Overview

- We first look briefly at how an operating system gets loaded and started
- As an operating system sits between the application and hardware layers it must deal with both
- We examine the system service interface and how system calls are invoked
- We briefly look at the hardware interface, how the operating system and hardware devices talk to each other

Booting

What happens when we turn on a computer?

- Your machine is hardwired such that, when switched on, the CPU executes code stored in ROM called the Basic Input/Output System (BIOS)
- This code first performs a Power-On Self Test (POST) to check machine hardware is functioning correctly
- With the POST passed the BIOS searches for active bootable devices in the order specified by the CMOS (complementary metal oxide semiconductor) settings, e.g.
  1. CDROM
  2. Floppy disk
  3. Hard disk

Booting

- Once an active boot device is located the BIOS loads from it its Master Boot Record (MBR)
- The MBR holds a partition table describing disk organisation
- The partition table is searched for an active partition i.e. one marked bootable
- The active partition's Partition Boot Record (PBR) is located and read and it goes on to boot the OS

Booting

What if we have more than one operating system installed?

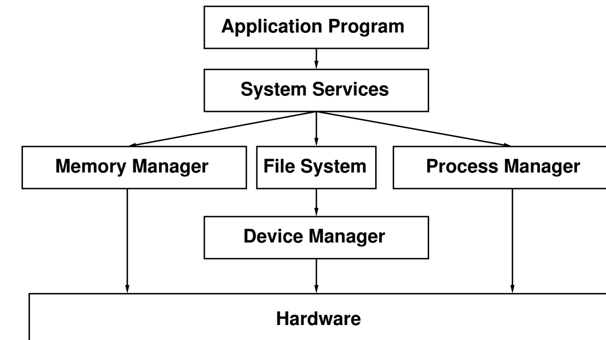
- We could manually reset the active partition in the MBR every time we wanted to boot a new operating system but this would be a rather awkward approach
- Instead we have the MBR (or PBR) launch a sophisticated "boot loader" that allows us to pick from a list which OS we would like to boot
- Examples of such boot loaders include LILO (LInux LOader) and GRUB (GRand Unified Boot Loader)
- These loaders know how to boot a variety of OSes across a variety of file systems (installed on our lab machines)

Operating System Design

- An operating system is a large piece of software (Windows Vista contains over 50 million lines of code)
- Clearly it must be built in layers with each layer dependent on the one beneath and a modular design allows new modules to be added without serious knock-on effects
- Linux and Windows are “monolithic kernels” i.e. although the kernel is composed of a number of modules they are assembled to form one big program
- Microkernels take a different approach, with little functionality in the kernel, userland components do most of the work and pass messages to each other

Operating System Design

- In this course we study a selection of the modules that go into making up an operating system



Operating System Interfaces

- The operating system sits between the application and hardware layers
- Requests and data must be passed from one layer to the next and two interfaces define how this happens
- Userland applications talk to the operating system through the system call interface
- The operating system talks to the hardware through the hardware interface

System Service Interface

- Like applications make talking to the operating system more user-friendly so an operating system makes the hardware more user-friendly
- Applications talk to the operating system by invoking the system services it provides: this is “systems programming” i.e. writing code that makes system calls
- System services are functions the operating system offers to perform for applications and there are about 200 of them in Linux
- To the programmer, a system call looks like a function call but the code executes inside the kernel, in kernel mode

System Service Interface

- For example, compilers translate high-level language I/O requests into `read` and `write` system calls
- As assembly instructions define what we can ask of the CPU, system services define what we can ask of the operating system and are therefore sometimes referred to as the "software instruction set"
- Written in C and compiled into the kernel, system calls are described fully in online manual pages and most return an integer which should be checked for success or failure

System Service Interface

- CPU operates in two modes:
  1. User mode: subset of instructions available
  2. Kernel mode: all, including privileged, instructions available
- In user mode, the CPU can execute only a subset of its instruction set, the "dangerous" instructions being disallowed for security reasons
- In kernel mode no such restriction applies, the full instruction set is available to the CPU

System Service Interface

- Can a programmer switch to kernel mode and execute dangerous instructions?
- No, the only way she can enter kernel mode is via a system call and so only the trusted services provided the operating system will ever be executed in kernel mode
- Since Linux system calls are written in C, the simplest way of invoking them is from a C program (this is what we do in the labs)

System Service Interface

Steps in using the `read` system call

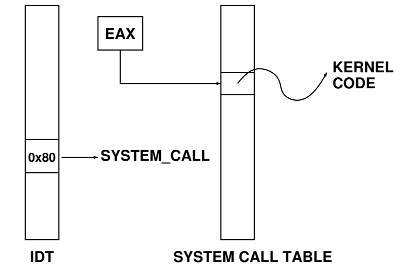
- To invoke the `read` system call your program fills in some registers and generates a software interrupt
- One register (`EAX`) says which system call the program wants to invoke
- Other registers contain extra information your program wishes to make available to the kernel
- The interrupt number the program generates is `0x80`

System Service Interface

- When the interrupt occurs we enter kernel mode
- The kernel looks up how to handle interrupt number `0x80` in its Interrupt Descriptor Table (IDT)
- This leads the kernel to its system call handling code
- The kernel then looks up how to handle this particular system call (`read` was recorded in `EAX`) in its System Call Table (SCT)
- It then executes the code that handles this system call before returning to run the rest of your program

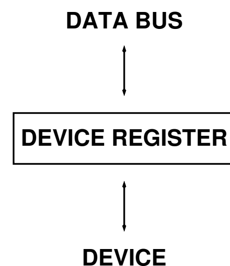
System Service Interface

- Two tables involved: Interrupt Descriptor Table (IDT) and System Call Table (SCT)
- In IDT, only `0x80` entry involved
- In SCT, `EAX` entry involved i.e. a variable which can index any entry in the SCT depending on the system call invoked



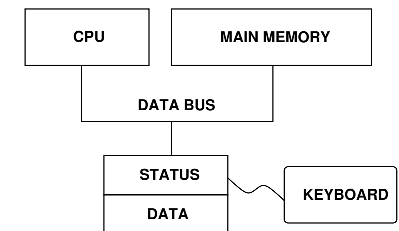
Hardware Interface

- How is data transferred between attached hardware devices and the operating system?
- Through <u>dual ported</u> device registers
- Both operating system and device can read data from/write data to such registers
- Synchronisation is a problem with such a simple arrangement



Hardware Interface

- New data may overwrite existing data or the operating system may grab same data twice
- The device needs a way to tell the operating system that some data is available
- The operating system needs a way to tell the device it got that data and is ready for more
- So a status register added and the device and operating system use it to keep each other informed

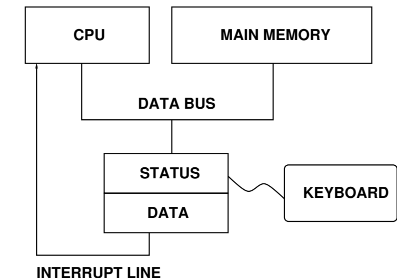


Hardware Interface

- There may be many devices attached, each with its own status register, how does the operating system know which ones have something interesting to say?
- Continuous polling of status registers?
- Could do so but the overhead would be too costly
- Devices need a way of attracting the CPU's attention
- An interrupt line is added and links devices to the CPU

Hardware Interface

- When data arrives the device alerts the CPU with an interrupt, CPU finishes the current instruction, polls to find the alerting device
- With many devices attached and hundreds of interrupts per second, we must do a better job of finding the device in need of attention



Hardware Interface

Single interrupt line

- New acknowledge line added
- CPU receives interrupt and asks "Who's calling?"
- Device sends back a number
- CPU uses this number as an index into the IDT where it finds a pointer to the code for handling that interrupt
- Entry in IDT points to device's interrupt service routine (part of a device driver)

Hardware Interface

Multiple interrupt lines

- Each line linked to a group of devices
- On receiving an interrupt the CPU polls corresponding group to find particular device responsible

Hardware Interface

Are all interrupts equal?

- No, interrupts have different priority levels
- When servicing an interrupt the CPU runs at the corresponding interrupt level and cannot be interrupted by same or lower priority interrupt
- Implementation handled by PIC, programmable interrupt controller and devices are registered with it at boot time
- System clock has highest priority interrupt and allows operating system to regain regular system control
- Lower priority interrupts can be interrupted by ones of a higher priority

Hardware Interface

Direct Memory Access (DMA)

- The above discussion assumes a single data register
- Rather inefficient for playing or recording 44.1kHz stereo audio in PCM format: that's a lot of interrupts
- With DMA the operating system specifies where and how much data is to be sent to or read from a device
- DMA (Direct Memory Access) controller does the transfer in the background and interrupts the CPU when all is done
- Meanwhile the CPU is getting on with useful work