Operating Systems

Petr Ročkai

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

- 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

- 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

- 24 hours to complete, 2 attempts possible
- 10 questions, picked from review questions
  - mid-term first 24, end-term second 24
- you need to pass either mid-term or end-term
- 7 out of 10 required for mid-term, 8 of 10 for end-term
- preliminary mid-term date: 11th of April, 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

- 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

- 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 that 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

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)

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

- 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

- generally, one lecture = one topic
- there will be most likely 12 lectures
- a 50-minute review in the last lecture
- online mid-term in April

Semester Overview

This section gives a high-level overview of the topics that will be covered in individual lectures.

2. System Libraries and APIs

- 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

- 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

- 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

- 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

- 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

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

8. Network Stack

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

9. Command Interpreters & User Interfaces

- 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

10. Users and Permissions

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

11. Virtualisation & Containers

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

12. Special-Purpose Operating Systems

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

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.

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

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?

- 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?

- 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, …

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?

- 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

What is an OS made of?

- 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

- 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.

Finaly, 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

- 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

System Daemons

- 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

User Interface

- 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

System Utilities

- 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

Optional Components

- 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

Part 1.2: Interfaces

Programming Interface

- 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

Message Passing

- 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

Portability

- 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

Hardware Platform

- 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

- 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

- 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

- 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)

- 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

- 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

- more complexity
- less efficiency
- leaky abstractions

Abstraction Benefits

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

Abstraction Trade-Offs

- 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

General-Purpose Operating Systems

- 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

- 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

- 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 thousand 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

- 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-fetching and catastrofic 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 afterthought 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

- 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

- 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?

- 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

Review Questions

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

- 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

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

- 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

- 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

- 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

- 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

(System) Libraries

- 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

```
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

- `/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

- 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

- 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

- 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`)

```
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`)

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

/* ... */

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

The POSIX C Library

- `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

System Calls: Numbers

- 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

System Calls: the `syscall` function

- 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)

System Calls: Wrappers

- 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 use `syscall()` internally

Portability

- 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

NeXT and Objective C

- 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 OOP features (like Smalltalk)
- this has been partly inherited into macOS
  - evolving into Swift

System Libraries: UNIX

- 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++`

System Libraries: Windows

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

Documentation

- 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`
- you can learn a lot from those sources

Part 2.3: Compiler & Linker

C Compiler

- 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

Object Files

- 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

Object File Formats

- `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)

Archives (Static Libraries)

- 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)

Linker

- 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

Symbols vs Addresses

- 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

Resolving Symbols

- 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
- objects can only use symbols already in the table
- resolving symbols = finding their addresses

Executable

- 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

Shared Libraries

- 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)

Addresses Revisited

- 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

Compiler, Linker &c.

- 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`

Part 2.4: File-Based APIs

Everything is a File

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

Why is Everything a File

- 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)

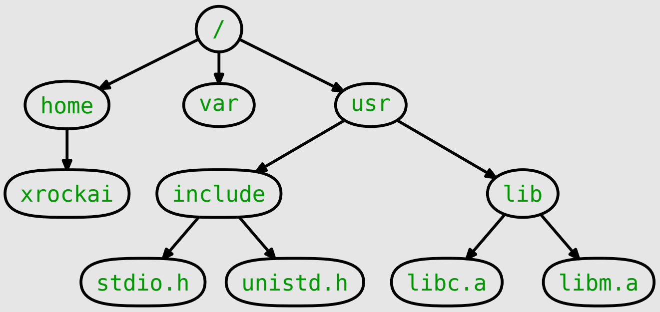
What is a Filesystem?

- 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 Paths

- 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`

The File Hierarchy



The Role of Files and Filesystems

- very central in Plan9
- central in most UNIX systems
  - cf. Linux pseudo-filesystems
  - `/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)

The Filesystem API

- 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

File Descriptors

- 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

Regular files

- 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

Directories

- 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()`

Mounts

- 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.)

Devices

- 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

Pipes

- 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)

Sockets

- 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

Socket Types

- 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

Review Questions

5. What is a shared (dynamic) library?
6. What does a linker do?
7. What is a symbol in an object file?
8. 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

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 at 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.

Reminder: Software Layering

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

Part 3.1: Privileged Mode

CPU Modes

- 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.

Privileged Mode

- 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

- 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

- 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

- 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

- 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 we can appreciate how important the MMU is for a modern operating system.

Memory Maps

- 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

Page Tables

- 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

Kernel Protection

- kernel memory is usually mapped into all processes
  - this substantially improves performance on many CPUs
  - well, 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

Part 3.2: Booting

Starting the OS

- 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

Boot Process

- 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

Boot Process (cont'd)

- 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

User-mode Initialisation

- `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

After Log-In

- 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

CPU Init

- 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

Bootloader

- historically limited to tens of kilobytes of code
- the bootloader locates the kernel on disk
  - it often allows 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

Modern Booting on `x86`

- on modern system, the bootloader 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

Booting ARM

- 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

Part 3.3: Kernel Architecture

Architecture Types

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

Microkernel

- handles memory protection

- (hardware) interrupts
- task / process scheduling
- message passing

- everything else is separate

Monolithic kernels

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

Microkernel Redux

- 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

Hybrid Kernels

- 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

Micro vs Mono

- 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

Exokernels

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

Type 1 Hypervisors

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

Unikernels

- 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

Exo vs Uni

- 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

Reminder: Kernel Protection

- 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)

System Calls

- 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

Trapping into the Kernel

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

Trap Example: `x86`

- 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)

Software Interrupts

- 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

Aside: SW Interrupts on PCs

- 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`

Trap Example: `amd64` / `x86_64`

- `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

Which System Call?

- 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

Reminder: System Call Numbers

- 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

System Call Sequence

- 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

System Call Handler

- 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 */
}
```

System Call Arguments

- 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

Part 3.5: Kernel Services

What Does a Kernel Do?

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

Additional Services

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

Reminder: Microkernel Systems

- 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

Kernel Services

- 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

User-Space Drivers in Monolithic Systems

- 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

Review Questions

9. What CPU modes are there and how are they used?
10. What is the memory management unit?
11. What is a microkernel?
12. What is a system call?

Part 4: File Systems

Lecture Overview

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

5. Advanced Features

Part 4.1: Filesystem Basics

What is a File System?

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

What is a (Regular) File?

- 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

What is a Directory?

- 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

What is an i-node?

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

Files are Anonymous

- 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

What Else is a Byte Sequence?

- 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

Writing Byte Sequences

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

Special Files

- 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

File System Types

- 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)

Multi-User Systems

- file ownership
- file permissions
- disk quotas

Ownership & Permissions

- 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

Disk Quotas

- 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

Part 4.2: The Block Layer

Disk-Like Devices

- 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

Aside: Disk Addressing Schemes

- CHS: Cylinder, Head, Sector
  - structured adressing 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 Addessing
  - linear, unstructured address space
  - started as 22, later 28, … now 48 bit

Block-Level Access

- 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

Aside: Access Times

- 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

Block Access Cache

- 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

Write Buffers

- 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

I/O Scheduler (Elevator)

- 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

RAID

- 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

RAID Performance

- 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

Block-Level Encryption

- 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

Storing Data in Blocks

- 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

Filesystem as Resource Sharing

- 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

- 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

Part 4.3: Virtual Filesystem Switch

Virtual File System Layer

- 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

VFS in OOP terms

- 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

The `filesystem` Class

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

Filesystem-Specific Operations

- `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

Standard IO

- 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

Memory-mapped IO

- 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

Sync-ing Data

- 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

Filesystem-Agnostic Operations

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

Executables

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

The `fcntl` Syscall

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

Special Files

- 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

File Locking

- 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

Mount Points

- 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

Part 4.4: The UNIX Filesystem

Superblock

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

I-Nodes

- 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

I-Node Allocation

- 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

I-Node Content

- 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

Attaching Data to I-Nodes

- 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

Fragmentation

- 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

Fragmentation Problems

- performance: can't use fast sequential IO
  - programs often read files sequentially
  - fragmention random IO on the device
- metadata size: can't use long extents
- internal: waste of disk space

Directories

- 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

File Name Lookup

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

Old-Style Directories

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

Hash-Based Directories

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

Tree-Based Directories

- 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

Hard Links

- 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

Soft Links (Symlinks)

- they exist to lift the one-device limitation
- soft links to directories are OK
  - 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

Free Space

- 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

File System Consistency

- 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

Journalling

- 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 by making consistency checks fast
  - may also prevent data loss

Part 4.5: Advanced Features

What Else Can Filesystems Do?

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

File Compression

- 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

File Encryption

- 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

Block De-duplication

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

Snapshots

- 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

Checksums

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

Redundant Storage

- 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

Review Questions

13. What is a block device?
14. What is an IO scheduler?
15. What does memory-mapped IO mean?
16. What is an i-node?

Part 5: Basic Resources & Multiplexing

Lecture Overview

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

Part 5.1: Processes and Virtual Memory

Prehistory: Batch Systems

- 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

History: Time Sharing

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

Processes: Early View

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

Process Resources

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

Process Memory Segments

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

Process Memory

- 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

- 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

Paging and TLB

- 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)

Processor Time Sharing

- 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

Multiple CPUs

- execution of a program is sequential
- instructions depend on results of previous instructions
- one CPU = one instruction sequence
- physical limits on CPU speed multiple cores

Threads

- how to use multiple cores in one process?
- threads: a new unit of CPU scheduling
- each thread runs sequentially
- one process can have multiple threads

What is a Thread?

- 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

Modern View of a Process

- 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

Memory Segment Redux

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

Fork

- 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

Lazy Fork

- 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

Lazy Fork: Faults

- 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

Init

- 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

Process Identifier

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

Process vs Executable

- 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

- 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`

Part 5.2: Thread Scheduling

What is a Scheduler?

- 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

Switching Threads

- 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

Fixed vs Dynamic Schedule

- 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

Preemptive Scheduling

- 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

Cooperative Scheduling

- 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

Scheduling in Practice

- 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

Waiting and Yielding

- 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

Run Queues

- runnable (not 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

Priorities

- 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

Fairness

- 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?

Fair Share Scheduling

- 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

Scheduling Strategies

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

Interactivity

- 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

Reducing Latency

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

Maximising Throughput

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

Multi-Core Schedulers

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

Scheduling and Caches

- 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

Core Affinity

- 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

NUMA Systems

- non-uniform memory architecture
  - different memory is attached to different CPUs
  - each symmetric 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

Part 5.3: Interrupts and Clocks

Interrupt

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

Hardware Interrupts

- 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

Interrupt Controllers

- 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

Timekeeping

- 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

Timer Interrupt

- 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

Timer Interrupt and Scheduling

- 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
- those checks are done on each tick
  - rescheduling is usually less frequent

Timer Interrupt Frequency

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

Tickless Kernels

- 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

Tickless Scheduling

- slice length (quantum) 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, ...

Other Interrupts

- 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

Interrupt Routing

- 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

Review Questions

17. What is a thread and a process?
18. What is a (thread, process) scheduler?
19. What do `fork` and `exec` do?
20. 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

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?

- 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?

- 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

Parallel Hardware

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

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 the communication is, in fact, *intra*-process in those cases).

Reminder: What is a Thread

- 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

Reminder: What is a Process

- 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

I/O vs Communication

- 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

Direction

- 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?

Communication Example

- 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

Files

- 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

A File-Based IPC Example

- 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

Directories

- 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

Pipes

- 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

UNIX and Pipes

- 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

- 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

Shared Memory

- 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 same piece of physical memory
  - this is the more traditional setting
  - hence also allows inter-process communication

Message Passing

- 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

Part 6.2: Synchronisation

Shared Variables

- 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`

Shared Heap Variable

```
void *thread( int *x ) { *x = 7; }
int main()
{
    pthread_t id;
    int *x = malloc( sizeof( int ) );
    pthread_create( &id, NULL, thread, x );
}
```

Race Condition: Example

- 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?

Race on a Variable

- memory access is not atomic
- take `x = x + 1`

```
a₀ ← load x  | b₀ ← load x
a₁ ← a₀ + 1  | b₁ ← b₀ + 1
store a₁ x   | store b₁ x
```

Critical Section

- 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

- (anomalous) behaviour that depends on timing
- typically among multiple threads or processes
- an unexpected sequence of events happens
- recall that ordering is not guaranteed

Races in a Filesystem

- 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

Mutual Exclusion

- 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`

Semaphore

- somewhat more general than a mutex
- allows multiple interchangeable instances of a resource
  - that many threads can enter the critical section
- basically an atomic counter

Monitors

- 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 any given time

Condition Variables

- 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

Spinlocks

- 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 between threads on the same CPU are very bad
  - but can be very efficient between CPUs

Suspending Mutexes

- 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

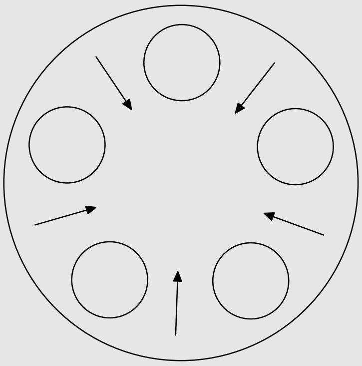
Condition Variables Revisited

- same principle as a suspending mutex
- the waiting thread goes into a wait queue
- the `signal` method moves the thread back to a run queue
- the busy-wait version is known as polling

Barrier

- 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

Dining Philosophers



Readers and Writers

- 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?

Read-Copy-Update

- 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?

Part 6.3: Deadlocks

Shared Resources

- 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

Network-based Sharing

- sharing is not limited to processes on one computer
- printers and scanners can be network-attached
- all computers on network may need to coordinate access
  - this could lead to multi-computer deadlocks

Locks as Resources

- 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

Preemptable Resources

- 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
- preemtability may also depend on context
  - maybe paging is not available

Non-preemptable Resources

- 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

Resource Acquisition

- 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

Waiting

- 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

Resource Deadlock

- two resources, A and B
- two 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

Deadlock Conditions

1. mutual exclusion
2. hold and wait condition
3. non-preemtability
4. circular wait

Deadlock is only possible if all 4 are present.

Non-Resource Deadlocks

- 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

Example: Pipe Deadlock

- 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

Deadlocks: Do We Care?

- 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

Deadlock Detection

- we can at least try to detect deadlocks
- usually by checking the circular wait condition
- keep a graph of who owns what and who waits for what
- if there is a loop in the graph deadlock

Deadlock Recovery

- 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

Deadlock Avoidance

- we can possibly deny acquisitions to avoid deadlocks
- we need to know the maximum resources for each process
- avoidance relies on safe states
  - worst case all processes ask for maximum resources
  - safe means we can avoid a deadlock in the worst case

Deadlock Prevention

- 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

Prevention via Spooling

- 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 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.

Prevention via Reservation

- 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

Prevention via Ordering

- 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

Livelock

- 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 as livelock and is just as bad as a deadlock

Starvation

- 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

Review Questions

21. What is a mutex?
22. What is a deadlock?
23. What are the conditions for a deadlock to form?
24. What is a race condition?

Part 7: Device Drivers

Lecture Overview

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

- 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?

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- allows the device to directly read/write memory
- this is a huge improvement over programmed IO
- interrupts only indicate buffer full/empty
- the device can read and write arbitrary physical memory
  - opens up security / reliability problems

IO-MMU

- 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)

- 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

- 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

- 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

- 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

- 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)

- 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

- 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
  - allows for enumeration and automatic configuration

PCI IDs and Drivers

- 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

- 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

- 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

- 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

- a set of vendor-neutral protocols
- HID = human-interface device
- mass storage = disk-like devices
- audio equipment
- printing

Other USB Uses

- ethernet adapters
- usb-serial adapters
- wifi adapters (dongles)
  - there isn't a universal protocol
  - each USB WiFi adapter needs
- bluetooth

ARM Busses

- 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

- 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

- People Can't Memorize Computer Industry Acronyms
  - PC = Personal Computer, MC = Memory Card
- hotplug-capable notebook expansion bus
- used for memory cards, network adapters, modems
- comes with its own set of drivers (cardbus)

ExpressCard

- 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

- those are physical interfaces, not special busses
- they provide some mix of PCIe, SATA and USB
  - also other protocols like I^{2}C, SMBus, …
- used mainly for compact SSDs and wireless
  - also GPS, NFC, bluetooth, …

Part 7.3: Graphics and GPUs

Graphics Cards

- 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

- 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

- 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 workstation had 3D accelerators (OpenGL '92)

GPU (Graphical Processing Unit)

- 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

- 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

- 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)

Mode Setting

- this part deals with screen configuration and resolution
- including support for e.g. multiple displays
- usually also supports primitive (SW-only) framebuffer
- often done by a kernel with minimum user-level support

Graphics Servers

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- serial, point-to-point replacement for ATA
- hardware-level incompatible to (parallel) ATA
  - but SATA inherited the ATA command set
  - legacy mode allows PATA drivers to talk to SATA drives
- hot-swap capable – replace drives in a running system

AHCI (Advanced Host Controller Interface)

- 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

- 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)

- 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

- 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

- 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

- 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

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

- 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

- 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)

- 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

- 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

- specifies the physical media
- 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

- 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

- 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

- 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

- 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 can be handled by different CPU cores

Checksum and TCP Offloading

- more advanced adapters can offload certain features
- commonly computation of mandatory packet checksums
- but also TCP-related features
- this needs both driver support and TCP/IP stack support

WiFi

- 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

- 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, ...)

Part 8: Network Stack

Lecture Overview

1. Networking Intro
2. The TCP/IP Stack
3. Using Networks
4. Network File Systems

Part 8.1: Networking Intro

Host and Domain Names

- hostname = human readable computer name
- hierarchical system, big-endian: `www.fi.muni.cz`
- FQDN = fully-qualified domain name
- the local suffix may be omitted (`ping aisa`)

Network Addresses

- address = machine-friendly and numeric
- IPv4 address: 4 octets (bytes): `192.168.1.1`
- IPv6 address: 16 octets
- Ethernet (MAC): 6 octets, `c8:5b:76:bd:6e:0b`

Network Types

- 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

Networking Layers

2. Link (Ethernet, WiFi)
3. Network (IP)
4. Transport (TCP, UDP, ...)

7. Application (HTTP, SMTP, ...)

Networking and Operating Systems

- 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

- 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)

System Libraries

- the socket and related APIs
- host name resolution (a DNS client)
- encryption and data authentication (SSL, TLS)
- certificate handling and validation

System Utilities

- network configuration (`ifconfig`)
- diagnostics (`ping`, `traceroute`)
- packet logging and inspection (`tcpdump`)
- route management (`route`, `bgpd`)

Networking Aspects

- packet format
  - what are the units of communication
- addressing
  - how are the sender and recipient named
- packet delivery
  - how a message is delivered

Protocol Nesting

- 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

Packet Nesting

- 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 an HTTP request

Stacked Delivery

- 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

Layers vs Addressing

- 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

ARP (Address Resolution Protocol)

- 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 MAC translations

Ethernet

- link-level communication protocol
- largely implemented in hardware
- the OS uses a well-defined interface
  - packed receive and submit
  - using MAC addresses (ARP is part of the OS)

Packet Switching

- 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

Bridging

- 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

Tunneling

- tunnels are virtual layer 2 or 3 devices
- they encapsulate traffic using a higher-level protocol
- tunneling is used to implement Virtual Private Networks
  - a software bridge can operate over an UDP tunnel
  - the tunnel is usually encrypted

PPP (Point-to-Point Protocol)

- 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

Wireless

- 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

Part 8.2: The TCP/IP Stack

IP (Internet Protocol)

- 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 Networks

- 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

Routing

- 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

Services and TCP/UDP Port Numbers

- 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 ar given names
  - port 25 is SMTP, port 80 is HTTP, …

ICMP: Internet Control Message Protocol

- 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`

TCP: Transmission Control Protocol

- 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

TCP Connections

- 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

Sequence Numbers

- 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

Packet Loss and Re-transmission

- 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

UDP: User (Unreliable) Datagram Protocol

- 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

Name Resolution

- 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

DNS: Domain Name Service

- 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

DNS Recursion

- 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`

DNS Recursion Example

```
$ 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
```

DNS Record Types

- `A` is for (IP) Address
- `AAAA` is for an IPv6 Address
- `CNAME` is for an alias

- `MX` is for mail servers
- and many more

Firewalls

- 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

Packet Filtering

- packet filtering is how firewalls are usually implemented
- 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

Packet Filter Operation

- 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

Packet Flter Examples

- 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

Part 8.3: Using Networks

Sockets Reminder

- 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()`

Socket Types

- 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

Creating Sockets

- 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()`

Resolver API

- `libc` contains a resolver
  - available as `gethostbyname` (and `gethostbyname2`)
  - also `gethostbyaddr` for reverse lookups
- can look in many different places
  - most systems support at least `/etc/hosts`
  - and DNS-based lookups

Network Services

- 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, ...

Network Service Examples

- (secure) remote shell – `sshd`
- 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

- the `ssh` command talks 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

Why Network Filesystems?

- 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

NAS (Network-Attached Storage)

- 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, ...

NFS (Network File System)

- 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 History

- 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

VFS Reminder

- 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

RPC (Remote Procedure Call)

- 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
  - this makes it easy to implement on UNIX-like systems

Port Mapper

- 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

The NFS Daemon

- 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

SMB (Server Message Block)

- 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 2.0

- simpler than SMB1 due to fewer retrofits and compat
- better performance and security
- support for symbolic links
- available since Windows Vista (2006)

Part 9: Shells & User Interfaces

Lecture Overview

1. Command Interpreters
2. The Command Line
3. Graphical Interfaces

Part 9.1: Command Interpreters

Shell

- programming language centered on OS interaction
- rudimentary control flow
- untyped, text-centered variables
- dubious error handling

Interactive Shells

- 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

- 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

- 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

- 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

- 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

- also known as `ksh`, released in 1983
- middle ground between `sh` and `csh`
- basis of the POSIX.2 requirements

- a number of implementations exists

Commands

- 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 doas a `fork` + `exec`
  - this means new process for each command
  - process creation is fairly expensive

Built-in Commands

- `cd` change the working directory
- `export` for setting up environment
- `echo` print a message
- `exec` replace the shell process (no `fork`)

Variables

- 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

- 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

- 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

- whitespace is an argument separator in shell
- multi-word arguments must be quoted
- quotes can be double quotes `""` or single `''`
  - double quotes allow variable substitution

Quoting and Substitution

- 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 $foo` will expand into `ls hello world` and executed with `argv[1] = "hello"` and `argv[2] = "world"`, in effect looking for two separate files. The latter, `ls "$foo""`, will be executed with `argv[1] = "hello world"`’.

Special Variables

- `$?` 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

- is like shell variables but not the same
- the environment is passed to all executed programs
  - but 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

- `$PATH` tells the system where to find programs
- `$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

Globbing

- 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

- 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)

- 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

- `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

- `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

- 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

- `;` (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

- 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

- 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

- 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

- 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

- the string printed when shell expects a command
- controlled by the `PS1` environment variable
- usually shows at least your username and the hostname
- also: working directory, battery status, time, weather, ...

Job Control

- 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

- 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 scrollback 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

- 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

- `sed` – stream editor, non-interactive
- `ed` – line oriented, interactive
- `vi` – visual, screen oriented
- `ex` – line-oriented mode of `vi`

TUI: Text User Interface

- 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

- 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

- 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

- 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

- Windows
- macOS, iOS
- X11
- Wayland
- Android

Portability

- 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

- 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

- 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

- 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

- 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)

- a traditional UNIX windowing system
- provides a C API (`xlib`)
- built-in network transparency (socket-based)
- core protocol version 11 from 1987

X11 Architecture

- 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

- 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

- 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)

- 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)

- 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 (Simple Protocol for Indep. 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

- 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).

Part 10: Access Control

Lecture Overview

1. Multi-User Systems
2. File Systems
3. Sub-user Granularity

Part 10.1: Multi-User Systems

Users

- originally a proxy for people
- currently a more general abstraction
- user is the unit of ownership
- many permissions are user-centered

Computer Sharing

- 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

Ownership

- 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

Process Ownership

- 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

File Ownership

- 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

Access Control Models

- 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

(Virtual) System Users

- 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

Principle of Least Privilege

- 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

Privilege Separation

- 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 the simplest feasible IPC

Process Separation

- recall that each process runs in its own address space
  - but shared memory can be requested
- each user has a view of the filesystem
  - a lot more is shared by default in the filesystem
  - especially the namespace (directory hierarchy)

Access Control Policy

- 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

Access Rights Subjects

- 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

Access Rights Verbs

- 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.

Access Rights Objects

- 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, …

Subjects in POSIX

- 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

User Management

- 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

LDAP and Active Directory are popular choices for centralised network-level user databases.

User and Group Identifiers

- 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

Changing Identities

- 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

Login

- a super-user process manages user logins
- the user types their name and provides credentials
  - upon successful authentication, `login` calls `fork()`
  - the child calls `setuid()` to the user
  - and uses `exec()` to start a shell for the user

User Authentication

- 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

Remote Login

- 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

Computer Authentication

- 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 will sign a message establishing its identity

2-factor Authentication

- 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)
  - what the user is (biometric)

Enforcement: Hardware

- 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

Enforcement: Kernel

- 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

Enforcement: System Calls

- 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

Enforcement: Service APIs

- 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 is rooted in kernel facilities

Part 10.2: File Systems

File Access Rights

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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`

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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`

- `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

Users are Not Enough

- 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

Naming Programs

- 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

Program as a Subject

- 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

Mandatory Access Control

- 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 a generalisation of user groups.

Capabilities

- 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

Dismantling the `root` User

- 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

Security and Execution

- 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

Untrusted Input

- 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

Process as a Subject

- 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
- processes are identified using their numeric `pid`
  - restrictions are inherited across `fork()`

Sandboxing

- 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

Untrusted Code

- traditionally, you would only execute trusted code

  - usually 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

API-Level Access Control

- 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

Android/iOS Permissions

- 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

Part 11: Virtualisation & Containers

Lecture Overview

1. Hypervisors
2. Containers
3. Management

Part 11.1: Hypervisors

What is a Hypervisor

- also known as a Virtual Machine Monitor
- allows execution of multiple operating systems
- like a kernel that runs kernels
- improves hardware utilisation

Motivation

- 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

Virtualisation in General

- many resources are “virtualised”
  - physical memory by the MMU
  - peripherals by the OS
- makes resource management easier
- enables isolation of components

Hypervisor Types

- type 1: bare metal
  - standalone, microkernel-like
- type 2: hosted
  - runs on top of normal OS
  - usually need kernel support

Type 1 (Bare Metal)

- IBM z/VM
- (Citrix) Xen
- Microsoft Hyper-V
- VMWare ESX

Type 2 (Hosted)

- VMWare (Workstation, Player)
- Oracle VirtualBox
- Linux KVM
- FreeBSD bhyve
- OpenBSD vmm

History

- started with mainframe computers
- IBM CP/CMS: 1968
- IBM VM/370: 1972
- IBM z/VM: 2000

Desktop Virtualisation

- x86 hardware lacks virtual supervisor mode
- software-only solutions viable since late 90s
  - Bochs: 1994
  - VMWare Workstation: 1999
  - QEMU: 2003

Paravirtualisation

- 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

The Virtual `x86` Revolution

- 2005: virtualisation extensions on `x86`
- 2008: MMU virtualisation
- unmodified guest at near-native speed
- most software-only solutions became obsolete

Paravirtual Devices

- special drivers for virtualised devices
  - block storage, network, console
  - random number generator
- faster than software emulation
  - orthogonal to CPU/MMU virtualisation

Virtual Computers

- 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

Essential Resources

- the CPU and RAM
- persistent (block) storage
- network connection
- a console device

CPU Sharing

- same principle as normal processes
- there is a scheduler in the hypervisor
  - simpler, with different trade-offs
- privileged instructions are trapped

RAM Sharing

- very similar to standard paging
- software (shadow paging)
- or hardware (second-level translation)
- fixed amount of RAM for each VM

Shadow Page Tables

- 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

- hardware-assisted MMU virtualisation
- adds guest-physical to host-physical layer
- greatly simplifies the VMM
- also much faster than shadow page tables

Network Sharing

- 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

Virtual Block Devices

- 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

Special Resources

- mainly useful in desktop systems
- GPU / graphics hardware
- audio equipment
- printers, scanners, …

PCI Passthrough

- 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

GPUs and Virtualisation

- 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)

Peripherals

- useful either via passthrough
  - audio, webcams, …
- or standard sharing technology
  - network printers & scanners
  - networked audio servers

Peripheral Passthrough

- virtual PCI, USB or SATA bus
- forwarding to a real device
  - e.g. a single USB stick
  - or a single SATA drive

Suspend & Resume

- 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

Migration Basics

- 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

Live Migration

- 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 Handoff

- 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

Memory Ballooning

- how to deallocate "physical" memory?
  - i. e. return it to the hypervisor
- this is often desirable in virtualisation

- needs a special host/guest interface

Part 11.2: Containers

What are Containers?

- OS-level virtualisation
  - e.g. virtualised network stack
  - or restricted file system access
- not a complete virtual computer
- turbocharged processes

Why Containers

- 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

Kernel Sharing

- 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)

Boot Time

- 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

`chroot`

- the mother of all container systems
- not very sophisticated or secure
- but allows multiple OS images under 1 kernel
- everything else is shared

`chroot`-based Containers

- 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

BSD Jails

- 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

Linux VServer

- like BSD jails but on Linux
  - FreeBSD jail 2000, VServer 2001
- not part of the mainline kernel
- jailed `root` user is partially isolated

Namespaces

- visibility compartments in the Linux kernel
- virtualizes common resources
  - the filesystem hierarchy (including mounts)
  - process tables
  - networking (IP address)

`cgroups`

- controls resource allocation in Linux
- a CPU group is a fair scheduling unit
- a memory group sets limits on memory use
- mostly orthogonal to namespaces

LXC

- 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.

User-Mode Linux

- 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

DragonFlyBSD Virtual Kernels

- very similar to User-Mode Linux
- part of DFlyBSD since 2007
- uses standard `libc`, unlike UML
- paravirtual ethernet, storage and console

User Mode Kernels

- 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

Migration

- 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

Part 11.3: Management

Disk Images

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- bundling makes deployment easier
- the bundled components have known behaviour
- but updates are much trickier
- this also prevents resource sharing

Security

- 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

- 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

- 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

- 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

- 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

Part 12: Review

What is an OS made of?

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

Basically every OS has those.

The Kernel

- 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

- 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

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

(System) Libraries

- 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

- 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

- 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?

- 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

- 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

- 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

- 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

- 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?

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

Kernel Architecture Types

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

System Call Sequence

- 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?

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

Disk-Like Devices

- 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)

- 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

- 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

- 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

- 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

Fragmentation

- 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

- 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

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

Process Memory

- 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

- 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?

- 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

- 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

- 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

- 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?

- 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

- 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

- 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?

- 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?

- 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

- 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

- (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

- 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

1. mutual exclusion
2. hold and wait condition
3. non-preemtability
4. circular wait

Deadlock is only possible if all 4 are present.

Starvation

- 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?

- 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

- 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

- 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

- 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

- allows the device to directly read/write memory
- this is a huge improvement over programmed IO
- interrupts only indicate buffer full/empty
- the device can read and write arbitrary physical memory
  - opens up security / reliability problems

GPU Drivers

- 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

- 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

2. Link (Ethernet, WiFi)
3. Network (IP)
4. Transport (TCP, UDP, ...)

7. Application (HTTP, SMTP, ...)

Networking and Operating Systems

- 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

- 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)

- 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

- 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

- 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

- 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)

- 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

- programming language centered on OS interaction
- rudimentary control flow
- untyped, text-centered variables
- dubious error handling

Interactive Shells

- 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

- 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

- 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

- 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)

- a traditional UNIX windowing system
- provides a C API (`xlib`)
- built-in network transparency (socket-based)
- core protocol version 11 from 1987

Users

- originally a proxy for people
- currently a more general abstraction
- user is the unit of ownership
- many permissions are user-centered

User Management

- 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

- 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

- 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

- 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

- 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

- 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

- type 1: bare metal
  - standalone, microkernel-like
- type 2: hosted
  - runs on top of normal OS
  - usually need kernel support

Paravirtual Devices

- special drivers for virtualised devices
  - block storage, network, console
  - random number generator
- faster than software emulation
  - orthogonal to CPU/MMU virtualisation

VM Suspend & Resume

- 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?

- OS-level virtualisation
  - e.g. virtualised network stack
  - or restricted file system access
- not a complete virtual computer
- turbocharged processes

Bundling vs Sharing

- bundling makes deployment easier
- the bundled components have known behaviour
- but updates are much trickier
- this also prevents resource sharing

The End

Actually...

- a 2-part, written final exam
- test: 9/10 required
  - pool of 44 questions (in the slides)
- free-form text
  - one of the 11 lecture topics
  - 1 page A4: be concise but comprehensive