Operating Systems

Second part of Operating Systems and Systems Programming Module

University of Birmingham

Eike Ritter (main author) Aad van Moorsel (edits)

Overview

Operating Systems

Second part of the course: Operating Systems
Outline of lecture:

- What does an operating system do?
- How to interact with the operating system
- Possible operating systems architectures
- Virtual machines
- Device drivers

Recommended books

Recommended books for this part of the course:

- Operating Systems Concepts Silberschatz *et al.*
- Linux Kernel Development. Robert Love.
- Linux Programming Interface, Michael Kerrisk
- The C Programming Language (2nd Edition) Kernighan and Ritchie

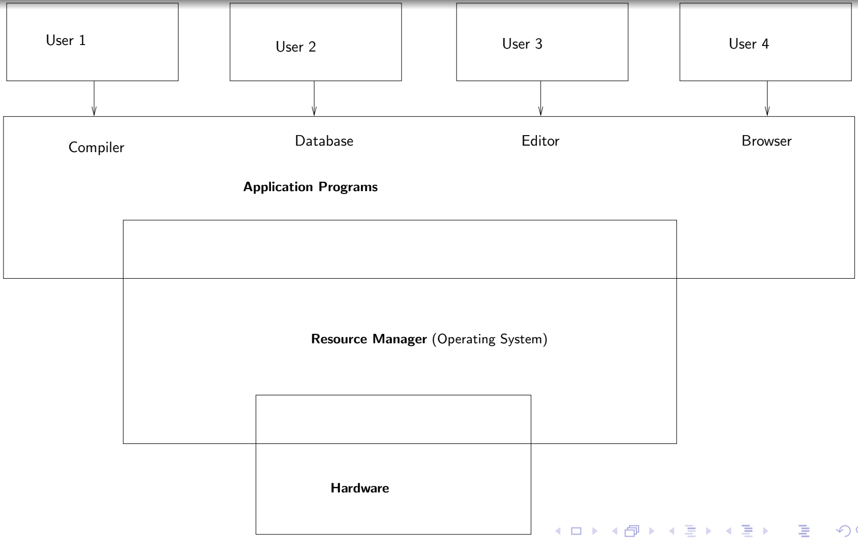
Note: most diagrams in these slides come from the book by Silberschatz, Galvin and Gagne

What is an Operating System?

Two valid descriptions:

- A program that acts as an intermediary between a ‘user’ of a computer and the computer hardware
- The one program that is *at all times* running on the computer, with all else being systems programs and application programs

OS Schematic



Main functions of an Operating System:

OS as a resource allocator:
: Manages all hardware resources and decides between conflicting requests for efficient and fair resource use (*e.g.* accessing CPU, 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)

Examples of Operating Systems

- Windows: current Windows OS based on Windows NT
- Mac OS: based on BSD Unix
- Linux: Based on Unix. Used also for Android.
- Newer operating systems for realtime applications and small resource usage (*e.g.* TinyOS for Internet of Things)

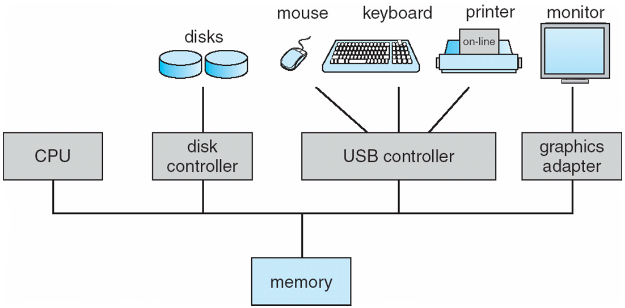
Operating System Basic Building Blocks

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

ROM = Read Only Memory
EPROM = Erasable Programmable ROM
BIOS = Basic Input/Output System

Sharing among Resources



- Device controllers are hardware within laptop/computer needed to connect to external resources
- Bus (the 'wire') connects CPUs, device controller and memory
- OS Kernel runs within the CPU and manages the devices through the device controllers

Sharing among Resources: Device Controllers

- 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*

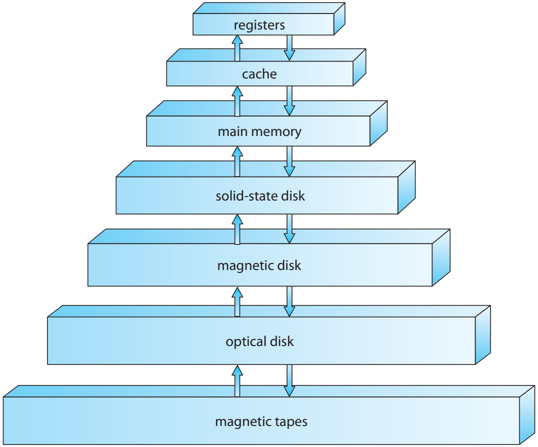
Interrupts

- Interrupts can be implemented in different way, but a common approach is through an interrupt vector
- Interrupt Vector is a reserved part of the memory, tracking which interrupts need to be handled
- For each interrupt, the OS executes the appropriate 'Interrupt Service Routine' to handle the interrupt
- The address of the interrupted instruction is saved so original processing can be resumed after completion of the interrupt
- Software programs can also generate interrupts, through **system calls**, for instance when an error occurs. A software-generated interrupt is called a **trap**.

Storage Structure: Memory vs. Secondary Storage

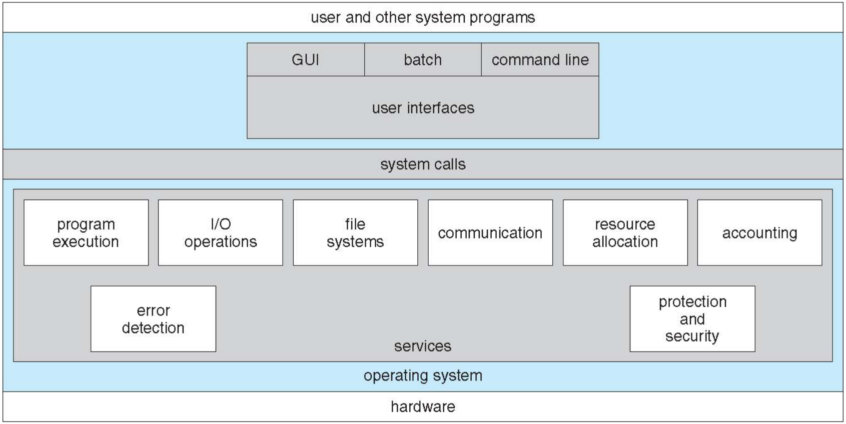
- Main memory - only large storage media that the CPU can access directly
- Secondary storage - provides large non-volatile storage capacity
- Important example: magnetic disks - rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into tracks, which are subdivided into sectors
  - The disk controller determines the logical interaction between the device and the computer
- Today also often flash memory. However, same logical division intro tracks and sectors still used

Storage Structure: Memory vs. Secondary Storage



OS Services

OS Structure with Services



Interfaces: Interacting 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.* Nowadays, this is the norm.
  - A command-line interface for running processes and scripts, browsing files in directories, *etc.* For Computer Scientists only.
- **Processes** (the programs in execution) interact with the Operating System by making *system calls* into the operating system *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

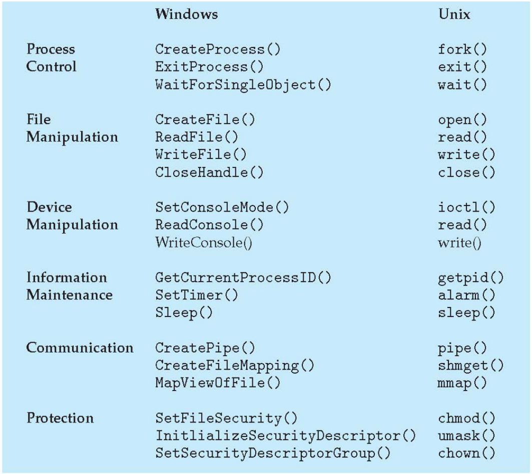
Services for the OS Itself

- 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

System Calls

- Programming interface to the services provided by the OS (*e.g.* open file, read file, *etc.*)
- Typically written in C (or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call.
- Two common APIs: Win32 API for Windows, POSIX API for UNIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X).
- The APIs allow application code to access the kernel without requiring root privileges: it transfers control from **user mode** to **kernel mode**.

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()

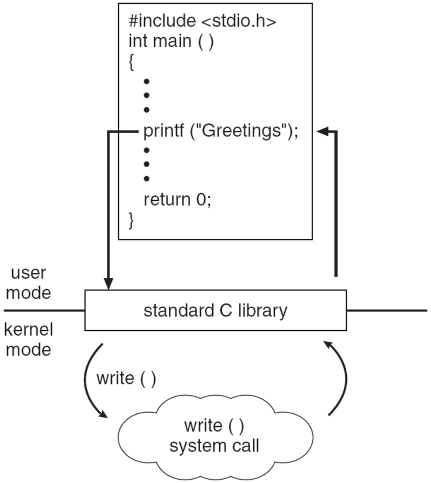
Device Manipulation	SetConsoleMode()
	ReadConsole()
	WriteConsole()
		ioctl()
		read()
		write()

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



System calls for file operations

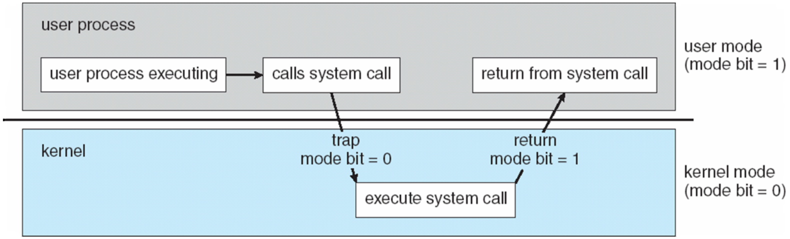
Have the following operations for files:

open	Register the file with the operating system. Must be called before any operation on the file. Returns an integer called the file descriptor - an index into the list of open files maintained by the OS.
read	Read data from the file. Returns number of bytes read or 0 for end of file.
write	Write data to a file. Returns number of bytes written.
close	De-registers the file with the operating system. No further operations on the file are possible.

These system calls return a negative number on error. Can use `perror`-function to display error.

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



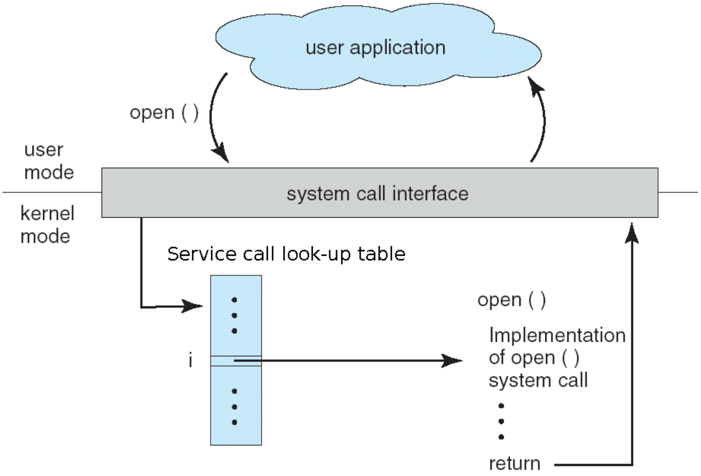
Note: **Wrapper** is a function that only makes the System Call, it has no other functionality.

Trapping to the Kernel

There is some ‘magic’ happening to move from user mode to kernel mode:

- 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



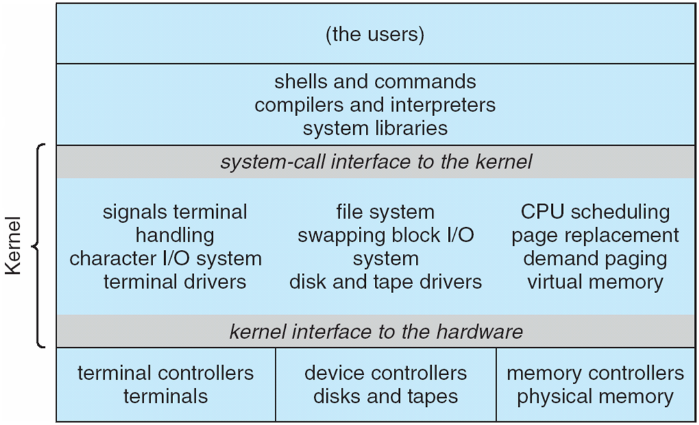
OS Architecture

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.

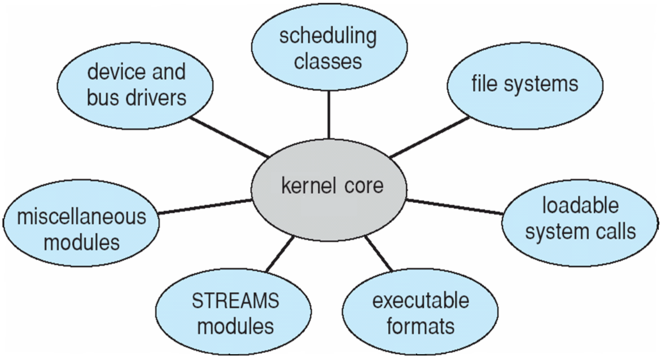
Traditional UNIX



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



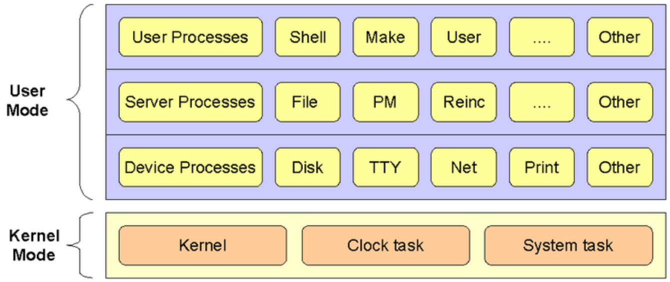
Microkernel

- 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 and L3/L4 are examples of microkernel architecture

Microkernel: MINIX



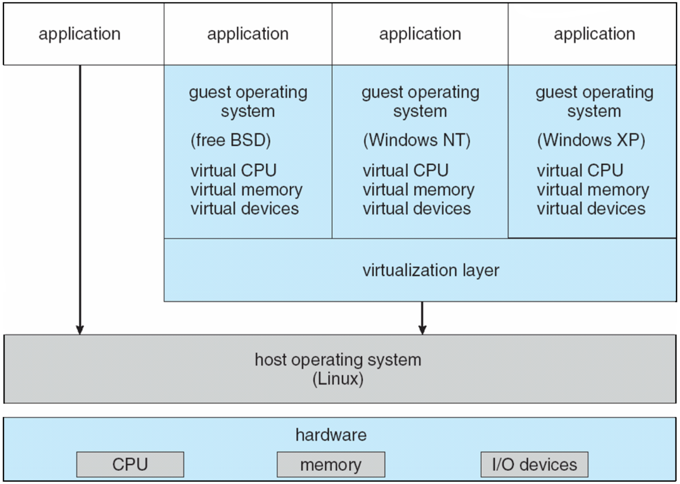
The MINIX 3 Microkernel Architecture

Virtual Machines

Virtual Machines

- A virtual machine allows to run one operating system (the guest) on another operating system (the host)
- 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 10 as a guest operating system on Linux.

VM Architecture



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 Virtualization Format" (OVF): 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.* Amazon EC2, Rackspace)

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.

Linux Device Drivers

Device Drivers and Device Controllers

- We already saw that a device controller is the hardware needed within laptop, PC or other computer to connect to an external device.
- A **device driver** is *software* that allows other programs to interface with external devices through the device controllers.
- Device drivers run in kernel mode (some device management software may well run in user mode, but then typically these wouldn't be called 'drivers').

Device drivers

View from user space:
Have special file in `/dev` associated with it, together with five systems calls:

- `open`: make device available
- `read`: read from device
- `write`: write to device
- `ioctl`: Perform operations on device (optional)
- `close`: make device unavailable

Kernel side

Each file may have functions associated with it which are called when corresponding system calls are made

`linux/fs.h` lists all available operations on files

Device driver implements at least functions for `open`, `read`, `write` and `close`.

Categorising devices

Kernel also keeps track of

- Physical dependencies between devices. Example: devices connected to a USB-hub
- Buses: Channels between processor and one or more devices. Can be either physical (eg pci, usb), or logical
- Classes: Sets of devices of the same type, eg keyboards, mice

Handling Interrupts in Device Drivers

Normal cycle of interrupt handling for devices:

- Device sends interrupt
- CPU selects appropriate interrupt handler
- Interrupt handler processes interrupt
  Two important tasks to be done:
  - Data to be transferred to/from device
  - Waking up processes which wait for data transfer to be finished
- Interrupt handler clears interrupt bit of device
  Necessary for next interrupt to arrive

Interrupt processing time must be as short as possible
Data transfer fast, rest of processing slow
⇒ Separate interrupt processing in two halves:

- Top Half is called directly by interrupt handler
  Only transfers data between device and appropriate kernel buffer and schedules software interrupt to start Bottom half
- Bottom half still runs in interrupt context and does the rest of the processing (eg working through the protocol stack, and waking up processes)