Operating Systems

Lecture Course in Autumn Term 2015
University of Birmingham

Erik Tews, David Oswald
based on a course by Eike Ritter

September 27, 2016

Course Details

- Lecture notes and resources:
- use canvas for discussions
- Recommended Course Books
  - OS Concepts (8th Edition) Silberchatz *et al.*
  - Linux Kernel Development. Robert Love.
  - Linux Programming Interface, Michael Kerrisk
  - The C Programming Language (2nd Edition) Kernighan and Ritchie

Course Details

- Assessment:
  - 80% exam, 20% coursework
  - Description of coursework and assessment criteria will be made available on canvas.
  - There will be 4 pieces of challenging coursework over the term that will put you through your paces, ultimately to give you a firm grasp on the subject.
  - Students on the extended module (check your registration status with the school office if you are unsure of this) will be given an additional piece of coursework.
  - Make use of the labs (two hours per week) to work on the coursework problems.

Prerequisites for this module

- Good C programming skills are essential prerequisite for this module
- Familiarity with pointers and explicit memory management expected
- See `http://www.cs.bham.ac.uk/~exr/teaching/lectures/opsys/13_14/exercises/ex1.pdf` for the level of C programming skills expected.
- Excercise 1 will be used to freshen up your C skills

What is an Operating System?

- A program that acts as an intermediary between a user of a computer and the computer hardware
- Operating system goals:
  - Execute user programs and make solving user problems easier
  - Make the computer system convenient to use
  - Use the computer hardware in an efficient manner

Two Popular Definitions of an OS

OS as a resource allocator:	Manages all resources and decides between conflicting requests for efficient and fair resource use (*e.g.* accessing disk or other devices)
OS as a control system:	Controls execution of programs to prevent errors and improper use of the computer (*e.g.* protects one user process from crashing another)

Computer System Structure

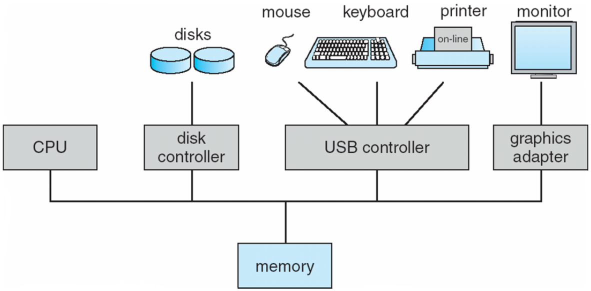
Computer system can be divided into four components:

- Hardware: provides basic computing resources CPU, memory, I/O devices
- Operating system: Controls and coordinates use of hardware among various applications and users
- Application programs: define the ways in which the system resources are used to solve the computing problems of the users Word processors, compilers, web browsers, database systems, video games
- Users: People, machines, other computers

Bootstrapping of the OS

- Small bootstrap program is loaded at power-up or reboot
    - Typically stored in ROM or EPROM, generally known as firmware (*e.g.* BIOS)
- Initializes all aspects of the system (*e.g.* detects connected devices, checks memory for errors, *etc.*)
- Loads operating system kernel and starts its execution

Computer System Organisation



- One or more CPUs, device controllers connect through common bus providing access to shared memory
- CPU(s) and devices compete for memory cycles (*i.e.* to read and write memory addresses)

Computer System Operation

- I/O devices and the CPU can execute concurrently
- Each device controller (*e.g.* controller chip) is in charge of a particular device type
- Each device controller has a local buffer (*i.e.* memory store for general data and/or control registers)
- CPU moves data from/to main memory to/from controller buffers (*e.g.* write this data to the screen, read coordinates from the mouse, *etc.*)
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an *interrupt*

Interrupts

- Interrupt transfers control to the interrupt service routine generally, through the interrupt vector, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction so original processing may be resumed
- Incoming interrupts are disabled while another interrupt is being processed to prevent a lost interrupt
- A trap is a software-generated interrupt caused either by an error or a user request

Storage Structure

- Main memory - only large storage media that the CPU can access directly
- Secondary storage - extension of main memory that provides large non-volatile storage capacity
- Magnetic disks - rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into tracks, which are subdivided into sectors
  - The disk controller determines the logical interaction between the device and the computer
- Today also often flash memory

Caching

- Important optimisation principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache often smaller than storage being cached
- Cache management is an important design problem
  - Determining cache size and replacement policy

Multiprocessors

- Most systems use a single general-purpose processor
- Multiprocessor systems growing in use and importance
  - Also known as parallel systems, tightly-coupled systems
- Advantages include
  - Increased throughput
  - Economy of scale
  - Increased reliability - graceful degradation or fault tolerance
- Two architectures
  - Asymmetric Multiprocessing - CPUs have different roles, usually one is the master of the others
  - Symmetric Multiprocessing - CPUs have identical roles, sharing process queues to service *ready* processes.

Fault Handling and Protection

- Software error or request creates exception or trap, which are essentially handled as special interrupts
  - Division by zero, request for operating system service
- Other process problems include infinite loop, processes modifying each others' or the operating system's code
- Dual-mode operation allows OS to protect itself and other system components
  - User mode and kernel mode
  - Mode bit provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Some instructions designated as privileged, only executable in kernel mode
    - System call changes mode to kernel, return from call resets it to user
    - Otherwise, a user process could manipulate hardware directly, leading to chaos.

Process Management

- A process is a program in execution. It is a unit of work within the system. Program is a passive entity (*e.g.* the stored code), process is an active entity.
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files, initialization data
- Process termination requires reclamation of any reusable resources
- Single-threaded process has one program counter, specifying location of next instruction to execute
  - Process executes instructions sequentially, one at a time, until completion
- Multi-threaded process has one program counter per thread
- Concurrency of systems achieved by multiplexing the CPUs among the processes/threads

Process Management Activities

The operating system is responsible for the following activities in connection with process management:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

Memory Management

- All data must be in memory before and after processing
- All instructions must be in memory in order to be executed
- Memory management determines what is in memory and when
    - Optimizing CPU utilization and computer response to users (*e.g.* avoiding excessive swapping of data between disk and memory)
- Memory management activities
    - Keeping track of which parts of memory are currently being used and by whom
    - Deciding which processes (or parts thereof) and data to move into and out of memory (*e.g.* to and from the disk - virtual memory)
    - Allocating and deallocating memory space as needed

Storage Management

- OS provides uniform, logical view of information storage
    - Abstracts physical properties to logical storage units: file and directories
- Each medium is controlled by device (*e.g.* disk drive, tape drive)
    - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
- File-System management
    - Access control on most systems to determine who can access what
    - OS activities include
        - Creating and deleting files and directories
        - Primitives (*i.e.* standardised functions) to manipulate files and directories
        - Mapping files within memory onto secondary storage

I/O Subsystems

- One purpose of OS is to hide peculiarities of hardware devices from the user
- I/O subsystem responsible for:
  - Memory management of I/O, including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs)
  - Abstract device-driver interface, so driver developers know how to interface their code with the OS (*e.g.* network cards, storage devices for different hardware use common interfaces).
  - Drivers for specific hardware devices

Protection and Security

- **Protection** - any mechanism for controlling access of processes or users to resources defined by the OS
- **Security** - defense of the system against internal and external attacks
    - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first distinguish among users, to determine who can do what
    - User identities (user IDs, security IDs) include name and associated number, one per user
    - User ID then associated with all files, processes of that user to determine access control
    - Group identifier (group ID) allows set of users to be defined and controls managed, then also associated with each process, file
    - Privilege escalation allows user to change to effective ID with more rights

Structuring Projects

- It is infeasible to try to write any useful system as a single source file
    - Especially if we'd like to reuse parts of it in other projects.
    - Or if we'd like someone else to clearly understand its structure.
- So today we will look at how we can structure our source into a project, splitting it into multiple files and possibly libraries.

Object File Compilation and Linking

- So far, we have allowed the compiler to hide from us a crucial step in building software: the **linking step**.
- When we run `gcc hello_world.c -o hello_world`, the compiler first **compiles** all of the files (in this case `hello_world.c`) to *object* files (the actual machine code representing each C file)
- Then it automatically **links** them together into a single file (the final executable), such that references to functions and variables foreign to each object file are resolved as calls (*i.e.* jumps to addresses) into the other object files.
- Additional to our object files, compiled code from the standard libraries is also pulled into the final executable (*e.g.* the machine code of, say, `printf`).
  - In fact, for a small program, most of the code in the executable will be code from such libraries.

Object File Compilation and Linking

- Often we require explicit control over the build process, so we first build the object files for each C file, using the *compile-only* flag, `-c`:
  - `gcc -c hello_world.c -o hello_world.o`
- Then we link the object files into the final executable (in this case `my_app`):
  - `gcc -o my_app hello_world.o linked_list.o ...`
- For the interested, if you run `objdump -d <some_compiled_binary>` on some object file or executable you will see a list of functions defined within the file along with their disassembled code.

Today's Code

- Today we will look at the code of a simple application, `todo_list`, that allows the user to enter a list of items, such that we will get a better understanding of what is involved in putting together a larger project.
  - First we will see how we could implement it as multiple files, looking at how header files my be defined and used.
  - Then we will see how we could split off the linked list code into to a library which can be re-used in other applications.

Building a Project with Make

- We can use the tool `make` to help us to build large software projects.
- `make` allows us to automate the running of long commands, through the definition rules in a special file called `Makefile`.

Structure of simple Makefile :

- Starts with declarations (assignment of values to variables)
- Then have list of targets and commands which re-create the target
  Need to have TAB-character at beginning of line containing the command!
- Can call `make` in directory `<dir>` via
  `make -C <dir>`

- Such a list of targets consists of one line containing target and dependencies
- dependencies are files which need to be present and newer than the target
- commands will be executed if target needs to be re-generated variables may be used

Conventions:

- Have target `all` which makes everything in this directory normally first target
- Have also target `clean` which removes all targets and temporary files created

Client-server architectures

- Have *server* process, which waits for client requests and processes them once they arrive
- Multiple clients may connect and request service
- Standard paradigm for services on the internet
- Have in addition to IP-addresses *port numbers* assigned to particular services
- Examples:
- Port 80 for http
- Port 25 for sending mail (smtp)
- Port 143 for reading mail via imap
- Port assignments listed in `/etc/services`
- Client-Server architecture implemented via *sockets*

Sockets

How to setup a client/server connection via sockets:
Initialisation phase for the server:

- Server creates endpoint via `socket`-system call
- Server specifies port number and protocol in structure `sockaddr_in`
- Server assigns information in `sockaddr_in`-structure to socket via `bind`-system call

Now server waits for incoming connection via `accept`-system call
When connection received, server reads data via `read`-system call and writes data back via `write`-system call
When server is finished with current connection, server closes connection via `close`-system call

Concurrency

- Good handling of concurrency vital for implementing sockets
- Will use `pthread`-library for this
  Library implements kernel-level threads together with synchronisation mechanisms
- Key point: Program may create arbitrary number of threads, which share memory and may run concurrently
- Synchronisation achieved by mutual exclusion:
  A section of code satisfies mutual exclusion if it can executed by only one thread at a time, and in addition no switching between threads happens while this section of code is executed.
- Such a section of code is called critical section

Important operations on threads

- pthread_create creates new threads
- pthread_exit exits threads
- pthread_mutex_lock(mutex) starts a critical section involving those threads using the specified mutex
- pthread_mutex_unlock(mutex) ends a critical section involving those threads using the specified mutex
- pthread_attr_init sets up default attributes
- pthread_join wait for termination for another thread
- pthread_attr_setdetachstate cause thread to terminate immediately when pthread_exit is called
  $\Rightarrow$ cannot use pthread_join

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.

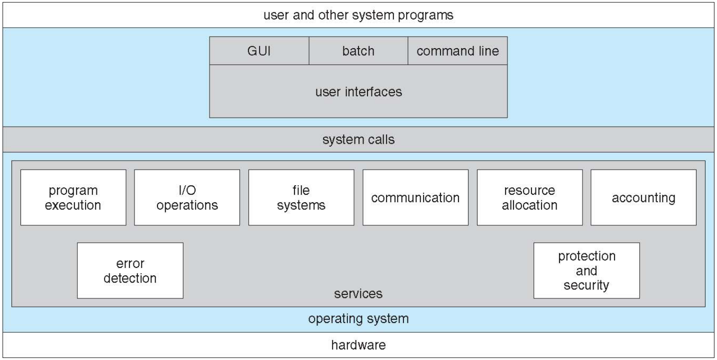
Services for Processes

- 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

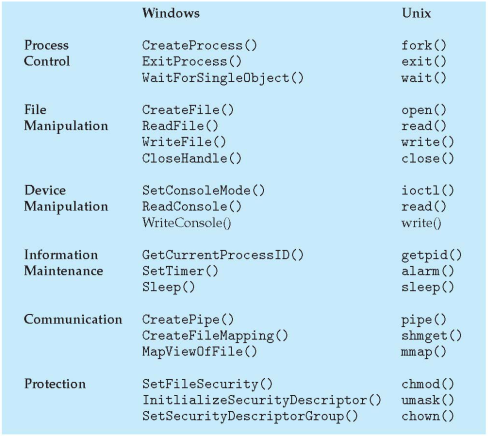
OS Structure with Services



System Calls

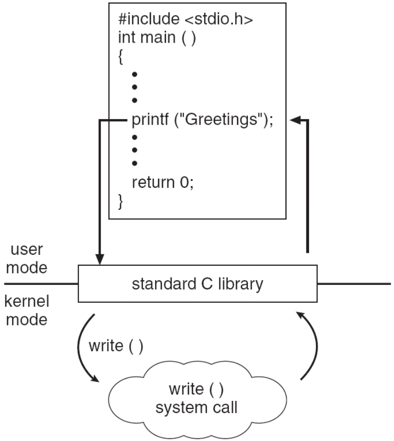
- Programming interface to the services provided by the OS (*e.g.* open file, read file, *etc.*)
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call.
- Three most common APIs are Win32 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



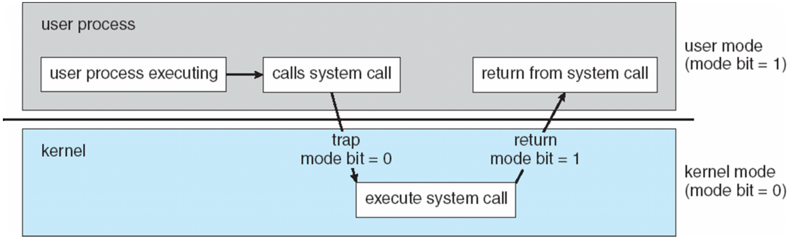
	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information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

An example of a System Call



Trapping to the Kernel

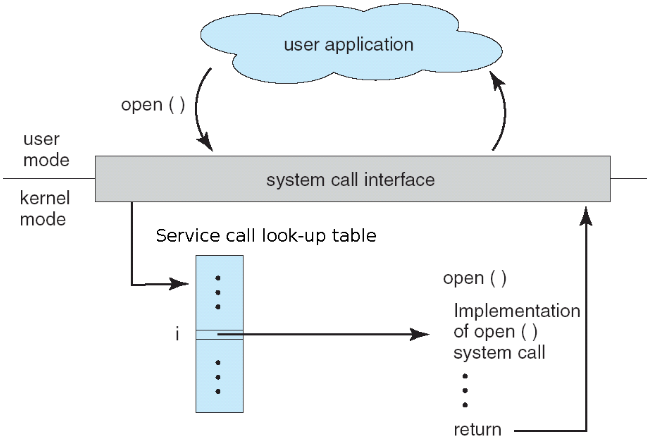
- The user process calls the system call wrapper function from the standard C library
- The wrapper function issues a low-level *trap* instruction (in assembly) to switch from user mode to kernel mode



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

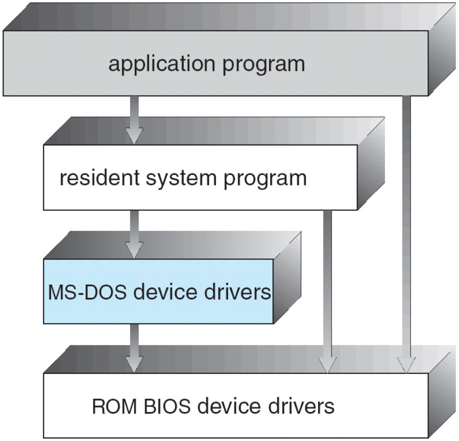
Separation of Policies and Mechanisms

- **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


application program
resident system program
MS-DOS device drivers
ROM BIOS device drivers

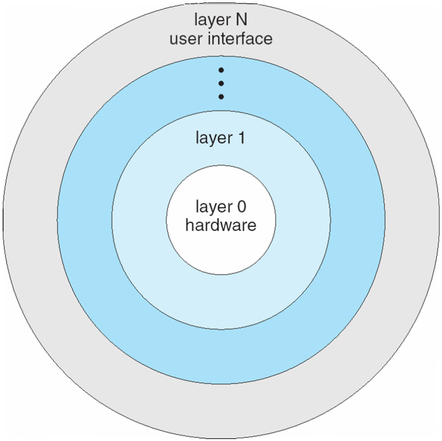
Strict Layered

- 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

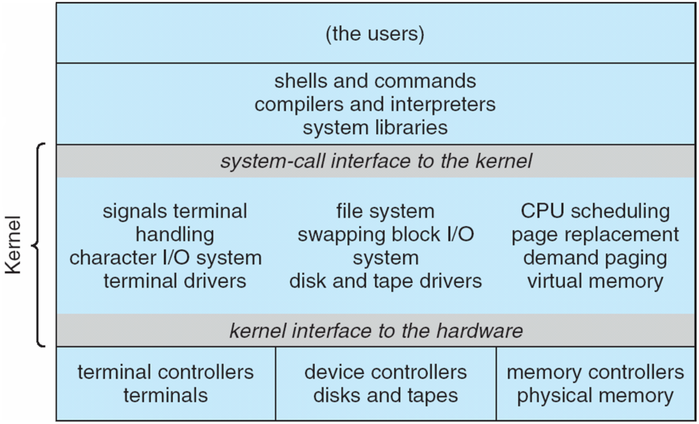


Traditional UNIX

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.

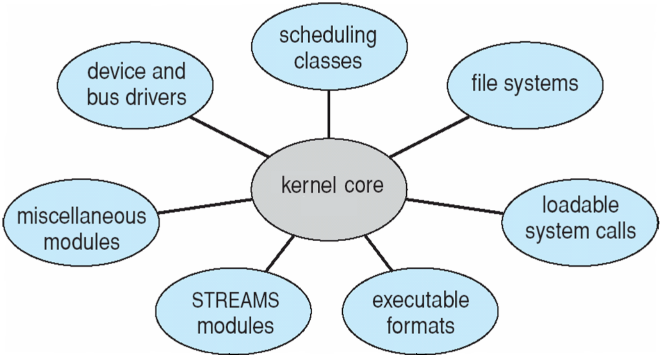
Traditional UNIX



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



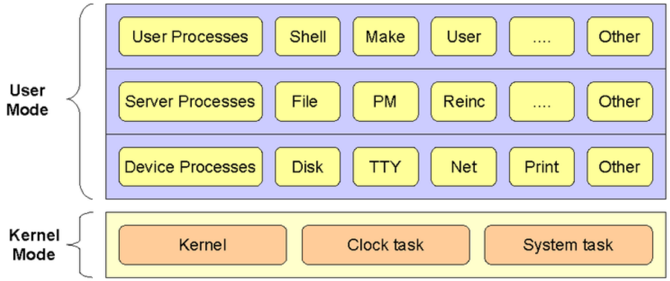
Microkernel

- 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

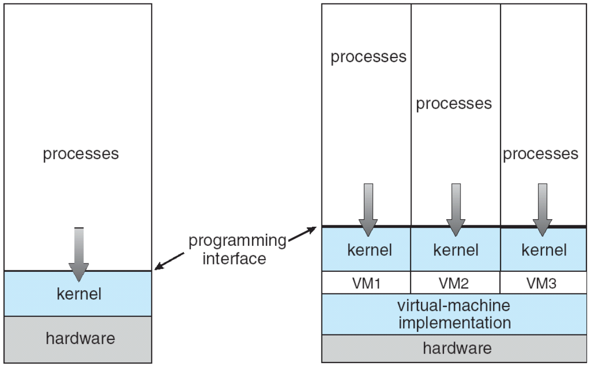


The MINIX 3 Microkernel Architecture

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)

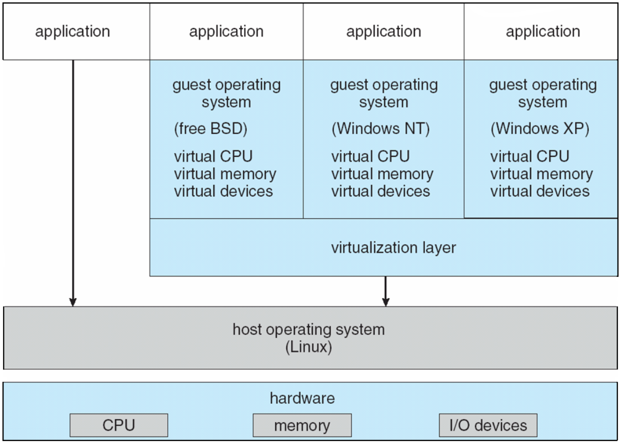
VMWare Architecture

- 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