

PB152 Operating Systems

Petr Ročkal

Part A: Preliminaries

These are **draft** (work in progress) lecture notes for PB152. It is expected to be gradually filled in, **hopefully** before the semester ends. For now, it is mostly a printable version of the slides.

Organisation 3

- lectures, with an **optional** seminar
- **written exam** at the end
 - multiple choice
 - free-form questions
- 1 online test mid-term, 1 before exam
 - mainly training for the exam proper

The written exam will consist of two parts, a multiple-choice test (with exactly one correct answer on each question) and a free-form question. Additionally, you will be required to pass a mid-term test with the same pool of multiple-choice questions. The question pool is known in advance: both the entire mid-term and the multiple-choice part of the final exam will contain questions that are listed in these notes (on the last slide in each lecture).

The possible answers will be, however, **not** known in advance, and will be randomized for each instance of the test. Do **not** try to learn the correct answer by its wording: you need to understand the questions and the correct answers to pass the test.

Seminars 4

- a separate, optional course (code **PB152cv**)
- covers operating systems from a practical perspective
- get your hands on the things we'll talk about here
- offers additional practice with C programming

The seminar is a good way to gain some practical experience with operating systems. It will mainly focus on interaction of programs with OS services, but will also cover user-level issues like virtualisation, OS installation and shell scripting.

Mid-Term and End-Term Tests 5

- **24 hours** to complete, **2 attempts** possible
- 10 questions, picked from review questions
 - mid-term → first 24, end-term second 24
- 8 out of 10 is required on each of them
- **preliminary** mid-term date: **7.4., 6pm**

Even though passing the mid-term/end-term online tests is easy (and it is easy for you to collaborate), I strongly suggest that you use the opportunity to evaluate your knowledge for yourself, honestly. The same questions will appear on the exam, and this time, it won't be possible to consult the slides, the internet or your friends. Also, you will be only allowed 1 mistake (out of 10 questions). Use the few opportunities to practice for the exam wisely.

Study Materials 6

- this course is undergoing a major update
- **lecture slides** will be in the **IS**
 - they will be added as we go
- you can also use slides from **previous years**
 - they are already in study materials
 - **but**: not everything is covered in those

You are reading the lecture notes for the course PB152 Operating Systems. This is a supplementary resource based on the lecture slides, but with additional details that would not fit into the slide format. These lecture notes should be self-contained in the sense that they only rely on knowledge you have from other courses, like PB150 Computer Systems (or PB151) or PB071 Principles of Low-Level programming. Likewise, being familiar with the topics covered in these lecture notes is sufficient to pass the exam.

Books 7

- there are a few good OS books
- you are encouraged to get and read them
- A. Tanenbaum: Modern Operating Systems
- A. Silberschatz et al.: Operating System Concepts
- L. Skočovský: Principy a problémy OS UNIX
- W. Stallings: Operating Systems, Internals and Design
- many others, feel free to explore

The books mentioned here usually cover a lot more ground than it is possible to include in a single-semester course. The study of operating systems is, however, very important in many sub-fields of computer science, and also in most programming disciplines. Spending extra time on this topic will likely be well worth your time.

Topics 8

1. Anatomy of an OS
2. System Libraries and APIs
3. The Kernel
4. File Systems
5. Basic Resources and Multiplexing
6. Concurrency and Locking

In the first half of the semester, we will deal with the basic components and abstractions used in general-purpose operating systems. The first lecture will simply give an overview of the entire OS and will attempt to give you an idea how those fit together. In the second lecture, we will cover the basic programming interfaces provided by the OS, provided mainly by system libraries.

Topics (cont'd)

9

- 7. Device Drivers
- 8. Network Stack
- 9. Command Interpreters & User Interfaces
- 10. Users and Permissions
- 11. Virtualisation & Containers
- 12. Special-Purpose Operating Systems

The second half of the semester will start with device drivers, which form an important part of operating systems in general, since they mediate communication between application software and hardware peripherals connected to the computer. In a similar fashion, the network stack allows programs to communicate with other computers (and software running on those other computers) that are attached to a computer network.

Related Courses

10

- PB150/PB151 Computer Systems
- PB153 Operating Systems and their Interfaces
- PA150 Advanced OS Concepts
- PV062 File Structures
- PB071 Principles of Low-level programming
- PB173 Domain-specific Development in C/C++

There is a number of courses that overlap, provide prerequisite knowledge or extend what you will learn here. The list above is incomplete. The course PB153 is an alternative to this course. Most students are expected to take PB071 in parallel with this course, even though knowledge of C won't be required for the theory we cover. However, C basics will be needed for the optional seminar (PB152cv).

Organisation of the Semester

11

- generally, **one lecture = one topic**
- there will be most likely 13 lectures
- the 13th lecture will be review
- **online mid-term** in April

Part B: Semester Overview

This section gives a high-level overview of the topics that will be covered in individual lectures. Think of it as an extended table of contents, or as a collection of abstracts, one for each of the upcoming lectures.

2 System Libraries and APIs

13

- **POSIX**: Portable Operating System Interface
- UNIX: (almost) everything is a **file**
- the least common denominator of programs: C
- user view: objects, archives, shared libraries
- **compiler**, linker

System libraries and their APIs provide the most direct access to operating system services. In the second lecture, we will explore how programs access those services and how the system libraries tie into the C programming language. We will also deal with basic artifacts that make up programs: object files, archive files, shared libraries and how those come about: how we go from a C source file all the way to an executable file through compilation and linking. Throughout this lecture, we will use POSIX as our go-to source of examples, since it is the operating system interface that is most widely implemented. Moreover, there is abundance of documentation and resources both online and offline.

3 The Kernel

14

- **privileged** CPU mode
- the boot process
- boundary enforcement
- kernel designs: micro, **mono**, exo, ...
- **system calls**

In the third lecture, we will focus on the kernel, arguably the most important (and often the most complicated) part of an operating system. We will start from the beginning, with the boot process: how the

kernel is loaded into memory, initialises the hardware and starts the user-space components (that is, everything that is not the kernel) of the operating system.

We will then talk about boundary enforcement: how the kernel polices user processes so they cannot interfere with each other, or with the underlying hardware devices. We will touch on how this enforcement makes it possible to allow multiple users to share a single computer without infringing on each other (or at least limiting any such infringement).

Another topic of considerable interest will be how kernels are designed and what is and what isn't part of the kernel proper. We will explore some of the trade-offs involved in the various designs, especially with regards to security and correctness vs performance.

Finally, we will look at the **system call** mechanism, which is how the user-space communicates with the kernel, and requests various low-level operating system services.

4 File Systems

15

- why and how
- abstraction over shared **block storage**
- directory **hierarchy**
- everything is a file revisited
- i-nodes, directories, hard & soft links

Next up are file systems, which are a very widely used abstraction on top of persistent block storage, which is what hardware storage devices provide. We will ask ourselves, first of all, why filesystems are important and why they are so pervasively implemented in operating systems, and then we will look at how they work on the inside. In particular, we will explore the traditional UNIX filesystem, which offers important insights about the architecture of the operating system as a whole, and about important aspects of the POSIX file semantics.

5 Basic Resources and Multiplexing

16

- **virtual memory**, processes
- sharing CPUs & **scheduling**
- processes vs threads
- **interrupts**, clocks

One of the basic roles of the operating system is management of various resources, starting with the most basic: the CPU cores and the RAM. Since those resources are very important to every process or program, we will spend the entire lecture on them. In particular, we will look at the basic units of resource assignment: threads for the CPU and processes for memory. We will also look at the mechanisms used by the kernel to implement assignment and protection of those resources, namely the virtual memory subsystem and the scheduler.

6 Concurrency and Locking

17

- inter-process **communication**
- accessing **shared resources**
- mutual exclusion
- **deadlocks** and deadlock prevention

Scheduling and slicing of CPU time is closely related to another important topic that pervades operating system design: concurrency. We will take a high-level, introductory look at this topic, since the details are often complicated, architecture-specific and require deep understanding of both hardware (SMP, cache hierarchies) and of kernels.

7 Device Drivers

18

- user vs kernel drivers
- interrupts &c.
- GPU
- PCI &c.
- block storage
- network devices, wifi
- USB
- bluetooth

One of the fundamental roles of an operating system is to mediate access to hardware devices. Some of the code that provides hardware access deals mainly with the software interfaces and APIs – this is known as hardware abstraction. However, to make this abstraction work, there is often a large amount of device-specific (or at least device-class-specific) ‘glue’ – also known as device drivers.

One of the important questions will be the interplay between processor-level protections and direct hardware access and what this means for drivers. We will see that for some (but not all) types of hardware, only privileged programs (either inside the kernel, or close to it) can reasonably mediate between hardware itself and between higher levels of the system (hardware abstraction layer, application software, etc.).

8 Network Stack

19

- TCP/IP
- name resolution
- socket APIs
- firewalls and packet filters
- network file systems

While there is a dedicated course about networking, we will spend one

of our lectures talking about networks: in modern operating systems, networking is an integral part of the package and networking considerations often influence other parts of the system. We will look at the ubiquitous TCP/IP stack, how it integrates into an operating system and what are the APIs that applications can use to take advantage of network services. We will also touch on network-related functionality that is often deeply integrated into operating systems: packet filtering and network file systems.

9 Command Interpreters & User Interfaces

20

- **interactive** systems
- history: consoles and terminals
- **text-based** terminals, RS-232
- bash and other Bourne-style shells, POSIX
- **graphical**: X11, Wayland, OS X, Windows, Android, iOS

The next lecture will focus on human-computer interaction, which is clearly a central aspect of the experience of using a computer and is therefore an important part of most general-purpose operating systems. Even computers that do not directly (physically) interact with humans usually present some form of an interface, usually mediated over the network.

We will first look at ‘traditional’ text-based interfaces, which are still in common use among system and network engineers and computer programmers, but we will also look in some depth at the graphics stacks that power modern devices (up to and including smartphones).

10 Users and Permissions

21

- **multi-user** systems
- **isolation**, ownership
- file system **permissions**
- capabilities

There are two important use-cases for computers (and hence operating systems) in which higher-level access control and permission management is important: first, when a single computer is shared by multiple users (this is the more traditional case), but in more modern times, also whenever we execute untrusted or semi-trusted programs on our devices (think application permissions on smartphones, web pages that execute javascript on your laptop and so on).

11 Virtualisation & Containers

22

- resource multiplexing redux
- **isolation** redux
- multiple kernels on a single system
- type 1 and type 2 **hypervisors**
- **virtio**

A computer, along with its operating system, is a natural ‘unit’ of computation resources – it conveniently packages up the resources themselves, with a software stack and configuration. Unfortunately, computers – being physical devices – are somewhat inflexible and unwieldy: they have to be procured, placed in racks in air-conditioned rooms, attached to a power source, to each other and to the larger network. Their physical components are prone to wear and failure, and need to be replaced or repaired regularly.

Virtualization makes it possible to detach the logical aspects of a computer – its installed software, data storage and configuration – from the physical box. This improves hardware utilization, decouples hardware maintenance from software aspects and makes everyone’s life easier

(most of the time, anyway).

In this lecture, we will peek under the hood of modern hypervisor-based virtual machines and how they are implemented in the current generation of operating systems.

12 Special-Purpose Operating Systems

23

- general-purpose vs special-purpose
- **embedded** systems
- **real-time** systems
- high-assurance systems (seL4)

Part 1: Anatomy of an OS

In the first lecture, we will first pose the question “what is an operating system” and give some short, but largely unsatisfactory answers. Since an operating system is a complex system, it is built from a number of components. Each of those components is described more easily than the entire operating system, and for this reason, we will attempt to understand an operating system as the sum of its parts.

Lecture Overview

25

1. Components
2. Interfaces
3. Classification

After talking about what is an operating system, we will give more details about its components and afterwards move on to the interfaces between those components. Finally, we will look at classifying operating systems: this is another angle that could help us pin down what an operating system is.

What is an OS?

26

- the **software** that makes the hardware tick
- and makes other software easier to write

Also

- catch-all phrase for **low-level** software
- an **abstraction layer** over the machine
- but the boundaries are not always clear

Our first (very approximate) attempt at defining an OS is via its responsibilities towards hardware. Since it sits between hardware and the rest of software, in some sense, it is what makes the hardware work. Modern hardware alone is rarely capable of achieving anything useful on its own. It needs to be programmed, and the basic layer of programming is provided by the operating system.

What is **not** (part of) an OS?

27

- firmware: (very) low level software
 - much more **hardware-specific** than an OS
 - often executes on auxiliary processors
- application software
 - runs **on top** of an operating system
 - this is what you got the computer for
 - eg. games, spreadsheets, photo editing, ...

Throughout most of the course, we will have talked about general-purpose operating systems: those that run on personal computers and servers. The last lecture will be dedicated to more specialised systems: those that run in washing machines, on satellites or on the Mars rovers. We will also briefly cover high-assurance systems, which focus on extremely high reliability and/or security.

One approach to understand what **is** an operating system could be to look at things that are related, but are **not** an operating system, nor a part of one. There is one additional software-ish layer below the operating system, usually known as **firmware**. In a typical computer, many pieces of firmware are present, but most of them execute on **auxiliary** processors – e.g. those in a WiFi card, or in the graphics subsystem, in the hard drive and so on. In contrast, the operating system runs on the main processor. There is one piece of firmware that typically runs on the main CPU: on older systems, it's known as BIOS, on modern systems it is simply known as “the firmware”.

In the other direction, on top of an operating system, there is a whole bunch of **application software**. While some software of this type might be **bundled** with an operating system, it is not, strictly speaking, a part of it. Application software is the programs that you use to get things done, like text editors, word processors, but also programming IDEs (integrated development environment), computer games or web applications (do I say Facebook?). And so on and so forth.

What does an OS do?

28

- **interact** with the user
- **manage** and multiplex **hardware**
- **manage** other **software**
- **organises** and manages **data**
- provides **services** for other programs
- enforces **security**

The tasks and duties that the operating system performs are rather varied. On one side, it takes care of the basic interaction with the user: a command interpreter, a graphical user interface or batch-mode job processing system with input provided as punch cards. Then there is the hardware, which needs to be managed and shared between individual programs and users. Installation of additional (application) software is another of the responsibilities of an operating system. Organisation and management of data is a major task as well: this is what file systems do. This again includes access control and sharing among users of the underlying hardware which stores the actual bits and bytes.

Finally, there is the third side that the operating system interfaces with: the application software. In addition to the user and the hardware, application programs need operating services to be able to perform their function. Among other things, they need to interact with users and use hardware resources, after all. It is the operating system that is in charge of both.

Part 1.1: Components

In this section, we will consider what an operating **consists of**, as means to understand what it **is**.

What is an OS **made of**? 30

- the kernel
- system libraries
- system daemons / services
- user interface
- system utilities

Basically **every OS** has those.

Operating systems are made of a number of components, some more fundamental than others. Basically all of the above are present, in some form, in any operating system (excluding perhaps the smallest, most special-purpose systems). The kernel is the most fundamental and lowest layer of an operating system, while system libraries sit on top and use the services of the kernel. They also broker the services of the kernel to user-level programs and provide additional services (which do not need to be part of the kernel itself).

The remaining layers are mostly made of programs in the usual sense: other than being a part of the operating systems, there isn't much to distinguish them from user programs. The first category of such programs are **system daemons** or **system services**, which are typically long-running programs which react to requests or perform maintenance tasks.

The user interface is slightly more complicated, in the sense that it consists of multiple sub-components that align with other parts here. The bullet point here summarises those parts of the user interface that are more or less standard programs, like the command interpreter.

The Kernel 31

- **lowest level** of an operating system
- executes in **privileged mode**
- manages all the other software
 - including other OS components
- enforces **isolation and security**
- provides **low-level services** to programs

The kernel is the lowest and arguably the most important part of an operating system. Its main distinction is that it executes in a special processor mode (often known as **privileged**, **monitor** or **supervisor** mode). The main tasks of the kernel are management of basic hardware resources (processor, memory) and specifically providing those resources to other software running on the computer. This includes the rest of the operating system.

Another crucial task is enforcement of isolation and security. The hardware typically provides means to isolate individual programs from each other, but it is up to the software (OS kernel) to set up those hardware facilities correctly and effectively.

Finally, the kernel often provides the lowest level of various services to the upper layers. Those are provided mainly in the form of **system calls**, and mainly relate (directly or indirectly) to hardware access.

System Libraries 32

- form a layer above the OS kernel
- provide **higher-level** services
 - use kernel services behind the scenes
 - **easier to use** than the kernel interface
- typical example: **libc**
 - provides C functions like **printf**
 - also known as **msvcrt** on Windows

One rung above the kernel reside system libraries: among other things, they provide an interface between the kernel and higher levels of the system. The interface provided by the library to the application is also one level of abstraction above kernel services, which typically makes them easier to use.

Like all libraries, they are **linked** into other programs and effectively become their part: as such, library code executes with the same privileges as application code – for functionality that is not purely computational, system libraries need to communicate with other parts of the operating system: either the kernel, or other privileged components (system services, also known as daemons).

System Daemons 33

- programs that run in the **background**
- they either directly **provide services**
 - but daemons are different from libraries
 - we will learn more in later lectures
- or perform **maintenance** or periodic **tasks**
- or perform tasks **requested by the kernel**

Daemons are long-running system programs: they take care of tasks which need to be done continuously or periodically, but at the same time do not need to reside in the kernel. This includes things like delivery of internet mail, remote access to the system (e.g. a secure shell daemon), synchronisation of the system clock (network time protocol daemon), configuration and leasing of network addresses (dynamic host control protocol daemon) and so on, printer spooling, domain name services, parts of the network file system, hardware health monitoring, system-wide logging and others.

User Interface 34

- mediates user-computer **interaction**
- the main **shell** is typically part of the OS
 - command line on UNIX or DOS
 - graphical interfaces with a desktop and windows
 - but also buttons on your microwave oven
- also **building blocks** for application UI
 - buttons, tabs, text rendering, OpenGL...
 - provided by system libraries and/or daemons

In most systems, application programs cannot directly drive hardware – it is therefore up to the operating system to provide an interface between the user (who operates the hardware) and the application program. This includes user inputs (like keystrokes, mouse movement, touchpad or touchscreen events and the like) and relaying the outputs of the application to the user (printing text, drawing windows on the screen, audio output, etc.).

System Utilities

35

- small programs required for OS-related tasks
- e.g. system configuration
 - things like the registry editor on Windows
 - or simple text editors
- filesystem maintenance, daemon management, ...
 - programs like `ls/dir` or `newfs` or `fdisk`
- also bigger programs, like file managers

Not all 'short-running' (i.e. non-daemon) programs are application software. There is a number of utilities which aid with the management of the operating system itself, configuration of services and of the underlying hardware and so on. Those utilities typically use the same type of interface that application software does – whether it is a command-driven interface or a graphical one.

The distinction between bundled application software and system utilities is sometimes blurry: things like the `file explorer` in Windows is a fairly big and complicated program, but it also serves a rather central role in day-to-day use of the system. It can be imagined, however, that someone would take a program like the Explorer and port it to a different operating system. Perhaps the effort would be non-trivial, but the program would probably not appear out of place in another GUI-based OS.

There is, however, a number of small programs with a clear-cut purpose, which are much easier to classify, like the network configuration tool `ifconfig` or the disk partitioning tool `fdisk`. Likewise with tools like `fsck` (or `chkdisk` on windows), which are quite meaningless outside of the operating system that they came with.

Optional Components

36

- bundled `application` software
 - web browser, media player, ...
- (3rd-party) `software management`
- a `programming` environment
 - eg. a C compiler & linker
 - C header files &c.
- source code

It is often the case that an operating system comes bundled with programs which are not an integral part of the operating system, nor are they in any way involved in its normal operation. A typical example would be a small collection of games – a tradition that dates back to the original Berkeley UNIX, if not further into the past, and has been observed by almost all general-purpose operating systems since: the Solitaire and Minesweeper games that come bundled with Windows have an almost iconic status. Of course there is other software that falls into this category: arguably, MS Paint or Windows Media Player is not in any way essential to operate Windows, nor is a web browser. On UNIX, the software that has traditionally been bundled is of a slightly different nature and stems from the different target audience (and also from a different era): a C compiler, a linker and comparatively advanced source code editors are often found distributed with UNIX-like systems. In some cases, part of those tools was in fact essential, since the user would have had to compile a kernel tailored for their computer. This has, however, not been the case for a while, but most UNIX users expect to have a C compiler available anyway. Along with a C compiler would usually come header files for system libraries and other files needed to create and build your own programs. As I said, a different era.

Finally, you may get the source code of the operating system: strictly speaking, this is not a software component, but rather a different

(human-readable, instead of machine-executable) representation of the same operating system. It is quite useful if you want to learn the low-level details of how computers and operating systems work.

Part 1.2: Interfaces

Another way to look at an operating system is from the point of view of its surroundings – what kinds of interfaces there are between the operating system and other components of a computer?

Programming Interface

38

- kernel provides `system calls`
 - `ABI`: Application Binary Interface
 - defined in terms of `machine instructions`
- system libraries provide APIs
 - Application `Programming` Interface
 - symbolic / `high-level` interfaces
 - typically defined in terms of `C functions`
 - system calls also available as an `API`

The most obvious interface of an operating system (if you are a programmer, anyway) is its API: the Application Programming Interface – as the name suggests, it is the functionality that application programs can obtain from the operating system. Most of this API is provided by `system libraries` – bundles of subroutines in the form of machine code that application programs (and system daemons and system utilities) can call into. Those subroutines often further communicate with the kernel, using a system-specific low-level protocol: this protocol is known as the ABI (Application Binary Interface) of the kernel. Programmers are, for the most part, not exposed to the details of the ABI. The API is typically described in terms of functions in a higher-level programming language: most often C, sometimes C++ or Objective C, rarely some other language. Programmers use those functions just like they use functions they have themselves implemented, but instead of providing definitions, the compiler translates them into calls into the corresponding machine code subroutines stored in system libraries.

Message Passing

39

- APIs do not always come as C functions
- message-passing interfaces are possible
 - based on `inter-process communication`
 - possible even `across networks`
- form of API often provided by `system daemons`
 - may be also wrapped by C APIs

Nonetheless, C functions (or C++ or Objective C or other programming language functions) are not the `only` API that there is. Often, there are interfaces described in terms of inter-process communication, most often some form of message passing. Those are often interfaces that are provided by system daemons, e.g. `syslogd` (usually a UNIX domain socket) or the mail daemon (often a TCP socket).

Portability

40

- some OS tasks require close **HW cooperation**
 - **virtual memory** and CPU setup
 - platform-specific **device drivers**
- but many do not
 - **scheduling** algorithms
 - memory **allocation**
 - all sorts of management
- porting: changing a program to run in a **new environment**
 - for an OS, typically new hardware

It is desirable that operating systems can run on different hardware platforms: this reduces costs in a number of ways. The best (and cheapest) code is the code that you don't need to write – using the same operating system on different hardware platforms achieves exactly this (to a degree). On the other side, this saves resources of application developers, who can target a single operating system and reach a number of different hardware devices, and also training costs – users can be migrated from one hardware platform to another without extensive retraining for new software.

While it is basically impossible to write an operating system in an entirely hardware-independent way, many of its components do not need to care about the particulars of the hardware platform. This includes even some of the core kernel components (again, to a degree). Any given thread scheduler can be often used without changes (and sometimes without any sort of additional tuning) on a different hardware platform, same goes for memory allocators, filesystem code and so on. Of course, there are also pieces that are closely tied to particular hardware: the boot sequence (which includes things like setting up the CPU and the virtual memory subsystem), device drivers (which are tied to particular devices) and so on.

Finally, when we talk about portability, porting the operating system itself is not the only concern: it is often desirable to port application programs to run on a different software stack. In general, portability is the ability of a program to be (easily) adapted to a new environment.

Hardware Platform

41

- CPU **instruction set** (ISA)
- busses, IO controllers
 - PCI, USB, Ethernet, ...
- **firmware**, power management

Examples

- x86 (ISA) – PC (platform)
- ARM – Snapdragon, i.MX 6, ...
- m68k – Amiga, Atari, ...

Platform & Architecture Portability

42

- an OS typically supports many **platforms**
 - Android on many different ARM SoC's
- quite often also different **CPU ISAs**
 - long tradition in UNIX-style systems
 - NetBSD runs on 15 different ISAs
 - many of them comprise 6+ different platforms
- special-purpose systems are usually less portable

Code Re-Use

43

- it makes a lot of sense to re-use code
- **majority** of OS code is **HW-independent**
- this was not always the case
 - pioneered by UNIX, which was written in C
 - typical OS of the time was in machine language
 - porting was basically “writing again”

Application Portability

44

- applications **care** more **about the OS** than about HW
 - apps are written in **high-level languages**
 - and use system libraries extensively
- it is enough to port the OS to new/different HW
 - most applications can be simply **recompiled**
- still a major hurdle (cf. Itanium)

Application Portability (2)

45

- same application can often run on **many OSes**
- especially within the POSIX family
- but same app can run on Windows, macOS, UNIX, ...
 - Java, Qt (C++)
 - web applications (HTML, JavaScript)
- many systems provide the same set of services
 - differences are mostly in programming interfaces
 - high-level libraries and languages can hide those

Abstraction

46

- **instruction sets** abstract over CPU details
- **compilers** abstract over **instruction sets**
- **operating systems** abstract over **hardware**
- portable runtimes abstract over operating systems
- applications sit on top of the abstractions

Abstraction Costs

47

- more complexity
- less efficiency
- leaky abstractions

Abstraction Benefits

- easier to write and port software
- fewer constraints on HW evolution

Abstraction Trade-Offs

48

- powerful hardware allows more abstraction
- embedded or real-time systems not so much
 - the OS is smaller & less portable
 - same for applications
 - more efficient use of resources

Part 1.3: Classification

Our last attempt at understanding what an operating system is will revolve around different types of operating systems and their differences.

General-Purpose Operating Systems 50

- suitable for use in **most** situations
- **flexible** but **complex** and big
- run on both **servers** and **clients**
- cut down versions run on **smartphones**
- support variety of hardware

The most important and interesting category is 'general-purpose operating systems'. This is the one that we will mostly talk about in this course. The systems in this category are usually quite flexible (so they can cover everything that people usually use computers for) but, for the same reason, also quite complex. Often the same operating system will be able to run on both so-called 'server' computers (those mainly sitting in data centres providing services to other computers remotely) and 'client' computers – those that interact with users directly.

Likewise, the same operating system can, perhaps in a slimmed down version, run on a smartphone, or a similar size- and power-constrained device. All current major smartphone operating systems are of this type. Historically, there were a few more specialised phone operating systems, mainly because at that time, phone hardware was considerably more constrained than it is today. Nonetheless, an OS like Symbian, for instance, could conceivably be used on personal computers assuming its hardware support was extended.

Operating Systems: Examples 51

- Microsoft Windows
- Apple macOS & iOS
- Google Android
- Linux
- FreeBSD, OpenBSD
- MINIX
- many, many others

There is a whole bunch of operating systems, even of general-purpose operating systems. While running the OS itself is not the primary reason for getting a computer (application software is), it does form an important part of user experience. Of course, it also interfaces with computer hardware and with application programs, and not all systems run on all computers and not all applications run on all operating systems.

Special-Purpose Operating Systems 52

- **embedded** devices
 - limited budget
 - **small**, slow, power-constrained
 - hard or impossible to update
- **real-time** systems
 - must **react** to real-world events
 - often **safety-critical**
 - robots, autonomous cars, space probes, ...

We have mentioned earlier, that general-purpose operating systems are usually large and complex. The smallest complete operating systems (if they are not merely educational toys) start around 100 thou-

sand lines of code, but millions of lines is more typical. It is not unheard of that an operating system contains more than 10 million lines of code. These amounts clearly represent thousands of man-years of work – writing your own operating system, solo, is not very realistic. That said, special-purpose systems are often much smaller. They usually only support far fewer hardware devices and they provide simpler and less varied services to the 'application' software.

Size and Complexity 53

- operating systems are usually large and complex
- typically **100K and more** lines of code
- **10+ million** is quite possible
- many thousand man-years of work
- special-purpose systems are much smaller

Let's recall that the kernel runs in privileged CPU mode. Any software running in this mode is pretty much all-powerful and can easily circumvent any access restrictions or security protections. It is a well-known fact that the more code you have, the more bugs there are. Since bugs in the kernel can have far-reaching and catastrophic consequences, it is imperative that there are as few as possible. Even more importantly, device drivers often need hardware access and the easiest (and sometimes only) way to achieve that is by executing in kernel (privileged) mode.

As you may also know, device drivers are often of rather questionable quality: hardware vendors often consider those an after-thought and don't pay too much attention to their software teams. If those drivers then execute in kernel mode, this is a serious problem. Different OS vendors employ different strategies to mitigate this issue.

Accordingly, we would like to make kernels small and banish as many drivers from the kernel as we could. It is, however, not an easy (or even obviously right) thing to do. There are two main design schools when it comes to kernel 'size':

Kernel Revisited 54

- bugs in the kernel are very bad
 - system crashes, data loss
 - **critical** security problems
- bigger kernel means more bugs
- third-party drivers inside the kernel?

Monolithic Kernels 55

- lot of code in the kernel
- less abstraction, less isolation
- **faster** and more efficient

Microkernels

- move as much as possible out of kernel
- more abstraction, **more isolation**
- slower and less efficient

The monolithic kernel is an older and in some sense simpler design. A lot of code ends up in the kernel, which is not really a problem until bugs happen. There is less abstraction involved in this design, fewer interfaces and in general, fewer moving parts for the same amount of functionality. Those traits then translate to faster execution and more efficient resource use. Such kernels are called monolithic because everything that a traditional kernel does is performed by a single (monolithic) piece of software.

The opposite of monolithic kernels are microkernels. The kernel proper in such a system is the smallest possible subset of code that must run in privileged mode. Everything that can be banished into user mode (of the processor) is. This design provides a lot more isolation and requires more abstraction. The interfaces within different parts of the low-level OS services are more complicated. However, subsystems are well isolated from each other and faults do not propagate nearly as easily. However, operating systems which use this kernel type run more slowly and use resources less efficiently.

Paradox?

56

- real-time & embedded systems often use microkernels
- isolation is good for reliability
- efficiency also depends on the **workload**
 - throughput vs latency
- real-time does not necessarily mean fast

Finally, there is a bit of a paradox around microkernels: it is a fact

that they are often used in embedded (real-time) systems – some of the most performance-critical software stacks around. However, one thing is more important than performance when it comes to embedded software: reliability. And reliability is where microkernels shine. Additionally, even in hard real-time systems, where we often consider performance to be paramount, raw speed is a bit of a red herring – what is important is latency, and even more important is an upper bound on this latency. Providing one is, however, much easier in a microkernel system, where the code base is small and much easier to reason about.

Review Questions

57

1. What are the roles of an operating system?
2. What are the basic components of an OS?
3. What is an operating system kernel?
4. What is an Application Programming Interface?

Part 2: System Libraries and APIs

In this section, we will study the programming interfaces of operating systems, first in some generality, without a specific system in mind. We will then go on to deal specifically with the C-language interface of POSIX systems.

Programming Interfaces

59

- kernel **system call** interface
- → **system libraries** / APIs ←
- inter-process protocols
- command-line utilities (scripting)

In most operating systems, the lowest-level interface accessible to application programs is the **system call** interface. It is, typically, specified in terms of a machine-language-level protocol (that is, an ABI), but usually also provided as a C API. This is the case for POSIX-mandated system calls, but also on e.g. Windows NT systems.

Lecture Overview

60

1. The C Programming Language
2. System Libraries
 - what is a library?
 - header files & libraries
3. Compiler & Linker
 - object files, executables
4. File-based APIs

In this lecture, we will start by reviewing (or perhaps introducing) the C programming language. Then we will move on to the subject of libraries in general and system libraries in particular. We will look at how libraries enter the program compilation process and what other ingredients there are. Finally, we will have a closer look at a specific set of file-based programming interfaces.

Sidenote: UNIX and POSIX

61

- we will mostly use those terms interchangeably
- it is a **family** of operating systems
 - started in late 60s / early 70s
- POSIX is a **specification**
 - a document describing what the OS should provide
 - including programming interfaces

We will **assume POSIX** unless noted otherwise

Before we begin, it should be noted that throughout this course, we will use POSIX and UNIX systems as examples. If a specific function or interface is mentioned without further qualification, it is assumed to be specified by POSIX and implemented by UNIX-like systems.

Part 2.1: The C Programming Language

The C programming language is one of the most commonly used languages in operating system implementations. It is also the subject of PB071, and at this point, you should be already familiar with its basic syntax. Likewise, you are expected to understand the concept of a **function** and other basic building blocks of programs. Even if you don't know the specific C syntax, the idea is very similar to any other programming language you might know.

Programming Languages

63

- there are many different languages
 - C, C++, Java, C#, ...
 - Python, Perl, Ruby, ...
 - ML, Haskell, Agda, ...
- but **C** has a **special place** in most OSes

Different programming languages have different use-cases in mind, and exist at different levels of abstraction. Most languages other than C that you will meet, both at the university and in practice, are so-called high-level languages. There are quite a few language families, and there is a number of higher-level languages derived from C, like

C++, Java or C#.

For the purposes of this course, we will mostly deal with plain C, and with POSIX (Bourne-style) **shell**, which can also be thought of as a programming language.

C: The Least Common Denominator 64

- except for assembly, C is the “bare minimum”
- you can almost think of C as **portable assembly**
- it is very easy to call C functions
- and to use C data structures

You can use C libraries in almost every language

You could think of C as a ‘portable assembler’, with a few minor bells and whistles in form of the standard library. Apart from this library of basic and widely useful subroutines, C provides: abstraction from machine opcodes (with human-friendly infix operator syntax), structured control flow, and automatic local variables as its main advantages over assembly.

In particular the abstraction over the target processor and its instruction set proved to be instrumental in early operating systems, and helped establish the idea that an operating system is an entity separate from the hardware.

On top of that, C is also popular as a systems programming language because almost any program, regardless of what language it is written in, can quite easily call C functions and use C data structures.

The Language of Operating Systems 65

- many (most) kernels are **written in C**
- this usually extends to system libraries
- and sometimes to almost the entire OS
- non-C operating systems provide **C APIs**

Consequently, C has essentially become a ‘language of operating systems’: most kernels and even the bulk of most operating systems is written in C. Each operating system (apart from perhaps a few exceptions) provides a C standard library in some form and can execute programs written in C (and more importantly, provide them with essential services).

Part 2.2: System Libraries

We have already touched the topic of system libraries last week, in the ‘anatomy’ section. It is now time to look at them in more detail: what they contain, how are they stored in the file system, how are they combined with programs. We will also briefly talk about system call wrappers (which mediate low-level access to kernel services – we will discuss this topic in more detail in the next lecture). Finally, we will look at a few examples of system libraries which appear in popular operating systems.

(System) Libraries 67

- mainly **C functions** and **data types**
- interfaces defined in **header files**
- definitions provided in **libraries**
 - static libraries (archives): **libc.a**
 - shared (dynamic) libraries: **libc.so**
- on Windows: **msvcrt.lib** and **msvcrt.dll**
- there are (many) more besides **libc / msvcrt**

In this course, when we talk about libraries, we will mean C libraries

specifically. Not Python or Haskell modules, which are quite different. That said, a typical C library has basically two parts, one is header files which provide a description of the interface (the API) and the compiled library code (an archive or a shared library).

The interface (as described in header files) consists of functions (for which, the types of arguments and the type of return value are given in a header file) and of data structures. The bodies of the functions (their implementation) is what makes up the compiled library code. To illustrate:

Declaration: **what** but not **how** 68

```
int sum( int a, int b );
```

Definition: **how** is the operation done?

```
int sum( int a, int b )
{
    return a + b;
}
```

The first example on this slide is a declaration: it tells us the name of a function, its inputs and its output. The second example is called a **definition** (or sometimes a **body**) of the function and contains the operations to be performed when the function is called.

Library Files 69

- **/usr/lib** on most Unices
 - may be mixed with **application libraries**
 - especially on Linux-derived systems
 - also **/usr/local/lib** for user/app libraries
- on Windows: **C:\Windows\System32**
 - user libraries often **bundled** with programs

The machine code that makes up the library (i.e. the code that was generated from function definitions) resides in files. Those files are what we usually call ‘libraries’ and they usually live in a specific filesystem location. On most UNIX system, those locations are **/usr/lib** and possibly **/lib** for system libraries and **/usr/local/lib** for user or application libraries. On certain systems (especially Linux-based), user libraries are mixed with system libraries and they are all stored in **/usr/lib**.

On Windows, the situation is similar in that both system and application libraries are installed in a common location. Additionally, on Windows (and on macOS), shared libraries are often installed alongside the application.

Static Libraries 70

- stored in **libfile.a**, or **file.lib** (Windows)
- only needed for **compiling** (linking) programs
- the code is **copied** into the executable
- the resulting executable is also called **static**
 - and is easier to work with for the OS
 - but also more wasteful

Static libraries are only used when building executables and are not required for normal operation of the system. Therefore, many operating systems do not install them by default – they have to be installed separately as part of the developer kit. When a static library is linked into a program, this basically entails copying the machine code from the library into the final executable.

In this scenario, after linking is performed, the library is no longer

needed since the executable contains all the code required for its execution. For system libraries, this means that the code that comes from the library is present on the system in many copies, once in each program that uses the library. This is somewhat alleviated by linkers only copying the parts of the library that are actually needed by the program, but there is still substantial duplication.

The duplication arising this way does not only affect the file system, but also memory (RAM) when those programs are loaded – multiple copies of the same function will be loaded into memory when such programs are executed.

Shared (Dynamic) Libraries 71

- required for **running** programs
- linking is done at **execution** time
- less code duplication
- can be **upgraded** separately
- but: dependency problems

The other approach to libraries is **dynamic**, or **shared** libraries. In this case, the library is required to actually run the program: the linker does not copy the machine code from the library into the executable. Instead, it only notes that the library must be loaded alongside with the program when the latter is executed.

This reduces code duplication, both on disk and in memory. It also means that the library can be updated separately from the application. This often makes updates easier, especially in case a library is used by many programs and is, for example, found to contain a security problem. In a static library, this would mean that each program that uses the library needs to be updated. A shared library can be replaced and the fixed code will be loaded alongside programs as usual.

The downside is that it is difficult to maintain binary compatibility – to ensure that programs that were built against one version of the library also work with a later version. When this is violated, as often happens, people run into dependency problems (also known as DLL hell on Windows).

Header Files 72

- on UNIX: `/usr/include`
- contains **prototypes** of C functions
- and definitions of C data structures
- required to **compile** C and C++ programs

Like static libraries, header files are only required when building programs, but not when using them. Header files are fragments of C source code, and on UNIX systems are traditionally stored in `/usr/include`. User-installed header files (i.e. not those provided by system libraries) live under `/usr/local/include` (though again, on Linux-based systems user and system headers are often intermixed in `/usr/include`).

Header Example 1 (from `unistd.h`) 73

```
int      execv(char *, char **);
pid_t    fork(void);
int      pipe(int *);
ssize_t  read(int, void *, size_t);
```

(and many more prototypes)

This is an excerpt from an actual system header file, and declares a few of the functions that comprise the POSIX C API.

Header Example 2 (from `sys/time.h`) 74

```
struct timeval
{
    time_t    tv_sec;
    long      tv_usec;
};

/* ... */

int gettimeofday(timeval *, timezone *);
int settimeofday(timeval *, timezone *);
```

This is another excerpt from an actual header – this time the snippet contains a definition of a **data structure**. The layout (order of fields and their types, along with hidden **padding**) of such structures is quite important, since that becomes part of the ABI. In other words, the definition above describes not just the high-level interface but also how bytes are laid out in memory.

The POSIX C Library 75

- **libc** – the C runtime library
- contains ISO C functions
 - `printf`, `fopen`, `fread`
- and a number of POSIX functions
 - `open`, `read`, `gethostbyname`, ...
 - C wrappers for system calls

As we have already mentioned previously, it is a tradition of UNIX systems that **libc** combines the basic C library and the basic POSIX library. For the following, a particular subset of the POSIX library is going to be rather important, namely the **system call wrappers**. Those are C functions whose only purpose is to invoke their matching **system calls**.

System Calls: Numbers 76

- system calls are performed at **machine level**
- which syscall to perform is decided by a **number**
 - e.g. `SYS_write` is 4 on OpenBSD
 - numbers defined by `sys/syscall.h`
 - different for each OS

At the level of the OS kernel (cue next week), system calls are represented by **numbers** (which are often given symbolic names like `SYS_write`, but are nonetheless just small integers and not memory addresses like with ordinary C functions). The numbers are specific to any given kernel. And of course, the **libc** must use the same numbering as the kernel.

System Calls: the `syscall` function

77

- there is a C function called `syscall`
 - prototype: `int syscall(int number, ...)`
- this implements the **low-level** syscall sequence
- it takes a **syscall number** and syscall parameters
 - this is a bit like `printf`
 - first parameter decides what are the other parameters
- (more about how `syscall()` works next week)

Typically, all system calls work essentially the same: the library takes the (syscall) number and some additional data (parameters), stores them at the pre-arranged location (registers, memory) and jumps into the kernel. Since this sequence is uniform across system calls, it is possible to have a single C function which can perform any system call, given its number.

This function actually exists and is called `syscall`. It's entirely possible to perform all your syscalls using this one C function, and never call the more convenient single-purpose wrappers (see also below).

System Calls: Wrappers

78

- using `syscall()` directly is inconvenient
- `libc` has a function for each system call
 - `SYS_write` → `int write(int, char *, size_t)`
 - `SYS_open` → `int open(char *, int)`
 - and so on and so forth
- those wrappers may use `syscall()` internally

To make programming a fair bit more convenient, instead of saying

```
syscall( SYS_write, fd, buffer, size );
```

we can use a function called `write`, like this:

```
write( fd, buffer, size );
```

Besides being shorter to type, it is also safer: the compiler can check that we passed the right number and types of arguments. The function might internally use the equivalent `syscall()` invocation – though in practice, we prefer to sacrifice this particular bit of abstraction to save a few instructions on the comparatively hot (read oft-invoked) syscall path.

Portability

79

- libraries provide an **abstraction layer** over OS internals
- they are responsible for **application portability**
 - along with standardised filesystem locations
 - and user-space utilities to some degree
- higher-level languages rely on system libraries

An important function of libraries is to provide a uniform API to the upper layers of the system. The designers of an operating system may decide to substantially depart from the traditional system call protocol, or even from the traditional set of system calls. However, even if the kernel looks quite non-POSIX-y, it is often still possible to provide a set of C functions that behave as POSIX specifies. This has been done more than once, most often on top of microkernels, e.g. Microsoft NT (Windows NT, XP and later) or on Mach (macOS, HURD). All those systems are capable of supporting POSIX programs without being built around a UNIX-like monolithic kernel.

Of course, the API alone is not sufficient to make POSIX programs work correctly: there are certain expectations about the filesystem

(both semantics of the file system itself, but also which files exist and what they contain) and other aspects of the system.

NeXTSTEP and Objective C

80

- the NeXT OS was built around **Objective C**
- system libraries had ObjC APIs
- in API terms, ObjC is very **different from C**
 - also very different from C++
 - traditional **OO** features (like Smalltalk)
- this has been partly inherited into **macOS**
 - evolving into Swift

Not all operating systems provide (exclusively) C APIs. Historically, one of the earlier departures was the NeXT operating system, which used Objective C extensively. While the procedural part of the language is simply C, the object-oriented part is based on Smalltalk, with pervasive late binding and dynamic types.

System Libraries: UNIX

81

- the math library `libm`
 - implements math functions like `sin` and `exp`
- thread library `libpthread`
- terminal access: `libcurses`
- cryptography: `libcrypto` (OpenSSL)
- the C++ standard library `libstdc++` or `libc++`

While `libc` is quite central, there are many other libraries that are part of a UNIX system. You would find most of the above examples on most UNIX systems in some form.

System Libraries: Windows

82

- `msvcrt.dll` – the ISO C functions
- `kernel32.dll` – basic OS APIs
- `gdi32.dll` – Graphics Device Interface
- `user32.dll` – standard GUI elements

System libraries look quite differently on Windows: there is no `libc`: instead, the C standard library has its own DLL (the `msvcrt`, from Microsoft Visual C RunTime) while operating system services (the low-level kind) live in `kernel32.dll`. The other two libraries allow applications to provide a graphical user interface. The libraries mentioned here all provide C APIs, though there are also C++ and C# interfaces (which are partly wrappers around the above libraries, but not exclusively).

Documentation

83

- manual pages on UNIX
 - try e.g. `man 2 write` on aisa.fi.muni.cz
 - section 2: system calls
 - section 3: library functions (`man 3 printf`)
- MSDN for Windows
 - [<https://msdn.microsoft.com>](https://msdn.microsoft.com)
- you can learn **a lot** from those sources

Most OS vendors provide extensive documentation of their programmer's interfaces. On UNIX, this is typically part of the OS installation itself (manual pages, command `man`), while on Windows, this is a separate resource (these days accessible online, previously distributed in

Part 2.3: Compiler & Linker

While compiling (and linking) programs is not core functionality of an operating system, it is quite useful to understand how these components work. Moreover, in earlier systems, a C compiler was considered a rather essential component and this tradition continues in many modern UNIX systems to this day. We will discuss different artefacts of compilation – object files, libraries and executables, as well as the process of linking object code and libraries to produce executables. We will also highlight the differences between static and shared (dynamic) libraries.

C Compiler

85

- many POSIX systems ship with a **C compiler**
- the compiler takes a **C source file** as input
 - a text file with a **.c** suffix
- and produces an **object file** as its output
 - binary file with machine code in it
 - but cannot be directly executed

Compilers transform human-readable programs into machine-executable programs. Of course, both those forms of the program need to be stored in memory: the first is usually in the form of **plain text** (usually encoded as UTF-8, or in older systems as ASCII). In this form, bytes stored in the file encode human-readable letters.

On the output side, the file is **binary** (which is really just a catch-all term for files that are not plain text), and stores machine-friendly **instructions** – primitive operations that the CPU can execute. Only the compiler output cannot be directly executed yet, even though most of the instructions are in their final form.

The missing piece are addresses: numbers which describe memory locations within the program itself (they may point at instructions or at data embedded in the program). At this stage, though, neither code nor data has been assigned to particular addresses, and hence the program cannot be executed (it will need to be **linked** first, more on that later).

Object Files

86

- contain native **machine** (executable) code
- along with static data
 - e.g. string literals used in the program
- possibly split into a number of **sections**
 - **.text**, **.rodata**, **.data** and so on
- and metadata
 - list of **symbols** (function names) and their addresses

The purpose of object files is to store this semi-finished machine code, along with any static data (like string literals or numeric constants) that appear in the program. All this is sorted into **sections** – usually one section for machine code (also called text and called **.text** in the object file), another for read-only data (e.g. string literals), called **.rodata**, another for mutable but statically-initialized variables – **.data**. Bundled with all this is **metadata**, which describes the content of the file (again in a machine-readable form).

One example of such metadata is a **symbol table**, which gives file-relative addresses of high-level functions that have been compiled into the object file. That is, the compiler will take a definition of a function that we wrote in C and emit machine code for this function. The **.text** section of an object file will consist of a number of such functions, one after another: the symbol table then tells us where each of the

functions begins.

Object File Formats

87

- **a.out** – earliest UNIX object format
- COFF – Common Object File Format
 - adds support for sections over **a.out**
- PE – Portable Executable (MS **Windows**)
- Mach-O – Mach Microkernel Executable (**macOS**)
- **ELF** – Executable and Linkable Format (all modern Unices)

There is a number of different physical layouts of object files, and each of those also carries slightly different semantics. By far the most common format used in POSIX systems is **ELF**. The other common formats in contemporary use are **PE** (used by MS operating systems) and **Mach-O** (used by Apple operating systems).

Archives (Static Libraries)

88

- static libraries on UNIX are called **archives**
- this is why they get the **.a** suffix
- they are like a **zip** file full of **object files**
- plus a table of symbols (function names)

An archive is the simplest way to bundle multiple object files. As the name implies, it is essentially just a collection of object files stored as a single file. Each object file retains its identity and its content does not change in any way when it is bundled into an archive.

The only difference from a typical data archive (a **tar** or a **zip** archive, say) is that besides the object files themselves, the archive contains an additional metadata section – a symbol table, or rather a symbol index. If someone (typically the linker) needs to find the definition of a particular function (symbol), it can first consult this archive-wide index to find which object file provides that symbol. This makes linking more efficient, since the linker does not need to sequentially scan each object file in the archive to find the definition.

Linker

89

- object files are **incomplete**
- they can refer to **symbols** that they do not define
 - the definitions can be in libraries
 - or in other object files
- a **linker** puts multiple object files together
 - to produce a **single executable**
 - or maybe a shared library

As pointed out earlier, it is the job of a **linker** to combine object files (and libraries) into executables. The process is fairly involved, so we will describe it across the next few slides. The **input** to the linker is a bunch of **object files** and the output is a single **executable** or sometimes a single **shared library**.

Even though archives are handled specially by the linker: object files which are given to the linker directly will always become part of the final executable. Object files provided in archives are only used if they provide symbols which are required to complete the executable.

Symbols vs Addresses

90

- we use symbolic **names** to call functions &c.
- but the **call** machine instruction needs an **address**
- the executable will eventually live in memory
- data and instructions need to be given **addresses**
- what a linker does is **assign** those addresses

The main entities that come up during linking are **symbols** and **addresses**. In a program, the machine code and the data is loaded in memory, and as we know, each memory location has an **address**. The program in its compiled form can use addresses to refer to parts of itself. For instance, to call a subroutine, we provide its starting address to a special **call** instruction, which tells the CPU to start executing code from that address.

However, when humans write programs, they do not assign addresses to pieces of data, to functions or to individual instructions. Instead, if the program needs to refer to a part of itself, we give those parts names: those names are known as **symbols**. It is the shared responsibility of the compiler and the linker to assign addresses to the individual symbols, in such a way that the objects stored in memory do not conflict (overlap). If you think about it, it would be very difficult to do by hand: we usually don't know how long the machine code will be for any given function, and we would need to guess and then add gaps in case we need to add more code to a function, and so on. And we would need to remember which code lives at which address and so on. It is all very uncomfortable, and even assembly programmers usually avoid assigning addresses by hand. In fact, one of the primary roles of an assembler is to translate from symbolic to numeric addresses. But I digress.

Resolving Symbols

91

- the linker processes one object file at a time
- it maintains a **symbol table**
 - mapping symbols (names) to addresses
 - dynamically updated as more objects are processed
- relocations are typically processed all at once at the end
- **resolving symbols** = finding their addresses

The linker works by maintaining an 'incomplete executable' and makes progress by merging each of the input object files into this work-in-progress file. The strategy for assigning final addresses is simple enough: there's a single output **.text** section, a single output **.data** section and so on. When an input file is processed, its own **.text** section is simply appended to the **.text** produced so far. The same process is repeated for every section.

The symbol tables of the input object files are likewise merged one by one, and the addresses adjusted as symbols are added. In addition to symbol **definitions**, object files contain symbol **uses** – those are known as relocations, and are stored in a relocation table. Relocations contain the **address of the instruction** that needs to be patched and the **symbol** the address of which is to be patched in. Like the sections themselves and the symbol table, the relocation table is built up.

The relocations are also processed by the linker: usually, this means writing the final address of a particular symbol into an as-of-yet incomplete instruction or into a variable in the data section. This is usually done once the output symbol table is complete.

The relocation and symbol tables are often discarded at the end (but may be retained in the output file in some cases – the symbol table more often than the relocation table).

Executable

92

- finished **image** of a program to be executed
- usually in the same format as **object files**
- but already complete, with symbols resolved
 - **but:** may use **shared libraries**
 - in that case, **some** symbols remain unresolved

The output of the linker is, in the usual case, an **executable**. This is a file that is based on the same format as object files of the given operating system, but is **complete** in some sense. In static executables (those which don't use shared libraries), all references and relocations are already resolved and the program can be loaded into memory and directly executed by the CPU, without further adjustments.

It is also worth noting that the addresses that the executable uses when referring to parts of itself are **virtual addresses** (this is also the case with shared libraries below). We will talk more about those in a later lecture, but right now we can at least say that this means that different programs on the same operating system can use overlapping addresses for their instructions and data. This is not a problem, because virtual addresses are private to each process, and hence each copy of each executing program.

Shared Libraries

93

- each shared library only needs to be in memory once
- shared libraries use **symbolic names** (like object files)
- there is a "mini linker" in the OS to resolve those names
 - usually known as a **runtime linker**
 - resolving = finding the addresses
- shared libraries can use other shared libraries
 - they can form a **DAG** (Directed Acyclic Graph)

The downside of static libraries is that they need to be loaded separately (often in slightly different versions) along with each program that uses them: in fact, since the linker embedded them into the program, they are quite inseparable from it.

As we have already mentioned, this is not very efficient. Instead, we can store the library code in separate executable-like files that get loaded into the address space of programs that need it. Of course, relocations in the main program that refer to symbols from shared libraries (and vice versa), and obviously also relocations in shared libraries that refer to other shared libraries, those need to be resolved. This is usually done either when the program is loaded into memory, or lazily, right before the relocation is first used.

In either case, there needs to be a program which will resolve those relocations: this is the **runtime linker** – it is superficially similar to the normal, compile-time linker, but in reality is quite different.

Addresses Revisited

94

- when you run a program, it is **loaded into memory**
- parts of the program refer to other parts of the program
 - this means they need to know **where** it will be loaded
 - this is a responsibility of the **linker**
- shared libraries use **position-independent code**
 - works regardless of the base address it is loaded at
 - we won't go into detail on how this is achieved

We mentioned that executables and libraries use virtual addresses to refer to their own parts. However, this does not help shared libraries as much as it helps with executables. The letdown is that we want to load

the same library along with multiple programs: but if the addresses used by the library are fixed, this means that the library needs to be loaded at the same start address into each program that uses that library. This quickly becomes impractical as we add more libraries into the system – no two libraries would be allowed to overlap and no none of them would be allowed to overlap with any of the executables.

In practice, what we instead do is that we compile the libraries in such a way that they don't use absolute addresses to refer to parts of themselves. This often adds a little execution overhead, but makes it possible to load the library at any address range that is available in the current process. This makes the job of the runtime linker much easier.

Compiler, Linker &c. 95

- the C compiler is usually called `cc`
- the linker is known as `ld`
- the archive (static library) manager is `ar`
- the `runtime` linker is often known as `ld.so`

On many UNIXes, the compiler and the linker are available as part of the system itself. The command names are standardized.

Part 2.4: File-Based APIs

On POSIX systems, the API for using the filesystem is a very important one, because it in fact provides access to a number of additional resources, which appear as 'abstract' (special) files in the system.

Everything is a File 97

- part of the UNIX `design philosophy`
- `directories` are files
- `devices` are files
- `pipes` are files
- network connections are (almost) files

File is an abstraction: it is an object from which we can read bytes and into which we can write bytes (not all files will let us do both). In regular files, we can read and write at any offset, and if we write something we can later read that same thing (unless it was rewritten in the meantime).

Directories are somewhat like this: we can read bytes from them to find out what files are present in that directory and how to find them in the file system. We can create new entries by writing into the directory. Incidentally, this is not how things are usually done, but it's not hard to imagine it could be.

Quite a few `devices` (peripherals) behave this way: all kinds of hard drives (just a big bunch of bytes), printers (write some bytes to have them printed), scanners (write bytes to send commands, read bytes with the image data), audio devices (read bytes from microphones, write bytes into speakers), and so on.

Pipes are like that too: one program writes bytes, and another reads them. And network connections are more or less just pipes that work across the network.

Why is Everything a File 98

- `re-use` the comprehensive `file system API`
- re-use existing file-based command-line tools
- bugs are bad → `simplicity` is good
- want to print? `cat file.txt > /dev/ulpt0`
 - (reality is a little more complex)

Since we already have an API to work with `abstract files` (because we need to work with real files anyway), it becomes reasonable to ask why not use this existing API to work with other objects that look like files. It makes sense not just at the level of C functions, but at the level of command-line programs too. In general, re-using existing mechanisms makes things more flexible, and often also simpler. Of course, there are caveats (devices often need to support operations that don't map well to reading or writing bytes, sockets are also somewhat problematic).

What is a Filesystem? 99

- a set of `files` and `directories`
- usually lives on a single block device
 - but may also be virtual
- `directories` and `files` form a `tree`
 - `directories` are internal nodes
 - `files` are leaf nodes

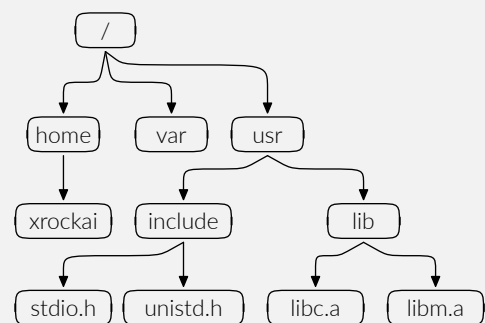
While we have a decent idea of what a `file` is, what about a `file system`? Well, a file system is a collection of files and directories, typically stored on a single block device. The directories and files form a tree (at least until symlinks come into play, at which point things start going south). Regular files are always leaf nodes in this tree.

File Paths 100

- filesystems use `paths` to point at files
- a string with `/` as a directory delimiter
 - the delimiter is `\` on Windows
- a leading `/` indicates the `filesystem root`
- e.g. `/usr/include`

Paths are how we refer to files and directories within the tree. The top-level (`root`) directory is named `/`. Each directory `entry` carries a name (and a link to the actual file or directory it represents) – this name can be used in the path to refer to the given entity. So with a path like `/usr/include`, we start at the `root directory` (the initial slash), then in that directory, we look for an entity called `usr` and when we find it, we check that it is a directory again. If that is so, we then look at its direct descendants again and look for an entity labelled `include`.

The File Hierarchy 101



That's an example of a file system tree. You can practice looking up various paths in the tree, using the algorithm described above.

The Role of Files and Filesystems

102

- **very** central in **Plan9**
- central in most UNIX systems
 - cf. Linux pseudo-file systems
 - `/proc` provides info about all processes
 - `/sys` gives info about the kernel and devices
- somewhat **reduced** in Windows
- quite **suppressed** in Android (and more on iOS)

Different operating systems put different emphasis on the file system. We will take the way POSIX positions the file system as the baseline – in this case, the file system is quite central, in addition to regular files and directories, all sorts of special files appear in the file system and provide access to various OS facilities. However, there are also many services and APIs that are not based on the file system, including e.g. process management, memory management and so on. In many UNIX-like systems, the reliance on FS-based APIs is notched up a bit: e.g. process management is done via a virtual `/proc` filesystem (many different systems), or device discovery and configuration via `/sys` (Linux). Another level above that is Plan9, where essentially everything that can be made into a file system is made into one. Another experimental system, GNU/Hurd, has a similar ambition.

If we go the other way from POSIX, we have the native Windows APIs, which emphasise the file system much less than would be typical in POSIX. Most objects have dedicated APIs, even if they are rather file-like. However, the file system is still prominently present both in the APIs and in the user interface. Both are further suppressed by modern ‘scaled-down’ operating systems like Android and iOS (even if both are POSIX-compatible under the hood, ‘normal’ applications are not allowed to access the POSIX API, or the file system, and it is usually also hidden from users).

The Filesystem API

103

- you **open** a file (using the `open()` syscall)
- you can `read()` and `write()` data
- you `close()` the file when you are done
- you can `rename()` and `unlink()` files
- you can use `mkdir()` to create directories

So how does the file system API look on POSIX systems? To work with a file, you usually need to **open** it first: you supply a **path** and some flags to tell the OS what you intend to do with the file. More on that in a short while. When you have a file open, you can **read** data from it and **write** data into it. When you are done, you use `close` to free up the associated resources. To work with directories, you usually don’t need to **open** them (though you can). You can rename files (this is a directory operation) using `rename`, remove them from the file system hierarchy using `unlink` (this erases the corresponding directory entry), and you can create new directories using `mkdir`.

File Descriptors

104

- the kernel keeps a table of open files
- the **file descriptor** is an index into this table
- you do everything using file descriptors
- non-Unix systems have similar concepts
 - descriptors are called **handles** on Windows

Remember **open**? When we want to work with a file, we need a way to identify that file, and paths are not super convenient in this respect:

someone could rename the file we were working with, and suddenly it is gone, or worse, the file could be replaced by a different file or even a directory. Additionally, looking up a file by its path is a comparatively expensive operation: the OS has to read every directory mentioned in the **path** and run a lookup on it. While this information is often cached in RAM, it still takes valuable time.

When we open a file, we get back a **file descriptor** – this is a small integer, and using this descriptor as an index into a table, the kernel can look up all the metadata it needs (to carry out reads and writes) in constant time. The descriptor is also associated with the file directly, so if the file is moved around or even unlinked from the directory tree, the descriptor still points to the same file.

Most non-POSIX file system APIs have a similar notion (sometimes `open` does not return a number but a different data type, e.g. a pointer, and sometimes this value is called a **handle** instead of a descriptor... but the concept is more or less the same).

Regular files

105

- these contain **sequential data** (bytes)
- may have inner structure but the OS does not care
- there is **metadata** attached to files
 - like when were they last modified
 - who can and who cannot access the file
- you `read()` and `write()` files

A regular file is what it appears to be. It is a sequence of bytes, stored on a persistent storage device and has metadata associated that makes it possible to locate all that data in actual disk sectors. The bytes inside the file are of no concern to the operating system. When data is read from a file, the operating system consults the file system metadata to find the particular sectors on disk that store the content. When data is overwritten, the same thing happens but those sectors are rewritten with the new data. When new data is appended, the operating system looks up some free space on the disk, then adjusts the file metadata to point at the (now taken) sectors and writes the data in there. There is some additional metadata stored alongside each file, like whom it belongs to or when it was modified.

Directories

106

- a **list** of files and other directories
 - internal nodes of the filesystem tree
 - directories give names to files
- can be opened just like files
 - but `read()` and `write()` is not allowed
 - files are created with `open()` or `creat()`
 - directories with `mkdir()`
 - directory listing with `opendir()` and `readdir()`

A directory is a (potentially) internal node in the file hierarchy: their role is to give **names** to files, making them accessible via **paths**. Like regular files, directories are self-contained objects, but instead of raw bytes, they contain structured data: namely, a directory maps file names to other files (those can be regular files, other directories, or one of the special file types we will talk about shortly).

In principle, it would be possible to implement `read` and `write` for directories, but this would be problematic: if those functions dealt with the actual on-disk representation of a directory, user programs could easily corrupt directory entries. This is quite undesirable: instead, under normal circumstances, directories are used via **paths**: when we present a file path to `open`, the operating system will automatically traverse directories as needed.

Of course, user programs sometimes need to iterate through all direc-

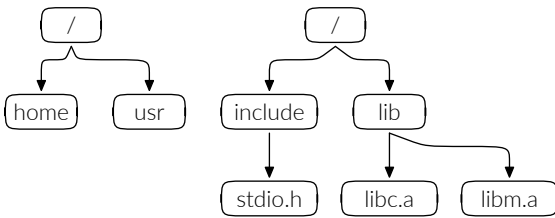
tory entries, i.e. list all files in a given directory. To this end, POSIX provides the `opendir` function along with `readdir`, `seekdir`, `closedir` and so on. These functions provide a high-level API for interacting with directories. Nonetheless, this API is read-only: directory entries are created whenever files are created using the corresponding path, e.g. using `mkdir` or `open` with the `O_CREAT` flag.

Mounts

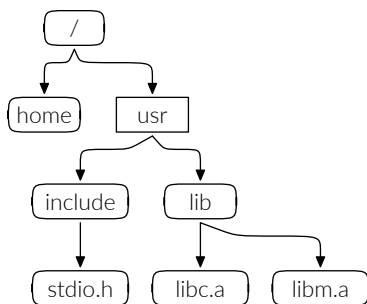
107

- UNIX joins all file systems into a single hierarchy
- the root of one filesystem becomes a directory in another
 - this is called a **mount point**
- Windows uses **drive letters** instead (`C:`, `D:` &c.)

A single computer (and hence, a single operating system) may have more than one hard drive available to it. In this case, it is customary that each such device contains its own file system: the question arises, how to present such multiple file systems to the user. The UNIX strategy is to present all the file systems within a single directory tree: to this end, one of the file systems is picked as a **root** file system: in this (and only in this) file system, the FS root directory `/` is the same as the system root directory. All other file systems are joined existing directories of other file systems at their root. Consider two file systems:



If we now **mount** the second file system onto the `/usr` directory of the first, we get the following unified hierarchy:



Usually, file systems are mounted onto **empty** directories: if `/usr` was not empty on the left (root) file system, its content would be hidden by the mount.

The other strategy is to present multiple file systems using multiple separate trees. This is the strategy implemented by the MS Windows family of operating systems: each file system is assigned a single letter, and each becomes its own, separate tree.

Pipes

108

- pipes are a simple communication device
- one program can `write()` data to the pipe
- another program can `read()` that same data
- each end of the pipe gets a **file descriptor**
- a pipe can live in the filesystem (**named pipe**)

Pipes are somewhat like files in that it's possible to write data (bytes) into them, and read data from them. In most cases, the program doing the writing is a different program from the one doing the reading. Unlike a regular file, the data is not permanently stored anywhere: it

disappears from the pipe as soon as it is read.

Of course, there is a buffer associated with a pipe, but it is only stored in RAM. This allows the writing process to write data even if the other end is not actively reading at the same time: the OS will buffer the write until such time it can be read.

Normally, a pipe is an anonymous device, only accessible via file descriptors. When these are closed, the device is destroyed. There is another variant of a pipe, though, called a **named pipe** which is given a name in the file system. This does not mean the **data** is stored anywhere: a named pipe operates just like an anonymous pipe, the difference is that it can be passed to `open` using its a path.

Devices

109

- **block** and **character** devices are (special) **files**
- block devices are accessed one **block at a time**
 - a typical block device would be a **disk**
 - includes USB mass storage, flash storage, etc
 - you can create a **file system** on a block device
- **character** devices are more like normal files
 - terminals, tapes, serial ports, audio devices

As we have already mentioned, many peripheral devices look like sequences of bytes, or possibly as sequences of blocks. A typical block device is **addressable**: the user can seek to a particular location of the device and read a chunk of data (an integer number of blocks). Unless someone writes to a particular location of the device, reading from the same address multiple times will yield the same data.

On the other hand, character devices often behave rather like pipes, in the sense that if a program writes some bytes into the device, reading the device will not yield those same bytes. Instead, character devices usually mediate communication with a peripheral that consumes bytes (which are written into the device) and/or provides some output (which is what the program gets when it reads from the device). Consider a printer: writing bytes into the printer's character device will cause those bytes to be printed (after possibly being interpreted by the printer).

Another example would be that after a scanner has been instructed to scan a document, the pixels captured by its optical sensor can be, in some form, extracted by reading from its character device. Essentially, these types of character devices behave like a pipe, but instead of another program, the other end is a hardware device (or rather its firmware).

Sockets

110

- the socket API comes from early BSD Unix
- socket represents a (possible) **network connection**
- sockets are more complicated than normal files
 - establishing connections is hard
 - messages get lost much more often than file data
- you get a **file descriptor** for an open socket
- you can `read()` and `write()` to sockets

Sockets are, in some sense, a generalization of pipes. There are essentially 3 types of sockets:

1. a **listening** socket, which allows many **clients** to connect to a single **server** – strictly speaking, these sockets do not transport data, instead, they allow processes to establish **connections**,
2. a **connected** socket, one of which is created for each connection and which behaves essentially like a bidirectional pipe (standard pipes being unidirectional),
3. a **datagram** socket, which can be used to send data without estab-

lishing connections, using special send/receive API.

While the third is rather special and un-pipe-like, the first two are usually used together as a point-to-multipoint means of communication: the server listens on an **address**, and any client which has this address can establish communication with the server. This is quite unlike pipes, which usually need to be pre-arranged (i.e. the programs must already be aware of each other).

Socket Types

111

- sockets can be **internet** or **unix domain**
 - internet sockets connect to other computers
 - Unix sockets live in the filesystem
- sockets can be **stream** or **datagram**
 - stream sockets are like files
 - you can write a continuous **stream** of data
 - datagram sockets can send individual **messages**

There are two basic address types: internet sockets, which are used for inter-machine communication (using TCP/IP), and **unix domain sockets**, which are used for local communication. A unix socket is like a named pipe: it has a path in the file system, and which client programs can use to establish a connection to the server.

Review Questions

112

- What is a shared (dynamic) library?
- What does a linker do?
- What is a symbol in an object file?
- What is a file descriptor?

Part 3: The Kernel

This lecture is about the kernel, the lowest layer of an operating system. It will be in 5 parts:

Lecture Overview

114

1. privileged mode
2. booting
3. kernel architecture
4. system calls
5. kernel-provided services

First, we will look at processor modes and how they mesh with the layering of the operating system. We will move on to the boot process, because it somewhat illustrates the relationship between the kernel and other components of the operating system, and also between the firmware and the kernel. We will look in more detail at kernel architecture: things that we already hinted at in previous lectures, and will also look at exokernels and unikernels, in addition to the architectures we already know (micro and monolithic kernels).

The fourth part will focus on system calls and their binary interface – i.e. how system calls are actually implemented at the machine level. This is closely related to the first part of the lecture about processor modes, and builds on the knowledge we gained last week about how system calls look at the C level.

Finally, we will look at what are the services that kernels provide to the rest of the operating system, what are their responsibilities and we will also look more closely at how microkernel and hybrid operating systems work.

There are two basic address types: internet sockets, which are used for inter-machine communication (using TCP/IP), and **unix domain sockets**, which are used for local communication. A unix socket is like a named pipe: it has a path in the file system, and which client programs can use to establish a connection to the server.

Review Questions

112

- What is a shared (dynamic) library?
- What does a linker do?
- What is a symbol in an object file?
- What is a file descriptor?

Part 3.1: Privileged Mode

CPU Modes

117

- CPUs provide a **privileged** (supervisor) and a **user** mode
- this is the case with all modern **general-purpose** CPUs
 - not necessarily with micro-controllers
- x86 provides 4 distinct privilege levels
 - most systems only use **ring 0** and **ring 3**
 - Xen paravirtualisation uses ring 1 for guest kernels

There is a number of operations that only programs running in supervisor mode can perform. This allows kernels to enforce boundaries between user programs. Sometimes, there are intermediate privilege levels, which allow finer-grained layering of the operating system, for instance, drivers can run in a less privileged level than the 'core' of the kernel, providing a level of protection for the kernel from its own device drivers. You might remember that device drivers are the most problematic part of any kernel.

In addition to device drivers, multi-layer privilege systems in CPUs can be used in certain virtualisation systems. More about this towards the end of the semester.

Reminder: Software Layering

115

- → the **kernel** ←
- system **libraries**
- system services / **daemons**
- utilities
- **application** software

Privileged Mode

118

- many operations are **restricted** in **user mode**
 - this is how **user programs** are executed
 - also **most** of the operating system
- software running in privileged mode can do ~anything
 - most importantly it can program the **MMU**
 - the **kernel** runs in this mode

The kernel executes in privileged mode of the processor. In this mode, the software is allowed to do anything that's possible. In particular, it can (re)program the memory management unit (MMU, see next slide). Since MMU is how program separation is implemented, code executing in privileged mode is allowed to change the memory of any program running on the computer. This explains why we want to reduce the amount of code running in supervisor (privileged) mode to a minimum. The way most operating systems operate, the kernel is the only piece of software that is allowed to run in this mode. The code in system libraries, daemons and so on, including application software, is restricted to the user mode of the processor. In this mode, the MMU cannot be programmed, and the software can only do what the MMU allows based on the instructions it got from the kernel.

Memory Management Unit

119

- is a subsystem of the processor
- takes care of **address translation**
 - user software uses **virtual addresses**
 - the MMU translates them to **physical addresses**
- the mappings can be managed by the OS kernel

Let's have a closer look at the MMU. Its primary role is **address translation**. Addresses that programs refer to are **virtual** – they do not correspond to fixed physical locations in memory chips. Whenever you look at, say, a pointer in C code, that pointer's numeric value is an address in some virtual address space. The job of the MMU is to translate that virtual address into a physical one – which has a fixed relationship with some physical capacitor or other electronic device that remembers information.

How those addresses are mapped is programmable: the kernel can tell the MMU how the translation goes, by providing it with translation tables. We will discuss how page tables work in a short while; what is important now is that it is the job of the kernel to build them and send them to the MMU.

Paging

120

- physical memory is split into **frames**
- virtual memory is split into **pages**
- pages and frames have the same size (usually 4KiB)
- frames are places, pages are the content
- **page tables** map between pages and frames

Before we get to virtual addresses, let's have a look at the other major use for the address translation mechanism, and that is **paging**. We do so, because it perhaps better illustrates how the MMU works. In this viewpoint, we split physical memory (the physical address space) into **frames**, which are **storage areas**: places where we can put data and retrieve it later. Think of them as shelves in a bookcase.

The virtual address space is then split into **pages**: actual pieces of data of some fixed size. Pages do not physically exist, they just represent some bits that the program needs stored. You could think of a page as

a really big integer. Or you can think of pages as a bunch of books that fits into a single shelf.

The page table, then, is an catalog, or an address book of sorts. Programs attach names to pages – the books – but the hardware can only locate shelves. The job of the MMU is to take a name of the book and find the physical shelf where the book is stored. Clearly, the operating system is free to move the books around, as long as it keeps the page table – the catalog – up to date. Remaining software won't know the difference.

Swapping Pages

121

- RAM used to be a scarce resource
- paging allows the OS to **move pages** out of RAM
 - a page (content) can be written to disk
 - and the frame can be used for another page
- not as important with contemporary hardware
- useful for **memory mapping files** (cf. next lecture)

If we are short on shelf space, we may want to move some books into storage. Then we can use the shelf we freed up for some other books. However, hardware can only retrieve information from shelves and therefore if a program asks for a book that is currently in storage, the operating system must arrange things so that it is moved from storage to a shelf before the program is allowed to continue.

This process is called swapping: the OS, when pressed for memory, will evict pages from RAM onto disk or some other high-capacity (but slow) medium. It will only page them back in when they are required. In contemporary computers, memory is not very scarce and this use-case is not very important.

However, it allows another neat trick: instead of opening a file and reading it using **open** and **read** system calls, we can use so-called memory mapped files. This basically provides the content of the file as a chunk of memory that can be read or even written to, and the changes are sent back to the filesystem. We will discuss this in more detail next week.

Look Ahead: Processes

122

- process is primarily defined by its **address space**
 - address space meaning the valid **virtual** addresses
- this is implemented via the MMU
- when changing processes, a different page table is loaded
 - this is called a **context switch**
- the **page table** defines what the process can see

We will deal with processes later in the course, but let me just quickly introduce the concept now, so that we can appreciate how important the MMU is for a modern operating system. Each process has its own **address space** which describes what addresses are valid for that process. Barring additional restrictions, the process can write to any of its valid addresses and then read back the stored value from that address.

The fact that the address space of a process is **abstract** and not tied to any particular physical layout of memory is quite important. Another important observation is that the address space does not need to be contiguous, and that not all physical memory has to be visible in that address space.

Memory Maps

123

- different view of the same principles
- the OS **maps** physical memory into the process
- multiple processes can have the same RAM area mapped
 - this is called **shared memory**
- often, a piece of RAM is only mapped in a **single process**

We can look at the same thing from another point of view. Physical memory is a resource, and the operating system can 'hand out' a piece of physical memory to a process. This is done by **mapping** that piece of memory into the address space of the process. There is nothing that, in principle, prevents the operating system from mapping the same physical piece of RAM into multiple processes. In this case, the data is only stored once, but either process can read it using an address in its virtual address space (possibly at a different address in each process). For 'working' memory – that is both read and written by the program – it is most common that any given area of physical memory is only mapped into a single process. Instructions are, however, shared much more often: for instance a shared library is often mapped into multiple different processes. An executable itself may likewise be mapped into a number of processes if they all run the same program.

Page Tables

124

- the MMU is programmed using **translation tables**
 - those tables are stored in RAM
 - they are usually called **page tables**
- and they are fully in the management of the kernel
- the kernel can ask the MMU to replace the page table
 - this is how processes are isolated from each other

The actual implementation mechanism of virtual memory is known as **page tables**: those are translation tables that tell the MMU which virtual address maps to which physical address. Page tables are stored in memory, just like any other data, and can be created and changed by the kernel. The kernel usually keeps a separate set of page tables for each process, and when a context switch happens, it asks the MMU to replace the active page table with a new one (the one that belongs to the process which is being activated). This is usually achieved by storing the physical address of the first level of the new page table (the page directory, in x86 terminology) in a special register. Often, the writable physical memory referenced by the first set of page tables will be unreachable from the second set and vice versa. Even if there is overlap, it will be comparatively small (processes can request shared memory for communicating with each other). Therefore, whatever data the previous process wrote into its memory becomes completely invisible to the new process.

Kernel Protection

125

- kernel memory is usually mapped into **all processes**
 - this **improves performance** on many CPUs
 - (until **meltdown** hit us, anyway)
- kernel pages have a special 'supervisor' flag set
 - code executing in user mode **cannot touch them**
 - else, user code could **tamper** with kernel memory

Replacing the page tables is usually a rather expensive operation and we want to avoid doing it as much as possible. We especially want to avoid it in the **system call** path (you probably remember system calls from last week, and we will talk about system calls in more detail later

today). For this reason, it is a commonly employed trick to map the kernel into **each process**, but make the memory inaccessible to user-space code. Unfortunately, there have been some CPU bugs which make this less secure than we would like.

Part 3.2: Booting

The boot process is a sequence of steps which starts with the computer powered off and ends when the computer is ready to interact with the user (via the operating system).

Starting the OS

127

- upon power on, the system is in a **default state**
 - mainly because **RAM is volatile**
- the entire **platform** needs to be **initialised**
 - this is first and foremost **the CPU**
 - and the **console** hardware (keyboard, monitor, ...)
 - then the rest of the devices

Computers can be turned off and on (clearly). When they are turned off, power is no longer available and dynamic RAM will, without active refresh, quickly forget everything it held. Hence when we turn the computer on, there is nothing in RAM, the CPU is in some sort of default state and variations of the same are true of pretty much every sub-device in the computer. Except for the content of persistent storage, the computer is in the state it was when it left the factory door. The computer in this state is, to put it bluntly, not very useful.

Boot Process

128

- the process starts with a built-in hardware init
- when ready, the hardware hands off to the **firmware**
 - this was BIOS on 16 and 32 bit systems
 - replaced with EFI on current **amd64** platforms
- the firmware then loads a **bootloader**
- the bootloader **loads the kernel**

We will not get into the hardware part of the sequence. The switch is flipped, the hardware powers up and does its thing. At some point, firmware takes over and does some more things. The hardware and firmware is finally put into a state, where it can begin loading the operating system. There is usually a piece of software that the firmware loads from persistent storage, called a **bootloader**. This bootloader is, more or less, a part of the operating system: its purpose is to find and load the kernel (from persistent storage, usually by using firmware services to identify said storage and load data from it). It may or may not understand file systems and similar high-level things. In the simplest case, the bootloader has a list of disk blocks in which the kernel is stored, and requests those from the firmware. In modern systems, both the firmware and the bootloader are quite sophisticated, and understand complicated, high-level things (including e.g. encrypted drives).

Boot Process (cont'd)

129

- the kernel then initialises **device drivers**
- and the **root filesystem**
- then it hands off to the **init** process
- at this point, the **user space** takes over

We are finally getting to familiar ground. The bootloader has loaded the kernel into RAM and jumped at a pre-arranged address inside the

kernel image. The instructions stored at that address kickstart the kernel initialization sequence. The first part is usually still rather low-level: it puts the CPU and some basic peripherals (console, timers and so on) into a state in which the operating system can use them. Then it hands off control into C code, which then sets up basic data structures used by the kernel. Then the kernel starts initializing individual peripheral devices – this task is performed by individual device drivers. When peripherals are initialized, the kernel can start looking for the **root filesystem** – it is usually stored on one of the attached persistent storage devices (which should now be operational and available to the kernel via their device drivers).

After mounting the root filesystem, the kernel can set up an empty process and load the **init** program into that process, and hand over control. At this point, kernel stops behaving like a sequential program with **main** in it and fades into background: all action is driven by user-space processes from now on (or by hardware interrupts, but we will talk about those much later in the course).

User-mode Initialisation

130

- **init** mounts the remaining file systems
- the **init** process starts up user-mode **system services**
- then it starts **application services**
- and finally the **login** process

We are far from done. The **init** process now needs to hunt down all the other file systems and mount them, start a whole bunch of **system services** and perhaps some **application services** (daemons which are not part of the operating system – things like web servers).

Once all the essential services are ready, **init** starts the **login** process, which then presents the familiar login screen, asking the user to type in their name and password. At this point, the boot process is complete, but we will have a quick look at one more step.

After Log-In

131

- the **login** process initiates the **user session**
- loads **desktop** modules and **application software**
- drops the user in a (text or graphical) **shell**
- now you can start using the computer

When the user logs in, another initialization sequence starts: the system needs to set up a **session** for the user. Again, this involves some steps, but at the end, it's finally possible to interact with the computer.

CPU Init

132

- this depends on both **architecture** and **platform**
- on **x86**, the CPU starts in **16-bit** mode
- on legacy systems, BIOS & bootloader stay in this mode
- the kernel then switches to **protected mode** during its boot

Let's go back to start and fill in some additional details. First of all, what is the state of the CPU at boot, and why does the operating system need to do anything? This has to do with backward compatibility: a CPU usually starts up in the most-compatible mode – in case of 32b x86 processors, this is 16b mode with the MMU disabled. Since the entire platform keeps backward compatibility, the firmware keeps the CPU in this mode and it is the job of either the bootloader or the kernel itself to fix this. This is not always the case (modern 64b x86 processors still start up in 16b mode, but the firmware puts them into **long mode** – that is the 64b one – before handing off to the bootloader).

Bootloader

133

- historically limited to tens of **kilobytes** of code
- the bootloader locates the kernel **on disk**
 - may allow the operator to choose different kernels
 - **limited** understanding of **file systems**
- then it **loads the kernel** image into **RAM**
- and hands off control to the kernel

A bootloader is a short, platform-specific program which loads the kernel from persistent storage (usually a file system on a disk) and hands off execution to the kernel. The bootloader might do some very basic hardware initialization, but most of that is done by the kernel itself in a later stage.

Modern Booting on x86

134

- the bootloader nowadays runs in **protected mode**
 - or even the long mode on 64-bit CPUs
- the firmware understands the **FAT** filesystem
 - it can **load files** from there into memory
 - this vastly **simplifies** the boot process

The boot process has been considerably simplified on x86 computers in the last decade or so. Much higher-level APIs have been added to the standardized firmware interface, making the boot code considerably simpler.

Booting ARM

135

- on ARM boards, there is **no unified firmware** interface
- U-boot is as close as one gets to unification
- the bootloader needs **low-level** hardware knowledge
- this makes writing bootloaders for ARM quite **tedious**
- current U-boot can use the **EFI protocol** from PCs

Unlike the x86 world, the ARM ecosystem is far less standardized and each system on a chip needs a slightly different boot process. This is extremely impractical, since there are dozens of SoC models from many different vendors, and new ones come out regularly. Fortunately, U-boot has become a de-facto standard, and while U-boot itself still needs to be adapted to each new SoC or even each board, the operating system is, nowadays, mostly insulated from the complexity.

Part 3.3: Kernel Architecture

In this section, we will look at different architectures (designs) of kernels: the main distinction we will talk about is which services and components are part of the kernel proper, and which are outside of the kernel.

Architecture Types

137

- **monolithic** kernels (Linux, *BSD)
- microkernels (Mach, L4, QNX, NT, ...)
- **hybrid** kernels (macOS)
- type 1 **hypervisors** (Xen)
- exokernels, rump kernels

We have already mentioned the main two kernel types earlier in the course. Those types represent the extremes of mainstream kernel de-

sign: microkernels are the smallest (most exclusive) mainstream design, while monolithic kernels are the biggest (most inclusive). Systems with **hybrid** kernels are a natural compromise between those two extremal designs: they have 2 components, a microkernel and a so-called **super-server**, which is essentially a gutted monolithic kernel – that is, the functionality covered by the microkernel is removed.

Besides ‘mainstream’ kernel designs, there are a few more exotic choices. We could consider type 1 (bare metal) hypervisors to be a special type of an operating system kernel, where **applications** are simply virtual machines – i.e. ‘normal’ operating systems (more on this later in the course). Then there are **exokernel** operating systems, which drastically cut down on services provided to applications and **unikernels** which are basically libraries for running entire applications in kernel mode.

Microkernel

138

- handles **memory protection**
- (hardware) interrupts
- task / process **scheduling**
- **message passing**
- everything else is **separate**

A microkernel handles only the essential services – those that cannot be reasonably done outside of the kernel (that is, outside of the privileged mode of the CPU). This obviously includes programming the MMU (i.e. management of address spaces and memory protection), handling interrupts (those switch the CPU into privileged mode, so at least the initial interrupt routine needs to be part of the kernel), thread and process switching (and typically also scheduling) and finally some form of inter-process communication mechanism (typically message passing). With those components in the kernel, almost everything else can be realized outside the kernel proper (though device drivers do need some additional low-level services from the kernel not listed here, like DMA programming and delegation of hardware interrupts).

Monolithic kernels

139

- all that a microkernel does
- plus device drivers
- file systems, volume management
- a network stack
- data encryption, ...

A monolithic kernel needs to include everything that a microkernel does (even though some of the bits have a slightly different form, at least typically: inter-process communication is present, but may be of different type, driver integration looks different). However, there are many additional responsibilities: many device drivers (those that need interrupts or DMA, or are otherwise performance-critical) are integrated into the kernel, as are file systems and volume (disk) management. A complete TCP/IP stack is almost a given. A number of additional bits and pieces might be part of the kernel, like cryptographic services (key management, disk encryption, etc.), packet filtering, a kitchen sink and so on. Of course, all that code runs in privileged mode, and as such has complete power over the operating system and the computer as a whole.

Microkernel Redux

140

- we need a lot more than a microkernel provides
- in a “true” microkernel OS, there are many modules
- each **device driver** runs in a **separate process**
- the same for **file systems** and networking
- those modules / processes are called **servers**

The question that now arises is who is responsible for all the services listed on the previous slide (those that are part of a monolithic kernel, but are missing from a microkernel). In a ‘true’ microkernel operating system, those services are individually covered, each by a separate process (also known as a **server** in this context).

Hybrid Kernels

141

- based around a microkernel
- **and** a gutted monolithic kernel
- the monolithic kernel is a big server
 - takes care of stuff not handled by the microkernel
 - easier to implement than true microkernel OS
 - strikes middle ground on performance

In a hybrid kernel, most of the services are provided by a single large server, which is somewhat isolated from the hardware. It is often the case that the server is based on a monolithic OS kernel, with the lowest-level layers removed, and replaced with calls to the microkernel as appropriate.

Hybrid kernels are both cheaper to design and theoretically perform better than ‘true’ (multi-server) microkernel systems.

Micro vs Mono

142

- microkernels are more **robust**
- monolithic kernels are more **efficient**
 - less context switching
- what is easier to implement is debatable
 - in the short view, monolithic wins
- hybrid kernels are a **compromise**

The main advantage of microkernels is their robustness in face of software bugs. Since the kernel itself is small, chances of a bug in the kernel proper are much diminished compared to the relatively huge code base of a monolithic kernel. The impact of bugs outside the kernel (in servers) is considerably smaller, since those are isolated from the rest of the system and even if they provide vital services, the system can often recover from a failure by restarting the failed server.

On the other hand, monolithic kernels offer better performance, mainly through reduced context switching, which is still fairly expensive even on modern, virtualisation-capable processors. However, as monolithic kernels adopt technologies such as kernel page table isolation to improve their security properties, the performance difference becomes smaller.

Implementation-wise, monolithic kernels offer two advantages: in many cases, code can be written in direct, synchronous style, and different parts of the kernel can share data structures without additional effort. In contrast, a proper multi-server system often has to use asynchronous communication (message passing) to achieve the same goals, making the code harder to write and harder to understand. Long-term, improved modularity and isolation of components could outweigh the short-term gains in programming efficiency due to more direct programming style.

Exokernels

143

- smaller than a microkernel
- much **fewer abstractions**
 - applications only get **block** storage
 - networking is much reduced
- only **research systems** exist

Operating systems based on microkernels still provide the full suite of services to their applications, including file systems, network stacks and so on. The difference lies in where this functionality is implemented, whether the kernel proper, or in a user-mode server. With exokernels, this is no longer true: the services provided by the operating system are severely cut down. The resulting system is somewhere between a paravirtualized computer (we will discuss this concept in more detail near the end of the course) and a 'standard' operating system. Unlike virtual machines (and unikernels), process-based application isolation is still available, and plays an important role. No production systems based on this architecture currently exist.

Type 1 Hypervisors

144

- also known as **bare metal** or **native** hypervisors
- they resemble microkernel operating systems
 - or exokernels, depending on the viewpoint
- "applications" for a hypervisor are **operating systems**
 - hypervisor can use **coarser abstractions** than an OS
 - entire storage devices instead of a filesystem

A bare metal hypervisor is similar to an exokernel or a microkernel operating system (depending on a particular hypervisor and on our point of view). Typically, a hypervisor provides interfaces and resources that are traditionally implemented in hardware: block devices, network interfaces, a virtual CPU, including a virtual MMU that allows the 'applications' (i.e. the guest operating systems) to take advantage of paging.

Unikernels

145

- kernels for running a **single application**
 - makes little sense on real hardware
 - but can be very useful on a **hypervisor**
- bundle applications as **virtual machines**
 - without the overhead of a general-purpose OS

Unikernels constitute a different strand (compared to exokernels) of minimalist operating system design. In this case, process-level multi-tasking and address space isolation are not part of the kernel: instead, the kernel exists to support a single application by providing (a subset of) traditional OS abstractions like a networking stack, a hierarchical file system and so on. When an application is bundled with a compatible unikernel, the result can be executed directly on a hypervisor (or an exokernel).

Exo vs Uni

146

- an exokernel runs **multiple applications**
 - includes process-based isolation
 - but **abstractions** are very **bare-bones**
- unikernel only runs a **single application**
 - provides more-or-less **standard services**
 - e.g. standard hierarchical file system
 - socket-based network stack / API

Part 3.4: System Calls

In the remainder of this lecture, we will focus on monolithic kernels, since the more progressive designs do not use the traditional system call mechanism. In those systems, most 'system calls' are implemented through message passing, and only services provided directly by the microkernel use a mechanism that resembles system calls as described in this section.

Reminder: Kernel Protection

148

- kernel executes in **privileged** mode of the CPU
- kernel memory is protected from user code

But: Kernel Services

- user code needs to ask kernel for **services**
- how do we **switch the CPU** into privileged mode?
- **cannot** be done arbitrarily (security)

The main purpose of the system call interface is to allow secure transfer of control between a user-space application and the kernel. Recall that each executes with different level of privileges (at the CPU level). A viable system call mechanism must allow the application to switch the CPU into privileged mode (so that the CPU can execute kernel code), but in a way that does not allow the application to execute its own code in this mode.

System Calls

149

- hand off execution to a **kernel routine**
- pass **arguments** into the kernel
- obtain **return value** from the kernel
- all of this must be done **safely**

We would like system calls to behave more-or-less like standard sub-routines (e.g. those provided by system libraries): this means that we want to pass arguments to the subroutine and obtain its return value. Like with the transfer of control flow, we need the argument passing to be safe: the user-space side of the call must not be able to read or modify kernel memory.

Trapping into the Kernel

150

- there are a few possible mechanisms
- details are very **architecture-specific**
- in general, the kernel sets a fixed **entry address**
 - an instruction changes the CPU into privileged mode
 - while **at the same time** jumping to this address

Security from execution of arbitrary code by the application is achieved

by tying the privilege escalation (i.e. the entry into the privileged CPU mode) to a simultaneous transfer of execution to a fixed address, which the application is unable to change. The exact mechanism is highly architecture-dependent, but the principle outlined here is universal.

Trap Example: x86

151

- there is an `int` instruction on those CPUs
- this is called a **software interrupt**
 - interrupts are normally a **hardware** thing
 - interrupt **handlers** run in **privileged mode**
- it is also synchronous
- the handler is set in `IDT` (interrupt descriptor table)

On traditional (32 bit) x86 CPUs, the preferred method of implementing the system call trap was through **software interrupts**. In this case, the application uses an `int` instruction, which causes the CPU to perform a process analogous to a hardware interrupt. The two important aspects are:

1. the CPU switches into privileged mode to execute the **interrupt handler**,
2. reads the address to jump to from an **interrupt handler table**, which is a data structure stored in RAM, at an address given by a special register.

The kernel sets up the interrupt handler table in such a way that user-level code cannot change it (via standard MMU-based memory protection). The register which holds its address cannot be changed outside of privileged mode.

Software Interrupts

152

- those are available on a range of CPUs
- generally **not very efficient** for system calls
- extra level of indirection
 - the handler address is retrieved from memory
 - a **lot of CPU state** needs to be saved

A similar mechanism is available on many other processor architectures. There are, however, some downsides to using this approach for system calls, the main being their poor performance. Since the mechanism piggy-backs on the hardware variety of interrupts, the CPU usually saves a lot more computation state than would be required. As an additional inconvenience, there are multiple entry-points, which must therefore be stored in RAM (instead of a register), causing additional delays when the CPU needs to read the interrupt table. Finally, arguments must be passed through memory, since registers are reset by the interrupt, again contributing to increased latency.

Aside: SW Interrupts on PCs

153

- those are used even in **real mode**
 - legacy 16-bit mode of 80x86 CPUs
 - BIOS (firmware) routines via `int 0x10` & `0x13`
 - MS-DOS API via `int 0x21`
- and on older CPUs in 32-bit **protected mode**
 - Windows NT uses `int 0x2e`
 - Linux uses `int 0x80`

On the ubiquitous x86 architecture, software interrupts were the preferred mechanism to provide services to application programs until the end of the 32-bit x86 era. Interestingly, x86 CPUs since 80386 offer a mechanism that was directly intended to implement operating system

services (i.e. syscalls), but it was rather complex and largely ignored by operating system programmers.

Trap Example: amd64 / x86_64

154

- `sysenter` and `syscall` instructions
 - and corresponding `sysexit` / `sysret`
- the entry point is stored in a **machine state register**
- there is only **one entry point**
 - unlike with software interrupts
- quite a bit **faster** than interrupts

When x86 switched to a 64-bit address space, many new instructions found their way into the instruction set. Among those was a simple, single-entypoint privilege escalation instruction. This mechanism avoids most of the overhead associated with software interrupts: computation state is managed in software, allowing compilers to only save and restore a small number of registers across the system call (instead of having the CPU automatically save its entire state into memory).

Which System Call?

155

- often there are **many** system calls
 - there are more than 300 on 64-bit Linux
 - about 400 on 32-bit Windows NT
- but there is only a **handful of interrupts**
 - and only one `sysenter` address

Usually, there is only a single entry point (address) shared by all system calls. However, the kernel needs to be able to figure out which service the application program requested.

Reminder: System Call Numbers

156

- each system call is assigned a **number**
- available as `SYS_write` &c. on POSIX systems
- for the “universal” `int syscall(int sys, ...)`
- this number is passed in a CPU register

This is achieved by simply sending the **syscall number** as an argument in a specific CPU register. The kernel can then decide, based on this number, which kernel routine to execute on behalf of the program.

System Call Sequence

157

- first, `libc` prepares the system call **arguments**
- and puts the system call **number** in the correct register
- then the CPU is switched into **privileged mode**
- this also transfers control to the **syscall handler**

The first stage of a system call is executed in user mode, and is usually implemented in `libc`.

System Call Handler

158

- the handler first picks up the system call **number**
- and decides where to continue
- you can imagine this as a giant **switch** statement

```
switch ( sysnum )
{
    case SYS_write: return syscall_write();
    case SYS_read: return syscall_read();
    /* many more */
}
```

After the switch to privileged mode, the kernel needs to make sense of the arguments that the user program provided, and most importantly, decide which system call was requested. The code to do this in the kernel might look like the above **switch** statement.

System Call Arguments

159

- each system call has **different arguments**
- how they are passed to the kernel is **CPU-dependent**
- on 32-bit **x86**, most of them are passed **in memory**
- on **amd64** Linux, all arguments go into **registers**
 - 6 registers available for arguments

Since different system calls expect different arguments, the specific argument processing is done after the system call is dispatched based on its number. In modern systems, arguments are passed in CPU registers, but this was not possible with protocols based on software interrupts (instead, arguments would be passed through memory, usually at the top of the user-space stack).

Part 3.5: Kernel Services

Finally, we will revisit the services offered by monolithic kernels, and look at how they are realized in microkernel operating systems.

What Does a Kernel Do?

161

- **memory** & process management
- task (thread) **scheduling**
- device drivers
 - SSDs, GPUs, USB, bluetooth, HID, audio, ...
- file systems
- networking

The first two points are a core responsibility of the kernel: those are rarely 'outsourced' into external services. The remaining services are a core part of an **operating system**, but not necessarily of a kernel. However, it is hard to imagine a modern, general-purpose operating system which would omit any of them. In traditional (monolithic) designs, they are all part of the kernel.

Additional Services

162

- inter-process **communication**
- timers and time keeping
- process tracing, profiling
- security, sandboxing
- cryptography

A monolithic kernel may provide a number of additional services, with varying importance. Not all systems provide all the services, and the implementations can look quite differently across operating systems. Out of this (incomplete) list, IPC (inter-process communication) is the only item that is quite universally present, in some form, in microkernels. Moreover, while dedicated IPC mechanisms are common in monolithic kernels, they are more important in a microkernel.

Reminder: Microkernel Systems

163

- the kernel proper is **very small**
- it is accompanied by **servers**
- in "true" microkernel systems, there are **many servers**
 - each device, filesystem, etc. is separate
- in **hybrid** systems, there is one, or a few
 - a "superserver" that resembles a monolithic kernel

Recall that a microkernel is small: it only provides services that cannot be reasonably implemented outside of it. Of course, since the operating system as a whole still needs to implement those services. Two basic strategies are available:

1. a single program, running in a single process, implements all the missing functionality: this program is called a superserver, and internally has an architecture that is rather similar to that of a standard monolithic kernel,
2. each service is provided by a separate, specialized program, running in its own process (and hence, address space) – this is characteristic of so-called 'true' microkernel systems.

There are of course different trade-offs involved in those two basic designs. A hybrid system (i.e. one with a superserver) is easier to initially design and implement (for instance, persistent storage drivers, the block layer, and the file system all share the same address space, simplifying the implementation) and is often considerably faster, since communication between components does not involve context switches. On the other hand, a true microkernel system with services and drivers all strictly separated into individual processes is more robust, and in theory also easier to scale to large SMP systems.

Kernel Services

164

- we usually don't care **which server** provides what
 - each system is different
 - for services, we take a **monolithic view**
- the services are used through **system libraries**
 - they abstract away many of the details
 - e.g. whether a service is a **system call** or an **IPC call**

From a user-space point of view, the specifics of kernel architecture should not matter. Applications use system libraries to talk to the kernel in either case: it is up to the libraries in question to implement the protocol for locating relevant servers and interacting with them.

User-Space Drivers in Monolithic Systems

165

- not **all** device drivers are part of the kernel
- case in point: **printer** drivers
- also some **USB devices** (not the USB bus though)
- part of the GPU/graphics stack
 - memory and output management in kernel
 - most of OpenGL in **user space**

While user-space drivers are par for the course in microkernel systems,

there are also certain cases where drivers in operating systems based on monolithic kernels have significant user-space components. The most common example is probably printer drivers: low-level communication with the printer (at the USB level) is mediated by the kernel, but for many printers, document processing comprises a large part of the functionality of the driver. In some cases, this involves format conversion (e.g. PCL printers) but in others, the input document is rasterised by the driver on the main CPU: instead of sending text and layout information to the printer, the driver sends pixel data, or even a stream of commands for the printing head.

The situation with GPUs is somewhat analogous: low-level access to the hardware is provided by the kernel, but again, a large part of the driver is dedicated to data manipulation: dealing with triangle meshes,

textures, lighting and so on. Additionally, modern GPUs are invariably *programmable*: a shader compiler is also part of the driver, translating high-level shader programs into instruction streams that can be executed by the CPU.

We will deal with device drivers in more detail in lecture 8.

Review Questions

166

- What CPU modes are there and how are they used?
- What is the memory management unit?
- What is a microkernel?
- What is a system call?

Part 4: File Systems

File systems are an integral part of general-purpose operating systems. Besides storing user data, the programs and other components that make up an operating system also reside in the file system. In this lecture, we will look at some of the inner workings of a typical POSIX-compatible file system layer.

Lecture Overview

168

1. Filesystem Basics
2. The Block Layer
3. Virtual Filesystem Switch
4. The UNIX Filesystem
5. Advanced Features

We will first revisit some of the basic concepts that have been already introduced in previous lectures. Afterwards, we will look at the **block layer**, which exists below the file system and provides a simpler data storage abstraction. Afterward, we will have a look at **VFS**, a system that kernels typically use to allow multiple implementations of the on-disk structures while providing a uniform system call interface on the outside.

Afterwards, we will look at both the semantics and the on-disk organization of a typical UNIX file system. Finally, we will take a quick look at some of the more advanced features present in modern file systems (and below, in the block layer).

Part 4.1: Filesystem Basics

In this section, we will consider the basic elements of a file system, as understood by POSIX – some of the material will also repeat concepts that we have seen earlier.

What is a File System?

170

- a collection of **files** and **directories**
- (mostly) **hierarchical**
- usually exposed to the user
- usually **persistent** (across reboots)
- file managers, command line, etc.

The first concern is the definition of a file system as a whole. The most abstract view is that a file system is a collection of **files** and **directories**, where files are essentially byte sequences and directories give **names** to files and other directories. Since directories can contain other directories, the whole structure forms a tree (with some exceptions).

File systems are usually user-visible: the user can directly access the directory hierarchy and explore both directory and file content, using tools provided by the operating system. Another typical property of a file system is that it is **persistent**: rebooting the computer will not erase or alter the data stored in the file system. (Note that both these properties have exceptions: file systems can be stored on virtual devices that use RAM for storage, and those are not persistent; likewise, some operating systems do not allow the user to directly access the file system).

What is a (Regular) File?

171

- a sequence of **bytes**
- and some basic **metadata**
 - owner, group, timestamp
- the OS does **not** care about the content
 - text, images, video, source code are all the same
 - executables are somewhat special

Since we have defined a file system as a collection of files and directories, we will need to also give a definition of a **file**. Again, we can give a fairly abstract definition that works: a file is an object that consists of a sequence of bytes (the data) and some additional **metadata** – information about the file. The content (i.e. the byte sequence) is typically not interpreted by the operating system (apart from special cases, like executable files). The **metadata** contains things like the identifier of the **owner** of the file, a few timestamps (e.g. the time of last modification) and access permissions. Please note that the file metadata does **not** include a name: file name is not a property of the file itself in POSIX systems.

What is a Directory?

172

- a list of **name** → **file mappings**
- an associative container if you will
 - semantically, the value types are not homogeneous
 - syntactically, they are just **i-nodes**
- one directory = one component of a path
 - **/usr/local/bin**

The last part of the definition of a file system that we did not explain yet is a **directory**. Directories are maps (associative arrays, dictionaries), where the keys are **names** and the values are **files** and other directories. In fact, both files and directories are represented by the same data structure in the file system: an **i-node**.

We also remember that directories form a tree, and we use **paths** to

identify individual files and directories in the file system. Each component (a section delimited by / on either side) of such a path is used as a key in a single directory.

What is an i-node? 173

- an **anonymous**, file-like object
- could be a **regular file**
 - or a directory
 - or a special file
 - or a symlink

The **i-node** is how files and directories (and any other file system object) is represented by the POSIX file system. The **i-node** stores references to data blocks (more on that later) and the metadata of the file.

Files are Anonymous 174

- this is the case with UNIX
 - not all file systems work like this
- there are pros and cons to this approach
 - e.g. open files can be **unlinked**
- names are assigned via **directory entries**

As we have already mentioned, files do not carry names – i.e. the **i-node** does not have a field which would store the name of the object. Instead, a name is given to the file by a **directory entry**, which ties a string (anything goes as long as there are no / characters) to an **i-node**. Among other things, this means that it is possible to **unlink** files (remove their directory entries) without erasing their data: this happens when the file is currently open. When the last process closes its last file descriptor tied to that **i-node**, the data is finally erased from the disk.

What Else is a Byte Sequence? 175

- characters coming from a **keyboard**
- bytes stored on a magnetic **tape**
- audio data coming from a **microphone**
- pixels coming from a **webcam**
- data coming on a **TCP connection**

We have previously mentioned that a **byte sequence** is a broadly applicable abstraction. There are many things that can be treated (with a bit of generosity) as byte sequences. Most of those are not persistent in the way files are, but this is not really a problem as far as programming interfaces go.

Writing Byte Sequences 176

- sending data to a **printer**
- playing back **audio**
- writing text to a **terminal** (emulator)
- sending data over a **TCP stream**

In addition to reading bytes, files (and many other things) support **writing** bytes, i.e. storing or otherwise processing a sequence of bytes that is sent (written) to the object in question.

Special Files 177

- many things look somewhat **like files**
- let's exploit that and unify them with files
- recall part 2 on APIs: "everything is a file"
 - the **API** is the same for special and regular files
 - **not** the implementation though

The generality of the abstraction makes it possible (and desirable) to represent all such objects uniformly, using the file system API. However, besides using the same API, many of those objects literally appear in the file system as **i-nodes** with special properties. Of course, when such a file is opened, reads and writes to the file descriptor are not handled by the same kernel routines that deal with regular files. More on this in a later section, on **VFS**.

File System Types 178

- fat16, fat32, vfat, exfat (DOS, flash media)
- ISO 9660 (CD-ROMs)
- UDF (DVD-ROM)
- NTFS (Windows NT)
- HFS+ (macOS)
- ext2, ext3, ext4 (Linux)
- ufs, ffs (BSD)

Of course, there are many different implementations of the abstract idea of a file system. While many have identical or similar semantics (in most cases those described in this lecture and codified by POSIX), this is not always the case: file systems in the **FAT** family, for instance, do not have the concept of an i-node and file names are an intrinsic property of the file.

While the semantics are often similar, the underlying disk format can vary rather widely: while the **ext** family and **ufs** family use a fairly traditional approach, many modern file systems, such as **ZFS**, **btrfs** or **hammer** are internally built on B-trees, extendible hashing or other scalable data structures, which can handle much bigger volumes with many more files than the traditional, rather naive approaches. At the same time, modern file systems provide better resilience against corruption of their data structures in the event of errors or unexpected power loss.

Multi-User Systems 179

- file ownership
- file permissions
- disk quotas

Multi-user systems come with an additional set of challenges when it comes to file system implementation. It is usually desirable that files of each user are inaccessible to any other user, unless the owner of the file desires to share that particular file. Likewise, we want to be able to limit how much space each user can take up, which is usually done via disk quotas.

Ownership & Permissions

180

- we assume a **discretionary** model
- whoever creates a file is its **owner**
- ownership can be transferred
- the owner decides about **permissions**
 - basically read, write, execute

For these reasons, the system must be able to track:

1. file system ownership, i.e. remember which file belongs to which user,
2. file permissions, i.e. what the owner of the file deems as appropriate use of their file – whether other users (and which users) can read the content of the file, or even write new data or replace existing data in the file.

Under a discretionary access model, the owner can freely decide about the permissions. There are other models, where this ability is limited (usually to improve security).

Disk Quotas

181

- disks are big but **not infinite**
- bad things happen when the file system fills up
 - denial of service
 - programs may fail and even corrupt data
- **quotas** limits the amount of space per user

Besides limiting what can be done with files, the system must also be able to manage **free space** in the file system: since file systems are stored on some physical medium, the amount of data that can be stored is limited. Disk quotas are a mechanism by which the operating system ensures fair allocation of the space among users (or at least prevents a single user to hog all the space in the file system).

Part 4.2: The Block Layer

The block layer (of an operating system) takes care of low-level access to persistent storage devices, such as hard drives, solid-state drives and other similar devices. The file system itself then does not need to understand the details of the underlying device or the protocol by which to communicate with that device. Instead, it uses a uniform API which allows the file system to store and retrieve blocks of data. Or, in other words, a **block device** is an abstraction which makes file system implementation both easier (fewer details to take care of) and more universal (does not depend on the particulars of a given storage device).

Disk-Like Devices

183

- disk drives provide **block-level** access
- read and write data in 512-byte chunks
 - or also 4K on big modern drives
- a big numbered **array of blocks**

A block device (i.e. a disk-like device) has the following basic operations: it can read or write a **block**, which is a chunk of data of a fixed size. The specific size depends on the type of device, but the usual block sizes are either 512 bytes (in older devices) or 4096 bytes in more modern hardware. The blocks are numbered and the entire device essentially 'looks' like a big array of such blocks. Reading and writing blocks means, in this context, transferring the data stored in that block to or from main memory (RAM).

Aside: Disk Addressing Schemes

184

- CHS: Cylinder, Head, Sector
 - **structured** addressing used in (very) old drives
 - exposes information about relative seek times
 - useless with variable-length cylinders
 - 10:4:6 CHS = 1024 cylinders, 16 heads, 63 sectors
- LBA: Logical Block Addressing
 - **linear**, unstructured address space
 - started as 22, later 28, ... now 48 bit

Old rotational drives used an addressing scheme which reflected their physical geometry. The address consisted of 3 numbers: the cylinder (distance from the center of the platter), the head (which platter or rather which side of which platter stores the sector) and the sector number (the angle at which the sector is stored). This allowed the driver and operating system to predict operation latency: reading sectors from the same cylinder would be usually fast, reading sectors from a different cylinder would require expensive head movements (known as 'seeking').

The CHS scheme was abandoned as disk capacities increased and the address space began to run out. The replacement, known as LBA or Logical Block Addressing, uses an unstructured, flat address space (the same as main memory, but the unit of addressing is a block, not a byte).

Block-Level Access

185

- disk drivers only expose **linear addressing**
- one block (sector) is the **minimum read/write size**
- many sectors can be written 'at once'
 - **sequential** access is faster than random
 - maximum **throughput** vs **IOPS**

Higher layers of an operating system (starting with the block layer and file system implementation) do not care about the interface between the disk driver and the disk drive itself and always use linear addressing. Within the block layer, transfers are often re-organized to maximize sequential writes (we will discuss this shortly), because most disk drives can read or write a contiguous sequence of blocks much faster than they can read them one at a time.

The sequential access regime is important when we ask about the **throughput** of the system (how many bytes the device can supply or store per second), while the number of blocks that it can read or write when they are randomly distributed across the entire device is known as **IOPS** – input/output **operations** per second.

Aside: Access Times

186

- block devices are **slow** (compared to RAM)
 - RAM is **slow** (compared to CPU)
- we cannot treat drives as an extension of RAM
 - not even fastest modern flash storage
 - latency: HDD 3–12 ms, SSD 0.1 ms, RAM 70 ns

The entire design of the storage stack, starting with hardware buses, all the way to file systems and even application-level software, most often database systems, is strongly influenced by the **slowness** of block devices. Even the latest flash-based drives are many times slower than RAM (which is many times slower than CPU caches, which are still slower than the computational units in the CPU).

Block Access Cache

187

- **caching** is used to hide latency
 - same principle between CPU and RAM
- files recently accessed are kept in RAM
 - many **cache management** policies exist
- implemented entirely in the OS
 - many devices implement their own caching
 - but the amount of fast memory is usually limited

You already know that CPU relies on **cache memory** to **hide** latency of RAM: data that has been recently used, or that the system suspects will be required shortly, are stored in, or prefetched into, the much faster (but much smaller) **cache**. This means that for data that is most often used, the CPU does not need to wait out the entire RAM latency time.

The same principle is used by operating systems to hide the latency of block devices, but in this case, the 'fast' and 'small' memory is the main memory (RAM) while the big and slow memory is the block device. Unlike CPU cache, this so-called **disk cache** is implemented entirely in software.

However, there is usually another layer (or two) of caching, which is done in hardware or in drive firmware and which uses dedicated hardware buffers (usually implemented as dynamic RAM or even flash storage) on the device side of the bus. This is distinct from the OS-level cache and we will ignore it from now on.

Write Buffers

188

- the **write** equivalent of the block cache
- data is kept in RAM until it can be processed
- must synchronise with caching
 - other users may be reading the file

There is one major difference between the CPU cache and the block device cache maintained by the OS: there is a persistence mismatch between the cache (RAM) and the main storage (disk). When the system powers down, the content of the RAM is lost and along with it, any modifications which were not yet replicated from the cache onto the persistent storage device. This creates a lot of complexity, since application software (and users) expect data written to disk to stay there. This is especially important (and problematic) in systems which need to be resistant to unexpected power loss, though operating system crashes can cause the same effect even if the system is hooked up to reliable power.

For this reason, many operating systems use a split cache design: data to be written is stored in write buffers and flushed to disk as bandwidth and other resources permit. Of course, data stored in write buffers must be replicated into the read cache (or otherwise shared with it), since other processes may be reading the same blocks (usually via the file system) and should see the new content.

I/O Scheduler (Elevator)

189

- reads and writes are requested by users
- access ordering is crucial on a mechanical drive
 - not as important on an SSD
 - but sequential access is still **much preferred**
- requests are **queued** (recall, disks are slow)
 - but they are **not** processed in **FIFO** order

We have already mentioned that sequential access is much faster than

randomly distributed access, and that latency in rotational drives depends on physical distance between the locations where individual pieces of data are stored. Since there is usually a lot of parallelism going on in the IO subsystems (different processes reading and writing different files at the same time), it often pays to re-order the requests for IO operations.

The desired effect is that a completely random sequence of IO operations coming in from multiple processes, say 16 requests where each request is in a different physical location from the previous, is re-ordered into 4 sets of 4 operations happening in physical proximity (or even in a sequence). This reordering can easily improve overall throughput of the system (in this example situation) 3-4x, since a single batch of 4 sequential operations takes barely any extra time on top of the random-access latency incurred on the first operation in that sequence. The reordering is more easily done with buffered writes: in this case, the block layer is free to shuffle the write queue (subject to ordering constraints imposed by the file system or perhaps by the application). However, reading is more complicated: the OS can certainly speculate what read operations will be requested next. In systems with true asynchronous IO (where the application does not block until the IO is finished), the read queue may be longer, but still more limited than the write queue.

RAID

190

- hard drives are also **unreliable**
 - backups help, but take a long **time to restore**
- RAID = Redundant Array of Inexpensive Disks
 - **live**-replicate same data across multiple drives
 - many different configurations
- the system **stays online** despite disk failures

While **speed** of persistent storage is a major problem, its (lack of) **reliability** is also rather important. While the problem of speed is tackled with caching, reliability is usually improved through **redundancy**: in case a component fails, other components can take over. In case of storage, that means that the data must be replicated across multiple devices, in such a way that if one of them fails, the others can still put together a complete copy of the data.

RAID is a low-level realization of this principle. RAID can be implemented in hardware or in software, the latter being more common in present-day systems. Software RAID is part of the block layer of an operating system, and is usually presented to upper layers as a single virtual device. Reading and writing to this virtual device will cause the RAID subsystem of the block layer to distribute the data across multiple physical devices. Most RAID configurations can then continue operating without data loss (and without interruption) if one of the physical devices fails.

RAID Performance

191

- RAID affects the performance of the block layer
- often **improved reading** throughput
 - data is recombined from multiple channels
- write performance is more **mixed**
 - may require a fair amount of computation
 - **more data** needs to be written for **redundancy**

In a fully functioning RAID array, read performance is usually significantly enhanced: the data is collected from multiple devices in parallel, each contributing part of the overall bandwidth.

Writing is less clear-cut: depending on the RAID configuration, writes may be faster or slower than a single disk. Writing to a software RAID array also consumes additional CPU cycles, since the operating system

must slice the data (and often also compute checksums).

Block-Level Encryption

192

- **symmetric** & length-preserving
- encryption key is derived from a **passphrase**
- also known as “full disk encryption”
- incurs a small **performance penalty**
- very important for **security** / privacy

Another feature commonly present in modern operating systems is block-level **encryption** of data. This hardens the system against off-line attacks: e.g. if the computer is stolen, the data remains inaccessible without the encryption key (which is usually derived from a passphrase).

Disk encryption uses **symmetric crypto** (usually hardware-accelerated AES) and in most implementations, the encryption is **length-preserving**: a single block of data results in a single block of ciphertext which is then directly stored in the corresponding physical block of the persistent storage device. For most of the software stack, this type of encryption is completely transparent. Not even the file system implementation in the operating system needs to be aware that it is stored on an encrypted device.

Storing Data in Blocks

193

- splitting data into fixed-size chunks is unnatural
- there is no permission system for individual blocks
 - this is unlike **virtual (paged) memory**
 - it'd be really inconvenient for users
- processes are not persistent, but **block storage** is

We are used to working with arbitrary-sized files, but this expectation does not map very neatly onto the block device abstraction, where data is stored in fairly big chunks. Additionally, the block layer does not offer a permission system or any kind of sharing or multiplexing: there are just blocks and a process can either read or write any of them, or none at all.

While main memory is also a rather dumb array of bytes, organized into pages (which are, in fact, rather block-like), the content of this main memory is much more dynamic and much less important to the user. Part of the distinction comes down to **persistence**, but other than that, it is really mostly just a historical accident.

Filesystem as Resource Sharing

194

- usually only 1 or few disks per computer
- many programs want to store **persistent** data
- file system **allocates space** for the data
 - which blocks belong to which file
- different programs can write to different files
 - no risk of trying to **use the same block**

Nonetheless, there needs to be **some** mechanism for sharing persistent storage among different processes (and different users). This mechanism is, of course, the **file system**. Among other things, it manages allocation of free space into files, and maintains persistent identities of such files.

Filesystem as Abstraction

195

- allows the data to be **organised** into files
- enables the user to **manage** and review data
- files have arbitrary & **dynamic size**
 - blocks are **transparently** allocated & recycled
- **structured** data instead of a flat block array

Block storage is not very convenient, and in almost all cases requires additional abstractions. We can draw an analogy with main memory: the flat array of bytes exposed by the CPU is not what programmers directly use: instead, this memory is managed by a memory allocator like **malloc**, which slices it up into logical objects of varying size which can be created and destroyed as needed.

The file system plays the same role for persistent storage. However, since the file system is **shared** among many processes, and even directly exposed to users, it needs to be organized more rigidly and intuitively. Instead of storing numeric pointers at arbitrary locations within files (as is the case with in-memory data structures), objects within a file system are strictly separated into objects which carry pointers (structure), i.e. directories, and data-carrying objects, i.e. regular files. Since each pointer in a directory is given a name, this organizational principle gives rise to the directory hierarchy as was discussed in lecture 2.

Part 4.3: Virtual Filesystem Switch

A typical kernel must be able to accommodate a fairly large number of different file system implementations: the native file system of the operating system, of course, but at minimum also file systems that are typically used on portable media, such as USB flash drives (**FAT** or **exFAT**) or optical media (ISO 9660, UDF). It is also often desirable that non-native file systems can be mounted, such as accessing **NFS** volumes on Unix systems. Finally, there are file systems which are ported to a particular operating system because of their desirable properties, such as high performance, reliability or scalability and not for compatibility reasons alone (e.g. the SGI **xfs** ported from IRIX to Linux, or Sun/Oracle **ZFS** ported from Solaris to FreeBSD).

Additionally, a number of operating systems employ various virtual file systems (such as **proc** on many UNIX-like systems, or **sys** on Linux), which also present a unified interface with the other file systems.

Virtual File System Layer

197

- many different filesystems
- the OS wants to treat them **all alike**
- VFS provides an internal, **in-kernel API**
- filesystem **syscalls** are hooked up to VFS

Since all the different file systems provide essentially the same semantics, and need to be available via a single unified API to the outside world, the kernel would ideally also treat them all alike. This is the role of the virtual file system switch or virtual file system layer.

VFS in OOP terms

198

- VFS provides an abstract class, **filesystem**
- each filesystem implementation derives **filesystem**
 - e.g. `class iso9660 : public filesystem`
- each actual file system gets an instance
 - `/home, /usr, /mnt/usbflash` each one
 - the kernel uses the abstract interface to talk to them

If you have encountered object-oriented programming, this paradigm should be quite familiar: there is an **interface** which describes a file system, from the point of view of the kernel, in the abstract. Each specific file system implements this abstract interface, but from the point of view of the API, it doesn't really matter which file system it is, since they all provide a uniform API. Unlike the block layer, this abstraction is typically not visible from outside of the kernel. Nonetheless, it is still an important abstraction.

The filesystem Class

199

```
struct handle { /* ... */ };
struct filesystem
{
    virtual int open( const char *path ) = 0;
    virtual int read( handle file, ... ) = 0;
    /* ... */
}
```

If we consider the VFS interface as a C++ class, this is approximately how it would look like. In particular, the handle is a possibly complex data structure: the file descriptor abstraction exists between the kernel and the user-space. Within kernel, it is much less useful, since it is, among other things, process-specific. When a file system implementation needs to work with a file, what it needs is a reference to the particular i-node which represents the file (or rather to its in-memory version), and perhaps an iterator into the file: ideally one that is more efficient than a mere offset, i.e. a data structure which can, without scanning through block lists in the i-node, pinpoint specific block (or blocks) which need to be fetched from disk.

Filesystem-Specific Operations

200

- **open**: look up the file for access
- **read, write** – self-explanatory
- **seek**: move the read/write pointer
- **sync**: flush data to disk
- **mmap**: memory-mapped IO
- **select**: IO readiness notification

The VFS operations partially reflect the user-level file access API, with the above-mentioned caveats. The above list is an abridged version of the VFS interface as implemented in the Linux kernel. Since VFS is internal to the kernel, it is not standardized in any way, and different kernels approach the problem differently.

Standard IO

201

- the **usual** way to use files
- open the file
 - operations to read and write **bytes**
- data has to be **buffered** in user space
 - and then **copied** to/from kernel space
- not very efficient

The standard API for input and output is based around the **read** and **write** operations. Unfortunately, these are not always efficient, since the data must be copied between the file system cache and the (private) memory of the process requesting the operation. Efficiency of read-heavy workloads can be significantly improved by using memory-mapped IO instead. This is the case whether the program needs random access to the content of a file (i.e. seeks from place to place) or whether it simply wants to process a single byte at a time.

Memory-mapped IO

202

- uses **virtual memory** (cf. last lecture)
- treat a file **as if** it was swap space
- the file is **mapped** into process memory
 - **page faults** indicate that data needs to be read
 - **dirty** pages cause writes
- available as the **mmap** system call

Memory-mapped IO uses the virtual memory subsystem to reduce the amount of copying and context switching involved in IO operations. Under this scheme, file data is (at least initially) **shared** between the block access cache and the process.

Modification of data can be handled in two ways:

1. modifications are **private**, e.g. when the process simply needs to adjust the data as part of processing, but does not wish to write the changes into the original file,
2. the changes are **shared**, i.e. the process intends to make changes to the data and write that data into the file afterwards.

In the first case, the mapping is done in a copy-on-write regime: the pages from cache are marked read-only for the process and write attempts cause the operating system to make a new copy of the data in physical memory. Future reads and writes are then redirected into this new copy. In the second case, the pages are write-through – modifications by the process affect both the cache and, eventually, also the on-disk file (when the dirty pages are flushed to disk).

Sync-ing Data

203

- recall that the disk is very **slow**
- waiting for each write to hit disk is inefficient
- but if data is held in RAM, what if **power is cut**?
 - the **sync** operation ensures the data has hit disk
 - often used in database implementations

A write operation on a file is, usually, **asynchronous** with regards to the physical medium. That is, control returns from a **write** system call long before the data is durably stored. This means that in case of a crash or a power loss, an application may be unable to recover data that it has already written. If it is important that a write is complete before computation continues, operating systems offer a **sync** system call (or a variation, like **fsync** and **fdatasync**) which ensures that all outstanding writes are sent to the device.

Filesystem-Agnostic Operations

204

- handling **executables**
- **fcntl** handling
- special files
- management of **file descriptors**
- file **locks**

Within the kernel, part of file-system-related functionality is independent of a particular file system implementation, and is instead implemented in other kernel modules. One example would be the code which deals with executable files: let's assume that the file system implements memory mapping of files (which itself is partially implemented in the block layer and virtual memory subsystems). When a program (stored in a regular file on some file system) is executed, the responsible module of the operating system will request a memory mapping of the file through VFS, but the remainder of the code dealing with the **exec** system call does not need any file-system-specific

knowledge.

Executables

205

- **memory mapped** (like `mmap`)
- may be paged in **lazily**
- executables must be **immutable while running**
- but can be still unlinked from the directory

There are additional provisions regarding executable files. One of them is, that they are often „paged in“ lazily: this is a concept that we will explore in more detail in lecture 5. This is, however, a compromise: it means that if a given executable is ‘running’ – that is, if a process exists which is currently executing the program stored in said executable – the operating system must disallow any writes to the file. Otherwise, the image, which may partially reside in memory and partially on disk, may be corrupted as pages that are already in memory are combined with pages that are loaded lazily from disk.

However, since the **name** of the file is external to the file itself, it is not a problem for running executables to be **unlinked**. When the last process which uses the executable terminates, the reference count on the i-node drops to zero and the i-node is reclaimed (including any data blocks it was using).

File Locking

206

- multiple programs writing the same file is bad
 - operations will come in **randomly**
 - the resulting file will be a mess
- file locks fix this problem
 - multiple APIs: `fcntl` vs `flock`
 - differences on networked filesystems

The kernel offers a mechanism for **locking** files, making it possible for multiple processes to read and write a shared file safely (i.e. without corrupting it by issuing conflicting `write` calls concurrently). In POSIX, there are historically two separate mechanisms for locking files: the `flock` call, which locks the entire file at once, and the `fcntl` call, which can lock a range of bytes within a file.

There are, unfortunately, subtle problems with these locking APIs which makes them easy to misuse, related both to the `close` system call and to network file systems. We will not go into details, but if you intend to use file locking, it is a good idea to do some research on this.

The `fcntl` Syscall

207

- mostly operations relating to **file descriptors**
 - **synchronous** vs **asynchronous** access
 - blocking vs non-blocking
 - close on exec: more on this in a later lecture
- one of the several **locking** APIs

This is another bit of the file system interface that does not normally need to interact with the VFS or with a specific file system implementation. In addition to exposing an API for locking files, the `fcntl` system call mainly interacts with the file descriptor table and various file descriptor flags: for instance, it can be used to enable synchronous IO, meaning that every `write` call will only return to the caller after the data has been sent to the storage device.

Likewise, `fcntl` can be used to set blocking (default) or non-blocking mode on a file descriptor: in non-blocking mode, if a write buffer is full, the system call will not wait for space to become available: instead, it will indicate to the program that the operation cannot be completed

and that it should be retried at a later point. Likewise, if there is no data available (e.g. in a pipe or in a socket), a `read` operation will return immediately indicating that this is the case, instead of waiting for more data to become available (as it would in blocking mode).

The `close-on-exec` flag instructs the kernel to internally perform a `close` operation on this file descriptor in the event that an `exec` call is performed while the file descriptor is still open.

Special Files

208

- device nodes, pipes, sockets, ...
- only **metadata** for special files lives on disk
 - this includes **permissions** & ownership
 - type and **properties** of the special file
- they are just different kind of an **i-node**
- **open**, **read**, **write**, etc. bypass the filesystem

The special files that we have considered in lecture 2 are, likewise, mostly independent of the particular file system. The mechanics of reading and writing such files are not tied to any particular file system. The variants of those file-like objects which are linked into the file system hierarchy (named pipes, UNIX domain sockets, devices) are, for the purpose of the file system, simply a special type of i-node, and the file system only needs to store their metadata. When such a file is opened, the file system only obtains the metadata and hands off to a different part of the kernel.

Mount Points

209

- recall that there is only a **single directory tree**
- but there are **multiple disks** and filesystems
- file systems can be **joined** at directories
- **root** of one becomes a **subdirectory** of another

Finally, mount points are another VFS-layer feature that does not extend into individual file system implementations. In fact, arguably, the main reason for VFS to exist is that multiple different on-disk file systems can be seamlessly integrated into a single tree. The system calls which perform the joining (and un-joining, as it were) are `mount` and `unmount`, respectively.

Part 4.4: The UNIX Filesystem

In this section, we will describe, in relatively low-level terms, how the traditional UNIX file system is organized, including how things are stored on disk. We will also discuss some of the problems which are tied to organisation of data on a medium with severe penalties for non-sequential data access, and with rather inflexible read and write operations (i.e. the problems that arise because the device only provides whole-block operations).

Superblock

211

- holds **toplevel** information about the filesystem
- locations of i-node tables
- locations of i-node and **free space** bitmaps
- **block size**, filesystem size

There is considerable amount of metadata that the file system must store on disk. It is usually impractical to allocate fixed-address regions for all this metadata, hence the locations of these **metadata blocks** must be stored somewhere in the file system itself. Of course, at least some metadata must have a fixed, well-known location, to bootstrap the in-

memory data structures at `mount` time. This fixed part of the metadata is known as the **superblock**. Since it is quite essential, and changes only rarely, it is usually stored in multiple fixed locations throughout the disk, in case the primary copy is damaged.

I-Nodes

212

- recall that i-node is an **anonymous file**
 - or a directory, or a special
- i-nodes only have **numbers**
- **directories** tie names to i-nodes

Until now, we have treated i-nodes as an abstract concept: they simply represent an object in the file system, be it a regular file, a directory, or some kind of special file. In traditional UNIX file systems, the i-node is also an actual, physical data structure stored on disk. Since the size of the i-node is fixed, they can be stored in large arrays and indexed using numbers: these indices form the basis of the i-node numbering system.

Since i-node tables need to be consulted quite often, and the latency of random access to a traditional hard drive depends on the physical distance of the different pieces of data, the i-node array is often split into multiple chunks, each stored in a different physical location. The system then attempts to keep both references to the i-node, and data that the i-node refers to, close together.

I-Node Allocation

213

- often a **fixed number** of i-nodes
- i-nodes are either **used or free**
- free i-nodes may be stored in a **bitmap**
- alternatives: B-trees

In case i-nodes are stored in an array, this array usually has a fixed size, which means some of the entries are unused. Since creating (and erasing) files are fairly common operations, the system must be able to quickly locate an unused i-node, and also quickly mark a previously used i-node as unused. Scanning the entire table of i-nodes to find an unused one would be very slow, since only a few i-nodes fit into a single disk sector.

A common method of speeding this up is through bitmaps: in addition to i-nodes themselves, the file system stores an array of bits, one bit per i-node. This way, even though the system still has to do a linear scan for a free i-node in the bitmap, it will do so at a rate of 512 or 4096 i-nodes per sector.

Consider this example (the sector size is 512 bytes): a file system is organized into groups, each with around 200MiB of data, and each equipped with an array of 26000 i-nodes. This works out to around 8KiB of data per i-node, which is the average expected size of a file in this file system (if it was less than that, the file system might run out of i-nodes before it runs out of data space). Each i-node is 128 bytes, hence 4 i-nodes are stored per sector and the entire i-node array thus takes up 6500 sectors (or 3.25MiB). The i-node bitmap is, on the other hand, 51 sectors (or about 25KiB).

I-Node Content

214

- exact content of an i-node depends on its type
- regular file i-nodes contain a list of **data blocks**
 - both direct and indirect (via a data block)
- symbolic **links** contain the target **path**
- special devices **describe** what **device** they represent

The arguably most important part of an i-node is the **data pointers** that it contains: i.e. the addresses of data blocks that hold the actual data (whether the bytes of a regular file, or the data structure which maps names to i-node numbers in case of directories).

Attaching Data to I-Nodes

215

- a few **direct** block addresses in the i-node
 - eg. 10 refs, 4K blocks, max. 40 kilobytes
- **indirect** data blocks
 - a block full of addresses of other blocks
 - one indirect block approx. 2 MiB of data
- **extents**: a contiguous range of blocks

A traditional approach lists each data block separately. A common optimization makes the observation, that most files are stored in a small number of contiguous spans of blocks. Those spans are called **extents**, and more modern file systems store, instead of a simple array of block addresses, a list of extents (i.e. in addition to a block address, they also keep a number that tells the file system how many consecutive blocks are taken up by the file system). This is, essentially, a form of run-length encoding. The obvious downside is that finding the address of the block which contains any given offset is now a linear operation (instead of constant-time), but in a data structure that is usually much smaller (most files will only have a single-digit number of extents, as opposed to perhaps hundreds of individual data blocks).

Fragmentation

216

- **internal** – not all blocks are fully used
 - files are of variable size, blocks are fixed
 - a 4100 byte file needs 2 blocks of 4 KiB each
 - this leads to **waste** of disk **space**
- **external** – free space is non-contiguous
 - happens when many files try to grow at once
 - this means new **files are also fragmented**

Every time structured data is stored in an unstructured array of bytes, certain trade-offs come up. One of them has to do with storage efficiency: packing data more tightly often makes many operations slower, and the required metadata more complicated.

One of the well-known problems of this type is **fragmentation**, of which there are two basic types. Internal fragmentation is caused by alignment: it is much more efficient to start each file at a block boundary, and hence to allocate a whole number of blocks to each file. But because files have arbitrary sizes, there is often some unused space at the end of each file. This space is overhead – it does not store any useful data. Doing things this way, though, makes file access faster. In other words, at the end of most files, there is a small fragment of disk space that cannot be used (because it is smaller than the minimum size that can be allocated for a file, i.e. one block).

External Fragmentation Problems

217

- **performance**: can't use fast sequential IO
 - programs often read files sequentially
 - fragmentation → random IO on the device
- metadata size: can't use long extents

External fragmentation, on the other hand, does not directly waste space. In this case, as files are created and destroyed, the free space is eventually distributed across the entire file system, instead of being concentrated in a single contiguous chunk.

When creating new files, finding space for them is more work, since it has to be 'glued' from many smaller fragments. Metadata grows bigger, since average extent length goes down and hence files need more extents.

Finally, and perhaps most importantly, when free space is fragmented, so are the files that eventually make use of it. Access to these files is then less efficient, because every time there is a discontinuity in the data, the system experiences additional latency, due to non-sequential access to the underlying hard drive

Directories

218

- uses **data blocks** (like regular files)
- but the blocks hold **name → i-node maps**
- modern file systems use **hashes** or **trees**
- the format of directory data is **filesystem-specific**

Like regular files, directories use data blocks to store their content. However, while the content of regular files is completely arbitrary (i.e. user- or application-defined), directories are interpreted by the file system itself. The simplest format one could adopt to store directories is to simply concatenate all the (null-terminated name, i-node number) tuples after each other. Of course, this would be very inefficient. Most file systems use a more sophisticated data structure (an on-disk hash table, or a balanced tree).

File Name Lookup

219

- we often need to find a file based on a path
- each component means a directory search
- directories can have many thousands entries

One of the most common operations in a file system is that of **file name lookup**. This operation is performed many times for each path that needs to be resolved (once for each component of the path). It is therefore very important that this can be done quickly.

While it is important that the on-disk format allows for fast lookups, even the fastest on-disk structure would be too slow: almost all operating systems employ an in-memory cache to speed up 'common' lookups (i.e. they keep a cache of directories or directory entries which have been recently used in lookups).

Old-Style Directories

220

- unsorted sequential list of entries
- new entries are simply appended at the end
- unlinking can create holes
- lookup in large directories is very inefficient

As we have mentioned earlier, the simplest (but very inefficient) method of storing directories is to simply keep a linear list of directory entries (i.e. tuples which map a name to an i-node number). This causes a whole lot of problems and no serious contemporary file system uses this approach.

Hash-Based Directories

221

- only need one block read on **average**
- often the most efficient option
- **extendible** hashing
 - directories can grow over time
 - gradually allocates more blocks

A common alternative is to use hash tables, which usually make name lookup very fast: the expected outcome is that, based on the hash of the name, the first sector that is fetched from the disk will contain the requested name/i-node pair, even for moderately large directories.

Of course, there are downsides, too: traversing the directory in, say, alphabetical order, will cause a random-access pattern in the underlying disk IO, since the directory is sorted on the hash, which is essentially random. Additionally, pathological directories can cause very bad behaviour, in case all (or most) directory entries hash to the same bucket.

Tree-Based Directories

222

- self-balancing search trees
- optimised for block-level access
- **B trees**, B+ trees, B* trees
- logarithmic number of reads
 - this is worst case, unlike hashing

Another fairly common strategy is to use balanced trees, which have slightly worse average performance, but much better worst-case guarantees, compared to hash tables. Additionally, the entries are stored in sorted order, making certain access patterns more efficient.

Hard Links

223

- **multiple names** can refer to the **same i-node**
 - names are given by **directory entries**
 - we call such multiple-named files **hard links**
 - it's usually forbidden to hard-link directories
- hard links cannot cross device boundaries
 - i-node numbers are only unique within a filesystem

An immediate consequence of how files and directories are stored in the file system is the existence of **hard links**. Please note that those are **not** special entities: they are merely a name for a situation where multiple directory entries refer to the same i-node. In this case, each of the 'hard links' is indistinguishable from all the others, and the same file simply appears in multiple places in the directory hierarchy. Since i-nodes keep a reference count, it is usually possible to tell that the file is available under multiple different paths. Since files are only destroyed when their reference count drops to zero, removing a file from a directory (called **unlinking**) may or may not cause the file to be actually erased.

Soft Links (Symlinks)

224

- they exist to lift the one-device limitation
- soft links to directories are allowed
 - this can cause **loops** in the filesystem
- the soft link i-node contains a **path**
 - the meaning can change when paths change
- **dangling** link: points to a non-existent path

Sometimes, it is useful to refer to a file not by i-node number, but by a path. This can be done by employing **soft links**: unlike hard links, those are actual objects, stored in the file system as a special type of i-node. When such a file is opened, the operating system will instead look up a file by the path stored in the soft link. Of course, this leads to additional problems: for instance, the target path may not exist (this is by far the most common failure mode).

If it does, an i-node of the file corresponding to the path is obtained the usual way. Unlike in standard directory entries, this new i-node may reside on a different file system.

Free Space

225

- similar problem to i-node allocation
 - but regards data blocks
- goal: **quickly** locate data blocks to use
 - also: keep data of a single file **close together**
 - also: **minimise** external **fragmentation**
- usually bitmaps or B-trees

Like i-nodes, the file system needs to be able to quickly locate an empty data block, either when files and directories are created, or when existing ones are extended. The task is slightly more complex for data blocks: for i-nodes, it is sufficient to allocate a single i-node, and placement of i-nodes relative to each other is not very important (though of course it is useful to keep related i-nodes close together).

However, many files require more than a single data block, and it is rather important that those blocks reside next to each other, if possible. To make matters worse, it is not always clear how big the file is going to be, and the file system should probably reserve some additional blocks on top of those required for the initial set of writes. Armed with a size estimate, the system then needs to find free space of that size, ideally without impinging on headroom for existing files, and ideally as a single contiguous run of blocks. Of course this may not be possible, in which case it will try to split the file into multiple chunks of free space.

Compared to the i-node case, the on-disk data structures are, however, pretty much the same: either free block bitmaps, or balanced search trees. Most of the difference is in the inputs and algorithms.

File System Consistency

226

- what happens if **power is cut**?
- data buffered in RAM is **lost**
- the IO scheduler can **re-order** disk writes
- the file system can become **corrupt**

File systems, and the data structures they employ, face a somewhat unique challenge: the changes the file system makes to its metadata can be cut off at an arbitrary point, even in the middle of an operation, most often by a power loss. Even worse, the individual writes the system has issued can be re-ordered (to improve performance), which means that there isn't a sharp cut-off point.

As an example, consider creation of a regular file: an i-node must be allocated (this means a write to the i-node bitmap), filled in (write to the i-node itself) and linked into a directory (write, or multiple writes, into the data structure of the directory). If operations are performed in this sequence and power is cut off, two things could happen:

1. the bitmap is updated, but the remaining operations are lost: in this case, the i-node is lost, unless a consistency check is performed and discovers that an unused i-node is marked as allocated in the bitmap,
2. the bitmap and the i-node itself are updated, but it is not linked to

the directory hierarchy, giving essentially the same outcome, but in a form that is harder to detect during a consistency check.

In both cases, some resources are lost, which isn't good, but isn't terrible either. However, consider the case when the writes are re-ordered, and the directory update comes first. In this case, the file system has a directory entry that points to an uninitialized i-node, making a garbage file accessible to users. Depending on the content of the uninitialized i-node, bad things could happen if the file is accessed.

Fortunately, the file system can impose a partial order on the writes (i.e. guarantee that all outstanding writes are finished before it makes further changes). Using this mechanism carefully, the amount of damage to data structures can be limited, without significantly impacting performance.

The last line of defence is a **dirty flag** in the superblock: when a file system is mounted, the dirty flag is written to disk, and when it is unmounted, after all outstanding changes are written, it is cleared. The file system will refuse to mount if the dirty flag is already set, enforcing a consistency check.

Journalling

227

- also known as an **intent log**
- write down what was going to happen **synchronously**
- fix the actual metadata based on the journal
- has a **performance penalty** at run-time
 - **reduces downtime** due to faster consistency checks
 - may also **prevent data loss**

A particular technique (that relies on imposing a partial order on writes) is so-called intent log, perhaps better known from relational database systems. In this approach, the file system maintains a dedicated area on disk and before commencing any non-atomic data structure updates, writes the description of the operation (atomically) into the intent log. In this case, if the composite operation is later interrupted, the log captures the **intent** and can be used to undo the effects of the incomplete operation quickly. This means that in most scenarios, a full consistency check (which is usually quite expensive) is not required.

Part 4.5: Advanced Features

This section briefly introduces some additional concepts that often appear in the context of file systems. We won't have time to delve into too many details in this course, but it is important to be aware of those features.

What Else Can Filesystems Do?

229

- transparent file compression
- file encryption
- block de-duplication
- snapshots
- checksums
- redundant storage

There are three rough categories of features in the above list: storage efficiency (compression, de-duplication), security (encryption) and reliability (checksums, redundant storage). The ability to make snapshots falls somewhere between efficiency and reliability, depending on the use case and implementation.

File Compression

230

- use one of the standard compression algorithms
 - must be fairly **general-purpose** (i.e. **not** JPEG)
 - and of course **lossless**
 - e.g. **LZ77**, **LZW**, Huffman Coding, ...
- quite **challenging to implement**
 - the length of the file changes (unpredictably)
 - efficient **random access** inside the file

Of course, it's always possible to compress files at the application level, by simply using an appropriate compression program. However, the user then needs to manually decompress the file before using it and then re-compress it afterwards again. This is quite inconvenient. Of course, this could be automated, and some programs can do the (de)compression automatically when they open a file, though this is neither very common, nor very efficient.

An alternative, then, is for the file system to implement transparent file compression, that is, compress the data when it is stored on disk, but present user-space programs with uncompressed data when reading, and transparently compress newly written data before storing it.

This is not without challenges, of course. The biggest problems stem from the fact that data is never uniformly compressible, and hence, different sections of a file will compress at different ratios. This means that seeking to a specific **uncompressed** offset will be hard to implement. Likewise, writes in the middle of a file (which normally preserve file length) will cause the file to shrink or lengthen, making the operation much more complicated.

File Encryption

231

- use **symmetric** encryption for individual files
 - must be **transparent** to upper layers (applications)
 - symmetric crypto is length-preserving
 - **encrypted directories**, inheritance, &c.
- a new set of challenges
 - **key** and passphrase **management**

Unlike compression, most encryption algorithms are length-preserving, making the whole affair much simpler in some sense. Nonetheless, there is an important matter of dealing with secrets which makes the code complicated in a different way. Additionally, unlike with compression, depending on the use case, the system might have to encrypt metadata as well (i.e. not just file content). This would, most importantly, cover directories: not only file names, but also the overall directory structure.

Additionally, a failure to compress a file that the user intended to be compressed is not a big problem. With encryption, such a mistake would be quite fatal.

Since block-level encryption is considerably simpler, and hence less likely to contain fatal flaws, most modern systems use it instead of file-system-level encryption as described here.

Block De-duplication

232

- sometimes the same **data block** appears **many times**
 - virtual machine images are a common example
 - also **containers** and so on
- some file systems will identify those cases
 - internally point many files to the **same block**
 - **copy on write** to preserve illusion of separate files

There are numerous use-cases where either entire files or fragments of files are stored multiple times on a given file system. In those cases, locating duplicated blocks can lead to significant space savings. In modern file systems, it is usually not a problem to make the same data block part of multiple files. Like with other copy-on-write implementations, such shared blocks must be specifically marked as such, so that any writes that would affect them can un-share them (i.e. make a private copy).

De-duplication is somewhat expensive, since it is not easy to find identical blocks: a naive scanning comparison would take $O(n^2)$ time to find duplicated blocks (even though it only uses constant amount of memory). Of course, that is impractical, considering how $n = 10^9$ is only about 4TB of storage and $n^2 = 10^{18}$ is a **very** large number. Hash tables can make the operation essentially linear, though it requires on the order of 4GiB of RAM per 1TB of storage. Probabilistic algorithms and data structures can further reduce the constant factors on both time and memory.

In most cases, de-duplication is an offline process, i.e. one that is scheduled to run at some intervals, preferably during light load, since it is too resource-intensive to be performed continuously with each write (that is, online).

Snapshots

233

- it is convenient to be able to **copy** entire filesystems
 - but this is also **expensive**
 - snapshots provide an **efficient** means for this
- snapshot is a **frozen image** of the filesystem
 - cheap, because snapshots share storage
 - easier than de-duplication
 - again implemented as **copy-on-write**

If file system metadata is organized in a suitable data structure (usually a variation of B trees), it is possible to implement copy-on-write not just for data blocks, but also for this metadata. In that case, it is not very hard to provide efficient **snapshots**.

Taking a snapshot is, semantically, equivalent to making a copy of the entire file system **while it is not mounted**: that is, the copy is made atomically with regards to any concurrent writes. In yet other words, if a program overwrite two files, say A to A' and later it overwrites B to B', if a copy was taken the usual (non-atomic) way, in the copy, it could easily be the case that files A and B' are present (it is an easy exercise to work out how this could happen).

The other major advantage of a snapshot is that initially (i.e. when it does not differ very much from the live file system) it takes up very little space. Of course, as the live file system begins to diverge, more and more data blocks need to be copied upon modification, increasing the storage demands, in the worst case approaching the space requirements of a standard, full copy. Nonetheless, the time cost associated with this are amortized over the life of the snapshot. Finally, in case (read-only) snapshots are made regularly, there are permanent savings to be had from copy-on-write, since in those cases, at least some of the data will be shared between the snapshots themselves.

Checksums

234

- hardware is **unreliable**
 - individual bytes or sectors may get **corrupted**
 - this may happen without the hardware noticing
- **checksums** may be stored along with metadata
 - and possibly also **file content**
 - this protects the integrity of the filesystem
- beware: **not** cryptographically secure

Unfortunately, storage devices are not 100% reliable, and can sometimes quietly corrupt data. That is, reading a block may yield different data than what was written to that block previously. A file system may fight this by storing checksums (e.g. CRC) along with metadata (or even along with data). This will not prevent corruption as such, but at least allows the file system to detect it early. When detected, the corrupt objects can be restored from backup: since making a backup involves reading the file system, under this scheme, such corruption should be always detected before a good copy of the affected data is lost.

Redundant Storage

235

- like filesystem-level RAID
- data and metadata blocks are **replicated**
 - may be between multiple local block devices
 - but also across a **cluster** / many computers
- drastically improves **fault tolerance**

Finally, file systems (especially distributed file systems) may employ redundant storage for both data and metadata. Essentially this means that each data block and each metadata object is stored in multiple copies which are all kept synchronized by the file system, each copy stored on a different physical storage device. In some cases, multiple computers may be involved, improving resilience in face of failures in non-disk components (which usually knock out the entire computer).

Review Questions

236

- What is a block device?
- What is an IO scheduler?
- What does memory-mapped IO mean?
- What is an i-node?

Part 5: Processes, Threads & Scheduling

In this lecture, we will look at the 2 basic resources that the computer offers, and which every program needs: CPU and memory. The main question then will be how the operating system achieves the illusion that every thread seemingly has its own processor and every process has its own memory (this is what we mean by multiplexing – turning a single physical resource into a larger set of virtual instances of the same).

Lecture Overview

238

1. processes and virtual memory
2. thread scheduling
3. interrupts and clocks

There will be 3 parts. We will first look at virtual memory, and the unit of memory isolation in the operating system: a process. We will then look at CPU (processor) sharing and the unit of CPU allocation, a **thread**. Finally, we will look in more detail how CPU sharing is implemented in terms of hardware.

History: Time Sharing

241

- “mini” computers could run programs **interactively**
- teletype **terminals**, screens, keyboards
- **multiple users** at the same time
- hence, **multiple programs** at the same time

Computers were quite expensive at the time, and programs started to offer more interactivity. That is, a program could interact with the operator by asking questions and then waiting for inputs to be provided, for instance. Together, those two considerations made the sequential, one program at a time paradigm quite impractical.

This is the origin of *time sharing* systems, where a computer could execute multiple programs at seemingly the same time, by quickly switching from one program to another. As soon as this is possible, it makes sense to allow multiple users to interact with the same computer (each user through a different program).

Processes: Early View

242

- process is an **executing** program
- there can be **multiple processes**
- various **resources** belong to a process
- each process belongs to a particular **user**

Since traditional programs were internally sequential, the original notion of a process encompassed both types of basic resources: memory and processor. In normal operation, a process P gets to execute on the processor for some amount of time, before it is interrupted and the system switches to executing some other process. At some later point, process P will be awoken and continue execution where it left off.

Part 5.1: Processes and Virtual Memory

Prehistory: Batch Systems

240

- first computers ran **one program** at a time
- programs were scheduled **ahead of time**
- we are talking punch cards &c.
- and computers that took an entire room

The first generation of computers was strictly sequential: one program could execute at any given time, and that was that. To execute another program, the first must have finished first.

Process Resources

243

- **memory** (address space)
- **processor** time
- open files (descriptors)
 - also working directory
 - also network connections

The most basic resource associated with a process is **memory**: this is where the program itself (the instructions) are stored, and also where its data (both static and dynamic) resides. This memory is typically private: one process cannot see the memory of another process.

Process Memory Segments

244

- program **text**: contains instructions
- data: static and dynamic **data**
 - with a separate read-only section
- **stack** memory: execution stack
 - return addresses
 - automatic variables

The memory of a program is organized into functionally distinct **segments**. Traditionally, those segments were contiguous chunks of address space, but in modern operating systems, this is not quite true (and the whole idea of a segment is losing its appeal). Nonetheless, the traditional split is into program **text**, which contains the instructions of the program (i.e. the executable code, typically produced by a compiler), a **stack** which holds the C stack (i.e. the values of local variables and return addresses, along with other bookkeeping) and finally a **data segment** which holds, unsurprisingly, data.

In a modern system, none of those segments needs to be contiguous: the **text** consists of multiple mappings: one for the main program, and one for each shared library it uses. Those mappings do not need to be adjacent in the virtual address space (much less so in physical memory). Likewise, since one process can contain multiple threads (more on that later), there may be multiple stacks, again, not necessarily allocated next to each other. Finally, static (and especially read-only) data comes from executable images too, hence there might be one mapping per executable, just like with text. The dynamic data, then, is entirely unconstrained in its shape: the implementation of **malloc** in **libc** will request memory from the operating system as needed, with virtual addresses often being assigned randomly to each new region of memory.

Process Memory

245

- each process has its own **address space**
- this means processes are **isolated** from each other
- requires that the CPU has an MMU
- implemented via **paging** (page tables)

The defining characteristic of a modern process is its **address space**: the virtual addresses allocated to a process is private to that process and as such invisible to any other process. The data is mostly stored in RAM, but each process only (typically) sees a small portion of the physical memory – the majority is simply not visible in its virtual address space. Behind the scenes, this separation of address spaces is built on paging, which is, in turn, provided by the MMU (memory management unit) of the CPU.

Process Switching

246

- switching processes means switching **page tables**
- physical addresses do **not** change
- but the **mapping** of virtual addresses does
- large part of physical memory is **not mapped**
 - could be completely unallocated (unused)
 - or belong to **other processes**

In a typical time-sharing operating system (this covers pretty much every modern general-purpose OS), an illusion of virtually unlimited concurrency (the ability to run an arbitrary number of processes at the same time, regardless of the number of available CPU cores) is achieved by rapidly switching the execution from one process to the next. Or, to be more precise, execution is switched from one **thread** to another (which we will discuss shortly), but often, the threads will be part of different processes. In those cases, the operating system must instruct the processor to switch into the virtual address space of the new process.

Of course, in this process, the physical memory is not rearranged in any way – that would be way too expensive. Instead, the MMU is instructed to start using a different set of mapping rules (page tables) that map virtual addresses to physical addresses. Usually, the overlap between physical addresses that have been visible in the old virtual address space and those visible in the new one is minimal: as outlined earlier, the virtual address space is almost entirely private to each process.

Thus, most of the physical memory is, from the point of view of any given process, inaccessible: it is either unused, or it belongs to a different process.

Paging and TLB

247

- address translation is **slow**
- recently-used pages are stored in a TLB
 - short for Translation Look-aside Buffer
 - very fast **hardware** cache
- the TLB needs to be **flushed** on process switch
 - this is fairly **expensive** (microseconds)

An important consideration is the **cost** associated with switching processes. On the surface, this involves a thread switch (which, as we will see later, consists of storing the content of user-accessible registers in memory and loading a new set of values from a different location in memory) and an additional register write (to load the new page table into the MMU).

Unfortunately, under the surface, that single register write causes a cascade of other effects: in particular, all modern processors use a very fast **cache** for storing recent address translations (i.e. which physical addresses correspond to a small set of recently-used virtual addresses), known as the TLB (translation look-aside buffer).

The TLB is required because otherwise, translating a virtual address to a physical one involves multiple round-trips into the main memory, where page tables are stored. In modern computers, each of those round-trips will take hundreds or even thousands of processor cycles – clearly, repeating this process on every address translation would make computation extremely slow.

Nonetheless, upon a process switch, the mapping of virtual addresses to physical addresses changes, and the information stored in the TLB becomes invalid, since it refers to the previous address space. The simplest remedy is simply erasing everything that is stored in the TLB. This is, in itself, a quick operation, since the TLB is implemented using very fast on-die memory. Most of the price comes from the subsequent

address translations, which cannot be answered from the (now empty) TLB and must perform multiple round-trips into main memory (or, depending on cache pressure, into one of the general-purpose caches – L1, L2 or L3).

Incidentally, modern processors often use tagged TLBs, which make the entire process more efficient, but this is well outside of the scope of this course.

Threads

248

- the modern unit of CPU scheduling
- each thread runs sequentially
- one process can have **multiple threads**
 - such threads share a single address space

While processes are the basic unit of memory management in the operating system, computation is captured by **threads**. Each process has at least one thread, but might have more than one. While processor time **accounting** is usually done at process level, **execution** is tied to threads, since threads is what the processor ultimately executes.

What is a Thread?

249

- thread is a **sequence** of instructions
 - instructions depend on results of previous instructions
- different threads run different instructions
 - as opposed to SIMD or many-core units (GPUs)
- each thread has its own **stack**

A thread can be thought of as an instruction stream: essentially what we imagine a traditional, sequential program to be. Within a thread, an instruction can freely use results of previous instructions (data dependencies) and make decisions (based on those results) as to which instruction to execute next (branching, or control flow). This is how a processor core executes a program (at least from the point of view of the programmer). Hence at any given time, each processor core can execute a single thread. The threads running concurrently on different cores or different CPUs can be completely unrelated (they don't even need to belong to the same process: each core has its own MMU and which can each use a different page table, and hence, each core can be executing threads in different address spaces).

Of course, this also means that each thread needs its own execution stack (i.e. the memory that stores local variables in a C program, along with return addresses and other execution-related book-keeping).

Processor Time Sharing

250

- CPU time is sliced into **time shares**
- time shares (slices) are like memory **frames**
- process **computation** is like memory **pages**
- processes are allocated into **time shares**

Besides memory, the other main commodity in a computer is **computation time**. Like memory, there is a limited amount, and it needs to be distributed among multiple threads (and processes). Since different instructions take wildly different amount of time to execute, allocation of computation is based on **time** instead of instruction count. As a bonus, time is easier to measure.

Multiple CPUs

251

- execution of a thread is **sequential**
- one CPU = one **instruction sequence** at a time
- physical limits on CPU speed → **multiple cores**
- more CPU cores = more throughput

With time sharing, it is possible to execute any number of threads on a single CPU core. However, physical limits in CPU design have made it, in the last decade, much easier to build processors which can execute twice as many threads (using twice as many processor cores), instead of making them execute a given number of threads twice as fast. As long as there are always enough threads ready to execute, the overall computational throughput of the system is doubled either way. Of course, this puts strain on software, which needs to be separated into more and more threads to saturate the computational power of a modern processor.

Modern View of a Process

252

- in a modern view, process is an **address space**
- threads are the right **scheduling abstraction**
- **process** is a unit of **memory management**
- **thread** is a unit of **computation**
- old view: one process = one thread

To recap, there are two main abstractions that center about memory and computation. While historically, operating systems made the assumption that 1 process = 1 thread (and hence older textbooks about operating systems might follow the same design), this no longer makes sense in modern systems.

Instead, threads are the abstraction that covers computation itself (the sequence of instructions to be executed), while processes cover memory and most other resources.

Memory Segment Redux

253

- one (shared) **text** segment
- a shared read-write **data** segment
- a read-only **data** segment
- one **stack** for **each thread**

TODO why is this slide here?

Fork

254

- how do we create **new processes**?
- by **fork**-ing existing processes
- fork creates an **identical copy** of a process
- execution continues in both processes
 - each of them gets a different **return value**

Let's now look at processes from the perspective of programs (and users). The first thing to consider is how processes are created: on POSIX-like systems, this is almost exclusively through the use of the **fork** system call, which simply makes an identical copy of the current process. The processes are then very slightly tweaked: in one, **fork** returns a different value, and in the process tables, one is taken to be a **parent** and the other to be a **child**. The return value of **fork** tells each process which of the two it is.

Lazy Fork

255

- paging can make **fork** quite efficient
- we start by copying the **page tables**
- initially, all pages are marked **read-only**
- the processes start out **sharing** memory

Making a copy of an entire process would be a comparatively expensive exercise. However, like we have seen with file systems before, there are tricks (based on the copy-on-write implementation technique) which make it possible to make **fork** itself quite efficient. At the time of **fork**, the only thing that needs to be copied is the **page table** which is much smaller than the entire address space of the process in question.

At the same time, all pages in both copies of the page table are marked as read-only, since they are now shared by two processes, which should not have access to each other's address space. Of course, as long as they are only reading from memory, whether there are two physical copies or a single shared copy does not make a difference.

Lazy Fork: Faults

256

- the shared memory becomes **copy on write**
- **fault** when either process tries to write
 - remember the memory is marked as read-only
- the OS checks if the memory is **supposed** to be writable
 - if yes, it makes a **copy** and allows the write

As soon as either of the processes attempts to issue a memory write, this will cause a fault: the pages are marked read-only in the page tables. When this happens, the processor will invoke the fault handler (which is part of the kernel) to resolve the situation. The fault handler will then consult its data structures (which may be partially embedded in the page table itself) to distinguish actual faults (i.e. the process tried to write into truly read-only memory) from copy-on-write events.

In the latter case, the page is semantically read-write, but is marked read-only because multiple processes are using a single physical copy. At this point, the fault handler will split the mapping: it will make a new physical copy of the data (i.e. it will allocate a new frame for it) and adjust the page table of the process which attempted to write, so that the offending virtual address is translated to point into this new physical copy.

Finally, the CPU is instructed to restart the offending instruction: since the page table entry is now marked as read-write (pointing to the new physical location), the instruction will execute without further impediment.

Init

257

- on UNIX, **fork** is the only way to make a process
- but **fork** splits existing processes into 2
- the **first process** is special
- it is directly spawned by the kernel on boot

Earlier, we have claimed that **fork** is essentially the only way to create a new process. Of course, this cannot be entirely true, since **fork** may only be executed by an existing process. For this reason, there is one special process, also called **init** or **pid 1** which is directly created by the kernel upon boot. All other processes in a running system, however, descend from this original process by a sequence of forks.

Process Identifier

258

- processes are assigned **numeric identifiers**
- also known as PID (Process ID)
- those are used in **process management**
- used calls like **kill** or **setpriority**

To facilitate process management, each process is assigned a numeric identifier (known as PID, short for process identifier). Process management syscalls, then, take this number as an argument.

Traditionally, the 'namespace' of processes is global: a PID identifies a process globally (unlike, for instance, a file descriptor, which is local to each process). That is, for all users and all processes, the given number always represents the same process (until it is terminated). Please note that in container-based virtualisation, this is no longer strictly true, since different containers will get different PID namespaces even though they share the same kernel.

Process vs Executable

259

- process is a **dynamic** entity
- executable is a **static** file
- an executable contains an initial **memory image**
 - this sets up memory layout
 - and content of the **text** and **data** segments

In previous lectures, we have touched the subject of executables: those are **files** which contain programs. It is very important to understand the difference between an executable (a program, so to speak, at rest) and a process (a program in the process of being executed).

An executable contains, essentially, an initial image of process memory: when a program starts executing, the kernel populates its virtual address space with data from the executable, most importantly the **text** segment which contains instructions, but also the static parts of the **data** segment. Additionally, the executable contains the (virtual) address of the **entry point**: this is the address of the first instruction of the program that should be executed.

Exec

260

- on UNIX, processes are created via **fork**
- how do we **run programs** though?
- **exec**: load a new **executable** into a process
 - this completely **overwrites** process memory
 - execution starts from the **entry point**
- running programs: **fork** + **exec**

However, when a process is created, its memory is **not** populated from an executable: instead, it is a clone of an existing process. To execute a new program, an existing process needs to call an **exec** system call, which then simply replaces the entire memory of the current process with the content of the executable. Only environment variables and command-line arguments (which are both stored in process memory) are copied over to the new address space.

The common 'start a new program' operation (e.g. what happens when the user runs a command in the shell) is then implemented as a **fork** followed by an **exec** in the child process.

Part 5.2: Thread Scheduling

In this section, we will look in more detail how operating systems decide which thread to run when and for how long.

What is a Scheduler?

262

- scheduler has two related tasks
 - **plan** when to run which thread
 - actually **switch** threads and processes
- usually part of the **kernel**
 - even in micro-kernel operating systems

The **scheduler** is a component of the kernel which fills two basic roles: planning and thread switching (including preemption).

Switching Threads

263

- threads of the **same process** share an address space
 - a **partial** context switch is needed
 - only **register state** has to be saved and restored
- no TLB flushing – lower **overhead**

To switch execution from one thread to another, within a single process, it is sufficient to store the current state of user-visible registers into memory, and load the state which corresponds to the other thread (the state of which was saved when it was last interrupted and suspended). Since the address of the (top of the) stack is stored in a register, simply replacing values in registers from a backup also has the effect of switching the active stack.

In this case, the TLB does not need to be flushed, making the switch quite efficient, even if not entirely free (there is still a small penalty due to branch prediction and speculative execution).

Fixed vs Dynamic Schedule

264

- **fixed** schedule = all processes known **in advance**
 - only useful in special / embedded systems
 - can **conserve resources**
 - planning is not part of the OS
- most systems use **dynamic scheduling**
 - what to run next is **decided periodically**

When it comes to the **schedule**, that is, the plan of which thread to execute in which time slot(s), there are two basic types:

1. static schedules are computed ahead of time, for a fixed set of threads with pre-determined priorities and with their relative computational costs also known in advance,
2. dynamic schedules, in which the above information is not known.

Running on a static schedule makes the runtime scheduler particularly simple and robust. However, this approach is unsuitable for all but the simplest use cases, and is usually only found in high-assurance embedded systems.

A dynamic scheduler, on the other hand, allows threads and processes to be created in an ad-hoc manner, during execution. Likewise, the priorities of threads can be determined (and adjusted) at will. The scheduler periodically evaluates the situation and decides what to run either now, or in the very near future.

Preemptive Scheduling

265

- tasks (threads) just run as if they owned the CPU
- the OS forcibly **takes the CPU** away from them
 - this is called **preemption**
- pro: a faulty program **cannot block** the system
- somewhat **less efficient** than cooperative

In most modern operating systems, the scheduler is **preemptive**, that is, it can suspend threads at its own discretion, without cooperation from the threads themselves. In this approach, the user-space software does not need, in principle, be aware that it is running on a time-sharing system: when a thread is preempted and later resumed, the execution continues where it left off, without any immediately detectable effect on the executing program.

A major advantage of preemptive scheduling is that uncooperative (or faulty) programs cannot endanger the system as a whole: the operating system can, at any time, decide to suspend a process, regardless of what code it executes at the time.

Cooperative Scheduling

266

- threads (tasks) **cooperate** to share the CPU
- each thread has to explicitly **yield**
- this can be **very efficient** if designed well
- but a **bad program** can easily **block the system**

A different approach is known as **cooperative scheduling**. In this case, there is no preemption: the currently running thread has to **yield** (give up) the processor voluntarily. This makes it possible to optimize task switching, but it also means that a program which does not yield the processor will simply continue to run unimpeded.

While general-purpose operating systems no longer use preemptive scheduling globally, some programs implement so-called **green threads** which are scheduled cooperatively – scheduling of those threads is implemented entirely in the user-space. Usually, a large number of such cooperative ‘green’ threads is dispatched onto a small number of ‘real’ OS threads (which are preemptively scheduled by the kernel).

Scheduling in Practice

267

- cooperative on Windows 3.x for everything
- **cooperative** for **threads** on classic Mac OS
 - but **preemptive** for **processes**
- **preemptive** on pretty much every modern OS
 - including real-time and embedded systems

The last mainstream operating system to use cooperative scheduling at the operating system level was ‘classic’ Mac OS (before OS X), though it used preemptive scheduling for processes. This way, switching threads within a process could take advantage of the improved efficiency, while processes were unable to block each other. The last mainstream system with fully cooperative scheduling was MS Windows 3.11, released in 1993.

Waiting and Yielding

268

- threads often need to **wait** for resources or **events**
 - they could also use software timers
- a waiting thread should **not consume** CPU time
- such a thread will **yield** the CPU
- it is put on a list and later **woken up** by the kernel

In most programs, it is common that a thread cannot continue its execution until some event takes place, or a resource becomes available. In this case, it is undesirable that the thread should wait actively, that is, spin in a loop that checks whether it can continue. Instead, kernels provide mechanisms which allow threads to be **suspended** until an event arrives, or a resource becomes available (at which point it is resumed by the kernel). This process is not completely free from overhead, but unless the wait is very short (a few dozen CPU cycles), suspending the thread will give a much better overall system throughput.

Run Queues

269

- **runnable** (non-waiting) threads are **queued**
- could be **priority**, round-robin or other queue types
- scheduler **picks** threads from the run queue
- **preempted** threads are put back

There are two reasons while a thread is suspended, that is, it is no longer running: it's either waiting for an event (see above) or its time slot has ended and it was preempted. In the latter case, the thread is put on a **run queue**, which is a list of threads that are ready to run, usually sorted by a **dynamic priority**, which is a number that indicates how soon the thread should run, computed by the scheduler.

Whenever a thread is suspended, the scheduler picks the next thread to execute from a run queue.

Priorities

270

- what **share** of the CPU should a thread get?
- **priorities** are static and dynamic
- **dynamic** priority is adjusted as the thread runs
 - this is done by the system / scheduler
- a **static** priority is assigned by the **user**

Not all threads are equally important, and some should take priority over others whenever they need to run. This is achieved through a combination of priorities: a **static** priority is assigned to each thread by the user. The static priority then feeds into the computation of a **dynamic** priority, which governs scheduling: while a thread is running, its dynamic priority drops and while it is suspended, its dynamic priority increases.

Fairness

271

- **equal** (or priority-based) share per **thread**
- what if one process has many **more threads**?
- what if one user has many **more processes**?
- what if one user group has many **more active users**?

In the system that we have outlined above, scheduling is only concerned with individual threads. Such a system is clearly unfair: processes and users with many threads will get a much higher share of the CPU than processes and users with fewer threads. To mitigate this effect, most operating systems implement some form of fair sched-

uling.

Fair Share Scheduling

272

- we can use a **multi-level** scheduling scheme
- CPU is sliced fairly first among **user groups**
- then among **users**
- then among **processes**
- and finally among **threads**

There are different levels on which a fair scheduler may operate: almost all systems will consider processes, while many will also consider users. In this scheme, the scheduler strives to achieve the following two goals:

1. as long as the given entity (process, user, group) has enough runnable threads, those threads get equal time when compared to any other entity with the same property (i.e. if both process A and process B have runnable threads, the total time slice of process A is equal to that of process B, assuming equal priorities),
2. resources are allocated efficiently: if an entity does not have sufficient runnable threads, the resources it cannot use are distributed among the remaining entities of the same type (i.e. if process A has 1 runnable thread, process B has 3 runnable threads and there are 4 processor cores, process B should not be hampered by process A only being able to use one of them – the system should use all 4 cores in this scenario).

Scheduling Strategies

273

- first in, first served (**batch** systems)
- earliest **deadline** first (realtime)
- round robin
- **fixed priority** preemptive
- **fair share** scheduling (multi-user)

When it comes to computing a dynamic schedule, there is a number of approaches. The simplest of all is a batch scheduler, which is basically no scheduler at all: when a program is executed, it runs until it is done; afterwards, the next program is executed until termination, and so on. In real-time systems, a fairly simple scheduler is known as 'earliest deadline first': in such systems, each task comes with a **deadline** which tells the system when the task must be done, at the latest. In a preemptive setting, then, it is an optimal strategy to first execute the task with the earliest deadline.

A naive general-purpose, preemptive scheduler is known as round robin: it runs all available threads in a fixed order, switching from one to the next as they run out of their time slice. In this system, the per-thread throughput can be prioritised (by tweaking the relative sizes of time slices of different threads), but latency can not.

A preemptive scheduler with priorities fixes the latency problem: if a high-priority thread wakes up, it will be able to run very soon (with low latency), since it can 'jump the queue', unlike in a round-robin scheduler.

Interactivity

274

- **throughput** vs **latency**
- **latency** is more important for **interactive** workloads
 - think phone or desktop systems
 - but also web servers
- **throughput** is more important for **batch** systems
 - think render farms, compute grids, simulation

A scheduler will often need to make a compromise between maximising total computational **throughput** and minimising **latency**. In interactive settings, the dominant concern is latency, while in computational contexts, it's the total throughput that is usually a priority.

Reducing Latency

275

- **shorter** time slices
- more willingness to switch tasks (more **preemption**)
- **dynamic** priorities
- priority boost for **foreground** processes

When optimising for low latency, the scheduler should use short time slices (so that threads that wake up do not need to wait too long for the processor to free up, even if they are not high-priority threads). Likewise, the scheduler should be willing to switch to a higher-priority thread whenever one wakes up, preempting running threads (even if they did not consume their current time slice yet).

Both decisions of course lead to more context switches, which are expensive and hence negatively affect total throughput (less total useful computation is performed in any given time, since more time is spent on overhead – both in the scheduler, but mainly in switching threads and processes).

Maximising Throughput

276

- **longer** time slices
- **reduce context switches** to minimum
- cooperative multitasking

The opposite is true for high-throughput systems: the time slice should be as long as is practical, to minimize the number of context switches the CPU needs to do. Likewise, the scheduler should be unwilling to switch tasks before their current time slice runs out, pushing back threads that woke up in reaction to an event. Whenever possible, preemption should be avoided in favour of cooperation (though this is, nowadays, mainly an application-level concern).

Multi-Core Schedulers

277

- traditionally one CPU, many threads
- nowadays: many threads, many CPUs (cores)
- more complicated **algorithms**
- more complicated & **concurrent-safe** data structures

In traditional designs, an operating system would only expect a single processor to be available, in which case the scheduler would be comparatively simple. However, essentially all modern computers are multi-core systems, and schedulers need to take this into account in two major ways:

1. the algorithm to decide which threads run when **and on which core** is much more complicated,

2. the scheduler must be able to run concurrently with itself: multiple cores must be able to decide which thread to run next in parallel, which means that the data structures used by the scheduler to keep track of threads must be concurrent-safe.

Scheduling and Caches

278

- threads can **move** between CPU **cores**
 - important when a different **core is idle**
 - and a runnable thread is **waiting for CPU**
- but there is a **price** to pay
 - thread / process data is extensively **cached**
 - caches are typically **not shared** by all cores

It is entirely possible to suspend a thread on one core, and wake it up later on another core. Since processor cores are all identical, this does not make a semantic difference – the thread will not be able to directly detect that it has moved. However, if some of the data associated with that thread was still in the per-core part of the cache (typically at least L1, but also L2 and L3 if the core is part of a different package) this will cause a performance hit, as the new core has to re-fetch the data that was already cached by the other core.

Core Affinity

279

- modern schedulers try to **avoid moving threads**
- threads are said to have an **affinity** to a core
- an extreme case is **pinning**
 - this altogether prevents the thread to be **migrated**
- practically, this practice **improves throughput**
 - even if nominal core **utilisation** may be lower

For the above reason, schedulers will try to avoid moving threads between cores. Of course, there are competing concerns: if a thread is runnable and a core other than its preferred core is idle, inefficiency creeps in: the system keeps a processor core idle while a thread is waiting to run. A realistic scheduler needs to strike a balance between those two concerns: if the preferred core is about to be freed up, it will wait, otherwise it will migrate the thread.

Of course, this will sometimes go wrong: a higher-priority thread with affinity to the same core might wake up and the original waiting thread will have to be migrated anyway. For this reason, stronger affinity will cause a drop in core utilisation, but in a well-tuned scheduler, this should not cause a drop in throughput.

NUMA Systems

280

- **non-uniform memory** architecture
 - different memory is attached to different CPUs
 - each SMP block within a NUMA is called a **node**
- **migrating** a process to a **different node** is expensive
 - thread vs node ping-pong can kill performance
 - threads of **one process** should live on one node

In a traditional SMP (symmetric multiply-processor) system, each CPU has equal access to all RAM. However, in large computers with many CPUs, it is advantageous (from the point of view of hardware design) to violate this principle, and allow RAM to be attached locally to a single processor package. Cores from another such **node** then need to access this memory indirectly, adding a penalty both in terms of throughput (how many bytes per unit of time can be transferred from memory to the CPU cache) and in terms of latency (how many cycles it takes from

the time data is requested to the time it is available to the processor). A scheduler needs to take this into account: the affinity of threads (and entire processes) to a particular NUMA node is usually much stronger than the affinity of threads to individual cores within an SMP system: unlike with purely cache-related overhead, the performance hit of running on a different NUMA node is permanent: the data is not automatically migrated into a 'closer' block of memory.

Part 5.3: Interrupts and Clocks

In this section, we will look at how preemptive scheduling is actually implemented, through periodic hardware interrupts.

Interrupt

282

- a way for hardware to **request attention**
- CPU **mechanism** to divert execution
- partial (CPU state only) **context switch**
- switch to **privileged** (kernel) CPU mode

Interrupts allow peripherals to request the attention of the operating system. This is a low-level CPU feature, where asserting an interrupt line (in hardware) causes the processor to stop executing instructions and execute an **interrupt handler**.

When an interrupt comes in, the CPU will consult a table of interrupt handlers, which is set up by the operating system. Additionally, the CPU will store its state (the values stored in registers) into memory (usually on the active stack).

Usually, only a partial context switch is done by the CPU (i.e. the page table is unaffected), but the interrupt routine will execute in privileged mode (hence it can, if it needs to, adjust page tables). In traditional designs, the kernel memory is mapped (but inaccessible from user mode) into all processes, and hence the interrupt handler can directly and immediately read and write kernel memory.

When the interrupt routine is finished, it uses a special instruction (**iret** on x86) which instructs the CPU to restore its state from the stack (which it stored upon entering the interrupt handler) and drop back into user (unprivileged) mode.

Hardware Interrupts

283

- **asynchronous**, unlike software interrupts
- triggered via **bus signals** to the CPU
- IRQ = interrupt request
 - just a different **name** for hardware interrupts
- PIC = programmable **interrupt controller**

We have already talked about software interrupts earlier: interrupt handlers are quite alike in both cases (that is, for hardware and software interrupts), but there are other differences. In particular, a hardware interrupt is **asynchronous**: the CPU will invoke the handler regardless of the instruction that is currently being executed. Hardware-wise, interrupts are realized through bus signalling – a peripheral (or rather an interrupt controller) will send a signal down a wire to the CPU, which will react by starting its interrupt handling sequence.

Interrupt Controllers

284

- PIC: **simple circuit**, typically with 8 input lines
 - peripherals connect to the PIC with wires
 - PIC delivers prioritised signals to the CPU
- APIC: advanced programmable interrupt controller
 - split into a shared **IO APIC** and per-core **local APIC**
 - typically 24 incoming **IRQ lines**
- OpenPIC, MPIC: similar to APIC, used by e.g. Freescale

An interrupt controller is essentially a hub which provides a number of interrupt lines to peripherals, and signals the CPU using a single interrupt line whenever any of the peripherals raises an interrupt. The interrupt controller of course informs the CPU which peripheral is the origin of the interrupt.

Timekeeping

285

- PIT: **programmable interval timer**
 - crystal oscillator + divider
 - **IRQ line** to the CPU
- local APIC timer: built-in, **per-core clock**
- HPET: high-precision event timer
- RTC: real-time clock

A (programmable) timer is another traditionally discrete component that has long been integrated into the main CPU die. Its role is to keep time (historically using a quartz crystal), and raise an interrupt in regular, configurable intervals.

Timer Interrupt

286

- generated by the PIT or the local APIC
- the OS can **set the frequency**
- a hardware interrupt happens on each **tick**
- this creates an opportunity for bookkeeping
- and for **preemptive scheduling**

Every time the configured interval expires, the timer will generate an interrupt. When this happens, the OS kernel has a chance to run, regardless of the currently executing process. Among other things, it is possible to do a context switch by doing an **iret** (or equivalent) into a different thread or process from the one that was interrupted. This is how preemptive scheduling is usually implemented.

Timer Interrupt and Scheduling

287

- measure how much time the current thread took
- if it ran out of its slice, **preempt it**
 - **pick** a new **thread** to execute
 - perform a context switch
- checks are done on each tick
 - **rescheduling** is usually less frequent

The (scheduler part of) the interrupt handler for the timer interrupt quickly checks whether anything needs to be done: that is, it looks up how much time has the currently running thread left of its time slice. If it ran out, then a **reschedule** is in order: the handler will return into a different thread (that is, perform a context switch... if needed, this includes replacing the active page table). The slice check is done on every tick, since it is cheap. On most ticks, however, no rescheduling

will take place, since a typical slice length is many ticks (and is much more expensive).

Timer Interrupt Frequency

288

- typical is 100 Hz
- this means a 10 ms **scheduling tick** (quantum)
- 1 kHz is also possible
 - harms throughput but **improves latency**

The traditional timer frequency on UNIX systems is 100 Hz, or one tick every 10 milliseconds. This means that time slices available to the scheduler are multiples of 10 milliseconds. For interactive systems, the frequency can be increased, with 1 kHz being the highest commonly used setting (yielding a quantum just 1 ms long, cutting down on scheduling latency quite a bit). Of course, the CPU will spend a lot more time in the timer interrupt handler, reducing the amount of useful work it can perform, thus harming throughput.

Tickless Kernels

289

- the timer interrupt **wakes up** the CPU
- this can be **inefficient** if the system is **idle**
- alternative: use **one-off** timers
 - allows the CPU to **sleep longer**
 - this **improves power efficiency** on light loads

Modern processors implement aggressive power management, shutting down parts of the CPU that are not in use. This technique is of course more efficient, if the idle circuits do not need to be woken up very often. If nothing at all is going on, then an entire core can be put to sleep. Unfortunately, the timer interrupt interferes with this somewhat: while the system is completely idle (no threads are ready to run), the processor will still need to process the interrupt, 100 times every second. This has a considerable energy cost, just to figure out that nothing needs to be done.

Instead of using a periodic timer, it's possible to configure a one-off, non-repeating timer, and when it fires, compute when the next scheduling interrupt is needed. This way, during idle times, the processor can sleep undisturbed for much longer periods of time.

Tickless Scheduling

290

- **quantum length** becomes part of the planning
- if a core is idle, wake up on next **software timer**
 - synchronisation of software timers
- other interrupts are **delivered as normal**
 - network or disk activity
 - keyboard, mice, ...

This is done by simply doing away with the fixed scheduling quantum. The scheduler can compute when the next reschedule needs to happen, and not waste any time in the timer interrupt unless the slice is over. Sleeping threads may be blocked for any number of reasons, but the most common are waiting for events or resources – one of the events that are commonly awaited is a timer: the thread has some periodic work to do, and may ask the kernel to be woken up in a second, then do some of its work, and then sleep for another second, and so on. If, at a given moment, all processes are sleeping, then the next timer interrupt needs to happen whenever the soonest of the threads sleeping on a timer is due to wake up. Of course, other interrupts are still delivered as usual, so if a thread is blocked waiting for data to arrive from disk or network, the respective interrupts can cause that thread

to be woken up and scheduled for execution.

Other Interrupts

291

- serial port
 - **data is available** on the port
- **network** hardware
 - data is available in a packet queue
- keyboards, mice
 - **user** pressed a key, moved the mouse
- USB devices in general

Besides the timer interrupt, there are many others that come up, and many of them indirectly affect scheduling too (usually because one of the threads was waiting for the event that the interrupt signalled).

Interrupt Routing

292

- not all CPU cores need to see all interrupts
- APIC can be told how to deliver IRQs
 - the OS can **route IRQs** to CPU cores
- multi-core systems: **IRQ load balancing**
 - useful to **spread out** IRQ overhead
 - especially useful with **high-speed networks**

Finally, let us consider systems with multiple cores (probably the majority of computers nowadays) and how this affects interrupts: clearly, if a packet comes in from the network, it does not make sense to interrupt all cores: we only need a single core to execute the handler to fetch the data or otherwise deal with the event.

What more, this does not need to be the same core every time the interrupt happens: if there are many interrupts, we would like them spread out somewhat evenly across all the cores. This is what IRQ load balancing does: it will route interrupts to cores in such a way that each core has approximately the same amount of work handling them.

Finally, since scheduling is done per-core, it is useful to have a separate timer interrupt for each core: those interrupts are of course not subject to load-balancing or other re-routing: the interrupt is, in fact, generated locally on each core (in the local APIC) and the per-core timer which generates the interrupt is programmed separately.

Review Questions

293

- What is a thread and a process?
- What is a (thread, process) scheduler?
- What do **fork** and **exec** do?
- What is an interrupt?

Part 6: Concurrency and Locking

This lecture will deal with the issues that arise from running multiple threads and processes at the same time, both using time-sharing of a single processor and by executing on multiple physical CPU cores.

Lecture Overview

295

1. Inter-Process Communication
2. Synchronisation
3. Deadlocks

In the first part, we will explore the basic why's and how's of inter-process and inter-thread communication. This will naturally lead to questions about shared resources, and to the topic of thread synchronisation, mutual exclusion and so on. Finally, we will deal with waiting and deadlocks, which arise whenever multiple threads can wait for each other.

What is Concurrency?

296

- events that can happen at the same time
- it is not important if it **does**, only that it **can**
- events can be given a **happens-before** partial order
- they are **concurrent** if unordered by **happens-before**

Generally speaking, an event **happens-before** another event when they are causally connected: the first event makes the other event possible, or directly causes it. However, just as often, events are **not** causally connected: they might happen in any order. We say that such events are **concurrent**: they can happen in either order, or they can happen, figuratively speaking, at the same time.

Why Concurrency?

297

- problem decomposition
 - different **tasks** can be largely independent
- reflecting external concurrency
 - serving **multiple clients** at once
- performance and hardware limitations
 - **higher throughput** on multicore computers

Concurrency arises, in software systems, for a number of reasons: first, and perhaps most important, is problem decomposition: it is much easier to design a large system without explicitly ordering all possible events. A complex system will perform a large number of naturally independent – concurrent – tasks. While it would be certainly possible to impose an artificial order on such tasks, it is not a very practical approach.

Another important reason is that the concurrency came from outside: if there are external events that are concurrent, it is hard to imagine that the responses of the system would be strictly ordered: normally, the actual order of concurrent, external events cannot be predicted, and imposing an order on reactions would mean that at least some of the reactions would be unacceptably delayed.

Finally, concurrency in software allows the underlying hardware to perform better: concurrent instructions can be executed in parallel, without much concern about their relative ordering – which allows the hardware to use its limited resources more efficiently.

Parallel Hardware

298

- hardware is inherently parallel
- software is inherently sequential
- something has to give
 - hint: it's not going to be hardware

Unlike software, which usually consists of a **sequence** of instructions that are executed in order, hardware consists of spatially organized circuits. Most of the circuitry can operate independently of other circuits. For instance, a typical CPU will contain circuits for multiplying numbers, and other circuits for adding numbers. Those circuits are almost entirely independent, and can operate at the same time, on different pairs of numbers. If the processor is adding numbers, it could be multiplying some other numbers at the same time, without any ill effect on the process of addition happening in another part of the processor.

Of course, multi-core processors are an extreme manifestation of this: the separate cores are simply (mostly) independent copies of the same (very complicated) circuit.

Since we cannot make sequential execution much faster than it already is, the best we can hope for is leveraging concurrency to execute as many things in parallel (on parallel hardware) as we can.

Part 6.1: Inter-Process Communication

Communication is an important part of all but the most trivial software. While the mechanisms described in this section are traditionally known as inter-**process** communication (IPC), we will also consider cases where threads of a single process use those mechanisms (and will not use a special name, even though the communication is, in fact, **intra**-process in those cases).

Reminder: What is a Thread

300

- thread is a **sequence** of instructions
- each instruction **happens-before** the next
 - or: happens-before is a total order on the thread
- basic unit of **scheduling**

Before we go any further, we need to recall, from the last lecture, that a thread is a sequence of instructions: as such, each instruction **happens before** the next instruction (for the more mathematically inclined, this simply means that on the instructions of a single thread, happens-before is a linear ordering). Additionally, threads are a basic unit of scheduling: each processor core is running, at any given time, a single thread.

Reminder: What is a Process

301

- the basic unit of **resource ownership**
 - primarily **memory**, but also open files &c.
- may contain one or more **threads**
- processes are **isolated** from each other
 - IPC creates gaps in that isolation

A process, then, is a unit of resource ownership – processes own memory (virtual address spaces), open files, network connections and so on. Each process has at least one (but possibly multiple) threads, which carry out the computation. Since each process is isolated in its own

virtual address space, there is no direct way that threads from different processes can interact (communicate, synchronize). However, some degree of communication is desirable, and required: the operating system therefore provides a few primitives, which allow processes to talk to each other, creating controlled gaps in the isolation.

I/O vs Communication

302

- take standard input and output
 - imagine process A **writes** a file
 - later, process B **reads** that file
- **communication** happens in **real time**
 - between two running threads / processes
 - automatic: without user intervention

Another term that we want to delineate is that of **communication**: in particular, we need to differentiate it from 'offline' input and output. Technically speaking, reading a file from disk is communication, because some program wrote that file earlier. However, this is not the kind of thing that we are interested in: we are only interested in the instances that happen in real time. More formally, there should be alternation of actions from two threads, ordered by happens-before, i.e. at least one instruction of thread A is happens-before-sandwiched between instructions of thread B.

Direction

303

- **bidirectional** communication is typical
 - this is analogous to a conversation
- but unidirectional communication also makes sense
 - e.g. sending commands to a child process
 - do acknowledgments count as communication?

Strictly speaking, using the above definition, all communication is bidirectional. However, truly unidirectional communication is quite rare: even semantically unidirectional communication often includes an acknowledgement or other form of reply, satisfying the definition above.

Communication Example

304

- **network services** are a typical example
- take a web server and a web browser
- the browser **sends a request** for a web page
- the server **responds** by sending data

An intuitive, if technically complicated, example of communication would be the interaction of a web browser with a web server. In this case, the browser will use a network connection to send a request, which will be processed by the server. Finally, the server will send a reply, for which the browser is waiting: it is easy to see the causal connections: the request happens-before the reply which happens-before the browser showing the content to the user.

Files

305

- it is possible to communicate through **files**
- multiple processes can open the **same file**
- one can write data and another can process it
 - the original program picks up the results
 - typical when using **programs as modules**

Multiple programs (processes) can open the same file, and read or write

data into it at the same time. With a little care, this can be done safely. It is not hard to imagine, then, that the two programs could use this capability for communication, as defined above: one of the programs writes some data on at a particular offset, the other program reads them and perhaps confirms this, possibly using a different mechanism.

A File-Based IPC Example

306

- files are used e.g. when you run `cc file.c`
 - it first runs a preprocessor: `cpp -o file.i file.c`
 - then the compiler proper: `cc1 -o file.o file.i`
 - and finally a linker: `ld file.o crt.o -lc`
- the **intermediate** files may be hidden in `/tmp`
 - and deleted when the task is completed

As an example, even if a little specific, let's recall how the compiler driver communicates with the individual compilation stages (the compilation process has been covered in the second lecture, in case you need to review it).

Each step looks the same: the driver arranges an intermediate form of the program to be written to a file, then invokes a subprocess which reads that file and writes its output into another file. Clearly, the invocation of next stage **happens after** the output of the previous has been written.

Directories

307

- communication by **placing** files or links
- typical use: a **spool** directory
 - clients drop files into the directory for processing
 - a server periodically **picks up** files in there
- used for e.g. **printing** and **email**

Another approach for communication through the file system leverages directories: a daemon sets up a directory, into which its clients can drop files. These files are then picked up and processed by the daemon, and then unlinked from the directory.

This approach is often seen in print spooling: the client program drops a file (in suitable format) that it wants printed into a **spool directory**, where it is picked up by the printer daemon, which arranges for the file to be printed and removed from the queue.

Email systems use an analogous process, where mail is either dropped into spool directories as part of internal processing (most mail servers are built from multiple services connected by IPC), or for pickup by individual users.

Pipes

308

- a device for moving **bytes** in a **stream**
 - note the difference from messages
- one process **writes**, the other **reads**
- the reader **blocks** if the pipe is **empty**
- the writer **blocks** if the pipe buffer is **full**

We have already talked about pipes earlier: recall that pipes move bytes from one process to another, using the standard file system API: data is sent using **write** and received using **read**.

An important consideration is that each pipe has a (finite) buffer attached to it: when the buffer is full, a process trying to **write** data will be blocked, until the other process issues a **read**, removing some of the data from the buffer. Likewise, when the buffer is empty, attempting to **read** from the pipe will block, until the other side issues a **write** that and hence provides some data for **read** to return.

UNIX and Pipes

309

- pipes are used extensively in UNIX
- **pipelines** built via the shell's `|` operator
- e.g. `ls | grep hello.c`
- most useful for processing data in **stages**

Especially in the case of user-setup pipes (via shell pipelines), the boundary between IPC and “standard IO” is rather blurry. Programs in UNIX are often written in a style, where they read input, process it and write the result as their output. This makes them amenable for use in pipelines. While pipes are arguably “more automatic” than dropping the output in a file and running another command to process the file, there is also a degree of manual intervention. Another way to look at the difference may be that a pipe is primarily an IPC mechanism, while a file is primarily a storage mechanism.

Sockets

310

- similar to, but **more capable** than pipes
- allows one **server** to talk to many clients
- each **connection** acts like a bidirectional pipe
- could be local but also connected via a **network**

You might remember that sockets are pipe-like devices, but more powerful (and more complicated). A socket allows a single server to open communication channels to multiple clients at the same time (possibly over a network), providing a bidirectional channel between each of the clients and the server.

Additionally, sockets also come in a ‘datagram’ flavour, in which case they do not establish connections and do not behave like pipes at all: instead, in this mode, they can be used to send messages.

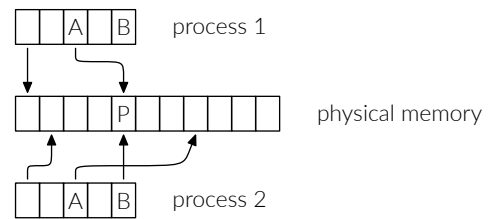
Shared Memory

311

- memory is **shared** when multiple threads can access it
 - happens naturally for **threads** of a single process
 - the primary means of inter-thread communication
- many processes can map the same physical location
 - this is the more traditional setting
 - hence also allows inter-**process** communication

Shared memory is, in some sense, the most straightforward communication mechanism possible: memory access is, after all, a very common operation in a program. Among threads of the same process, all memory is ‘shared’ in the sense that all threads can read and write all memory (since they share the same page table). Of course, in most programs, only a small, carefully delineated part of the memory is used for exchanging data between threads.

Additionally, processes can set up a piece of shared memory between them. However, since there is no shared virtual address space, the each process may have the shared block of memory mapped at different virtual address (let’s call the relevant virtual addresses A and B). However, there is only one block of physical memory involved: if the shared memory resides at the physical address P. Inter-process shared memory is then set up in such a way, that for the first process, A translates to P and in the second process, B translates to P.



The address B may or may not be a valid address in the first process, and the same holds for A in the second process.

Message Passing

312

- communication using discrete **messages**
- we may or may not care about **delivery order**
- we can decide to tolerate **message loss**
- often used across a **network**
- can be implemented on top of sockets

While it is hard to beat the efficiency of shared memory as a communication mechanism, message passing has other advantages: first of all, it is much safer (communication through shared memory can be quite tricky to get right) and, for some use cases, considerably easier to use. Moreover, message passing can be done over networks, which makes it especially suitable for distributed systems.

Part 6.2: Synchronisation

In this section, we will deal with a specific form of communication, where no (application) data is exchanged. The purpose of synchronisation primitives is to ensure that actions of multiple threads or processes proceed in a correct order (of which, there are usually many). Our main motivation will be prevention of **race conditions**, which are, essentially, the **wrong orderings** of **concurrent actions** (for a suitable value of ‘wrong’).

Shared Variables

314

- structured view of **shared memory**
- typical in **multi-threaded** programs
- e.g. any **global** variable in a program
- but may also live in memory from **malloc**

Before we proceed to talk about synchronisation proper, we will have a look at a proper communication device, which we will use as an example: a **shared variable** is simply a named piece of shared memory, available in multiple threads under the same name.

Shared Heap Variable

315

```
void *thread( int *x ) { *x = 7; }
int main()
{
    pthread_t id;
    int *x = malloc( sizeof( int ) );
    pthread_create( &id, NULL, thread, x );
}
```

It is also possible to have anonymous shared variables, allocated dynamically on the heap. They are accessed through a pointer.

Race Condition: Example

316

- consider a **shared counter**, `i`
- and the following **two threads**

```
int i = 0;
void thread1() { i = i + 1; }
void thread2() { i = i - 1; }
```

What is the value of `i` after both finish?

Before we attempt to define a race condition, let's look at an example. In the above program, two threads modify the same shared variable: one adds, and another subtracts, one. Try to think about the possible outcomes, before you look ahead.

Race on a Variable

317

- memory access is **not** atomic
- take `i = i + 1 / i = i - 1`

```
a0 ← load i | b0 ← load i
a1 ← a0 + 1 | b1 ← b0 - 1
store a1 i | store b1 i
```

To understand why the innocent-looking code from the previous slide fails, we need to look at a low-level description of the program (at the level of, essentially, machine instructions). Notice that each of the two statements translates into three separate instructions: load a value from memory, perform the arithmetic operation, and store the result back into memory. At this level of detail, it should be easy to see how the instructions from the two threads can be ordered to give a wrong (or at least unexpected) result, without violating the happens-before relation (remember that instructions of each thread separately are linearly ordered).

Critical Section

318

- any section of code that must **not** be **interrupted**
- the statement `x = x + 1` **could** be a critical section
- what is a critical section is **domain-dependent**
 - another example could be a bank transaction
 - or an insertion of an element into a linked list

If we want the result of the above program to be always 0, we must demand that each of the two statements is an instruction sequence that belongs to a common **critical section**: that is, a span of code that must not be interrupted by any other span of code that belongs to the same critical section. In this case, this means that once either of the **load** instructions starts executing, the thread which did so must first get all the way to the corresponding **store** before the other thread can execute its **load**.

Critical sections do not modify the happens-before relation: the critical sections, as units, can happen in either order (they are still concurrent). However, the entirety of the critical section behaves, with regard to other instances of the same critical section, as a single atomic instruction.

Race Condition: Definition

319

- (anomalous) behaviour that **depends on timing**
- typically among **multiple threads** or processes
- an **unexpected sequence** of events happens
- recall that ordering is not guaranteed

We can now attempt a definition: a **race condition** is a behaviour which

1. depends on timing (in the sense that it depends on a specific ordering of concurrent events),
2. was not anticipated by the programmer.

With the above definition, a race condition can be benign, and the term is often used in this sense. In our study of the phenomenon, however, it is more useful to also include a third clause:

3. the behaviour is erroneous.

Race conditions often lurk in places where there is a lot of concurrency: threaded programs are a prime example (almost all instructions are concurrent with a large number of other instructions coming from other threads).

However, it is important to keep in mind, that some form of communication is necessary for a race condition to arise – concurrency alone is not sufficient. This is because **completely concurrent** processes cannot influence each other, and hence their behaviour cannot depend on the particular ordering of the events that are concurrent among them.

Races in a Filesystem

320

- the file system is also a **shared resource**
- and as such, prone to race conditions
- e.g. two threads both try to create the **same file**
 - what happens if they both succeed?
 - if both write data, the result will be garbled

Since file system is a resource that is shared by otherwise unrelated processes, it creates an environment prone to race conditions: there is ample concurrency, but also a lot of (sometimes accidental) communication.

Mutual Exclusion

321

- context: only one thread can access a resource at once
- ensured by a **mutual exclusion device** (a.k.a **mutex**)
- a mutex has 2 operations: **lock** and **unlock**
- **lock** may need to wait until another thread **unlocks**

Now that we have basic understanding of the problems associated with concurrency, let's look at some of the available remedies. The first, and in some sense simplest synchronisation primitive is a **mutual exclusion device**, designed to enforce **critical sections**.

It is often the case that the critical section is associated with a **resource**: that is, all the code between each **acquisition** and the corresponding **release** of the resource is part of a single critical section. This essentially means that one of the steps of the acquisition process is **locking a mutex** and, analogously, the mutex is **unlocked** as part of the release process associated with the resource.

Semaphore

322

- somewhat **more general** than a mutex
- allows **multiple** interchangeable **instances of a resource**
 - consider there are N identical printers
 - then N processes can be printing at any given time
- basically an atomic counter

Semaphores are, in a sense, a generalisation of a mutex – however, they are only useful when dealing with resources (i.e. it can only be used to guard a general critical section in the case it is exactly equivalent to a mutex). The difference between a mutex and a semaphore is that multiple threads can enter the section of code guarded by a semaphore, but the number is capped by a constant. If the constant is set to 1, then the semaphore is the same as a mutex.

Monitors

323

- a programming **language** device (not OS-provided)
- internally uses standard **mutual exclusion**
- data of the monitor is only accessible to its methods
- only **one thread** can enter the monitor at once

Essentially, a monitor is a fully encapsulated **object** (as defined in object oriented programming), with one additional restriction: each instance of a monitor has a critical section associated with it, and each of its method is, in its entirety, part of this critical section.

As long each public method leaves the object in a consistent state (which is normally required of all objects), a monitor is automatically thread-safe, that is, multiple threads can call its methods without risking race conditions.

Condition Variables

324

- what if the monitor needs to **wait** for something?
- imagine a bounded queue implemented as a monitor
 - what happens if it becomes **full**?
 - the writer must be **suspended**
- condition variables have **wait** and **signal** operations

Since critical sections are often associated with communication, it may happen that code currently in a critical section cannot proceed until some other thread performs a particular action. This use case is served by another synchronisation device, known as a **condition variable**: the variable can be **waited for**, which blocks the calling thread, and **signalled**, which wakes up the waiting thread and allows it to proceed.

Spinlocks

325

- a **spinlock** is the simplest form of a **mutex**
- the **lock** method repeatedly tries to acquire the lock
 - this means it is taking up **processor time**
 - also known as **busy waiting**
- spinlocks contention on the **same CPU** is very **bad**
 - but can be very efficient **between** CPUs

A spinlock is a particularly simple **implementation** of the abstract mutex concept. The state of the device is a single bit, the lock operation uses an atomic compare and swap to change the bit from 0 to 1 **atomically** (i.e. in such a way that the operation fails in case some other thread succeeded in changing it to 1 while the operation was running), looping until it succeeds. The unlock operation simply resets the bit to

0.

There are a few upsides: the implementation is very simple, the state is very compact and the latency is as good as it gets, assuming that the contention is against another CPU core. The uncontended case is pretty much optimal.

There is, however, one serious (and in many scenarios, fatal) drawback: contention on the same CPU core has the worst possible behaviour (maximal latency and minimal throughput). Another important problem, though less severe, is that throughput strongly depends on the average length of the protected critical section: locks that are only held for a short time perform well, but longer critical sections will waste significant resources by holding up a CPU core in the busy-waiting loop.

Overall, spinlocks are useful for protecting critical sections which are short and which are guaranteed to be contended by threads running on different CPU cores. This often happens inside the kernel, but only rarely in user-space programs.

Suspending Mutexes

326

- these need cooperation from the OS **scheduler**
- when lock acquisition fails, the thread **sleeps**
 - it is put on a **waiting** queue in the scheduler
- **unlocking** the mutex will **wake up** the waiting thread
- needs a system call → **slow** compared to a spinlock

A different, much more complicated, but also much more versatile, implementation of a mutex is a **suspending mutex**. The main difference lies in the contended case of the **lock** operation: whenever there is an attempt to lock an already-locked mutex, the unlucky thread is suspended and placed on a wait queue of the scheduler. The **unlock** operation, of course, needs to be augmented to check if there are any threads waiting for the mutex, and if yes, wake up one of them.

The interaction with the scheduler means that both the **lock** and **unlock** operations must perform a system call, at least in some cases. Compared to atomic instructions, system calls are much more expensive, increasing the overhead of the mutex considerably.

A common implementation technique uses a combination of a spinlock and a suspending mutex: the lock operation runs a small number of loops trying to acquire the mutex, and suspends the thread if this fails. This approach combines the good behaviours of both, but is the most complicated implementation-wise, and the state of the mutex is also at least as big as that of a suspending mutex.

Condition Variables Revisited

327

- same principle as a **suspending** mutex
- the waiting thread goes into a wait queue
- **signal** moves the thread back to a run queue
- the busy-wait version is known as **polling**

The usual way to implement a condition variable is to interact with the scheduler, allowing the waiting thread to free up the CPU core it was occupying. A busy-waiting, spinlock-like alternative is possible, though not commonly used.

Barrier

328

- sometimes, **parallel** computation proceeds in **phases**
 - **all** threads must finish phase 1
 - before **any** can start phase 2
- this is achieved with a barrier
 - blocks all threads until the **last one arrives**
 - waiting threads are usually **suspended**

Another synchronisation device that we will consider is a barrier. While the devices that we have described so far can be used in scenarios with more than two threads, their behaviour is always governed by pairwise interactions.

A barrier is different: it synchronises a number of threads at a single point. Only when all the participating threads gather at the barrier are they allowed to continue.

Readers and Writers

329

- imagine a **shared database**
- many threads can read the database at once
- but if one is writing, no other can read nor write
- what if there are always some readers?

Let us consider another synchronisation problem, again involving more than two threads: there is a piece of data (a database if you like) that many threads need to consult, and sometimes update.

The naive solution would be to simply wrap all access to the data structure in a critical section: this is certainly correct, but wasteful, since multiple readers do not interact with each other, but still need to queue up to interact with the data.

The synchronisation device that solves the problem is called an rwlock, or read-write lock, with 3 operations: **lock for reading**, **lock for writing** and **unlock**. The invariant is that if the rwlock is locked for writing, it is locked by exactly one thread. Otherwise, there might be multiple threads holding a read lock (which of course prevents any write locks from being taken until the read locks are all released).

Read-Copy-Update

330

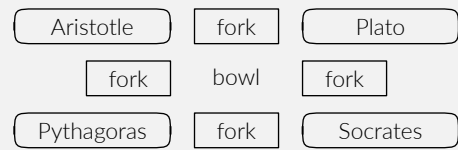
- the fastest lock is **no lock**
- RCU allows **readers** to work while **updates** are done
 - make a copy and **update** the copy
 - point **new readers** to the updated copy
- when is it safe to **reclaim memory**?

In some cases, the readers & writers scenario can be solved without taking any locks at all (not even a critical section is required). The way this works is that a writer, instead of modifying the data in place (and therefore interfering with concurrent reads), makes a copy of the data (which is a read operation, and hence can be safely performed concurrently), performs the updates in this new copy of the data, and then instructs all future readers to use this copy instead, usually by updating a pointer.

Part 6.3: Deadlocks and Starvation

Dining Philosophers

332



In the 'dining philosophers' problem, there is a bowl of food in the middle of a table, a number of philosophers seated around the table, and between each pair of philosophers, there is a single fork. To eat, each philosopher needs to use two forks at the same time. The philosophers sit and think, but when they become hungry, they pick up the forks and eat.

This scenario is used to illustrate the problem of deadlocks and starvation in concurrent systems. Can you come up with instructions for the philosophers that would ensure that each hungry philosopher gets to eat (in finite time)? Consider why simple algorithms fail.

Shared Resources

333

- hardware comes in a **limited** number of **instances**
- many devices can only do **one** thing at a time
- think **printers**, DVD writers, tape drives, ...
- we want to use the devices **efficiently** → **sharing**
- resources can be **acquired** and **released**

The most natural setting to contemplate deadlocks is one in which multiple threads (and/or processes) compete for finite **resources**. In this case, a resource is an abstract object, which can be acquired, used, and released. A resource cannot be used before it has been acquired, and can no longer be used after being released. The thread that has acquired (but not yet released) a resource is said to **own** it.

Network-based Sharing

334

- **sharing** is not limited to processes on one computer
- printers and scanners can be **network-attached**
- the entire network may need to **coordinate access**
 - this could lead to multi-computer **deadlocks**

In practice, some resources are 'remote' – made available to processes on computer A by another computer B attached to the same network. This does not make any practical difference to our consideration (other than perhaps realizing, that a single deadlock may span multiple computers).

Locks as Resources

335

- we explored **locks** in the previous section
- locks (mutexes) are also a form of **resource**
 - a mutex can be acquired (locked) and released
 - a locked mutex **belongs** to a particular thread
- locks are **proxy** (stand-in) resources

In the following, we will consider locks (or even critical sections as such) to be simply a form of resource: the operations on an abstract resource neatly map to operations on an (abstract) lock.

Preemptable Resources

336

- sometimes, held resources can be **taken away**
- this is the case with e.g. **physical memory**
 - a process can be **swapped** to disk if need be
- preemptability may also depend on **context**
 - maybe paging is not available

Some resources can be taken away from their owner temporarily, without causing ill effects, or at an acceptable cost (in terms of latency, loss of throughput, or resource-specific considerations, such as waste of material). A canonic example would be a **page of memory**: even though it is acquired by a particular process, the system might take it away without cooperation from the process, and transparently give it back whenever it is actually used (this technique is known as swapping).

Non-preemptable Resources

337

- those resources **cannot** be (easily) taken away
- think photo printer in the middle of a page
- or a DVD burner in the middle of writing
- **non-preemptable** resources can cause **deadlocks**

Many resources are, however, practically non-preemptable; that is, once acquired by a thread or a process, they cannot be unilaterally taken away without considerable damage, e.g. killing the owning process or irreparably damaging its output through the device.

Resource Acquisition

338

- a process needs to **request access** to a resource
- this is called an **acquisition**
- when the request is granted, it can use the device
- after it is done, it must **release** the device
 - this makes it available for other processes

Now that we have discussed resources in a bit more detail, let's just quickly revisit the protocol for their acquisition and release. In the following, we operate on the assumption that only a limited number of processes can own a particular resource at any given time (most often just one, in fact).

Waiting

339

- what to do if we wish to **acquire** a **busy** resource?
- unless we don't really need it, we have to **wait**
- this is the same as waiting for a **mutex**
- the thread is moved to a wait queue

We will also assume that acquisition is **blocking**: if a resource is busy (owned by another process), the process will wait until it is released before continuing execution. This does not always need to be the case, but it is the most common approach.

Resource Deadlock

340

- two resources, A and B
- two threads (processes), P and Q
- P **acquires** A, Q **acquires** B
- P tries to **acquire** B but has to **wait** for Q
- Q tries to **acquire** A but has to **wait** for P

Now we are finally equipped to look at (resource) **deadlocks**. We will look at the simplest possible case, with two threads and two resources. After the above sequence of events, there can be no further progress: without an outside intervention, both P and Q will be blocked forever. This is what we call a **deadlock**. Please note that it is **usually** the case that both acquisitions in P are concurrent with both acquisitions in Q. Of course, the above situation can be generalized to any number of threads and resources (as long as there are at least 2 of each).

Resource Deadlock Conditions

341

1. mutual exclusion
2. hold and wait condition
3. non-preemptability
4. circular wait

Deadlock is only possible if all 4 are present.

A number of sound design decisions conspire to create the conditions for a deadlock to occur. It is certainly natural that a resource should only be used by a single thread at any given time – that is, after all, the nature of resources.

The **hold and wait** condition arises when a single thread can request multiple resources at once, performing the acquisitions in a sequence. It is hard to object here too, since it fits the linear nature of most programs – the program (or rather each thread of a program) is essentially a sequence of instructions, and each instruction is only performed after the previous has finished. Resource acquisition cannot finish while a resource is busy, and the only way to have more than a single resource is to acquire each in turn.

In some cases, non-preemptability is not even negotiable: it is a property of the resource in question. There is some maneuvering space with critical sections that cause no interaction with any external entity (resource, other threads) with the exception of their associated lock. In this case, it might be possible to roll back the effects of the critical section and attempt to restart it. This is, however, not easy to achieve. Let us consider a static **resource dependency graph**, in which nodes are resources and edges indicate that a hold-and-wait condition exists between them, that is, an edge $A \rightarrow B$ is present in the graph if there is a thread which attempts to acquire B while holding A. The **circular wait** condition is satisfied whenever there is a cycle in this graph.

Non-Resource Deadlocks

342

- not all deadlocks are due to **resource** contention
- imagine a **message-passing** system
- process A is **waiting** for a message
- process B sends a message to A and **waits** for reply
- the message is **lost** in transit

Quite importantly, the above four conditions only pertain to **resource**

deadlocks: there are other types of deadlocks with a different set of conditions. This means that even if we can eliminate one of the 4 conditions, and hence prevent resource deadlocks, this does not mean, unfortunately, that our system will be deadlock-free.

Example: Pipe Deadlock 343

- recall that both the **reader** and **writer** can **block**
- what if we create a pipe in **each direction**?
- process A writes data and tries to read a reply
 - it blocks because the **opposite** pipe is **empty**
- process B reads the data but **waits for more** → deadlock

A typical example of a non-resource deadlock has to do with pipes: remember that an empty pipe blocks on read, while a full pipe blocks on write. There is a number of ways this can go wrong, if there is more than one pipe involved. You can probably spot similarities between this type of deadlock, and the resource deadlock we have discussed at length.

Deadlocks: Do We Care? 344

- deadlocks can be very **hard to debug**
- they can also be exceedingly **rare**
- we may find the risk of a deadlock **acceptable**
- just **reboot** everything if we hit a deadlock
 - also known as the ostrich algorithm

Many (probably most) deadlocks arise from race conditions, and hence will not happen very often. The so-called **ostrich algorithm** takes advantage of this rarity: if a deadlock happens, kill all the involved processes, or possibly just reboot the entire system. Of course, deciding that a deadlock has happened can be tricky.

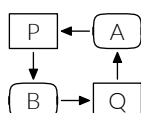
Deadlock Detection 345

- we can at least try to **detect** deadlocks
- usually by checking the **circular wait** condition
- keep a graph of ownership vs waiting
- if there is a **loop** in the graph → **deadlock**

One way to detect deadlocks is to use a dynamic counterpart of the static **circular wait** condition: in this case, the dependency graph has two types of nodes: threads and resources. Edges are always between a thread T and a resource R: $T \rightarrow R$ exists if thread T currently owns resource R, and $R \rightarrow T$ exists, if thread T is waiting for the acquisition of R. If a cycle exists in this graph, the system is in a deadlock (and the threads which lie on the cycle are all blocked). Let's revisit the example with two threads, P and Q, and two resources, A and B that we saw earlier:

1. P successfully **acquires** A (giving rise to edge $A \rightarrow P$)
2. likewise, Q **acquires** B, meaning $B \rightarrow Q$
3. P tries to **acquire** B but has to **wait**, hence $P \rightarrow B$
4. Q tries to **acquire** A but has to **wait**, hence $Q \rightarrow A$

This is the resulting graph, with an obvious cycle in it:



Deadlock Recovery 346

- if a preemptable resource is involved, **reassign** it
- otherwise, it may be possible to do a **rollback**
 - this needs elaborate **checkpointing** mechanisms
- all else failing, **kill** some of the processes
 - the devices may need to be **re-initialised**

A deadlock involving at least one preemptable resource can still happen, but unlike in the 'standard' case, it is possible to recover and continue computation without forcibly terminating any of the threads or processes involved. Instead, one of the resources is preempted, breaking the cycle and allowing the system to make progress again.

If the deadlock instead includes a restartable critical section (as outlined earlier), then this critical section can be rolled back and restarted, again allowing other threads to make progress.

Finally, if all else fails, the system can pick a victim thread on the cycle and terminate it, releasing any resources it might have been holding. Like the other cases, this allows the system to progress again.

Deadlock Avoidance 347

- we can possibly **deny acquisitions** to avoid deadlocks
- must know the **maximum** resources for each process
- avoidance relies on **safe states**
 - worst case: **all processes** ask for **maximum resources**
 - **safe** means deadlocks are **avoided** in the **worst case**

In general, deadlock avoidance is an approach where the system will deny resource acquisitions if they might later lead to a deadlock. The best-known instance of deadlock avoidance is Banker's Algorithm, invented by E. Dijkstra. This algorithm applies in cases where each resource has a number of fungible instances greater than any single thread might request at once. The input to the algorithm is the maximal number of instances of a resource that can be requested by each thread. The **safety** invariant is this:

1. there is a thread T, such that the available resources are sufficient to satisfy its maximal resource allocation,
2. when the thread T terminates (returning all its resources currently allocated resources), the invariant still holds.

Any resource acquisitions which would violate this invariant are denied. The initial conditions imply that, at the outset, we are in a safe state; thus, we have demonstrated that the algorithm is correct. The algorithm assumes that nothing apart from resource acquisitions might block any of the threads. The following table illustrates the algorithm (P, Q and R are threads, the numbers are resources held / maximal allocation):

step	P	Q	R	available	action
1	0/3	0/2	0/4	5/5	P acquires 1
2	1/3	0/2	0/4	4/5	R acquires 2
3	1/3	0/2	2/4	2/5	Q acquires 1
4	1/3	1/2	2/4	1/5	P is denied 1
5	1/3	1/2	2/4	1/5	Q acquires 1
6	1/3	2/2	2/4	0/5	Q terminates
7	1/3	-	2/4	2/5	P acquires 1
8	2/3	-	2/4	1/5	

Notice that in step 4, the request by P has been denied, because there would no longer be any thread guaranteed to be able to make sufficient progress to release resources.

Deadlock Prevention

348

- deadlock avoidance is typically **impractical**
- there are 4 **conditions** for deadlocks to exist
- we can try attacking those conditions
- if we can remove one of them, deadlocks are **prevented**

Deadlock avoidance is, unfortunately, not very practical in general-purpose operating systems: it is often the case, that a particular resource has exactly 1 instance available, and the above algorithm would then force all programs which might use the resource to run one after another. Moreover, the assumption that there are no other interactions between threads is not very realistic either.

Prevention via Spooling

349

- this attacks the **mutual exclusion** property
- multiple programs could write to a printer
- the data is **collected** by a spooling daemon
- which then sends the jobs to the printer **in sequence**

This approach can trade a deadlock on the printer for a deadlock on disk space. However, disk space is much more likely to be preemptable in this scenario, since a job blocked by a full disk can be canceled (and erased from disk) and later retried (unlike a half-printed page).

Prevention via Reservation

350

- we can also try removing **hold-and-wait**
- for instance, we can only allow **batch acquisition**
 - the process must request everything at once
 - this is usually **impractical**
- alternative: release and **re-acquire**

It is certainly conceivable that we might require that all resource acquisitions are done in batches, always requesting, as a single atomic action, all the resources required in a particular section of code. The resources may be released one at a time. If an additional resource is required while other resources are already held, the program must first release everything it holds before requesting the new batch.

Prevention via Ordering

351

- this approach eliminates **circular waits**
- we impose a **global order** on **resources**
- a process can only acquire resources **in this order**
 - must release + **re-acquire** if the order is wrong
- it is impossible to form a cycle this way

Finally, the perhaps most practical approach **within a single program** (or another closed system, but not so much in an operating system) is to impose a global order on resource acquisition, which each thread must observe. This implies that the static **resource dependency graph** is acyclic and deadlocks cannot occur.

Livelock

352

- in a deadlock, no progress can be made
- but it's not much better if processes go back and forth
 - for instance releasing and re-acquiring resources
 - they make no **useful** progress
 - they additionally consume resources
- this is a **livelock** and is just as bad as a deadlock

In a deadlock, the blocked threads are all usually suspended (not runnable). If some or all of the threads involved in a deadlock-like condition are active (busy waiting, polling or otherwise attempting an action which is bound to fail forever), we talk about a livelock: threads are executing instructions, but progress is not being made.

Starvation

353

- starvation happens when a process **can't** make **progress**
- **generalisation** of both **deadlock** and **livelock**
- for instance, **unfair** scheduling on a busy system
- also recall the **readers and writers** problem

Finally, the most general notion is **starvation**, which is a condition when a thread cannot make progress, for whatever reason: it is blocked on a resource, it is denied processor time, etc. An example that is neither a deadlock nor a livelock would arise in a naive solution to the readers & writers problem from earlier: if there are sufficiently many readers so that there is always a read lock on the resource, writers will be blocked indefinitely, and hence starved.

Review Questions

354

- What is a mutex?
- What is a deadlock?
- What are the conditions for a deadlock to form?
- What is a race condition?

Part 7: Device Drivers

Abstracting hardware is one of the major roles of an operating system. While we have already discussed the basic hardware resources (CPU and memory) in detail in previous lectures, so-called peripherals also play an important role. In this lecture, we will look at the interface between the operating system and the peripheral hardware (network interface cards, persistent storage, removable storage, displays, input devices and so on).

Lecture Overview

356

1. Drivers, IO and Interrupts
2. System and Expansion Busses
3. Graphics
4. Persistent Storage
5. Networking and Wireless

Part 7.1: Drivers, IO and Interrupts

Input and Output

358

- we will mostly think in terms of IO
- peripherals produce and consume **data**
- **input** – reading data produced by a device
- **output** – sending data to a device

What is a Driver?

359

- piece of **software** that talks to a **device**
- usually quite specific / **unportable**
 - tied to the particular **device**
 - and also to the **operating system**
- often part of the **kernel**

Kernel-mode Drivers

360

- they are part of the kernel
- running with full **kernel privileges**
 - including **unrestricted** hardware access
- no or **minimal** context switching **overhead**
 - fast but dangerous

Microkernels

361

- drivers are **excluded** from microkernels
- but the driver still needs **hardware access**
 - this could be a special **memory region**
 - it may need to **react** to **interrupts**
- in principle, everything can be done **indirectly**
 - but this may be quite **expensive**, too

User-mode Drivers

362

- many drivers can run completely in **user space**
- this improves **robustness** and **security**
 - driver bugs can't bring the **entire system** down
 - nor can they compromise system **security**
- possibly at some **cost** to **performance**

Drivers in Processes

363

- user-mode drivers typically run in their own **process**
- this means **context switches**
 - every time the device demands attention (interrupt)
 - every time **another process** wants to use the device
- the driver needs **system calls** to talk to the device
 - this incurs even more overhead

In-Process Drivers

364

- what if a (large portion of) a driver could be a **library**
- best of both worlds
 - **no** context switch **overhead** for requests
 - bugs and security problems remain **isolated**
- often used for GPU-accelerated 3D graphics

Port-Mapped IO

365

- early CPUs had very limited **address space**
 - 16-bit addresses mean 64KB of memory
- peripherals got a **separate** address space
- **special instructions** for using those addresses
 - e.g. **in** and **out** on **x86** processors

Memory-mapped IO

366

- devices **share** address space with memory
- **more common** in contemporary systems
- IO uses the same instructions as memory access
 - **load** and **store** on RISC, **mov** on **x86**
- allows **selective** user-level access (via the MMU)

Programmed IO

367

- input or output is **driven** by the **CPU**
- the CPU must **wait** until the device is ready
- would usually run at **bus speed**
 - 8 MHz for ISA (and hence ATA-1)
- PIO would talk to a **buffer** on the device

Interrupt-driven IO

368

- peripherals are **much** slower than the CPU
 - **polling** the device is expensive
- the peripheral can **signal** data availability
 - and also **readiness** to accept more data
- this **frees up CPU** to do other work in the meantime

Interrupt Handlers

369

- also known as **first-level** interrupt handler
- they must run in **privileged** mode
 - they are part of the **kernel** by definition
- the low-level interrupt handler must finish **quickly**
 - it will mask its own interrupt to avoid **re-entering**
 - and **schedule** any long-running jobs for later (SLIH)

Second-level Handler 370

- does any expensive **interrupt-related** processing
- can be executed by a **kernel thread**
 - but also by a user-mode driver
- usually not time critical (unlike first-level handler)
 - can use standard **locking** mechanisms

Direct Memory Access 371

- allows the device to directly read/write **memory**
- this is a **huge** improvement over **programmed IO**
- **interrupts** only indicate buffer **full/empty**
- devices can read and write arbitrary **physical** memory
 - opens up **security** / reliability problems

IO-MMU 372

- like the MMU, but for DMA transfers
- allows the OS to **limit** memory access per device
- very useful in **virtualisation**
- only recently found its way into **consumer** computers

Part 7.2: System and Expansion Busses

History: ISA (Industry Standard Architecture) 374

- 16-bit system **expansion** bus on IBM PC/AT
- **programmed IO** and **interrupts** (but no DMA)
- a fixed number of hardware-configured **interrupt lines**
 - likewise for I/O port ranges
 - the HW settings then need to be **typed back** for SW
- parallel data and address transmission

MCA, EISA 375

- MCA: Micro Channel Architecture
 - **proprietary** to IBM, patent-encumbered
 - 32-bit, **software-driven** device configuration
 - expensive and ultimately a market **failure**
- EISA: Enhanced ISA
 - a 32-bit extension of ISA
 - mostly created to avoid MCA licensing costs
 - short-lived and replaced by PCI

VESA Local Bus 376

- memory mapped IO & **DMA** on otherwise ISA systems
- **tied** to the 80486 line of Intel CPUs (and AMD clones)
- primarily for **graphics cards**
 - but also used with hard drives
- quickly fell out of use with the arrival of PCI

PCI: Peripheral Component Interconnect 377

- a 32-bit successor to ISA
 - 33 MHz (compared to 8 MHz for ISA)
 - later revisions at 66 MHz, PCI-X at 133 MHz
 - added support for **bus-mastering** and DMA
- still a **shared**, parallel bus
 - all devices share the same set of wires

Bus Mastering 378

- normally, the CPU is the bus **master**
 - which means it initiates communication
- it's possible to have multiple masters
 - they need to agree on a conflict resolution protocol
- usually used for accessing the memory

DMA (Direct Memory Access) 379

- the most common form of bus mastering
- the CPU tells the device what and where to write
- the device then sends data directly to RAM
 - the CPU can work on other things in the meantime
 - completion is signaled via an interrupt

Plug and Play 380

- the ISA system for IRQ configuration was **messy**
- MCA pioneered software-configured devices
- PCI further improved on MCA with "Plug and Play"
 - each PCI device has an ID it can **tell** the system
 - enables **enumeration** and automatic **configuration**

PCI IDs and Drivers 381

- PCI allows for device enumeration
- device **identifiers** can be paired to device **drivers**
- this allows the OS to load and configure its drivers
 - or even download / install drivers from a vendor

AGP: Accelerated Graphics Port 382

- PCI eventually became too **slow** for GPUs
 - AGP is based on PCI and only **improves performance**
 - enumeration and configuration stays the same
- adds a dedicated **point-to-point** connection
- multiple transfers per clock (up to 8, for 2 GB/s)

PCI Express 383

- the current high-speed peripheral bus for PC
- builds on / **extends** conventional PCI
- point-to-point, **serial** data interconnect
- much improved **throughput** (up to ~30GB/s)

USB: Universal Serial Bus 384

- primarily for **external** peripherals
 - keyboards, mice, printers, ...
 - replaced a host of **legacy ports**
- later revisions allow **high-speed** transfers
 - suitable for storage devices, cameras &c.
- device enumeration, capability **negotiation**

USB Classes 385

- a set of **vendor-neutral** protocols
- HID = human-interface device
- mass storage = disk-like devices
- audio equipment
- printing

Other USB Uses 386

- ethernet adapters
- usb-serial adapters
- wifi adapters (dongles)
 - there isn't a universal protocol
 - each USB WiFi adapter needs a special driver
- bluetooth

ARM Busses 387

- ARM is typically used in System-on-a-Chip designs
- those use a **proprietary** bus to connect peripherals
- there is less need for enumeration
 - the entire system is baked into a single chip
- the peripherals can be **pre-configured**

USB and PCIe on ARM 388

- USB nor PCIe are exclusive to the PC platform
- most ARM SoC's support USB devices
 - for slow and medium-speed off-SoC devices
 - e.g. used for **ethernet** on RPi 1
- some ARM SoC's support PCI Express
 - this allows for **high-speed** off-SoC peripherals

PCMCIA & PC Card 389

- People Can't Memorize Computer Industry Acronyms
 - PC = Personal Computer, MC = Memory Card
 - IA = International Association
- **hotplug**-capable notebook **expansion** bus
- used for memory cards, network adapters, modems
- comes with its own set of drivers (cardbus)

ExpressCard 390

- an **expansion card** standard like PCMCIA / PC Card
- based on PCIe and USB
 - can mostly **re-use** drivers for those standards
- not in wide use anymore
 - last update was in 2009, introducing USB 3 support
 - the industry association **disbanded** the same year

miniPCIe, mSATA, M.2 391

- those are **physical interfaces**, not special busses
- they provide some mix of PCIe, SATA and USB
 - also other protocols like I²C, SMBus, ...
- used mainly for compact SSDs and wireless
 - also GPS, NFC, bluetooth, ...

Part 7.3: Graphics and GPUs

Graphics Cards 393

- initially just a device to **drive displays**
- reads pixels from **memory** and provides **display** signal
 - basically a DAC with a clock
 - the memory can be part of the graphics card
- evolved **acceleration** capabilities

Graphics Accelerator 394

- allows common **operations** to be done in **hardware**
- like drawing lines or filled **polygons**
- the pixels are computed directly in video RAM
- this can **save** considerable **CPU time**

3D Graphics 395

- rendering 3D scenes is **computationally intensive**
- CPU-based, **software-only** rendering is possible
 - texture-less in early flight simulators
 - bitmap textures since '95 / '96 (Descent, Quake)
- CAD workstations had 3D accelerators (OpenGL '92)

GPU (Graphical Processing Unit) 396

- a term coined by nVidia near the end of '90s
- originally a purpose-built **hardware renderer**
 - based on polygonal meshes and Z buffering
- increasingly more **flexible** and **programmable**
- on-board RAM, high-speed connection to system RAM

GPU Drivers

397

- split into a number of components
- graphics output / frame buffer access
- **memory management** is often done in kernel
- geometry, textures &c. are prepared **in-process**
- front end API: OpenGL, Direct3D, Vulkan, ...

Shaders

398

- current GPUs are **computation** devices
- the GPU has its own machine code for **shaders**
- the GPU driver contains a **shader compiler**
 - either all the way from a high level language (HLSL)
 - or starting with an intermediate code (SPIR)

HLSL = High-Level Shader Language SPIR = Standard Portable Intermediate Representation

Mode Setting

399

- deals with **screen** configuration and **resolution**
- including support for e.g. **multiple displays**
- usually also supports primitive (SW-only) **framebuffer**
- often in-kernel, with minimum user-level support

Graphics Servers

400

- multiple apps cannot all drive the graphics card
 - the graphics hardware needs to be **shared**
 - one option is a **graphics server**
- provides an IPC-based **drawing** and/or **windowing** API
- performs **painting** on behalf of the applications

Compositors

401

- a more direct way to share graphics cards
- each application gets its **own buffer** to paint into
- painting is mostly done by a (context-switched) GPU
- the individual buffers are then **composed** onto screen
 - composition is also hardware-accelerated

GP-GPU

402

- general-purpose GPU (CUDA, OpenCL, ...)
- used for **computation** instead of just graphics
- basically a return of vector processors
- close to CPUs but not part of normal OS scheduling

Part 7.4: Persistent Storage

Drivers

404

- split into adapter, bus and device drivers
- often a single driver per device type
 - at least for disk drives and CD-ROMs
- bus **enumeration** and **configuration**
- data addressing and **data transfers**

IDE / ATA

405

- Integrated Drive Electronics
 - disk controller becomes part of the disk
 - standardised as ATA-1 (AT Attachment ...)
- based on the ISA bus, but with cables
- later adapted for non-disk use via ATAPI

ATA Enumeration

406

- each ATA **interface** can attach only 2 drives
 - the drives are HW-configured as master/slave
 - this makes enumeration quite simple
- multiple ATA interfaces were standard
- no need for specific HDD drivers

PIO vs DMA

407

- original IDE could only use **programmed IO**
- this eventually became a serious **bottleneck**
- later ATA revisions include **DMA** modes
 - up to 160MB/s with highest DMA modes
 - compare 1900MB/s for SATA 3.2

SATA

408

- **serial**, point-to-point replacement for ATA
- hardware-level incompatible to (parallel) ATA
 - but SATA inherited the ATA **command set**
 - legacy mode lets PATA drivers talk to SATA drives
- hot-swap capable – replace drives in a **running system**

AHCI (Advanced Host Controller Interface) 409

- **vendor-neutral** interface to SATA controllers
 - in theory only a single 'AHCI' driver is needed
- an alternative to 'legacy mode'
- NCQ = Native Command Queuing
 - allows the drive to re-order requests
 - another layer of IO scheduling

The PATA-compatible mode hides most features.

ATA and SATA Drivers 410

- the host controller (adapter) is mostly vendor-neutral
- the **bus driver** will expose the ATA command set
 - including support for **command queuing**
- device driver uses the bus driver to talk to devices
- partially re-uses SCSI drivers for ATAPI &c.

SCSI (Small Computer System Interface) 411

- originated with minicomputers in the 80's
- more complicated and **capable** than ATA
 - ATAPI basically encapsulates SCSI over ATA
- device **enumeration**, including **aggregates**
 - e.g. entire enclosures with many drives
- also allows CD-ROM, tapes, scanners (!)

SCSI Drivers 412

- split into: a host bus adapter (HBA) driver
- a generic SCSI bus and command component
 - often re-used in both ATAPI and USB storage
- and per-**device** or per-class drivers
 - optical drives, tapes, CD/DVD-ROM
 - standard disk and SSD drives

iSCSI 413

- basically SCSI over TCP/IP
- entirely **software-based**
- allows standard computers to serve as **block storage**
- takes advantage of fast cheap ethernet
- re-uses most of the **SCSI driver stack**

NVMe: Non-Volatile Memory Express 414

- a fairly simple protocol for PCIe-attached storage
- optimised for SSD-based devices
 - much bigger and more **command queues** than AHCI
 - better / faster interrupt handling
- stresses **concurrency** in the kernel block layer

415

USB Mass Storage

- an USB device class (vendor-neutral protocol)
 - one driver for the entire class
- typically USB **flash drives**, but also external **disks**
- USB 2 is not suitable for high-speed storage
 - USB 3 introduced UAS = USB-Attached SCSI

Tape Drives 416

- unlike disk drives, only allow **sequential** access
- needs support for media **ejection**, **rewinding**
- can be attached with SCSI, SATA, USB
- parts of the driver will be **bus-neutral**
- mainly for data **backup**, capacities 6-15TB

Optical Drives 417

- mainly used as a **read-only** distribution medium
- laser-facilitated reading of a rotating disc
- can be again attached to SCSI, SATA or USB
- conceived for **audio playback** → very slow seek

Optical Disk Writers (Burners) 418

- behaves more like a **printer** for optical **disks**
- drivers are often done in **user space**
- attached by one of the standard **disk busses**
- **special programs** required to burn disks
 - alternative: packet-writing drivers

Part 7.5: Networking and Wireless

Networking 420

- networks allow **multiple computers** to exchange **data**
 - this could be files, streams or messages
- there are **wired** and **wireless** networks
- we will only deal with the **lowest layers** for now
- NIC = Network Interface Card

Ethernet 421

- specifies the **physical** medium
- **on-wire** format and **collision** resolution
- in modern setups, mostly **point-to-point** links
 - using active **packet switching** devices
- transmits data in **frames** (low-level packets)

Addressing

422

- at this level, only **local** addressing
 - at most a single LAN segment
- uses baked-in MAC addresses
 - MAC = Media Access Control
- addresses belong to **interfaces**, not computers

Transmit Queue

423

- **packets** are picked up from **memory**
- the OS **prepares** packets into the transmit **queue**
- the device picks them up **asynchronously**
- similar to how SATA queues commands and data

Receive Queue

424

- data is also **queued** in the other direction
- the NIC copies packets into a **receive queue**
- it invokes an **interrupt** to tell the OS about new items
 - the NIC may batch multiple packets per interrupt
- if the queue is not cleared quickly → **packet loss**

Multi-Queue Adapters

425

- fast adapters can **saturate** a CPU
 - e.g. 10GbE cards, or multi-port GbE
- these NICs can manage **multiple** RX and TX queues
 - each queue gets its own interrupt
 - different queues → possibly different **CPU cores**

Checksum and TCP Offloading

426

- more advanced adapters can **offload** certain features
- e.g. computation of mandatory packet **checksums**
- but also TCP-related features
- needs both **driver** support and **TCP/IP stack** support

WiFi

427

- **wireless** network interface – “wireless ethernet”
- **shared** medium – electromagnetic waves in air
- (almost) mandatory **encryption**
 - otherwise easy to **eavesdrop** or even actively **attack**
- a very **complex** protocol (relative to hardware standards)
 - assisted by **firmware** running on the adapter

Bluetooth

428

- a **wireless** alternative to USB
- allows **short-distance** radio links with **peripherals**
 - input (keyboard, mice, game controllers)
 - audio (headsets, speakers)
 - data transmission (e.g. smartphone sync)
 - gadgets (watches, heartrate monitoring, GPS, ...)

Review Questions

429

- What is memory-mapped IO and DMA?
- What is a system bus?
- What is a graphics accelerator?
- What is a NIC receive queue?

Part 8: Network Stack

In this lecture, we will look at networking from the point of view of the operating system. We will mainly focus on the internet stack: that is TCP/IP and related protocols and host name resolution. We will also look at network file systems (i.e. file systems which are stored by one computer on a network, but can be used by multiple other computers on the same network).

Lecture Overview

431

1. Networking Intro
2. The TCP/IP Stack
3. Using Networks
4. Network File Systems

We will first do a quick recap of networking terminology and of the basic concepts in general terms. Afterwards we will look at the TCP/IP stack more specifically, and how it matches the more general notions introduced earlier. The next part of the lecture will focus on network-related application programming interfaces. Finally, we will look at file system sharing in a network environment.

Part 8.1: Networking Intro

In this section, we will mostly deal with familiar network-related concepts, so that we have sufficient context down the line, when we delve into a bit more detail and into OS-level specifics.

Host and Domain Names

433

- **hostname** = human readable computer name
- **hierarchical** system, little endian: `www.fi.muni.cz`
- FQDN = fully-qualified domain name
- the **local suffix** may be omitted (`ping aisa`)

The first thing we need to understand is how to identify computers within a network. The primary means to do this is via **hostnames**: human-readable names, which come in two flavours: the name of the computer itself, and a fully-qualified name, which includes the name of the network to which the computer is connected, so to speak.

Network Addresses

434

- address = **machine**-friendly and numeric
- IPv4 address: 4 octets (bytes): **192.168.1.1**
 - the octets are ordered MSB-first (big endian)
- IPv6 address: 16 octets
- Ethernet (MAC): 6 octets, **c8:5b:76:bd:6e:0b**

While humans prefer to refer to computers using human-readable names, those are not suitable for actual communication. Instead, when computers need to refer to other computers, they use numeric addresses (just like with memory locations or disk sectors). Depending on the protocol, the size and structure of the address may be different: traditional IPv4 uses 4 octets, while the addresses in the newer IPv6 use up 16 (128 bits). One other type of address that you can commonly encounter is MAC (from media access control), which is best known from the Ethernet protocol.

Network Types

435

- LAN = Local Area Network
 - Ethernet: **wired**, up to 10Gb/s
 - WiFi (802.11): **wireless**, up to 1Gb/s
- WAN = Wide Area Network (the Internet)
 - PSTN, xDSL, PPPoE
 - GSM, 2G (GPRS, EDGE), 3G (UMTS), 4G (LTE)
 - also LAN technologies – Ethernet, WiFi

Networks are broadly categorized into two types: local area, spanning an office, a household, maybe a building. LAN is usually a single **broad-cast domain**, which means, roughly speaking, that each computer can directly reach any other computer attached to the same LAN. The most common technologies (layers 1 and 2) used in LANs are the wired **ethernet** (the most common variety running at 1Gb/s, less common but still mainstream at versions at 10Gb/s) and the wireless **WiFi** (formally known as IEEE 802.11).

Wide-area networks, on the other hand, span large distances and connect a large number of computers. The canonic WAN is the internet, or the network of an ISP (internet service provider). Wide area networks often use a different set of low-level technologies.

Networking Layers

436

1. Link (Ethernet, WiFi)
2. Internet / Network (IP)
3. Transport (TCP, UDP, ...)
4. Application (HTTP, SMTP, ...)

The standard model of networking (known as Open Systems Interconnection, or OSI for short) splits the stack into 7 layers, but TCP/IP-centric view of networking often only distinguishes 4, as outlined above. The link layer roughly corresponds to OSI layers 1 (physical) and 2 (data), the internet layer is OSI layer 3, the transport layer is OSI layer 4 and the rest (OSI layers 5 through 7) is lumped under the application layer.

We will follow the simplified TCP/IP model, **but** whenever we refer to layers by number, those are the OSI numbers, as is customary (specifically, IP is layer 3 and TCP is layer 4).

Networking and Operating Systems

437

- a **network stack** is a standard part of an OS
- large part of the stack lives in the **kernel**
 - although this only applies to **monolithic** kernels
 - microkernels use **user-space** networking
- another chunk is in system **libraries & utilities**

For the last two decades or so, networking has been a standard service provided by general-purpose operating systems. In systems with a monolithic kernel, a significant part of the network stack (everything up to and including the transport layer) is part of the kernel and is exposed to user programs via the sockets API.

Additional application-layer functionality is usually available in system libraries: most importantly domain name resolution (DNS) and encryption (TLS, short for transport-layer security, which is confusingly enough an application-layer technology).

Kernel-Side Networking

438

- device **drivers** for networking **hardware**
- network and transport **protocol** layers
- **routing** and packet filtering (firewalls)
- networking-related **system calls** (sockets)
- network **file systems** (SMB, NFS)

The link layer is generally covered by device drivers and the client and server sides of TCP/IP are exposed via the socket API. There are additional components in TCP/IP networks, though: some of them, like routing and packet filtering can be often done in software, and if this is the case, they are usually implemented in the kernel. Bridging and switching (which belong to the link layer) can be done in software too, but is rarely practical. However, many operating systems implement one or both to better support virtualisation.

A few application-layer network services may be implemented in the kernel too, most notably network file systems, but sometimes also other protocols (e.g. kernel-level HTTP acceleration).

System Libraries

439

- the **socket** and related APIs
- host **name resolution** (a DNS client)
- **encryption** and data **authentication** (SSL, TLS)
- **certificate** handling and validation

Strictly speaking, the socket API is the domain of system libraries (though in most monolithic kernels, the C functions will map 1:1 to system calls; however, in microkernels, the networking stack is split differently and system libraries are likely to pick up a bigger share of the work).

Since nearly all network-related programs need to be able to resolve hostnames (translate the human-readable name to an IP address), this service is usually provided by system libraries. Likewise, encryption is ubiquitous in the modern internet, and most operating systems provide an SSL/TLS stack, including certificate management.

System Utilities & Services

440

- network **configuration** (`ifconfig`, `dhclient`, `dhcpd`)
- route management (`route`, `bgpd`)
- **diagnostics** (`ping`, `traceroute`)
- packet logging and inspection (`tcpdump`)
- other network services (`ntpd`, `sshd`, `inetd`)

The last component of the network stack is located in system utilities and services (daemons). Those are concerned with configuration (including assigning addresses to interfaces and autoconfiguration, e.g. DHCP or SLAAC) and route management (especially important for software-based routers and multi-homed systems).

A suite of diagnostic tools is also usually present, at very least the `ping` and `traceroute` programs which are useful for checking connectivity, perhaps tools like `tcpdump` which allow the operator to inspect packets arriving at an interface.

Networking Aspects

441

- packet format
 - what are the **units of communication**
- addressing
 - how are the sender and recipient **named**
- packet delivery
 - how a message is **delivered**

When looking at a network protocol, there are three main aspects to consider: the first is, what constitutes the unit of communication, i.e. how the packets look, what information they carry and so on. The second is addressing: how are target computers and/or programs designated. Finally, packet delivery is concerned with how messages are delivered from one address to another: this could involve routing and/or address translation (e.g. between link addresses and IP addresses).

Protocol Nesting

442

- protocols run **on top** of each other
- this is why it is called a network **stack**
- higher levels make use of the lower levels
 - HTTP uses abstractions provided by TCP
 - TCP uses abstractions provided by IP

Since we are talking about a **protocol stack**, it is important to understand how the individual layers of the stack interact with each other. Each of the above aspects cuts through the stack slightly differently – we will discuss each in a bit more detail in the following few slides.

Packet Nesting

443

- higher-level **packets** are just **data** to the lower level
- an Ethernet **frame** can carry an **IP packet** in it
- the **IP packet** can carry a **TCP packet**
- the **TCP packet** can carry (a fragment of) an **HTTP request**

When we consider packet structure, it is most natural to start with the bottom layers: the packets of the higher layers are simply data for the lower layer. The overall packet structure looks like a matryoshka: an ethernet frame is wrapped around an IP packet is wrapped around an UDP packet and so on,

From the point of view of the upper layers, packet size is an important consideration: when packet-oriented protocols are nested in other

packet-oriented protocols, it is useful if they can match their packet sizes (most protocols have a limit on packet size). With the size limitations in mind, in the view ‘from top’, a packet is handed down to the lower layer as data, the upper layer being oblivious to the additional framing (headers) that the lower layer adds.

Stacked Delivery

444

- delivery is, in the abstract, **point-to-point**
 - routing is mostly **hidden** from upper layers
 - the upper layer requests **delivery** to an **address**
- lower-layer protocols are usually **packet-oriented**
 - packet size mismatches can cause **fragmentation**
- a packet can pass through **different** low-level **domains**

When it comes to delivery, the relationships between layers are perhaps the most complicated. In this case, the view from top to bottom is the most appropriate, since lower layers provide delivery as a service to the upper layer.

Since the delivery on the internet layer (OSI layers 3 and up) is usually much wider in scope than that of the link layer, it is quite common that a single IP packet will traverse a number of link-layer domains.

Layers vs Addressing

445

- not as straightforward as packet nesting
 - address relationships are tricky
- **special protocols** exist to translate addresses
 - DNS for hostname vs IP address mapping
 - ARP for IP vs MAC address mapping

Finally, since (packet, data) delivery is a service provided by the lower layers to the upper layers, the upper layer must understand and provide correct lower-level addresses. The easiest way to look at this aspect is pairwise: the link layer and the internet layer obviously need to interact, usually through a special protocol which executes on the link layer, but logically belongs to the internet layer, since it deals with IP addresses.

Situation between the internet and transport layers is much simpler: the address at the transport layer simply contains the internet layer address as a field (e.g. a TCP address is an IP address + a port number). Finally, the relationship between the application layer and the transport layer is analogous (but not entirely the same) to the internet/link situation. The application layer primarily uses host names to identify computers, and uses a special protocol, known as DNS, which operates using transport-layer addresses, but otherwise belongs to the application layer.

ARP (Address Resolution Protocol)

446

- finds the MAC that corresponds to an IP
- required to allow **packet delivery**
 - IP uses the **link layer** to deliver its packets
 - the link layer must be given a **MAC address**
- the OS builds a **map** of IP \rightarrow MAC **translations**

The address resolution protocol, which straddles the link/internet boundary, enables the internet layer to deliver its packets using the services of the link layer. Of course, to request link-layer delivery of a packet, a link address is required, but the IP packet only contains an IP address. The ARP protocol is used to find link addresses of IP nodes which exist in the local network (this includes routers, which operate on the internet layer – in other words, packets destined to leave the

local network are sent to a router, using the router's IP address, which is translated into a link-layer address using ARP as usual).

Ethernet

447

- **link-level** communication protocol
- largely implemented **in hardware**
- the OS uses a well-defined interface
 - packet receive and submit
 - using MAC addresses (ARP is part of the OS)

Perhaps the most common link layer protocol is ethernet. Most of the protocol is implemented directly in hardware and the operating system simply uses an unified interface exposed by device drivers to send and receive ethernet frames.

Packet Switching

448

- **shared media** are inefficient due to **collisions**
- ethernet is typically **packet switched**
 - a **switch** is usually a **hardware device**
 - but also in software (usually for virtualisation)
 - physical connections form a **star topology**

High-speed networks are almost exclusively **packet switched**, that is, a node sends packets (frames) to a **switch**, which has a number of physical ports and keeps track of which MAC addresses are reachable on which physical ports. When a frame arrives to a switch, the recipient MAC address is extracted, and the packet is forwarded to the physical port(s) which are associated to that MAC address.

Bridging

449

- bridges operate at the **link layer** (layer 2)
- a bridge is a two-port device
 - each port is connected to a **different LAN**
 - the bridge joins the LANs by **forwarding** frames
- can be done in hardware or software
 - **brctl** on Linux, **ifconfig** on OpenBSD

Bridges are analogous to switches, with one major difference: while the expectation for a switch is that there are many physical ports, but each has only one MAC address attached to it (with perhaps the exception of a special 'uplink' port). A bridge, on the other hand, is optimized for the case of two physical ports, but each side will have many MAC addresses associated with it.

Tunneling

450

- tunnels are **virtual layer 2 or 3 devices**
- they **encapsulate** traffic using a higher-level protocol
- tunneling can implement **Virtual Private Networks**
 - a **software bridge** can operate over an UDP tunnel
 - the tunnel is usually **encrypted**

Tunnelling is a technique which allows lower-layer traffic to be nested in the application layer of an existing network. The typical use case is to tie physically distant computers into a single broadcast (link layer) or routing (internet layer) domain.

In this case, there are two instances of the network stack: the VPN software implements an application layer protocol running in the outer stack, while also acting as a link-layer interface (or an internet-layer

subnet) that is bridged (routed) as if it was just another physical interface.

PPP (Point-to-Point Protocol)

451

- a **link-layer** protocol for **2-node networks**
- available over many **physical connections**
 - phone lines, cellular connections, DSL, Ethernet
 - often used to connect endpoints to the ISP
- supported by most operating systems
 - split between the **kernel** and **system utilities**

The point-to-point protocol is another somewhat important and ubiquitous example of a link-layer protocol and is usually found on connections between LANs, or between a LAN and a WAN.

Wireless

452

- WiFi is mostly like (slow, unreliable) Ethernet
- needs **encryption** since anyone can listen
- also **authentication** to prevent **rogue connections**
 - PSK (pre-shared key), EAP / 802.11x
- encryption needs **key management**

Finally, WiFi is, from the point of view of the rest of the stack, essentially a slow, unreliable version of ethernet, though internally, the protocol is much more complicated.

Part 8.2: The TCP/IP Stack

In this section, we will look at the TCP/IP stack proper, and we will also discuss DNS in a bit more detail.

IP (Internet Protocol)

454

- uses 4 byte (v4) or 16 byte (v6) addresses
 - split into **network** and **host** parts
- it is a packet-based protocol
- is a **best-effort** protocol
 - packets may get lost, reordered or corrupted

IP is a low-overhead, packet-oriented protocol in wide use across the internet and most local area networks (whether they are attached to the internet or not). Quite importantly, its low-overhead nature means that it does not guarantee delivery, nor the integrity of the data it transports.

IP Networks

455

- IP networks roughly correspond to LANs
 - hosts on the **same network** are located with ARP
 - **remote** networks are reached via **routers**
- a **netmask** splits the address into network/host parts
- IP typically runs on top of Ethernet or PPP

Within a single IP network, delivery is handled by the link layer – the local network being identified by a common address prefix (the length of this prefix is part of the network configuration, and is known as the netmask).

Routing

456

- routers **forward** packets **between networks**
- somewhat like **bridges** but **layer 3**
- routers act as normal **LAN endpoints**
 - but represent entire remote IP networks
 - or even the entire Internet

Packets for recipients outside the local network (i.e. those which do not share the network part of the address with the local host) are **routed**: a layer 3 device, analogous to a layer 2 switch, forwards the packet to one of its interfaces (into another link-layer domain). The **routing tables** are, however, much more complex than the information maintained by a switch, and their maintenance across the internet is outside the scope of this subject.

ICMP: Internet Control Message Protocol

457

- **control** messages (packets)
 - destination host/network unreachable
 - time to live exceeded
 - fragmentation required
- **diagnostic** packets, e.g. the **ping** command
 - **echo request** and **echo reply**
 - combine with TTL for **traceroute**

ICMP is the 'service protocol' used for diagnostics, error reporting and network management. The role of ICMP was substantially extended with the introduction of IPv6 (e.g. to include automatic network configuration, via router advertisements and router solicitation packet types). ICMP does not directly provide any services to the application layer.

Services and TCP/UDP Port Numbers

458

- networks are generally used to **provide services**
 - each computer can host multiple
- different **services** can run on different **ports**
- port is a 16-bit number and some are given names
 - port 25 is SMTP, port 80 is HTTP, ...

As we have briefly mentioned earlier, transport-layer addresses have two components: the IP address of the destination computer and a **port number**, which designates a particular service or application running on the destination node.

TCP: Transmission Control Protocol

459

- a **stream-oriented** protocol on top of IP
- works like a **pipe** (transfers a byte sequence)
 - must respect **delivery order**
 - and also **re-transmit** lost packets
- must establish **connections**

The two main transport protocols in the TCP/IP protocol family are TCP and UDP, with the former being more common and also considerably more complicated. Since TCP is stream-oriented and reliable, it needs to implement the logic to slice a byte stream into individual packets (for delivery using IP, which is packet-oriented), consistency checks (packet checksums) and retransmission logic (in case IP packets carrying TCP data are lost).

TCP Connections

460

- the endpoints must establish a **connection** first
- each connection serves as a separate **data stream**
- a connection is **bidirectional**
- TCP uses a 3-way handshake: SYN, SYN/ACK, ACK

To provide stream semantics to the user, TCP must implement a mechanism which creates the illusion of a byte stream on top of a packet-based foundation. This mechanism is known as a **connection**, and essentially consists of some state shared by the two endpoints. To establish this shared state, TCP uses a 3-way handshake.

Sequence Numbers

461

- TCP packets carry **sequence numbers**
- these numbers are used to **re-assemble** the stream
 - IP packets can arrive **out of order**
- they are also used to **acknowledge reception**
 - and subsequently to manage re-transmission

Sequence numbers are part of the connection state, and allow the byte stream to be reassembled in the correct order, even if IP packets carrying the stream get reordered during delivery.

Packet Loss and Re-transmission

462

- packets can get **lost** for a variety of reasons
 - a **link goes down** for an extended period of time
 - **buffer overruns** on routing equipment
- TCP sends **acknowledgments** for received packets
 - the ACKs use **sequence numbers** to identify packets

Besides packet reordering, TCP also needs to deal with **packet loss**: an event where an IP packet is sent, but vanishes without trace en-route to its destination. A lost packet is detected as a gap in sequence numbers. However, it is the **sender** which must learn about a lost packet, so that it can be retransmitted: for this reason, the recipient of the packet must **acknowledge** its receipt, by sending a packet back (or more often, by piggybacking the acknowledgement on a data packet that it would send anyway), carrying the sequence numbers of packets that have been received.

If an acknowledgement is not received within certain time (dynamically adjusted) of the sending of the original packet, the packet is sent again (retransmitted).

UDP: User (Unreliable) Datagram Protocol

463

- TCP comes with non-trivial **overhead**
 - and its guarantees are **not always required**
- UDP is a much **simpler** protocol
 - a very thin wrapper around IP
 - with **minimal overhead** on top of IP

Not all applications need the comparatively strong guarantees that TCP provides, or conversely, cannot tolerate the additional latency introduced by the algorithms that TCP employs to ensure reliable, in-order delivery. For those cases, UDP presents a very light-weight layer on top of IP, essentially only adding the port number to the addresses, and a 16-bit checksum to the packet header (which is, in its entirety, only 64 bits long).

Firewalls

464

- the **name** comes from building construction
 - a fire-proof barrier between parts of a building
- the idea is to **separate networks** from each other
 - making attacks harder from the outside
 - **limiting damage** in case of compromise

Firewall is a device which separates two networks from each other, typically by acting as the (only) router between them, but also examining the packets and dropping or rejecting them if they appear malicious, or attempt to use services that are not supposed to be visible externally. Often, one of these networks is the internet. Sometimes, the other network is just a single computer.

Packet Filtering

465

- packet filtering is an **implementation** of a **firewall**
- can be done on a **router** or at an **endpoint**
- **dedicated** routers + packet filters are **more secure**
 - a **single** such **firewall** protects the **entire network**
 - less opportunity for mis-configuration

Like with other services, it usually pays off to centralize (within a single network) the responsibility for packet filtering, reducing the administrative burden and the space for misconfigured nodes to endanger the entire network. Of course, it is reasonable to run local firewalls on each node, as a second line of defence.

Packet Filter Operation

466

- packet filters operate on a set of **rules**
 - the rules are generally **operator**-provided
- each incoming packet is **classified** using the rules
- and then **dispatched** accordingly
 - may be **forwarded**, **dropped**, **rejected** or edited

A packet filter is, essentially, a finite state machine (perhaps with a bit of memory for connection tracking, in which case it is a **stateful** packet filter) which examines each packet and decides what action to take on it. The specific classification rules are usually provided by the network administrator; in simple cases, they match on source and destination IP addresses and port numbers, and on the connection status (which is remembered by the packet filter), for TCP packets.

After they are classified, the packets can be forwarded to their destination (as a standard router would), quietly dropped, rejected (sending an ICMP notification to the sender) or adjusted before being sent along (most commonly for network address translation, or NAT, the details of which are out of scope of this subject).

Packet Filter Examples

467

- packet filters are often part of the **kernel**
- the rule parser is a system utility
 - it loads rules from a **configuration file**
 - and sets up the kernel-side filter
- there are multiple **implementations**
 - **iptables**, **nftables** in Linux
 - **pf** in OpenBSD, **ipfw** in FreeBSD

There are usually two components to a packet filter: one is a system utility which reads a human-readable description of the rules, and

based on those, compiles an efficient matcher for use in the kernel component which does the actual classification.

Name Resolution

468

- users do not want to remember **numeric addresses**
 - phone numbers are bad enough
- host **names** are used instead
- can be stored in a file, e.g. **/etc/hosts**
 - not very practical for more than 3 computers
 - but there are millions of computers on the Internet

In the last part of this section, let's have a look at hostname resolution and the DNS protocol. What we need is a directory (a yellow pages sort of thing), but one that can be efficiently updated (many updates are done every hour) and also efficiently queried by computers on the network. The system must be scalable enough to handle many millions of names.

DNS: Domain Name System

469

- hierarchical **protocol** for name resolution
 - runs on top of TCP or UDP
- domain **names are split** into parts using dots
 - each domain knows whom to ask for the next bit
 - the name database is effectively **distributed**

Essentially, at the internet scale, we need some sort of a distributed system (i.e. a distributed database). Unlike relational databases though, delays in update propagation are acceptable, making the design simpler. The name space of host names is organized hierarchically, and the structure of DNS follows this organisation: going from right to left, starting with the top-level domain (a single dot, often left out), one of the DNS servers for that domain is consulted about the name immediately to the left, usually resulting in the address of another DNS server which can get us more information. The process is repeated until the entire name is resolved, usually resulting in an IP address of the host.

DNS Recursion

470

- take **www.fi.muni.cz.** as an example domain
- resolution starts from the right at **root servers**
 - the root servers refer us to the **cz.** servers
 - the **cz.** servers refer us to **muni.cz**
 - finally **muni.cz.** tells us about **fi.muni.cz**

The process described above is called **recursion** and is usually performed by a special type of DNS server, which performs the recursion on behalf of its clients and caches the results for subsequent queries. This also means that it can, most of the time, start from the middle, since the name servers of the one or two topmost domains are most likely in the cache

DNS Recursion Example

471

```
$ dig www.fi.muni.cz. A +trace
.                IN NS j.root-servers.net.
cz.              IN NS b.ns.nic.cz.
muni.cz.        IN NS ns.muni.cz.
fi.muni.cz.     IN NS aisa.fi.muni.cz.
www.fi.muni.cz. IN A 147.251.48.1
```

To observe recursion in practice (and perform other diagnostics on DNS), we can use the `dig` tool, which is part of the ISC (Internet Software Consortium) suite of DNS-related tools.

DNS Record Types

472

- `A` is for (IP) Address
- `AAAA` is for an IPv6 Address
- `CNAME` is for an alias
- `MX` is for mail servers
- and many more

Besides `NS` records, which tell the system whom to ask for further information, there are many types of DNS records, each carrying different type of information about the name in question. Besides IPv4 and IPv6 addresses, there are free-form `TXT` records (which are used, for instance, by spam filtering systems to learn about authorized mail servers for a domain), `SRV` records for service discovery in local networks, and so on.

Part 8.3: Using Networks

In this section, we will briefly look at the socket API which allows applications to use and provide network services (on POSIX operating systems, that is) and at a couple examples of application-level network services.

Sockets Reminder

474

- the `socket` API comes from early BSD Unix
- socket represents a (possible) `network connection`
- you get a `file descriptor` for an open socket
- you can `read()` and `write()` to sockets
 - but also `sendmsg()` and `recvmsg()`
 - and `sendto()` and `recvfrom()`

Remember that socket is a file-like object, accessible through a `file descriptor`. On connected stream sockets, programs can use the usual `read` and `write` system calls, with semantics akin to pipes. While these are also possible on datagram sockets, a different API is often preferred, one of the reasons being that with `read`, it is impossible to distinguish datagrams coming from different sources.

The system calls `sendto`, `recvfrom` allow the program to specify (or learn, in case of `recvfrom`) the address of the recipient (sender) of the packet.

Socket Types

475

- sockets can be `internet` or `unix domain`
 - internet sockets work across networks
- `stream` sockets are like files
 - you can write a continuous `stream` of data
 - usually implemented using TCP
- `datagram` sockets send individual `messages`
 - usually implemented using UDP

Communication on IP networks is done using `internet sockets` (with `domain` set to `AF_INET` or `AF_INET6`). If the socket is a `stream socket` (it's `type` is `SOCK_DGRAM`) the communication is executed using TCP (stream-type sockets must be explicitly `connected` by a call to the `connect` or `accept` system call, which in case of internet sockets perform the TCP handshake).

Datagram sockets (`type` set to `SOCK_DGRAM`) may be optionally 'connected', though this only sets up a default destination for datagrams to be sent

to. Communication is performed using UDP.

Creating Sockets

476

- a socket is created using the `socket()` function
- it can be turned into a `server` using `listen()`
 - individual `connections` are established with `accept()`
- or into a `client` using `connect()`

All types of sockets are created using the `socket` system call, and specialize into server and client sockets based on the subsequent API calls performed on them. A server socket is obtained through `listen` and `bind`, while a client socket is obtained using `connect`. The server then repeatedly calls `accept` which returns a `new file descriptor` which then represents the TCP connection.

Resolver API

477

- `libc` contains a `resolver`
 - available as `gethostbyname` (and `getaddrinfo`)
 - also `gethostbyaddr` for `reverse lookups`
- can look in many different places
 - most systems support at least `/etc/hosts`
 - and DNS-based lookups

The socket API only deals with numeric IP addresses. If an application needs to be able to connect to computers using their host names, it needs to use the `resolver API` which, behind the scenes, uses the appropriate database or protocol to find the corresponding IP addresses. The exact sequence of steps depends on system configuration, but usually the resolver consults the `/etc/hosts` file and a recursive DNS server (the IP address of which is again part of system configuration).

Network Services

478

- servers `listen` on a socket for incoming connections
 - a client actively establishes a `connection` to a server
- the network simply `transfers data` between them
- interpretation of the data is a `layer 7` issue
 - could be `commands`, file transfers, ...

Most network services operate in a client-server regime, on top of TCP: a server passively awaits connections on a particular transport-layer address (i.e. an IP address coupled with a port number). The client, on the other hand, actively connects to a listening server, establishing a bidirectional channel (the TCP connection) between them. From that point on, the network stack simply transfers data across that channel. The data usually conforms to some application-level protocol (SMTP, HTTP, ...) though it does not need to be standardized or well-known.

Network Service Examples

479

- (secure) remote shell - `ssh`
- the internet `email suite`
 - MTA = Mail Transfer Agent, speaks SMTP
 - SMTP = Simple Mail-Transfer Protocol
- the `world wide web`
 - web servers provide content (files)
 - clients and servers speak HTTP and HTTPS

Client Software

480

- the `ssh` command uses the SSH protocol
 - a very useful system utility on virtually all UNIXes
- `web browser` is the client for world wide web
 - browsers are complex `application` programs
 - some of them bigger than even operating systems
- `email client` is also known as a MUA (Mail User Agent)

Part 8.4: Network File Systems

We have learned earlier that file systems are an important, ubiquitous abstraction. It is only natural to allow a file system to be accessed remotely (from another computer) using the API that is used for local access, making the 'network' part almost entirely transparent to the program.

Why Network Filesystems?

482

- copying files back and forth is impractical
 - and also `error-prone` (which is the latest version?)
- how about storing data in a `central location`
- and `sharing` it with all the computers on the LAN

Perhaps the most compelling case for network file systems arises from the need to make workstations (desktop computers) at an institution fungible: that is, allow any user to log in onto any of the available workstations and immediately have all their data and settings at hand.

NAS (Network-Attached Storage)

483

- a (small) `computer` dedicated to `storing files`
- usually running a cut down operating system
 - often based on Linux or FreeBSD
- provides `file access` to the network
- sometimes additional `app-level services`
 - e.g. photo management, media streaming, ...

Another use case comes from the desire to store data which is shared by multiple users on a central device, where it is easy to back up and accessible from all computers (and hence by all users, even when some of the other computers are powered down).

NFS (Network File System)

484

- the traditional UNIX `networked filesystem`
- hooked quite deep into the kernel
 - assumes generally reliable network (LAN)
- filesystems are `exported` for use over NFS
- the client side `mounts` the NFS-exported volume

NFS is one of the first implementations of a network file system. It is based, essentially, on hooking up the VFS interface and exporting it over a remote procedure call interface to other kernels on the network. To create an NFS share, the local file system must be `exported` on the would-be NFS server; afterwards, it can be `mounted` by clients, making the share part of their local file system hierarchy.

NFS History

485

- originated in `Sun Microsystems` in the 80s
- v2 implemented in System V, DOS, ...
- v3 appeared in '95 and is `still in use`
- v4 arrives in 2000, improving `security`

Network file system is a rather old technology (nearly 40 years old), but it has seen significant evolution over the first 20 or so years, with version 4 mainly addressing security concerns.

VFS Reminder

486

- `implementation mechanism` for multiple FS types
- an object-oriented approach
 - `open`: `look up` the file for access
 - `read`, `write` – self-explanatory
 - `rename`: rename a file or directory

Recall, from lecture 4, that VFS (virtual file system switch) is a mechanism inside the kernel that allows multiple file system implementations to present a unified interface to the rest of the kernel. NFS takes advantage of this existing interface and makes it available over the network. Of course, unlike VFS itself, the semantics of the NFS functions is standardized across implementations (NFS clients and servers are mostly compatible across different UNIX-like operating systems).

RPC (Remote Procedure Call)

487

- any `protocol` for `calling functions` on `remote hosts`
 - ONC-RPC = Open Network Computing RPC
 - NFS is based on ONC-RPC (also known as Sun RPC)
- NFS basically runs VFS operations using RPC
 - `easy to implement` on UNIX-like systems

The way the NFS interface is exposed to the network is via a remote procedure call mechanism, which essentially takes a procedure call (the name of the function, along with the arguments that it should be called with), packs them into a byte string and sends it over the network to another computer, which then actually performs the call and sends the result back. The protocol has a mechanism to send data buffers, in addition to primitive values (integers).

Port Mapper

488

- ONC-RPC is executed over TCP or UDP
 - but it is more `dynamic` wrt. available services
- TCP/UDP `port numbers` are assigned `on demand`
- `portmap` translates from RPC services to port numbers
 - the port mapper itself listens on port 111

In modern systems, ONC-RPC is implemented exclusively on top of the TCP/IP stack. Since the protocol can expose multiple services on each machine, the need arises to translate between those RPC services and TCP/UDP port numbers. In most cases, an RPC service called 'portmapper' takes care of this need, itself running on a fixed port (number 111).

The NFS Daemon

489

- also known as `nfsd`
- provides NFS access to a **local file system**
- can run as a system service
- or it can be part of the kernel
 - this is more typical for **performance** reasons

Given an RPC stack, NFS is provided by an `nfsd`, which registers itself as a service with the RPC stack. The daemon can be a proper, user-space daemon, but it can also be part of the kernel (running as a kernel thread).

SMB (Server Message Block)

490

- a **network file system** from Microsoft
- available in Windows since version 3.1 (1992)
 - originally ran on top of NetBIOS
 - later versions used TCP/IP
- SMB1 accumulated a lot of cruft and **complexity**

SMB is a completely different implementation of a network transparency layer for file systems. Like NFS, it is not tied to a particular on-disk format. SMB saw many incremental changes with each new Microsoft operating system that came along, while at the same time it

was kept backward compatible, so that older operating systems could interoperate, both as clients and as servers. This made the protocol extremely complicated, making further extensions impractical.

SMB 2.0

491

- **simpler** than SMB1 due to **fewer retrofits** and **compat**
- better **performance** and **security**
- support for **symbolic links**
- available since Windows Vista (2006)

Microsoft designed a new protocol for networked filesystems in their Windows Vista operating system, under the name SMB 2.0. Like NFSv4 a few years earlier, SMB 2 addressed many of the security weaknesses of its predecessor, while also improving performance and extending the protocol to support new file system features, such as symlinks.

Review Questions

492

- What is ARP (Address Resolution Protocol)?
- What is IP (Internet Protocol)?
- What is TCP (Transmission Control Protocol)?
- What is DNS (Domain Name Service)?

Part 9: Shells & User Interfaces

This lecture will focus on human-computer interaction and the role of an operating system in this area. We will look at both text-based interaction modes (mainly command-line interfaces, i.e. **shells**) and at graphical interfaces, driven by a pointing device (mouse, trackpad) or a touch screen.

Lecture Overview

494

1. Command Interpreters
2. The Command Line
3. Graphical Interfaces

The first part will focus on **shell** as a simple programming language, while in the second we will briefly look at terminals (or rather terminal emulators), interactive use of **shell** and at other text-mode programs. Finally, the third part will be about graphical interfaces and how they are built.

Part 9.1: Command Interpreters

Shell

496

- **programming language** centered on OS interaction
- rudimentary **control flow**
- **untyped**, text-centered variables
- dubious error handling

Interactive Shells

497

- almost all shells have an **interactive mode**
- the user inputs a single statement on keyboard
- when confirmed, it is immediately **executed**
- this forms the basis of **command-line interfaces**

Shell Scripts

498

- a **shell script** is an (executable) file
- in simplest form, it is a **sequence of commands**
 - each command goes on a separate line
 - executing a script is about the same as typing it
- but can use **structured programming** constructs

Shell Upsides

499

- very easy to write simple scripts
- first choice for simple automation
- often useful to save repetitive typing
- definitely **not** good for big programs

Bourne Shell

500

- a specific language in the “shell” family
- the first shell with consistent programming support
 - available since 1976
- still widely used today
 - best known implementation is `bash`
 - `/bin/sh` is mandated by POSIX

The name `bash` stands for Bourne Again Shell (bad puns should be a human right).

C Shell

501

- also known as `csh`, first released in 1978
- more C-like syntax than `sh` (Bourne Shell)
 - but not really very C-like at all
- improved interactive mode (over `sh` from '76)
- also still used today (`tcsh`)

Korn Shell

502

- also known as `ksh`, released in 1983
- middle ground between `sh` and `csh`
- basis of the POSIX.2 requirements
- a number of implementations exist

Commands

503

- typically a name of an executable
 - may also be control flow or a built-in
- the executable is looked up in the filesystem
- the shell does a `fork + exec`
 - this means new process for each command
 - process creation is fairly expensive

Built-in Commands

504

- `cd` change the working directory
- `export` for setting up environment
- `echo` print a message
- `exec` replace the shell process (no `fork`)

Variables

505

- variable names are made of letters and digits
- `using` variables is indicated with `$`
- setting variables does **not** use the `$`
- all variables are global (except subshells)

```
VARIABLE="some text"
echo $VARIABLE
```

Variable Substitution

506

- `variables` are substituted as `text`
- `$foo` is simply replaced with the content of `foo`
- `arithmetic` is not well supported in most shells
 - or any expression syntax, e.g. relational operators
 - consider `z=$(($x + $y))` for addition in `bash`

Command Substitution

507

- basically like variable `substitution`
- written as ``command`` or `$(command)`
 - first `executes` the command
 - and captures its standard output
 - then replaces `$(command)` with the output

Quoting

508

- whitespace is an `argument separator` in shell
- multi-word arguments must be `quoted`
- quotes can be double quotes `"x"` or single `'x'`
 - double quotes allow variable `substitution`

Quoting and Substitution

509

- `whitespace` from variable substitution must be `quoted`
 - `foo="hello world"`
 - `ls $foo` is different than `ls "$foo"`
- bad quoting is a very common source of `bugs`
- consider also `filenames` with spaces in them

The first command, `ls $foo` will expand into `ls hello world` and execute with `argv[1] = "hello"` and `argv[2] = "world"`, in effect looking for two separate files. The latter, `ls "$foo"`, will be executed with `argv[1] = "hello world"`.

Special Variables

510

- `$?` is the result of last command
- `$$` is the PID of the current shell
- `$1` through `$9` are positional parameters
 - `$#` is the number of parameters
- `$0` is the name of the shell - `argv[0]`

Environment

511

- is `like` shell variables but not the same
- the environment is passed to `all` executed `programs`
 - a child cannot modify environment of its parent
- variables are moved into the environment by `export`
- environment variables often act as `settings`

Important Environment Variables

512

- `$PATH` tells the system where to find programs
- `$HOME` is the home directory of the current user
- `$EDITOR` and `$VISUAL` set which text editor to use
- `$EMAIL` is the email address of the current user
- `$PWD` is the current working directory

Globbering

513

- patterns for quickly **listing** multiple **files**
- e.g. `ls *.c` shows all files ending in `.c`
- `*` matches any number of characters
- `?` matches one arbitrary character
- works on entire **paths** – `ls src/*/*.c`

Conditionals

514

- allows **conditional execution** of commands
- `if cond; then cmd1; else cmd2; fi`
- also `elif cond2; then cmd3; fi`
- `cond` is also a command (the exit code is used)

`test` (evaluating boolean expressions)

515

- originally an **external program**, also known as `[`
 - nowadays **built-in** in most shells
 - works around lack of expressions in shell
- evaluates its arguments and returns **true** or **false**
 - can be used with `if` and `while` constructs

`test` Examples

516

- `test file1 -nt file2` → 'nt' = newer than
- `test 32 -gt 14` → 'gt' = greater than
- `test foo = bar` → string equality
- combines with variable substitution (`test $y = x`)

Loops

517

- `while cond; do cmd; done`
 - `cond` is a command, like in `if`
- `for i in 1 2 3 4; do cmd; done`
 - allows globs: `for f in *.c; do cmd; done`
 - also command substitution
 - `for f in `seq 1 10`; do cmd; done`

Case Analysis

518

- selects a command based on **pattern matching**
- `case $x in *.c) cc $x;; *) ls $x;; esac`
 - yes, `case` really uses unbalanced parens
 - the `;;` indicates end of a case

Command Chaining

519

- `;` (semicolon): run two commands in sequence
- `&&` run the second command **if** the first succeeded
- `||` run the second command **if** the first failed
- e.g. compile and run: `cc file.c && ./a.out`

Pipes

520

- shells can run **pipelines** of commands
- `cmd1 | cmd2 | cmd3`
 - all commands are run **in parallel**
 - output of `cmd1` becomes input of `cmd2`
 - output of `cmd2` is processed by `cmd3`

`echo hello world | sed -e s,hello,goodbye,`

Functions

521

- you can also define **functions** in shell
- mostly a light-weight **alternative** to **scripts**
 - no need to **export** variables
 - but cannot be invoked by non-shell programs
- functions can also **set** variables

Recall that the environment is only passed down, never back up. This means that a shell script setting a variable will not affect the parent shell. In functions (and when scripts are invoked using `.`), variables can be set as normal.

Part 9.2: The Command Line

Interactive Shell

523

- the shell displays a **prompt** and waits
- the user **types** in a **command** and hits enter
- the command is **executed** immediately
- **output** is printed to the **terminal**

Command Completion

524

- most shells let you use TAB to **auto-complete**
 - works at least for command names and file names
 - but “smart completion” is common
- interactive history: hit “up” to recall a command
 - also interactive history search, e.g. `C-r` in `bash`

Prompt

525

- the string printed when shell **expects a command**
- controlled by the `PS1` environment variable
- usually shows your **username** and the **hostname**
- or working **directory**, battery status, time, weather, ...

Job Control

526

- only one program can run in the **foreground** (terminal)
- but a running program can be **suspended** (C-z)
- and **resumed** in background (bg) or in foreground (fg)
- use **&** to run a command in background: `./spambot &`

Terminal

527

- can **print text** and read text from a **keyboard**
- normally everything is printed on the last line
- the text could contain **escape** (control) sequences
 - for printing colourful text or clearing the screen
 - also for printing text at a **specific coordinate**

Older text scrolls upwards: this is the mode used in normal shell usage. This scrollbar behaviour is automatic in the terminal. Full-screen terminal applications (which use coordinate-based printing) will not use the capability.

Terminal (emulator) is typically a program these days, but used to be a dedicated piece of hardware.

Full-Screen Terminal Apps

528

- applications can use the **entire terminal screen**
- a library abstracts away the low-level **control sequences**
 - the library is called **ncurses** for **new curses**
 - different terminals use different control sequences
- special characters exist to draw **frames** and **separators**

UNIX Text Editors

529

- **sed** – stream editor, non-interactive
- **ed** – line oriented, interactive
- **vi** – visual, screen oriented
- **ex** – line-oriented mode of **vi**

TUI: Text User Interface

530

- the program draws a **2D interface** on a terminal
- these types of interfaces can be quite comfortable
- they are often **easier to program** than GUIs
- very low bandwidth requirements for **remote use**

Part 9.3: Graphical Interfaces

Windowing Systems

532

- each application runs in its **own window**
 - or possibly multiple windows
- **multiple applications** can be shown on screen
- windows can be moved around, resized &c.
 - facilitated by frames around window content
 - generally known as **window management**

Window-less Systems

533

- especially popular on **smaller screens**
- applications take the entire screen
 - give or take status or control widgets
- **task switching** via a dedicated screen

A GUI Stack

534

- graphics card **driver**, mode setting
- **drawing/painting** (usually hardware-accelerated)
- multiplexing (e.g. using windows)
- **widgets**: buttons, labels, lists, ...
- **layout**: what goes where on the screen

Well-known GUI Stacks

535

- Windows
- macOS, iOS
- X11
- Wayland
- Android

Portability

536

- GUI “toolkits” make **portability** easy
 - Qt, GTK, Swing, HTML5+CSS, ...
 - many of them run on **all major platforms**
- **code** portability is not the only issue
 - GUIs come with **look and feel** guidelines
 - portable applications may **fail to fit**

Text Rendering

537

- a surprisingly **complex** task
- unlike terminals, GUIs use variable pitch fonts
 - brings up issues like **kerning**
 - hard to predict **pixel width** of a line
- bad interaction with **printing** (cf. WYSIWIG)

Bitmap Fonts

538

- characters are represented as **pixel arrays**
 - usually just black and white
- traditionally pixel-drawn **by hand**
 - very time consuming (many letters, sizes, variants)
- the result is **sharp** but **jagged** (not smooth)

Outline Fonts

539

- Type1, TrueType – based on **splines**
- they can be **scaled** to arbitrary pixel sizes
- same font can be used for **screen** and for **print**
- rasterisation is usually done in **software**

Hinting, Anti-Aliasing 540

- screens are **low resolution** devices
 - typical HD displays have DPI around 100
 - laser printers have DPI of 300 or more
- **hinting**: deform outlines to better fit a pixel grid
- **anti-aliasing**: smooth outlines using grayscale

X11 (X Window System) 541

- a traditional UNIX windowing system
- provides a C API (**xlib**)
- built-in **network transparency** (socket-based)
- core protocol version 11 from 1987

X11 Architecture 542

- X **server** provides graphics and input
- X **client** is an application that uses X
- a **window manager** is a (special) client
- a **compositor** is another special client

Remote Displays 543

- **application** is running on computer A
- the display is **not** the console of A
 - could be a dedicated **graphical terminal**
 - could be another **computer** on a LAN
 - or even across the internet

Remote Display Protocols 544

- one approach is **pushing pixels**
 - VNC (Virtual Network Computing)
- X11 uses a custom **drawing** protocol
- others use **high-level** abstractions
 - NeWS (PostScript-based)
 - HTML5 + JavaScript

VNC (Virtual Network Computing) 545

- sends **compressed pixel data** over the wire
 - can leverage regularities in pixel data
 - can send **incremental updates**
- and **input events** in the other direction
- no support for **peripherals** or file sync

Basically the only virtue of VNC is simplicity. Security is an afterthought and not super-compatible across implementations. It is mainly designed for low-bandwidth, high-latency networks (i.e. the Internet).

RDP (Remote Desktop Protocol) 546

- more sophisticated than VNC (but proprietary)
- can also send **drawing commands** over the wire
 - like X11, but using DirectX drawing
 - also allows remote **OpenGL**
- support for audio, remote USB &c.

RDP is primarily based on the pixel-pushing paradigm, but there is a number of extensions that allow sending high-level rendering commands for local, hardware-accelerated processing. In some setups, this includes remote accelerated OpenGL and/or Direct3D.

SPICE 547

- Simple Protocol for Independent Computing Env.
- open protocol somewhere between VNC and RDP
- can send OpenGL (but only over a **local socket**)
- two-way **audio**, USB, **clipboard** integration
- still mainly based on **pushing** (compressed) **pixels**

Remote Desktop Security 548

- the user needs to be **authenticated** over network
 - passwords are easy, biometric data less so
- the data stream should be **encrypted**
 - not part of the X11 or NeWS protocols
 - or even HTTP by default (used for HTML5/JS)

For instance, RDP in Windows 10 does not support fingerprint logins (it was supported on earlier versions, but was disabled due to security flaws).

Review Questions 549

- What is a shell?
- What does variable substitution mean?
- What is an environment variable?
- What belongs into the GUI stack?

Part 10: Access Control

This lecture will focus on basic security considerations in an operating system, with focus on file systems, which are typically the most visible instance of access control in an OS.

Lecture Overview 551

1. Multi-User Systems
2. File Systems
3. Sub-user Granularity

We will first look at the motivation and implementation of **users**, the basic unit of ownership and access control in an operating system. We will also look at some consequences and some applications of multi-user computing, and discuss how access control is implemented and enforced. In the second part, we will focus on the canonical case study in access control: file systems. Finally, the last part will explore what happens when per-user access control is not sufficient and we need a more granular permission system.

Part 10.1: Multi-User Systems

Multi-user systems had been the norm until the rise of personal computers circa mid-80s: earlier computers were too expensive and too bulky to be allocated to a single person. Instead, earlier systems used some form of multi-tenancy, whether implemented administratively (batch systems) or by the operating system (interactive, terminal-based computers).

Users

553

- originally a proxy for **people**
- currently a more **general abstraction**
- user is the unit of **ownership**
- many **permissions** are user-centered

The concept of a **user** has evolved from the need to keep separate accounts for distinct people (the eponymous users of the system). In modern systems, a **user** continues to be an abstraction that includes accounts for individual humans, but also covers other needs. Essentially, **user** is a unit of ownership, and of access control.

Computer Sharing

554

- computer is a (often costly) **resource**
- efficiency of use is a concern
 - a single user rarely exploits a computer fully
- data sharing makes access control a necessity

While efficient resource usage is what drove multi-tenancy of computer systems, it is the global shared file system that drove the requirement for access control: users do not necessarily wish to trust all other users of the system with access to their files.

Ownership

555

- various **objects** in an OS can be **owned**
 - primarily **files** and **processes**
- the owner is typically whoever **created** the object
 - though ownership can be **transferred**
 - restrictions usually apply

The standard model of access control in operating systems revolves around **ownership** of **objects**. Generally speaking, ownership of an object confers both rights (to manipulate the object) and obligations (owned objects count towards quotas). Depending on circumstances, object ownership may be transferred, either by the original owner, or by system administrators.

Process Ownership

556

- each **process** belongs to some user
- the process acts **on behalf** of the user
 - the process gets the same privilege as its owner
 - this both **constrains** and **empowers** the process
- processes are **active** participants

The perhaps most important ownership relationship is between users and their processes. This is because processes execute code on behalf of the user, and all actions a user takes on a system are mediated by some process or another. In this sense, processes act on behalf of their owner and the actions they perform are subject to any restrictions which apply to the user in question.

File Ownership

557

- each **file** also belongs to some user
- this gives **rights** to the **user** (or rather their processes)
 - they can **read** and **write** the file
 - they can **change permissions** or ownership
- files are **passive** participants

Like processes, files are objects which are subject to ownership. However, unlike processes, files are passive: they do not perform any actions. Hence in this case, ownership simply gives the owner certain rights to perform actions on the file (most importantly change access control rights pertaining to that file).

Access Control Models

558

- **owners** usually decide who can access their objects
 - this is known as **discretionary** access control
- in high-security environments, this is not allowed
 - known as **mandatory** access control
 - a central authority decides the policy

There are two main approaches to access control: the common **discretionary** model, where owners decide who can interact with their files (or other objects, as applicable) and **mandatory**, in which users are not trusted with matters of security, and decisions about access control are placed in the hands of a central authority.

In both cases, the operating system grants (or denies) access to object based on an **access control policy**: however, only in the latter case this policy can be thought of as a coherent, self-contained document (as opposed to a collection of rules decided by a number of uncoordinated users).

(Virtual) System Users

559

- users are an useful ownership **abstraction**
- various system services get their own 'fake' users
- this allows them to **own files** and **processes**
- and also **limit** their **access** to the rest of the OS

Users have turned out to be a really useful abstraction. It is common practice that services (whether system- or application-level) run under special users of their own. This means that these service can own files and other resources, and run processes under their own identity. Additionally, it means that those services can be restricted using the same mechanisms that apply to 'normal' users.

Principle of Least Privilege

560

- entities should have **minimum** privilege required
 - applies to **software** components
 - but also to **human** users of the system
- this **limits** the scope of **mistakes**
 - and also of security compromises

The **principle of least privilege** is an important maxim for designing secure systems: it tells us that, regardless of the subject and object combination, permissions should only be granted where there is genuine need for the subject to manipulate the particular object. The rationale is that mistakes happen, and when they do, we would rather limit their scope (and hence damage): mistakes cannot endanger objects which are inaccessible to the culprit.

Privilege Separation

561

- different parts of a system need different privilege
- least privilege dictates **splitting** the system
 - components are **isolated** from each other
 - they are given only the rights they need
- components **communicate** using very simple IPC

An important corollary of the principle of least privilege is the design pattern known as **privilege separation**. Systems which follow it are split into a number of independent components, each serving a small, well-defined and security-wise self-contained function. Each of these modules can be then isolated in their own little sandbox and communicate with the rest of the system through narrowly defined interfaces (usually built on some form of inter-process communication).

Process Separation

562

- recall that each process runs in its own **address space**
 - **shared memory** must be explicitly requested
- each **user** has a view of the **filesystem**
 - a lot more is shared by default in the filesystem
 - especially the **namespace** (directory hierarchy)

There is not much need for access control of memory: each process has their own and cannot see the memory of any other process (with small, controlled exceptions created through mutual consent of the two processes).

The file system is, however, very different: there is a global, shared namespace that is visible to all users and all processes. Moreover, many of the objects (files) are **meant** to be shared, in a rather ad-hoc fashion, either through 'well-known' paths (this being the case with many system files) or through passing paths around. Importantly, paths are **not** any sort of access token and in almost all circumstances, withholding a path does not prevent access to the object (paths can be easily discovered).

Access Control Policy

563

- there are 3 pieces of information
 - the **subject** (user)
 - the **action/verb** (what is to be done)
 - the **object** (the file or other resource)
- there are many ways to **encode** this information

We have mentioned earlier, that the totality of the rules that decide

which actions are allowed, and which disallowed, is known as an **access control policy**. In the abstract, it is a rulebook which answers questions of the form 'Is (subject) allowed to perform (action) on (object)?' There are clearly many different ways in which this rulebook can be encoded: we will look at some of the most common strategies later.

Access Rights Subjects

564

- in a typical OS those are (possibly virtual) **users**
 - sub-user units are possible (e.g. programs)
 - **roles** and **groups** could also be subjects
- the subject must be **named** (names, identifiers)
 - easy on a single system, **hard** in a **network**

The most common access control **subject** (at least when it comes to access policy **specification**), are, as was already hinted at, **users**, whether 'real' (those that stand in for people) or virtual (which stand for services).

In most circumstances, it must be possible to **name** the subjects, so that it's possible to refer to them in rules. Sometimes, however, rules can be directly attached to subjects, in which case there is no need for these subjects to have stable identifiers attached.

Access Rights Actions (Verbs)

565

- the available 'verbs' (actions) depend on **object** type
- a typical object would be a **file**
 - files can be **read**, **written**, **executed**
 - **directories** can be **searched** or **listed** or **changed**
- network connections can be established &c.

The particular choice of actions depends on the object type: each such type has a fixed list of actions, which correspond to operations, or variants of operations, that the operating system offers through its interfaces.

The actions may be affected the policy directly or indirectly – for instance, the **read** permission on a file is not enforced at the time a **read** call is performed: instead, it is checked at the time of **open**, with the provision that **read** can be only used on file descriptors that are **open for reading**. That is, the program is required to indicate, at the time of **open**, whether it wishes to read from the file.

Access Rights Objects

566

- anything that can be **manipulated** by **programs**
 - although not everything is subject to access control
- could be **files**, **directories**, **sockets**, shared **memory**, ...
- object **names** depend on their type
 - file paths, i-node numbers, IP addresses, ...

Like subjects, objects need to have names unless the pieces of policy relevant to them are directly attached to the objects themselves. However, in case of objects, this direct attachment is much more common: it is rather typical that an i-node embeds permission information.

Subjects in POSIX

567

- there are 2 types of **subjects**: **users** and **groups**
- each **user** can belong to **multiple groups**
- users are split into **normal** users and **root**
 - **root** is also known as the **super-user**

In POSIX systems, there are two basic types of subjects that can appear in the access control policy: users and groups. Since POSIX only covers access control for the file system, objects do not need to be named: their permissions are attached to the i-node.

A special user, known as `root`, represents the system administrator (also known as the super-user). This account is not subject to permission checking. Additionally, there is a number of actions (usually not attached to particular objects) which only the `root` user can perform (e.g. reboot the computer).

User and Group Identifiers

568

- users and groups are represented as **numbers**
 - this improves **efficiency** of many operations
 - the numbers are called **uid** and **gid**
- those numbers are valid on a **single computer**
 - or at most, a local network

In the access control policy, users and groups are identified by numbers (each user and each group getting a small, locally unique integer). Since these identifiers have a fixed size, they can be stored very compactly in i-nodes, and can be also very efficiently compared, both of which have been historically important considerations. Besides efficiency, the numeric identifiers also make the layout of data structures which carry them simpler, reducing scope for bugs.

User Management

569

- the system needs a **database** of **users**
- in a network, user **identities** often need to be **shared**
- could be as simple as a **text file**
 - `/etc/passwd` and `/etc/group` on UNIX systems
- or as complex as a distributed database

The user database serves two basic roles: it tells the system which users are authorized to access the system (more on this later), and it maps between human-readable user names and the numeric identifiers that the system uses internally.

In local networks, it is often desirable that all computers have the same idea about who the users are, and that they use the same mapping between their names and id's. LDAP and Active Directory are popular choices for centralised network-level user databases.

Changing Identities

570

- each **process** belongs to a particular **user**
- ownership is **inherited** across `fork()`
- **super-user** processes can use `setuid()`
- `exec()` can sometimes change a process owner

Recall that all processes are created using the `fork` system call, with the exception of `init`. When a process forks, the child process inherits the ownership of the parent, that is, it belongs to the same user as the parent does (whose ownership is not affected by `fork`).

However, if a process is owned by the super-user, it can change its owner by using the `setuid` system call. Additionally, `exec` can sometimes change the owner of the process, via the so-called `setuid` bit (not to be confused with the system call of the same name). The `init` process is owned by the super-user.

Login

571

- a super-user process manages **user logins**
- the user types in their name and **password**
 - the `login` program **authenticates** the user
 - then calls `setuid()` to change the process owner
 - and uses `exec()` to start a shell for the user

You may recall that at the end of the boot process, a `login` process is executed to allow users to authenticate themselves and start a session. The traditional implementation of `login` first asks the user for their user name and password, which it checks against the user database. If the credentials match, the `login` program sets up the basic environment, changes the owner of the process to the user who just authenticated themselves and executes their preferred shell (as configured in the user database).

User Authentication

572

- the user needs to **authenticate** themselves
- **passwords** are the most commonly used method
 - the **system** needs to recognize the right password
 - user should be able to change their password
- **biometric** methods are also quite popular

By far, the most common method of authenticating users (that is, ascertaining that they are who they claim they are) is by asking for a secret – a password or a passphrase. The idea is that only the legitimate owner of the account in question knows this secret.

In an ideal case, the system does not store the password itself (in case the password database is compromised), but stores instead information that can be used to check that a password that the user typed in is correct. The usual way this is done is via (salted) cryptographic hash functions.

Besides passwords, other authentication methods exist, most notably cryptographic tokens and biometrics.

Remote Login

573

- authentication over **network** is more complicated
- **passwords** are easiest, but not easy
 - **encryption** is needed to safely transmit passwords
 - along with **computer authentication**
- **2-factor** authentication is a popular improvement

While password is simply short string that can be quite easily sent across a network, there are caveats. First, the network itself is often insecure, and the password could be snooped by an attacker. This means we need to use cryptography to transmit the password, or otherwise prove its knowledge.

The other problem is, in case we send an encrypted password, that the computer at the other end may not be the one we expect (i.e. it could belong to an attacker).

Since the user is not required to be physically present to attempt authenticating, this significantly increases the risk of attacks, making strong passwords much more important. Besides strong passwords, security can be improved by 2-factor authentication (more on this shortly).

Computer Authentication

574

- how to ensure we send the password to the **right party**?
 - an attacker could **impersonate** our remote computer
- usually via **asymmetric cryptography**
 - a private key can be used to **sign** messages
 - the server signs a challenge to establish its **identity**

When interacting with a remote computer (via a network), it is rather important to ensure that we communicate with the computer that we intended to. While the most immediate concern is sending passwords, of course this is not the only concern: accidentally uploading secret data to the wrong computer would be as bad, if not worse.

A common approach, then, is that each computer gets a unique private key, while its public counterpart (or at least its fingerprint) is distributed to other computers. When connecting, the client can generate a random challenge, and ask the remote computer to sign in using the secret key associated to the computer that we intended to contact, in order to prove its identity. Unless the target computer itself has been compromised, an attacker will be unable to produce a valid signature and will be foiled.

2-factor Authentication

575

- 2 different types of authentication
 - harder to spoof **both** at the same time
- there are a few factors to pick from
 - something the user **knows** (password)
 - something the user **has** (keys, tokens)
 - what the user **is** (biometric)

Two-factor (or multi-factor) authentication is popular for remote authentication (as outlined earlier), since networks make attacks much cheaper and more frequent. In this case, the first factor is usually a password, and the second factor is a cryptographic **token** – a small device (often in the form of a keychain) which generates a unique sequence of codes, one of which the user transcribes to prove ownership of the token. Remote biometric authentication is somewhat less practical (though not impossible).

Of course, two-factor authentication can be used locally too, in which case biometrics become considerably more attractive. Cryptographic tokens or smart cards are also common, though in the local case, they usually communicate with the computer directly, instead of relying on the user to copy a code.

Enforcement: Hardware

576

- all **enforcement** begins with the hardware
 - the CPU provides a **privileged mode** for the kernel
 - DMA memory and IO instructions are **protected**
- the MMU allows the kernel to **isolate processes**
 - and protect its own integrity

Now that we have an access control policy and we have established the identity of the user, there is one last thing that needs to be addressed, and that is **enforcement** of the policy. Of course, an access control policy is useless if it can be circumvented.

The ability of an operating system to enforce security stems from hardware facilities: software alone cannot sufficiently constrain other software running on the same computer. The main tool that allows the kernel to enforce its security policy is the MMU (and the fact that only the kernel can program it) and its control over interrupt handlers.

Enforcement: Kernel

577

- kernel uses **hardware facilities** to implement security
 - it stands between **resources** and **processes**
 - access is mediated through **system calls**
- **file systems** are part of the kernel
- **user** and **group abstractions** are part of the kernel

Hardware resources are controlled by the kernel: memory via the MMU, processors via the timer interrupt, memory-mapped peripherals again through the MMU and through the interrupt handler table. Since user programs cannot directly access physical resources, any interaction with them must go through the kernel (via system calls), presenting an opportunity for the kernel to check the requested actions against the policy.

Enforcement: System Calls

578

- the kernel acts as an **arbitrator**
- a process is trapped in its own **address space**
- processes use system calls to access resources
 - kernel can decide what to allow
 - based on its **access control model** and **policy**

When a system call is executed, the kernel knows the owner of that process, and also any objects involved in the system call. Armed with this knowledge, it can easily consult the access control policy to decide whether the requested action is allowed, and if it is not, return an error to the process, instead of performing the action.

Enforcement: Service APIs

579

- userland processes can enforce access control
 - usually system services which provide IPC API
- e.g. via the **getpeereid()** system call
 - tells the caller **which user** is **connected** to a socket
 - user-level access control relies on **kernel** facilities

Just as the kernel sits on resources that user programs cannot directly access, the same can principle can be applied in userspace programs, especially services.

Probably the most illustrative example is a relational database: the database engine runs under a dedicated (virtual) user and stores its data in a collection of files. The permissions on those files are set such that only the owner can read or write them – hence, the kernel will disallow any other process from interacting with those files directly. Nonetheless, the database system can selectively allow other programs to interact with the data it stores: the programs connect to a database server using a UNIX socket. At this point, the database can ask the operating system to provide the user identifier under which the client is running (using **getpeereid**).

Since the server can directly access the files which store the data, and hence can, on the behalf of the client, execute queries and return the results. It can, however, also disallow certain queries based on its own access control policy and the user id of the client.

Part 10.2: File Systems

File Access Rights

581

- file systems are a case study in access control
- all modern file systems maintain permissions
 - the only extant exception is FAT (USB sticks)
- different systems adopt different representation

Representation

582

- file systems are usually object-centric
 - permissions are attached to individual objects
 - easily answers “who can access this file”?
- there is a fixed set of verbs
 - those may be different for files and directories
 - different systems allow different verbs

The UNIX Model

583

- each file and directory has a single owner
- plus a single owning group
 - not limited to those the owner belongs to
- ownership and permissions are attached to i-nodes

Access vs Ownership

584

- POSIX ties ownership and access rights
- only 3 subjects can be named on a file
 - the owner (user)
 - the owning group
 - anyone else

Access Verbs in POSIX File Systems

585

- read: read a file, list a directory
- write: write a file, link/unlink i-nodes to a directory
- execute: exec a program, enter the directory
- execute as owner (group): setuid/setgid

Permission Bits

586

- basic UNIX permissions can be encoded in 9 bits
- 3 bits per 3 subject designations
 - first comes the owner, then group, then others
 - written as e.g. `rwxr-x---` or `0750`
- plus two numbers for the owner/group identifiers

Changing File Ownership

587

- the owner and root can change file owners
- `chown` and `chgrp` system utilities
- or via the C API
 - `chown()`, `fchown()`, `fchownat()`, `lchown()`
 - same set for `chgrp`

Changing File Permissions

588

- again available to the owner and to root
- `chmod` is the user space utility
 - either numeric argument: `chmod 644 file.txt`
 - or symbolic: `chmod +x script.sh`
- and the corresponding system call (numeric-only)

setuid and setgid

589

- special permissions on executable files
- they allow `exec` to also change the process owner
- often used for granting extra privileges
 - e.g. the `mount` command runs as the super-user

Sticky Directories

590

- file creation and deletion is a directory permission
 - this is problematic for shared directories
 - in particular the system `/tmp` directory
- in a sticky directory, different rules apply
 - new files can be created as usual
 - only the owner can unlink a file from the directory

Access Control Lists

591

- ACL is a list of ACE's (access control elements)
 - each ACE is a subject + verb pair
 - it can name an arbitrary user
- ACL is attached to an object (file, directory)
- more flexible than the traditional UNIX system

ACLs and POSIX

592

- part of POSIX.1e (security extensions)
- most POSIX systems implement ACLs
 - this does not supersede UNIX permission bits
 - instead, they are interpreted as part of the ACL
- file system support is not universal (but widespread)

Device Files 593

- UNIX represents **devices** as **special i-nodes**
 - this makes them subject to normal **access control**
- the particular device is described in the **i-node**
 - only a **super-user** can create device nodes
 - users could otherwise gain access to any device

Sockets and Pipes 594

- **named** sockets and pipes are just **i-nodes**
 - also subject to standard file permissions
- especially useful with **sockets**
 - a service sets up a **named socket** in the file system
 - **file permissions** decide who can talk to the service

Special Attributes 595

- flags that allow **additional restrictions** on file use
 - e.g. **immutable** files (cannot be changed by anyone)
 - **append-only** files (for logfile integrity protection)
 - compression, copy-on-write controls
- **non-standard** (Linux **chattr**, BSD **chflags**)

Network File System 596

- NFS 3.0 simply transmits numeric **uid** and **gid**
 - the numbering needs to be **synchronised**
 - can be done via a **central user database**
- NFS 4.0 uses **per-user** authentication
 - the user authenticates to the server directly
 - filesystem **uid** and **gid** values are mapped

File System Quotas 597

- **storage space** is limited, **shared** by users
 - files take up storage space
 - file ownership is also a **liability**
- **quotas** set up **limits** space use by users
 - exhausted quota can lead to **denial** of **access**

Removable Media 598

- access control at **file system** level makes no sense
 - other computers may choose to **ignore** permissions
 - **user names** or id's would not make sense anyway
- option 1: **encryption** (for denying reads)
- option 2: **hardware-level** controls
 - usually read-only vs read-write on the entire medium

The **chroot** System Call 599

- each process in UNIX has its own **root directory**
 - for most, this coincides with the **system root**
- the root directory can be changed using **chroot()**
- can be useful to **limit** file system **access**
 - e.g. in **privilege separation** scenarios

Uses of **chroot** 600

- **chroot** alone is **not** a security mechanism
 - a super-user process can **get out** easily
 - but not easy for a **normal user** process
- also useful for **diagnostic** purposes
- and as lightweight alternative to **virtualisation**

Part 10.3: Sub-User Granularity

In this section, we will explore a few cases where a more precise notion of an access control subject is required or useful.

Users are Not Enough 602

- users are not always the right abstraction
 - **creating users** is relatively **expensive**
 - only a super-user can create new users
- you may want to include **programs** as **subjects**
 - or rather, the combination user + program

One of the main drawbacks of the user-centric security paradigm is heavyweight and requires super-user privileges. Moreover, normal users cannot easily constrain processes under auxiliary users (only via a **setuid** helper, which must again be configured by the **root** user).

A natural extension of the concept of an **access control subject** is to include the currently running program in the description – allowing the policy to say things like **/home/xuser/mail** can be accessed by thunderbird (a mail client) running under the account of **xuser**, but not by firefox (a web browser) running under the same account.

Naming Programs 603

- users have user names, but how about programs?
- option 1: cryptographic **signatures**
 - **portable** across computers but **complex**
 - establishes **identity** based on the **program itself**
- option 2: i-node of the **executable**
 - simple, local, identity based on **location**

Unfortunately, attaching policy rules to programs is much harder than it is for files or users, since their identity is rather elusive. There might be any number of programs called thunderbird, some of which may be different versions or builds of the same software, but some might just claim to be thunderbird to get to one's email.

A fairly good, if complicated, solution is to embed a cryptographic signature into executables, stating the rough equivalent of 'this program is Firefox, signed by Mozilla'. Assuming we trust Mozilla (we probably do since we run their software), we can refer to 'Firefox by Mozilla' in our access control policy. A variation of this approach is used by mobile operating systems, like Android and iOS.

The other option, much simpler, is to add a note like 'this program

is Firefox' to the i-node of the executable. This approach is used by systems like SELinux (where the note is realized as a **security label**).

Program as a Subject

604

- program: passive (file) vs active (processes)
 - only a **process** can be a subject
 - but program **identity** is attached to the file
- rights of a **process** depend on its **program**
 - `exec()` will change privileges

Now that we have managed to delineate what is a program and how to identify it, a new problem pops up: in both cases, we have attached the identity to a file, but it actually belongs to a process. However, processes being much more dynamic than files, assigning identifiers to them is even less practical. In this case, we can use the same trick that was used for `setuid` programs: the `exec` system call can examine the binary and adjust the privileges of the process accordingly.

Mandatory Access Control

605

- delegates permission control to a **central authority**
- often coupled with **security labels**
 - classifies **subjects** (users, processes)
 - and also **objects** (files, sockets, programs)
- the owner **cannot** change object permissions

Security labels are, in some sense, a generalisation of user groups. They can be attached to both objects and subjects, and can `exec` will update the labels attached to a process based on the labels attached to the executable (file).

Under mandatory access control, the users are not allowed to change permissions on objects. However, in practical systems, both modes are usually combined: discretionary permissions are attached to files as usual, and applied to an action whenever the mandatory rules alone would have allowed it.

Capabilities

606

- not all verbs (actions) need to take objects
- e.g. shutting down the computer (there is only one)
- mounting file systems (they can't be always named)
- listening on ports with number less than 1024

The term 'capabilities' is often used to mean one of two forms of access control policy rules:

1. where the object is a singleton, i.e. there is only a single object for the given action, or
2. where it is impractical to name the objects or to attach permission information to them.

Dismantling the `root` User

607

- the traditional `root` user is **all-powerful**
 - "all or nothing" is often unsatisfactory
 - violates the principle of least privilege
- many special properties of `root` are capabilities
 - `root` then becomes the user with all capabilities
 - other users can get selective privileges

In many cases, the simple split between `root` and normal users (which, incidentally, mirrors the split between the kernel and user programs) is inadequate. There are three principal ways to address this:

1. `setuid` programs can extend some of the special `root`-only privileges to normal users (e.g. `mount`, `passwd`),
2. the system of **capabilities** adds the option of allowing certain users to perform some of the restricted operations,
3. the user-level approach mentioned at the end of section 1, where the service runs under `root` (e.g. PolicyKit).

Security and Execution

608

- security hinges on what is **allowed to execute**
- **arbitrary code execution** are the worst exploits
 - this allows **unauthorized** execution of code
 - same effect as **impersonating** the user
 - almost as bad as stolen credentials

Control over which code can execute (and with what privileges) is at the center of all access control restrictions. If a program can be tricked into executing code supplied by an attacker, all the privileges that the program had are automatically available to the attacker as well.

Untrusted Input

609

- programs often process **data** from **dubious sources**
 - think image viewers, audio & video players
 - archive extraction, font rendering, ...
- bugs in programs can be **exploited**
 - the program can be **tricked** into **executing data**

The most common way programs can be hijacked in this manner is through improper processing of **untrusted inputs**, that is, content coming from untrustworthy sources. If unexpected input data can derail program execution, this opens the door for an attacker to take control of the program.

The payload (the code that the attacker wants executed) is usually supplied as part of the input, and hence is normally treated as data by the program. However, in presence of certain bug, the program can be tricked into executing (or interpreting) this data as code.

Process as a Subject

610

- some privileges can be tied to a particular **process**
 - those only apply during the **lifetime** of the process
 - often **restrictions** rather than privileges
 - this is how **privilege dropping** is done
- restrictions are **inherited** across `fork()`

Programs (or parts of programs running in a separate process) can ask the operating system to remove some of their privileges (like file system access, network access, and so on). There are many ways to do this, though they are not very portable (i.e. they depend on non-POSIX features of particular operating systems, e.g. Linux user namespaces, `seccomp`, FreeBSD Capsicum, OpenBSD `pledge` and `unveil` and so on). One of the few portable approaches, known as privilege drop, is essentially a subset of privilege separation: a special user is created for the particular process and the process, after having done any privileged initialization operations that it needed to do, uses `setuid` and perhaps `chroot` to lock itself down.

Sandboxing

611

- tries to **limit damage** from code execution **exploits**
- the program **drops** all privileges it can
 - this is done **before** it touches any of the **input**
 - the attacker is stuck with the **reduced privileges**
 - this can often prevent a successful attack

Sandboxing is a collection of techniques (including some of the above) that tries to minimize the impact of a successful exploit against a program. Sandboxing can be voluntary (the program sets up its own sandbox) and involuntary (see also next slide).

Untrusted Code

612

- traditionally, you would only execute **trusted** code
 - often based on **reputation** or other **external** factors
 - this does not **scale** to a large number of vendors
- it is common to execute **untrusted**, even dubious code
 - this can be okay with sufficient **sandboxing**

Running code from questionable sources is always risky, but is essentially guaranteed to result in a compromise unless precautions are taken. However, since the modern web is full of executable code, we simply resort to locking it down as much as we can and hope for the best.

API-Level Access Control

613

- capability system for **user-level resources**
 - things like contact lists, calendars, bookmarks
 - objects not provided directly by the kernel
- enforcement e.g. via a **virtual machine**
 - not applicable to execution of **native code**
 - alternative: an IPC-based API

Selectively granting permissions to programs through user-level permission systems is also possible for non-root users. There are two commonly employed methods:

1. a (program-level) virtual machine, like the JVM or the javascript virtual machines built into web browsers, which enforce that the program only talks to the system through restricted APIs,
2. a strict sandbox with the only access to the system provided by a daemon running on the outside of the sandbox (e.g. snap and flatpak, to a degree).

Both approaches can be combined, with a common technique locking a VM using OS-level sandboxing to defend against security bugs in the VM itself.

Android/iOS Permissions

614

- applications from a store are **semi-trusted**
- typically **single-user** computers/devices
- permissions are attached to **apps** instead of users
- partially virtual users, partially API-level

On Android, for instance, each application gets its own virtual user with very limited permissions and interaction with the system is done almost exclusively through high-level APIs. These APIs then perform permission checks, possibly prompting the user for confirmation as needed.

Review Questions

615

- What is a user?
- What is the principle of least privilege?
- What is an access control object?
- What is a sandbox?

Part 11: Virtualisation & Containers

This lecture will focus on running multiple operating systems on the same physical computer. Until now, we have always assumed that the operating system (in particular the kernel) has direct control over physical resources. This week, we will see that this does not always need to be the case (in fact, it is increasingly rare in production systems). Instead, we will see that multiple operating systems may share a single computer in a manner similar to how multiple applications (processes) co-exist within an operating system.

We will also explore a compromise approach, known as **containers**, where only the user-space parts of the operating system are duplicated and isolated from each other, while the kernel remains shared and retains direct control of the underlying machine.

will discuss **containers**. Finally, we will look at a few topics which are common to both systems, and in some sense are also relevant when managing networks of physical computers.

Part 11.1: Hypervisors

In the domain of hardware-accelerated virtualisation, a **hypervisor** is the part of the VM software that is roughly equivalent to an operating system kernel.

Lecture Overview

617

1. Hypervisors
2. Containers
3. Management

The lecture is split into 3 parts: first part will introduce full-blown virtualisation and the concept of a **hypervisor**, while the second part

What is a Hypervisor

619

- also known as a Virtual Machine Monitor
- allows execution of **multiple operating systems**
- like a kernel that runs kernels
- improves **hardware utilisation**

While hypervisor itself behaves a bit like a kernel, standing as it does between the hardware and the virtualised operating systems, the virtualised operating systems running on top are, in a sense, like processes (including their kernels). In particular, they are isolated in physical

memory (by using either regular MMU and a bit of software magic, or using an MMU capable of second-level translation) and they time-share on the available processors.

Motivation

620

- OS-level sharing is tricky
 - **user isolation** is often **insufficient**
 - only **root** can install software
- the hypervisor/OS interface is **simple**
 - compared to OS-application interfaces

Virtualised operating systems allow a degree of autonomy that is not usually possible when multiple users share a single operating system. This is partially due to the simplicity of the interface between the hypervisor and the operating system: there are no file systems, in fact no communication between the operating systems (other than through standard networking), no user management and so on. Virtual machines simply bundle up some resources and make them available to the operating system.

Virtualisation in General

621

- many resources are “virtualised”
 - physical **memory** by the MMU
 - **peripherals** by the OS
- makes **resource management** easier
- enables **isolation** of components

Operating systems (or computers, if you prefer) are of course not the only thing that can be (or is) virtualised. If you think about it, a lot of operating system itself is built around some sort of virtualisation: virtual memory, file systems, network stack, device drivers – they all, in some sense, virtualise hardware resources. This in turn makes it possible for multiple programs, and multiple users, to share those resources safely and fairly.

Hypervisor Types

622

- type 1: bare metal
 - standalone, microkernel-like
- type 2: hosted
 - runs on top of normal OS
 - usually need **kernel support**

There are two basic types of hypervisors, based on how the overall system is layered. In type 1, the hypervisor is at the bottom of the stack (just above hardware), and is responsible for management of the basic resources (a bit like a simple microkernel): processor and RAM (scheduling and memory management, respectively).

On the other hand, type 2 hypervisors run on top of an operating system and reuse its scheduler and memory management: the virtual machines appear as actual processes of the host system.

Type 1 (Bare Metal)

623

- IBM z/VM
- (Citrix) Xen
- Microsoft Hyper-V
- VMWare ESX

Type 2 (Hosted)

624

- VMWare (Workstation, Player)
- Oracle VirtualBox
- Linux KVM
- FreeBSD bhyve
- OpenBSD vmm

History

625

- started with mainframe computers
- IBM CP/CMS: 1968
- IBM VM/370: 1972
- IBM z/VM: 2000

The first foray into running multiple operating systems on the same hardware was made by IBM in the late 60s and was made, on big iron, a rather standard feature soon after.

Desktop Virtualisation

626

- **x86** hardware lacks **virtual supervisor mode**
- **software-only** solutions viable since late 90s
 - Bochs: 1994
 - VMWare Workstation: 1999
 - QEMU: 2003

Small (personal) computers, for a long time, did not offer any OS virtualisation capabilities. Performance of PC processors became sufficient to do PC-on-PC emulation in mid-90s, but the performance penalty was initially huge and was only suitable to run legacy software (which was designed for much slower hardware).

Paravirtualisation

627

- introduced as VMI in 2005 by VMWare
- alternative approach in Xen in 2006
- relies on **modification** of the **guest OS**
- near-native speed without HW support

A decade later, VMWare has made a breakthrough in software-based virtualisation technology, by inventing paravirtualisation: this required modifications to the guest operating system, but by the time, open-source operating systems were gaining a foothold – and porting open-source systems to a paravirtualising hypervisor was not too hard.

The Virtual x86 Revolution

628

- 2005: virtualisation extensions on **x86**
- 2008: MMU virtualisation
- **unmodified** guest at near-native speed
- most **software-only** solutions became **obsolete**

Around the same time, vendors of desktop CPUs started to incorporate virtualization extensions, which in turn made it unnecessary to modify the guest operating system (at least in principle). By 2008, mainstream desktop processors offered MMU virtualisation, further simplifying x86 hypervisor design (and making it more efficient at the same time).

Paravirtual Devices

629

- special **drivers** for **virtualised devices**
 - block storage, network, console
 - random number generator
- **faster** and **simpler** than emulation
 - orthogonal to CPU/MMU virtualisation

However, paravirtualisation made a quick and dramatic comeback: while virtualisation of CPU and memory was, for the most part, handled by the hardware itself, a hardware-based approach is not economical for virtualisation of peripherals.

Additionally, paravirtualised peripherals do not need changes in the guest operating system: all that is required is a quite regular device driver that targets the respective protocol. The virtual peripherals offered by the host system then simply appear as regular devices through an appropriate device driver running in the guest.

Virtual Computers

630

- usually known as Virtual Machines
- everything in the computer is virtual
 - either via hardware (VT-x, EPT)
 - or software (QEMU, **virtio**, ...)
- much **easier to manage** than actual hardware

The entire system running under a virtualised operating system is known as a virtual machine (or, sometimes, a virtual computer), not to be confused with program-level VMs like the Java Virtual Machine.

Essential Resources

631

- the CPU and RAM
- persistent (block) storage
- network connection
- a console device

A typical virtual machine will offer at least a processor, memory, block storage (on which the operating system will store a file system), a network connection and a console for management. While other peripherals are possible, they are not very common, at least not on servers.

CPU Sharing

632

- same principle as normal **processes**
- there is a **scheduler** in the hypervisor
 - simpler, with different trade-offs
- privileged instructions are trapped

Most instructions (specifically those available to user-space programs) are simply executed without additional overhead by the host CPU, without direct involvement of the hypervisor. However, the hypervisor does manage the virtualised MMU. However, just as importantly, when the CPU encounters certain types of privileged instructions, it will invoke the hypervisor to perform the required actions in software.

RAM Sharing

633

- very similar to standard **paging**
- software (shadow paging)
- or hardware (second-level translation)
- fixed amount of RAM for each VM

Like CPU virtualisation, memory sharing is built on the same basic principles that standard operating systems use to isolate processes from each other. Memory is sliced into pages and the MMU does the heavy lifting of address translation.

Shadow Page Tables

634

- the **guest** system **cannot** access the MMU
- set up **shadow table**, invisible to the guest
- guest page tables are sync'd to the sPT by VMM
- the gPT can be made read-only to cause traps

The trap can then synchronise the gPT with the sPT, which are translated versions of each other. The 'physical' addresses stored in the gPT are virtual addresses of the hypervisor. The sPT stores real physical addresses, since it is used by the real MMU.

Second-Level Translation

635

- hardware-assisted MMU virtualisation
- adds guest-physical to host-physical layer
- greatly **simplifies** the VMM
- also much **faster** than shadow page tables

Shadow page tables cause a lot of overhead, trapping every change of the guest page table into the hypervisor. Unfortunately, page tables are rearranged by the guest operating system rather often (on real hardware, this is comparatively cheap).

However, modern processors offer another level of translation, which is inaccessible to the guest operating system. Since the MMU is aware of virtualisation, the guest can directly modify its page tables, without compromising isolation of VMs from each other (and from the hypervisor).

Network Sharing

636

- usually a paravirtualised NIC
 - transports **frames** between guest and host
 - usually connected to a **SW bridge** in the host
 - alternatives: routing, NAT
- a single physical NIC is used by everyone

In contemporary virtualisation solutions, networking uses a paravirtual NIC (network interface card) which is connected to an Ethernet tunnel pseudo-device in the host system (essentially a virtual network interface card that handles Ethernet frames). The frames sent on the paravirtual device appears on the virtual NIC in the host and vice versa. The pseudo-device is then either software-bridged to the hardware NIC (and hence to the outside ethernet), or alternatively, routing (layer 3) is set up between the pseudo-device and the hardware NIC.

Virtual Block Devices

637

- usually also paravirtualised
- often backed by normal **files**
 - maybe in a special format
 - e.g. based on **copy-on-write**
- but can be a real **block device**

Like networking, block storage is typically based on paravirtualisation. In this case, the host side of the device is either backed by a regular file in the file system of the host, or sometimes it is backed by a block device on the same (often virtualised, e.g. through LVM/device-mapper or similar technology, but sometimes backed directly by a hardware block device).

Special Resources

638

- mainly useful in **desktop systems**
- GPU / graphics hardware
- audio equipment
- printers, scanners, ...

Now that we have covered the essentials, let's briefly look at other classes of hardware. However, with the possible exception of compute GPUs, peripherals are only useful on desktop systems, which are a tiny market compared to server virtualisation.

PCI Passthrough

639

- an anti-virtualisation technology
- based on an IO-MMU (VT-d, AMD-Vi)
- a **virtual** OS can touch **real** hardware
 - only one OS at a time, of course

Let's first mention a very generic, but very un-virtualisation method of giving hardware access to a virtual machine, that is, exposing a PCI device to the guest operating system directly, via IO-MMU-mapped memory. An IO-MMU must be involved, because otherwise the guest OS could direct the hardware to overwrite physical memory that belongs to the host, or to another VM running on the same system. With that covered, though, there is nothing that stops the host system from handing over control of specific PCI endpoints to a guest (of course, the host system must not attempt to communicate with those devices though its own drivers, else chaos would ensue).

GPUs and Virtualisation

640

- can be **assigned** (via VT-d) to a **single OS**
- or **time-shared** using native drivers (GVT-g)
- paravirtualised
- shared by other means (X11, SPICE, RDP)

Of course, since a GPU is attached through PCI, it can be shared using the IO-MMU (VT-d) approach described above. However, modern GPUs all support time-sharing (i.e. they allow contexts to be suspended and resumed, just like threads and processes on a CPU). For this to work, the hypervisor (or the host OS) must provide drivers for the GPU in question, so that it can mediate access to individual VMs.

Another solution, is paravirtualisation: the guest uses a vendor-neutral protocol to send a command stream to the driver running in the hypervisor, which in turn does the multiplexing. The guest system still needs the userspace part of the GPU driver to generate the command

stream and to compile shaders.

Finally, existing network graphics protocols can be, of course, used between a guest and the host, though they are never quite as efficient as one of the specialised options.

Peripherals

641

- useful either via **passthrough**
 - audio, webcams, ...
- or **standard sharing** technology
 - network printers & scanners
 - networked audio servers

Finally, there is a wide array of peripherals that can be attached to a PC. Some of them, like printers and scanners, and in some cases (or rather, in some operating systems) audio hardware, can be shared over standard networks, and hence also between guests and the host over a virtual network. For this type of peripherals, there is either no loss in performance (printers, scanners) or possibly a small increase in latency (this mainly affects audio devices).

Peripheral Passthrough

642

- **virtual** PCI, USB or SATA bus
- **forwarding** to a real device
 - e.g. a single USB stick
 - or a single SATA drive

Of course, network-based sharing is not always practical. Fortunately, most peripherals attach to the host system through a handful of standard buses, which are not hard to either pass through, or paravirtualise. The devices then appear as endpoints on the virtual bus of the requisite type exposed to the guest operating system.

Suspend & Resume

643

- the VM can be quite easily **stopped**
- the RAM of a stopped VM can be **copied**
 - e.g. to a **file** in the host filesystem
 - along with **registers** and other state
- and also later **loaded** and **resumed**

An important feature available in most virtualisation solutions is the ability to suspend the execution of a VM and store its state in a file (i.e. create an image of the running virtualised OS). Of course this is only useful if the image can later be loaded and resumed 'as if nothing happened'.

On the outside, this looks rather like what happens when a laptop's lid is closed: the computer stops (in this case to save energy) and when it is opened again, continues where it left off. An important difference here is that in a VM, the guest operating system does not need to cooperate, or even be aware of the suspend/resume operation.

Migration Basics

644

- the stored state can be **sent over network**
- and resumed on a **different host**
- as long as the virtual environment is same
- this is known as **paused** migration

Of course, if an image can be stored in a file, it can just as well be sent over a network. Resuming an image on a different host is called

a 'paused' migration, since the VM is paused for the duration of the network transfer: depending on the size of the image, this can be long enough to time out TCP connections or application-level protocols. Of course, even if this does not happen, there will be a noticeable lag for any interactive use of such a system.

Of course, the operation is predicated on the requirement that the supporting environment **on the outside** of the VM is sufficiently compatible between the hosts: in particular, the backing storage for virtualised block storage, and the virtual networking infrastructure need to match.

Live Migration

645

- uses **asynchronous** memory snapshots
- host copies pages and marks them read-only
- the snapshot is sent as it is constructed
- changed pages are sent at the end

Live migration is an improvement over paused migration above in that it does not cause noticeable lag and does not endanger TCP or other stateful connections that use timeouts to detect broken connections.

The main idea that enables live migration is that the VM can continue to run as normal while its memory is being copied, with the provision that any subsequent writes must be tracked by the hypervisor: this is achieved through the standard 'copy-on-write' trick, where pages are marked read-only right before they are copied, and the hypervisor traps faults. As appropriate, it allows the write to proceed, but also marks the page as dirty. When the initial sweep is finished, another pass is made but this time only through dirty pages, marking them as clean.

Live Migration Handoff

646

- the VM is then paused
- registers and last few pages are sent
- the VM is **resumed** at the remote end
- usually within a **few milliseconds**

When the number of dirty pages is sufficiently small at the end of an iteration, the VM is paused, the remaining dirty pages and the CPU contexts are copied over and the VM is immediately resumed. Since the last transfer is only a few hundred kilobytes, the switchover latency is almost negligible.

Memory Ballooning

647

- how to **deallocate** "physical" memory?
 - i.e. return it to the hypervisor
- this is often desirable in virtualisation
- needs a special host/guest interface

One final consideration is that the hypervisor allocates memory to the guest VMs on demand, but normally, operating systems don't have a concept of 'deallocating' physical memory that they are not actively using. In these circumstances, if the VM sees a spike in memory use, this memory will be indefinitely locked by that VM, even though it has no use for it.

A commonly employed solution is a so-called 'memory ballooning driver' which runs on the guest side and returns unmapped 'physical' (from the point of view of the guest) memory to the host operating system. The memory is unmapped on the host side (i.e. the content of the memory is lost to the guest) and later mapped again if the demand arises.

Part 11.2: Containers

While hardware-accelerated virtualisation is rather efficient when it comes to CPU overhead, there are other costs associated. Some of them can be mitigated by clever tricks (like memory ballooning, TRIM, copy-on-write disk images, etc.) but others are harder to eliminate. When maximal resource utilization is a requirement, containers can often outperform full virtualisation, without significantly compromising other aspects, like maintainability, isolation, or security.

What are Containers?

649

- OS-level virtualisation
 - e.g. virtualised **network stack**
 - or restricted **file system** access
- **not** a complete virtual computer
- turbocharged processes

Containers use virtualisation (in the broad sense of the word) already built into the operating system, mainly based on processes. This is augmented with additional separation, where groups of processes can share, for instance, a network stack which is separate from the network stack available to a different set of processes. While both stacks use the same hardware, they have separate IP addresses, separate routing tables, and so on. Likewise, access to the file system is partitioned (e.g. with **chroot**), the user mapping is separated, as are process tables.

Why Containers

650

- virtual machines take a while to boot
- each VM needs its **own kernel**
 - this adds up if you need many VMs
- easier to **share memory** efficiently
- easier to cut down the OS image

There are two main selling points of containers:

1. so-called 'provisioning speed' – the time it takes from 'I want a fresh system' to having one booted,
2. more efficient resource use.

Both are in large part enabled by sharing a kernel between the containers: in the first case, there is no need to initialize (boot) a new kernel, which saves non-negligible amount of time. For the second point, this is even more important: within a single kernel, containers can share files (e.g. through common mounts) and processes across containers can still share memory – especially executable images and shared libraries that are backed by common files. Achieving the same effect with virtual machines is quite impossible.

Kernel Sharing

651

- multiple containers share a **single kernel**
- but not user tables, process tables, ...
- the kernel must explicitly support this
- another level of **isolation** (process, user, container)

Of course, since a single kernel serves multiple containers, the kernel in question must support an additional isolation level (on top of processes and users), where separate containers have also separate process tables and so on.

Boot Time

652

- a light virtual machine takes a second or two
- a container can take under 50ms
- but VMs can be suspended and resumed
- but dormant VMs take up a lot more space

Even discounting issues like preparation of disk images, on boot time alone, a container can be 20 times faster than a conventional virtual machine (discounting exokernels and similar tiny operating systems).

chroot

653

- the mother of all container systems
- not very sophisticated or secure
- but allows multiple OS images under 1 kernel
- everything else is shared

The `chroot` system call can be (ab)used to run multiple OS images (the user-space parts thereof, to be more specific) under a single kernel. However, since everything besides the file system is fully shared, we cannot really speak about containers yet.

chroot-based 'Containers'

654

- process tables, network, etc. are shared
- the superuser must also be shared
- containers have their **own view** of the filesystem
 - including **system libraries** and **utilities**

Since the process tables, networking and other important services are shared across the images, there is a lot of interference. For instance, it is impossible to run two independent web servers from two different `chroot` pseudo-containers, since only one can bind to the (shared) port 80 (or 443 if you are feeling modern).

Another implication is that the role of the super-user in the container is not contained: the `root` on the inside can easily become `root` on the outside.

BSD Jails

655

- an evolution of the `chroot` container
- adds **user** and **process table** separation
- and a virtualised network stack
 - each jail can get its own IP address
- `root` in the jail has limited power

The jail mechanism on FreeBSD is an evolution of `chroot` that adds what is missing: separation users, process tables and network stacks. The jail also limits what the 'inside' `root` can do (and prevents them from gaining privileges outside the jail). It is one of the oldest open-source containerisation solutions.

Linux VServer

656

- like BSD jails but on Linux
 - FreeBSD jail 2000, VServer 2001
- not part of the mainline kernel
- jailed `root` user is partially isolated

Similar work was done on the Linux kernel a year later, but was not

accepted into the official version of the kernel and was long distributed as a set of third-party patches.

Namespaces

657

- **visibility** compartments in the Linux kernel
- virtualizes common OS resources
 - the filesystem hierarchy (including mounts)
 - process tables
 - networking (IP address)

The solution that was eventually added to official Linux kernels is based around **namespaces** which handle each aspect of containerisation separately: when a new process is created (with a `fork`-like system call, called `clone`), the parent can specify which aspects are to be shared with the parent, and which are to be separated.

cgroups

658

- controls **HW resource allocation** in Linux
- a CPU group is a fair scheduling unit
- a memory group sets limits on memory use
- mostly orthogonal to namespaces

The other important component in Linux containers are 'control groups' which limit resource usage of a process sub-tree (which can coincide with the process sub-tree that belongs to a single container). This allows containers to be isolated not only with respect to their access to OS-level objects, but also with respect to resource consumption.

LXC

659

- mainline Linux way to do containers
- based on namespaces and `cgroups`
- relative newcomer (2008, 7 years after vserver)
- feature set similar to VServer, OpenVZ &c.

LXC is a suite of user-space tools for management of containers based on Linux namespaces and control groups. Since version 1.0 (circa 2014), LXC also offers separation of the in-container super user, and also unprivileged containers which can be created and managed by regular users (limitations apply).

User-Mode Linux

660

- halfway between a container and a virtual machine
- an early fully paravirtualised system
- a Linux kernel runs as a process on another Linux
- integrated in Linux 2.6 in 2003

Ports of kernels 'to themselves' so to speak: a regime where the kernel runs as an ordinary user-space process on top of a different configuration the same kernel, are somewhere between containers and full virtual machines. They rely quite heavily on paravirtualisation techniques, although in a rather unusual fashion: since the kernel is a standard process, it can directly access the POSIX API of the host operating system, for instance directly sharing the host file system.

DragonFlyBSD Virtual Kernels

661

- very similar to User-Mode Linux
- part of DFLyBSD since 2007
- uses standard `libc`, unlike UML
- paravirtual ethernet, storage and console

Another example of the same approach is known as 'virtual kernels' in DragonFlyBSD. In this case, the user-mode port of kernel even uses the standard `libc`, just like any other program. Unfortunately, no direct access to the host file system is possible, making this approach closer to standard VMs.

User Mode Kernels

662

- easier to retrofit securely
 - uses existing security mechanisms
 - for the host, mostly a standard process
- the kernel needs to be ported though
 - analogous to a new hardware platform

When it comes to implementation effort, user-mode kernels are simpler than containers, and offer better host-side security, since they appear as regular processes, without special status.

Migration

663

- not widely supported, unlike in hypervisors
- process state is much harder to serialise
 - file descriptors, network connections &c.
- somewhat mitigated by fast shutdown/boot time

One major drawback of both containers and user-mode kernels is lack of support for suspend and resume, and hence for migration. In both cases, this comes down to the much more complex state of a process, as opposed to a virtual machine, though the issue is considerably more serious for containers (the user-mode kernel is often just a single process on the host, whereas processes in containers are, in fact, real host-side processes).

Part 11.3: Management

Disk Images

665

- disk image is the embodiment of the VM
- the virtual OS needs to be installed
- the image can be a simple file
- or a dedicated block device on the host

Snapshots

666

- making a copy of the image = snapshot
- can be done more efficiently: copy on write
- alternative to OS installation
 - make copies of the **freshly installed** image
 - and run updates after cloning the image

Duplication

667

- each image will have a copy of the system
- copy-on-write snapshots can help
 - most of the base system will not change
 - regression as images are updated separately
- block-level de-duplication is expensive

File Systems

668

- disk images contain entire file systems
- the virtual disk is of (apparently) fixed size
- sparse images: unwritten area is not stored
- initially only filesystem metadata is allocated

Overcommit

669

- the host can allocate more resources than it has
- this works as long as not many VMs reach limits
- enabled by sparse images and CoW snapshots
- also applies to available RAM

Thin Provisioning

670

- the act of obtaining resources on demand
- the host system can be extended as needed
 - to keep pace with growing guest demands
- alternatively, VMs can be migrated out
- improves resource utilisation

Configuration

671

- each OS has its own configuration files
- same methods apply as for physical networks
 - software configuration management
- bundled services are deployed to VMs

Bundling vs Sharing

672

- bundling makes deployment easier
- the bundled components have known behaviour
- but updates are much trickier
- this also prevents resource sharing

Security

673

- hypervisors have a decent track record
 - security here means protection of host from guest
 - breaking out is still possible sometimes
- containers are more of a mixed bag
 - many hooks are needed into the kernel

Updates

674

- each system needs to be updated separately
 - this also applies to containers
- blocks coming from a common ancestor are shared
 - but updating images means loss of sharing

Container vs VM Updates

675

- de-duplication may be easier in containers
 - shared file system – e.g. link farming
- kernel updates: containers and type 2 hypervisors
 - can be mitigated by live migration
- type 1 hypervisors need less downtime

Docker

676

- automated container image management
- mainly a service deployment tool
- containers share a single Linux kernel
 - the kernel itself can run in a VM
- rides on a wave of bundling resurgence

The Cloud

677

- public virtualisation infrastructure
- “someone else’s computer”
- the guests are **not** secure against the host
 - entire memory is exposed, including secret keys
 - host compromise is fatal
- the host is mostly secure from the guests

Review Questions

678

- What is a hypervisor?
- What is paravirtualisation?
- How are VMs suspended and migrated?
- What is a container?

Part 12: Special-Purpose Operating Systems

In this lecture, we will take a look at **special-purpose** operating system (as opposed to general-purpose systems, which were the focus of much of the course). Those systems will lack some (and sometimes many) of the characteristics that we took for granted until now.
TBD.

Review Questions

680

- Question 1
- Question 2
- Question 3
- Question 4

Part 13: Review

What is an OS **made of**?

682

- the kernel
- system libraries
- system daemons / services
- user interface
- system utilities

Basically **every OS** has those.

The Kernel

683

- **lowest level** of an operating system
- executes in **privileged mode**
- manages all the other software
 - including other OS components
- enforces **isolation and security**
- provides **low-level services** to programs

System Libraries

684

- form a layer above the OS kernel
- provide **higher-level** services
 - use kernel services behind the scenes
 - **easier to use** than the kernel interface
- typical example: **libc**
 - provides C functions like **printf**
 - also known as **msvcrt** on Windows

Programming Interfaces

685

- kernel **system call** interface
- → **system libraries** / APIs ←
- inter-process protocols
- command-line utilities (scripting)

(System) Libraries

686

- mainly **C functions** and **data types**
- interfaces defined in **header files**
- definitions provided in **libraries**
 - static libraries (archives): **libc.a**
 - shared (dynamic) libraries: **libc.so**
- on Windows: **msvcrt.lib** and **msvcrt.dll**
- there are (many) more besides **libc / msvcrt**

Shared (Dynamic) Libraries

687

- required for **running** programs
- linking is done at **execution** time
- less code duplication
- can be **upgraded** separately
- but: dependency problems

Why is Everything a File

688

- **re-use** the comprehensive **file system API**
- re-use existing file-based command-line tools
- bugs are bad → **simplicity** is good
- want to print? **cat file.txt > /dev/ulpt0**
 - (reality is a little more complex)

What is a Filesystem?

689

- a set of **files** and **directories**
- usually lives on a single block device
 - but may also be virtual
- directories and files form a **tree**
 - directories are internal nodes
 - files are leaf nodes

File Descriptors

690

- the kernel keeps a table of open files
- the **file descriptor** is an index into this table
- you do everything using file descriptors
- non-Unix systems have similar concepts

Regular files

691

- these contain **sequential data** (bytes)
- may have inner structure but the OS does not care
- there is **metadata** attached to files
 - like when were they last modified
 - who can and who cannot access the file
- you **read()** and **write()** files

Privileged CPU Mode

692

- many operations are **restricted** in **user mode**
 - this is how **user programs** are executed
 - also **most** of the operating system
- software running in privileged mode can do ~anything
 - most importantly it can program the **MMU**
 - the **kernel** runs in this mode

Memory Management Unit

693

- is a subsystem of the processor
- takes care of **address translation**
 - user software uses **virtual addresses**
 - the MMU translates them to **physical addresses**
- the mappings can be managed by the OS kernel

What does a Kernel Do?

694

- **memory** & process management
- task (thread) **scheduling**
- device drivers
 - SSDs, GPUs, USB, bluetooth, HID, audio, ...
- file systems
- networking

Kernel Architecture Types

695

- **monolithic** kernels (Linux, *BSD)
- microkernels (Mach, L4, QNX, NT, ...)
- **hybrid** kernels (macOS)
- type 1 **hypervisors** (Xen)
- exokernels, rump kernels

System Call Sequence

696

- first, **libc** prepares the system call **arguments**
- and puts the system call **number** in the correct register
- then the CPU is switched into **privileged mode**
- this also transfers control to the **syscall handler**

What is an i-node?

697

- an **anonymous**, file-like object
- could be a **regular file**
 - or a directory
 - or a special file
 - or a symlink

Disk-Like Devices

698

- disk drives provide **block-level** access
- read and write data in 512-byte chunks
 - or also 4K on big modern drives
- a big numbered **array of blocks**

I/O Scheduler (Elevator)

699

- reads and writes are requested by users
- access ordering is crucial on a mechanical drive
 - not as important on an SSD
 - but sequential access is still **much preferred**
- requests are **queued** (recall, disks are slow)
 - but they are **not** processed in **FIFO** order

Filesystem as Resource Sharing

700

- usually only 1 or few disks per computer
- many programs want to store **persistent** data
- file system **allocates space** for the data
 - which blocks belong to which file
- different programs can write to different files
 - no risk of trying to **use the same block**

Filesystem as Abstraction

701

- allows the data to be **organised** into files
- enables the user to **manage** and review data
- files have arbitrary & **dynamic size**
 - blocks are **transparently** allocated & recycled
- **structured** data instead of a flat block array

Memory-mapped IO

702

- uses **virtual memory**
- treat a file **as if** it was swap space
- the file is **mapped** into process memory
 - **page faults** indicate that data needs to be read
 - **dirty** pages cause writes
- available as the **mmap** system call

Fragmentation

703

- **internal** – not all blocks are fully used
 - files are of variable size, blocks are fixed
 - a 4100 byte file needs 2 4 KiB blocks
- **external** – free space is non-contiguous
 - happens when many files try to grow at once
 - this means new **files are also fragmented**

Hard Links

704

- **multiple names** can refer to the **same i-node**
 - names are given by **directory entries**
 - we call such multiple-named files **hard links**
 - it's usually forbidden to hard-link directories
- hard links cannot cross device boundaries
 - i-node numbers are only unique within a filesystem

Process Resources

705

- **memory** (address space)
- **processor** time
- open files (descriptors)
 - also working directory
 - also network connections

Process Memory

706

- each process has its own **address space**
- this means processes are **isolated** from each other
- requires that the CPU has an MMU
- implemented via **paging** (page tables)

Process Switching

707

- switching processes means switching **page tables**
- physical addresses do **not** change
- but the **mapping** of virtual addresses does
- large part of physical memory is **not mapped**
 - could be completely unallocated (unused)
 - or belong to **other processes**

What is a Thread?

708

- thread is a **sequence** of instructions
- different threads run different instructions
 - as opposed to SIMD or many-core units (GPUs)
- each thread has its own **stack**
- multiple threads can **share** an address space

Fork

709

- how do we create **new processes**?
- by **fork-ing** existing processes
- fork creates an **identical copy** of a process
- execution continues in both processes
 - each of them gets a different **return value**

Process vs Executable 710

- process is a **dynamic** entity
- executable is a **static** file
- an executable contains an initial **memory image**
 - this sets up memory layout
 - and content of the **text** and **data** segments

Exec 711

- on UNIX, processes are created via **fork**
- how do we **run programs** though?
- **exec**: load a new **executable** into a process
 - this completely **overwrites** process memory
 - execution starts from the **entry point**
- running programs: **fork** + **exec**

What is a Scheduler? 712

- scheduler has two related tasks
 - **plan** when to run which thread
 - actually **switch** threads and processes
- usually part of the **kernel**
 - even in micro-kernel operating systems

Interrupt 713

- a way for hardware to **request attention**
- CPU **mechanism** to divert execution
- partial (CPU state only) **context switch**
- switch to **privileged** (kernel) CPU mode

Timer Interrupt 714

- generated by the PIT or the local APIC
- the OS can **set the frequency**
- a hardware interrupt happens on each **tick**
- this creates an opportunity for bookkeeping
- and for **preemptive scheduling**

What is Concurrency? 715

- events that can happen at the same time
- it is not important if it **does**, only that it **can**
- events can be given a **happens-before** partial order
- they are **concurrent** if unordered by **happens-before**

Why Concurrency? 716

- problem decomposition
 - different **tasks** can be largely independent
- reflecting external concurrency
 - serving **multiple clients** at once
- performance and hardware limitations
 - **higher throughput** on multicore computers

Critical Section 717

- any section of code that must **not** be **interrupted**
- the statement **x = x + 1** **could** be a critical section
- what is a critical section is **domain-dependent**
 - another example could be a bank transaction
 - or an insertion of an element into a linked list

Race Condition: Definition 718

- (anomalous) behaviour that **depends on timing**
- typically among **multiple threads** or processes
- an **unexpected sequence** of events happens
- recall that ordering is not guaranteed

Mutual Exclusion 719

- only one thread can access a resource at once
- ensured by a **mutual exclusion device** (a.k.a **mutex**)
- a mutex has 2 operations: **lock** and **unlock**
- **lock** may need to wait until another thread **unlocks**

Deadlock Conditions 720

1. mutual exclusion
 2. hold and wait condition
 3. non-preemptability
 4. circular wait
- Deadlock is only possible if all 4 are present.

Starvation 721

- starvation happens when a process **can't** make **progress**
- **generalisation** of both **deadlock** and **livelock**
- for instance, **unfair** scheduling on a busy system
- also recall the **readers and writers** problem

What is a Driver? 722

- piece of **software** that talks to a **device**
- usually quite specific / **unportable**
 - tied to the particular **device**
 - and also to the **operating system**
- often part of the **kernel**

Drivers and Microkernels 723

- drivers are **excluded** from microkernels
- but the driver still needs **hardware access**
 - this could be a special **memory region**
 - it may need to **react** to **interrupts**
- in principle, everything can be done **indirectly**
 - but this may be quite **expensive**, too

Interrupt-driven IO 724

- peripherals are **much** slower than the CPU
 - **polling** the device is expensive
- the peripheral can **signal** data availability
 - and also **readiness** to accept more data
- this **frees up CPU** to do other work in the meantime

Memory-mapped IO 725

- devices **share** address space with memory
- **more common** in contemporary systems
- IO uses the same instructions as memory access
 - **load** and **store** on RISC, **mov** on **x86**
- allows **selective** user-level access (via the MMU)

Direct Memory Access 726

- allows the device to directly read/write **memory**
- this is a **huge** improvement over **programmed IO**
- **interrupts** only indicate buffer **full/empty**
- devices can read and write arbitrary **physical** memory
 - opens up **security** / reliability problems

GPU Drivers 727

- split into a number of components
- graphics output / frame buffer access
- **memory management** is often done in kernel
- geometry, textures &c. are prepared **in-process**
- front end API: OpenGL, Direct3D, Vulkan, ...

Storage Drivers 728

- split into adapter, bus and device drivers
- often a single driver per device type
 - at least for disk drives and CD-ROMs
- bus **enumeration** and **configuration**
- data addressing and **data transfers**

Networking Layers 729

1. Link (Ethernet, WiFi)
2. Network (IP)
3. Transport (TCP, UDP, ...)
4. Application (HTTP, SMTP, ...)

Networking and Operating Systems 730

- a **network stack** is a standard part of an OS
- large part of the stack lives in the **kernel**
 - although this only applies to **monolithic** kernels
 - microkernels use **user-space** networking
- another chunk is in system **libraries & utilities**

Kernel-Side Networking 731

- device **drivers** for networking **hardware**
- network and transport **protocol** layers
- **routing** and packet filtering (firewalls)
- networking-related **system calls** (sockets)
- network **file systems** (SMB, NFS)

IP (Internet Protocol) 732

- uses 4 byte (v4) or 16 byte (v6) addresses
 - split into **network** and **host** parts
- it is a packet-based protocol
- is a **best-effort** protocol
 - packets may get lost, reordered or corrupted

TCP: Transmission Control Protocol 733

- a **stream-oriented** protocol on top of IP
- works like a **pipe** (transfers a byte sequence)
 - must respect **delivery order**
 - and also **re-transmit** lost packets
- must establish **connections**

UDP: User (Unreliable) Datagram Protocol 734

- TCP comes with non-trivial **overhead**
 - and its guarantees are **not always required**
- UDP is a much **simpler** protocol
 - a very thin wrapper around IP
 - with **minimal overhead** on top of IP

DNS: Domain Name Service 735

- hierarchical **protocol** for name resolution
 - runs on top of TCP or UDP
- domain **names are split** into parts using dots
 - each domain knows whom to ask for the next bit
 - the name database is effectively **distributed**

NFS (Network File System) 736

- the traditional UNIX **networked filesystem**
- hooked quite deep into the kernel
 - assumes generally reliable network (LAN)
- filesystems are **exported** for use over NFS
- the client side **mounts** the NFS-exported volume

Shell 737

- **programming language** centered on OS interaction
- rudimentary **control flow**
- **untyped**, text-centered variables
- dubious error handling

Interactive Shells 738

- almost all shells have an **interactive mode**
- the user inputs a single statement on keyboard
- when confirmed, it is immediately **executed**
- this forms the basis of **command-line interfaces**

Shell Scripts 739

- a **shell script** is an (executable) file
- in simplest form, it is a **sequence of commands**
 - each command goes on a separate line
 - executing a script is about the same as typing it
- but can use **structured programming** constructs

Terminal 740

- can **print text** and read text from a **keyboard**
- normally everything is printed on the last line
- the text could contain **escape** (control) sequences
 - for printing colourful text or clearing the screen
 - also for printing text at a **specific coordinate**

A GUI Stack 741

- graphics card **driver**, mode setting
- **drawing/painting** (usually hardware-accelerated)
- multiplexing (e.g. using windows)
- **widgets**: buttons, labels, lists, ...
- **layout**: what goes where on the screen

X11 (X Window System) 742

- a traditional UNIX windowing system
- provides a C API (**xlib**)
- built-in **network transparency** (socket-based)
- core protocol version 11 from 1987

Users 743

- originally a proxy for **people**
- currently a more **general abstraction**
- user is the unit of **ownership**
- many **permissions** are user-centered

User Management 744

- the system needs a **database** of **users**
- in a network, user **identities** often need to be **shared**
- could be as simple as a **text file**
 - **/etc/passwd** and **/etc/group** on UNIX systems
- or as complex as a distributed database

User Authentication 745

- the user needs to **authenticate** themselves
- **passwords** are the most commonly used method
 - the **system** needs to know the right password
 - user should be able to change their password
- **biometric** methods are also quite popular

Ownership 746

- various **objects** in an OS can be **owned**
 - primarily **files** and **processes**
- the owner is typically whoever **created** the object
 - ownership can be **transferred**
 - usually at the impetus of the original owner

Access Control Policy 747

- there are 3 pieces of information
 - the **subject** (user)
 - the **verb** (what is to be done)
 - the **object** (the file or other resource)
- there are many ways to **encode** this information

Sandboxing 748

- tries to **limit damage** from code execution **exploits**
- the program **drops** all privileges it can
 - this is done **before** it touches any of the **input**
 - the attacker is stuck with the **reduced privileges**
 - this can often prevent a successful attack

What is a Hypervisor 749

- also known as a Virtual Machine Monitor
- allows execution of **multiple operating systems**
- like a **kernel** that **runs kernels**
- **isolation** and resource sharing

Hypervisor Types

750

- type 1: **bare metal**
 - standalone, microkernel-like
- type 2: **hosted**
 - runs on top of normal OS
 - usually need **kernel support**

Paravirtual Devices

751

- special **drivers** for **virtualised** devices
 - block storage, network, console
 - random number generator
- **faster** than software emulation
 - **orthogonal** to CPU/MMU virtualisation

VM Suspend & Resume

752

- the VM can be quite easily **stopped**
- the RAM of a stopped VM can be **copied**
 - e.g. to a **file** in the host filesystem
 - along with **registers** and other **state**
- and also later **loaded** and **resumed**

What are Containers?

753

- OS-level **virtualisation**
 - e.g. virtualised **network stack**
 - or restricted file system access
- **not** a complete virtual computer
- turbocharged processes

Bundling vs Sharing

754

- **bundling** makes deployment easier
- the bundled components have **known behaviour**
- but **updates** are much trickier
- this also **prevents** resource **sharing**

Review Questions

755

- What does portability mean?
- What is a socket?
- What is a device driver?
- What is a directory?

1 The End

Actually...

756

- a 2-part, **written** final **exam**
- test: 9/10 **required**
 - pool of 52 questions (in the slides)
- free-form text
 - one of the 11 lecture topics
 - 1 page A4: be **concise** but **comprehensive**