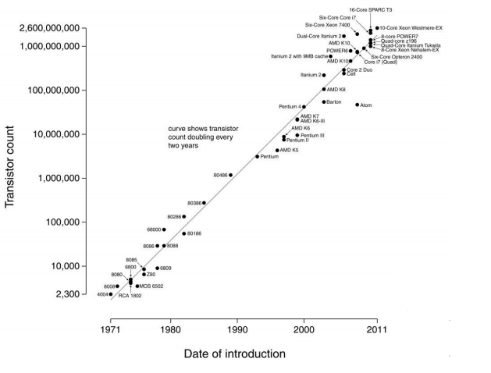
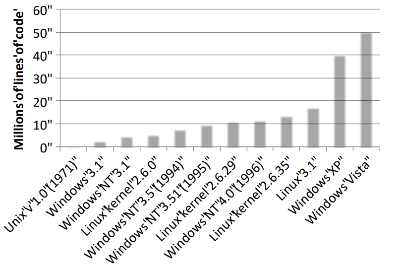
Operating Systems

Introduction

Hardware Complexity Increases

The number of transistors in chips has increased dramatically since the first transistor based microprocessor. This increase has followed a trend of roughly doubling every year, following what is known as Moore’s Law.

Software Complexity Increases

The hardware isn’t the only part of computers that’s gotten more complicated, as this graph shows Operating systems have become more and more complex to match.

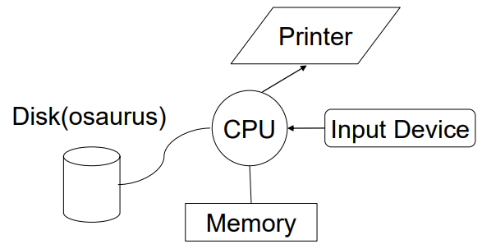
Hardware/Software Changes With Time

* 1960s: mainframe computers (IBM)
* 1970s: minicomputers (DEC)
* 1980s: microprocessors and workstations (SUN), local-area networking, the Internet
* 1990s: PCs (rise of Microsoft, Intel, Dell), the Web
* 2000s:
  + Internet Services/ Clusters (Amazon)
  + General Cloud Computing (Google, Amazon, Microsoft)
  + Mobile/ubiquitous/embedded computing (iPod, iPhone, iPad, Android)
* 2010s: Sensor networks, “data-intensive computing”, computers and the physical world
* 2020s: it’s up to us!

OS History

Operating systems are the result of a 60 year long evolutionary process, they were born out of need. Learning how they came about helps make clear what some of their functions are and why.

In 1943 T.J. Watson, the man who created IBM, famously said “I think there is a world market for maybe five computers”. Only seven years later in 1950, there are maybe 20 computers in the world. These computers were unbelievably expensive resulting in machine time being more valuable than human time, as such efficient use of this hardware was paramount. Operating systems were born out of this need for efficient use, modern operating systems are still fundamentally motivated by the same needs as in 1950.

Primordial Computer

The original computers were essentially just a CPU connected to Memory and a disk with only a keyboard for input and a printer for output (no monitors at the time).

The OS as a Linked Library

For these basic computers, the operating system was just a library of code that you linked into your program. Programs were loaded in their entirety into memory, and executed and the ‘OS’ had an ‘API’ that let you control the disk, or the printer etc… .The interfaces were literally switches and blinking lights and when you were done with running a program, you would leave and turn the computer over to the next person.

Asynchronous I/O

Initially the disk was really slow, and accessing the disk was done by the CPU, taking time that could have been used executing instructions. As such hardware was created so the disk could operate without tying up the CPU, this is known as the disk controller.

Programmers could now right code that starts an I/O, goes off and does something else, and then checks if the I/O is done later.

The upside of this is that it increases CPU utilisation, though it’s difficult to get right and its benefits are job specific.

Multiprogramming

To further increase system utilisation, multiprogramming OSs were developed. These allow multiple runnable jobs in memory at once, it overlaps the I/O of one job with the computing of another, while one job waits for I/O completion, another job gets to use the CPU. This can remove the need for asynchronous I/O within individual jobs simplifying the job of programmers as only the OS programmer needs to deal with asynchronous events. Polling or interrupts can be used to tell when devices are done.

Timesharing

To support interactive use, create a timesharing OS. There are multiple terminals for one machine, each user has illusions of the entire machine to themselves and the OS is optimized for response time, perhaps at the cost of throughput.

Timslicing is where you divide the CPU equally among the users, if the job is truly interactive, then you can jump between programs and users faster than users can generate and load them. This permits users to interactively view, edit and debug running programs.

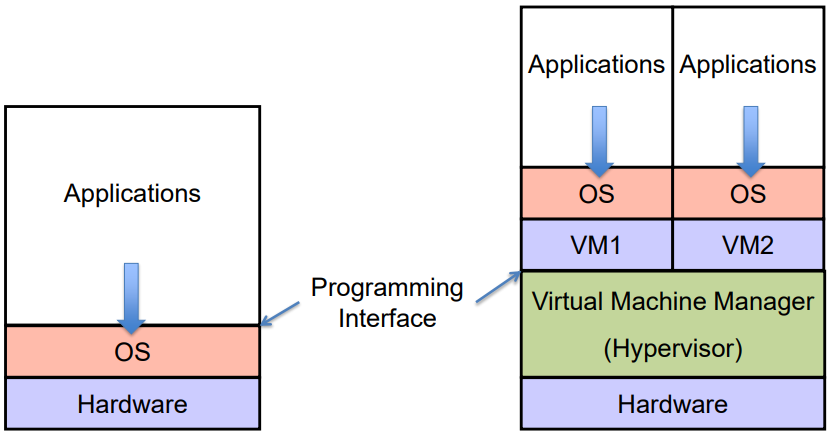
Parallel Systems

Some applications can be written as multiple parallel threads or processes. This can speed up the execution by running multiple threads/processes simultaneously on multiple CPUs. This requires an on and language primitives for dividing the program into multiple parallel activities and needs OS primitives for fast communication among activities with the degree of speedup dictated by the communication/computation ratio. Nowadays there are many different ‘flavours’ of parallel computers.

Distributed OS

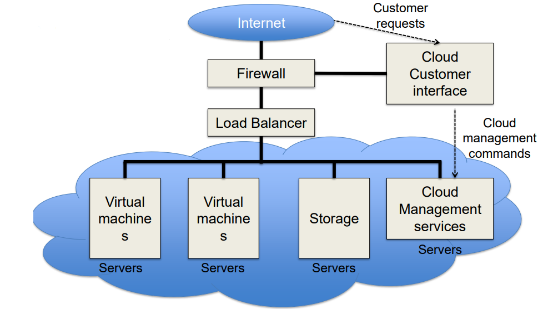
When a single computer isn’t enough to solve a problem we may need to use multiple all at once, such as supercomputers, cluster or data-centres etc… this brought rise to distributed systems.

Distributed systems facilitate the use of geographically distributed resources, e.g. work stations on a LAN or servers across the internet. It supports applications running among multiple computers, with interprocess communication (message passing and shared memory) and networking stacks. This also brought in the concept of sharing distributed resources (both hardware and software).

An evolution of the distributed system would be the client/server model for computing, for example mail server/service, file server/service, print server/service, compute etc… or Peer-to-Peer (p2p) systems where everyone is a client and server at the same time.

Virtualization

Virtualisation is a highly used technique, virtualisation applies the idea of timesharing to Operating systems. A virtual machine manager creates virtual machines for different operating systems, which each think they see the whole computer and are the only operating system while in reality the VMM is managing it all.

Cloud Computing

Cloud computing is one of the most used technologies nowadays. In a datacenter we run multiple services on many servers. This allows us to run application services on systems owned an maintained by other people, only paying for the computation time used, not the hardware.

OS Issues

* Structure: how is the OS organised?
* Sharing : how are resources shared across users?
* Naming: how are resources named (by users or programs)?
* Security: how is the integrity of the OS and its resources ensured?
* Protection: how is one user/program protected from another?
* Performance: how do we make it all go fast?
* Reliability: what happens if something goes wrong (either with hardware or with a program)?
* Extensibility: can we add new features?
* Communication: how do programs exchange information, including across a network?
* Concurrency: how are parallel activities (computation and I/O) created and controlled?
* Scale: what happens as demands or resources increase?
* Persistence: how do you make data last longer than program executions?
* Distribution: how do multiple computer interact with each other?
* Accounting: how do we keep track of resource usage, and perhaps charge for it?

Structure

Hardware Architecture Affects (and is Affected by) The OS

The operating systems supports sharing and protection of the hardware, e.g. multiple applications can run concurrently, sharing hardware resources and a buggy or malicious application should not disrupt other applications or the system.

The hardware architecture, in turn, determines what is viable (reasonably efficient, or even possible) for the OS, this includes the instruction set (synchronisation, I/O,….) and hardware components like MMU, DMA controllers etc…

Hardware Architecture Support for the OS

Architectural support can simply OS tasks for example:

* Early PC OSes (DOS, MacOS) lacked support for virtual memory, at that time PCs lacked necessary hardware support (MMU).
* Until recently, intel-based PCs didn’t support 64bit addresses, 64bit addressing has been available for decades on other hardware architectures (MIPS, ALPHA, IBM, etc), this change was driven by AMD’s 64-bit architecture.

Hardware Architectural Features Affecting OS

At the beginning, hardware and software were designed essentially together in order to improve performance, this isn’t the case anymore.

Nowadays many features of hardware are primarily built to support the OS:

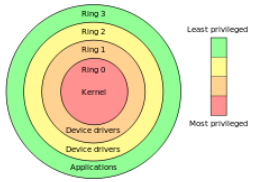
* Timer (clock) operations
* Memory protection
* I/O control operations
* Interrupts
* Protected mode(s) of execution
  + Kernel vs user mode
  + Privileged instructions
  + System calls
* Virtualization

Privileged Instructions

Privileged instructions are instructions that are restricted to the OS, meaning user programs can’t use them. For example only the OS can

* Directly access some classes of I/O devices
* Manipulate memory state management
  + Page table pointers, TLB loads, etc
* Manipulate special ‘mode bits’
  + Interrupt priority level

These restrictions provide safety and security

OS Protection

How does the processor know if a privileged instruction can be executed? The architecture must support at least two modes of operation, kernel mode and user mode (x86 supports 4 protection modes (rings)). This mode is set by a status bit in a protected processor register, user programs execute in user mode and the OS kernel executes in kernel (privileged, supervisor) mode.

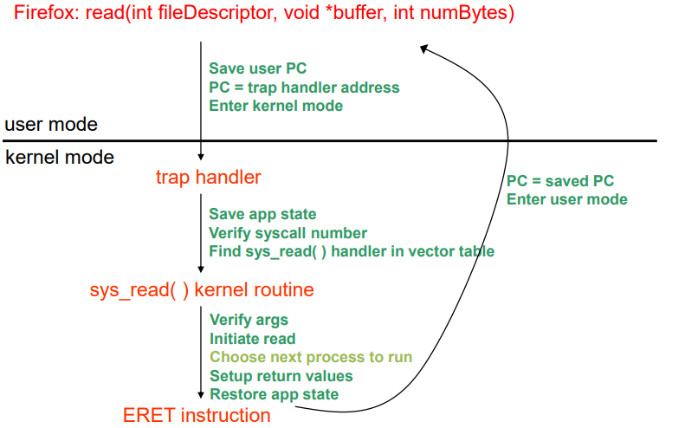
Privileged instructions can only be executed in kernel mode, when code running in user attempts to execute a privileged instruction the “Privileged Instruction” exception is triggered.

Crossing Protection Boundaries

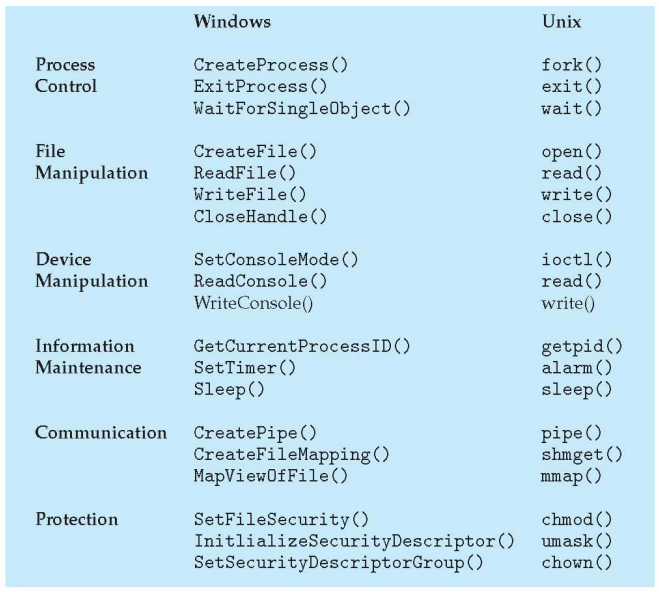
So how do uses perform a privileged activity, e.g. writing to disk, if you can’t execute an I/O instruction?

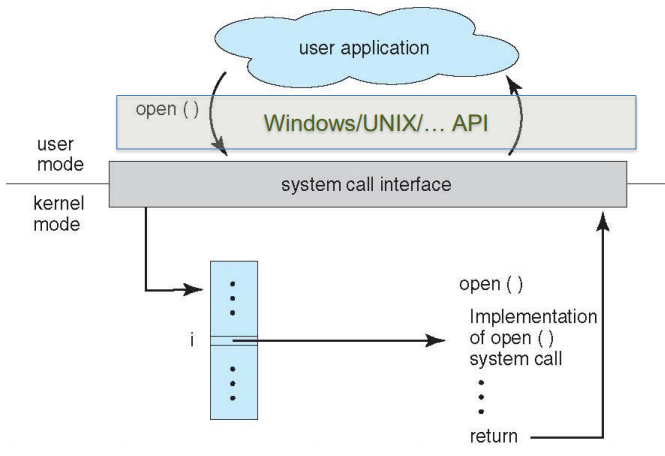
User programs must call an OS procedure, essentially asking the OS to do it for them. The OS defines a set of system calls and the user-mode program executes the appropriate system call instruction.

Syscall

The syscall instruction is a single instruction, that (on a single CPU/core) atomically saves the current PC, sets the execution mode to privileged and sets the PC to a handler address.

This is similar to a procedure call, the caller puts arguments in a place the callee expects (usually registers), one arg is a syscall number, indicating what OS function to call, the callee (OS) saves the caller’s state (registers etc..) so it can use the CPU. The OS function code runs (the arguments must be verified by the OS) and the OS returns using a special instruction that automatically sets the PC to return address and sets execution back to user mode.

What Syscall to Run

When a syscall is used, the system call interface checks a vector of pointers that store the location of each function implementation and runs that code.

System Call vs Subroutine Call

A syscall is not the same as a subroutine call with the caller specifying the next PC. The caller knows where the subroutines are located in memory, whereas a syscall is an ID, Subroutines trust each other and all subroutines share memory, this is not the case with syscalls.

The kernel saves the state when a syscall is used to prevent overwriting values, it also verifies arguments to prevent buggy code crashing the system, the kernel also has its own buffer to which data is copied for system calls.

OS Services

All entries to the OS occur via the mechanism we’ve shown, acquiring privileged mode and branching to the trap handler are inseparable.

Privileged instructions and resource sharing are the basis for almost everything OS-related, memory protection, protected I/O, limiting user resource consumption etc…

Important terminology:

* Exception: synchronous; unexpected problem with code
* Syscall: synchronous; intended transition to OS
* Interrupt: asynchronous; caused by an external device