defined functionality and can be tested independently or altered with minimal impact on the rest of the OS (*e.g.* lowest level could be adapted to different CPU architectures with minimal impact on higher layers)

Strict Layered

- Consider a file system. The actual file-system can be implemented in a layer above a layer that reads and writes raw data to a particular disk device, such that the file system will work with any device implemented by the lower layer (*e.g.* USB storage device, floppy disk, hard disk, *etc.*).
- In practice, however, it can be difficult to decide how many layers to have and what to put in each layer to build a general purpose operating system.

Strict Layered

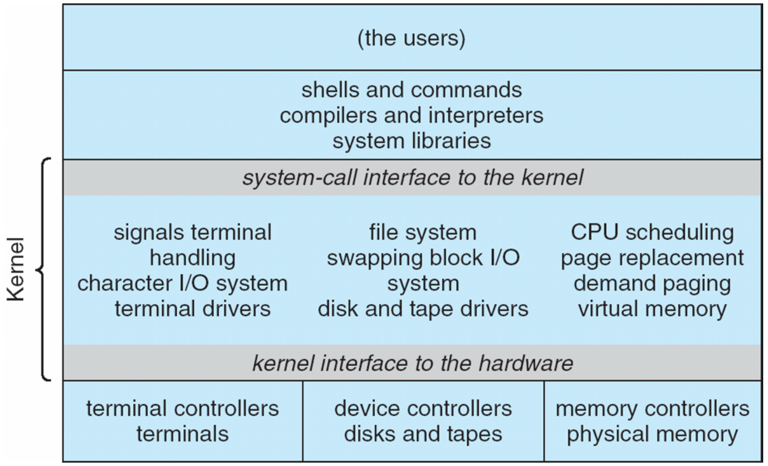


Traditional UNIX

UNIX - one big 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
- Limited to hardware support compiled into the kernel.

Traditional UNIX



(the users)

shells and commands
compilers and interpreters
system libraries

system-call interface to the kernel

Kernel

signals terminal handling
character I/O system
terminal drivers

file system
swapping block I/O system
disk and tape drivers

CPU scheduling
page replacement
demand paging
virtual memory

kernel interface to the hardware

terminal controllers
terminals

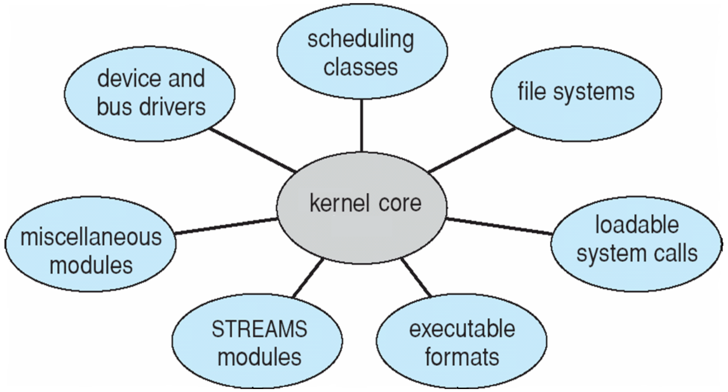
device controllers
disks and tapes

memory controllers
physical memory

Modular Kernel

- Most modern operating systems implement kernel modules
  - Uses object-oriented–like approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel, so you could download a new device driver for your OS and load it at run-time, or perhaps when a device is plugged in
- Overall, similar to layered architecture but with more flexibility, since all require drivers or kernel functionality need not be compiled into the kernel binary.
- Note that the separation of the modules is still only logical, since all kernel code (including dynamically loaded modules) runs in the same privileged address space (a design now referred to as monolithic), so I could write a module that wipes out the operating system no problem.
  - This leads to the benefits of micro-kernel architecture, which we will look at soon

Modular Kernel



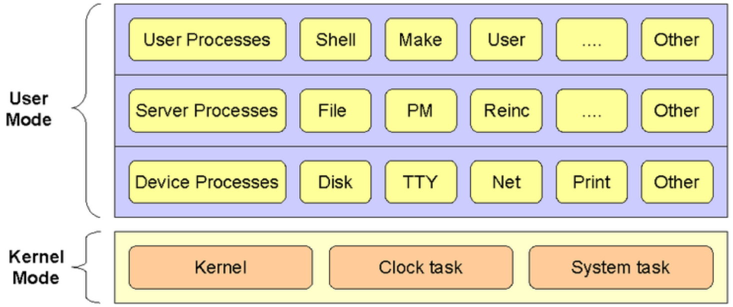
Microkernel

- Moves as much as possible from the kernel into less privileged “user” space (*e.g.* file system, device drivers, *etc.*)
- Communication takes place between user modules using message passing
  - The device driver for, say, a hard disk device can run all logic in user space (*e.g.* decided when to switch on and off the motor, queuing which sectors to read next, *etc.*)
  - But when it needs to talk directly to hardware using privileged I/O port instructions, it must pass a message requesting such to the kernel.

Microkernel

- Benefits:
  - Easier to develop microkernel extensions
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode) - if a device driver fails, it can be re-loaded
  - More secure, since kernel is less-complex and therefore less likely to have security holes.
  - The system can recover from a failed device driver, which would usually cause “a blue screen of death” in Windows or a “kernel panic” in linux.
- Drawbacks:
  - Performance overhead of user space to kernel space communication
- The Minix OS is an example of a microkernel architecture

Microkernel: MINIX



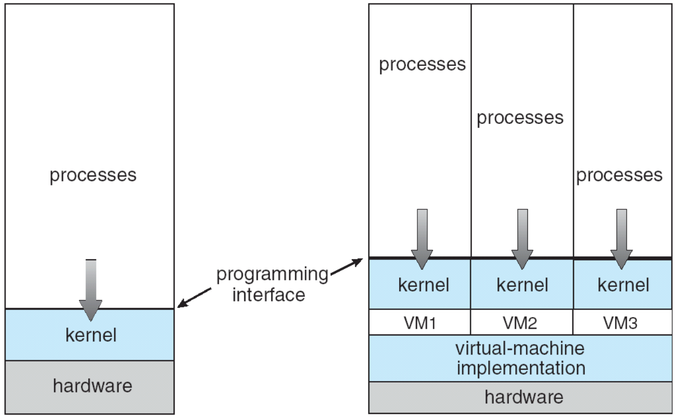
User Mode	User Processes	Shell	Make	User	....	Other
	Server Processes	File	PM	Reinc	....	Other
	Device Processes	Disk	TTY	Net	Print	Other
Kernel Mode	Kernel	Clock task	System task			

The MINIX 3 Microkernel Architecture

Virtual Machines

- A virtual machine takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware
- A virtual machine provides an interface identical to the underlying bare hardware
- The operating system host creates the illusion that a process has its own processor and (virtual memory)
- Each guest is provided with a (virtual) copy of underlying computer, so it is possible to install, say, Windows XP as a guest operating system on Linux.

Virtual Machines



Non-virtualised System

Virtualised System

Virtual Machines: History and Benefits

- First appeared commercially in IBM mainframes in 1972
- Fundamentally, multiple execution environments (different operating systems) can share the same hardware
- Protected from one another, so no interference
  - Some sharing of files can be permitted, controlled
  - Communicate with one another other and with other physical systems via networking
- Useful for development, testing, especially OS development, where it is trivial to revert an accidentally destroyed OS back to a previous stable snapshot.

Virtual Machines: History and Benefits

- Consolidation of many low-resource use systems onto fewer busier systems
- “Open Virtual Machine Format”, standard format of virtual machines, allows a VM to run within many different virtual machine (host) platforms.
- Not to be confused with emulation, where guest instructions are run within a process that pretends to be the CPU (*e.g.* Bochs and QEMU). In virtualisation, the goal is to run guest instructions directly on the host CPU, meaning that the guest OS must run on the CPU architecture of the host.

Para-virtualisation

- Presents guest with system similar but not identical to hardware (*e.g.* Xen Hypervisor)
- Guest OS must be modified to run on paravirtualized 'hardware'
    - For example, the kernel is recompiled with all code that uses privileged instructions replaced by hooks into the virtualisation layer
    - After an OS has been successfully modified, para-virtualisation is very efficient, and is often used for providing low-cost rented Internet servers (*e.g.* www.slicehost.com)

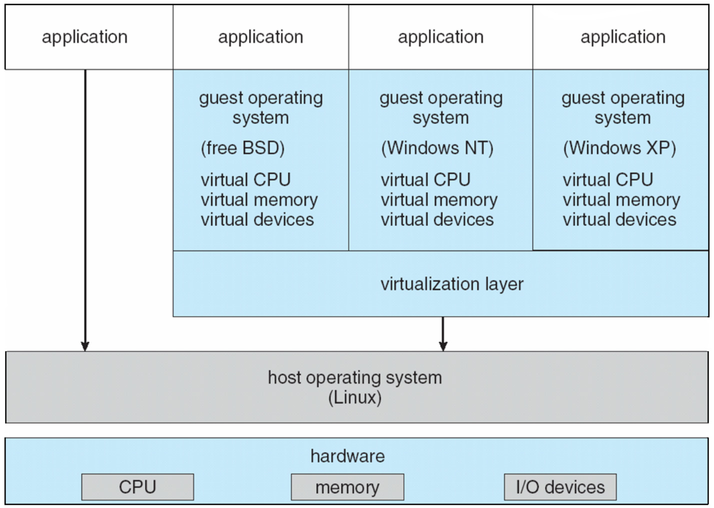
VMWare Architecture

- VMWare implements full virtualisation, such that guest operating systems do not require modification to run upon the virtualised machine.
- The virtual machine and guest operating system run as a user-mode process on the host operating system

VMWare Architecture

- As such, the virtual machine must get around some tricky problems to convince the guest operating system that it is running in privileged CPU mode when in fact it is not.
    - Consider a scenario where a process of the guest operating system raises a divide-by-zero error.
    - Without special intervention, this would cause the host operating system immediately to halt the virtual machine process rather than the just offending process of the guest OS.
    - So VMWare must look out for troublesome instructions and replace them at run-time with alternatives that achieve the same effect within user space, albeit with less efficiency
    - But since usually these instructions occur only occasionally, many instructions of the guest operating system can run unmodified on the host CPU.

VMWare Architecture


application
application
application
application
guest operating system
(free BSD)
virtual CPU
virtual memory
virtual devices
guest operating system
(Windows NT)
virtual CPU
virtual memory
virtual devices
guest operating system
(Windows XP)
virtual CPU
virtual memory
virtual devices
virtualization layer
host operating system
(Linux)
hardware
CPU
memory
I/O devices