

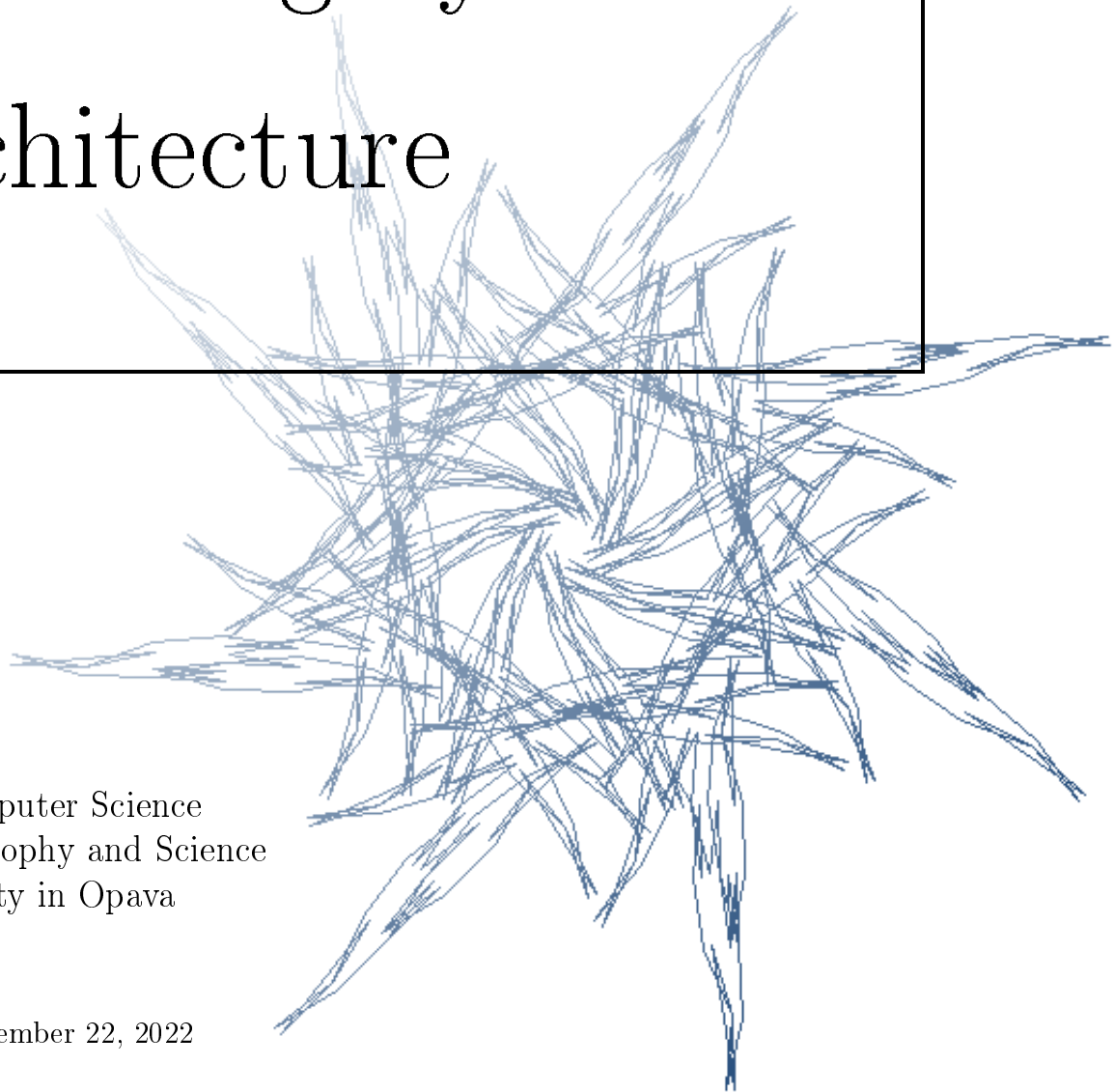


**SLEZSKÁ
UNIVERZITA**
FILOZOFICKO-
PŘÍRODOVĚDECKÁ
FAKULTA V OPAVĚ

Šárka Vavrečková

Study material

Operating Systems Architecture



Institute of Computer Science
Faculty of Philosophy and Science
Silesian University in Opava

Opava

Last updated: December 22, 2022

Summary: This document contains the material for the *Operating Systems Architecture* course. In particular, we deal with the structure of operating systems, memory management, processes and devices – in general and specifically in Windows and Linux operating systems.

Operating Systems Architecture

RNDr. Šárka Vavrečková, Ph.D.

Institute of Computer Science
Faculty of Philosophy and Science
Silesian University in Opava
Bezručovo nám. 13, Opava

Typeset with L^AT_EX

Preface








What we can find in this document



The lectures of the Operating Systems Architecture course discuss mainly theoretical concepts related to the structure of operating systems, the roles of individual parts of the kernel, and mechanisms for managing processes, memory, and devices, but each topic is then related to specific operating systems (usually Windows and Linux).

Some paragraphs or sections are “additional” (marked with purple icons), these are not discussed and they do not appear on the exam – their purpose is to motivate further independent study or experimentation or to assist in the future in acquiring further information. If there is a purple icon in front of a chapter (section) title, it applies to everything in that chapter or section.


Marking

We use the following colourful icons:

-  *Quick Preview*, in which we find out what it’s going to be about.
-  *Keywords*.
-  *Study Objectives* for a chapter tell us what new things we will learn in the chapter.
-  New *terms* are marked with the blue symbol, seen here on the left. This icon (as well as the following) can be found at the beginning of the paragraph in which the new concept is introduced.
-  *Methods*, procedures and tools (commands, programs, files, scripts), ways of solving various situations that an administrator may find himself in, etc. are also marked with a blue icon.
-  Some parts of the text are marked with a purple icon, indicating that they are *optional sections* that are not discussed (mostly; students can request them or study them on their own if they wish). Their purpose is to voluntarily expand students’ knowledge of advanced topics that usually don’t get much time in class.
-  The yellow icon indicates links where you can get *further information* about the topic. Most often, these icons are web links to sites where the authors go into more detail about the topic.

-
-  Red is the icon for *warnings* and notes.
 -  This marking means that you can choose between several alternatives for the test. Typically, this is a decision about whether to explain a procedure, term, structure, etc. on Windows or Linux.


If the amount of text belonging to a certain icon is larger, the whole block is delimited by a space with icons at the beginning and at the end, for example to define a new concept:

 **Definition**

In such an environment, we are defining a concept or explaining a relatively familiar but complex concept with multiple meanings or properties.




Similarly, the environment may look like a longer procedure or a longer note or more links to more information. Other environments may also be used:

 **Example**

This is what the environment looks like with an example, usually of a procedure. The examples are usually annotated to make it clear how to solve them.



 **Tasks**

Questions and tasks, suggestions for testing, which are recommended for practicing the material, are enclosed in this environment. If there are more than one task in the environment, they are numbered.



Contents


Preface	iii
1 Introduction to Operating Systems	1
1.1 What the Operating System Is	1
1.2 Operating System Functions	2
1.3 Types of Operating Systems	3
1.3.1 Basic Categorisation	3
1.3.2 Realtime Operating Systems	4
1.3.3 Distributed Operating Systems	5
2 Operating Systems Structure	8
2.1 Basic Types of Architectures	8
2.2 Layered Structure of Operating Systems	9
2.3 MS Windows	10
2.3.1 Older Windows – XP	10
2.3.2 Windows Vista/7/8/10/11	14
2.4 UNIX and UNIX-like Systems	16
2.4.1 UNIX Standards	16
2.4.2 Architecture of UNIX Systems	17
2.4.3 Linux Kernel	19
3 Memory Management	20
3.1 Memory Management Basics	20
3.1.1 Memory Strategies	20
3.1.2 Allocated Memory Space	21
3.1.3 Garbage Collection	22
3.2 Complex Memory Separation Models	22
3.2.1 Virtual Memory and Addresses	22
3.2.2 Paging	23
3.2.3 Segmentation	26
3.2.4 Paging with Segmentation	28

3.3	Virtual Memory Concepts	29
3.3.1	Page Replacement Algorithms	29
3.3.2	NUMA	30
3.4	Memory Management in Windows	31
3.5	Memory Management in Linux	32
4	Processes	34
4.1	Multiple CPU Cores	34
4.2	Obtaining Process Information	35
4.2.1	Processes in Windows	35
4.2.2	Processes in Linux	36
4.3	Process Concept	38
4.3.1	Program, Process, Thread	38
4.3.2	Process States	41
4.3.3	Process Control Block	42
4.4	Operations on Processes	43
4.4.1	Process Input and Output	43
4.4.2	Process Creation and Termination	46
4.4.3	Priorities	47
4.5	Multitasking	48
4.5.1	Context Switching	48
4.5.2	Types of Multitasking	49
4.6	Multithreading	50
4.7	Interprocess Communication	51
4.7.1	IPC concept	51
4.7.2	IPC in Windows	53
4.7.3	IPC in Linux	55
4.7.4	Jobs in UNIX	59
4.8	CPU Scheduling	64
4.8.1	Basic Concepts	64
4.8.2	Scheduling Algorithms	64
4.8.3	Scheduling in Windows	66
4.8.4	Scheduling in Linux	67
5	File Access and Permissions	70
5.1	File Access Permissions in Linux	70
5.1.1	Owner and Associated Group	70
5.1.2	Setting File Access Permissions	71
5.1.3	Special Permissions	72
5.2	Working under Different User Account in UNIX systems	77
5.2.1	The <code>su</code> Command	77
5.2.2	The <code>sudo</code> Command	79
5.3	Advanced Access Control Mechanisms in Linux	81


5.3.1	Attributes	81
5.3.2	POSIX ACLs	82
5.3.3	PAM	84
5.4	Security Policy Settings in Windows	85
6	Synchronization	88
6.1	Why Synchronize	88
6.2	Petri Nets	89
6.3	Basic Synchronization Tasks	90
6.3.1	Critical Section	90
6.3.2	Producer-Consumer Problem	92
6.4	Synchronization Tools	94
6.4.1	Waiting	94
6.4.2	Mutexes and Semaphores	96
6.4.3	Messages	97
6.4.4	Monitors	98
6.5	Additional Synchronization Problems	99
6.6	Synchronization in Operating Systems	103
6.6.1	Possibilities of Synchronization in Windows	103
6.6.2	Possibilities of Synchronization in Linux	105
7	Deadlock	109
7.1	Deadlock Characterization	109
7.1.1	Model	110
7.1.2	Resource-Allocation Graph	110
7.1.3	Deadlock Conditions	112
7.2	Deadlock Treating	112
7.3	Prevention	113
7.4	Avoidance	115
7.4.1	Safe State	115
7.4.2	Resource-Allocation Graph Algorithm	115
7.4.3	Banker's Algorithm	116
7.5	Detection	118
7.5.1	Wait-for Graph	118
7.5.2	Banker's Detection Algorithm	119
7.6	Recovery from Deadlock	120
8	I/O Management	121
8.1	I/O Devices	121
8.1.1	Types of I/O Devices	121
8.1.2	I/O System	122
8.1.3	I/O Buffering	123
8.2	Device Drivers	124


8.2.1	Drivers in Windows	124
8.2.2	Drivers in Linux	127
8.3	Interrupts and Exceptions	131
8.3.1	Mechanism of Interrupts and Exceptions	131
8.3.2	Interrupt Handling	131
8.3.3	Managing Interrupts in Various Systems	132
8.4	Running Non-Native Applications	133
8.4.1	Virtual Machine	134
8.4.2	Operating System Emulators and Subsystems	135
8.4.3	Server and Desktop Virtualization	136
9	Storage Media	138
9.1	Disks	138
9.1.1	Disk Format	138
9.1.2	Addressing	141
9.1.3	Scheduling	142
9.2	File Systems	143
9.2.1	File organization and Access mechanism	144
9.2.2	Directories	144
9.2.3	File Sharing	145
9.2.4	Journaling File Systems	146
9.3	Windows File Systems	147
9.3.1	Older Windows File Systems	147
9.3.2	NTFS	150
9.3.3	exFAT	152
9.3.4	Protection	153
9.3.5	Handling Partitions and File Systems	155
9.3.6	Comparison of Windows File Systems	158
9.4	Linux File Systems	159
9.4.1	VFS	159
9.4.2	File Database Structure	159
9.4.3	Hard Links and Soft Links	162
9.4.4	File Systems of the Type <code>extxfs</code>	163
9.4.5	Other Journaling File Systems	164
9.4.6	Comparison of Linux File Systems	165
9.4.7	Virtual File Systems	165
9.4.8	The <code>fstab</code> File	167
9.4.9	Handling Partitions and File Systems	170
	Bibliography	172

Introduction to Operating Systems

 *Quick preview:* This chapter is an introduction to the subject, we will learn the basic concepts, definition of an operating system, functions and types of operating systems.


In the following we will understand the term operating system a little more broadly than usual. We will also include software that is used to control any computer system, including programmed laser printers (i.e., also firmware).

 *Keywords:* Computing system, physical resources, logical resources, multitasking, multiprocessing, realtime operating system, distributed operating system.

 *Objectives:* The aim of this chapter is to give an introduction to operating systems. You will be able to explain what an operating system is and what types of operating systems exist.

1.1 What the Operating System Is

Under the term *computing system* we will understand any system performing automated calculations, i.e. not only a computer, laptop or tablet, but also a smartphone, smart watch, TV, network device, . . .

 *The physical resources* of a computing system are the components that the system uses in its operation: CPU, memory, storage media, input/output devices, etc.

The logical resources of a computing system also serve the system to perform its operations, but they are not physical components. Here we will clarify a few terms:

Instruction is the shortest, indivisible command intended to processor.

Contract is the assignment to be executed by a computing system.

Task (job) is the sequence of activities required to fulfil the contract, i.e. the specification of the procedure for solving the contract.

Task step is a part of a task, an element of a task execution sequence usually representing the execution of a specific program (a task can be a sequence of multiple programs that run simultaneously or sequentially).

Process is an instance of a task or a task step, it is executed in memory using specific data.

Processes, their abstract form (tasks, etc.) and their parts (instructions) are the logical means of the system.



Definition

The *Operating System* of a computing system is the manager of the physical resources of the system, which processes user-specified tasks using logical resources. The *software platform* of a system is usually understood as the operating system.



1.2 Operating System Functions

An operating system has many functions, some of which are necessary and arise from the definition of the operating system, others are not so necessary and not every operating system provides them. The following list is not exhaustive, specialized operating systems may provide many other functions. Separate chapters are devoted to the most important functions.

The main functions are the following:

Memory management means keeping records of memory, allocating memory to processes, dealing with memory shortage situations, managing virtual memory.

Processes management means keeping records of running processes, CPU scheduling, monitoring, inter-process communication.

I/O management means providing interfaces between I/O devices and system/processes, monitoring device status, resolving potential conflicts, etc.

System management – in modern systems it is common to distinguish different modes of system operation, at least user and privileged mode. In user mode normal activities take place, while privileged mode is for maintenance, installation, configuration. We can also include here the security features of the system – protection against malicious code (e.g. viruses), malfunctions and unauthorized access.

Files management means not only creating an interface that allows processes to access files (and other data) in a uniform way, but also maintaining information about the file structure on disk, controlling the access rights of processes to files.

Users management – the system maintains information about users and their activities, ensures logging in and logging out of users, data protection, especially the mutual separation of the space of individual users.


User interface is the interface between the user and the system. It is a set of programs and libraries used to communicate between the user and the operating system.

Program interface is the interface between programs (processes) and the computing and operating system, usually called API (Application Programming Interface). It is usually represented by a set of libraries that a program can use to do its work (graphical interface elements, dialog boxes, function elements, etc.).

1.3 Types of Operating Systems

1.3.1 Basic Categorisation

We will categorise operating systems according to various criteria.

 **Based on the number of controlled CPUs** we divide


- *single-processor systems* – Windows with the DOS kernel (95, 98, ME),
- *multi-processor systems* – UNIX-like systems including Linux, and Windows with the NT kernel (NT, 2000, XP, Vista, 7, 8, 10, 11), can schedule at least some tasks so that they can be processed on multiple processors simultaneously. UNIX-like systems in general can run on clusters with a large number of processors, while Windows depends on the specific edition and license (even common desktop editions support two processors).

We divide multiprocessor systems into two subcategories:

- when using *asymmetric multiprocessing* (ASMP), one processor is dedicated to system processes, and user processes run on other processors,
- with *symmetric multiprocessing* (SMP), any process can run on any processor.

Almost all modern systems use symmetric multiprocessing.

In fact, even in common desktop computers, which we do not refer to as multiprocessors, we find multiple processors. One of them is the main processor, the others are dedicated to specific activities and their purpose is to relieve the main processor from “routine” or special activities and to speed up the work of the whole system. For example, the graphics processor on the graphics card has such a function, taking over in particular the processing of 3D graphics requests. It is not a multiprocessor system, because the auxiliary processors do not process the common instruction set, but only their specific instruction set. OpenGL, Direct3D, etc. are examples of technologies using special graphics instruction sets.

 In multiprocessor systems (especially servers) we can encounter the *NUMA* (Non-Uniform Memory Access) architecture. Memory is divided into separate parts, nodes, and to each such node one or more processors are connected by bus (each node has its own memory bus). A processor can access the memory in “its” node very quickly, while the memory in other nodes can also be accessed, but slower. Therefore, the processor should primarily use local memory, on its own node.

The purpose of the NUMA architecture is to make the communication of processors with memory as efficient and scalable as possible, because in multiprocessor systems without NUMA the memory bus is a bottleneck that slows down the whole system; moreover, the address range is limited by the number of bits used to store the address, and NUMA allows to bypass this limit (each node has its own address space).

 **Based on the complexity of user management** we divide operating systems into


- *single-user* – Windows with the DOS kernel,
- *multi-user* – UNIX-like systems, Windows with an NT kernel, have sophisticated user management that allows multiple users to work on the system simultaneously (at the same time), users can log in either directly on the device or remotely via terminals or otherwise over network. In particular, these systems must ensure strict separation of resources (e.g. memory) used by different users, so that one user cannot access the data and settings of another user.

 **Based on the number of running programs** we distinguish operating systems

- *single-program* – only one program can be running in one time,
- *multi-program* – multiple programs can be running in one time parallelly, We further distinguish here the subgroup *multitask* systems, which allow, in addition, resource sharing between the processes of these programs (memory management etc.).


Multi-program systems that do not allow multitasking (i.e., they are *single-task*) solve this problem, for example, by deferring all memory space of the “settled” program to storage media or to a protected part of memory, and then restoring the state when the program should resume its operation.


1.3.2 Realtime Operating Systems

 Realtime operating systems operate in *almost* real time. They are used in particular where there are high requirements for system interactivity, where tasks must be completed almost immediately or in a suitably short time. These are, for example, systems for controlling aircraft, some factories, laboratories, power stations including nuclear power stations, in the automotive industry, etc.

The realtime system does not have to respond immediately, only a “upper time limit” is specified for each type of realtime request, i.e. the maximum response time in the worst case must be guaranteed. Ordinary operating systems with multitasking cannot guarantee this, especially if many processes are running, although they usually offer the possibility to assign a process a *realtime priority* significantly higher than the priority of ordinary processes. Still, there are ways to modify these operating systems to work as realtime.

Realtime priority also exists in classical operating systems, but it means only very high priority, it is not about guaranteeing processing time.


 Most realtime systems have a small kernel (*microkernel*) that performs only the most important functions (mainly process management, possibly memory management, etc.), the rest of the system is implemented as normal processes. This model corresponds to a client-server structure. If the system was created by rewriting a classical system, often the kernel of the original system is a microkernel and runs as only one of the processes (this is the case for many modified UNIX-like systems).

 **QNX** (pronounced [kju:nix]) is a realtime system based on a much modified UNIX clone, used mainly in cars and various embedded devices. It has a small microkernel and several most important servers (process management, memory management, etc.), the rest of the system runs as normal processes. It is compatible with the POSIX standard.


It is characterized by exceptional stability and speed, even when working in a graphical interface. It runs well even on weaker computers. It has excellent network support, and can also be used to access the Internet if the hard disk is unavailable for some reason.


The disadvantage is the lack of applications for this system, but since it is actually a UNIX system, it is not such a problem to port UNIX applications to QNX (there are many applications adapted for QNX on the Internet, including the manufacturer’s site).


It is a commercial system, currently it is a completely closed system (under the BlackBerry company).

 **RTLinux** is a modified Linux, it was originally designed mainly for industry (robot control in manufacturing, etc.). It was originally a commercial project, but commercial support is no longer provided and the successor of this project is RT-Preempt Patch (see below).

It has a realtime microkernel, the Linux kernel itself runs as a separate process with lower priority. The system has been designed to make as little interference with the original Linux as possible. Interrupt handling¹ (i.e., requests that could possibly be realtime) is handled by first intercepting the interrupt by the microkernel, and then, when the CPU time does not require any realtime process, the interrupt is passed to the original Linux kernel, which already handles it in the classical way.

 **The Realtime Preemption patch** (RT-Preempt Patch) is a special update (patch) for the Linux kernel that modifies the kernel to work realtime. This is mainly to modify the kernel's synchronization mechanisms (there is a separate chapter on synchronization in this document), timers and interrupt handlers. One of the areas of application is also industry.

 **RTX** (RealTime eXtension) is a module that extends the capabilities of Windows with NT kernel (NT/2000/ XP/7, 32-bit only) towards realtime systems. So it is not a realtime system, just an extension for Windows (basically a patch like RT-Preempt).

 **Realtime systems for Internet of Things.** Because IoT devices may need special handling, including guaranteed response time, special real-time operating systems exist today for this area as well. Another feature of these systems is their low energy consumption, as these devices are often battery-powered. Examples of RT systems used for the Internet of Things are the following:

- Contiki-NG,
- TizenRT,
- RT Preempt Patch,
- FreeRTOS,...



Additional information

- <https://blackberry.qnx.com/en>
- <http://www.tldp.org/HOWTO/RTLinux-HOWTO.html>
- https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO
- <https://www.contiki-ng.org/>
- <https://docs.tizen.org/application/tizen-studio/rt-ide/overview/>
- <https://www.freertos.org/>
- https://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems



1.3.3 Distributed Operating Systems

 Distributed system (we can also find the term *grid*) is a system that meets these conditions:

- runs on more than one processor (it can also be a properly designed and managed computer network),

¹Interruption of the normal execution of a program. For example, it can be a keyboard-generated interrupt (when a key is pressed, the program must know about it and react appropriately).


- has its program divided into (separate) parts that communicate with each other (usually by network protocols or via remote procedure calls – RPC, it is also possible to use object interfaces designed for this purpose – DCOM, CORBA, etc.),
- each such part is (can be) processed on a different processor, ensuring as much transparency as possible.

Distributed means that the computation of the system can be spread as far as possible over multiple sites working in parallel. There are two kinds of distributedness:

distribution with coarse granularity (coarse-grained) – parts of the system tend to be larger, more independent, communicate less with each other, applicable when there is a problem to ensure good and fast communication (less interconnection of computers – processors in the system),


distribution with fine granularity (fine-grained) – parts of the system are as small as possible, they communicate a lot with each other.

There are two kinds of distributed systems – distributed applications and distributed operating systems.

 **Distributed Application** is a distributed system running on multiple connected computers, each of the computers has its own operating system. This network of computers can also be the Internet. Distributed applications are encountered, for example, in the following cases:

- *distribution of data* – databases, content management systems, information systems, etc. – it is necessary to share data in many branches around the world,
- *distribution of computation* – complex calculations are distributed across multiple computers,
- *distribution of resources* – resources owned by one entity (e.g., processing power, storage, etc.) can be distributed (offered) to other entities, with the location and specific method of access hidden from those entities.

A typical and very common use of distributedness is in databases and (often related) information systems. Large databases and information systems tend to be distributed across many nodes and across the globe, although not always a truly distributed application that meets the requirement of transparency and flexibility (discussed later).

 Let us look at a few specific uses of distributed applications.

One of the most well-known distributed applications is *BOINC* (short for Berkeley Open Infrastructure for Network Computing²), which allows any user of a computer connected to the Internet to lend the computing capacity of their computer to a project using this application. These projects are, for example, Climateprediction.net (worldwide weather forecast), SETI@home (analysis of radio signals potentially coming from extraterrestrial civilizations), Einstein@home (search for gravitational waves generated by pulsars), MalariaControl.net (monitoring and predicting the spread of malaria), several projects in biomedicine (cells, proteins, etc.), etc.


Grids can be created at home, there are tools for creating grids in a small home network (used for time-consuming operations such as long compiling software from source – Gentoo Linux, multimedia processing, etc.).

In the context of Linux, we should also mention *distributed version control systems*. A version control system allows a group of programmers to work on the same project efficiently enough. The

²Information about BOINC can be found at <http://boinc.berkeley.edu>.

distributed version allows you to have code in multiple places (e.g. on programmers' machines). It is mainly about synchronizing accesses and changes in the source code, the system keeps a history of changes for each registered file, the last few versions, information (metadata) about the files and their authors, and also reacts in a certain way in case several users of the system want to change the same file – either the first accessing file is locked or “change merging” is performed.

There are quite large groups of programmers working on Linux programs and on Linux itself, physically located in different parts of the world. Therefore, it is often necessary to use a version control system that is distributed to synchronize their work fast enough (but for smaller projects this feature is unnecessary). Until recently, Linux developers used BitKeeper, but mainly for licensing reasons they are switching to the new *Git* system created by Linux creator Linus Torvalds himself. Git is not a full-fledged version control system, although it is sufficient for these purposes (it is also a distributed system). A variant of it, extended with additional scripts, *Cogito*, which is already a full version control system, is being promoted (by Petr Baudiš from Czechia).

 **Distributed Operating System** is a standalone operating system running on a network of processors that do not share a common memory, while giving the user the impression of a single computer.

Although it is physically located on different computers, this does not (should not) affect its operation, and the user does not determine where exactly his data is processed or where it is actually stored. In the following, we will focus only on distributed operating systems.


The basic features of a distributed operating system are:

1. transparency – the structure or procedure is not visible,
2. flexibility – the ability of the system to adapt to any changes in the environment in which it operates, including various failures and outages of parts of the system. It is also related to the migration transparency property,
3. scalability – increasing number of connected machines should not be a problem,
4. openness, reliability, sufficient performance, . . .

Transparency is the most important feature of a distributed operating system, it implies a certain impression of uniformity of the system for users and possibly for processes. This feature refers primarily to the relationship between processes and resources of the whole system.


Transparency can be understood in different ways, e.g.:

- access transparency – the process accesses local and remote resources in a uniform manner,
- location transparency – the process has not knowledge on the physical location of resources,
- migration transparency – resources can be freely moved and connected to different parts of the distributed system without affecting the operation of processes,
- execution transparency – processes can run on any processor and can even be moved to another processor while they are running in order to appropriately balance the load of different parts of the system,
- . . .


 **Existing distributed operating systems:** For example, LOCUS used to be popular (LOCUS is compatible with UNIX). Nowadays, there would be a place for distributed operating systems at clusters in data centres, where we can most often find some Linux distribution adapted to run in a cluster (and thus distributed to the devices that are connected in the cluster). Examples: Red Hat Enterprise Cluster, SUSE Linux Enterprise with High Availability Extension.


Chapter 2

Operating Systems Structure

 **Quick preview:** To understand how operating systems work, we need at least some basic information about their structure. In modern operating systems, the structure is designed primarily with security and stability of the whole system in mind, and there is always a division into a privileged part (privileged mode, kernel mode) and a user part (user mode, non-privileged mode), with processes running in the user part not being able to interfere in any way with the privileged part. Of course, more simple systems also have their own structure, they often just need a more simple structure.


In this chapter, we'll first discuss the basic types of structures, and then we'll look at the structure of some specific operating systems in the Windows and UNIX families.

 **Keywords:** Architecture, structure, kernel, privileged mode, user mode, Windows, UNIX, Linux, (security) ring.

 **Objectives:** The main goal of this chapter is to be familiarized with the structure of common operating systems.

2.1 Basic Types of Architectures


The following terms (structure types) apply not only to computing systems as a whole, they are also generally used, for example, for the structure of a system's kernel or the layers in the kernel.

 The *monolithic structure* means that the given system consists of the single file or single component. All common operating systems have the monolithic kernel, loaded from one file.

Example

The Windows kernel is loaded from the file `ntoskrnl.exe`, the Linux kernel is loaded from `vmlinuz...` (a version label is added).




 All operating systems have *layered architecture* as well, they are organized as a hierarchy of layers, where each layer communicates with the neighbor layers. This type of structure lies primarily in splitting the system into two parts with a different type of access to resources that is supported by


hardware:

- code executed in kernel mode (kernel space) – full access to resources,
- code executed in user mode (user space) – access only through system calls.


These parts can be divided into multiple (sub)layers. E.g. the kernel space is divided to the hardware-dependent layer and hardware-independent layer.


 The *modular structure* is common too. Operating systems have the monolithic kernel with dynamically loadable modules. The modules extend the functionality of the kernel, this structure ensures scalability of the whole solution. The kernel loads only the needed modules.

As modules we can introduce device drivers, firewall, network protocols implementation, and so on.

 The *virtual machines* (VM) are intended to separate a code from the rest of the system and to provide a special API for the isolated code. The common operating systems use this concept for running non-native applications (programmed for different systems, different hardware variants or old versions, for backward compatibility).

We also know this concept as special applications for running whole virtualized operating systems, such as Oracle Virtual Box or VMWare Workstation.

 The *client-server model* differentiates the server and client processes (the both types can run in the user mode, by the particular architecture). Server (system) processes provide services to client processes.

 The *microkernel architecture* (multiserver architecture) is a special type of a kernel, where only the most important parts remain inside the kernel (and work in kernel mode) – e.g. process management and IPC (Inter-Process Communication), memory management, processor scheduling, and the rest is implemented as a set of system processes working in the user space, including all drivers, file systems, . . .

Hybrid kernels have their properties between microkernels and monolithic kernels.

The main advantages of microkernel computers to monolithic kernels are high system stability, fault tolerance and attacks tolerance. The main disadvantage is worse performance.

The microkernel architecture is typical mainly for real-time systems, for example the QNX system has microkernel, PikeOS for embedded systems, or L⁴Linux, or Symbian OS.

Another example is the Mach kernel, but not all versions a variations of Mach are microkernels. Its variant called GNU Mach is microkernel, but Darwin (variant used in Apple MacOS) is a hybrid kernel.

 **Additional information**

- <https://blackberry.qnx.com/en>
- <https://www.gnu.org/software/hurd/microkernel/mach/history.html>



2.2 Layered Structure of Operating Systems

Modern processors use registers to implement hardware resource protection.

On Intel and AMD processors, this protection is implemented in the form of four *rings*. Each process runs in one of these rings, which determines its access to protected resources.

Everything in ring 0 has direct access everywhere (to system resources and hardware). Ring 1 is a bit limited, ring 2 means more limitations, and processes in ring 3 have no direct access to system resources and hardware, and all their needs are performed with system calls.

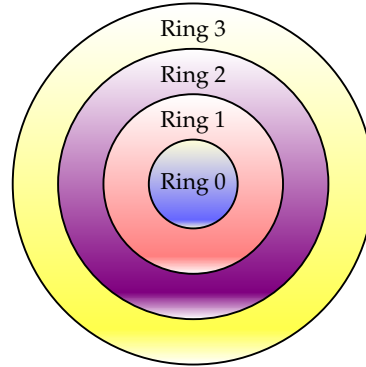


Figure 2.1: Security rings on Intel processors: farther from center = less privileges

Most operating systems use only two rings – ring 0 for the kernel and ring 3 for common processes. Ring 0 represents the kernel mode, ring 3 the user mode.

The kernel mode is usually divided to the hardware-dependent layer (hardware abstraction layer and device control), main kernel with loaded modules, and executive layer providing system calls from user space.



Additional information

The principle of rings is described in detail in <https://manybutfinite.com/post/cpu-rings-privilege-and-protection/>.



2.3 MS Windows

The Windows NT kernel was developed independently of MS-DOS, and the typical use of the system as a server or client on a network was already taken into account in its design, so stability and security options are the main considerations. The inspiration of UNIX systems is evident throughout the concept.

The system was designed as a multiprocessor (SMP – symmetric multiprocessing) multi-user multitask universal network system.

2.3.1 Older Windows – XP

Some elements are similar to parts of the Windows structure with a DOS kernel, but work differently internally. In particular, the division into two basic parts is important – the part running in *privileged mode* (kernel mode) and the part running in *user mode*.

Windows use the layered model, as we can see on Figure 2.2. It has monolithic kernel (loaded from `ntoskrnl.exe`) with loadable modules. The modules are loaded mainly from dynamic-link libraries in `C:\Windows\System32` (also in 64-bit system) – suppose that `C:` is the system partition. And the principle of client-server model is used in API.

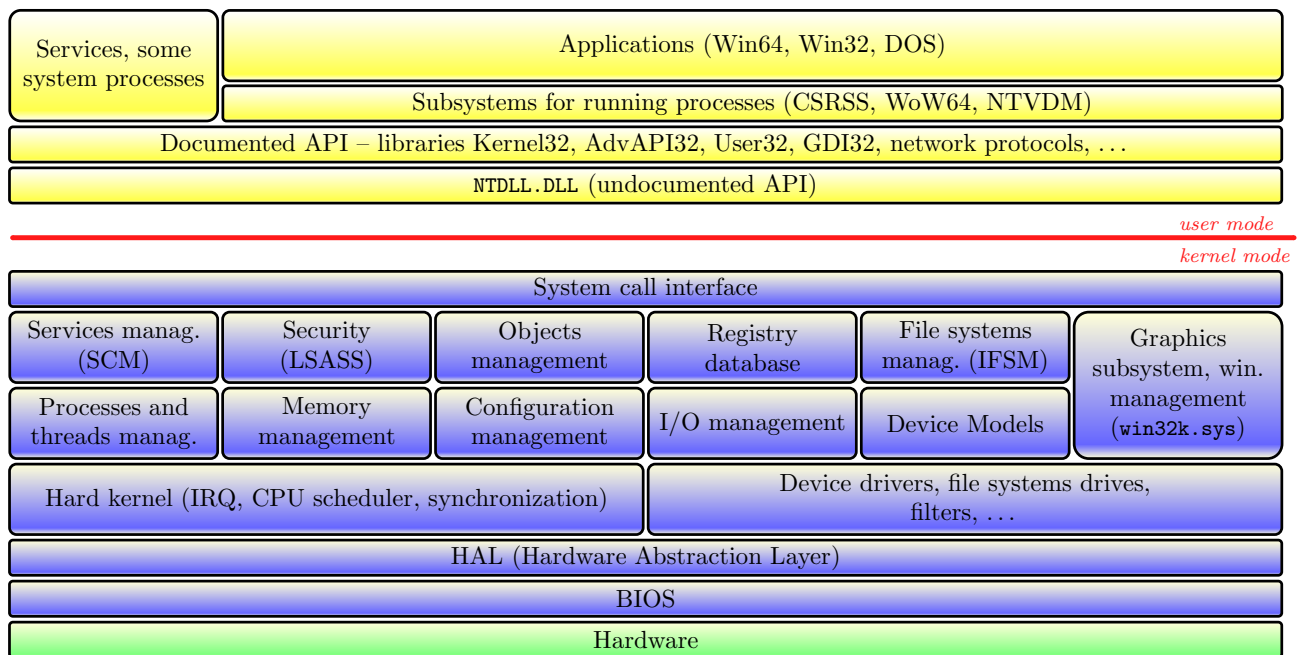






Figure 2.2: Structure of older Windows NT (till Windows XP)

 *HAL* is the Hardware Abstraction Layer, the interface between the hardware and the rest of the system kernel. It is loaded from the `HAL.DLL` file when the system boots. It is separated from the rest of the system to facilitate system portability between the (few) hardware platforms. Drivers communicate with devices only indirectly through this layer.

 *Kernel* (the so-called “main kernel”) and *executive* are physically stored in `NTOSKRNL.EXE`.¹ The exe is not directly visible from the picture, we can imagine that it is nearly everything inside the kernel (blue area) above the HAL layer.

The kernel captures and handles interrupts, performs processor management (synchronization of processor allocation), etc. The Executive is the control process of the operating system, it is responsible for managing the entire kernel running in privileged mode and the operation of modules, it mediates communication between various parts of the kernel and modules. The kernel and the executive are loaded from the file `NTOSKRNL.EXE` (a single file, i.e. the monolithic kernel) at system startup, other kernel components are linked to the kernel from dynamic libraries or `.SYS` files (modules, i.e. the modular structure of the running kernel).


 *Main system process* (in process management applications we see it under the name “System”) is actually a picture of what is running in the kernel, exported to user space (of course we can’t see it directly in the kernel). In fact, it is not a process, but a *container* for kernel threads (we will learn more about threads in the chapter on process management).


 *Drivers* are not just device-related, they are generally kernel modules that can be used to access devices, buses, etc., but they can also be filters that data passes through (encryption, compression,


¹In fact, it is not just `NTOSKRNL.EXE`. In a multiprocessor (or multi-core) system it is replaced by `NTKRNLMP.EXE`; in a system with the PAE function enabled (accessing memory located after the addressable memory), it is `NTKRNLPA.EXE` for a single-core system and `NTKRNPAMP.EXE` in a multi-core system. This applies to the naming of files on the installation CD; after installation or upgrade to a multiprocessor or PAE system, only the names `NTOSKRNL.EXE` or `NTKRNLPA.EXE` can be found in the filesystem (the originally differently named version is renamed during installation).


inter-module routing, filtering/aggregation/sorting, DRM, etc.).


There is a complex system for working with drivers, which involves several components marked in the picture, for example *Device Models* define a unified way of handling drivers.

 *IFSM* is an Installable File Systems Manager, it manages different types of file systems, e.g. NTFS, FAT16, VFAT (FAT32 with extensions), CDFS (for CD-ROM), UDF (for DVD), etc., through this component it communicates with storage devices (everything that corresponds to the *mass storage* standard with a file system that IFSM understands), including the cache memory.


 *Virtual Machine Manager* (VMM) controls the Windows options for concurrency with programs intended for (supported) different platforms. The *Virtual Device Drivers* (VxD) are the controllers that the virtual device manager needs to handle I/O devices for legacy programs.

 *Configuration Manager* cooperates in the management of drivers, for example it provides the Plug&Play and HotPlug functions, i.e. it constantly monitors the status of the buses and monitors the connection of new or previously connected devices, for new ones it tries to perform the installation and initialization procedure.

 *Security SubSystem* is mainly related to the LSASS (Local Security Authority SubSystem) module. It authenticates users who log in locally and determines access permissions according to the database in the SAM registry key.


 The *Service Control Manager* (SCM) is loaded from the `services.exe` file and ensures the running of *services* and communication with them. The services themselves run in user space, but usually with higher permissions and without binding to a specific user, and are communicated with primarily through the SCM module.

Since Windows NT version 4, the kernel is more or less object oriented. A database of objects is maintained in the kernel, and executive objects are exported to user space. The object database is maintained by *Object Manager*.

 *API kernel interface* represented by a series of *dynamically linked libraries* (containing functions, objects, etc.) are used by processes to access the system.

Documented interface represent functions and objects from system libraries whose names should be familiar – `User32.dll`, `GDI32.dll`, and others. Their purpose is basically similar to what we learned on older systems, the differences are in the way they are programmed.


The *Undocumented API* is the `NTDLL.DLL` file. The functions found here can already directly trigger system calls – so it might seem better to use the undocumented API right away (it would be faster), but the problem is that the functions provided by this interface may be different in each version of Windows and may require a different method of calling (triggering). An unpleasant consequence is that an application using an undocumented API may not work at all in some versions, or we may encounter various incompatibility issues when running it. Therefore, it is better to use a documented API that always behaves the same (documented) way. So the `NTDLL.DLL` file is a kind of interface between the kernel and the user space, but we usually access this interface indirectly.


 *Subsystems* in the user mode are interfaces that ensure that different types of processes run correctly and safely. These subsystems run applications that may not even be compatible with Windows NT. Subsystems provide an interface to applications that translates the communication (requests for information, resources, execution of a particular action, etc.) between the application and the operating system so that both parties can understand each other.


It is mainly a subsystem for applications programmed for 32-bit Windows, MS-DOS and applications for 16-bit Windows *Win32* (the only subsystem for all these types of applications, in which any virtual machines are then run), a subsystem for OS/2, *POSIX*, etc.

The subsystem for 32-bit Windows including NT (subsystem *Win32*, in 64-bit system it is simply called *Windows*) is represented by the file `CSRSS.EXE`, for *POSIX* it is mainly the file `PXSS.EXE` (it is the subsystem server). The Win32/Windows subsystem is also needed for running many system processes, so it is the only one that starts right after the computer boots, the other subsystems are started only on request when an application belonging to this subsystem is started.


Each subsystem needs, in addition to its control program (for example `CSRSS.EXE` in Win32), also libraries in which functions and objects are stored and which contain the API (Application Programming Interface) of the subsystem. For example, the Win32 subsystem libraries also include `KERNEL32.DLL`, `USER32.DLL`, and `GDI32.DLL`. While these three modules are intended for the Win32 subsystem, to avoid having to implement these functions in each subsystem separately, calls to graphical functions of other subsystems are translated to calls in the Win32 subsystem.

 The Win32/Windows subsystem includes a virtual machine mechanism. The application that physically runs virtual machines for legacy applications (DOS and Win16) is started by the `ntvdm.exe` file (NT Virtual DOS Machine). When attempting to run these applications, a new instance of `ntvdm.exe` is first started with the parameter – the name of the DOS or Win16 application being run with a path that already executes the specified application internally.

 On a 64-bit system, the situation is a bit more complicated. The Windows subsystem (`CSRSS.EXE`) is used to run 64-bit applications. For 32-bit applications, we have a special subsystem *WoW64* (Windows-on-Windows) inside the Windows subsystem, which serves as an interface for 32-bit applications (32-bit code is translated by this subsystem to 64-bit code, which can already be run via `CSRSS.EXE`).

 The *window and graphics management* modules run in Windows NT series from version 4 onwards in kernel mode to speed up applications that make heavy use of graphics devices. This part of the kernel is loaded from the `win32k.sys` file.

The placement of GUI code in kernel mode is unusual. However, the disadvantage of this approach is a higher security risk and the risk of system stability violation if this module fails (it works in kernel mode, so it has access to the memory of system processes). Another disadvantage is the more difficult procedure of replacing the user interface with an alternative one. In Windows Server 2008 onwards, it is possible to install the system without the GUI (and also without other components that require a GUI in any way) – the *Server Core* installation.

 The graphics subsystem in the kernel is the GDI module, in Windows XP also its extension GDI+. The dynamic libraries `gdi32.dll`, `gdiPlus.dll` and `gdi32Fu11.dll` are only access points to these modules in user space. GDI+ in Windows XP is an enhancement of the original GDI (for example, rational numbers can be used as coordinates, not just integers, support for various 2D operations, additional file formats including JPEG and PNG, etc.).

The `win32k.sys` file is technically part of the Win32/Windows subsystem running in kernel mode (note that it has a typical kernel-mode driver extension), but it is actually used by all subsystems of the environment, including *POSIX*. It contains an implementation of low-level UI functions, calling routines in GDI device drivers. It is used by all environment subsystems primarily to facilitate the functionality of these subsystems (so that each subsystem does not have to have its own part in the

kernel), i.e. calls to each subsystem are translated towards the kernel to calls generated by the Win32 subsystem.

 **How processes run:**

- Win32/Win64 applications run in a common (system) virtual machine, each with its own memory space (under the same numeric address, each of these applications sees different physical memory locations),
- DOS and Win16 applications each have their own virtual machine, within the virtual machine their own memory space.

2.3.2 Windows Vista/7/8/10/11

Figure 2.3 is mainly valid for Windows 10, although most of it is also valid for Vista and above (but e.g. Universal Apps are not in older versions). The DWM window manager is present since Vista, but it has been reprogrammed a lot over time (the widest changes are in Windows 7 and then Windows 10).

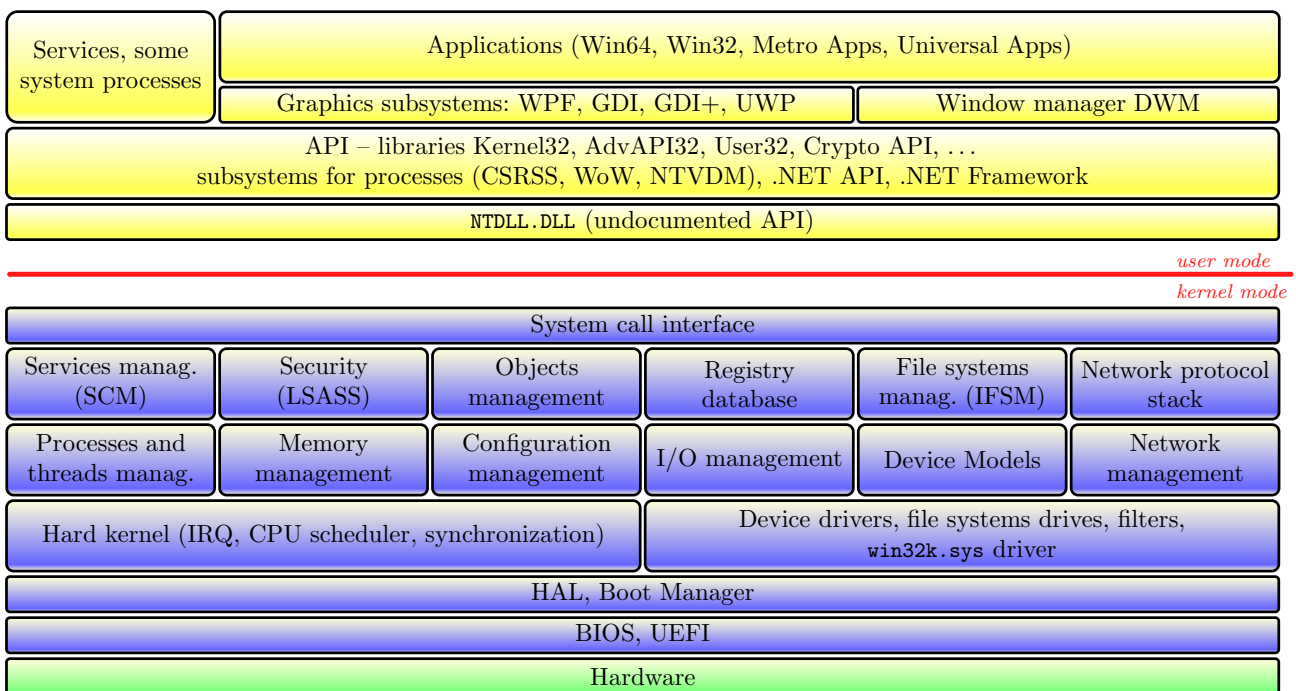



Figure 2.3: Structure of newer Windows NT

 **Vista.** The core of Windows Vista has been completely redesigned compared to its predecessors, although the basic principles remain. It has a modular internal structure.


An important change is the implementation of IPv6 in Vista. In addition, virtually the entire network stack (network support) has been moved to kernel mode, while part of the GUI implementation has been moved from the kernel to user space.

The installation DVD is the same for all Windows Vista editions for a given architecture (32-bit or 64-bit), so the *full-size* DVD contains all modules (specifically WIM files with module images) for any edition. During installation, the modules that are installed are determined primarily by the type of license (for example, Vista Home Premium has different modules availability than Vista Ultimate).


There are also separate modules where only the language settings are stored, the other modules are language independent. The consequence is that patch packages can also be *language independent* (i.e. in non-English speaking countries there is no need to wait for a patch package to be published for a given Windows language variant).

Since Vista SP1, UEFI support has been added and the system boots in a slightly different way. This does not mean that these systems cannot be installed on systems without UEFI (with the old BIOS), the installation and boot processes can handle both.

There are a lot of changes in the user space. Some of it is the same again (we also have subsystems for running processes like CSRSS, WoW etc. and documented and undocumented interfaces), but the graphical environment is no longer based only on the classic WinAPI (GDI module), but on WPF.

 *Windows Presentation Foundation* (WPF, also Avalon) is part of the .NET Framework. It is a graphical subsystem, i.e. it mainly registers windows and other graphical components inserted into windows in a tree structure taking into account their nesting, and provides management of windows (and other graphical components). The desktop is also treated as a window, as are the various panels, including the main desktop panel. Microsoft seems to have found inspiration here in the X Window System from the UNIX world.

Older applications have their GUI programmed in GDI or the newer GDI+, newer applications already use WPF. Since Vista, the system GUI is also programmed using WPF.


 However, the WPF module uses the services of the GDI library/module, so if you look at the list of dynamic libraries loaded by any GUI application, you will find `gdi32.dll` and possibly other similar files.





Additional information


<https://www.leadtools.com/help/leadtools/v19m/dh/to/differencesbetweengdiandwpcf.html>



 *Desktop Window Manager* (DWM) is a composite window manager (a reminder of the terminology in X Window) that renders windows whose structure is managed by WPF. While WPF takes care of the data, DWM renders, provides a sort of primitive 3D view (Flip3D, transparency, etc.), previews, animations, responds when the monitor resolution changes, etc.


 In Windows from Vista onwards, the *ASLR* (Address Space Load Randomization) function is implemented – libraries are not always stored in the same memory location as in XP when loaded into memory (after being requested by a process), but randomly to an address from the list. This feature is supposed to be a defense against memory overflow exploits (a hacker cannot guess which address to place the code to so that it will be in the “appropriate” place after a memory overflow). However, this protection has been broken long time ago, it is just time consuming to bypass it.

 **Windows 7.** In the case of Windows 7, not many changes have been made to the kernel in terms of functionality; most of the changes are in the internal structure of the kernel, in the management and operation of the GUI, and in the way the system is used and configured.

 For the kernel, the *MinWin* concept was used – the smallest possible base kernel (almost a microkernel), the other parts of the “broader kernel” (i.e. what runs in privileged mode) are modules, so again another step towards kernel modularity. MinWin primarily includes the main kernel. MinWin is more self-contained than the original part of the kernel, resulting in a faster boot time and better

overall responsiveness (once MinWin is loaded, multiple CPU cores can be used, so other parts of the system can be loaded in parallel).

We can also see some changes in the use of APIs, the use of virtual DLLs. There are significantly fewer services running in real life than in Windows Vista (which also makes the system run more quickly), and the settings are generally adapted to the new types of hardware (for example, Windows 7 can detect SSDs during installation and adjust some settings to accommodate this, such as disabling the defragmentation service and adjusting SuperFetch). Services running while the system is running can be temporarily stopped if SCM decides they are not needed.

 **Windows 8, 10.** Some parts of the kernel have been redesigned, the communication structure has become more complex (mainly due to DRM support for multimedia), but none of this is directly visible in our picture.

In Windows 8, *Metro Apps* appeared, and in Windows 10, *Universal Apps*. They also need their own graphical subsystem (not related to .NET, so no WPF) – for Metro Apps it was the WinRT API, for their successor Universal Apps it is *Universal Windows Platform* (UWP). In both cases it's basically the same thing (just different names) – Metro Apps are for different hardware platforms (desktop and mobile RT), Universal Apps are also for different hardware platforms (desktop, mobile, Xbox, etc).

The *Windows Subsystem for Linux* (WSL) has even moved into the Windows 10 kernel to allow running applications programmed for Linux. In fact, it is virtualization partially pushed to the kernel.

The main changes are again in the user space and graphical interface, including the availability of various tools (users have noticed especially the new *Immersive Control Panel* tool, changes in the management of updates, new applications or, on the contrary, the removal of some applications or replacement with universal ones, a new policy in the collection and management of personal information, more pressure on the use of the cloud, including authentication, etc.).

2.4 UNIX and UNIX-like Systems

2.4.1 UNIX Standards


UNIX systems must meet certain *standards* to be considered UNIX or UNIX-like systems. There have been several standards throughout history, but two are currently maintained – POSIX and the Single UNIX Specification (SUS).

Remark

UNIX is not just a specification (given by the SUS standard), it is also a trademark (i.e. the name UNIX is protected as such). The owner of the trademark is the Open Group Consortium, and only those systems that have been certified as UNIX by this consortium can be called UNIX. Operating systems that meet the SUS but have not been certified are referred to as *UNIX-like*.


For example, Solaris went through the certification process, whereas Linux is only the UNIX-like system.

Let's take a closer look at the POSIX and SUS standards.

 **POSIX** (Portable Operating System Interface) is a standard published by the IEEE standards body, in versions POSIX-1 and POSIX-2. It standardizes the form of system calls (i.e., the ways in

which a process communicates with the kernel), functions in system libraries, and the behavior of processes, including how they communicate with each other. The purpose is to simplify as much as possible the transfer of programs between different operating systems (called porting).


So the POSIX standard does not specify how processes (or the system) should look and function “inside”, it only specifies how communication between a process and the operating system kernel or between processes should look like. It therefore specifies the communication interface.

 **Single UNIX Specification** (SUS) – meeting this standard is a prerequisite for a system to be called UNIX. It is based on POSIX and includes certain specifications related to

- the programming interface of programs and libraries (C or similar language is assumed),
- a list of tools (meaning mostly programs–commands for users and administrators),
- requirements for the shell (the program by which the user interacts with the system),
- system calls, services including input/output.

No specific code requirements are included, so even systems that do not contain even a line of code from the original UNIX can comply with SUS.


SUS comes in several versions. The latest (version 4) from 2008 (with several newer editions/subversions, the newest from 2018) is also called UNIX V7. IBM AIX is certified under UNIX V7, Solaris 11 was in the past. HP-UX and Apple MacOS were certified under previous version of SUS.

 **Additional information**


- <https://pubs.opengroup.org/onlinepubs/9699919799/>
- <https://unix.org/version3/>
- <http://www.opengroup.org/certification/idx/unix.html>



2.4.2 Architecture of UNIX Systems

 The concept of HAL is mentioned above, this role is provided by the `hald` daemon. Some UNIX and UNIX-like systems always use it, but some of them left it, including Linux and FreeBSD. The role of HAL was taken by another module, in Linux by the UDEV mechanism represented by the `udev` daemon.

This change was also reflected in the fact that the hardware information was previously obtained by the `lshal` command, and now we use the `udevadm` command in Linux.

 Since “everything is a file” in UNIX systems, there are not only file systems for external storage media, but also other, abstract ones, providing access to information about the current state of the system, configuration, etc. (e.g. in Linux the *proc* file system) or combining other file systems or representing a part of another file system. File systems that do not belong to any particular storage medium, but are nevertheless treated as such (we work with files or what look like files through them), are called *virtual file systems*.

The main module for file systems is *VFS* (Virtual File System), and all other file systems (real and virtual too) are accessed over VFS. Its role is similar to IFSM in Windows, but VFS is more universal and scalable (we can add any new file system, with no problems).

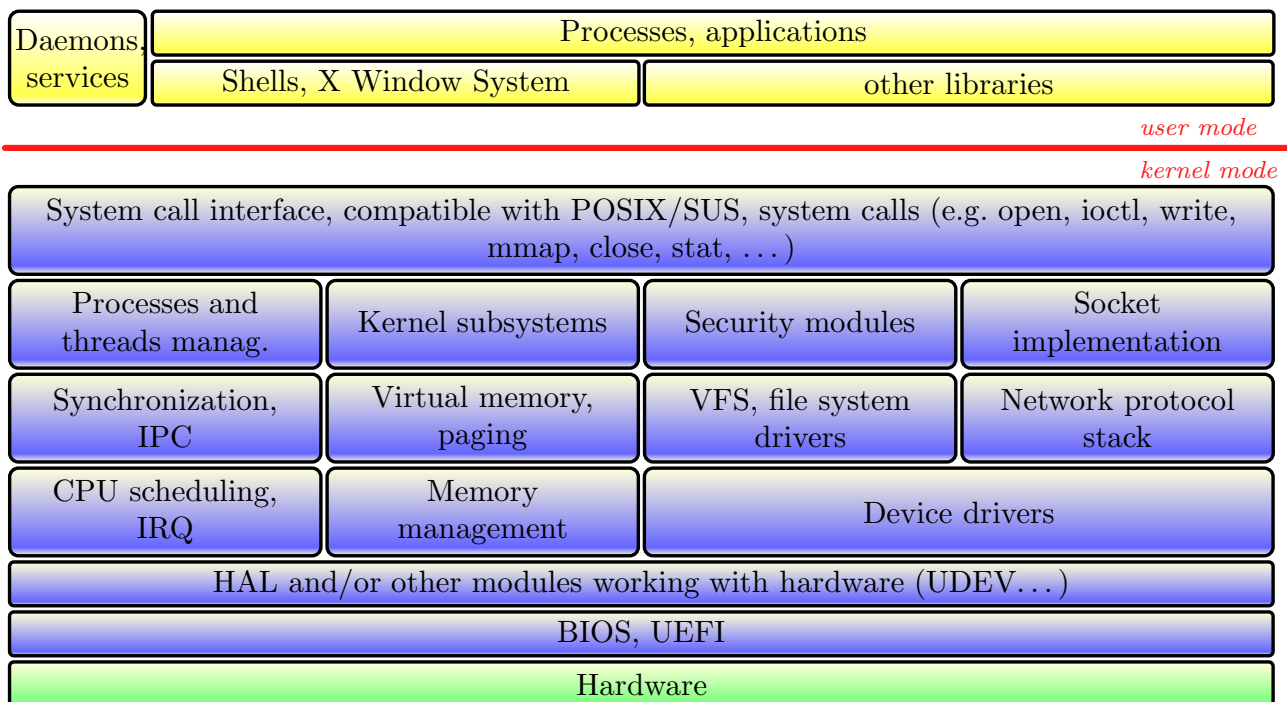




Figure 2.4: Structure of UNIX and UNIX-like systems

*FUSE*² (FileSystem in User Space) is a mechanism that allows filesystems to run in user space (regular filesystems must be part of the kernel). It consists of splitting the filesystem into two parts – the lower part, the FUSE module, is common to all filesystems of this type and runs in kernel mode. The upper part runs in user mode and uses the services of the FUSE module. Many filesystems are currently implemented in this way (for example, `ntfs-3g` and `ntfsmount` for NTFS operation, filesystems for data compression and encryption, multimedia interfaces, system tracking, version control, etc.).


 The *kernel subsystems* are e.g. encryption subsystem, multimedia subsystem and others.

Security Modules are actually subsystems as well. Depending on the specific system, which security modules are loaded, usually a firewall (for example, in Linux we use `NetFilter`), a module to increase system security (in Linux often `SELinux`), `AppArmor` and others.

The important kernel modules are *drivers*. There are block and character device drivers, including network and virtual devices, and other specialized drivers. File systems are also implemented as drivers.

 The *System Calls Interface* is the interface between the kernel and anything that can be directly influenced by the user (programs, shell commands, scripts). This layer can be interacted via libraries containing API function definitions. The main task is to ensure security, preventing user intervention in the kernel. *System Calls* are actually functions that can be used to communicate with the kernel.

There are a large number of libraries on the system, the most important of which is the `glibc` library (GNU C Library) on Linux, and `libc` on other UNIX and UNIX-like systems. This library mediates most of the communication between user-space processes and the system calls interface.

 The *shell* is any user interface. User interfaces are either for text communication (we type commands and read text information), or graphical. The widely used text shell in Linux is `bash`.

The base of graphics is provided by X Window System (or X Window, or simply X). There are

²<http://fuse.sourceforge.net/>

several implementations, the most used is X.org. Then we need a widget library (a library determining visible graphical objects such as buttons, check-boxes, menus, etc. – widgets) and a window manager (a program determining how to arrange widgets and how they shell behave, using X).

2.4.3 Linux Kernel

The image 2.5 shows a more detailed picture of the Linux kernel. Note that the HAL layer is missing, we won't really find it in newer versions of Linux. Its functions have been taken over by other kernel modules, most notably UDEV.

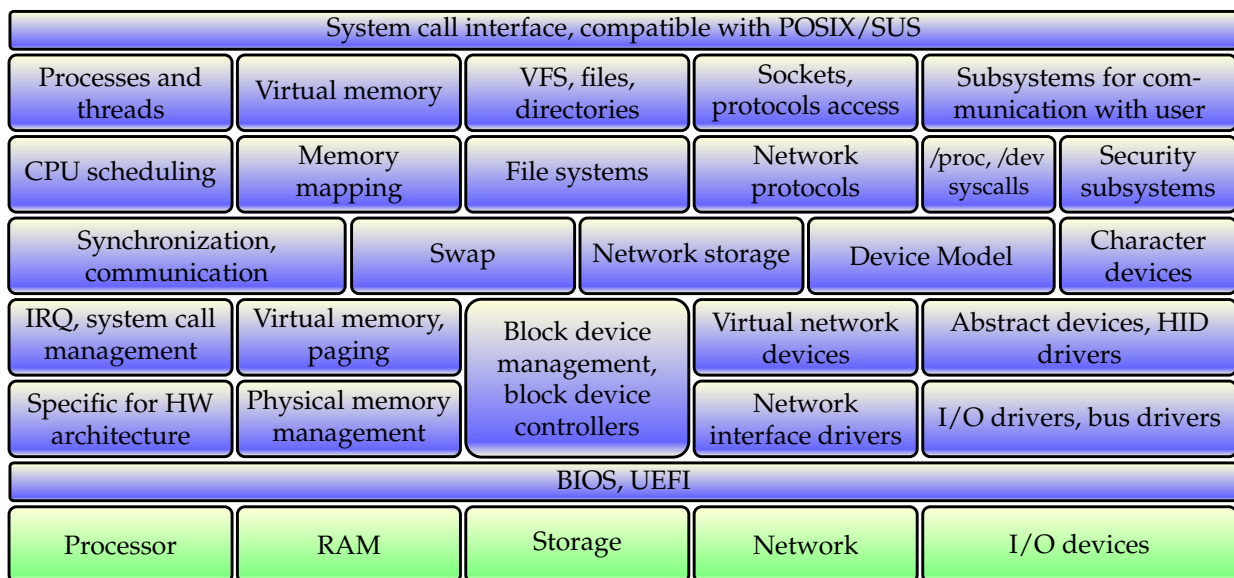


Figure 2.5: Linux kernel in newer versions

This image is purposely created so that the columns contain components that are related to each other in meaning, including the link to a specific piece of hardware. Some components are related to two hardware components, for example, swap (swap area) is related to both RAM (because pages are swapped from it) and storage media (because pages are swapped to it). Network storage is related to the network (because it is accessed over the network) and also to storage media (because it shares a way of dealing with them). Device models relate to both network interfaces and other I/O devices because they describe their structure.


Additional information


The information about the Linux kernel is in <https://www.kernel.org/>, including the current supported versions.




Chapter 3

Memory Management


 **Quick preview:** In this chapter, we will focus on memory management. First, we will discuss basic memory management concepts, practices, technologies, and strategies, then we will look at memory management in Windows and Linux.

 **Keywords:** Memory management, memory strategy, physical address, logical address, virtual memory, paging, segmentation, page replacement, NUMA, copy-on-write, memory mapping.

 **Objectives:** The goal of this chapter is to introduce the concepts of memory management in common operating systems.


3.1 Memory Management Basics

3.1.1 Memory Strategies

 When using multiprogramming, some *memory strategy* is necessary. There are three basic types of memory strategies:


- fetch strategies – when to move a piece of code or data from storage to memory,
- placement strategies – where in memory to move code or data of incoming program or data (new process) to,
- replacement strategies – if memory is full and free place is requested, what resides in memory and what is removed from memory into swap.

Current operating systems use combination of variations of these strategies.

 **Fetch strategies.** The current fetch strategy for memory allocation is the *anticipatory fetch strategy* (a new process has its code and data in memory immediately after its creation). Older systems have used the *demand fetch strategy* (a piece of code or data is placed into memory when a process references it). This proved inefficient because the process was constantly delayed by system calls during its run.


But the current operating systems for computers and laptops (Windows, all UNIX and UNIX-like systems including Linux and MacOS) use the *copy-on-write method*: if multiple processes use a specific dynamic link library or a shared memory section for read-only access, this object is in physical memory

only once, and all processes under “their own” logical address see the same physical address. When a process needs write access to an object, a private copy of that object is created in the physical memory only for that process, so copying of a library or shared memory section is performed after write access demand. So, modern operating systems also use a piece of the demand fetch strategy.


 **Placement strategies.** When creating a new process, it is necessary to find free space in memory for its address space. These strategies determine how to find this space – whether to search from the beginning and settle for the first place where the demand fits (first fit), or to go through the entire memory space and choose the place to fit with the lowest unused residue (best fit), or to select the largest free space (worst fit). The goal is:

- to keep the system high throughput (do not delay operation) – the first fit method,
- allocate memory as optimally as possible to avoid leaving many free fragments too small to be allocated to a process – the best fit method,
- not to task the system with a lot of overhead – the worst fit method.

Current operating systems avoid this problem by dividing memory space into many small parts (pages) and allocating as many pages as a process needs, using multilevel memory pages management, and by widening memory space by storage space (using virtual memory and swap space).


 **Replacement strategies.** When using virtual memory and cache, there are situations in which memory or cache must be released for a new object (memory page, cache entry, and so). Operating systems use algorithms that select the appropriate object (memory page, cache entry) as a victim to create this free space.

3.1.2 Allocated Memory Space

 When a new process is created, a memory space is searched to allocate for the given process. This space can be

- *contiguous* – the searched memory space is without “holes”, but if there is no sufficiently long free block in memory, the process cannot be started,
- *uncontiguous* – it is possible to divide the searched space into several blocks where
 - each block is intended for some purpose, various blocks are of various lengths depending on their purposes, we call these blocks by *segments* (code segment, static data segment, process stack, ...),
 - all blocks are of the same length, a process obtains as many blocks as it needs (up to the free space amount).

The second possibility allows to simplify addresses used by processes: the address of a block (of the first byte in a block) is registered by operating system or processor, and a process can use a short relative address (offset) valid inside such block.

 So, the *absolute address* of an object is the number of Bytes from the beginning of the memory address space to the first Byte of the given object. The mentioned blocks (their beginning) are always addressed absolutely, and they are usually placed in the address space at such addresses that are easy to work with (for example, less significant Bytes in the address are 0).

The *relative address* (offset) of an object stored in a block is the number of Bytes from the beginning of the block to the first Byte of the given object.


3.1.3 Garbage Collection

The above described techniques are used in one modern method of memory management for applications with own memory structure.

Garbage Collection is a form of automatic memory management used in runtime environment of multiple programming languages. The most known garbage collectors can be found inside runtime environments for Java, .NET, C++, Smalltalk, Haskell, Ruby...

The main tasks of the garbage collector are to look for sections in memory (objects) that are no longer used, with no reference; subsequently, these sections (objects) are freed and then their addresses are moved in memory to each other as needed to allocate a larger memory space later (for larger objects or sets of objects).

The last stated task uses the methods similar to those listed above.

 Let's look at the options for finding unused sections in memory. There are multiple possibilities to find unused sections/objects, for example:


- *Reference Counting*: each object has a counter with number of references leading to its object; if this counter value is zero, this object can be deleted.
- *Mark-and-Sweep algorithm*: the memory manager first sets the value "visited" to false for all objects. Then it goes through the memory, looks for references, and sets a value of "visited" to true for the target object of each reference. Then it will go through all the objects again, and remove all that have the "visited" set to false.

3.2 Complex Memory Separation Models

3.2.1 Virtual Memory and Addresses

In current computers, there is usually not enough physical memory to run multiple processes. Therefore, physical storage is increased by storage space, either as a swap partition or one or more swap files. In both cases we talk about swapping.

Some parts of memory can be swapped, some parts can not. For example, memory used by kernel usually cannot be swapped, or read-only files, locked memory sections or actively used pages.


 *Address space* is defined as metric on the memory space. All Bytes in memory space are "numbered" by addresses. In terms of virtual memory usage, we distinguish these types of addresses:


- *physical address* = address in physical memory,
- *logical address* = address in the continuous address space, this address is translated (mapped) to the corresponding physical address when accessing memory,
- *virtual address* = address in the virtual memory space, this address is translated (mapped) to the corresponding physical address as well, but two virtual addresses can lead to the same logical address, or some virtual pages are not mapped to any logical address space.

Some hardware architectures do not distinguish between logical and virtual addresses, others distinguish between them. It depends on the particular mapping algorithm, but also, for example, on the extent to which the segmentation method is implemented.


Usually we also distinguish whether it is a kernel or process address space, for example, there is a virtual kernel address and a virtual process address.

The virtual address space is divided to blocks called pages or segments, or it is possible to use the both possibilities. Pages are small blocks with the same size, segments are blocks of various sizes, but intended for a particular purpose (code segment, data segment, etc.).

 Address translation between logical (or virtual) address and physical address is provided with the hardware support – using MMU (Memory Management Unit) and several processor registers, depending on processor architecture. The process of translation is usually dynamic – we call it *Dynamic Address Translation* (DAO). It means that this translation is performed during execution the memory request.

 If the virtual address space is larger than the physical address space, we must use *virtual memory management methods*, that is, methods of expanding physical memory by swap space.

3.2.2 Paging


 Virtual address space is divided to *pages*, all pages are of the same size, this size depends on the hardware architecture. Physical address space and swap are divided to *frames* (page frames) with the same size as pages. Each page is either in a frame inside physical address space, or in a frame inside swap. The size of the pages and frames must be the same, a page must fit exactly into a frame.

Example

Various hardware architectures allow various page (and frame) size:

- x86 architecture (32-bit) uses page size from 4 KiB till 4 MiB, or till 2 MiB when using PAE (Physical Address Extension, extending physical memory above 4 GiB, page table entries are more complicated and it is necessary to transfer addresses in two cycles),
- x86-64 architecture for Intel and AMD processors uses page size from 4 KiB till 2 MiB or 1 GiB (if the processor has the PDPE1GB flag present and set),
- ARM architectures usually allow page size between 4 KiB and 1 MiB or 16 MiB (various for different manufacturers).


The chosen size also depends on the operating system and physical memory size. 

 **Translation.** Since we have a constant number of pages (memory is already partitioned when the operating system is started) and the pages are all equally long, the page count can be in a simple table with entries containing the page information. So, a memory manager uses hardware-supported dynamic address translation and a *page map table* (or simply page table). The page table is placed in the kernel memory space, and its address can be found in one of CPU registers (the CR3 register at x86 and x86-64).

A process can access only its virtual address space, and this address space appears to be continuous. Its length is

$$\text{address_space_length} = \text{number_of_pages_of_process} \times \text{page_size}$$


and the addresses from this interval can be used by this process: $0 \dots (\text{address_space_length} - 1)$.

 So, the virtual (logical) address space seems to be contiguous (the process pages look as if there is no gap/hole between them), but the corresponding frames usually are not contiguous.


Whenever accessing memory, the memory manager performs this *address translation*:

- $\text{offset} = \text{logical_address} \bmod \text{page_size}$
- $\text{page_index} = \text{logical_address} \text{ div } \text{page_size}$
- $\text{physical_address} = \left(\text{do_map_page}(\text{page_index}) \times \text{frame_size} \right) + \text{offset}$

Page mapping function is performed according to the list of pages assigned to the process, and the purpose of mapping is to find a physical frame belonging to the page.


 The first two formulas can be simplified, if the logical address can be divided into two parts – the first part specifies the page number ($\text{VPN} = \text{Virtual page number}$), the second part determines the offset. This simplification is used by all common hardware architectures. An offset, that is, a shift to a given address within a page or frame, is the same on the page and frame as well. So, a virtual address in the virtual address space can be of this form:

Virtual page number (VPN)	offset
---------------------------	--------

 **Page tables.** So, by the virtual page number we reached a *page table entry* (PTE). The page table entry contains information necessary to manipulate with the corresponding page:

- the frame number (identification of the appropriate frame in physical memory or swap),
- “Valid” bit – valid page is page in physical memory,
- “Accessed” bit – this bit is set if the page has been accessed during the last clock cycle,
- “Dirty” bit – this bit is set if the page has been changed after transfer from swap,
- R/W access protection bit, User/Supervisor mode bit, and other memory protection bits.

These items are hardware dependent, and all operating systems running at the same hardware platform use them in the same way (and with the same bit length).

 **Multilevel paging.** Most of the current architectures use multi-level paging, i.e. a VPN consists of two, three or four parts (that is, the address consists of three, four or five parts) – we talk about two-level, three-level, . . . paging.

There are several reasons for using multi-level paging:

- some parts of address space are not used (are empty), it is not necessary to map them for processes (so the advantage is to reduce page table fragmentation, some PTEs do not exist),
- for small (4KiB) pages, there are lots of pages and we need many bits to store the page number, but if the multi-level paging is used, we need less bits (because of the first item of this list),
- for long addresses (64-bit system), multi-level paging is necessary, because the number of pages is too large, even if we use larger pages,
- the last advantage is reducing TLB miss, i.e. reducing overhead.

A 32-bit system with 4KiB pages usually uses two levels of pages, a 64-bit system usually uses three or four levels.

In multi-level paging, we have one or more “container” levels – *page-directory tables*, and one level with *page tables*. Page-directory table entries always point to the tables from the following level; page table entries already point to pages/frames. The virtual address has hierarchical structure, e.g. for three-level paging:

Dir level 1 index	Dir level 2 index	Page table index	offset
-------------------	-------------------	------------------	--------

The translation process is a bit more complicated. In the case of three levels:

- the first index (the first part of the address) leads into the first level *directory* table,
- in the found directory table entry, there is a pointer to the second level directory table,
- we use the second index (the second part of the address) leading into the determined second level *directory* table,
- this entry contains a pointer to the third level *page* table,
- we use the third part of the address as the index leading into the found page table,
- and this entry finally contains the page information.

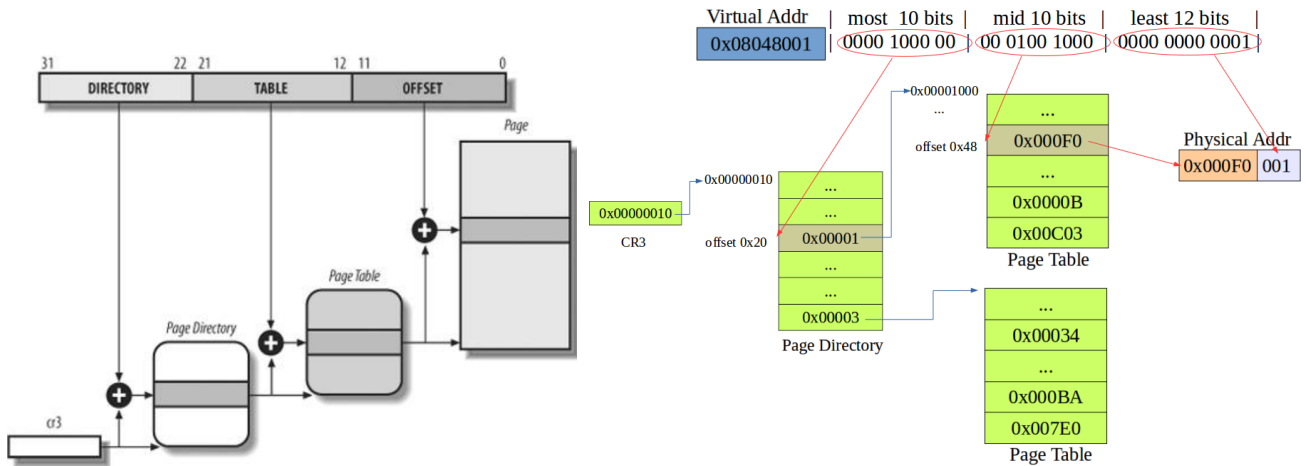


Figure 3.1: Paging at x86 with two-level page tables¹

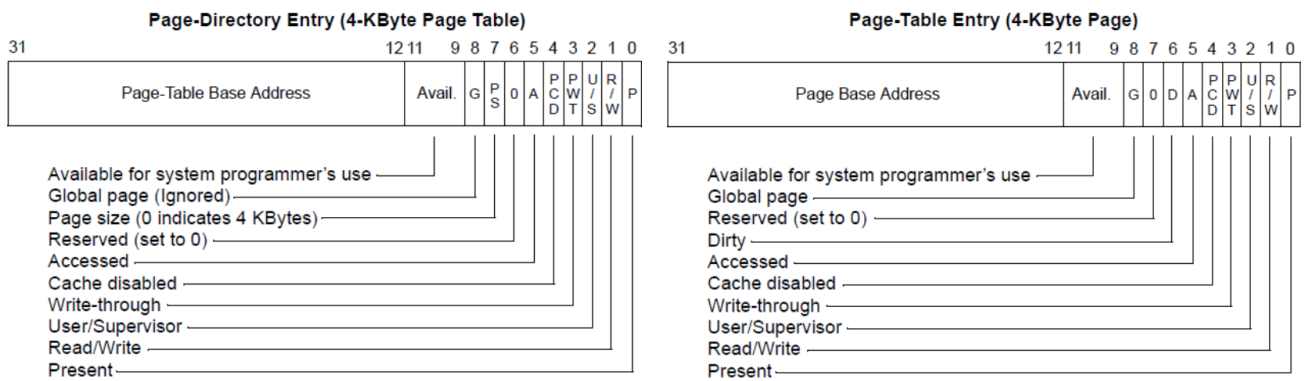


Figure 3.2: Page Directory Entry and Page Table Entry at x86 with two-level paging¹

Inverted page tables. While a standard (or multi-level) page table is indexed by page numbers, the *inverted page table* (IPT) is indexed by frame numbers. Thus, if two logical pages lead to one physical frame (i.e., a page is shared by two processes), there will be two entries in a standard page table, whereas only one entry in the inverted page table. Sharing pages is common in the current operating systems, so using inverted page tables can reduce number of page table entries very much.

Searching in IPT is more complicated: while with a standard page table, using the index (multiplied by the length of the item) moves us directly to the requested entry, here we have to search the requested table row line by line.

The inverted page table concept has been used for example at PowerPC, UltraSPARC, IA-64 (the Itanium processors), it is not used in the most current processors.

¹From: http://www.renyujie.net/articles/article_ca_x86_5.php

3.2.3 Segmentation

While pages do not have a specific purpose and all are of the same length, the *segments* usually have a specific purpose (code, data, etc.) and can be of different lengths as needed.

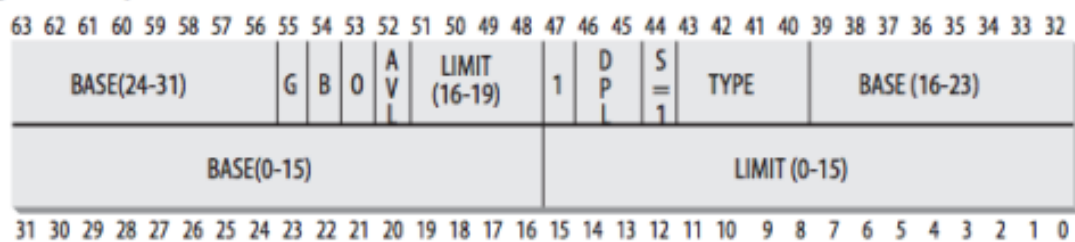
Usually, hardware segmentation support is available: set of the *segment registers*, where each register contains either the start address of the given segment, or the number from which this address can be calculated, or the identifier leading to the table with the necessary information, similarly as the previously stated page table. Translation is performed by a *segmentation unit* inside CPU, similarly as MMU for paging.

Segment address translation is similar to page address translation – there is a segment part of the address and offset part of the address. The segment part is used to get the segment base address, and then we add the offset part.

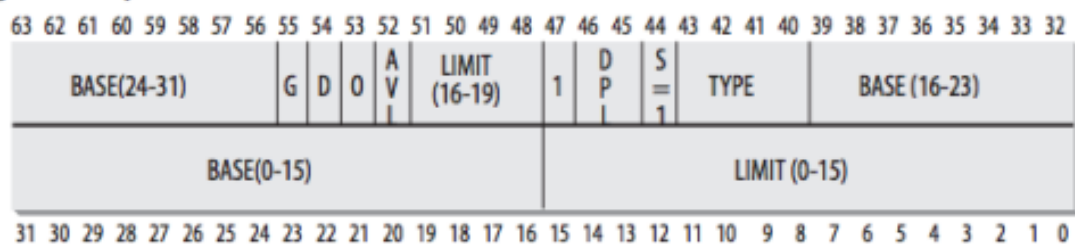
Architectures and segments. Some architectures use segments (simultaneously with other possibilities of memory organization), and some architectures do not.

We will focus on the x86 (32-bit) architecture. Each segment has a purpose, we use a code segment, a global data segment, a stack segment, and possibly other segments as required by the operating system. Each segment has its descriptor containing the segment start address, segment size, and segment properties flags (e.g. type, authorization level, whether recently used, ...).

Data Segment Descriptor



Code Segment Descriptor



System Segment Descriptor

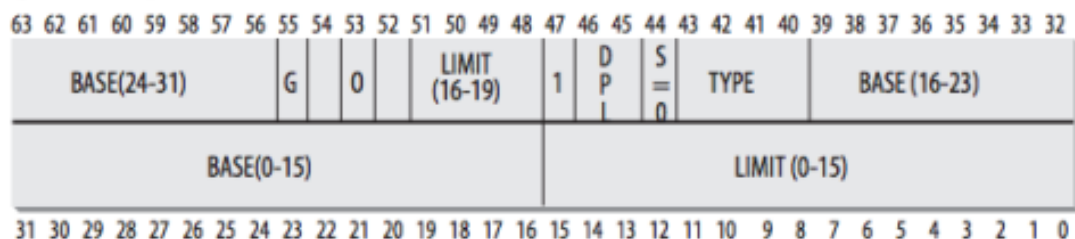



Figure 3.3: Segment descriptor format for x86²

²From: <https://notes.shichao.io/utlk/ch2/>


The segment descriptor format can be found in Figure 3.3. The individual parts:

- BA contains a base address of the target segment (separated in three fields),
- Limit – length of the segment,
- D/B – implicit length of operands (for code segments) or implicit length of word (for stack segments), 0 means 16bit, 1 means 32bit,
- S = non/system type of segment (0 means system, 1 means user),
- Type – type of segment (code, data, ...),
- P = present in descriptor table (see below), 0 means swapped segment,
- DPL – Descriptor Privilege level,
- G = Granularity, 0 means units in the Limit field in Bytes, 1 means units in the Limit field in 4KiB blocks,
- A = accessed, this field is used by page replacement algorithms.

 The operating system maintains a segment descriptor table for each process – *Local Descriptor Table* (LDT), which contains descriptors for all process segments, LDT is in the process context. In addition, *Global Descriptor Table* (GDT) exists for globally valid segments, including kernel segments, as well as descriptors leading to individual LDTs (so pointers to LDT tables). The address of the GDT is stored in the GDTR register, the address of the currently used LDT is in the LDTR register.

On this architecture, several segment registers can be used, which (if the processor uses memory protection with 4 rings) have *segment selectors* stored therein. A segment selector is like a pointer, but it contains other information (control bits) in addition to the destination address.


The commonly used segment registers are CS (Code Segment, also .text segment), DS (Data Segment, for global data), SS (Stack Segment for local data of functions), ES (Extra Segment for a programmer), and additional FS and GS (for specific purposes, depending on operating system).

 The x86-64 (64-bit) architecture does not use segments in the sense of x86 architecture. All the above mentioned segment registers are inside CPU, but the registers CS, DS, SS and ES contain 0, the registers FS and GS are mainly used for threads management (e.g. Linux uses GS to store the address of thread-local storage, TLS – each thread has its own memory space, beside the global one for a process).

But segments can exist, just without hardware support. For example, in 64-bit Linux we have four segments:

- kernel code segment,
- kernel data segment,
- user code segment,
- user data segment.

The corresponding segment registers are set to 0, so all these segments start at the same address, their differentiation is at the page translation level. The stack segment is used too, but it is software-implemented, such as a heap. The stack and heap parts of memory grow against each other (heap grows to the higher addresses, stack grows to the lower addresses).

 The ARM architecture does not use segments, but Linux running at ARM uses several software-implemented segments.

3.2.4 Paging with Segmentation

When combining paging and segmentation, each segment is laid out across multiple pages.

A segment address consists of (at least) three parts: a segment identification, a page number, and an offset. The process of address translation is depicted in Figure 3.4 (with only one-level page hierarchy).

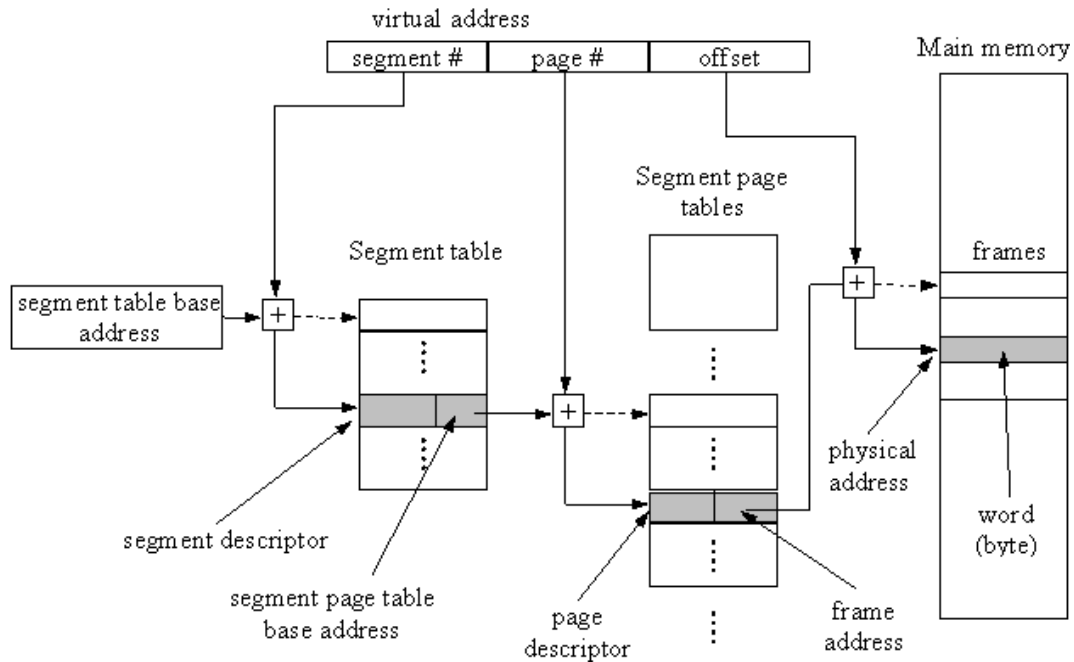


Figure 3.4: Combination of segmentation and paging³

As we can see, it is very similar to two-level page algorithm. According to Figure 3.4:

- the first part of the translated address is used as index leading to the segment table of the given process (to the segment descriptor entry),
- the segment descriptor contains the base address of the corresponding segment page table (each segment has its own page table),
- the second part of the address is used as index leading to the found page table,
- the frame number is found in the page table entry,
- now we know the frame number and the offset (the third part of the address), so we are at the finish.

The x86 architecture allows using combination of segmentation and multi-level paging, and all current operating systems for this architecture use this possibility. But the 64-bit successor x86-64 limited hardware support for segments, and using segments here no longer makes sense as for 32-bit architecture, as explained above. The ARM architecture does not use segments, nor the segment/page combination.

³From: <https://edux.pjwstk.edu.pl/mat/264/lec/main85.html>

3.3 Virtual Memory Concepts

3.3.1 Page Replacement Algorithms

Some pages are located in physical memory (valid), and some of them are swapped (not valid). If a swapped page is referenced by its owning process, it is necessary to find free frame in physical memory, load the page into, and update the corresponding page table entry.

if no free frame is available, the page fault is generated. A page replacement algorithm is used to find a suitable “victim” (a page located in memory eligible to swap) and then the found page is displaced.




Remark

If we move a page from swap to physical memory, it is not necessary to delete its content in the swap or overwrite it with another page if there is enough space in the swap. It can happen that the same page will soon be moving in the opposite direction, back to the swap – then we just release the frame in the memory and update the page table entry, transfer is not necessary.


With one exception: if the page content has been changed in the meantime, data transfer is necessary. Whether the page content has been changed is known by the “Dirty” bit (D) in the page table entry.




The main goal of page replacement algorithms is to reduce number of page faults and average page access time. There are various page replacement algorithms:

 **Optimal (OPT).** This algorithm is the most optimal, but not applicable. Such a page that will not be referenced in the near future is selected as a candidate to swap.

This algorithm shows absolutely minimal number of page faults and the best average page access times, so it is used as reference algorithm for comparison with another algorithms.

 **First-in-First-out (FIFO).** Such page is replaced that has been in physical memory the longest. The implementation of this method is quite simple, we only use a queue of valid pages. When a page fault occurs, the page with the entry at the front of the queue is swapped. The entry of the page we moved from swap into physical memory is placed at the end of the queue.

The main disadvantage is that usually the very often used pages are the most likely swapped.

 **Least Recently Used (LRU).** This algorithm is based on heuristics that those pages that have been used in the recent past are likely to be used in the near future. The victim is the page that was last used a long time ago.


There are several possible implementations of this algorithm.

- The first implementation is to use a counter in the page table entry, which is increased by 1 each time the page is accessed. If we need to find a page to swap, that one that has the lowest counter value is selected. The counter actually determines how many page-related instructions have already been executed. The problem is that these counters are quite large numbers and the method is only applicable with hardware support (because each instruction changes the entry for at least one page), it is very computationally demanding.
- The second implementation requires storing the timestamp in the table entry of the page being accessed. Victim is the page with the lowest timestamp. It is also very computationally demanding

to work with time stamps.


- The third implementation is to create a page entry queue, and the page entry that has just been used moves to the end of the queue. The victim's page is selected from the beginning of the queue. Again, computationally demanding, because each instruction intervenes in the queue.

This algorithm is quite close to optimal, but all the possible implementations are computationally demanding. The pages that have been widely used at the start of their process have a large counter, even though they are no longer used so much, while the pages of a new process have this counter near zero.

 **Least Frequently Used (LFU, Not Frequently Used, NFU).** The motivation is similar to the previous algorithm, only the frequency of page usage in a time interval is calculated, and the implementation is a bit more simple.

Each page table entry has a counter, and this counter is incremented by the value of the “Accessed” bit (A) for each clock cycle. I.e. if the page was used during this time interval, number one is added to the counter, otherwise nothing is added. The “Accessed” bit is reset to zero each time.

The implementation is similar to the first implementation of the previous algorithm, but it is less precise (not all access operations are allowed for). it is a bit less computationally demanding. Other disadvantages remain.


 **Not Used Recently (NUR).** A page that has not been used recently is the candidate to swap. This algorithm uses two bits in page table entry:

- “Accessed” bit (so called referenced bit) – set if the page has been accessed during the clock cycle,
- “Modified” bit – set if the page has been modified during the clock cycle.

At the beginning of the clock cycle, both bits of all pages are zeroed.

When page fault occurs, first a page with unset “Accessed” bit is searched. If no page with this bit unset is found, we search a page which is accessed but not modified.

This algorithm is quite close to optimal, and it is quite computationally optimal too. But it can be implemented only at architectures using the both mentioned bits. For example, Intel and AMD processors do not support “Modified” bit (they support “Dirty” bit, but it is not the same, “Dirty” bit is zeroed only in case that the page is transmitted into swap).

 **Clock algorithm.** This algorithm is hybrid between FIFO and LRU. The “Referenced” (“Accessed”) bit for each page is used (set each time the page is accessed), it is very similar to the NUR algorithm, but does not require the “Modified” bit. There are several implementations of this algorithm, it is very popular e.g. in UNIX systems including Linux.

3.3.2 NUMA

Current operating systems use the SMP (Symmetric MultiProcessing) principle on a multiprocessor system, where generally all processes share a memory bus, reducing system throughput.

NUMA (Non-Uniform Memory Access) fixes this issue. The system is divided into *nodes* (or zones), each node contains one or more processors, and each node has its own physical memory allocated. Instead of one common memory bus, memory buses exist in every node, each processor preferably uses physical memory inside its node and the memory bus inside the node. In addition, there is a superior memory bus (interconnection bus) for communication between nodes, if necessary.

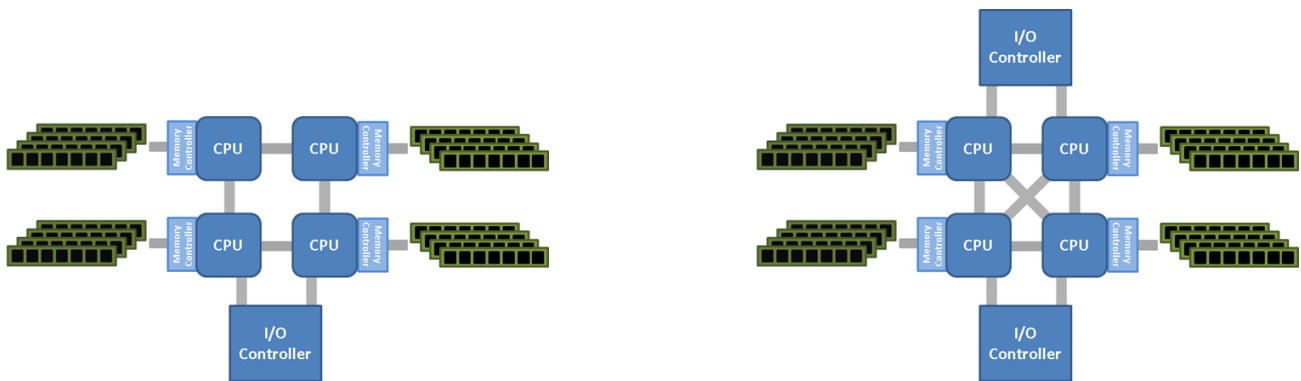


Figure 3.5: NUMA architecture for AMD processors (HT bus) and Intel processors (QPI bus)⁴

3.4 Memory Management in Windows

Most of the process memory is *paged* (can be swapped), but processes (and especially the kernel) also have *non-paged memory* used by time-critical functions (for example, IRQ or DPC calls).

Memory virtualization. Windows uses a virtual method segmentation with paging on demand. When it is necessary to release frames in the physical memory, a variant of Clock algorithm is used on the single-processor systems for the “victim” selection, on multiprocessor systems variant of the FIFO algorithm.

The paging file is usually called `pagefile.sys` and is on the system disk, but in fact we can have multiple paging files (in which case everyone should be on a different hard drive). Names of paging files are in the Registry database, the key `HKLM/SYSTEM/CurrentControlSet/Control/Session Manager/Memory Management`, value `PagingFiles`.

If we have multiple paging files, everyone should be on a different disk (if there are more on the same physical disk, even on other partitions, it can even slow down the system).

Placing a paging file into a separate partition on the disk (separate from the system and data partitions) may be useful, because there is no risk of paging file fragmentation (but on the other hand, there is a top memory area boundary that can be used for paging).

Segments and pages. Segments are used in Windows as described on page 26, so there is one Global Descriptor Table (GDT) in the kernel space, and each process has its Local Descriptor Table (LDT), the entries of LDT (and GDT) hold information about segments (e.g. the segment base address, the control bits, etc.).

In the user space, segments are addressed by *selectors*. A selector is a structure containing

- the index of the entry in LDT or GDT for the given segment,
- one bit determining if the selector leads to GDT (0) or LDT (1),
- two bits RPL (Requested Privilege Level) – 00 for Ring0 (kernel mode) or 11 for Ring3 (user mode).

So, the address of anything in the user address space (variable, instruction, ...) consists of two parts:

Selector	Offset
----------	--------

⁴From: <https://www.sqlskills.com/blogs/jonathan/understanding-non-uniform-memory-accessarchitectures-numa/>

A *virtual address* is actually an address pointing inside a segment. This address is translated to a *linear address* of the length 32 or 64 bits (depending on the architecture – x86 or x86-64) consisting of two, three or four parts (based on paging levels used), and this address is translated by MMU to a *physical address*.

So, the virtual address does not contain page indexes, they are discovered by the selector translation.

At x86, there are these types of segments in Windows:

- the code segment of a process,
- the data segment of a process, the stack segment is implemented inside the data segment,
- the system segments for code or data.


The format of their descriptors is shown in Figure 3.3 (page 26).

At x86-64 architecture, the segments are used too, but their base address is 0.

At x86, two-level page tables (for 4KiB pages) or one-level page tables (for 4MiB pages) are used. At x86-64, four-level page tables are used, and it is possible to use 4KiB and 2MiB pages.

Two page lengths can be combined. In this case, entries of longer pages are placed one level up in a multilevel table structure (so, it is possible only if at least two levels of page tables are used).

3.5 Memory Management in Linux


 **Address space allocation.** If it is a 32-bit system, 4 GB of virtual memory is available for the process, usually a 3+1 split is used (i.e. 3 GB for user space, 1 GB for privileged mode).


On 64-bit systems, 256 TB of memory is available and is usually split in half (i.e. 128 TB for process, 128 TB for privileged mode).

Additional information

<https://www.berthon.eu/wiki/foss:wikishelf:linux:memory>



 **Memory virtualization.** Almost all of today's UNIX systems, including Linux, use paging on demand or paging with segmentation on demand, according to architecture (this is also the case with Linux). Page tables are multi-level, number of levels vary on different hardware platforms – two levels are sufficient on a 32-bit processor (global page directory and page tables), or a third level is added (middle page directory), three or four levels are used on a 64-bit processor.

 As the replacement algorithm is used the Clock algorithm, with respect to multilevel paging. The indication of old pages (eligible to swap) is not only in one bit, but it is a number of the range from 0 till 20, where 0 stands for old page. Each time the page is accessed, this value is increased (by default by 3 to maximum value), the memory manager constantly (clock algorithm) goes through all these values and decreases them (by default by 1).

Example

We can have multiple swap files (and a swap partition). An additional swap file can be created as follows:

- we create a continuous file on a disk filled with zero symbols #0 (not the zero numbers!),
- we mark this file as swapping and we turn it on the swapping process.


For example (creation of a swap file of the length 65536, pages of the length 4 KiB):

```
dd if=/dev/zero of=/novy_swap bs=4096 count=65536
mkswap /novy_swap
swapon /novy_swap
```

To use this file after restart, it is necessary to add this row into the file `/etc/fstab`:

```
/new_swap none swap sw 0 0
```





 **Sharing and Mapping.** These systems support several interesting memory management features, for example, in addition to the well-known and commonly used *copy-on-write*, we encounter the *program code sharing* mechanism: if multiple processes – instances of the same program – are running, they can share the part of memory where the program code is loaded (the code segment).

The *file mapping* function works in such a way that any file can be mapped into the address space of a process, no matter where it is physically located (it does not have to be in RAM). There are usually one of two reasons for mapping:

- we want to be able to work with the file directly on the disk (generally a storage) in the same way as if it were loaded into the memory (more precisely, we want to work with the file as with the memory, except for speed, of course), but we don't actually want this file in the memory (for example, because of the large size of the file), it may also be the executable code of a larger program (i.e. a segment of the code would remain on the disk),
- *shared memory* implementation of any size – shared memory directly in the memory is a security risk, so it is much easier and safer to create a (simulated) shared memory section accessible to multiple processes on a storage media.

The mapping mechanism is very versatile, in fact it is also used, for example, when swapping.

 **Address Space.** Each process has assigned a *memory descriptor* (one or more), which is a structure containing all information about the virtual memory allocated to the process, as well as related functions (e.g., access to the structure to facilitate page lookups, the number of mapped areas, the address of the first mapped area, the total size of the virtual memory mapped to the process, memory-related synchronization objects, area unmapping functions, etc.). The memory descriptor is available in the *process descriptor*.⁵

 **Additional information**


- <https://stackoverflow.com/questions/31578233/how-does-arm-linux-maintain-segments>
- <http://www.makelinux.net/reference>
- <https://pdos.csail.mit.edu/6.828/2014/readings/i386/toc.htm>
- <https://notes.shichao.io/utlk/ch2/>
- https://answers.microsoft.com/en-us/windows/forum/windows_10-performance-winpc/physical-and-virtual-memory-in-windows-10/e36fb5bc-9ac8-49af-951c-e7d39b979938




⁵In the process descriptor we find mainly structures and references related to allocated resources and communication tools, such as pointers to file descriptors, pointers to memory descriptors, the current working directory of the process, the terminal on which the process is running, the structure for registering signals, etc.

Chapter 4

Processes

 *Quick preview:* In this chapter, you will learn the basics of process management. We will define a process and its properties, discuss the basic forms of multitasking, look at the issue of processor allocation and the possibilities of process synchronization.


 *Keywords:* Process, multitasking, multithreading, concurrent processes, CPU scheduling, inter-process communication.


 *Objectives:* The goal of this chapter is to understand the principles of process management in current operating systems and to learn how to work with processes.

4.1 Multiple CPU Cores

Nearly all current processors (Intel, AMD, ARM, . . .) have multiple (physical) cores. Most processors have two computing threads in each core: Intel calls this property by hyperthreading, AMD calls it by SMT (Symmetric Multiprocessing) – an operating system can see them as logical cores.


Servers commonly have multiple processors (with multiple cores, each core with two computing threads), and current operating systems are able to utilize them. So, all modern operating systems support multiprocessing. Microsoft uses a relatively complicated licensing policy for Windows, where the license fee depends on the number of processors and cores. For UNIX systems, including Linux, support for multiple processors is commonly granted.


 There are two types of multiprocessing: *ASMT* (asymmetric multiprocessing) and *SMT* (symmetrical multiprocessing). ASMT dedicates one processor to the kernel and the other processors are allocated to common processes, SMT fairly distributes processors among kernel and processes. Nowadays all operating systems use SMT.

 Systems with more processors working in SMT usually use the *NUMA* architecture (Non-Uniform Memory Access): each processor or set of processors has its own part of memory (memory modules), these parts of memory are independently addressed. The consequence is that such a computer system may have a lot more memory than a single processor could address, so this system is more scalable. NUMA is supported by Windows Server and all UNIX systems too, including Linux.

4.2 Obtaining Process Information

4.2.1 Processes in Windows

 We can use *Task manager* to obtain information about processes in Windows. The fastest way to run it is the shortcut `Ctrl+Shift+ESC` (on the left side of a keyboard, two keys in the bottom left corner, one key in the upper left corner).

 Another way is in the text mode:

- in Command Line (`cmd.exe`) we can use the `tasklist` command to list processes and basic information, the `taskkill` command can destroy processes,
- in PowerShell (`powershell`) we can use the `get-process` tasklet

And we can use the third-party software.

Example

Let us try the `tasklist` command.

`tasklist` displays a list of running processes (jobs) on the local computer (name, PID, session, allocated memory)

`tasklist /S some_computer` the same, but on a different computer in the local network (computers must “see” each other)

`tasklist /V` more detailed information (with status, user name, CPU time, window title)

`tasklist /SVC` also services running in processes are displayed

`tasklist /M` list of all processes with modules (dll libraries etc.) linked into these processes

`tasklist /M ntdll.dll` we can specify the library – in this example: what processes use undocumented interface `ntdll.dll`?

`tasklist /fi "imagename eq svchost.exe"` filtering the output – we want only the processes with the `svchost.exe` image

`tasklist /m /fi "pid eq 8482"` list of all modules linked into the process with the given PID

`tasklist /fi "username eq NT AUTHORITY\SYSTEM"` all processes running over the system account

`tasklist /fi "memusage le 98000"` all processes using at most 98 000 kB of memory (`le` means “less of equal”), we can use another operators, e.g. `ge`, `lt`, `ge`, `ne` (not equal)

`tasklist /FO list` the output will be list, not tabular

`tasklist /FO csv >> proc.csv` the output will be in the CSV format, we redirect it into the given file (CSV files can be open in text editors, and in Excel too)

`tasklist /?` displaying help for this command

The `taskkill` command is used to terminate processes:

`taskkill /pid 5810` terminates the process with the given PID, it is a proper ending just like we would click on the cross in the upper right corner of the process window

`taskkill /F /pid 5810` *kills* the process with the given PID (F as “Force”), we can use it if the process does not respond and refuses to quit normally

`taskkill /T /pid 5810` terminates the given process and all its child processes recursively

`taskkill /IM firefox.exe` we can terminate all processes created from the given image

`taskkill /fi "username eq some_user"` the same filters as for the `tasklist` command



**Example**

The PowerShell command `get-process` has the different syntax:

```
get-process    list of running processes
```

```
get-process -name firefox.exe -module    list of modules linked in the given process denoted by name
      (if there are more processes with the same name, we obtain very long list with modules in all
      these processes)
```

```
get-process -id 9372 -module    list of modules linked in the given process denoted by PID
```

```
stop-process -name fi* -confirm    stops all processes with names beginning with “fi”, asks for confir-
      mation (so we can stop only those fitting processes we really want to end)
```

```
get-process | where-object -filterscript {$_.Responding -eq $false} | stop-process    stops
      all not-responding processes
```

```
get-process -id 11264 | select-object -expandProperty threads    details on all threads running in-
      side the process with the given PID
```

```
get-help get-process    displaying help for this command
```



There are also tools for working with services (a service is a special type of process) – whether in the graphical environment (`services.msc`) or in a text environment (for example, the `sc` command, or the PowerShell tasklet `get-service`).

It is more convenient to use third-party software that additionally provides more information than a Task manager.



Process Explorer is probably the most widely used program for process analysis in Windows. This freeware is from Sysinternals, which has been part of Microsoft for many years, and it is portable application (so we do not install it, we only run it). Administrators usually have a USB flash disk with useful diagnostic programs, and this application is one of them.

**Additional information**

When entering [sysinternals.com](https://docs.microsoft.com/cs-cz/sysinternals/) into your web browser, you will be redirected to <https://docs.microsoft.com/cs-cz/sysinternals/>. If no redirection occurs, immediately go out – you probably mistyped. This address is very popular, and hackers know it: similar addresses are registered by various persons.

We have three possibilities:

- to get only one application – by choosing *Process Explorer* in *Process Utilities*,
- to get the entire suite of diagnostic applications (*Sysinternals Suite*),
- to run these apps directly from web without download (*Sysinternals Live*).

To get more information about system processes, run this application as an administrator.

**4.2.2 Processes in Linux**

There are many different graphical interfaces in Linux, each with a tool similar to Windows Task Manager. When using the GNOME environment or some similar (Mate, Cinamon, . . .), we have *System Monitor* (the name of this application can be different).



Figure 4.1: Sysinternals web page

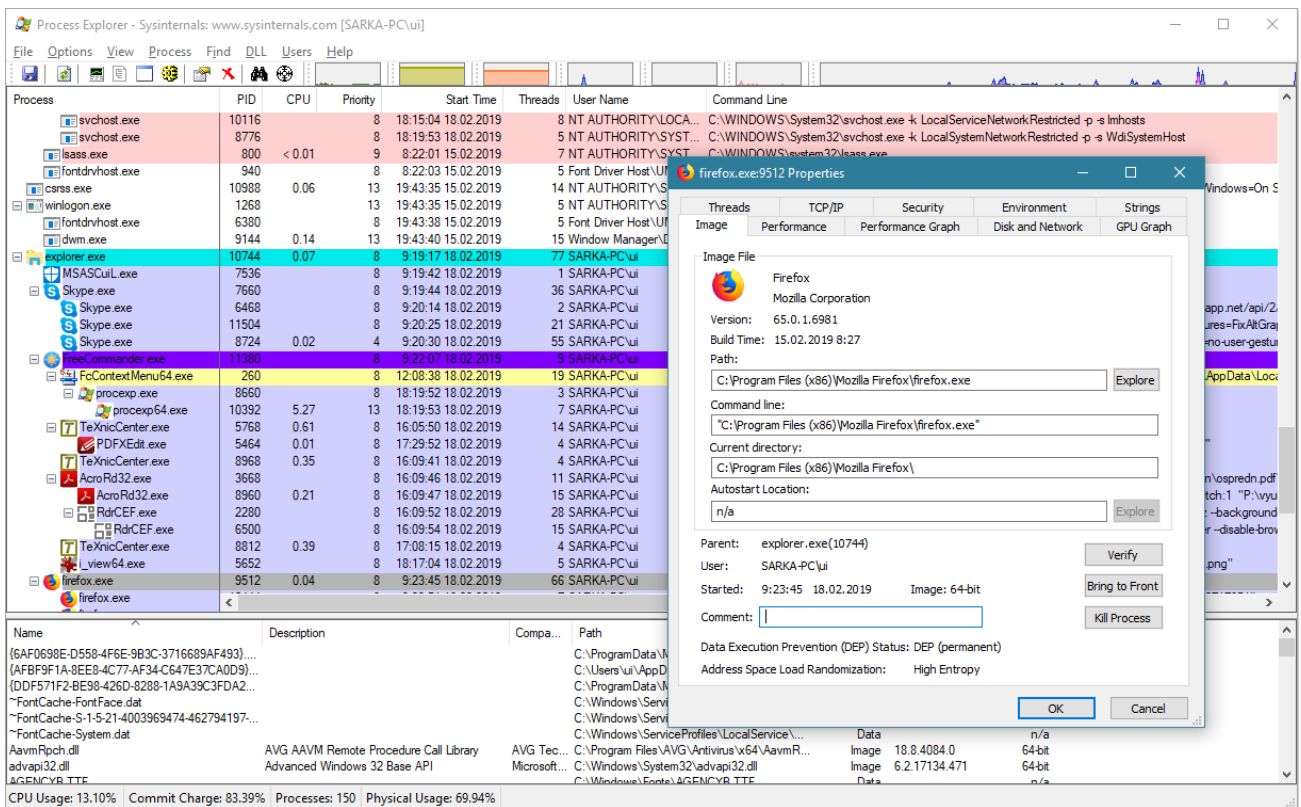


Figure 4.2: Process Explorer

In the text shell, we have several commands to work with processes. The most popular commands are the following:

- ps to list running processes,
- pstree to display a process tree,
- top is an interactive program to display run-time information about active processes,
- htop is similar to top, more information and a bit different environment.

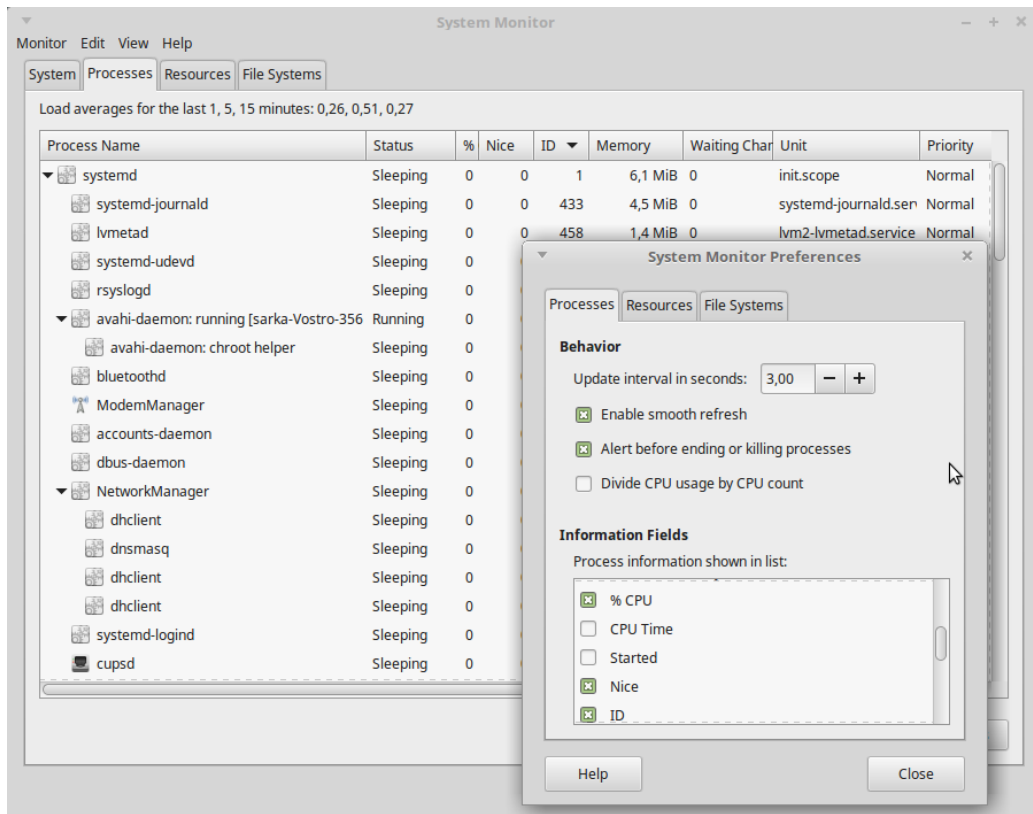


Figure 4.3: Mate System Monitor in Linux

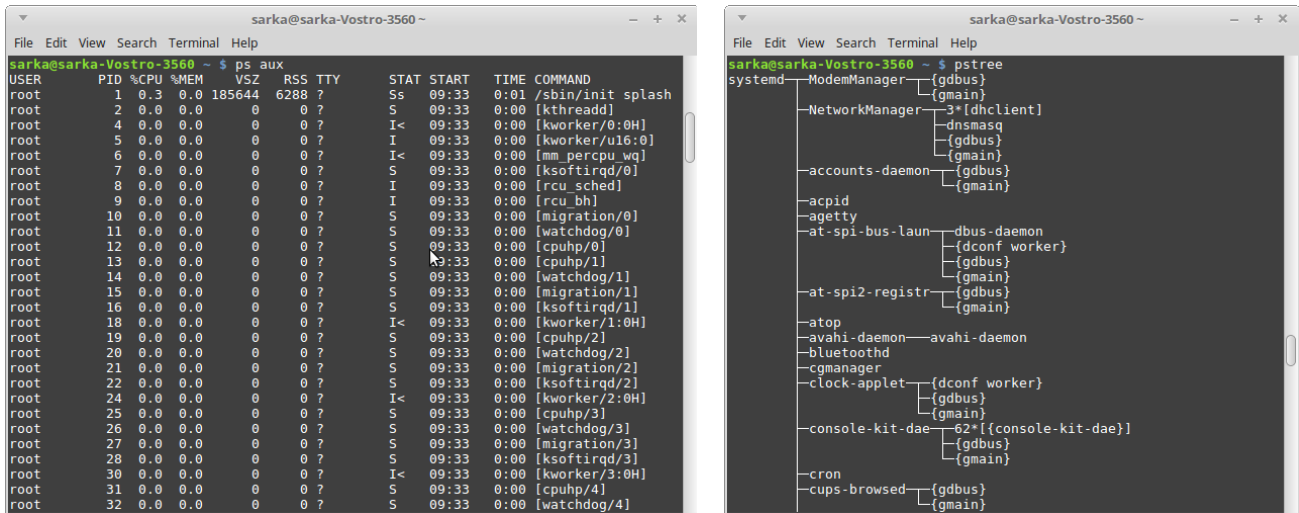



Figure 4.4: Output of “ps aux” and “pstree”

4.3 Process Concept

4.3.1 Program, Process, Thread

 A *program* is an executable file containing code, constant data, etc., stored on a storage.

Programs for a particular operating system are always in a system-specific format. This format is typical for executable files, and for dynamically linked libraries as well.

The figure shows two terminal windows. The top window displays the output of the 'top' command, showing system statistics and a list of processes. The bottom window displays the output of the 'htop' command, showing a more detailed view of processes with color-coded bars and additional statistics.

```

sarka@sarka-Vostro-3560 ~ $ top
top - 09:39:10 up 5 min, 1 user, load average: 0.17, 0.46, 0.26
Tasks: 233 total, 1 running, 159 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.7 us, 0.6 sy, 0.0 ni, 96.1 id, 2.6 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 8040476 total, 6875128 free, 432448 used, 732900 buff/cache
KiB Swap: 1951740 total, 1951740 free, 0 used, 7234808 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM    TIME+  COMMAND
 2373 sarka    20   0 518280 50760 35088 S   6.3   0.6   0:22.10 mate-system-mon
 1356 root     20   0 529736 66928 56064 S   0.7   0.8   0:08.78 Xorg
    220 root     20   0 0 0 0 I   0.3   0.0   0:00.18 kworker/u16:4
 2027 sarka    20   0 558804 24220 20180 S   0.3   0.3   0:00.52 mate-multiload-
 2030 sarka    20   0 491224 26864 22688 S   0.3   0.3   0:00.77 mate-netspeed-a
 3166 sarka    20   0 43568 4064 3336 R   0.3   0.1   0:00.06 top
    1 root     20   0 185644 6288 4020 S   0.0   0.1   0:01.49 systemd
    2 root     20   0 0 0 0 S   0.0   0.0   0:00.00 kthreadd
    4 root     0 -20 0 0 0 I   0.0   0.0   0:00.00 kworker/0:0H
    5 root     20   0 0 0 0 I   0.0   0.0   0:00.06 kworker/u16:0
    6 root     0 -20 0 0 0 I   0.0   0.0   0:00.00 mm_percpu_wq
    7 root     20   0 0 0 0 S   0.0   0.0   0:00.01 ksoftirqd/0
    8 root     20   0 0 0 0 I   0.0   0.0   0:00.20 rcu_sched
    9 root     20   0 0 0 0 I   0.0   0.0   0:00.00 rcu_bh
   10 root    rt   0 0 0 0 S   0.0   0.0   0:00.00 migration/0
   11 root    rt   0 0 0 0 S   0.0   0.0   0:00.00 watchdog/0
   12 root    20   0 0 0 0 S   0.0   0.0   0:00.00 cpuhp/0
   13 root    20   0 0 0 0 S   0.0   0.0   0:00.00 cpuhp/1
   14 root    rt   0 0 0 0 S   0.0   0.0   0:00.00 watchdog/1
   15 root    rt   0 0 0 0 S   0.0   0.0   0:00.00 migration/1
   16 root    20   0 0 0 0 S   0.0   0.0   0:00.00 ksoftirqd/1

sarka@sarka-Vostro-3560 ~ $ htop
 1 [  ] 1.3% 5 [  ] 0.0%
 2 [  ] 0.0% 6 [  ] 0.7%
 3 [  ] 0.7% 7 [  ] 0.0%
 4 [  ] 0.7% 8 [  ] 0.0%
Mem[|||||] 537M/7.67G Tasks: 100, 200 thr; 1 running
Swp[ ] 0K/1.86G Load average: 0.10 0.29 0.23
Uptime: 00:07:54

  PID USER      PRI  NI   VIRT   RES   SHR  S  CPU%  MEM%    TIME+  Command
 3379 sarka    20   0 27852 4072 3188 R   1.3   0.1   0:00.30 htop
 3140 sarka    20   0 558M 36528 28616 S   0.7   0.5   0:01.76 mate-terminal
 1065 root     20   0 440M 17136 1892 S   0.7   0.2   0:00.97 /usr/sbin/NetworkManager --no-daemon
 2098 sarka    20   0 733M 32972 28844 S   0.7   0.4   0:00.59 nm-applet
 1356 root     20   0 511M 66784 58072 S   0.0   0.8   0:12.08 /usr/lib/xorg/Xorg :0 -audit 0 -auth /var/lib/mdm/:0.X
 2030 sarka    20   0 479M 20864 22688 S   0.0   0.3   0:01.24 /usr/lib/mate-applets/mate-netspeed-applet
 1826 sarka    20   0 617M 31540 28164 S   0.0   0.4   0:00.75 mate-panel
 1038 avahi    20   0 44912 3012 1688 S   0.0   0.0   0:00.23 avahi-daemon: running [sarka-Vostro-3560.local]
 2027 sarka    20   0 545M 24220 20180 S   0.0   0.3   0:00.80 /usr/lib/mate-applets/mate-multiload-applet
 1821 sarka    20   0 482M 33384 28616 S   0.0   0.4   0:02.67 marco
    1 root     20   0 181M 6288 4020 S   0.0   0.1   0:01.51 /sbin/init splash
 1582 root     20   0 4098M 6296 6212 S   0.0   0.1   0:00.02 /usr/sbin/console-kit-daemon --no-daemon
 1894 sarka    20   0 897M 74200 40268 S   0.0   0.9   0:01.21 caja
 1395 root     20   0 511M 66784 58072 S   0.0   0.8   0:00.07 /usr/lib/xorg/Xorg :0 -audit 0 -auth /var/lib/mdm/:0.X
 1396 root     20   0 511M 66784 58072 S   0.0   0.8   0:00.07 /usr/lib/xorg/Xorg :0 -audit 0 -auth /var/lib/mdm/:0.X
 1052 mesagebu 20   0 44112 4948 1604 S   0.0   0.1   0:00.49 /usr/bin/dbus-daemon --system --address=systemd: --nof
 1899 sarka    20   0 534M 30944 28328 S   0.0   0.4   0:01.07 /usr/lib/mate-panel/wmck-applet
 1142 root     20   0 19572 268 0 S   0.0   0.0   0:00.02 /usr/sbin/irqbalance --pid=/var/run/irqbalance.pid
 433 root     20   0 35380 4560 124 S   0.0   0.1   0:00.21 /lib/systemd/systemd-journald
 458 root     20   0 94772 1476 1300 S   0.0   0.0   0:00.00 /sbin/lvmtdad -f
 461 root     20   0 46792 1652 1140 S   0.0   0.1   0:00.83 /lib/systemd/systemd-udevd
 1058 syslog  20   0 250M 1296 680 S   0.0   0.0   0:00.02 /usr/sbin/rsyslogd -n
 1059 syslog  20   0 250M 1296 680 S   0.0   0.0   0:00.00 /usr/sbin/rsyslogd -n
 1060 syslog  20   0 250M 1296 680 S   0.0   0.0   0:00.02 /usr/sbin/rsyslogd -n
F1 Help F2 Setup F3 Search F4 Filter F5 Free F6 Sort By F7 Nice F8 Nice F9 Kill F10 Quit

```

Figure 4.5: Output of “top” and “htop”

MS Windows use the following formats:

- PE – Portable Executable for 32-bit Windows,
- PE+ for 64-bit Windows,
- .NET PE for .NET applications,
- UWP format (Universal Windows Platform) for universal apps in Windows 10.

These formats are used in files .EXE, .SYS, .DLL, .TTF (True Type Font), .NLS (National Language Services – language drivers), .OCX (ActiveX),...

Linux uses the ELF format (Executable and Linkable Format). Executables in Linux usually have no extension, and libraries have the extension .so (Shared Object), often with added version number. Another interesting binary files are .ko (Kernel Object) for kernel modules, e.g. drivers.

 A *process* is an instance of a program created by running the program.


If the process originated by running from a binary executable file (program), we call this file a *process image*.

Each process is characterized by:

- its code, loaded from its image,
- execution context: contents of processor registers, including program counter,
- global data (constants and variables),
- one or two stacks for function parameters, values of local variables, etc.,
- heap for dynamically allocated variables,...

A process can be created by another process, its *parent process*. The *tree structure of processes* is based on this type of relation – structure of superior parent processes and subordinate child processes.

Processes in current operating systems have a unique identification number PID – *Process Identifier*. Most processes keep a record of their parent process – PPID – *Parent PID*.

 A process can be divided into several parts called *threads*. A thread (co called light-weight process) is relatively independent computational unit of a process. Each process has at least one thread, and other threads can be created by running functions that are programmed for these purposes.

If the concept of threads is implemented by operating system, its kernel assigns the processor to the threads, not the processes. So, more threads of one process can run parallelly on different cores of processor, or on different processors at multiprocessor system. Each thread has its own identification number: *TID* (Thread ID).

Splitting the process into multiple parts – threads – is advantageous if the process consists of more independent pieces of code (they do not affect each other).

A typical example is an application that communicates with the user, allows him to work or enjoy it (one thread), and the “background” code copies files (the second thread). Each of these threads works independently, one does not affect the activity of the other except for possible communication (the first thread can tell the user on a suitable graphic element how far the second thread is in the copy, the second thread always sends message to the first thread after copying a single file or a certain quantity of Bytes).

 **Multithreaded Programming** can be useful in the following cases:

- games,
- video codecs (e.g. the H.264 codec is able to effectively utilize up to 8 processor cores), modelling programs, making animations, multimedia applications,
- math applications, computationally demanding calculations,
- comprimation, encryption,
- applications programmed according to the Model-View-Controller concept,...

These boundary situations may occur when programming multi-threaded applications:

1. the threads are independent of each other, do not share any resources, do not communicate with each other – ideal case, they can run without problems at different processors or processor cores at the same time,
2. the threads share resources, one has to wait for the result of another thread, or they communicate with each other – need to synchronize, mutual waiting, these threads can not run at the same time.

**Remark**

We will talk about processes, but very often the topic will be more about threads. Threads have their states, are scheduled to processor,...

**4.3.2 Process States**

Each process is in some state, and it passes between different states while running. The set of possible states is determined by operating system, most systems use these process *states*:

- *new* – the process is being created, and resources are allocated to it (memory, open files/modules,...),
- *running* – the process has a processor (core) to be assigned to, it executes its code,
- *ready* – the process is waiting to be assigned to a processor, it is ready to execute its code (waiting in a processors queue),
- *waiting* (blocked) – the process is waiting for a resource or event to occur, the synchronization mechanisms block processes as well,
- *terminated* – the process has finished its execution, but the structures with its data are still in the kernel.

Each system has its own typical additional states.



In Windows, a process is in the state *suspend*, if a part of its memory (some memory pages) is moved from RAM into a swap (e.g. into `pagefile.sys`). Processes usually go to this state from the state *blocked* (waiting) in case that there is a lack of space in RAM, or the process works only sometimes, in regular intervals (so it sleeps between times of work), or the parent of the process or a user requests to suspend this process, or the system may suspend any process too.



In Linux/UNIX, nearly all processes are in the *sleeping* state – if a process has nothing to execute or has to be synchronized, it sleeps. Sleeping is interruptible or uninterruptible, depending on ability to respond to IRQs. Synchronizations mechanisms cause transition into the uninterruptible state.

Some processes can be *stopped* by user, by parent process (the both can stop process by sending the “SIGSTOP” communication signal) or by system (process is being traced). Stopped process can continue into the state *ready* by obtaining the “SIGCONT” signal, or during tracing.

A process can be *zombie*: if a parent process runs some child process with the property “I want result”, the child process is terminated but the parent process does not collect the result, the child

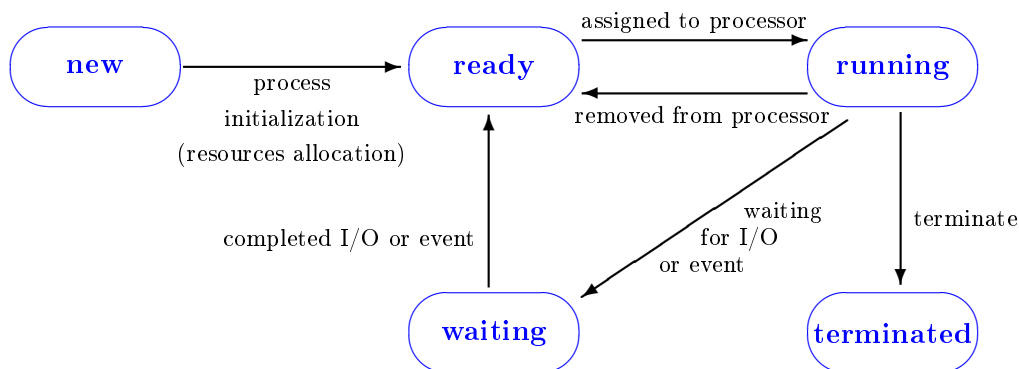



Figure 4.6: Process Life Cycle


process becomes zombie. The parent process is informed by the “SIGCHILD” signal, but if it has not this signal handled. . .

4.3.3 Process Control Block


 Each operating system keeps records of its processes in certain kernel structures. The general designation for the process information structure is *PCB* (Process Control Block). These blocks are at least as many as processes, and PID of such process can be understood as an index in the array with these structures. So, if the system needs information about a process with a particular PID, it will find it in the structure at the appropriate position in that array.

The contents of PCB is specific for operating system, some “general” entries:

- image file name (the process is created from),
- process state,
- security information – effective user whose access permissions are used by the process to access resources (another designations: security descriptor, security token),
- create time information,
- accounting information – time limits, kernel time (amount of time spent in kernel mode), user time (amount of time spent in user mode),
- CPU registers contents, including the Program Counter,
- memory management structures (what memory pages are used by the process, etc.),
- CPU-scheduling information including process priority,
- I/O status information including assigned resources and I/O priority,
- thread list, . . .

 **PCB in Windows.** The following data structures are used by Windows kernel to keep information about processes:

- *EPROCESS* – contains some properties of a process (PID, process name, . . .) and links leading to other data structures related to this process,
- *ETHREAD* – similar structure for thread, the *EPROCESS* structures contain links leading to the *ETHREAD* structures of all threads belonging to the process,
- *PEB* (Process Environment Block) – this structure is located directly in the address space of the process (not in the kernel) and is accessible from *EPROCESS*; *PEB* contains information needed in the user space (heap address, synchronization information, list of loaded modules, . . .),
- *TEB* (Thread Environment Block) – the same for threads,
- *KPROCESS*, also *PCB* – part of *EPROCESS* with information necessary for CPU scheduling (timestamps, priority, affinity, process state, quantum, . . .),
- etc.

 **PCB in Linux.** A process in Linux is represented by the following structure (the output is shortened):

```
struct task_struct {
    volatile long state;
    unsigned int flags;
```

```

void *stack;
int prio, static_prio;
struct mm_struct *mm, *active_mm;

pid_t pid;
struct task_struct *parent;
struct list_head children;
struct task_struct *group_leader;
...
};

```

The `state` variable determines the state of the process (values `TASK_RUNNING`, `TASK_INTERRUPTIBLE`, `TASK_STOPPED`,...). The `flags` word is something similar to state, it says what the given process does work (values `PF_STARTING` for a new process, `PF_MEMALLOC` if the process is currently allocating memory,...).

The `stack` pointer leads to the currently used stack (each process has two stacks for local information about functions – one for user mode and one for kernel mode when a system call is handled). A new process acquires a basic priority (`static_prio`) that changes over time (dynamic priority, `prio`). The variables `mm` and `active_mm` represent the address space of the given process.

Each process has its PID (the `pid` variable). Every process has to know its parent (`parent` is the pointer to the structure of the parent process) and its children (the `children` array). Processes are grouped together in groups, each group has a leading process of the group (the `group_leader` pointer).

The `task_struct` structure is very long, contains a lot of information.




Additional information

- <https://tampub.uta.fi/bitstream/handle/10024/96864/GRADU-1428493916.pdf>
- <https://www.ibm.com/developerworks/library/l-linux-process-management/index.html>
- <https://unix.stackexchange.com/questions/80038/what-is-the-structure-of-a-linux-process>
- <https://www.microsoftpressstore.com/articles/article.aspx?p=2233328>
- <https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/eprocess>



4.4 Operations on Processes

4.4.1 Process Input and Output

 In common operating systems, processes can read from the standard input (`stdin`), write to the standard output (`stdout`) and write errors to the standard error output (`stderr`). The default direction of this type of communication is from/to console, it means input from a keyboard (for `stdin`) and output to a display (`stdout`, `stderr`). It all can be redirected, e.g. we can redirect input from keyboard to a file (so we use the same method for reading from file instead from keyboard), and we can redirect `stdout` and `stderr` to a file instead of a display.

Besides, they can also open files (for reading, writing, or both), and processes with a graphical interface can work with structures representing their windows (forms).

We focus on standard input, output and error output. This possibility is applicable to all processes, including those that do not have a window, in all operating systems.

There are the following ways to use the concept:

- a program writes its output into a display by default, but a user working with this program wants to save output to a file; so the user can redirect output to a file, similarly for stdin and stderr,
- a programmer can handle some data over either to another process or write them to a display or store to a file, but he does not want to deal with several different commands with conditions, he can leave it to a user of the program (as in the previous item),
- working with standard input and output can help with interprocess communication, this option is used in communication between parent and child processes.



In Windows and UNIX systems, this syntax is used (it comes from UNIX):

`program > file` the standard output of the given program is redirected to the given file, the original content of the file is deleted, replaced by the program output (if the file has not yet existed, is created)

`program >> file` the standard output of the given program is redirected as in the previous case, but the standard output is appended to the original content (at the end of the given file)

`program 2> file` the standard error output is redirected to the given file

`program > file1 2> file2` we redirect the both standard output and standard error output, into different files

`program &> file` we redirect the both standard output and error output, into the same file (syntax for UNIX and UNIX-like systems)

`program > file 2>&1` we redirect the both standard output and error output, into the same file (Windows syntax)

`program < file` the standard input is redirected, we read from the given file instead of keyboard

`program1 | program2 | program3...` the standard output of each program is redirected to the standard input of the next program, we call this structure by “*pipeline*”

`program1 | program2 > file` we can combine these concepts

We can use either common files, or some special files with special meaning. For example, the output file `NUL` in Windows, `/dev/null` in UNIX systems, is “black hole” – whatever is here redirected, it is discarded, is not stored or written anywhere. We use it if the output or error output is not important, or it could bother users.

The input files `/dev/random` and `/dev/urandom` are used in UNIX systems to input random numbers. In Windows, we use the variable `random` for these purposes.



Example

In Windows we can type these commands:

`dir > file` the output of the command `dir` is redirected to the file, instead of display

`dir nonsense 2> file` the error output of the command `dir` is redirected to the file

`dir nonsense > file 2>&1` the output and the error output are redirected to the file

`dir nonsense 2> NUL` the error output is discarded

`dir nonsense > NUL 2>&1` the both the output and error output are discarded

`dir /s /b *.sys > c:\users\some_user\kernel_driver_files.txt` (first type `c:` and `cd\`) the output of the command `dir /s /b *.sys` is redirected into the given file located in the user's profile

`dir c:\windows | sort | more` the output of `dir` is sorted, and the last command allows to scroll through the output (Enter causes moving one row down, spacebar causes moving one page down)

`sort < file1 > file2` contents of the first file is sorted and stored into the second file

`tasklist /v | findstr /i "svchost.exe"` we want to filter the output of the first command



Example

In the newer versions of Windows we can use *alternate data streams*. Some operations with them are allowed only with using redirection. Let us try:

`echo xxx > file.txt` we created the given file, with the content “xxx”

`echo yyy > file.txt:something` we created alternative data stream associated to the file

`dir fi*` only the “official” file is visible, the alternative data stream is not visible – output:

```
Volume in drive C has no label.
Volume Serial Number is 9C71-C2F2

Directory of C:\Users\ui

19.02.2019  18:09                6 file.txt
                1 File(s)                6 bytes
```

`dir /r fi*` the alternative data stream is visible too – output:

```
Volume in drive C has no label.
Volume Serial Number is 9C71-C2F2

Directory of C:\Users\ui

19.02.2019  18:09                6 file.txt
                6 file.txt:somestream:$DATA
                1 File(s)                6 bytes
```

`type file.txt` only the “official” content is visible

`type file.txt:something` error message

`more file.txt:something` error message

`more < file.txt:something` this output is ok



Example

In Linux, some commands are different, but the redirection principle is similar.

`ls > file` redirection of the standard output

`ls nonsense 2> /dev/null` error output is discarded

`time >> file.log` current time is redirected to the end of the given file


`tr 'a-z' 'A-Z' < file1 > file2` capitalizes all characters in the first files, the result is stored into the second file


```
find /etc -name hosts 2> /dev/null    error messages will be lost
ls -la | head -10                    we want only first 10 rows of the output
ps -ef | grep firefox                we want information about the firefox processes
dmesg | grep -i usb                  something is wrong with the USB interface, so we want know if there was some
                                     problem with loading USB drivers
```



4.4.2 Process Creation and Termination

The process is (almost) always created by another process, which determines the parent-child relationship. The parent process either continues to execute its code concurrently with the child process, or waits for termination of the child process.

 In Windows, the most common way to create a child process, is the function `CreateProcess()`. This function allows to pass various parameters to the created process, including standard input and output redirection. It checks the type of the file which is to become the image of the new process, and if it is not a 64-bit PE file, the function will ensure that the virtual environment is started (WoW64, ntvdm, ...).

 In UNIX and UNIX-like systems, new processes are created by combination of the functions `fork()` and `exec()` (or some variant of this function, e.g. `execve()`). `fork()` makes copy of the original process, `execve()` replaces the code in order to allow the new process to interpret its own executable file, and restarts the program counter.

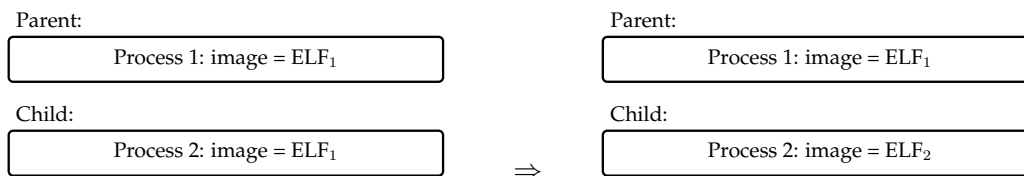


Figure 4.7: Image of the child process before and after `execve()` calling




Example


A new process in Linux is created as follows:

```
// make a copy of me (of this process), the new instance (process) has different PID,
// but the same code and registers content:
pid_t child_pid = fork();
if (child_pid == 0) {
    // fork in first instance (parent) returns PID of the created child process,
    // in the new process (child) returns 0
    // if this is child, replace the original code with a new code (from another image):
    execve(...)
}
// parent's code
```



 Some operating systems (such as UNIX systems) create the *process tree*. Each process has, in addition to its own PID, a parental identification number (PPID), this number is used to construct the process tree. In Windows, the concept of process structure is lightweight, there is no single-root process tree. On the other hand, in UNIX systems, the tree structure is strictly adhered to, the tree process root being `init` or `systemd` or another similar process.

There is a relationship of dependence between the parent and the child process. Usually, after terminating of the parent process, all processes of its subtree are terminated as well. A typical example is the termination of all processes started by the user after logging out (all in the subtree of his login or initialization process).

 This means that after process termination

- all child processes are terminated too, or
- all child processes become orphans, they run but without parent process.

The first possibility is typical for UNIX and UNIX-like systems (all processes in the subtree obtain the `SIGTERM` signal), the second possibility for Windows.

If in UNIX we want a user process to run even after the user logs off, it must be created with the `nohup` flag and its parent becomes the root process (`init`, `systemd`, etc.).


 **Additional information**

<http://tuxthink.blogspot.com/2012/06/using-execve.html>




4.4.3 Priorities

Each process has its priority assigned. This priority is used for various purposes, especially for CPU scheduling.

 There are these types of priorities:

- *base priority* – each process obtains this priority when created,
- *dynamic priority* – it moves around the base priority, it is used to temporarily favor or, on the contrary, to disadvantage the process in connection with the use of resources,
- *static priority* – realtime processes do not use a dynamic priority, their priority stands still, they have unchanging static priority.


 **Priorities in Windows.** The range for process priorities is 1–31. The values 1–15 are dynamic for common processes, the values 16–31 are intended to real-time processes. The priority values are in these levels:

- 0 – system level, for special system thread
- about 4 – Lowest (Idle), these processes work if no other process works
- about 6 – Below-normal
- about 8 – Normal
- about 10 – Above-normal
- about 13 (max. 15) – Highest
- about 24 – Realtime
- 31 – Time-critical realtime

User processes have the base priority 8 (normal). System processes have the base priority 8 as well, only several the most important processes have a bit higher priority (e.g. `csrss` and `winlogon` have 13, `services.exe` has usually 9).


 We can discover the priority in


- *Task Manager*,
- *Process Explorer*.

 We can run a process with the chosen priority in this way:

```
start /low notepad.exe
```

(the given process is started with the low priority). For other possibilities see `start /?`.

 **Priorities in Linux.** The range of priorities stored in kernel structures is 1–40 for user processes (dynamic priorities), realtime processes use static priorities with the range 1–99.


 From a user’s perspective, the priorities map to the range called *nice*. Nice is actually a base priority, the dynamic priority moves around with a variance of approximately 5. The “nice” values are understood to be exactly the opposite of the priority values: the higher priority means the lower nice value, and vice versa. This metric can be imagined as a degree of “niceness” of the process in relation to other processes – a process with a higher “nice” is more comfortable with other processes, using less time in processor, while a process with a lower “nice” is less useful for other processes, more processor time is left to itself.

The “nice” value is:

- 0 – common priority,
- 1...19 – increasing *nice*, this process is nicer to other processes, its priority is lowered,
- (–20)...(–1) – decreasing *nice*, this process has higher priority.


 We can discover the priority in

- *Process monitor* or another similar application,
- some commands, e.g. `top`, or `ps -eo pid,nice,cmd,start` (displays PID, nice, command and start time for each process)

 We can run a process with the chosen priority (nice):

```
nice -n 4 pstree
```

(the given process is started with higher nice value = lower priority).


 We can change priority of an existing process, we of course have to know PID of this process:

```
renice +4 1082
```

(the nice value has been increased, priority has been decreased).

4.5 Multitasking


4.5.1 Context Switching

 The *process context* is a summary of runtime information about the process. In particular, here we include information that could be lost when processes are changed on the processor.


Usually, these items may be in process context:

- address registers contents, including the program counter,
- flags register, data or general registers,
- registers of mathematical coprocessor,...

The kind of information stored in the process context depends on the type of multitasking.


 When processes change on a single processor, we call the procedure by *context switching*, that is, a change in runtime information stored in “global” locations (e.g. processor registers), so the context of the currently running process must be saved into the process structure (PCB) or into the space of the process (its stack) and the context of the next running process must be restored into the global locations. So – context switching means saving the current context and restoring the new context.

4.5.2 Types of Multitasking

 **Cooperative multitasking** allows one process running *on foreground*, the remaining processes are *on background*. The process on the foreground has a processor, but if it does not need it (e.g. waiting for an event, such as keyboard entry), the processor may be assigned to a background process, but only for a short time. Next, the processor is returned to the foreground process or, if this process is still waiting for the event, again to some background process. The user determines which process is in the foreground (e.g. it moves from the text editor to the calculator).


Processes must cooperate on multitasking, and from time to time return processor by calling the system service (the foreground process when waiting, the background processes after a dedicated processor assignment time). But it is up to the process itself (its programmer) to call up the appropriate service, and if it does not bother, other processes cannot work.

This type of multitasking has been used in Windows with DOS kernel (version 3.x or older) or Apple MacOS before the “X” version.

 **Preemptive multitasking** consists in switching processes when events arrive (IRQs, etc.). The processes do not cooperate on multitasking (and even may not know about it), and any process can be suspended at any time.


The context switching comes at each event in the system, and the processor is allocated to the target of the event (for example, after an IRQ from the keyboard, the processor is assigned to a process which the keystroke is intended to, such as a text editor). It is not necessary that the context switches after each interruption because an IRQ can be (and often is) intended to the currently active process.

The system has to preempt loss of intermediate results and addresses, so the process context is usually larger to avoid this problem. Processes assume their time to run is continuous, they do not “know” about suspensions.


 **Preemptive multitasking with Time-slicing** is an improvement of the previous method. Context switching occurs not only when an event is generated, but also at the given time intervals, and very short (units up to tens of milliseconds). Processes are routed in processor usage, and so quickly that the user experiences the impression of parallel job processing. The process is interrupted after a certain period of time, or earlier, if the previous event processing has been interrupted by another event during the allocated interval, or when the work is completed before the end of the interval.

This type of multitasking is used in all current operating systems – UNIX and UNIX-like systems including Linux, Apple MacOS, Windows.

4.6 Multithreading


 Multithreading is actually the parallel processing of multiple parts within a single process, sort of like multitasking within a process. Splitting a process into several such parts, subprocesses, threads, is advantageous if the process consists of several independent pieces of code (they do not affect each other, it does not matter in which order they are executed).

A typical example is an application that communicates with the user, allows him to work or entertains him (one thread) and “in the background” for example copies files (the other thread). Each of these threads works independently, one of them does not affect the activity of the other except for possible communication (the first thread can show the user on a suitable graphic element how far the second thread is in copying, the second thread always sends a message to the first thread after copying one file or a certain quantum of Bytes).


 In operating systems that support multithreading, a process (job) consists of one or more threads called *threads*, one thread is usually the main thread and is started when the process starts. The process is only a passive owner of memory space, all activity is performed by threads. Therefore, a process that has no threads can be terminated. Just as a process has a PID number, each thread is assigned a TID number, which in operating systems tends to be unique for the entire system.

CPU is not allocated to processes, but to threads. Each thread has its own code (or can be shared within a process, depending on the implementation) and a pointer to it (program counter), stack, CPU time, context. Threads may each have their own memory space or may access a shared memory space (the latter is more common), it is not necessary to apply such strict memory protection mechanisms or other security methods between threads (threads belong to the same process, they cooperate, they do not compete), however some synchronization may be needed when accessing resources available to multiple threads of the same process.

Threads can be implemented in several ways.

 **1:1 model.** *Threads are implemented in the kernel* of the system. The kernel treats threads as processes, so context switching is done at the thread level. This solution increases the system throughput (the system is more responsive, multiple system calls can be processed at the same time if the kernel can be also multi-threaded), but it is more challenging to solve problems related to the synchronization of system threads (related to shared system data).

This method is used, for example, by OS Mach, and therefore also by Apple MacOS, as well as in Windows NT series and in Linux (in Linux, however, the kernel itself is implicitly single-threaded) with the NPTL (Native Posix Threads Library). This specification is part of the POSIX standard.


 **N:1 model.** Threads are implemented *at the user level*. Thread support is implemented in libraries, the kernel is only single-threaded and “see” only processes, not threads. The system does not support threads, the implementation is only on the process side. Threads of a single process share the CPU time allocated to that process. Threads of a single process cannot run on multiple processors, so this model is not suitable for multiprocessor systems.

Because the switching of threads of the same process is not implemented “centrally”, it is much faster (if the process does not communicate too much with the kernel), therefore the response of individual user applications is better.

The advantage is less complications when accessing system data, the disadvantage is the possibility

to process only one system call at a time within the kernel. When a thread makes a kernel service call (system call), all threads in the process stop.

This solution is used in some languages as a custom thread implementation (for example, Java or Ruby). We also see it in the form *Windows Fibers* – in Windows, in addition to threads, there are also threads whose scheduling is fully in the control of the application (i.e. they are visible only in the user space), using the `SwitchToFiber` function.


 **N:M model.** Hybrid approach. Threads are implemented at the kernel level (kernel-thread) and at the running process level (user-thread). This model removes the disadvantages of the previous two models – thread switching is fast, threads can run on multiple processors, the kernel can handle multiple system calls at once).

Each user thread of a process that makes a system call is connected to a kernel thread; other user threads that do not communicate with the kernel do not need this connection.

This model is used in most commercial UNIX systems (such as Solaris).


 **Remark**

The different models are derived from how many of which thread types are mapped between user space and the kernel.

The 1:1 model means that one thread in user space is mapped to one thread in the kernel (i.e., the kernel works directly with user threads). The N:1 model means that multiple user-space threads (i.e., threads of a single process) are mapped to a single common kernel thread (the kernel does not see threads, only processes). The N:M model represents a solution in which a set of user threads can be mapped to a (generally varying) set of kernel threads, i.e. there may even be a situation where one user thread is mapped to multiple kernel threads, which would be impossible in previous models. 

4.7 Interprocess Communication

4.7.1 IPC concept

 One of the advantages of multitasked operating system is possibility of easy communication among processes – *IPC* (InterProcess Communication).

 One process is *sender*, one or more processes are *receivers*. A sender can send

- direct data, text string, etc.,
- address of data (an address in memory or storage, it can be a temporary file),
- signal (number with determined meaning, e.g. `SIGTERM` with the meaning “terminate” or `SIGSTOP` with the meaning “stop working”).

There are two types of IPC:

- direct – sender knows receivers, sends them directly (sending messages),
- indirect – sender does not know receivers, communication passes through a shared location (clipboard, socket, pipeline).

If the receiver is only one and the sender directly addresses it, the communication model is called *unicast*; if the message (or any data) is intended for all who can communicate, then it is the *broadcast* model; if there are more specific addressed receivers, the model is called *multicast*.

 **Direct communication** is usually implemented as sending messages or signals. We assume the following functions (or similarly named):

- `send(P, message)` – the sender uses this function to send message, P is the receiver,
- `receive(Q, message)` or `receive(message)` – the receiver collects message from its message queue.

 **Remark**

These function names are really just illustrative. Actual names of functions depend on the operating system and the specific type of communication.


For example: in UNIX systems, the most common communication mechanism is the sending of signals. A signal can be sent by calling one of the functions `kill()`, `killpg()` or `sigqueue()`, we have no `send()` function. For example, this command in the C language sends SIGTERM:

```
kill(receiver_pid, SIGTERM);
```


We do not even have any `receive()` function, instead, each process has a signal capture mechanism that defines a default response for each signal (for example, terminating the process as response to the SIGTERM signal), and where the programmer can hang up its own function to a signal. For example, if he wants to run the `on_sigterm(int num)` function on the SIGTERM signal, he uses this command in the C language:

```
signal(SIGTERM,on_sigterm);
```



 Direct communication can be divided into two categories:


- *symmetrical* – the both sender and receiver are able to identify each other,
- *asymmetric* – the receiver does not need to know the sender, only the sender knows who the message is sending to.

 There are the following types of direct communication:


- *asynchronous* – the sender does not have to wait for reply,
- *synchronous* – the sender must wait for the confirmation of the message or the response (until then it is blocked), in the waiting/blocked/suspend state.

The synchronous communication needs the third function, we use:

- `send(P, message)`,
- `receive(Q, message)` or `receive(message)`,
- `reply(P, message)`.

 **Indirect communication** takes place via the interface represented by a connection point, usually called port, gateway, socket, clipboard. We assume the following functions (or similar):


- `send(port_ID,data)` – sender puts data into the given port,
- `receive(port_ID,data)` – receiver collects data from the given port.

 A *socket* is a communication interface with the “client-server” type of communication. There are two types of sockets: network sockets and UNIX domain sockets.

A network socket is intended for communication via computer network. A socket address is a pair (IP address, port number), the port number in the OSI transport layer sense (protocols TCP, UDP, SCTP, etc.). So, each connection using TCP, UDP, . . . in one direction between two computing devices

is determined by a source socket and a destination socket. Network sockets are in one direction only, so e.g. in the TCP communication we need at least two sockets, one for each direction.



A UNIX domain socket is implemented in UNIX kernel, not in network protocols. This concept is used for communication inside one instance of operating system, not between two systems via network, it is used to transfer data between processes, unlike the network socket in both directions. The API is similar to network sockets API, but addresses of sockets are special files in UNIX file systems (files of the type “socket”).

 A *pipe* is similar to a socket, but the implementation is different. This concept is strictly in one direction, a sender only sends data, a receiver only receives data. Pipes are implemented as special file interfaces too, a sender writes into the file and receiver reads from this file.

Pipes are anonymous or named. An anonymous pipe typically transfers data between a parent and child process, or in command line (text shell) between neighbor processes in pipeline, e.g. `program1 | program2 | ...`. Named pipes transfer data from one sender to one or more receivers, and all processes knowing the name of this pipe can be receivers.

The implementation of pipes is different in various operating systems, including type of the shared communication point. In some systems, a full-duplex pipe can be used, although the original concept is one-direction only.

4.7.2 IPC in Windows

 **Window messages.** Communication in user space is primarily via window messages. Each window has an associated function *Window procedure*, which is executed whenever a message is delivered to this window. For example: when clicking the cross in the corner of the main application window , we send the message “terminate” to the application (its main window).

The Window procedure of the main window of the process is the most important – if a process does not call it too long time (does not work with its message queue), this window stops responding and e.g. it is not possible to terminate the application, application may be frozen.

Each window is uniquely identified by its *handler*, like any other Windows object, in this case of the type HWND (Handle to Window). The handle of the target window is one of the parameters of the message, the next important parameter is the identifier of the message type (e.g. WM_PAINT to indicate that the window should be redrawn because the displayed data has changed).



Additional information

- <https://docs.microsoft.com/en-us/windows/desktop/winmsg/window-procedures>
- <https://docs.microsoft.com/en-us/windows/desktop/learnwin32/writing-the-window-procedure>





Messages are related to *events* – applications are managed by events, and when a specific application is created or generated, the application is informed by the message. An application at the appropriate intervals (when the application has no code to execute) checks whether there is a message to receive.

There are two categories of messages:


- messages intended to put into message queue,
- messages not intended to put into message queue.


The most messages are of the first type (e.g. the above mentioned `WM_PAINT`, `WM_KEYDOWN` or `WM_QUIT` for regular closing window or application termination). The time-critical messages have to be that can not wait in the queue and need to be processed right away, e.g. `WM_SETFOCUS` (window obtained focus because of keyboard IRQ, input from keyboard is directed into this window), `WM_WINDOWPOSCHANGED` (position of window has changed), `WM_ACTIVE` (window has been activated, e.g. by keyboard or mouse), etc. These messages are sent to window directly.

 **System calls.** System calls are API functions, threads use them when asking a service from the kernel. These functions are something as secure interface to kernel.


 **LPC (Local Procedure Call).** This mechanism is not supported in kernel API, it is an internal kernel mechanism (although it is implemented in `NTDLL.DLL`, it is not documented and common user processes are not accessible to it). It is a client-server communication, only in one direction.

LPC serves primarily to communicate within the kernel (for example, Winlogon communicates with the LSASS subsystem in this way). System processes running in user mode have access to this mechanism, not user processes. For example, the `CSRSS.EXE` (Client-Server Runtime Subsystem, part of the Windows subsystem running in user mode) uses LPC to communicate with some libraries over the kernel.


 **RPC (Remote Procedure Call).** This mechanism is intended for user processes, in contrast to LPC, it is very often used part of API. This is a call of the procedure that may be in the address space of another process (thread) or library in the supported format (`PE`, `PE+`, ...). It can be a local RPC (within a single system), or remotely called RPC (to another computer on a network). However, you can remotely call a procedure running on another computer only when the “Remote Procedure Call” service is running.

 **APC (Asynchronous Procedure Call).** APC is a mechanism that makes execution of external code possible in the context of the process (so with usage of the resources owned by this process). When the process waits for an event (e.g. I/O), the wait time can be exchanged for an APC call, that is, let a foreign code run in its context.


We distinguish between system and user APC procedures. The system APC is used to implement system calls, so the kernel code is executed in the contexts of the calling user process (kernel uses resources of the calling process, the process provides its resources to attend the system call).

 **DPC (Deferred Procedure Call).** DPC is additional interrupt handling when the processing of IRQ itself would take too long time. If an interrupt procedure requires some data transfers that are time-consuming, or an error occurs and some action is required to repeat or treat an error, the part of the interrupt handling is being implemented as DPC.

An IRQ routine is executed immediately, in contrast to DPC, which is scheduled to processor in the same way as common processes. But DPC has higher priority than processes. So, DPC is somewhere between IRQ routine handling and process execution, in terms of priority.


 **Pipes and sockets.** The concept of pipes was taken from UNIX, but in Windows is implemented differently. A pipe is a shared memory to which the sender writes his entire output, and then the recipient uses the file as his input. The sender and recipient may not exist in parallel, the recipient can be created after the sender has finished.

Windows sockets (Winsock) are implementation of network sockets, they cooperate with various network protocols (mainly with TCP/IP suite).

 **File mapping and memory sharing.** *File mapping* is intended to treat the contents of a file without transfer the contents into the address space of the process. It is advantageous for very long files – mapped files do not block space in RAM (only addresses are assigned), and a process can access this file in the same way as a directly open file. This concept is of the UNIX origin, as many other concepts.

A mapped file (or something that can be treated as a file) is either on the hard disk or anywhere else, mapping means creating a unified interface to access data.

Memory sharing is a special type of file mapping, where the source of mapping is in the address space of another process, not on disk.

 **Other possibilities.** A *clipboard* is a shared data-exchange interface for all processes with user interface (not for services without user interface) – owners of an object WinSta0. This type of communication is known all common users.


The *DDE* (Dynamic Data Exchange) is a dynamic form of data sharing between processes. Both processes must be running at the same time. This technology is actually an extension of clipboard. For example, DDE is encountered in office applications where it is used to insert some external data links. DDE binding is not considered secure.

The *COM* (Component Object Model) has been designed to make components independent of the programming language and share them among processes. Each component has its unique identifier – CLSID (Class ID), and defined its binary interface, that can be accessed, also from other processes. Another technologies are based on the COM model, including OLE, OCX, ActiveX, and partially the .NET Framework.

The *OLE* (Object Linking and Embedding) is an object technology that allows an object to be embedded in the process's data structure. This process reserves a particular area for a foreign shared object – a part of a data structure, window, document etc., and if there is a need to work with that object, a process that works with the object is run.

The OLE technology is used by various applications, for example, office applications use it to share objects between different parts (for example, if we put an Excel spreadsheet into a Word document), web browsers use OLE to display those types of files they do not know, in their own window. The shared object can be placed either in the source document, or in the target document.

4.7.3 IPC in Linux

 **Signals.** For communication between the processes, signals are very often used. They are ideal for sending simple information, which can be represented by one number.

The number of signal types depends on hardware architecture, usually with at least 30 types of signals. They are represented by a number or by a word (both forms can be used in commands, depending on what we remember better). The most used values are in Table 4.1. The signals SIGUSR1, SIGUSR2 can be defined according to own requirements, e.g. the parent process defines their meaning to communicate with its child processes (if children know the signals with the same meaning).

The SIGHUP signal: a typical process response to this signal (especially for daemons) is to retrieve the configuration files, which is actually similar to restart the process (but actually the process is

<i>Name</i>	<i>Number</i>	<i>Meaning</i>
SIGHUP	1	do change in the parent process (e.g. it was terminated), retrieve its configuration files (for daemons), or quit (common processes),
SIGINT	2	interruption (termination) of the receiving process, the same as pressing the <code>Ctrl+C</code> keyboard shortcut
SIGILL	4	Illegal (incorrect) instruction, error in instruction
SIGFPE	8	exception related to floating-point numbers (Floating-Point Exception)
SIGKILL	9	immediate termination, the receiving process cannot ignore it (unlike SIGTERM), we send it to unresponsive processes
SIGTERM	15	terminate (regular termination, the receiving process is able to clean up its resources including dynamical structures)
SIGUSR1	30,10,16	user-defined signal no 1 (process-defined)
SIGUSR2	31,12,17	user-defined signal no 2 (process-defined)
SIGCHLD	20,17,18	child process changed its state (including termination); the parent process should pick up the result
SIGSTOP	19,23	stop working (the equivalent of the <code>Ctrl+Z</code> keyboard shortcut)
SIGCONT	18,25	if stopped by SIGSTOP, continue working


Table 4.1: Usual signals in UNIX systems

running all the time), common processes are terminated upon receipt of the signal.

If we want a process in that case not to respond to the SIGHUP signal (i.e., a daemon does not reload the configuration, or a common process did not terminate) we use the following command to run this process:

```
nohup command &
```

A signal may come at any time, it is actually as IRQ, the programmer must expect it. Even the system call can be interrupted by a signal (the usual response is to immediately terminate the system call handling by making this call either be established or restarted).

 A process is able for a particular signal to

- ignore (drop) this signal – this action is default for some signals, e.g. SIGCHLD (this signal has no meaning for most processes), but some signals cannot be ignored (e.g. SIGKILL),
- block with later delivery – the signal is not dropped, but it waits for unblocking, then it is processed,
- let processed by an default handler routine – this routing either terminates the target process (for SIGTERM or SIGHUP), or stop it (SIGSTOP), or terminates process and runs debugger (SIGILL, SIGFPE),
- implement the own handler routine – e.g. when obtaining SIGTERM, we want to close files and to release dynamically allocated memory.

Signals can be understood as controlling simple events without the need to create a cycle. The handler routine should always be as short as possible, as is the case with hardware IRQs.

**Example**

If we want to terminate a process with PID 3189, we can use one of the following commands:

```
kill -s 15 3189      kill -15 3189      kill -TERM 3189
```

If the process is not responding:

```
kill -s 9 3189      kill -9 3189      kill -KILL 3189
```

The `kill -l` prints a list of signals for the current platform (“l” as “list”).



Sending signals with examples is also discussed later in Subsection 4.7.4, page 59.

**Additional information**

<https://www.thegeekstuff.com/2012/03/catch-signals-sample-c-code>



Pipes. The inventor of the pipe mechanism is Doug McIlroy, one of the most important creators of the early UNIX. Turning on and disconnecting after end of communication is done only in one process (usually parent), but opening and closing must be done in both communication processes.

This means that after turning on the pipe file (the `mkfifo` command), we open this pipe (file) with the `fopen` command in one process for reading, in the second process for writing, and when it is open at the both ends, we can transfer data. After using the file, we close it and then disconnect it with the `unlink` command.

Usage of named pipes is similar to “normal” files, only the functions have a bit different name. Anonymous pipes are similar to temporary files, and unlike named ones, their size is limited (for older kernels to 4 KB, for newer kernels to 64KB).

Anonymous pipe can be created by the `pipe` command in program code, or by the `|` symbol in shell: `ls | less` means that the output of the `ls` command is paged (for paging, we can use `more` or `less`, the second possibility is better).

Programs used in pipeline are called *filters*. A filter has its standard input and standard output, and does not have to worry about where these channels point to (file, screen, etc.), the program still does the same with these channels. The task of the filter is to transform (e.g. modify, sort, search, split, print, translate, etc.) its standard input and then pass it on to its standard output.

**Example**

An anonymous pipe is a fairly common way of communicating in UNIX systems. It is not just about use in programs chaining in a text shell, but programmers should be able to use pipes as well.

```
int pipe_descriptors[2];
int pipe_reading;
int pipe_writing;

// we create a pipe, the parameter contains descriptors of pipe ends:
pipe (pipe_descriptors);

pipe_reading = pipe_descriptors[0];    // pipe tail for reading
pipe_writing = pipe_descriptors[1];    // pipe tail for writing
```

Next, we use descriptors like file descriptors, that is, we use them in functions for reading a file or writing to file.





Example

Anonymous pipes are often intended to communication between a parent process and a child process.

```

... // including stdlib.h, stdio.h, unistd.h,...
int main() {
    int    mydescriptors[2];
    pid_t  child_pid;

    pipe(mydescriptors);
    child_pid = fork();

    if (child_pid == (pid_t) 0) { // *** child ***
        FILE *myfile;
        char buffer[1024];
        close(mydescriptors[1]); // close unused tail
        myfile = fdopen (mydescriptors[0], "r");
        while (!feof (myfile) && !ferror (myfile) && fgets (buffer, sizeof (buffer), myfile) != NULL)
            do_something(buffer); // ... do something with data
        close (mydescriptors[0]);
    }

    else { // *** parent *** ; child_pid > 0 (assuming no error)
        FILE *myfile;
        close (mydescriptors[0]); // close unused tail
        myfile = fdopen (mydescriptors[1], "w");
        ... // writing into pipe
        close (mydescriptors[1]);
    }
    return 0;
}

```



Sockets. Implementation of sockets in Linux is in the `sys/socket.h` library – the both: network sockets and UNIX domain sockets. The socket address can be of the type `AF_INET`, `AF_INET6` for network sockets, `AF_UNIX` or `AF_LOCAL` for UNIX domain sockets (these two types are synonyms). Network socket addresses are pairs of IP address and port number, UNIX domain socket addresses are file descriptors.

The third type of sockets in Linux is *netlink sockets* – it is the Linux kernel interface for IPC among kernel and user space processes. Netlink socket addresses are process PIDs, their type is denoted by `AF_NETLINK`. Netlink sockets are used by many newer programs, e.g. `iproute2` command family uses this concept for communication with network stack placed in Linux kernel (commands `ip`, `ss`, etc. – see `man ip`, `man ss`).


A socket can be imagined as a pipe with many extra features. It is also the use of predefined communication points and it is a client-server communication. Communication can be a stream that works similarly to a pipe (a data stream is being sent), or a datagram, when a datagram is first assembled and then sent as a batch with a header.



Additional information


- https://www.gnu.org/software/libc/manual/html_node/Pipes-and-FIFOs.html
- https://www.gnu.org/software/libc/manual/html_node/Sockets.html
- http://www.linuxhowtos.org/C_C++/socket.htm



 **POSIX Message Queues.** This mechanism allows processes to create custom (named) message queues. Each message has a certain priority (at least 32 priority levels are used, but may be more than 32 768 levels in Linux), messages with higher priority are delivered preferentially.

Creating a message queue is similar to creating a file (including access permissions), and the queues are also treated similarly to files or rather with files (but the function names are different). The Queue attribute is the queue length (maximum number of messages) and the maximum length of the message. When loading messages from the queue, the block of data in the form of a (long) string terminated by a null symbol is retrieved.

A process can either watch its queues by itself, or it can set a signal alert, or directly enter a service function (which will start automatically if a new message arrives in the queue).

 **Memory mapping and memory sharing.** Memory mapping is intended to treat the contents of any data source (located in memory, storage, ...) into the address space of the process.

There are two types of memory mapping – private (mmap) and shared (shm). The both are in the `sys/mman.h`.


This concept is also used as a basis for some other mechanisms in UNIX systems, including swap implementation or memory sharing.



Additional information

- <https://www.poftut.com/mmap-tutorial-with-examples-in-c-and-cpp-programming-languages/>
- <https://gist.github.com/drmalex07/5b72ecb243ea1f5b4fec37a6073d9d23>



 **System calls, library calls, RPC.** There are also system calls in Linux. They are used to communicate with kernel. Kernel API consists of set of functions to ask kernel services, their calling means switching between user mode and kernel mode. The calling process only gets to the resulting value (usually zero or positive value for successful evaluation, negative value for failure). Function arguments are transmitted through the registers, or the address of a larger amount of data may be in a register.

The *Library functions* are functions in the *standard C Library* (stdlib). Calling these functions is denoted by library call, and not all functions require kernel mode.

In Linux, we can use RPC (Remote Procedure Call) – calling functions implemented in libraries in other (remote) instances of operating systems, simply other computers. So, the meaning of RPC is a bit different from Windows.




Additional information

- <https://www.thegeekstuff.com/2012/07/system-calls-library-functions/>
- <https://www.linuxjournal.com/article/2204>




4.7.4 Jobs in UNIX

 A *process group* is a collection of one or more processes, usually with a parental binding. The main reason to create a process group is easy transition of signals – if a signal is directed to a process group, all processes in this group obtain it.

Each process group has its *Process Group ID* (PGID), taken from the PID of the main process in the given group (a group leader), usually the parent of the remaining processes in the group.


Most system processes have PGID set to 0. No process with PID 0 exists, so they are not included in any group. However, this does not apply to all system processes. For example, in Linux using the older kernel up to version 2.6, the `hald` process exists (the low-level hardware access daemon, the HAL layer of kernel), and this process is the main process of its own process group (its child processes). So, its PID is the PGID of its group.

The `smbd` daemon (“Samba” technology allows sharing files and other resources within the local network) has child processes originated from the same file (also `smbd`), all belonging to the same group, and the same principle applies to some other daemons (such as `avahi-daemon`).

 A *session* is a collection of one or more process groups. It is also a subset of the process tree, for example, it may relate to the processes of a single logged-on user (typically associated with the terminal from which the processes are started).

A typical reason for the inclusion of processes in one session is the possibility of cascaded sending signals (processes distribute the signal to their children, those to their children etc., recursively). Processes belonging to one group (having the same PGID) also belong to one session, but not vice versa.

Each session has a *Session ID* (SID), and it is usually PID of the main process in the session – the session leader PID. Most system processes have their SID set to 0, as well as their PGID.

 A *job* is a shell’s representation of a process group, so for example all processes in one pipeline belong to the same job. Jobs management is always performed within a shell, i.e. within a terminal or console.

While running jobs, full multitasking is provided, even if we do not use graphics mode. This means that in terminal we can run as many processes as we want, in parallel. One of them can run in the *foreground*, others run in the *background*. The both foreground and background processes can write their output to the terminal.

We can manipulate with processes using their PIDs (but PIDs are big numbers), or using the job numbers. Job numbers are small, jobs are numbered relative to shell. In order to differentiate these two types of numbering, the job numbers begin with %, so the designation of jobs is %1, %2, ... We can use these commands:

`jobs` lists all jobs on terminal, each row begins with the job number, followed by “+” for the last process that went in the background, or “-” for the previous one

`fg [job_num]` sends the given job to foreground; if no parameter is used, the last backgrounded job is chosen,

`bg [job_num]` lets the given job to continue, if it is stopped; the job remains in the background, but it can work,

`some_command &` the given command is started in the background, the prompt of the command line is displayed as we are able to enter additional commands.

When starting a command with the `&` sign, then if the command has “something to work”, it really gets the processor and runs (it is in the “running” state); if it has something to write, its output appears in the terminal.



Example

Let us play with processes – jobs. First, we start several processes/jobs in background. The prompt is displayed in blue.

```
sarka@DellVostro $ cat &
[1] 7345
sarka@DellVostro $ rev &
[2] 7346
sarka@DellVostro $ yes > /dev/null &
[3] 7347
```

We have started three processes in the background, their job numbers are written in the square brackets ([1],...), followed by their process numbers.

Now we write list of processes running in the terminal (if we wanted to list all running processes, we would use `ps aux` or `ps -ef`, but we only want processes at our terminal):

```
sarka@DellVostro $ ps
PID TTY          TIME CMD
7321 pts/0        00:00:00 bash
7345 pts/0        00:00:00 cat
7346 pts/0        00:00:00 rev
7347 pts/0        00:00:08 yes
7349 pts/0        00:00:00 ps

OR

sarka@DellVostro $ ps -Ho pid,pgid,sid,comm
PID  PGID  SID  COMMAND
7321  7321  7321  bash
7345  7345  7321  cat
7346  7346  7321  rev
7347  7347  7321  yes
7350  7350  7321  ps
```

The second used command displays three identifiers of the processes, including PGID and SID. As we can see, all processes are in the same session (have the same SID), but different PGID. The leading process in session is `bash` (the shell used in our terminal). The list of jobs in the terminal:

```
sarka@DellVostro $ jobs
[1]-  Stopped                  cat
[2]+  Stopped                  rev
[3]   Running                 yes > /dev/null &
```

The first two processes/jobs wait for user action, they are stopped (the `cat` command without parameters is something as “echo” – it writes everything written back to terminal, the `rev` does the same, but also performs reversal). The third process is active, running – writing the symbol “y” into the terminal, so we have redirected its output into the “trash” represented by the special file `/dev/null`.

We send the third job to the foreground, so the given process “takes” prompt and runs, but its output goes to trash:

```
sarka@DellVostro $ fg 3
yes > /dev/null
```

To send this job to the background, we press the keyboard shortcut `Ctrl+Z`, it means sending the SIGSTOP signal:

```
^Z
[3]+  Stopped                  yes > /dev/null
sarka@DellVostro $ jobs
[1]   Stopped                  cat
[2]-  Stopped                  rev
[3]+  Stopped                  yes > /dev/null
```

We want the (paused/stopped) job 3 to run in the background:

```
sarka@DellVostro $ bg 3
[3]+ yes > /dev/null &
```

```
sarka@DellVostro $ jobs
[1]- Stopped          cat
[2]+ Stopped          rev
[3]  Running           yes > /dev/null &
```

It no longer has fun, so we end the job with the job number %1, and the all remaining jobs:

```
sarka@DellVostro $ kill %1
sarka@DellVostro $ jobs
[1]  Terminated      cat
[2]+ Stopped          rev
[3]- Running           yes > /dev/null &
sarka@DellVostro $ kill %2 %3
sarka@DellVostro $ jobs
[2]- Done             rev
[3]+ Terminated      yes > /dev/null
```



Example

Now let us have a look at the possibility of more processes in one process group. For example, parent and child processes communicating over pipe are often in one process group.

The `man` command is intended for listing manual pages of commands, configuration files, functions, etc. To write a manual page for the `ls` (for listing file contents, similarly as `dir` in Windows command line), we type this command:

```
man ls
```

The manual page is displayed in interactive mode using a paging command (usually `less` is used, but it can be renamed “`pager`”). So, two processes are running: `man` and `less/pager`.

```
sarka@DellVostro $ man ls
```

Now we press `Ctrl+Z` and list jobs running in our terminal:

```
sarka@DellVostro $ jobs
[1]+ Stopped          man ls
```

However, the list of processes running in our terminal looks a bit different:

```
sarka@DellVostro $ ps -Ho pid,ppid,pgid,sid,command
PID  PPID PGID  SID  COMMAND
2197 2191 2197 2197  bash
3839 2197 3839 2197  man ls
3851 3839 3839 2197  pager
4052 2197 4052 2197  ps -Ho pid,ppid,pgid,sid,command
```


The processes with PID 3839 (`man ls`) and 3851 (`pager`) are in the same job, and in the same process group (they have the same PGID 3839 = PID of the first of these two processes). The `pager` process is the child process of the parent process `man ls` (see the PPID column).

Let us run the job %1 in the foreground (it is not necessary to enter the job number, we have only one job):

```
sarka@DellVostro $ fg
```

This process can be terminated by pressing the `q` key, as most interactive programs, or we can use the keyboard shortcut `Ctrl+C`.



 A signal can be sent to a process whose PID is known. This can be done both in the binary code of a running process and in a text shell (for example, `kill`, `killall`, `pkill`, etc.). One of the parameters is the signal number or word, and the further parameter determines the process to which the signal is intended. This is usually a PID, the `pkill` function also accepts other process identification types – process name, PID, PGID, SID, etc.

Example

We show usage of the commands `kill`, `pkill` and `killall`.

`kill -l` list of all accepted signals for the given platform

`kill -9 6223` sends the SIGKILL signal to the process with PID 6223

`kill -KILL 6223` the same

`kill -9 %2` send the SIGKILL signal the the job #2

`kill %2` if no signal is assigned, the SIGTERM signal (#15) is sent (standard termination of a process, but some processes ignore it)

`pkill -HUP syslogd` sends the SIGHUP signal to all processes with the name `syslogd` (logging daemon), it means that the daemon has to reload its konfiguration files (common processes terminate while obtaining this signal)

`pkill -P 5421` sends SIGTERM to all processes with the given PPID (parent PID) – all child processes of the given process

`pkill -u 1002` terminates all processes started by the given user (with this UID – user ID)


`killall -g 3981` terminates all processes in the given process group (we assign the group leader PID = PGID)



Additional information

- <https://www.tldp.org/LDP/abs/html/x9644.html>
- https://en.wikibooks.org/wiki/A_Quick_Introduction_to_Unix/Job_Control
- http://linuxcommand.org/lc3_lts0100.php



 If a process must not be terminated with the end of the session, we need to do one of the two actions:

1. We start the process inside different session – with the `setsid()` system call (in a program code, not in a shell), but we are able to do it only if the given process is not a session leader.
2. We ensure that the process is not in the list of active jobs and that SIGTERM (or SIGKILL) is not sent to it – we have the `nohup` and `disown` commands:

```
nohup some_program &
disown %job_number
```


4.8 CPU Scheduling

4.8.1 Basic Concepts

We generally talk about CPU scheduling for processors, but in most current operating systems CPU is allocated to threads.

 Two parts (e.g. modules) of kernel cooperate in CPU scheduling:

- *CPU scheduler* – this module uses process queues (with ready processes) and determines which process is assigned to processor and how long time for,
- *dispatcher* – performs context switching and processor assignment, thus saves the context of the currently running process including the program counter, retrieves the context of the process to which the processor is currently allocated, detects the value of the program counter, and determines the location of the process code run, and if multiple CPU modes are supported (kernel mode, user mode), then the dispatcher performs switching between these modes.

 The *burst time* (execution time) of a process is the amount of time required by a process for its execution. The usual burst time is in the order of tens to hundreds of milliseconds.


Processes are:

- *CPU-bound* – these processes use a processor a lot (such as services),
- *I/O-bound* – *interactive processes*, they more use I/O devices, including graphical output or various types of input.

Each of these types of processes has a little different processor requirements. CPU-bound processes typically use the entire assigned value of the burst time; I/O-bound processes, with higher probability of I/O interruption, have lower burst time. The CPU scheduler should distinguish between these process groups so that the CPU usage is optimal.

The scheduling algorithms used by a CPU scheduler are


- *preemptive* – a process can be interrupted – placed into the ready queue – before depleting its burst time,
- *nonpreemptive* – a process cannot be interrupted, it uses up its burst time and usually goes to a different queue than the ready queue (e.g. waiting for I/O, performing system call).

 A *time quantum* (timeslice, processor slice) in preemptive multitasking is the period of time that the process can spend on the processor while steadily subtracting its real burst time pieces from this time.

The multitasking functionality and hence the quality of the CPU scheduling algorithm depend at the appropriate burst time. If too short, the overhead time for context switching is high compared to the actual running time, and the system is disproportionately slow. If the burst time is unnecessarily long, then processes using a lot of I/O devices use only a small portion of their quantum, the processor must be switched more often, and the system is less interactive.


Next, we discuss basic CPU scheduling algorithms. The typical use is a combination of several of these algorithms.

4.8.2 Scheduling Algorithms

 **First-Come First Served (FCFS).** This basic algorithm is simply a FIFO (First-in First-out) queue, the processes are put to the end of the ready queue and are taken from the beginning.


It is a nonpreemptive method, processes use the processor until the interrupt is generated or if they pass processor themselves.

The disadvantage is that CPU-bound processes reserve too much processor time and therefore I/O-bound processes are disadvantaged. This method is therefore only implementable in combination with process priorities (I/O-bound processes should have higher priority).


 **Round-Robin (RR).** Round-Robin is similar to the previous algorithm, we also use a queue with the FIFO organizations. But this method is preemptive.

When a process spends its burst time (or is interrupted), it is placed at the end of the queue of ready processes if it is not included in a different queue (e.g. when it is interrupted).

If the time quantum is too large, the functionality of this algorithm corresponds to the previous method. Again, CPU-bound processes are favored because they anticipate waiting I/O-bound processes.

 **Shortest Job First (SJF).** The processor is assigned to the process that are expected to work the shortest time (they have the shortest burst time). The queue is preceded by the priority being determined by the size of the assumed used quantum.

The method has a preemptive and nonpreemptive version. In preemptive scheduling, when a process with shorter expected burst time then the running process comes to the queue, the running process is immediately interrupted and the processor is assigned to the incoming process.

 It is necessary to estimate the burst time for the current (running) stretch of the process. There are several methods to estimate the burst time (n is number of passed processor assignments to the given process), R is the array of the real burst times, and S is the array of estimates of burst time.

1. The following burst time is often the same as the previous burst time, so we can assume that the next time the process will need is about as much time as it used during the last assignment.

$$S[n + 1] = R[n] \quad (4.1)$$

2. Exponential average – for each process, we record the length of the time actually used for the processor allocation in the past, and we estimate the appropriate burst time by calculating the arithmetic average of all previous real burst times. The formula can be simplified by using the previous estimate and the corresponding real burst time.

$$S[n + 1] = \frac{1}{n} \cdot \sum_{i=1}^n \cdot R[i] \quad (4.2)$$

$$\begin{aligned} &= \frac{1}{n} \cdot R[n] + \frac{1}{n} \sum_{i=1}^{n-1} \cdot R[i] \\ &= \frac{1}{n} \cdot R[n] + \frac{n-1}{n} \cdot S[n] \end{aligned} \quad (4.3)$$

3. We will combine both approaches (according to the formulas 4.1 and 4.3), so we can assume that the next burst time will not be too different from the previous, but we take account of history (with less weight).


We choose the appropriate constant c , $0 < c < 1$. If this constant is closer to one, the weight of the last real burst time is greater, and if it is closer to zero, history has greater weight. The first estimate ($S[1]$) is usually set to 0.

$$S[n + 1] = c \cdot R[n] + (1 - c) \cdot S[n] \quad (4.4)$$

The meaning of the constant c is evident from the recursive distribution of the formula:


$$\begin{aligned}
 S[n+1] &= c \cdot R[n] + (1-c) \cdot (c \cdot R[n-1] + (1-c) \cdot S[n-1]) \\
 &\quad \vdots \\
 &= c \cdot R[n] + (1-c) \cdot c \cdot R[n-1] + \dots \\
 &\quad \dots + (1-c)^{n-1} \cdot c \cdot R[1] + (1-c)^n \cdot S[1]
 \end{aligned} \tag{4.5}$$

This algorithm favors I/O-bound processes whose burst time is less, and greatly disadvantages CPU-bound processes. Longer running processes can be aging, so they lose importance. If there is an error in a process (infinite loop), then the malfunctioning process does not block the processor and can be easily detected (stays at the end of the queue, it is constantly waiting and not running).

 **Priority scheduling.** When applying this method, we assign the processor to a process with the highest priority, that is, we use the priority queue. The method has a preemptive and non-preemptive variant, as well as the previous.

Alternatively, we can consider the SJF (previous) method, where the priority derives from the process burst time (the smaller quantum means the higher priority).


Dynamic priority is used, it reduces risk of aging low priority processes, priority may be gradually increased in longer waiting processes.

 **Multilevel queue scheduling.** This algorithm uses several separate queues. The particular queues are intended for a group of processes (depending on their priority, type, etc.), and each queue has its priority.

4.8.3 Scheduling in Windows

Windows *CPU Scheduler* schedules threads regardless of the number of threads in each process (i.e., the threads wait in queues). In the kernel, there is a module for planning the allocation of the processor (its cores) to the threads – CPU scheduler, and the Vista version and later also provide I/O scheduler (schedules access to other resources).

Dispatcher, which performs context switching according to scheduler requirements, is not a specific function or library, its components are in different kernel modules. It is based on the fact that switching takes place during events (event-driven switching).

 When planning threads, preemptive multi-queue planning is used, the processor thread can be interrupted at any time if the processor is requested by a thread with higher priority. If the thread does not use the entire burst time, it can leave this unused time to another thread of the same process by calling `SwitchToThread()`.

There is one queue of ready processes for each priority, a total of 32 priorities, 0–31. If the thread has to wait for a processor, it is queued according to its dynamic priority. Higher priority threads always have precedence over threads with lower priority, so the processor is preferably allocated to threads waiting in queues with higher priority numbers.

The processor is allocated for the time derived from *system clock interval*. This interval is fixed at a value somewhere between 10–15 ms, the actual value can be determined, for example, by the `clockres` program by Sysinternals.

By default, a thread runs for (a burst time is) 2 intervals on desktop (12 on server). For each additional step, the quantum of the running thread is subtracted, the quantum decreases slightly even while waiting for IRQ. After the quantum is exhausted, a new quantum is allocated for a thread.

A short burst time is advantageous in a system with many interactive processes (increasing system throughput, typically on a desktop with a lot of “window applications”), long burst time is advantageous for a system with many computational processes often running in the background (typically servers).

Dynamic priority of a thread can be lowered when the thread exhausted the previously allocated quantum and is assigned a new one. For example, the priority may be increased when the I/O operation or the interactive thread wakes up as a result of a window event.



Example

In GUI we can determine whether the burst time is short (2 intervals) or long (12 intervals): *System* ⇒ *Advanced System Settings* under *Performance* the *Settings* button, then the *Advanced* tab, as shown in Figure 4.8.

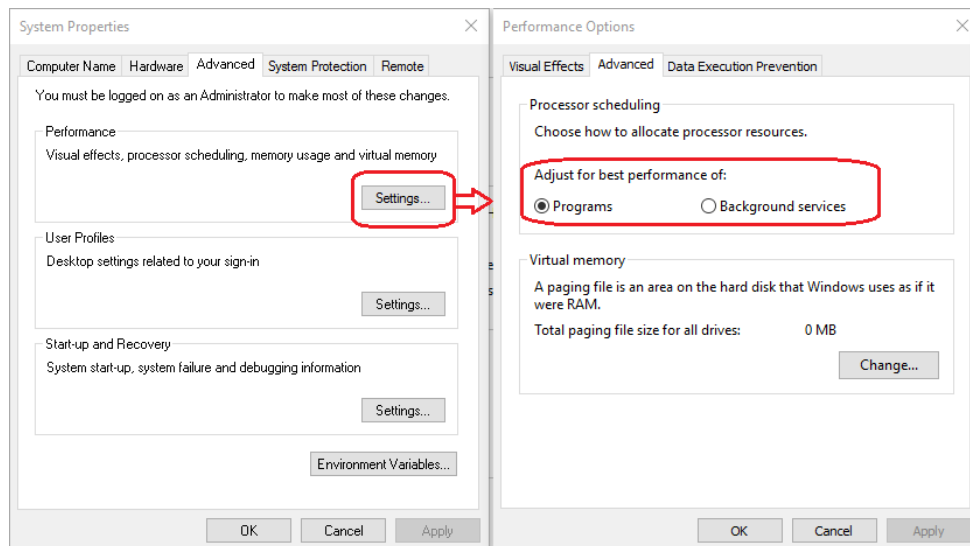



Figure 4.8: Setting burst time




 **Process affinity or thread affinity** is setting processors (or processor cores) on which the process (thread) can run. This set of processors (cores) can be limited by setting an *affinity mask*, for example, in Process Explorer (in the context menu of the process, selecting *Set Affinity*), or similarly in Task manager.

In a program code, either the API function `SetThreadAffinityMask` or `SetProcessAffinityMask` can be used. This setting is called the “hard affinity”, the “soft affinity” is the rule that the thread is preferentially scheduled for the most recently run processor.

4.8.4 Scheduling in Linux


The Linux kernel can be nonpreemptive (the `CONFIG_PREEMPT_NONE` flag), or fully preemptive (the `CONFIG_PREEMPT` flag), or voluntarily preemptive (the `CONFIG_PREEMPT_VOLUNTARY` flag).

For a desktop system, a choice of voluntary preemptive behavior is a good option for the kernel (a kernel thread, such as a system call handler, may voluntarily give up the processor), nonpreemptive behavior is recommended for servers. Fully preemptive behavior means high response, but higher switching overhead (context-switching comes more often), it is useful e.g. for embedded systems.

 In Linux, a processor is always scheduled for the time we call *epoch*. At the beginning of an epoch, each process (thread) gets a time quantum that is gradually consuming. When all processes have consumed their time quantum, a new epoch begins and all processes receive another time quantum.

Common threads use *dynamic priorities* – the priority of a long-waiting thread grows, the priority of a long-running task decreases. Real-time tasks are scheduled with a *static priority*.

The process with the priority 0 is *idle*. This process has no code, it only creates the *init/systemd* process and is scheduled in time when no other process is running.

 In old versions of Linux kernels (up to 2.4), there were *multiple queues* of ready processes (threads), each queue can have its own scheduler. These schedulers can be used:

- **SCHED_OTHER** – for common threads, it uses the “nice” system for mapping priorities into the user space, in combination with dynamic priorities (it takes account of how much the thread uses processor), the dynamic priority affects the length of the quantum. It is not possible for threads with the lowest priority not to receive a processor in the epoch.
- **SCHED_BATCH** – for batch threads (noninteractive, CPU-bound, often running in the background), the dynamic priority is used too, but in other way.
- **SCHED_FIFO** – for realtime processes (threads). This scheduler is nonpreemptive, it uses 99 levels of static priorities (1–99). Realtime processes are always preferred, even before the threads planned by another scheduler.
- **SCHED_RR** (Round Robin) – similar to the previous scheduler, but it schedules preemptively.

In the versions up to 2.6.22 there was only one – *O(1) scheduler* (with the $O(1)$ complexity, constant time scheduler), very efficient for massive amount of running threads.

In the newer versions of Linux kernel we use the *Completely Fair Scheduler* (CFS) with better interactive performance. Instead of queues, this scheduler uses a red-black tree. Its complexity is $O(\log N)$, where N is number of ready tasks.

The schedulers $O(1)$ and CFS use the sleeper-fairness principle: sleeping and waiting threads (mostly interactive) get a comparable share of processor time when they need it.

 **Remark**

“Big O” Notation represents the time complexity of an algorithm, it is the upper bound of the growth rate of a function and depicts the worst case scenario. Let n be the number of elements that the algorithm receives as its input (or the length of the input).

The linear complexity $O(n)$: the algorithm run length is linearly dependent on the value of n , i.e. the length of the input.

The logarithmic complexity $O(\log n)$ means a bit more optimal run length of the algorithm, especially for very long inputs, because logarithmic functions grow slower than linear functions.

Quadratic or even exponential functions grow much faster, so algorithms with $O(n^2)$ or $O(2^n)$ time complexity are considered time-consuming for large inputs.

The constant complexity $O(1)$ means that the run length of the given algorithm is not dependent on the input length.




Additional information


https://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html




Chapter 5

File Access and Permissions

 *Quick preview:* This short chapter is an insight into the principle of access permissions for users, groups and processes.

 *Keywords:* File access, permissions, owner, group, SUID, SGID, Sticky, su, sudo, attributes, POSIX ACL, PAM


 *Objectives:* The goal of this chapter is to introduce the concepts of file access permissions.

5.1 File Access Permissions in Linux

5.1.1 Owner and Associated Group

Each file has an owner and an associated group, and access permissions are set in three categories – permissions for the owner, for members of the associated group, and for others: “rest of the world”. It follows that the primary way to affect file access permissions is to specify the owner and the associated group.

The owner of a file is usually its creator or, in the case of system files, the root user. The associated group is a bit more complicated – when we create a new file (or directory, which is actually a file too), the associated group becomes the group we are currently working under (if we changed the active group with `newgrp`, it will be the one we set). If the parent directory has the SGID flag set, the new file inherits the associated group from its parent directory.

 **The `chown` command** is used to set the owner of the file (change owner). In the parameters of `chown` we specify the new owner and the file, or we can use different switches. Basic syntax:

```
chown user_new_owner /path/file
```

The change of ownership can also be recursive (we would use the `-R` or `-recursive` switch, usually the `-h` switch is added to avoid affecting the targets of symbolic links). Other switches would be found at the manual page of the command.

We can change the associated group together with the owner:

```
chown user:group /path/file
```

**Remark**

But the owner of a file can't be changed by just anyone – if we're working under a regular user account, we can't even give ownership of a file to someone else (transfer ownership), whereas root can change ownership of files at will.

Why? Because ownership implies a certain responsibility. If “harmful content” (defamatory statements or other ethically problematic content, pirated films, leaked secret information, etc.) is discovered during an audit, who will be held accountable? The owner.



The `chgrp` command sets the associated file group. The parameters and switches are similar to `chown`, including providing recursion. The basic syntax is

```
chgrp group /path/file
```

A regular user can use this command only on files that he/she owns, and can change their associated group to one of which he/she is a member. Root is not restricted in use of this command.

5.1.2 Setting File Access Permissions

Access permissions are represented either by a string consisting of the letters `r` (read), `w` (write), `x` (execute) and possibly dashes, or numerically.

For directories, the `r` permission means the ability to view the contents of this directory (either in text mode using e.g. `ls` or `echo`, or in the application in graphical mode), the `w` permission is the ability to edit the contents of the directory (especially to create or delete files inside), the `x` permission allows to use this directory as a working directory (e.g. by `cd`). If we want to get inside such a directory and work with something in its subtree, we also need the `x` permission.

**Example**

Several examples of access permissions string and number:

- “`rwxr-xr-x`” means that the owner can do anything with this file (`rwX`), the members of the associated group and others can only read and execute; the binary representation is

`111|101|101`, we encode it as

$$(1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0) \|(1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0) \|(1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0) = \\ = (4 + 2 + 1) \|(4 + 0 + 1) \|(4 + 0 + 1) = 755,$$

- “`rw-r-----`” means that the owner can read and write, the members of the associated group can read, others can nothing; in binary:

`110|100|000`, we encode it as the following mode:

$$(4 + 2 + 0) \|(4 + 0 + 0) \|(0 + 0 + 0) = 640,$$


- the mode `710` means that
 - the file owner has all access permissions set, $4 + 2 + 1 = 7$, `rwX`,
 - the associated group has only the right to run (it might be a program), so $0 + 0 + 1 = 1$, `--x`,
 - the others have no privileges ($0 + 0 + 0 = 0$, `---`),

the corresponding string is “`rwX--x---`”,

- the mode `754` means that
 - the file owner has all access permissions set, $4 + 2 + 1 = 7$, `rwX`,

- the group has only read and execute permissions (apparently it is a directory, so it is possible to set the directory as working), so $4 + 0 + 1 = 5$, **r-x**,
- the others have the permissions to read (display the directory contents, $4 + 0 + 0 = 4$, **r--**), the corresponding string is “**rwxr-xr--**”.




 When changing file access permissions, use one of these representation types:

- *numeric* – we specify the numeric form,
- *symbolic* – is mainly used for relative representation of changes (we want to add or subtract some permissions, but we don’t change other permissions), but there is also an option of absolute form (we change everything).

The Symbolic form consists of three defining parts:

- to whom permissions are to be changed – **u**=owner, **g**=group, **o**=others, **a**=all,
- whether some permissions should be added (+), subtract (-) or set (=),
- what permissions we mean – **r**, **w**, **x**.

The *absolute* form specifies a full change (we directly enter the resulting permissions), whereas the *relative* form means that we change only some permissions and do not override others.

 **The chmod command** is used to set the file mode, i.e. to affect file access permissions. Permissions can be specified numerically or symbolically, absolutely or relatively.

Example

We will show typical uses of **chmod**, starting with symbolic notation.

```
chmod g+w file    we have added privilege to write for members of the associated group
chmod -R g+w directory  the same, recursively
chmod ug+x,o-w file  the owner and the members of the group have been given the right to execute,
                    while the others have been deprived of the right to write
chmod g=r-x,o=r file  for members of the group, we set the string r-x, the others can only read
chmod u=rw,go-w file  group members and others will have their permissions to write removed (but
                    if they have had permission to read, they will continue to have it)
chmod a-x file    we remove the x permission for all (here we cancel the executability of a file)
```

Now let’s look at the numerical notation, which is in principle absolute:

```
chmod 754 file    the permission string for the file is set to rwxr-xr--
chmod 640 file    the permission string for the file is set to rw-r-----
```



5.1.3 Special Permissions

When something new is created (a process, a new file, etc.), its access permissions (for a process) or its access rights (for a new file) must also be established.


Usually, a newly created object inherits access permissions from its creator, so when a new file (including a directory) is created, the person who created the file becomes the owner, and when a process is created, its access permissions are also inherited from the user who started it. The same

is true for a newly created directory. However, the inheritance of these properties can be affected by special access permissions flags – *SUID* (SetUID), *SGID* (SetGID) and *Sticky*.

Any file or directory can have these flags set, although most do not. They are only relevant for directories and executable files.

The meaning of the SUID, SGID and Sticky bits is clearly explained in the following tables (SUID is used only for processes, Sticky for directories; SGID for both processes and directories).

First, let's look at the processes. When we run an executable, a process is created. A process accesses various objects (files, devices, etc.) and needs certain access permissions to access them, so we view it similarly to a user.

 The *effective user* of a process is the user whose permissions the process applies, *effective group* is the group whose permissions the process applies. Typically, the effective user is the user who started the process, the effective group is the user's primary group (or another group to which the user belongs if the user switched between groups). The SUID bit set on the executable may affect the effective user setting, the SGID bit may affect the effective group setting.

Executable file:	= 0	= 1
SUID =	When we run an executable, we become the effective user of the resulting process; the process takes over access permissions from whoever started it.	When we run a file, the effective user of the resulting process becomes the file owner; the process inherits access permissions from the file owner.
SGID =	The effective group of a process is the group of the user who started the process. The access permissions for the group are determined by the group which the running user belongs to.	The effective process group is the group set for the executable. The access permissions for the group are determined by the group set for the file.

Table 5.1: Special permissions for processes

Setting the SUID to 1 is used for programs that are normally run by a regular user, but that require higher access permissions to run.


Example

When changing the password, a regular user must run a process that accesses the `/etc/shadow` file, but the permissions on this file are as follows: `rw-r-----`, with the root user as the owner. This implies that no one but root has write access to this file, and neither does a process run by a regular user.

However, the executable that is used to change the password (`/usr/bin/passwd`) has the SUID bit set and its owner is root, so when run by a regular user, a file is created whose effective user is root \Rightarrow this process has permission to write to the `/etc/shadow` file and the regular user can change the password this way.

The GUI application is only the frontend (access interface) to the `/usr/bin/passwd` executable, so the same applies to it.



 In addition to `/usr/bin/passwd`, some other executables have the SUID bit set, such as `/bin/ping` (to verify the availability of devices on the network), `/bin/mount` (to mount storage media), `/usr/bin/sudo` (to run programs with higher privileges).


The SGID bit has for example the `/usr/bin/chage` command for setting user account parameters related to validity and password (e.g. you can deactivate the account, set the maximum password validity time, etc.), `/usr/bin/wall` for communication between users (they write “on the wall”), `/usr/bin/ssh-agent` (authentication agent) or some games.




Remark

SUID bit of an executable can be dangerous (actually also SGID) – executables owned by root and with the SUID bit set should be kept to a minimum, only in the most urgent cases.




 Now let’s look at the special flags for directories. When a new file (including a subdirectory) is created in a directory, we need to determine its owner, the assigned group and the access permission string. The owner is the person who creates the file, and the group of the user who creates the file is usually used as the associated group. The SGID bit of a directory affects the group assignment for files created within it.

 The sticky bit has an effect of a slightly different type. We usually need write permissions for the parent directory to delete a file (yes, we can delete a file that we have no permissions on, we just need the `w` permission on the parent directory), because it is actually a change operation on the contents of the parent directory. In some circumstances, this can be quite annoying – for example, if a group of people are working on the same project and have the project files in one shared directory (on a server, for example), then of course they will all need write access to that directory (so that they can add new files there if necessary). But then someone in the group could (either intentionally or unintentionally) delete a file created by someone else. This is wrong. However, if the parent directory has the Sticky bit set, the deletion authority is limited: only the owner of the deleted file (or root) can delete files. Adding new files is not a problem, but deleting is restricted, although the `w` permission for the parent directory would otherwise be sufficient for both.


Directory:	= 0	= 1
SGID =	When we create a new file in this directory, our group becomes the associated group.	When we create a new file in this directory, the associated group is inherited from the parent directory (the one with the SGID bit).
Sticky =	If the user has the “w” permission assigned to the directory, he can add and delete items in the directory. He/she can also modify an entry (change the contents of a file) if he has the “w” permission on that entry.	If a user has “w” assigned to a directory, he can add (and possibly modify) items, but only the owner of the parent directory or <i>root</i> can delete them.

Table 5.2: Special permissions for directory

 Setting special permissions will also be reflected in the access permissions string, in places where “x” is usually used for execution.

- The SUID bit is represented by “s” or “S” instead of “x” in the first third of the string,
- The SGID bit is represented by “s” or “S” instead of “x” in the second third of the string,
- The sticky bit is manifested by “t” or “T” instead of “x” in the third third of the string.

In all three cases, a lowercase letter means that the “x” right is also assigned, and a capital letter means that the “x” right is not assigned. In the case of the SUID and SGID bits, a capital letter is an error, because without execute permissions, these two bits are meaningless (so beware – if you see a capital “S” in the first or second third of the permission string, it indicates a problem).

 **Example**


Let’s look at a few permission strings with special flags set.


`rwsrwx---` SUID bit is set (the letter “s” in the first third), the owner and members of the associated group have read, write, and execute permissions \Rightarrow the process created from this executable takes access permissions from the owner of the file, not from whoever it was started by,

`rwSrwx----` similarly, but the execute permission is not set for the owner, so the SUID bit does not apply in practice (the correction of the error would consist either in removing the SUID bit or in assigning the execute permission to the owner),

`rwrxwsr--` SGID bit is set (the letter “s” in the second third), the owner and members of the group have the permission to read, write, and execute \Rightarrow if it is an executable file, the resulting process will have as its effective group the one assigned to the file, not the group of the executing user (it will inherit the group from the executable file); if it is a directory, then the items created in it will inherit the group from the parent directory, it will not get it from the creating user,

`rwrxwSr--` again a mistake, it is necessary to either clear the SGID bit or set the permissions to execute (for example if it was an executable file, it would not even be possible to execute it under the given group, let alone create a process with the permissions of the given group),

`rwrxwx--T` Sticky bit is set (the letter “T” at the end), the owner and group members have all permissions, the others have no permissions (not even execute) – this is not a problem here, the Sticky bit does not require execute permissions for “rest of the world” \Rightarrow if we want to delete files and subdirectories in this directory, we need to be their owners or have root privileges. 

 **Example**

Now that we know how to work with accounts, groups, and permissions, let’s use a cumulative example to show how to create a workspace for a group of employees collaborating on a project.

1. Log in as root or otherwise gain higher permissions.
2. We will create a new group to include employees working on the project:

```
groupadd new_group
```

3. The users `u1`, `u2`, `u3`, ... are to collaborate on the project, so we’ll put them into the new group:

```
usermod -aG new_group u1
usermod -aG new_group u2
```

```
...
```

4. Create a directory where the project files will be stored:

```
mkdir /vol/new_directory
```

5. Set the associated group and owner (group leader – user u1) for this directory, as well as access permissions:

```
chgrp new_group /vol/new_directory
chown u1 /vol/new_directory
```

6. We usually don't have to deal with access permissions much, the existing ones will probably be enough, just “rest of the world” disable access and set the sticky flag for safety:

```
chmod o-rwx /vol/new_directory
chmod +t /vol/new_directory
```



Tasks

1. Decode these permission strings:

Files:

- `rw-rw-r--`
- `rwxr-x---`
- `rwsr-xr-x`
- `rwxr-S---`

Directories:

- `rwxr-x---`
- `rwxrwS---`
- `rwxrwsr--`
- `rwxrwxr-t`

2. For the commands in the `/bin` directory, find out who the owner is, what the assigned group is, and what the access permissions are.
3. Find the `/usr/bin/passwd` file and see what permissions are set on it (including special permissions).



When converting the permission string to a number, we add one more digit (at the beginning) for the special bits. We handle the triple SUID + SGID + Sticky just like the triple `rwX`: the number 4 means SUID, 2 means SGID and 1 means Sticky. These numbers can also be combined. If the file has SGID and Sticky bits set, it will be $2+1=3$.



Example

The string `rwxr-xr-x` means the number 755. The string `rwsr-xr-x` means 4755, because the SUID bit is set.

The string `rwxr-sr--` corresponds to the number mode 2754, because the SGID bit is set.

The string `rwxr-sr-t` corresponds to the number mode 3755, because the SGID and Sticky bits are set. The string `rwxr-sr-T` corresponds to 3754, because the others do not have the “x” permission.



Example

We can also set or clear special flags for a file in a symbolic way:

```
chmod u+s file    the SUID flag has been set
```

```
chmod u-s file    the SUID flag has been unset
```

`chmod g+s file` the SGID flag has been set

`chmod +t file` the Sticky bit has been set (there is no letter before the “+” sign)

`chmod 4755 file` the permission string for the file has been set to `rwsr-xr-x` (we also set the SUID bit, which we can tell at a glance by the fact that the mode is four digits and the first digit is 4)

How to find all programs with the SUID bit set:

```
find / -perm /4000
```




5.2 Working under Different User Account in UNIX systems

When we need to increase access privileges in text mode, the following commands are available:

- `su`
- `sudo`

Each of them has slightly different advantages and disadvantages, and also the method of use.

5.2.1 The `su` Command

 **The `su`** (Substitute User) command is used to obtain privileges of another user. It is typically used to get root privileges (but we need to know the password).


The advantage is simplicity – we use this command to indicate that we want to use another user’s privileges, enter the password, and from that point on we become “other user”. We return to our original identity with `exit`.

Remark

Actually, with this command we start a new shell (a new process that will be a descendant of our original shell), the effective user of the new process is the user whose permissions we have obtained in this way. Therefore, all commands/programs/processes run in this shell are also executed with the permissions we have obtained. The `exit` command exits the new shell, the process with “other” permissions ceases to exist, and we return to the shell whose process has our original identity as the effective user. Alternatively, we can use the `logout` command.



If we enter the command without parameters, it means that we want to get root permissions (this is the most common case). Otherwise, we specify the login name of the user whose account we want to work under as a parameter.

 The default behavior of the command is that the UID we are working under is changed (i.e. the effective user of the shell process is changed) and actually the active group is changed to the primary group of that user, but all other settings remain “our” – home directory, environment variables (including the contents of `PATH`), etc. If we want to override these settings as well, we do so by specifying the `--login` or `-l` parameter, or simply by using the dash itself.

The following options are equivalent:

- `su --login user`
- `su -l user`
- `su - user`

So if we want to work as root, including the home directory and variables, and we don't want to wear out the keyboard unnecessarily, we type:

```
su -
```

(don't forget the hyphen).



Example

Suppose there is a user `anna` on the system. We are now logged in as `anna`. Let's look at this sequence of commands:

- `whoami`
output: `anna`
- `id`
output: `uid=1005(anna) gid=1005(anna) groups=1005(anna),...`
- `su -`
(we are asked for the root password, assuming we have it and have been granted higher privileges)
- `useradd -m beata`
`passwd beata`
(we have created a test account, set a password for it; we are asked for the password, then to repeat it; now there is an account `beata`, whose password we have also set to be functional)
- `exit`
(we left the root shell)
- `su beata`
(enter the password of the specified user)
- `whoami`
output: `beata`
- `id`
output: `uid=1008(beata) gid=1008(beata) groups=1008(beata)`
- `pstree`
part of the output will be:


```

      ... bash
         |
         | su
         | |
         | | bash
      
```
- `ps aux | grep beata`
output:


```

      root    ...  su - beata
      beata   ...  -su
      beata   ...  ps aux
      beata   ...  grep beata
      
```

 (it is strange that `ps` and `pstree` each interpret processes/tasks slightly differently)
- `exit`
`su -`
`userdel -fr beata`
`exit`




Remark

The `$` symbol usually appears at the end of a normal user's prompt. If you are working with root privileges (for example, with the `su` command), the `$` symbol will appear at the end of the prompt instead of `#`. Also, the color of the prompt will probably change to red (but it depends on the configuration).

This is so that a user who needs to work as root from time to time has an overview of what permissions he is currently working with, so that he does not accidentally start a process with higher permissions that he is unsure of (i.e. when there is a security risk).



5.2.2 The `sudo` Command

 **The `sudo` command** (SUperuser Do) is directly intended to temporarily gain higher privileges for a specific purpose. The default behavior of this command is that we usually directly specify the command we want to execute with higher privileges as a parameter.

Unlike the `su` command, this command is configurable, and a variety of other switches and parameters can be used, so we can specify in great detail what we actually want to do, whose permissions are to be used, and what specifically is to be overridden. Of course, we usually don't want anything too complicated.

Compared to `su`, there is one more significant difference from the user's point of view – while the `su` command asks for the password of the user we entered (i.e. typically the root password), the `sudo` command will ask for our own password (usually, depending on the configuration and what we actually want to run).

The configuration is stored in the `/etc/sudoers` file. Here it is specified who can use the `sudo` command and how, or who cannot use this mechanism.

 This follows from the principle of the *sudo mechanism*:


- It is specified in the `/etc/sudoers` file that if a particular user (for example, `anna`) using `sudo` wants to run the `xyz` program, it will be allowed, and the `xyz` program is run with the privileges of the `john`.
- So if the user `anna` enters the command

```
sudo xyz
```

must first properly prove that he is indeed `anna`, and then a new process from `xyz` is started as a subprocess of the original shell, with `john` as its effective user.

- The proof of identity is valid for a certain period of time (depending on the configuration, usually 5 to 15 minutes), so if `anna` uses the `sudo` mechanism again during this time, it is not forced to enter its password each time.

In practice, the `sudo` mechanism is used to run processes with root privileges, i.e. to *escalate*. So in the configuration file, we typically have specified that a particular user can run certain (in the simpler case, all) programs with root privileges (but they must always enter their password to prove that they came by the identity legally and are a “physical” user).

 As mentioned above, the `sudo` mechanism is configured in the `/etc/sudoers` file. It is a text file, but it is not a good idea to interfere with it in (any) text editor, because a possible error could be quite fatal. The editing is done in `/usr/sbin/visudo`, which is actually some other editor, but in addition, certain syntax elements specific to this configuration file are highlighted in colour. In most Linux distributions, `vi` is used as the base editor (which is not very convenient unless the user is used to certain special features of this program), while in other distributions (often based on Ubuntu) you will find the `nano` editor.

Of course not everyone can use `/usr/sbin/visudo`, for that we need higher access permissions.



Procedure

Let's take a closer look at the `/etc/sudoers` file. Assume we have sufficient privileges and have used the `/usr/sbin/visudo` command.

For root, there is usually an entry in `/etc/sudoers`

```
root ALL=(ALL) ALL
```

This configuration can also be used for other users, but only if we fully trust them. Typically it can be used for alternative administrators.

The general syntax is

```
some_user computer=(effective user) command
```

The first entry indicates the user for whom the row applies. If we want to specify the whole group instead of the user, then we write the symbol `%` before the group name to distinguish the user from the group, and we put the symbol `+` before the network group name.

This is followed by the computer on which the setting is valid (i.e. if the entry is in a file on another computer, it is not valid). This can be directly the computer name, IP address, (sub)network address, etc.

Inside parentheses is the name of the user account to which the user is temporarily logged in, and the last parameter is the list of commands that the user can execute.

Let's have a look to a few lines that may be in the `/etc/sudoers` file.

```
john johnscomputer.company.com = (ALL) /bin/kill,/bin/killall
```

the specified user can run commands on his computer (even remotely) to terminate other commands

```
john johnscomputer.company.com = (ALL) NOPASSWD: /bin/kill,/usr/bin/killall
```

the same, plus the root password will not be required

```
%wwwadmin www.webserver.cz = (www) /usr/local/apache/bin/apachectl
```

members of the specified group can run the specified command on the specified web server as the *www* user

```
john mailserver = /bin, !/bin/su
```

we allow the specified user on the mailserver to run anything from the `/bin` directory except the `su` command (here we see the negation notation), they have to authenticate themselves (they work under their own account, there is nothing in brackets)

```
fileadmin ALL, !mailserver = ALL
```

the specified user can run everything on all machines except the mailserver

```
ALL = (ALL) NOPASSWD: /sbin/shutdown
```

everyone everywhere can run the command that shuts down the computer, no password will be required



Example

If we are working in a system that uses the `sudo` mechanism, we precede each command that requires higher privileges with `sudo`. E.g. we want to create a new user and set a password for that user:

```
sudo useradd -m beata
```

```
sudo passwd beata
```


When entering the first command, we will be asked for (our) password, but not for the next command, the system will remember us for a certain number of minutes.

The sudo mechanism can also be used when previewing a password hash file:

```
sudo cat /etc/shadow
```



Procedure

The `su` and `sudo` commands can also be combined. If we specify

```
sudo su -
```

we are prompted for *our password* (this is provided by `sudo`), but at the same time a new shell is created with root privileges, the obtained root privileges are valid for the other commands listed (until `exit` or `logout` is entered). So we get the benefits of both commands.



Tasks

Create a new test user, set a password and any other properties:

```
sudo useradd -m test
```

```
sudo passwd test
```

Check the result:

- in the `/etc/passwd` file, check to see if a new entry has been added and what information is listed there,
- in the `/etc/group` file, also check if a group has been created for this user (assuming you have a system using UPG – User Private Group),
- using the sudo mechanism, check the `/etc/shadow` file for the password hash of the new user,
- in `/etc/login.defs` you can check what type of hash it is:

```
less /etc/login.defs
```

or

```
cat /etc/login.defs | grep -i ^encrypt
```

(don't forget “^” before the search string)


Then remove the test user:

```
sudo userdel -rf test
```



5.3 Advanced Access Control Mechanisms in Linux

5.3.1 Attributes

 **Basic Attributes in Filesystems.** In Windows we have attributes H (Hidden), S (System), D (Directory) and others, in UNIX file systems we use different attributes. We'll look at attributes in ext2, ext3 and ext4 file systems, but we also find support for attributes (in a slightly different way) in XFS and others. There are quite a lot of attributes, among others, such as

`i` (= immutable) the file cannot be modified, deleted, renamed nor linked to,

`a` the file can only be opened in *append* mode (i.e. we can only add data to the end, but not modify the original content), useful e.g. for important LOG files,

S (uppercase S) after any changes are made, immediate synchronization occurs, i.e. data is immediately written to disk,

A (uppercase A) means that the *atime* (access time) value will not be updated when the file is accessed, which can speed up work with frequently opened files and is useful for SSDs,

e means that the file is using extents (new in the ext4 filesystem) – protects large files to be fragmented (keeps the space occupied by the given file contiguous), etc.

 We work with attributes using the following commands:

lsattr

lists the set attributes for the specified file or multiple files (we use the file names as parameters of the command), we can also pass the list of files (or directories) to this command via a pipe (e.g. `ls -a | lsattr`)

chattr

is used to change file attributes, as a parameter we write the name of the file whose attributes we want to set, then the attribute and possibly other options (e.g. for recursive setting of attributes)



Example

The following command sets the “s” attribute for the specified file:

```
sarka@notebook:~$ chattr +s file.txt
```

```
sarka@notebook:~$ lsattr file.txt
s----- file.txt
```



Extended attributes. The extended attribute mechanism allows you to define custom attribute names (and of course their values). Thus, we are not restricted to a limited set of file system related attributes. Extended attributes are mainly used in the context of increased security, e.g. in the SELinux module, and they are also the basis of the POSIX ACL mechanism, which we will see below.

To work with extended attributes, use the commands `getfattr` and `setfattr`.



Example

The following command lists extended attributes of all hidden files:

```
lsattr .*
```




5.3.2 POSIX ACLs

The POSIX standard also defines security *access control lists*, ACLs. The ACLs themselves are stored as file system metadata in the form of *extended attributes*. Similar to ACLs in Windows or network devices, the idea here is to define access permissions for named users or group members (a finer division than owner-group-others). The principle of `rwX` access permissions is basically preserved, but we have more options for assigning them.

There are three *types of ACLs*:

- ACLs for users,
- ACLs for groups,
- ACL for others

 The main commands used for working with POSIX ACLs are `getfacl` and `setfacl`, which are used to display and modify ACLs. Basic syntax:

```
getfacl file    list of ACLs for the given file
setfacl -m u:some_user:some_permissions file    adds a user ACL entry for the file, e.g.
    setfacl -m u:john:rw,charlie:r,anna:rwx file
setfacl -m g:some_group:some_permissions file    similar for a group
setfacl -x u:some_user file    deleting a user ACL entry for the given file
```



Example

```
sarka@notebook:~$ getfacl ~/.bashrc
```

```
# file: .bashrc
# owner: sarka
# group: users
user::rw-
group::r--
other::r--
```

If an ACL is applied to the specified file, for example, the user `boss` should be explicitly set to `rw-`, then there would be one extra line:

```
user:boss:rw-
```

We can change ACLs as follows:

```
setfacl -m u:boss:rw-,g::rw-,g:apache:rwx,o:--- file
```

We have set read and write permissions for one user, the file group as well, the group created for the apache web server we have enabled everything (be careful if this is really necessary) and we have disabled everything for the rest of the world. We can delete most of the hyphens in the permission strings:

```
setfacl -m u:boss:rw,g::rw,g:apache:rwx,o:- file
```



Example

Again, one summary example: we create a new test user, set a password for him, and create a new file in his home directory. Then we add a special ACL entry for ourselves to allow ourselves to read and write to this file, even though we didn't have these permissions originally. Finally, we return to our shell and try to write to the new file.

- First the preparatory operations:


```
sudo su -
useradd -m janedoe
passwd janedoe
getfacl -m u:sarka:x /home/janedoe    (why do you think this will be beneficial to me?)
exit
```
- We create a new file and modify its ACL:


```
sudo su - janedoe
touch test.txt
getfacl test.txt    (we have listed the ACL of our file)
setfacl -m u:sarka:rw test.txt
```

```
getfacl test.txt
exit
```

- Now let's try access under the own account:

```
echo "some interesting string" > /home/janedoe/test.txt ; cat /home/janedoe/test.txt
```

- Cleaning:


```
sudo su -
userdel -fr janedoe
exit
```



5.3.3 PAM


PAM (Pluggable Authentication Modules) is a mechanism for extending security modules used on Linux, FreeBSD, Solaris and some other systems. With PAM, we can, e.g., allow logins only at a certain time and only from a certain location (terminal), set limits on resources (number of running processes per session, memory consumption, etc., so we can prevent DoS attacks to some level), choose different password encryption methods, etc.

It is a system based on a stack of modules (`pam_stack`), where the stack is traversed from top to bottom during user authentication and each module must “pass”, i.e. each module in the stack must agree to authentication. We can imagine, e.g., that the system puts certain rules in the stack (as a LIFO structure), and the process to be authenticated removes individual rules from the stack one by one by satisfying them. If it fails to satisfy a rule, it is not possible to get to the next rule. Successful authentication corresponds to an empty stack.


 The PAM mechanism is used by various services that users authenticate to (including, but not limited to, the user login service or some daemons to access their configuration). If a service wants to use PAM, it creates its own stack of modules, which it specifies in a file in the `/etc/pam.d` directory. In general, we need to create our own file in this directory and specify our stack in it. If the service does not want to create its own file, it can use the default one `/etc/pam.d/other`.

A service can create its stack from modules stored in files with the `.so` extension (i.e. libraries). There are quite a lot of modules, e.g.:

- `pam_access` – it is possible to limit the locations from which a user (or group of users) can log in (also applies to remote logins, IP addresses can also be specified)
- `pam_time` – login time can be limited
- `pam_securetty` – limiting the possibility of root login from specified devices (we define “secure” terminals for root login)
- `pam_cracklib` – checking passwords to make sure they are not easily cracked by a dictionary attack
- `pam_pwhistory` – password history is stored, which can prevent users (who are forced to change their passwords occasionally) from, for example, cyclically choosing alternating passwords
- `pam_tally` – this module monitors the number of unsuccessful logins that follow directly after each other; we set a threshold (e.g. 3) and after crossing this threshold we can conclude that it is a dictionary attack on the password, so we block the account
- `pam_userdb` – Berkeley DB database is used for authentication

 We can get the list of modules from the manual pages:

```
apropos -s 8 pam
```

 Each PAM module must be configured somehow. The configuration is usually stored in the `/etc/security/name.conf` files, e.g. the configuration of the `pam/access` module (limiting the places from which it can log in) is in the `/etc/security/access.conf` file. These files are usually well commented, so if we want to change the configuration, it is usually not a problem.

Example

However, the specific configuration can also be listed directly next to the module name. E.g. we add a line into `/etc/pam.d/login`

```
account    required    /lib/security/pam_tally.so deny=3 no_magic_root
```

This means that after three login failures (`deny=3`) the account will be locked, and this also applies to root logins. There are also other parameters that we would find in the man page (`man pam_tally`, e.g. the length of time the account should be locked).



5.4 Security Policy Settings in Windows

In Windows (except Home edition) we have a security policy settings for this purpose. We can access them, for example, by running the `secpol.msc` console file.

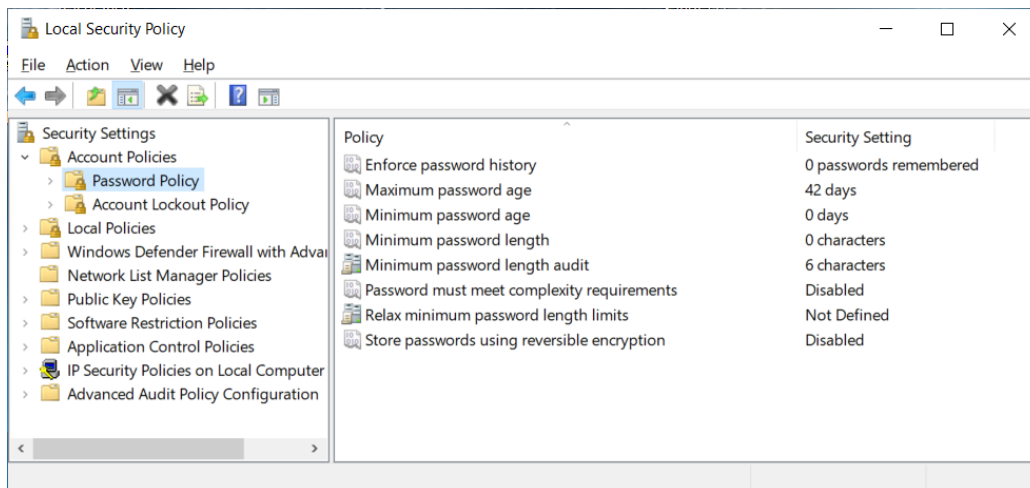


Figure 5.1: Local Security Policy in Windows

In all editions we can use the `net user`, `net localgroup` and `net accounts` commands.

Example

Examples of usage `NET USER`:

```
net user    lists the users (in several columns)
```

```
net user janedoe    lists detailed information about the user janedoe, including the groups they belong to, password information (whether the user can change the password, when it was set, when it expires, etc.), when the user last logged in, whether the account is active, etc.
```

```

net user janedoe password    sets the password for janedoe
net user janedoe *           like the previous one, but we are asked for the password, no characters are
                             displayed when entering it (in case the colleague can see our screen)
net user janedoe /active:no   we deactivate the given account
net user janedoe /active:yes  we activate the inactive user account, it also works with the admin-
                             istrator account (in newer Windows it is usually deactivated for security reasons and the admin
                             cannot log in this way)
net user janedoe /passwordchg:no  from now on, the specified user has no possibility to change his
                             password (this setting is typical for the guest account)
net user janedoe password /add /fullname:"Jane Doe"
                             creates a new user with the specified login name, heslem a zobrazovaným jménem
net user janedoe /delete      removes the given user
net user janedoe /times:M-F,6-16  allows a given user to be logged in only on the specified days
                             and hours (here on workdays between 6am and 4pm; the days are M, T, W, Th, F, Sa, and Su)

```



Usage of the `net localgroup` is similar, for local groups.

We can use cmdlets in the PowerShell as well:

Example

How to find cmdlets in Powershell for working with users and groups? For users:


```
PS C:\Users\user> get-command *user*
```

CommandType	Name	Version	Source
Cmdlet	Disable-LocalUser	1.0.0.0	Microsoft.PowerShell.LocalAccounts
Cmdlet	Enable-LocalUser	1.0.0.0	Microsoft.PowerShell.LocalAccounts
Cmdlet	Get-LocalUser	1.0.0.0	Microsoft.PowerShell.LocalAccounts
Cmdlet	New-LocalUser	1.0.0.0	Microsoft.PowerShell.LocalAccounts
Cmdlet	Remove-LocalUser	1.0.0.0	Microsoft.PowerShell.LocalAccounts
Cmdlet	Rename-LocalUser	1.0.0.0	Microsoft.PowerShell.LocalAccounts
Cmdlet	Set-LocalUser	1.0.0.0	Microsoft.PowerShell.LocalAccounts
Application	quser.exe	6.1.760...	C:\Windows\system32\quser.exe
Application	UserAccountControlSettings.exe	6.1.760...	C:\Windows\system32\UserAccountControl...
Application	userinit.exe	6.1.760...	C:\Windows\system32\userinit.exe

So, for example, cmdlet `get-localuser` lists all local users. For more detailed help on the command, including parameters, we would use the cmdlet `get-help -detailed get-localuser`.

Similar for local groups: `get-command *group*`.



 To work with password security policies, we use the `NET ACCOUNTS` command. This is a security policy that applies to all accounts.

Examples of using the `NET ACCOUNTS` command:

```


net accounts    lists the currently defined properties of user accounts
net accounts /minpwlen:8  sets the minimum required length of the user's password to 8 characters
                             (the "pw" abbreviation appears in several parameters, it means that they are related to the
                             password setting)

```

`net accounts /maxpwage:120` the password is always valid for a maximum of 120 days, after which the user must choose a new password

`net accounts /maxpwage:unlimited` the user is not forced to change his password regularly, its temporal validity is not practically limited

`net accounts /maxpwage:30 /uniquepw:6` the password is always valid for a maximum of 30 days, the user (with this relatively restrictive period) can choose a password that he already had, but only after at least 6 password changes

 We also have *ACLs* in Windows (although they are not POSIX ACLs). We can access them through the context menu of the file, as seen in Figure .

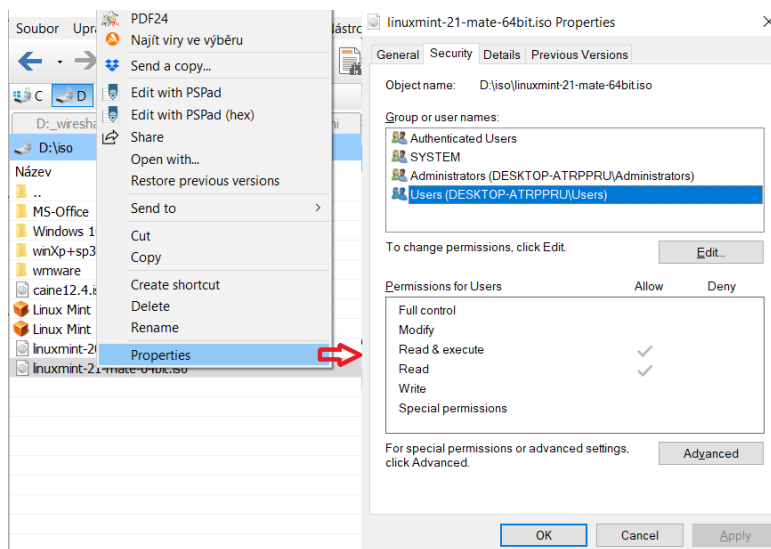


Figure 5.2: Windows ACL for files

Another possibility is via Powershell.



Example

We use cmdlet `get-acl`:

```
PS C:\Users\user> get-acl p:\order_cartridge.pdf | fl
```


```
Path      : Microsoft.PowerShell.Core\FileSystem::P:\order_cartridge.pdf
Owner     : 0:S-1-5-21-1939040898-431487083-1465283063-1000
Group     : G:S-1-5-21-1939040898-431487083-1465283063-513
Access    : BUILTIN\Administrators Allow FullControl
           NT AUTHORITY\SYSTEM Allow FullControl
           NT AUTHORITY\Authenticated Users Allow Modify, Synchronize
           BUILTIN\Users Allow ReadAndExecute, Synchronize
Audit     :
Sddl      : 0:S-1-5-21-1939040898-431487083-1465283063-1000 (shorten)
```


To change the ACL of a file we would use the cmdlet `set-acl`, or the `cacls.exe` and `icacls.exe` for the Command Line are available.




Chapter 6

Synchronization

 **Quick preview:** In a multitasking system, it is common for multiple processes to need to access the same resource. This resource can be a common I/O device such as a screen, keyboard, or printer, but it can also be a shared memory area, variable, or a file. In this chapter, we describe the basic problems associated with process synchronization, and the methods that can be used to solve them.

 **Keywords:** Synchronization, consistent state, Petri net, critical section, producer-consumer, model-view, readers-writers, concurrent processes, raice-condition, passive and active waiting, Bakery algorithm, semaphore, mutex, messages, IRQL, spinlock

 **Objectives:** The goal of this chapter is to learn how to synchronize process or thread access to shared resources.

6.1 Why Synchronize


When multiple processes access the same resource, the main issue is ensuring *consistent resource state*. In the case of shared memory, it is about data consistency, i.e. if a process writes to this memory, another process should not read until the writing process finishes its work, because it could only read partially modified data. The data is in a consistent state before the write starts and after the write completes.

Resources that are not in a consistent state must be protected, so access to them is synchronized.

We synchronize access to resources:

- among kernel threads (sharing hardware resources and kernel structures),
- among processes sharing resources,
- among threads of the same process (sharing common variables, open files, pipes, etc.).

As in the whole of this chapter, we will talk about processes here, but in fact the issue also concerns the synchronization of the threads within a process, or kernel.

 A *race condition* is such situation that can occur when processes running in parallel request access to a same object. Although the processes store the contents of the registers independently (when context is switched, the contents of the registers are stored in order not to be overwritten by another

process), but it does not apply to shared variables. If manipulation of the variable is not atomic (i.e., it is longer than one instruction and can be interrupted), inconsistency may occur.

The system will either deal with it or it won't. In the worst case, the object may be assigned to both processes or neither, in the better case, the system decides which process takes precedence.

6.2 Petri Nets

Petri nets are a visualization tool that clearly captures data flow or any parallel or pseudo-parallel processes at an abstract level. It is a mathematical modelling language for describing distributed or other systems working in parallel, using a graphical notation.

In this document, we use Petri nets for simulation of synchronization problems.

 *Petri net* is an oriented graph with two types of nodes:

—○— *places*, representing process states or system states,

—|— *transitions*, representing a certain activity of a process or system between two states (places).

Places and transitions alternate in the graph, there should not be two nodes of the same type directly behind each other. In places, there may be *tokens* (dots) representing permissions to continue further in the graph. Each edge (arc) is labeled by a natural number (if no number is given, it is 1), this number means the multiplicity of the edge.

Each transition has one or more *source places*, from which an edge leads to this transition. For the transition to be feasible for execution, in each source place there must be at least as many tokens as the multiplicity of the edge (number in label of the edge).

Each transition has one or more *target places*, to which an edge leads from this transition.

The transition is executed as follows:

- 1) for each source place: if the edge labeled by n leads from the source place into the transition, we take n tokens from the place.
- 2) for each target place: if the edge labeled by m leads from the transition into the target place, we give m tokens to the place.

Example

There are two transitions in Figure 6.1, but only the first one is feasible, and the second one is not feasible because there is no token in **C** and only one token in **B**, we need two. When the first transition is executed, one token (only one, the edge is not numbered) is removed from the place **A**, one token is put to **B** and one token is put to **C**.

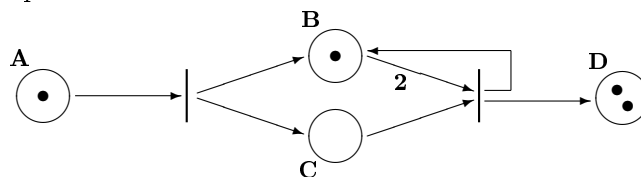


Figure 6.1: Example of Petri net

Now the second transition is feasible. When executing it, two tokens are removed from **B**, one token is removed from **C**, one token is added to **B** and one to **D**. There are three tokens in **D**.



**Example**

Figure 6.2 shows a Petri net describing the simplified run of a process using the processor with no other process running.

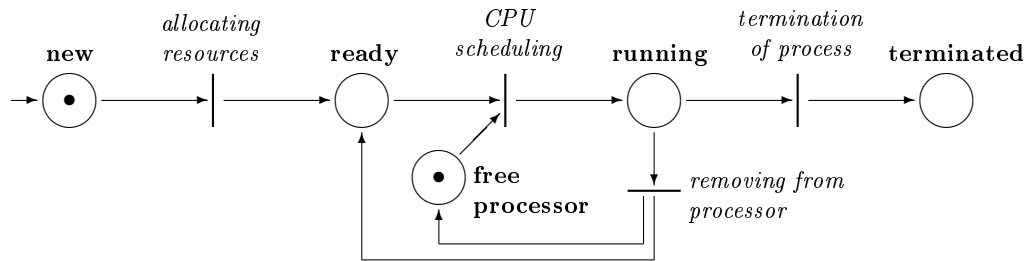


Figure 6.2: Petri net describing states of a simple process

The Token in the place *new* can be understood as the current state of the process. All transitions are labeled by 1 (number 1 does not need to be listed).

The place *free processor* represents the state of the system in which a processor can be assigned. It contains a period only when the processor is free and can be assigned. During executing the transition *removing from processor*, we remove one token from *ready* and put one token to *free processor* and one token to the place *running*.

If multiple processes are run, all of these processes use the place *free processor* (their own states would be paths parallel to the process displayed). Whenever a process gets into a running state, it removes a token from that place and the other processes have to wait in the waiting state, until the running process returns the token.



We will use simplified Petri nets, where a single place or transition can represent an entire subnetwork whose parts do not need to be distinguished.

6.3 Basic Synchronization Tasks

We will discuss the basic problems that are solved in process synchronization and outline their solution at the abstract level using petri nets. In current operating systems, threads are also synchronized, although we will use the term process for generality.

6.3.1 Critical Section

A critical section is a code working with any shared object, data structure, communication interface, device, piece of memory or another resource, and the object is shared by n processes $\{P_0, P_1, \dots, P_{n-1}\}$. The structure of code inside each of the n processes is the following:

...

Entry section	Critical section	Exit section	Remainder section
---------------	------------------	--------------	-------------------

 ...

The code of all n processes coincides in the critical section, this code works with a shared object. Another way to illustrate this situation is the Petri net in Figure 6.3.

In order for a process to execute its part of the code accessing a critical section, there must be a token in the guard place. In our case, process P_2 comes to the transition to access the critical section,

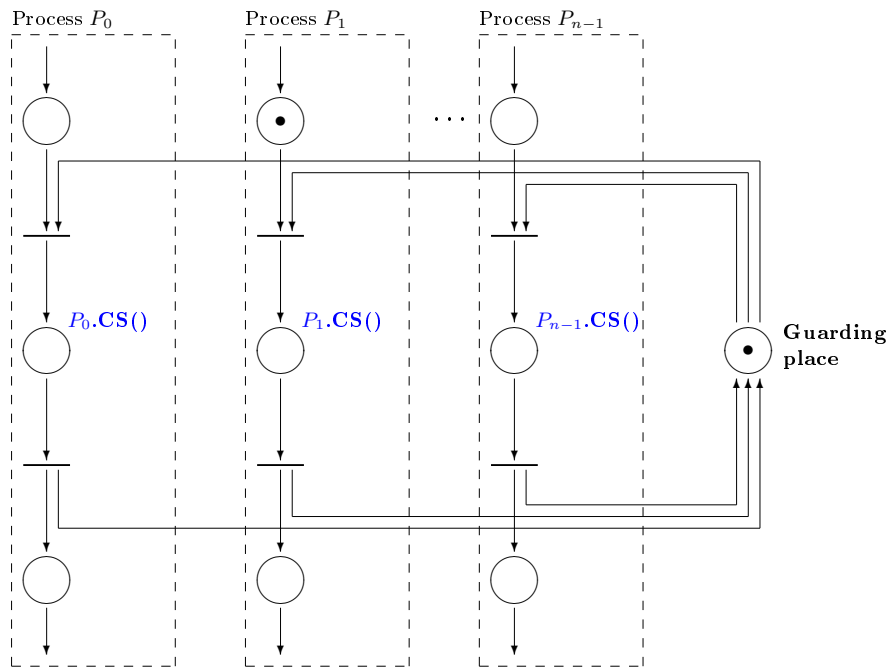



Figure 6.3: Petri net for the Critical section Problem

and since there is a token in the guard place, this means that no other process is currently accessing the shared resource and therefore P2 can continue.

After the evaluation of the input transition, the token is removed not only from the process state of the process location before the critical section, but also from the guard place, and added to the place inside the critical section evaluation by process P2. Then the process is in the critical section evaluation state, at location P2.KS(). After a transition is performed that means leaving the critical section, the token is returned to the guard place and also added to the place of process P2 after the critical section, i.e. the process continues its activity, and another process can enter the critical section.

If another process wanted to enter the critical section when process P2 is in the critical section (and thus there is no token in the guard place), it must wait until process P2 returns the token to the guard place before it can proceed.

 The requirements for the solution are as follows:

- the data must be in a consistent state if the process expects it,
- no more than one process is allowed in the critical section (*mutual exclusion*),
- the process is in the critical section for the finite amount of time,
- the process waits a finite amount of time to enter the critical section (depends on the previous one: *bounded waiting*).

This task is the basis for other tasks, essentially always about – how to ensure consistency of data accessed by multiple various processes or threads.

 **Remark**


How do we get it programmed? For example, the guard place is represented by an integer variable, the number of tokens in corresponds to the value of this variable. In the simplest case, each process would test the variable in a loop before entering this section (as long as its value is 0, the loop continues), and when the loop completes (greater than 0, i.e. at least one token in the guard place), it would decrease

the value of the variable by 1, execute the code of the critical section, and then increase the value of the variable by 1.

This would only work if the processes trust each other (which is not quite common, perhaps only between threads of the same process). Moreover, there could be a problem on a system with a multi-core processor, because then there could be a situation where two processes (threads) running in parallel try to “collect” the last token at the same time, i.e. decrement the variable to 0. We will discuss later how to solve these problems.




6.3.2 Producer-Consumer Problem

 A *producer* is a process producing data (e.g. obtaining them from sensors) and a *consumer* is a process that accepts this data and executes them or does something with them (e.g. displays). The purpose for producer(s) and consumer(s) is to work more or less independently, usually *asynchronously*, in different speeds.

A case with one producer and one consumer will be described, but this case can be extended to practically any number of producers and consumers.

The task can be solved in several ways, depending on how much shared memory is available to all participating processes:

- 1) Unlimited buffer – we have at our disposal any amount of memory (dynamic data structure).
- 2) Limited buffer – we have a limited number of memory locations available, the upper boundary is set (static data structure).

 **ad. 1) Unlimited buffer.** The solution with using the Petri net is in Figure 6.4. We have a queue whose length changes dynamically, the requirement is to ensure that the consumer stops when the queue is empty, and that the data remains consistent. In Figure 6.4 we can see a system where there are four entries in the queue and the *write* and *read* transitions are active (so both processes can work).

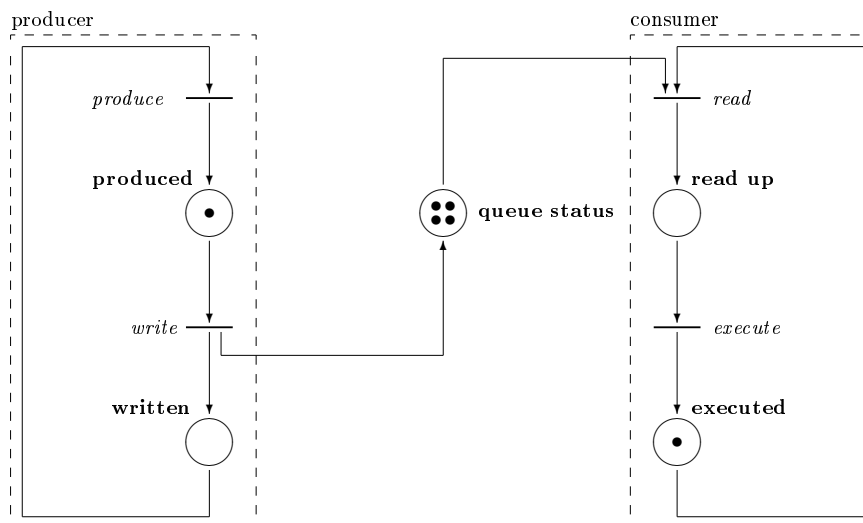



Figure 6.4: Petri net for the Producer-consumer problem, unlimited buffer

How would we extend the task to more than one producer and/or consumer? We would simply link other producers and consumers to the *queue status* place in the same way as the existing ones.

 The requirements for the solution are as follows:

- the data must be in a consistent state whenever any process accesses data (for the both reading and writing),
- consumers can only read from non-empty buffer,
- *bounded waiting* as for the critical section problem.




Remark

How to get it programed? We need an integer variable representing the location of *queue status*, then the queue itself (like a dynamic list, depending on what the programming language offers us).

The producer *first* saves the new element in the queue and *then* increases the value of the variable (the order of operations is important to maintain data consistency, especially if the queue was empty before loading).

The consumer *first* collects (retrieves) the element from the queue and *then* decreases the value of the variable (order is important for the same reason – preserving data consistency).



 **ad. 2) Limited buffer.** The solution is in Figure 6.5. The limited buffer can be implemented as a static round robin queue.

In Figure 6.5, only the *write* transition can be executed, the consumer must wait before the transition *read* (nothing to read, the queue is empty). After executing the *write* transition, the transitions *produce* and *read* are executable. The total number of queued places is the sum of the tokens in the *free*, *taken*, *produced* and *read up* places.

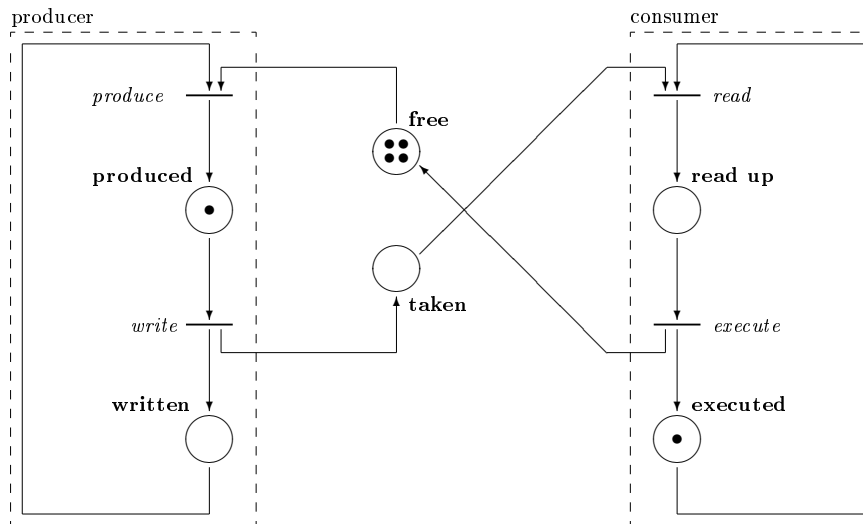



Figure 6.5: Petri net for the Producer-consumer problem, unlimited buffer

 The requirements for the solution are as follows:

- the data must be in a consistent state whenever any process accesses data (for the both reading and writing),
- consumers can read from non-empty buffer only,
- producers can write to non-full buffer only,
- *bounded waiting* as for the critical section problem.

**Remark**

The way of programming would be similar, we just need two variables, one for each shared place, and then a static queue (array, etc.). The order of operations (storing or removing an element and working with variables) are similar to the previous case, to preserve the data consistency in the given queue element (first or last).



6.4 Synchronization Tools

6.4.1 Waiting

A hardware assistance for synchronization is in supporting locking instructions. A critical section can be “locked” by entering process, and no other process can go inside.

Simple locking can be implemented in software, with a locking variable: A variable `locked` can be set to 0 (false, free) or 1 (true, taken). If set to 1, a process performs an empty loop.

**Example**

Simple implementation of shared locking variable:

```
shared int locked = 0;
// inside the process code:
while (locked) {}; // waiting with empty loop
// waiting has finished:
locked = 1; // we reserve the protected object
CS(); // we use the protected object (critical section)
locked = 0; // we released the protected object, leaving the critical section
```



This mechanism may fail on a multi-processor or multi-core processor system – there may be situations where two parallel running processes at the same time “find out” that the critical section is free, and at the same time attempt to change the variable `locked`.

The hardware solution is in using locking instructions for active waiting: TSL (Test and Set Lock), `swap` or `XCHG` – for various hardware architectures.

All these instructions perform a value exchange of the locking variable and the given variable in some way. They are used to permanently set the lock to 1 (locked) and to find out the original lock value before this setting. They can be used as part of a more complex active waiting tool.

**Example**

Let us show usage of the `XCHG` instruction (for Intel and AMD processors). The “`slock`” variable is shared locking variable; when `slock == 1` the critical section (CS) is locked.

```
CS: mov EAX, 1h // we store 1 into the EAX register
    xchg slock, EAX // we call swapping instruction (it swaps the given values)
    jnz CS // loop -- Jump if Not Zero (until EAX reaches 0)

// if there is 0 in slock before swapping, CS is free, and performing the instruction
// causes locking (the unlocked CS is immediately locked and we can follow in processing
// into the CS)
... // some code inside CS
leaving: mov slock, 0h // we have to unlock CS when leaving it
```



**Remark**

Hardware synchronization can also be implemented as an interrupt masking (if a common IRQ is disabled, then the process that is assigned to the processor will not be interrupted), but it is not usable for multi-processor and multi-core systems, or when using it inside kernel, the kernel should be single-threaded.



Operating systems offer functions encapsulating a similar instruction, because, in current operating systems, a programmer has only limited access to processor.

**Example**

When programming in various languages, we can encounter the system call `swap`:

```
shared int slock = 0;
...
// inside the process code:
int checking;                                // checking variable for swapping
...
checking = 1;
while (checking == 1)                          // or simply   while (checking)
    swap (slock, checking);

// if checking is 0, end of waiting:
...                                           // do something inside the critical section
slock = 0;
```



Note that these methods do not solve the race condition problem that occurs when processes (process threads) run in parallel on different processors or processor cores, because multiple processes can end the waiting loop at the same time.



Lamport's Bakery Algorithm. Each process receives an order number when requesting access to the critical section.

The process with the lowest order number has the highest priority, and if multiple processes have the same order number, the scheduler decides between them according to another criterion (PID or comparison of process names by alphabet). The algorithm works as follows:

1. The `ordnum` array: each process waiting for the given resource has an order number assigned (if the resource is not requested there is a number 0),
2. The `assigns` array: each process has 1 here if the order assignment is in progress for this process; after the assignment this value is returned back to 0. Only the given process (thread) changes this value.

The order number is not consistent during the order assignment, so this value is protected in this way (waiting `while(assigns[j])`).

3. The waiting process then passes the `ordnum` array, and while it encounters a process whose order number is smaller (or the same but has a lower PID), then waits in the empty loop until the process finishes waiting and executes its part of the code in critical section.

When the process passes through the whole array, then no process has a lower order number (none of them are waiting for this resource with better precedence).

**Example**

The Bakery algorithm has the following implementation:

```
shared int ordnum[n] = (0,0,...,0);
shared int assigns[n] = (0,0,...,0);
...
// while assigning order number, the field with ordinal number is protected
assigns[i] = 1;
ordnum[i] = GetHighest(ordnum) + 1; // obtaining order number
assigns[i] = 0; // end of assigning order number, end of protection

for (j=0; j<n; j++) { // discovering processes with higher priority
    while (assigns[j]) {}; // j-th process is obtaining order number - during protection
    while ((!ordnum[j]) && // j-th process wants the same resource
        ((ordnum[j]<ordnum[i]) || // j-th process has higher priority
         (ordnum[j]==ordnum[i]&&(j<i)))) {}; // the same order number but smaller PID
}

CS();
ordnum[i] = 0; // we no longer need the given resource
```



The Bakery algorithm is applicable to multiprocessor systems. It is a unique and simple algorithm that avoids aging of processes.

6.4.2 Mutexes and Semaphores



A *mutex* (short for “mutual exclusion”) or *spinlock* is a simple locking mechanism implemented in software. There are two functions:

- **acquire()** called when entering the critical section, for testing if the critical section is free (if not, the process is blocked by this function until the critical section is free), then the resource is locked out of this process,
- **release()** called when leaving the critical section, unlocking the shared resource.

These two functions are usually implemented using the above mentioned hardware mechanisms.

**Example**


We can use mutex in this way:

```
do {
    ...
    acquire(M);
    CS();
    release(M);
    ...
} while(true);
```



A *semaphore* for processes has similar meaning to a semaphore for cars. It has “free” (green) and “taken” (red) states, as a result of the “taken” state means suspending (blocking) the calling process. The value of 0 means “red”. A semaphore consists of an integer value (as a semaphore state) and from a queue with waiting processes.

```
typedef struct {
    int value;
    struct pcb *proc; // queue of waiting processes
} TSemaphore;
```


 There are two basic types of semaphores:

- *binary semaphore* – its internal variable has only one of two states: 0 or 1,
- *counting semaphore* – its internal variable has a range as required.

The binary version is similar to mutexes.

As in mutexes, a process calls two functions:

- `wait()` (or `down()`, or `lock()`, according to platform) – a process calls this function when entering the critical section,
- `signal()` (or `up()`, or `unlock()`) – a process calls this function when leaving the critical section.

```
...
wait (semaphore);
CS();
signal (semaphore);
...
```

The functions `wait` and `signal` should be atomic (uninterruptible). The `wait` and `signal` functionality can be easily set up on a single processor system (for example, by disabling interrupts while executing the function code), but this simple method can not be used in a multiprocessor system. Usually, this problem is solved by identifying these functions themselves for critical sections and their software solutions.

Example

Let us show using the counting semaphores to solve the synchronization problem of a Producer-Consumer with a limited buffer. We need two semaphores:

```
extern struct TSemaphore s_free, s_taken;
```

Meaning of the given semaphores:

- `s_free` prohibits the producer to exceed the buffer size, we initiate it for the number of positions in the protected array (shared memory),
- `s_taken` prohibits the consumer to read from empty buffer, we set its initial value to 0.

Producer:

```
do {
    produce (data);
    // wait for a free slot:
    wait (s_free);
    write (data);
    // inc. # of taken positions:
    signal (s_taken);
} while (1);
```

Consumer:


```
do {
    // wait if queue is empty:
    wait (s_taken);
    read (data);
    // inc. # of free positions:
    signal (s_free);
    execute (data);
} while (1);
```



6.4.3 Messages

Processes can also be synchronized by a messaging mechanism. By this term we mean not only direct messaging (`send` and `receive`), but also indirect communication by sending messages through ports (sockets).

The method is also suitable for multiprocessor or distributed environments, including synchronization within a computer network. The addressing of communicating processes or objects must be adapted to this.

 Processes work with messages using functions (system calls) usually called **send** and **receive**. When using direct addressing, communication works as described above (section 4.7, page 51), with indirect addressing these functions work as follows:

- the **send** function checks if the mailbox is full; if it is not full, the sending process sends a message, otherwise the sending process is blocked and the message is sent only after it is unblocked (when there is a free slot in the mailbox),
- the **receive** function called by the recipient checks if there is something in the mailbox; if it is not empty, the process accepts the message, otherwise the process is blocked until any message is delivered to the mailbox.

Example

The following code is a solution to the Producer–Consumer task for a fixed length buffer. Two *mailboxes* are defined:

- **s_free** – when the producer picks up a message from this mailbox, he can produce (it is initially filled with messages informing about the possibility to produce), the consumer sends a message here after each item processing,
- **s_taken** – the producer sends messages with the produced items to this mailbox, the consumer picks them up.

```
#define MAX 20 // the maximum number messages in mailboxes
struct TMessageQueue s_free, s_taken; // two mailboxes
for (i=0; i<MAX; i++) // initialization, prepayment
    send (s_free, NULL);
```

Producer:


```
do {
    receive (s_free, data);
    produce (data);
    send (s_taken, data);
} while (1);
```

Consumer:


```
do {
    receive (s_taken, data);
    process (data);
    send (s_free, NULL);
} while (1);
```



6.4.4 Monitors

 The monitor is an object-type synchronization resource. It encapsulates a group of data structures (variables), processes can access them only through interfaces defined by access functions (methods). Functions can be any number and determine different ways of working with monitor data.

Data structures encapsulated in the monitor are referred to as *conditions*. These conditions can be implemented using semaphores, and the semaphore operations **wait** and **signal** are used by monitor access functions (not processes), each access function can use one or more various conditions.

 The idea is that each condition can only be used by one function at a time. Therefore, the accessor function will immediately call the **wait** function for each condition it will use, thus preventing any other

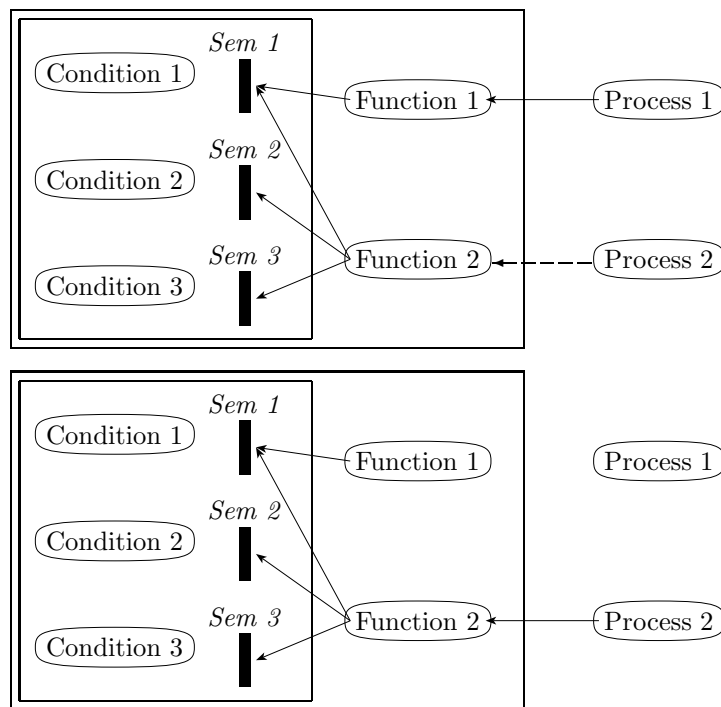


Figure 6.6: Simple monitor with two access functions

function that would also like to use that condition from running. Just before terminating, the function then unblocks the reserved conditions again with their `signal` functions.

In Figure 6.6 we can see a simple monitor where the first process calls the function 1. This function locks (sets to red) one of the semaphores, and thus prevents the second function (called by the second process) from being called. The second function waits until this semaphore is unlocked before it can execute its code (it needs to lock all three semaphores for itself to work), which we can see in the figure below.

6.5 Additional Synchronization Problems

Model-View. This problem is similar to the Producer-Consumer problem. We solve it when it is necessary to track and execute not all items but always the current state of the tracked quantity.

Typical usage is e.g. when a producer monitors the status of a sensor (temperature, humidity, etc.), stores the detected value in a shared variable (only one, no queue), and the consumer reads the value of this variable at regular intervals, e.g. for the purpose of displaying or triggering the alarm.

The requirements for the solution are as follows:

- the data must be in a consistent state whenever any process accesses data (for the both reading and writing),
- *bounded waiting* as for the critical section problem.

Readers-Writers. In this task, processes are divided into two groups – the group of *readers* and the group of *writers*. A process can belong into the both groups. We need to know how many readers (reading processes) are.

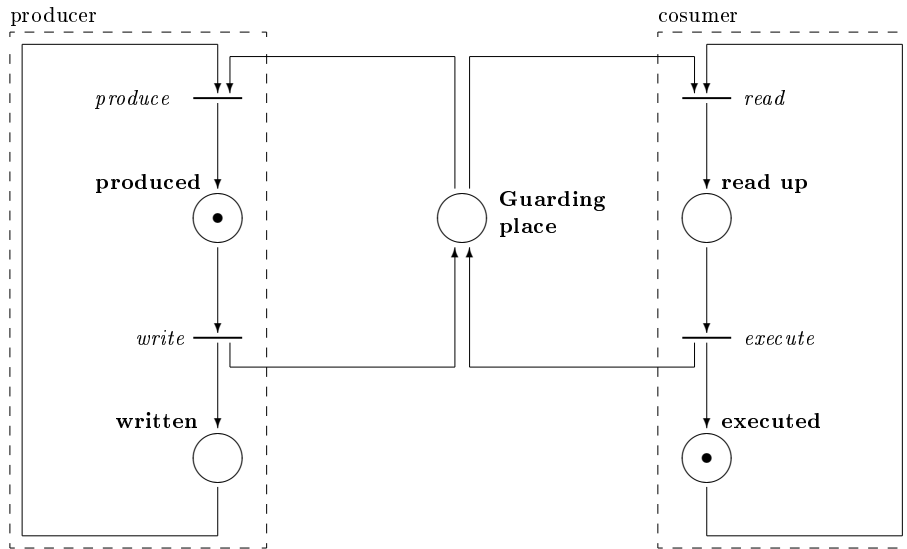


Figure 6.7: Petri net for the Model-View problem

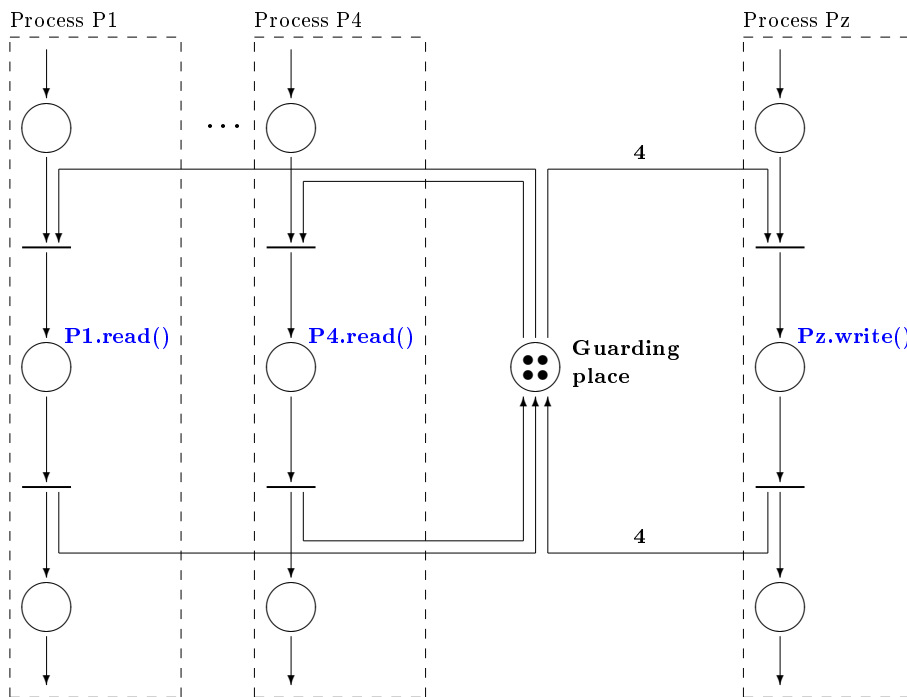



Figure 6.8: Petri net for the Readers-Writers problem

At the time a process writes, no reading or writing can be performed by another process, while reading operations can run more simultaneously (do not interfere with data consistency).

 In Figure 6.8, the task is solved for four readers and one writer. Every reader picks up a token from the guard place before reading. The writer tries to pick up four tokens when attempting to write, that is, one token for each existing reader. This makes it possible to write only if no reader reads (all the tokens are in the guard place). If a writer writes, all readers have to wait for the writer to finish writing and put back the tokens to the guard place.

If there were multiple writers, any writer would be blocked again if a reader reads and if another writer writes. From the guard place, the edge would be determined by the number of reading processes.

✂ The requirements for the solution are as follows:

- the data must be in a consistent state whenever any process accesses data,
- readers can read simultaneously,
- writing is excluded with any other operation (the both reading and writing),
- *bounded waiting* as for the critical section problem.

✍ **The Dining Philosophers Problem.** It is a typical task of parallel programming. The name of the task is derived from a known problem: (Chinese) philosophers are sitting at a round table and alternately thinking or eating. Everybody needs two chopsticks, but there are only five chopsticks on the table, one between each adjacent pair of philosophers. If a philosopher does not have a chopstick on his left and right hand, he has no choice but to think.

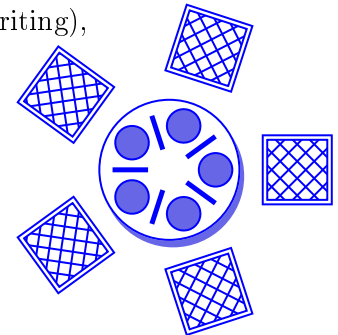


Figure 6.9: Five dining philosophers

The philosopher whose neighbors take his chopsticks alternately does not have a chance to eat, there is an aging process (the process constantly awaits the necessary resources). If all of them suddenly lift their chopsticks with their right hands, they are deadlocked, because they all hold one chopstick in the right hand and wait for the left one, which is just held by the left-side neighbor and therefore unavailable.

When applying to processes, we have five (generally n) resources used by five (generally n) processes (philosophers). Suppose that the resources are interchangeable (no matter how unhygienically it sounds).

The solution to the problem lies in the fact that not all five philosophers are dropped at the table at a time (and so do not allow all processes at once), but at most four ($n - 1$ if we have n resources). The consequence is that at least one philosopher is satisfied, so even if all of them suddenly

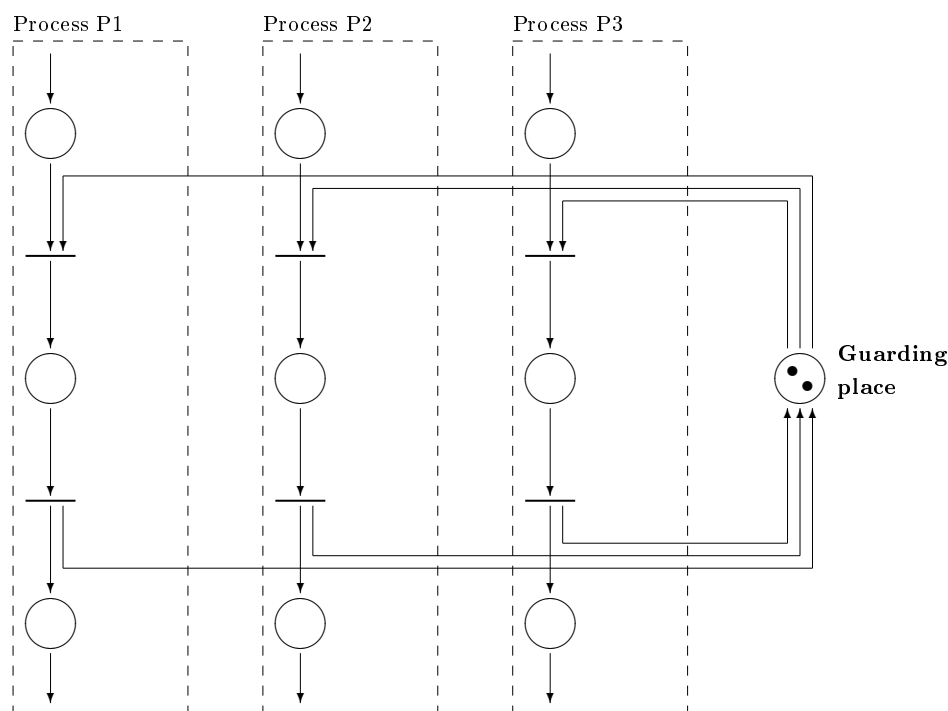


Figure 6.10: Petri nets for the Dining Philosophers problem (for three processes and three resources)

lift a chopstick with their right (or left) hands, in the processes case, at least one process can use the necessary resources and then release it for the next process. Another option is to order one philosopher to take the chopsticks in reverse order than others (which is not applicable to processes).

In Figure 6.10 the solution is indicated on a group of three processes and three resources. Only two processes are allowed to work at once to be able to use the resources they need. Generally, the number of processes is irrelevant, and it is important that we have a maximum of 1 token less than the resources in the *guard place*.



Example

Let us solve the Dining Philosophers problem using semaphores. We have N critical sections for N resources, so the array of N semaphores. We initiate them all to 1. One additional semaphore serves to monitor the resources to be used at most by $N-1$ processes (initialized to $N-1$).

The following code is for the i -th process:

```
#define N 5                // number of shared resources

struct TSemaphore sem[N]; // semaphores for resources
struct TSemaphore S;      // semaphore for guarding processes

for (i=0; i<N; i++)      // initialize all semaphores
    sem[i].state = 1;
S.state = N-1;

// for the i-th process:
do {
    think (i);           // i-th process does not need a resource yet
    wait (S);            // ...needs now
    wait (sem[i]);       // reserve the first resource
    wait (sem[(i+1) % N]); // ...the second resource
    eat (i);             // we have the both resources
    signal (sem[(i+1) % N]); // we return the both resources
    signal (sem[i]);
    signal (S);          // another process can use resources
} while (1);
```



Concurrent Processes. The task of concurrent processes is solved in a parallel system, where it is necessary to synchronize the operation of two processes running on different processors or on different nodes in a network (i.e. concurrently working processes). These processes need to be synchronized so that they perform a part of the code together.

If one process takes a common part of the code before the other one, it has to wait (in Figure 6.11 the process P2 has to wait), this code can only be executed when the two processes arrive at the beginning of the common part.

This process synchronization method can be used e.g. in the RPC mechanism (in Figure 6.11 the process P2 calls a procedure of a process P1) or in any synchronous data exchange of parallel processes.

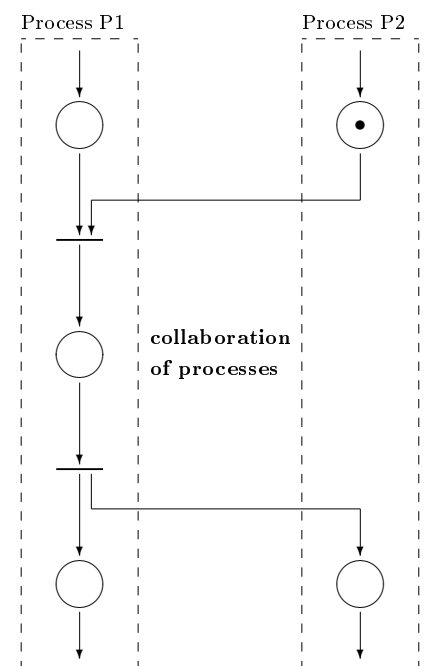


Figure 6.11: Petri net for Concurrent processes

6.6 Synchronization in Operating Systems

6.6.1 Possibilities of Synchronization in Windows

In Windows there are the following synchronization tools:

- IRQL (IRQ Levels) – classical IRQ masking was only applicable on a single processor system, whereas IRQL is applicable on multiprocessor systems as well, and it is not only about IRQs,
- spinlock, queued spinlock (spinlock with queue),
- mutexes and semaphores (we call them dispatcher objects), fast mutexes, guard mutexes,
- critical sections, events, reader-writer locks,...


 **IRQL (IRQ Levels).** IRQL is a mechanism that covers various types of events often linked to interrupts (IRQs). It is divided into so-called levels.


Table 6.1 shows the existing IRQL levels for the x86 architecture (32-bit Windows). However, the numbering and content of each level varies on architectures (x86, amd64).

31	High (“catastrophic errors”)	} Hardware interrupts
30	Power failure	
29	Interprocessor interrupt for cache consistency	
28	Clock timer	
27	Synchronization across processors	
:	Device interrupts (common IRQ)	} Software interrupts
2	DPC (Deferred Procedure Call), thread scheduler	
1	APC, Page Fault	
Priorities of threads 0–31 {	0	Low: user threads and most kernel-mode operations

Table 6.1: Levels of the IRQL mechanism

User processes (running in the user mode) with common priority have IRQL 0 (that is, they run on the IRQL 0 level). Kernel threads performing APC are on IRQL 1, higher IRQLs are then associated with other kernel activities performed with the majority of hardware interrupts.

The “high” level is used to shutdown the system mostly due to a kernel error (BSOD = Blue Screen of Death); the termination process has priority over everything else. Level 30 is documented, but it is not used, theoretically a system would have to go to it in the event of a power outage. Level 29 is used to communicate with other processors. Level 28, on the other hand, is commonly used – for updating system clocks and generally for various time allocation activities. Profiling (IRQL 27) is a method of sampling the state of execution of a process, so it is possible to monitor the activity of the selected process. IRQL 3–26 are intended to prioritize interrupts from devices.

 Using IRQL, Windows can always mask all IRQLs up to a certain level. For example, if the IRQL is masked up to 27, the requests of all common processes (IRQL 0), APC calls (IRQL 1), etc. up to level 27, are ignored. The consequence is that APC takes precedence over a common process, hardware interrupts take precedence over software interrupts. IRQL 0 is used for most of the system run time.

Processors (cores of a processor) require the index in the above table when an interrupt occurs. This index tells them if the level is masked, which should be ignored.

**Additional information**

All of the above applies to a 32-bit system. On a 64-bit system (amd64), the IRQL table looks a bit different (paradoxically, it has fewer levels – only up to 15). Additional information can be found on <https://blogs.msdn.microsoft.com/doronh/2010/02/02/what-is-irql/>.

**Spinlock.**

Spinlocks are only used in kernel for synchronization in a multiprocessor system and are actually mutexes. They have two states – “free” and “busy” (“locked” and “unlocked”). The spinlock itself is a very simple structure, and is always used with a more complex global data structure whose consistency protects, typically by queuing procedures or data structures to a particular device.

For example, if in a multiprocessor system a processor selects an entry from any process queue to run it, this queue must be locked by spinlock during thread selection, and unlocked after a thread has been removed from the queue, to have the queue in a consistent state.

Or, if a device driver that is currently running on a single processor works with data structures of its device (e.g. it sends data to the device input), it locks these structures because another part of this driver running on another processor could also want these structures work at the same time.

Spinlocks in kernel usually have an IRQL 2 level (DPC) or higher.

**Mutex and semaphore.**

A *semaphore* is seen in Windows as a mutex, which allows prepayment (or vice versa, the mutex can be taken as a binary semaphore). They are always associated with a particular object to which the access is synchronized. In the user space, they are typically used to synchronize thread activity inside one process.



A *mutex* is created by calling the `CreateMutex()` function, whose attributes are also a string that identifies the mutex (a shared variable of the given type) – inside one of the cooperating threads. This function returns the handle to the created mutex (each mutex is object). Other threads call the same function or function `OpenMutex()` with the same string. Logging out of the mutex is done by calling `ReleaseMutex()`, having a mutex identifier as a parameter.

In addition, the mutex may be used by calling `WaitForSingleObject()` or in the case more than one object by calling `WaitForMultipleObjects()`, which requires the mutex object’s handler as one of the parameters. When mutex is no longer needed, it is released similarly to other objects by the `CloseHandle()` function.

Semaphore is treated similarly, only we use the string “semaphore” instead of “mutex” in names of functions, and there are more parameters.



In the kernel mode, instead of mutex, the term *mutant* is used. There are several additional types of mutexes (such as fast mutexes).

**Critical section.**

The simplest synchronization mechanism is the critical section. This is not an object, so the initialization function does not return handle.



We initialize the critical section by the `InitializeCriticalSection(&cs)` function, we use the object that we want to synchronize, as a parameter. The `EnterCriticalSection(&cs)` function is used to enter the section and `LeaveCriticalSection(&cs)` to exit the section. Critical sections are designed to synchronize simple objects or variables, such as counters.

**Event.**

It is possible to create an event related to anything in various objects, threads are then linked to this event (waiting for occurrence of an event with specified parameters). Unlike the critical

section, an event is closer to mutexes and semaphores, it is an object, and we use the handle of this object in its handling functions.

In Windows, we can also wait for the end of a specific process or thread, an I/O operation (usually related to file usage), a timer, etc.



Additional information

<https://docs.microsoft.com/en-us/windows/desktop/sync/synchronization>



6.6.2 Possibilities of Synchronization in Linux



Mutex and futex. The basic synchronization object is mutex. Mutex is a kernel object, but it can be exported to the user space, and then it is called *futex* (fast mutex). Futexes are represented by a variable declared as atomic, the purpose of this variable is to speed up detection of the current state of the futex (otherwise the detection would have to be done by system calls, it is time consuming).

Most other synchronization mechanisms are based on futexes, the atomic variable of futex is a practical and rapid solution. Implementation can be found in the `libthread` library (because in the user space we usually synchronize threads of one process).

Mutexes can be used for both active and passive waiting.



Example



Let us show how to create a mutex (futex) for threads synchronization and how to use it. Suppose the required library – `libthread` – is loaded.

```
pthread_mutex_t  mymutex;                // data type for mutexes

if (pthread_mutex_init (&mymutex, NULL) != 0) {
    ...                                  // error handling
}
pthread_mutex_lock (&mymutex);          // locking the mutex
...                                     // code in "critical section"
pthread_mutex_unlock (&mymutex);        // unlocking the mutex
...
pthread_mutex_destroy (&mymutex);       // we do not need the mutex
```


If the mutex is locked by another thread when calling the locking function, the thread goes to the passive waiting instead of repeated locking, it is suspended.


Suppose we want to use active waiting. Then, instead of calling the locking function, we only test the mutex state, and – according to the return value – we know when it is being unlocked (if it is, this test function locks it up for us):

```
... // declaration, initialization
if (pthread_mutex_trylock (&mymutex) == 0) { // is it unlocked?
    ... // code inside "critical section"
    pthread_mutex_unlock (&mymutex); // unlocking the mutex
} ...
```




Mutexes are widely configurable in Linux. For example, there is a time-lock version (an object is always locked for a specified time interval), a non-recursive version that allows multiple locking of the mutex, a robust mutex capable of functioning even after a thread which has locked it and then has not unlocked has been terminated.

 **Priorities.** One of properties of a mutex is also the ability to set the priority ceiling. The priority of a process (a thread) that has locked a mutex, can be temporarily elevated above the level of all other processes (threads) that have subscribed to the use of this mutex. The `pthread_mutex_getprioceiling()` function is used to determine the priority ceiling for the given mutex, a similar function (`set` instead of `get`) changes this ceiling.

 **Rwlock.** This mechanism makes it possible to distinguish between a read lock and a write lock, which is the equivalent of the *Readers-Writers* synchronization problem.

Example

 We'll demonstrate the use of an rwlock. Notice the difference in locking for reading or writing. The read lock has to be done in loop because the number of possible locks of a single read lock is limited and the thread must be locked for so long until it is released to the code that is protected, or it is suspended. There is no need for locking for writing because the other threads are immediately suspended.


```
pthread_rwlock_t  mylock;


if (pthread_rwlock_init (&mylock, NULL) != NULL) {
    ...                               // error handling for lock creation
}


// locking for reading ("rdlock"):
if (pthread_rwlock_rdlock (&mylock) == EAGAIN) {}
...                                   // we use the lock for reading
pthread_rwlock_unlock (&mylock);

// locking for writing ("wrlock"):
pthread_rwlock_wrlock (&mylock);
...                                   // we use the lock for writing
pthread_rwlock_unlock (&mylock);
...
pthread_rwlock_destroy (&mylock);
```




 **Spinlock.** This synchronization object is used mainly inside the kernel, and it represents the possibility of active waiting. It is not recommended to use it too often (mutexes are better in most cases), it is intended to provide interprocessor operations. The spinlock is treated similarly to the mutex, only we write the string `spinlock` instead of `mutex` in names of handling functions, and while waiting, it is necessary to use a loop (for example with an empty command, but it can be any sequence of commands as well).

 **Barrier.** This is a simple synchronization object that is used not to synchronize access to an object, but rather to synchronize the achievement of a particular point in the code (to coordinate multiple working threads). The barrier is locked until all synchronized threads reach the specified point in their codes – as in the *Concurrence of processes* synchronization problem.


 **Condition variable.** Condition variables are similar to “events” in Windows. We use them when we want to wake up one or more specified threads by fulfilling a certain condition. The appropriate variable is tested by the thread, and if it has the value “not fulfilled”, the testing thread is automatically suspended.

The consistence of this condition must be ensured, so it is necessary to use a mutex for every access, including testing its value.

 **Semaphore.** Semaphores are generalized mutexes. There are two types of semaphores – named and unnamed.

There are two possible implementations of semaphores (in two libraries) – originating from POSIX standard and from System V (one of two UNIX systems branches), these two implementations have different syntax. In the example below we use the second type.

Example

 Let us demonstrate programming of a named semaphore. Each semaphore has to be declared and initialized.

```
// declare and initialize the semaphore, "prepay" the value 5:
sem_t *mysemaphore = sem_open ("/fmysemaphore", O_CREAT, S_IWUSR | S_IRUSR, 5);
if (mysemaphore == SEM_FAILED) {
    ... // error handling while initializing the semaphore
}


if (sem_wait (mysemaphore) == 0) {
    ... // the critical section, we use the protected object
    sem_post (mysemaphore); // end of the critical section
}

// closing and unlinking the semaphore:
sem_close (mysemaphore);
sem_unlink ("/fmysemaphore");
```



Named semaphores have their name that we have specified as a parameter when opening and unlinking. This is actually a device file that can be found in `/dev/shm`.


Example


 Unnamed semaphores are actually defined over a shared memory area. It is also possible to control the dynamically allocated memory (or access to this type of memory), the threads should have accessible the both shared memory and semaphore. In the following example, the semaphore is prepaid to 5 too.

```
// we allocate a memory area as the base for the semaphore:
void *sem_memory = ..... // any suitable allocation function
// we declare and initialize the semaphore:
sem_t *mysemaphore = sem_memory;
if (sem_init (mysemaphore, 1, 5) != 0) {
    ... // error handling
}


if (sem_wait (mysemaphore) == 0) {
    ...
    sem_post (mysemaphore);
}
```


```
sem_destroy (mysemaphore);           // closing the semaphore
...                                  // releasing memory,...
```

We can see that the use of unnamed semaphores is similar to named variant, it differs only in the way it is used. 

 All the synchronization objects described above can be shared not only among threads of a single process but also among processes. To do this, we need to set attributes for that object (enable sharing) and allow other (selected) processes to access synchronized memory.

All above described programming techniques are intended for synchronization in the user mode (except spinlock). Mutexes, semaphores and other synchronization mechanisms can be also used in the kernel, but with using different data structures and functions.


 In the kernel we also can find other mechanisms that can't be used in the user mode, such as sequential locks, RCU (Read-Copy-Update, for data that are often read but little changed), Completion (waiting to terminate another job), etc.


 **Additional information**


- <http://www.informit.com/articles/article.aspx?p=2085690&seqNum=6>
- https://docs.oracle.com/cd/E26502_01/html/E35303/sync-11157.html
- <https://0xax.gitbooks.io/linux-insides/SyncPrim/>



Deadlock


 *Quick preview:* Process deadlock occurs when a process is waiting for a resource that is allocated to other waiting processes. Of course, a deadlock is undesirable, so it is advisable to either design the system so that it cannot occur, or to prevent the situation by trying to predict the deadlock, or, if it does occur, to handle it as gently as possible with respect to the system and the processes.

 *Keywords:* Deadlock, waiting, class, safe state, Resource-allocation Graph, Banker's Algorithm, Wait-for Graph, Banker's Detection Algorithm, recovery.

 *Objectives:* The aim of this chapter is to understand the mechanism of dealing with the processes deadlock.


7.1 Deadlock Characterization

If there are shared resources that processes (threads) have to reserve, there is a risk of a deadlock. This situation can occur when a process waits for a resource that is allocated to other waiting processes.

 *Deadlock* is a situation where a set of processes (threads) are blocked because each process from this set is holding a resource and simultaneously is waiting for a resource held by another process belonging to the set. This situation can also be described as “mutual waiting”.

Another definition: The set of processes is in deadlock if each process in this set is waiting for an event that only any of the processes in the same set can invoke.


Deadlock is mainly about waiting for release of a resource used by some process, or some types of communication (e.g. waiting for confirmation of a message).


 Resource *starvation* is infinite blocking of a process by infinite waiting for resources. The affected process is perpetually unable to get necessary resources to perform its work.


The methods described below can be used by an operating system to handle deadlock of processes or by a programmer to handle deadlock of single process threads.


7.1.1 Model

Each (operating) system manages a finite set of resources to be used by competing processes and by the system self.

 This set of resources can be divided to *classes* (types), every class consists of the identical or mutually interchangeable resources, e.g. class of memory frames consists of all frames in the memory, or a class of open files consists of all accessible open files.

 **Remark**

We can construct a class of all printers shared in LAN, but it depends if the given printers are interchangeable. Two printers, the first one in the first floor and the second one in the seventh floor, are difficult to consider as interchangeable. 


 Each class therefore consists of a number of mutually interchangeable resources, which we call an *instance* of a given class. For example, memory pages are instances of the class of virtual memory pages. Some classes have only one instance.

In common operating systems, if a process (running in Ring 3) wants to use a resource, it needs to ask for a kernel (running in Ring 0) using a system call. The kernel either allocates the requested resource to the process, or it temporarily refuses and the process must wait.


 The resources are treated at the following stages:



1. *Request* – the process requests the resource. The result is:
 - (a) the operation is permitted, so the process obtains the requested resource, we continue with the step #2.
 - (b) the operation is denied, the process must wait (active or passive waiting).
2. *Use* – the process can use the assigned resource.
3. *Release* – if the process no longer needs the resource, it is necessary to release the resource to free it for another requesting process.


 **Remark**

Deadlock is discussed in the previous chapter. This situation can occur if processes (threads) share several synchronization mechanisms, as we have seen in the “dining philosophers” task. An example of a code with possibility of deadlock between threads of a same process using two shared mutexes is on page 317 in [Silberschatz2013]. 

7.1.2 Resource-Allocation Graph


 A **resource-allocation graph** is the directed graph used to describe the resource allocation status in system. There are two types of *vertices* in this graph:

-  process node – all active processes in the system have the own process node,
-  resource node – all resource classes have the own resource node.

Inside each resource we have dots (similar to tokens in Petri nets) representing the particular instances of the resource, e.g. the node  has three instances.

There are two types of directed *edges*:

- *request edge* – leading from a process to a resource $\text{○} \rightarrow \text{□}$, it signifies that the process requests the given resource,
- *assignment edge* – leading from an instance of a resource to a process $\text{□} \rightarrow \text{○}$, it signifies that the instance of a resource is assigned to the given process.

 When a process P_i requests an instance of a resource R_j (by a system call), the request edge leading from P_i to R_j is created.

When the request is approved, the request edge is removed and the assignment edge leading from the assigned instance of R_j to the process P_i is created.

If there is no cycle in the graph, there is no deadlock over resources. If a cycle appears in the graph, it means the *possibility* of a deadlock, but not a certainty (a non-zero probability of a deadlock).

Example

The left picture in Figure 7.1 shows three processes and three resources. The resource R_1 contains two instances, the both allocated to processes. The resource R_2 has only one instance, allocated to P_3 . The resource R_3 has three instances – two instances are allocated to P_2 , one instance is free. The process P_2 is requesting an instance of R_1 , the processes P_1 and P_2 have enough resources and can work.

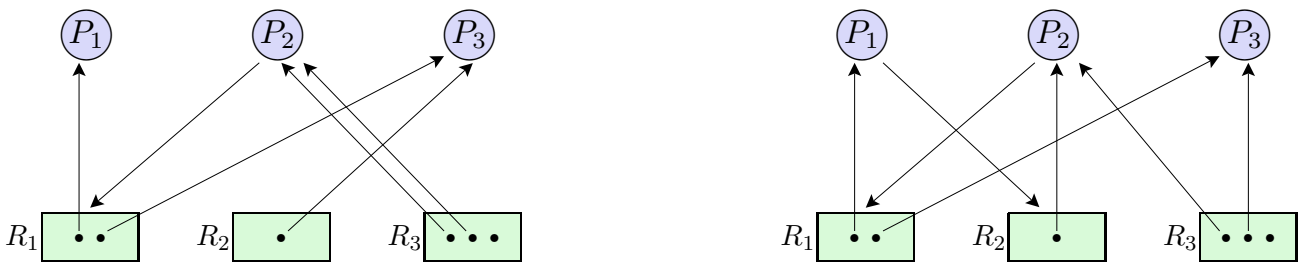


Figure 7.1: Resource-allocation graphs without deadlock

The request may be approved after the process P_3 releases its resource from R_2 . It is possible, because P_3 can work, it is not waiting for resources.

The right picture in Figure 7.1 is a bit different – two processes are requesting for resources (P_1 and P_2). P_1 is requesting for the resource belonging to P_2 , and P_2 is requesting for the resource which one instance is belonging to P_1 . There is this cycle in the graph: $P_1 \rightarrow R_2 \rightarrow P_2 \rightarrow R_1 \rightarrow P_1$.

But it is not deadlock – if the process P_3 releases its instance of R_1 , the request of P_2 can be approved and the cycle is away.

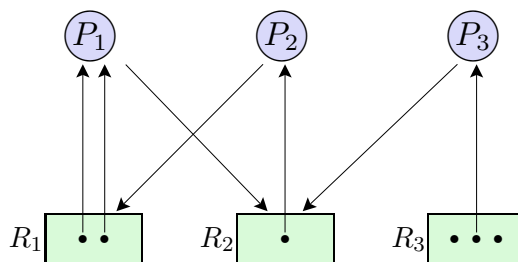


Figure 7.2: Resource-allocation graph with deadlock

Figure 7.2 shows the graph with deadlock. There is a cycle in this picture: $P_1 \rightarrow R_2 \rightarrow P_2 \rightarrow R_1 \rightarrow P_1$, all allocated instances in R_1 and R_2 belong to the processes P_1 and P_2 (so they mutually wait each other) and there is no way to break it, if the allocation is non-preemptive.



Remark

The existence of a cycle in the resource-allocation graph is a necessary, but not a sufficient condition for deadlock.



7.1.3 Deadlock Conditions

Deadlock is not something we would meet commonly. In order for it to come, certain *conditions* must be met:

Mutual exclusion. There are some unshareable resources in the system (only one process can use such resources). If we have no unshareable resources, deadlock cannot occur.

Hold and wait. A process must be holding a resource and waiting for an additional resource currently held by another process.

No preemption. Resources cannot be preempted; a process releases the resource after it completes its task (system is not able to release resources). If resource allocation can be preempted (i.e. if system can release a resource in order to allocate it for another process), deadlock cannot occur.

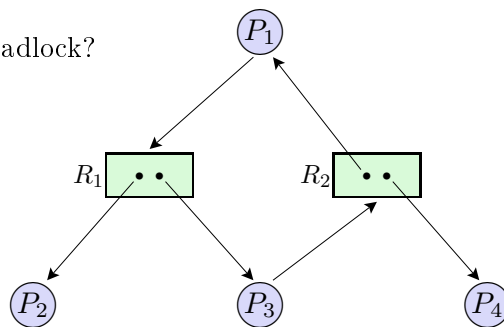
Circular wait. A set $\{P_0, P_1, \dots, P_n\}$ of waiting processes must exist where $P_0 = P_n$ and each process P_i is waiting for a resource held by P_{i+1} for $0 \leq i \leq (n - 1)$.

The last condition is an implication of the previous conditions.



Tasks

Look at the system – does this picture describe deadlock?



7.2 Deadlock Treating

As we can see, the deadlock is a very problematic state of a system. What shall we do with it?


- *Prevention* in design – we prevent deadlock by designing the system so that it cannot enter the deadlock state.
- *Avoidance* (prevention by checking) – we avoid the deadlock state by checking the current state after each request for a resource.
- *Detection* and recovery – we allow to enter the deadlock state, but we detect it and recover.

- *Ignoring* – we ignore possibility of deadlock, a user will solve it himself.

Let us look at the first three possibilities.

7.3 Prevention

Deadlock prevention is set of methods consisting in excluding at least one of the deadlock conditions (subsection 7.1.3). We will go through the particular conditions and the way to use them to prevent deadlock.


 **Mutual exclusion.** If unshareable resources (where synchronization is necessary) are a problem, we will try to ensure that all resources are shareable.

Files open for writing are unshareable. Some of them can be open for reading only, this operation may not be synchronized. But this solution is not applicable to all files.

There is another solution for some other resources: creating an access interface, a special process that will have dedicated access to the resource, other processes will only access this resource through this special process.

This solution is commonly used for printers. The role of the special process here is fulfilled by the operating system's print service (e.g. *Print Spooler* in Windows, *cupsd* in Linux). This service manages the print queue. If a process needs to print a document, it passes its print job to the print service and does not have to wait for the request to be handled.


⇒ This condition can be eliminated only partially.

 **Hold and wait.** To avoid this condition, a requesting process does not hold any other resources. There are two possibilities:

1. The first possibility is to allow processes to request resources only when they are newly created (in the “new” state), and later they can no longer request for resources. So only the first system calls are of the type “request for resources”. When a process performs any different system call, requests for resources are forbidden. This possibility is not suitable for most types of resources in the current operating systems.
2. The second possibility is to allow processes to request resources only after releasing all previously assigned resources. It is suitable only for some resources which may not be owned all the time (e.g. CPU time for the given process may not be continuous, and no information is lost when leaving CPU, if the process context is saved).

Some resource classes can be managed according to the first possibility, some of the remaining classes according to the second possibility.

⇒ This condition can be eliminated only partially too, it is not possible to release some resources, and some resources are needful simultaneously, e.g. memory frames when their content is necessary for using another requested resource. Resources assigned by the first possibility are used in a non-optimal way – the owning process is not motivated to release them.

 **No preemption.** The solution for this problem is to assign some resources preemptively. This method is only suitable for those classes of resources that can be released without the risk of damage to the process. The previous method was similar, but non-preemptive (the processes deliberately release

resources, with cooperation), while in this method, the resources are released without cooperation of the processes.

Preemption is used for such resources as a process CPU time (CPU scheduling) or page fault exception (virtual memory management): in general, for resources whose state can be saved and restored later.

One of methods suitable for preemptively accessed resources is to preemptively release resources only on demand.



Example


Denote R_1, R_x resources, P, Q are processes where P is requesting for R_1 :

```
if (is_free(R1))
{
    assign (P, R1)
}
else if (exists Q: (belongs(R1,Q) && exists Rx: (is_waiting_for(Q,Rx)))
{
    release (R1)
    assign (P, R1)
}
```

If the resource R_1 requested by a process is free, the resource is allocated to that process. If not, we find out the process owning the requested tool, it is Q . If Q owns another resources, the resource R_1 is preemptively released (we suppose that Q can request this resource later) and allocated to the requesting process.



⇒ This condition can be eliminated only partially too, for the resources eligible to preemptively release.

 **Circular wait.** To avoid this condition, we determine a total ordering of all resource classes, where each process may request only in an increasing order.

Let us define a function $order(class)$ returning the order number of the given class, so

$$order: class \rightarrow \mathcal{N}.$$



Example

If a process holds resources belonging to classes R_1, R_4, R_5 and is requesting a resource from R_8 , it is allowed, because $order(R_8) > \max(order(R_1), order(R_4), order(R_5))$.



The second possibility to solve this problem is the requirement that a process requesting a particular resource has to release all resources belonging to the higher order classes beforehand.

The order for resources is used (often in form of recommendation) mainly for synchronization objects.


⇒ This condition can be eliminated only partially. The first possibility prescribes the order in which the resources need to be requested, the second possibility is dynamic version of the first one. In any case, the success of the method depends on determining the order of the classes.


The method is practically applicable only to synchronization objects, with only one instance in each class (ie, we work with individual synchronization objects, not classes, we determine their order).

7.4 Avoidance


When using this method, processes are not forced to release prematurely the allocated resources, but the principle is to estimate when the allocation of additional resources could cause a deadlock and delay that allocation.

7.4.1 Safe State

 The algorithms in this section require that each process declares the *maximum number of resources* of each class to possibly request in future. This information is used in simulation to verify that allocation of another requested resource would not result in a deadlock.

 A *safe sequence* for the current allocation state is the sequence of processes $\{P_1, \dots, P_n\}$ where for each P_i , the maximal resource requests (as mentioned above) can be satisfied by the currently available resources increased with the resources held (and released) by the processes P_j for $1 \leq j < i$.

The safe sequence can be constructed in simulation where we nondeterministically choose the order of P_1, \dots such as a process P_i may wait for resources held by the previous processes, but must not wait for resources held by the following processes.

 A system is in a *safe state* if there exists a safe sequence, so if there is at least one resource allocation order in which all processes can successfully complete their activity without deadlock.

A state that is not safe (i.e. a system is in an *unsafe* state) does not necessarily mean deadlock, but may lead to it.

7.4.2 Resource-Allocation Graph Algorithm

A variant of the above described graph can be used to avoid deadlock, but only in case of resources where each class has exactly one instance.

Besides the stated edge types, we use one additional edge type: a *claim edge*, leading from a process to a resource, but unlike the request edge, it is dotted and determines that the process is claiming the resource in the future (it does not need it now, but it declared that it will probably need later).

Claim edges for a particular process arise when the process is created. If the process requests a resource to which the claim edge is leading from it (it cannot request the resource to which the claim edge does not lead), the claim edge changes to the assignment edge (the edge orientation changes too), and when released, changes to the claim edge again (re-orientation).

Claim edges for a particular process arise when we run this process if the process requests a resource to which the claim edge is leading from it (it cannot request the resource to which the claim edge does not); the resource changes to the resource allocation edge (the edge orientation changes), and when released, changes to the claim edge again (including re-orientation).

A request edge can be changed to an allocation edge (and hence a resource is allocated) only when the change does not create a circle – the change means changing the orientation of the edge. Therefore, the algorithm only simulates the change of edge orientation and starts the circle detection procedure on the graph.

Example

Figure 7.3 shows a resource-allocation graph with two processes and three resources (each with one instance, so we do not depict it).

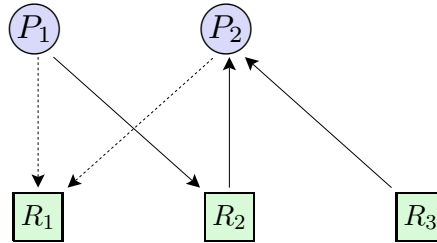


Figure 7.3: Resource-Allocation Graph for avoidance algorithm

The process P_1 is requesting the resource R_2 , and in future this process will request the process R_1 . The process P_2 has R_2 and R_3 allocated, and in future it probably will need the resource R_1 .

Suppose that P_2 can use the resource R_1 . In the simulation, we change the claim edge to the assignment edge, as we can see in Figure 7.4.

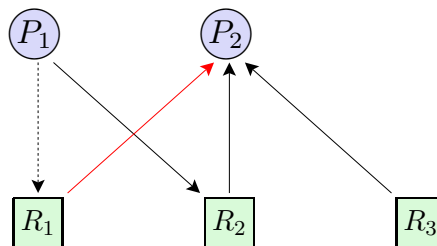


Figure 7.4: Resource-Allocation Graph for avoidance algorithm

There is no cycle in the graf, so deadlock will not occure and the resource can be allocated to P_2 .

In case the resource allocation was risky (it could lead to a deadlock), the “tested” edge would change to the request edge.



Remark


If we have exactly one instance in each class (and therefore we don't work with classes but directly with resources), the existence of a cycle in the graph is a necessary and sufficient condition for deadlock. The algorithm is simple – we only change one edge and use the graph algorithm to search a cycle.



7.4.3 Banker's Algorithm

In case of resources where multiple instances can be in classes, the graph algorithm is not applicable. We can use the *banker's algorithm*.

Each process must announce when it is started how many resources it will need for its activity. Whenever a process requests for resources, the system will find out how much other processes might need, and allocates them only if it considers that allocating the requested resources will not lead to an unsafe state.

 Suppose that there are n processes and m different classes of resources. We need the following data structures:

AVAILABLE – a vector of length m which indicates the number of free (available) instances for each class of resources.

ALLOCATION – an $n \times m$ matrix determining the number of resources of each class currently allocated to the particular processes. This matrix can be understood as the vector of vectors, where **ALLOCATION**[i] is a vector of all resources allocated to the given process P_i . This matrix corresponds to the allocation edges in the resource-allocation graph.

MAX – an $n \times m$ matrix determining the maximum demand of each process, it is the number of the remaining resources needed by the particular processes to complete their work.

NEED – an $n \times m$ matrix for the “future needs” of the particular processes. This matrix corresponds to the claim edges in the resource-allocation graph.

It is obvious that $\text{NEED}[i][j] == \text{MAX}[i][j] - \text{ALLOCATION}[i][j]$.


After the process is started, the appropriate matrix line in **MAX** is filled with information about how much of resources it will require in future.



Remark

We will use the \leq relation (or \leq in code) for vectors defined as follows: let V_1 and V_2 be vectors of the length m . $V_1 \leq V_2$ if and only if $\forall i (V_1[i] \leq V_2[i])$, $1 \leq i \leq m$. In words: the first vector is less than or equal to the second vector if all its elements are smaller or equal to elements with the same index of the second vector.



 **The safety algorithm.** This sub-algorithm (function) of the banker’s algorithm tests if the current system state is safe. We need two additional structures:

WORK – a vector of length m indicating the available resources during simulation.

FINISH – a vector of length n of the values true/false, the value **FINISH**[i] = *true* means that the process P_i has finished its work during simulation and its resources can be allocated to another processes.

Initialization:

- **WORK** = **AVAILABLE**
- for ($i=0$; $i<n$; $i++$) **FINISH**[i] = *false* (for all processes P_i)

Iterative simulation:

1. Find an index i such that the both the following *conditions* are fulfilled:
 - **FINISH**[i] == *false* (the i -th process has not been not “finished” yet)
 - **NEED**[i] \leq **WORK** (its “future” needs can be met)


If no such process exists go to step 3.

2. **WORK** += **ALLOCATION**[i]

FINISH[i] = *true*

Go to step 1.

3. If **FINISH**[i] == *true* for all i (all processes have been “finished” during the simulation), then the system is in a safe state.

 **The resource-request algorithm.** The main part of the banker's algorithm is used whenever a process is requesting for a resource. We need one additional structure:

REQUEST – an $n \times m$ matrix for requests of processes. This matrix corresponds to the request edges in the resource-allocation graph. **REQUEST**[i] contains all the requests of the i -th process.

When the i -th process requests for a resource (and this request is set to the vector **REQUEST**[i]), these steps are performed:

1. If **REQUEST**[i] \leq **NEED**[i], go to the next step. Otherwise, refuse (e.g. raise an exception), because the process exceeded its declared maximum needs.
2. If **REQUEST**[i] \leq **AVAILABLE**[i], go to the next step. Otherwise, suspend P_i to a waiting state, there are not enough resources.
3. Beginning of simulation. Modify the structures as follows:

AVAILABLE $-=$ **REQUEST**[i] (available resources are decreased by the required resources)

ALLOCATION[i] $+=$ **REQUEST**[i] (resources are allocated for simulation purposes)

NEED[i] $-=$ **REQUEST**[i]

4. Test if the system is in the safe state (the previous algorithm).
 - The state is safe: do **REQUEST**[i] = (0,0,...,0), all the requested resources are allocated.
 - The state is unsafe: return all changes made in the previous step, the request data remain in the vector **REQUEST**[i].

7.5 Detection

Again, we distinguish two cases: the first method (using a graph) is for a system where there is just one resource inside each resource class, the second method (modification of the banker's algorithm) for a system where multiple instances are allowed in resource classes.

7.5.1 Wait-for Graph

A *wait-for graph* intercepts the interactions between processes (a process waits for another process to release a resource). As we are only interested in what process has been trapped at the moment, we do not need information about which resources the processes are waiting for (it is easier to detect a cycle).

Example

We can make the wait-for graph by collapsing the resource-allocation graph, as we can see in Figures 7.5 and 7.6.

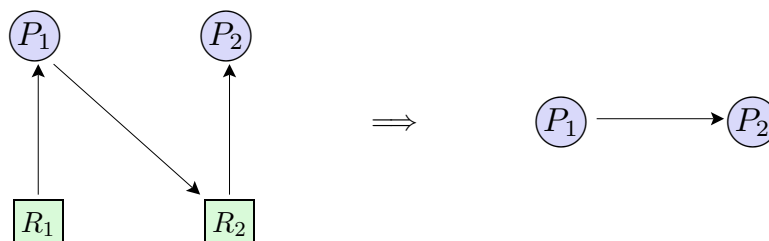


Figure 7.5: Resource-allocation graph and the corresponding wait-for graph, without deadlock

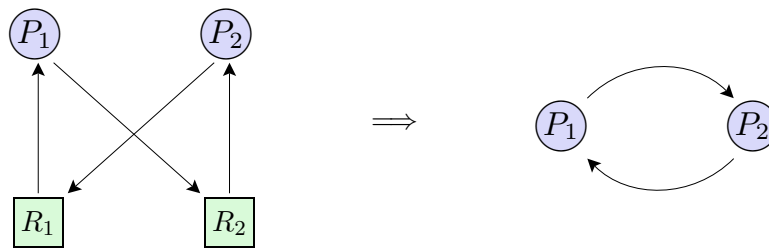


Figure 7.6: Resource-allocation graph and the corresponding wait-for graph, with deadlock



7.5.2 Banker's Detection Algorithm


This algorithm is similar to the “full” banker's algorithm, but it only detects existing deadlock. The used structures are similar, but processes need not declare their “future” needs.

 We need the following data structures:

AVAILABLE – a vector of length m which indicates the number of free (available) instances for each class of resources.

ALLOCATION – an $n \times m$ matrix determining the number of resources of each class currently allocated to the particular processes. This matrix corresponds to the allocation edges in the resource-allocation graph.

REQUEST – an $n \times m$ matrix for requests of the particular processes. This matrix corresponds to the request edges in the resource-allocation graph.

 We need two additional structures for each simulation:

WORK – a vector of length m indicating the available resources during simulation.

FINISH – a vector of length n of the values true/false, the value $\text{FINISH}[i] = \text{true}$ means that the process P_i has finished its work during simulation and its resources can be allocated to another processes.

Initialization:

- $\text{WORK} = \text{AVAILABLE}$
- for ($i=0$; $i < n$; $i++$) $\text{FINISH}[i] = \text{false}$ (for all processes P_i)

Iterative simulation:

1. Find an index i such that the both the following conditions are fulfilled:
 - $\text{FINISH}[i] == \text{false}$ (the i -th process is not finished yet)
 - $\text{REQUEST}[i] \leq \text{WORK}$ (its “future” needs can be met)


If no such process exists go to step 3.

2. $\text{WORK} += \text{ALLOCATION}[i]$
 $\text{FINISH}[i] = \text{true}$
 Go to step 1.

3. If $\text{FINISH}[i] == \text{false}$ for some i (some processes are not “finished” during the simulation), then the system is in an unsafe state, and the processes with “false” in the vector **FINISH** are deadlocked.

7.6 Recovery from Deadlock


The deadlock detection algorithm cannot be run too often, it is possible to run it at regular intervals or hang it on an event when CPU usage falls below a specified level.

 If we work with resources non-preemptively, the only solution is to terminate processes as long as there is a deadlock state, the resources of terminated processes are released and allocated to other processes.

Again, it is important to choose the victim(s) because a forcibly terminated process cannot, of course, complete its work. There are processes that can be terminated and then restarted without the risk of data loss or data inconsistency.

The system must specify conditions for terminating deadlocked processes. For example:

- using priority of processes,
- type of processes (interactive or batch, system or user),
- if the resources can be accessed preemptively,
- amount of resources used by a process,
- how long a process has computed,...

 With the preemptive work with resources, we gradually release the resources that are allocated to the deadlocked processes, and allocate them to other processes until the deadlock is removed.

The key is the choice of the victim(s) – the processes that will gradually lose their resources, and it should also be ensured that, after deadlock elimination, these processes can gradually recover the resources and be allowed to complete their work.

So, these issues need to be addressed:

- *Victim selection*: which resources of which processes are to be preempted? It is necessary to minimize cost.
- *Rollback*: when releasing resources of some process the given process must remain in a safe state, so we need to ensure that the process can continue its work after a certain waiting.
- *Starvation*: if deadlocks occur frequently, it is possible that some process becomes a victim repeatedly. So we have to make sure that when a deadlock situation is repeated, some process is not chosen too often.





Remark


Common operating systems such as Windows and UNIX systems ignore deadlocks, there is possibility of deadlock detection. Windows is able to use deadlock detection only for drivers – in Windows XP and later versions.



I/O Management

 *Quick preview:* In this chapter, we will first look at the structure of the I/O system, the types of peripherals, the drivers, and then briefly discuss the issue of low-level access to peripherals using interrupts.


 *Keywords:* I/O system, peripherals, buffering, drivers, driver models, kernel-mode drivers, user-mode drivers, device file, IRQ, interrupt handling.

 *Objectives:* The aim of this chapter is to become familiar with the principle of I/O device management, especially to get into the issue of drivers and interrupt management.


8.1 I/O Devices

8.1.1 Types of I/O Devices


We distinguish various types of I/O devices – type of working or communication (some of these types are related to a device interface).

 **Character-stream or block:** Character-stream devices transfer bytes or words – as a continuous data stream. Character-stream devices are e.g. a terminal, keyboard, mouse.


Block devices transfer blocks of data, similar to packets, attached with metadata. Examples of block devices are HDDs, SSD, RAM disks (for temporary data).

 **Sequential or random access:** If a sequential device needs to work with data at a particular address, it must first pass through all the addresses between the current and the searched location, it cannot be moved directly to the destination. A random access device can move to that address directly.


Magnetic tapes are sequential, whereas disks or semiconductor memories are random access devices.


 **Synchronous or asynchronous:** A synchronously communicating device is synchronized with other communicating devices using the same clock signal. An asynchronously communicating device uses its own clock, it must be synchronized with remaining devices on the bus by a special signal when initializing the device and then continuously during transmission.

Storage devices are synchronous devices, a keyboard is asynchronous.

 **Sharable or dedicated:** A sharable device can be used by multiple processes simultaneously, access to a dedicated device must be synchronized by some of the synchronization mechanisms.

Fully shared devices are e.g. keyboard, mouse, timer. A fully dedicated device is a printer. Shared devices can be partitioned, with a different process accessing each (dedicated) part, such as memory with page frames.


 **Device speed.** Various devices have various speed parameters: bandwidth, transfer rate, seek time, latency, . . . These parameters are also related to the I/O interface.

 **I/O operation direction.** A process can read from an input device, write to an output device, and some devices allow the both types of operations.

A simple keyboard is an input device, a printer (without scanner) is an output device, a touch screen or multifunction printer enables the both reading and writing: it is an input-output device.

8.1.2 I/O System


The role of the *I/O System* (I/O kernel subsystem, respectively) is to manage and control I/O devices.

 A device communicates with a computer via a *connection point* – port, socket, shared memory, etc. Further, the communication line continues with a bus (PCI Express, SATA, Thunderbolt, memory bus, etc.).


A *controller* is an integrated circuit (on a PCB or a separate chip) designed to communicate with connected devices. The controller is either simple, or it can be a circuit comparable to a conventional processor. For SSD controllers, ARM chips have often been used recently.

The next level of the communication hierarchy consists of *device drivers* and device *firmware*. Drivers can be accessed directly only by kernel modules (and they are usually kernel models themselves); common processes have to use a system call when necessary.

Depending on the particular operating system, we may also encounter other components involved in communication between I/O devices and processes. For example, in the kernel, *I/O Scheduler* typically handles access requests of processes.

 When a process (a thread) performs a system call with impact to a device (e.g. file access, memory allocation), the *I/O request* is generated. This request is put into the queue of the device by the I/O scheduler.

The kernel maintains the device status information in the *device-status table*. For each device, its status and all pending I/O requests are in its table entry.


 **I/O handling.** A processor needs to be informed about I/O request, status of request handling, and fulfilling the request. The I/O system accesses devices in one of the following ways:

- *Program-driven I/O* uses a special process or routine that has full control over access to devices. This process periodically checks the device status table, and ensures all the corresponding operations (discovering I/O request, passing the request to the I/O scheduler, . . .).

The problem is the checking loop, the process spends too much time by waiting in this loop.


- *Interrupt-driven I/O* does not check any table, but the processor is informed of all phases (including confirmation of operation completion from the device) by interruption.

8.1.3 I/O Buffering

 A *buffer* is a memory storing data being transferred between two devices or between a device and a process.

The most common reason for using a buffer is to balance communication between two devices operating at very different speeds, for example, a file to be printed after being sent from the print queue is waiting in the printer buffer, or a large file to be sent to the network is waiting in the network interface buffer. Buffering is used in the communication in the producer-consumer model.

Another reason is to equalize the difference in the size of the processed blocks (or bus width), or the need to complete a larger block before handing over to the target device.

 Sometimes it is necessary to use *multiple buffers*. For example, if a single buffer is not sufficient to balance the device's speed (a sender is too fast), we will use an additional buffer, and alternate the two additional buffers in communication.

In advanced graphics methods, we use at least two buffers in video memory – a frame is gradually processed in the back buffer, and then moved to the front buffer (to send through a graphical interface into a display).

Three buffers are sometimes needed to synchronize communication between CPU and GPU in video memory, and the I/O interface. The both these units, and I/O transfer work in parallel, the CPU encodes operations to create a display frame and the GPU executes them to generate a frame. Each vertex buffer (front, first back and second back) holds information for one frame, multiple frames are completed in parallel without data hazard and delay. Figure 8.1 shows using three buffers for this purpose.

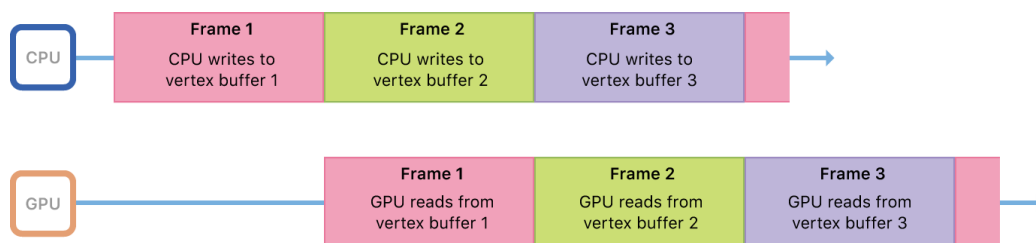


Figure 8.1: Triple buffer model in V-sync¹

Remark

Beside buffer, we also use cache. What is the difference between buffer and cache? All transmitted data passes through the buffer in the original order, and can also be completed there. In contrast, cache needs mapping function and predictive algorithms, data is only copied between two locations, cache is placed between them.

Additional information

- <https://www.displayninja.com/what-is-freesync/>
- <https://lifelhacker.com/learn-the-basics-of-nvidias-g-sync-and-amds-freesync-mo-1831578473>

¹From: https://developer.apple.com/documentation/metal/synchronization/cpu_and_gpu_synchronization

8.2 Device Drivers

A device driver is a software communicating with a device controller, providing a software interface to a hardware device. A driver enables kernel to access functions of the device without knowledge of implementation details specific to the particular device.

Each operating system defines certain types of drivers as well as *driver architecture*. Driver architecture determines the driver structure, its interface, the way it communicates with the device, the kernel modules, or processes.

8.2.1 Drivers in Windows

In Windows, we distinguish different types of drivers according to various criteria. The overall structure is quite complicated, the description below has been simplified.

There are these types of drivers in Windows, two basic *models* (i.e. how a driver is programmed, what it can implement and how it communicates with other components):

- WDM (Windows Driver Model) – most drivers of common devices (keyboard, mouse, sound card, etc.), used since Windows 2000,
- WDDM (Windows Display Driver Model) – special model for multimedia drivers (graphic cards etc.), used since Windows Vista,
- older drivers (predecessors of WDM) – e.g. PMD (Protected Mode Driver), and RMD (Real Mode Driver) for very old devices, we do not usually meet them at current systems.

The models WDM and WDDM exist in several versions, their specifications may vary slightly.

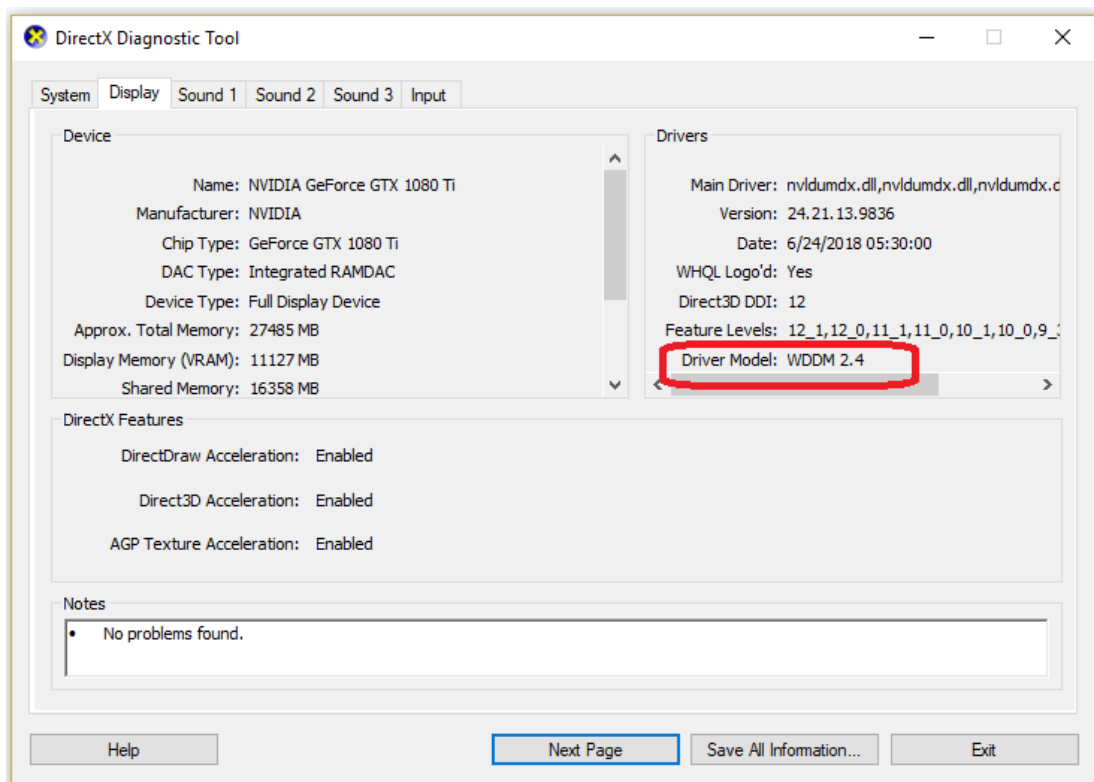




Figure 8.2: DirectX Diagnostic Tool, display driver information

Figure 8.2 shows the *DirectX Diagnostic Tool* (to run this tool, use the `dxdiag` command), its second tab for display driver information, and as we can see, its type is WDDM. The next tab contains information about sound devices, the driver type is WDM.


 The driver data can be found in the registry database, the same keys as for services (Windows handle drivers and services in the same way): in the key `HKLM\SYSTEM\CurrentControlSet\services`. Each driver or services has its own key here, and the variables for each key are:

- **DisplayName** – long name of the driver or service, or the location of this string in some library or executable file,
- **ImagePath** – path leading to the file of the driver or service (executable file, .SYS file, library),
- **Type** – type of service or driver, e.g.
 - 1 (Service Kernel Driver) is a device driver working in the kernel mode,
 - 2 (Service File System Driver) is a file system driver (e.g. NTFS),
 - 16 (Service Win32 Own Process) is a service with the own process, no other service is inside its process,
 - 32 (Service Win32 Share Process) is a service running in a shared process with multiple services,
 - 272 (Service Win32 Own Process Interactive) similar as 16, but it can work interactively (e.g. a tool with a window),
 - 288 (Service Win32 Share Process Interactive) similar as 32, but it can work interactively,
- **Start** – when the driver or service should start, e.g.
 - 0: a driver or service runs during system boot,
 - 1: when the kernel is initialized (after all modules with the value 0),
 - 2: automatically after system start,
 - 3: it can be started, but not automatically,
 - 4: disabled, it is not possible to start this module,
- **Tag** – ordinal number indicating the order in which drivers and services with the same value in “Start” are started (it is used only for these modules which depend on other modules, and their order is important for their dependence),

 We can divide drivers by code location (or form of communication with system):

- *kernel-mode drivers* – these drivers are kernel modules, they are mostly loaded from .SYS files,
- *user-mode drivers* – more similar to services, they usually run inside a host process, mainly in `svchost.exe`.

For example, the NTFS file system driver `ntfs.sys` runs at kernel mode.

 Each of these types of drivers has its auxiliary subsystem providing their running:

- *Kernel-Mode Driver Framework* (KMDF) – subsystem for kernel-mode drivers,
- *User-Mode Driver Framework* (UMDF) – subsystem for user-mode drivers, its part is also the *Driver Manager* module providing communication of drivers with processes (similar module exists for services – SCM, Service Control Manager).

The kernel-mode drivers have better communication capabilities (with all kernel modules including device drivers), which mainly affects communication throughput (it is not necessary to switch between

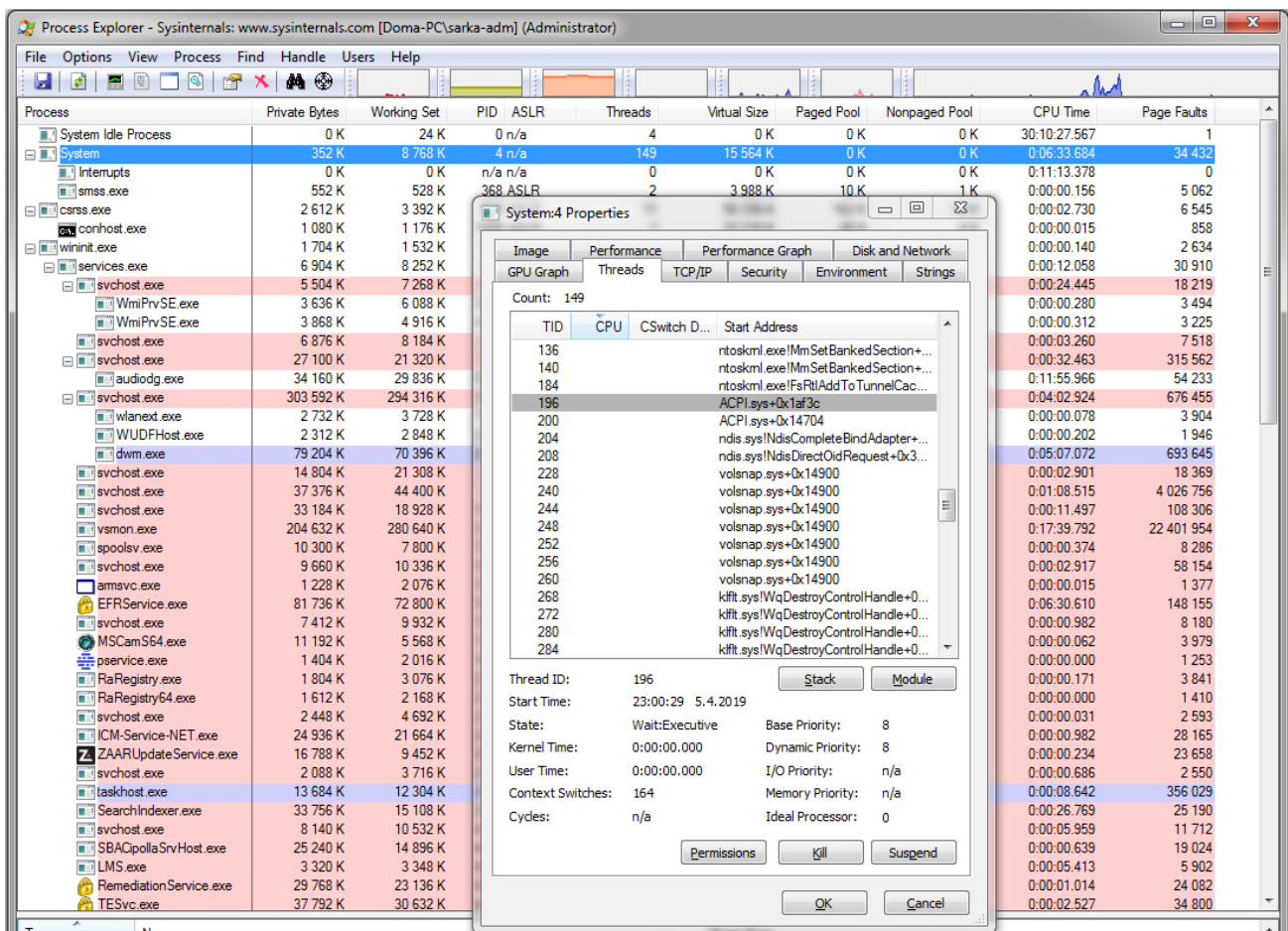



Figure 8.3: Threads of the System process in Process Explorer

kernel mode and user mode as often), but on the other hand, they are at risk for the kernel – whatever goes wrong in the kernel, it affects the entire system (blue screen, etc.).

As we can see in Figure 8.3, many kernel-mode drivers work as one or more system process threads, here, for example, the `ACPI.sys` driver runs as a thread with the priority 8 (normal).

 There are two types of drivers according to the *installation method*:

- *Plug-and-Play drivers* – they are related to a specific device that makes sense to consider this function (removable media, some types of expansion cards, keyboards, mice, printers, etc.), and communication with the power manager is also assumed for these devices,
- *non-Plug-and-Play drivers* – they are usually not hardware-related, or they are, but they communicate with the associated device via another driver (typically communication protocol drivers).




Remark


When installing a driver (not only a device driver), above all, the system needs an `.INF` file containing driver information, compatibility (e.g. supported OS version), installation file location, and installation method including registry keys to change or add. Many drivers, in particular the general drivers, have their `.INF` files in the `... \Windows \Inf` folder. If the appropriate file is not found, then the driver installation file is not available and we are asked to supply a driver (again, the location of some `.INF` file must be provided).

The .INF files are text files with the standardized structure, similar to .ini files. They are divided into sections, each section has its name (written in square brackets), and inside each section there are value settings (of the form `variable = value`). The .PNF files (with the same name) are the binary version of the .INF files.

We can explore the files e.g. `disk.inf` (generic disk driver information), `display.inf` (generic display driver info), `netip6.ini` (IPv6 support driver info) or `winusb.inf` (USB support driver info).

 The WDM drivers can be divided according to their *function* in the system, and to integration into the kernel communication structure:

- *function drivers* – these drivers communicate directly with the particular device, their task is to provide an interface to the device,
- *bus drivers* – manage buses (such as a PCI or USB driver), these drivers detect Plug-and-Play devices connected to the bus, ensure bus power, etc.,
- *filter drivers* – affect communication from or to a function driver, thus either extend functionality of the linked function driver or change it in some way; e.g. encryption, conversion, monitoring, file systems handling, etc.

 Device drivers (the function drivers in general) are divided into *classes*. Each class has its own characteristic procedures, functions and data structures (e.g. class for storage media). Each class has its standard driver which allows to access devices from different vendors in a standardized way so that the device works even if its features are not fully utilized.

 Most “full” drivers consist of two parts:

- a *port driver* – this part implements all standard features for such driver class (thus this driver is common for all devices of the same class),
- a *miniport driver* – this part implements vendor-specific and device-specific features.

Additional information

- <https://msdn.microsoft.com/en-us/windows/hardware/drivers/wdf/overview-of-the-umdf>
- <https://msdn.microsoft.com/en-us/windows/hardware/drivers/wdf/user-mode-driver-framework-frequently-asked-questions>
- <http://technet.microsoft.com/en-us/library/cc778056%28WS.10%29.aspx>

8.2.2 Drivers in Linux

 There are several *types of drivers* in Linux:

- device drivers – drivers of existing devices,
- file system drivers – all file system needs its driver,
- network protocols – modules with network protocols implementation.

 There are two basic types of drivers by code location:

- *kernel-space drivers* – work as kernel modules,
- *user-space drivers* – they work in the user space as services/daemons, and they cooperate with an “agent” providing access to kernel resources.

The first type of drivers has the advantage of having direct access to kernel structures and various communication options with other kernel modules, but on the other hand, their code needs to be thoroughly debugged because any error could fatally affect the kernel's operation. It is necessary to take great care to use mutexes, spinlocks and other synchronization mechanisms, in addition to locking also unlocking, to thoroughly analyze anything that comes from outside (such as an input device) because it could be a hacker attack.

The modules are stored in the .KO files (kernel object), inside the files `/lib/modules/kernel_version/kernel/drivers/class_of_module/name_of_module.ko`.

In contrast, drivers running in the user space have the advantage of minor security issues (but this does not mean that their programming might be fuzzy), but on the other hand they need a way into the kernel. This is mostly done by the *FUSE* kernel module (FileSystem in UserSpace), which acts as the mediator for communication with the kernel.



Remark

A huge number of drivers are being handled today through FUSE (File System in User Space, but it is not only for file systems), just for security reasons (and also easier to program, libraries with pre-built code are available). These are mostly used just as a data filter (all of which are actually in the UNIX systems as a file system), also for encryption, compression, logging, etc. Besides Linux, other UNIX or UNIX-like systems use FUSE: FreeBSD, OpenSolaris and others.

Examples of projects using the FUSE module: *ntfs-3g* (NTFS file system driver), *EncFS* (file system for encryption), *FuseCompress* (compression, among others, using the *gzip* algorithm), *ClamFS* (antivirus file access control), *sshfs* (SSH implementation), etc.



Additional information

- <https://github.com/libfuse/libfuse/wiki/Filesystems> (not all projects)
- <https://developer.ibm.com/articles/l-fuse/>



Other disadvantages of user-space drivers are similar to those of conventional processes, such as their memory pages can be swapped. Their communication with anything in the kernel is slower (it is necessary to switch between modes) – this can be seen, for example, with Gigabit Ethernet cards, if their drivers are designed in this way.



Several commands for working with (kernel-space) drivers and other kernel modules:

- `/sbin/lsmmod` – list of modules loaded into the kernel, with the basic properties,
- `/sbin/modprobe` – getting information about modules loaded in kernel, including their dependencies, and it can the both load and remove a module into/from the kernel,
- `/sbin/modinfo name_of_module` – basic module information.

Modules can be started with parameters, especially watchdogs (modules guarding some device and system property) use this option.



Example

The `lsmmod` command lists all currently loaded kernel modules and the list is very long. If we are looking for something specific, it is a good idea to filter the output. For example:


```
lsmod | grep wifi
```

```
iwlwifi      188416 1 iwldvm
cfg80211     524288 3 iwlwifi,mac80211,iwldvm
```

It lists the name of the module, its size, how many other kernel modules it is used by, and which ones they are. More detailed information about the `iwlwifi` module can be obtained as follows:

```
modinfo iwlwifi
```

It will list the name and location of the module file, license, description, dependencies on other modules, as well as digital signature information (kernel modules need a digital signature) and other information.



Each device (including virtual devices) has its driver (or multiple devices can share one driver). Each device has also its *device file* – the low-level communication interface (so when we need to pass data to some device, we write to its device file, and when we need to receive data from the device, we read them from its device file, and that all using standard system input/output calls). The device file name serves as the identification of the device in the user space.

In the kernel space, devices are identified by two numbers: the *major and minor number*. The major number determines the class of the device (thus, all disks belong to the same class, so they have the same major number), the minor number is an index of the device, various devices at the same class have different minor numbers.

Example

The major and minor device number can be seen in the following command listing:

```
ls -la /dev
```



The *character devices* (a keyboard, a mouse, etc.) are identified in the user space only by their device files, we do not need any other information. The *block devices* (a disk, a disk partition) are a bit more demanding: we need

- a device file for low-level access and identification,
- a mount point through which the contents of the storage medium (files) are available.

Linux includes all partitions and removable media in a single abstract tree, and the mount point of a partition or media is a subdirectory through which a user or process can access files on that partition or media. The root of the tree is denoted by `/` (slash), and this symbol is used for separation of directories of a path in this tree. The main partition with Linux installed in is mounted to the mount point `/`. If we have e.g. users' home directories in the separate partition, the mount point for this partition is `/home` (or if it is not a separate partition, this directory is simply a common directory). The mount point intended for removable media is `/media`, all these mount points are here, e.g. `/mount/usbflash` can be used for a USB flash disk contents (the files are accessed in the subtree of this node).

Some devices are not “physical”, they are only virtual. Some of them are used by users:

- `/dev/null` is a trash can – everything we write into is discarded, we use this output device to redirect unwanted output or error output of commands,
- `/dev/random` and `/dev/urandom` are input devices generating random numbers,
- `/dev/zero` is input device generating zeros (zero-value bytes).

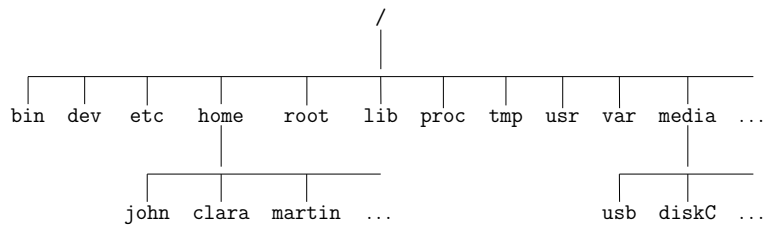




Figure 8.4: Simple structure of directories in Linux

 The kernel must ensure that access to devices is as abstract as possible – so that processes do not need to address the technical details of each category of devices. Until recently, in Linux, the *HAL* (Hardware Abstraction Layer, abstraction of access) module, in conjunction with *udev* (dynamic device handling) module, performed this task, but all newer kernels use *udev* only, it assumed the role of HAL as well.

 Device information can be found in a virtual system called *sysfs*. This file system is mounted to the `/sys` directory, and contains running information about devices. Its subdirectories hold information according to various criteria:

- `/sys/devices` – physical relations among devices (connections among devices),
- `/sys/bus` – information and structure related to buses (PCI, USB, ...),
- `/sys/class` – information and structure related to classes (device types),
- `/sys/block` – access to block devices, mainly disks,
- `/sys/power` – low-level access to the power management.

Remark

If you want to access a (physical) USB flash drive inside a virtual machine, this is possible. In the case of VirtualBox, you need to have Virtual Box Extensions installed, and the procedure for a specific USB flash drive is outlined in the figure below (of course first connect the given flash drive).

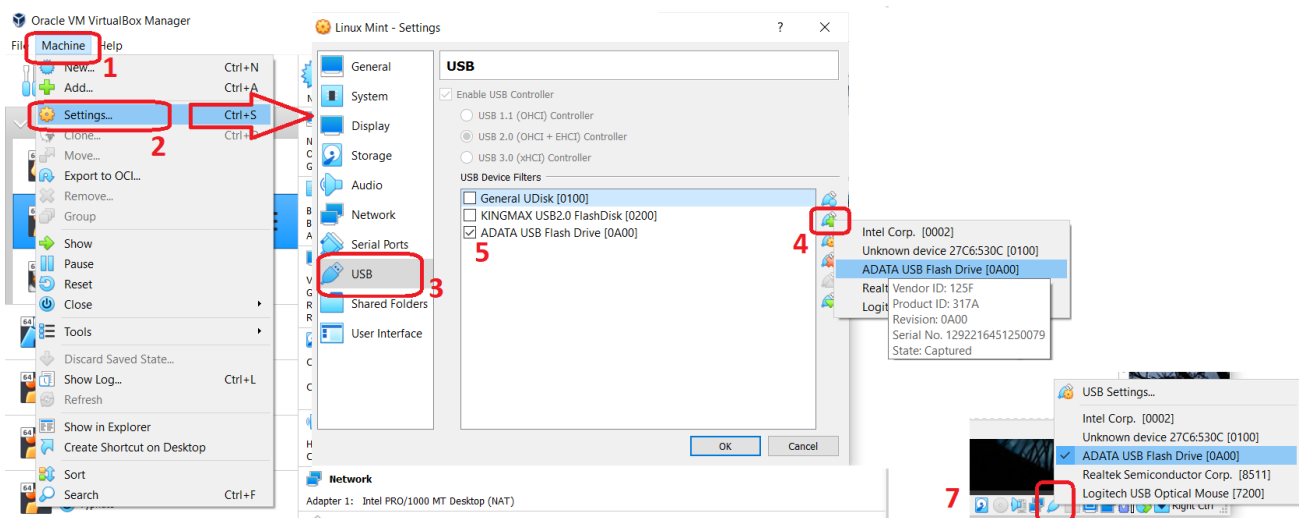



Figure 8.5: USB flash drive inside VirtualBox



8.3 Interrupts and Exceptions

8.3.1 Mechanism of Interrupts and Exceptions


 The term *interrupt* is understood as an interruption of the normal running of a process (the sequence of executed instructions of its program). In a multitask system, the interrupt causes a change in the state of the running process (the processor may be removed), but only after the finishing of the currently processed instruction.

Interrupts can be generated either by hardware, then we are talking about *hardware interrupt*, these interrupts have assigned numbers *IRQ* (Interrupt Request), or software by the operating system, then it is *software interrupt*.


Hardware interrupts are, for example, generated by I/O devices such as the keyboard (key press) or mouse (movement or button press), but also by the processor, interrupts generated by a timer (at regular intervals, preset in advance), in the case of hardware implementation of memory protection, an interrupt is generated by the processor in case of unauthorized access to protected memory.

Software interrupts are generated by a process, e.g. when requesting a resource (including output to the screen) or when trying to invoke an event and thus its service routine (within a process or even by another process or operating system), or by the operating system, e.g. when a system security violation occurs, or a page fault.


The basic characteristic of interrupts is that they come “unexpectedly”, without direct relation to the program code being executed.

 An *Exception* is similar to an interrupt (it notifies a situation that needs to be responded to), but unlike an interrupt, it results from the code being executed and would be generated again if the same code is repeated in the same situation (running the same processes, etc.) with the same data.

We also distinguish between hardware and software exceptions (e.g. a divide-by-zero error is a software exception, a bus error is a hardware exception) – although for hardware exceptions the line between exception and interrupt is unclear.

 In the operating system kernel we can find *interrupt handler module*, which handles interrupts from the point of view of the operating system.

8.3.2 Interrupt Handling

 In order to send interrupt signals to the processor, the device must register *interrupt handling routine*, and the same applies to exceptions. The interrupt handler contains the code to be executed when the device generates this interrupt.


The handling routine must not block the processor for long, and since it also often has the permission to access synchronized kernel objects, strict requirements are placed on it:

- must be as short as possible,
- to use only static data structures,
- atomic operations.


If code that does not meet the requirements needs to be executed, it can be split into parts:

- upper part (must be executed immediately, matches handler requirements),
- lower part (time-consuming but not critical operations, etc.).


The lower part (the less critical part) can be implemented in several different ways (depending on the specific operating system), usually having less priority on the processor than the interrupt handler itself, but more than the normal processes.

 Under certain circumstances, interrupts intended for a given process must not be handled (i.e. they must be ignored), e.g. for synchronization reasons, then we speak of *interrupt ban*. Disabled interrupts are *masked* (masked interrupts are ignored), but some interrupts cannot be masked and the process must receive a message about them (e.g. division by zero). This means that interrupts can be divided into two groups:

- *maskable interrupt* – this includes most interrupts from various devices,
- *nonmaskable interrupt* (NMI) – never masked, for example interrupts indicating hardware problems, and on UNIX systems interrupts causing a reboot after a system has frozen.

 On UNIX systems, shared interrupts cannot be masked. Masking is usually applied to one specific interrupt, although it is possible to disable all interrupts that can be disabled locally (on a single processor or core) at once. It is recommended to disable a single interrupt only in absolutely unavoidable cases, and it is even more recommended to avoid interrupt masking in general if possible.

8.3.3 Managing Interrupts in Various Systems


 **MS-DOS.** Peripheral management must first ensure that interrupts are handled correctly. In the simplest case, this is done by *interrupt vectors* specifying the address of the handling routine. In MS-DOS, interrupt handling is performed as follows:

- for each interrupt is defined program code (handler), which is to be executed in case this interrupt is generated,
- the interrupt vector is an ordered pair (a vector of two elements) [*segment,offset*] that specifies the address in memory (i.e. it's actually a pointer) where this program code is, and if we know where to look for this vector, then we can easily execute the handler for the interrupt that occurred,
- interrupt vectors are stored starting from an address that is known not only to the system but also to all programs, each vector contains two entries, each taking up 2 B, so in total the vector takes up 4 B of memory,
- vectors are stacked in a *table of interrupt vectors*, so when we know the number of interrupts that occurred (interrupts are numbered from 0), we just do the calculation

$$\text{starting address} + \text{interrupt number} * 4,$$

we get the address where the interrupt vector is with the address of the code serving the interrupt with that number,

- if an interrupt is generated, the program execution is interrupted and the interrupt handler specified by the vector is processed, then the program execution can resume.


 **Linux.** After each instruction is executed, the processor determines whether an interrupt was generated during its execution and, if so, proceeds as follows:

1. A running process is interrupted and placed in a queue, its context is saved.


2. Control is taken over by the operating system, or its interrupt handler module, which finds out what the interrupt is and creates a data structure with data related to the interrupt (type, how it was invoked, related data, . . .), if such structures are required.
3. If the interrupt channel for a given IRQ is not shared, the handler is called directly, but if the channel is shared, all handlers registered to that channel are called *in turn* until the processor is notified that the interrupt has been serviced. The
 ⇒ part of the handler should be to determine if the interrupt is indeed coming from a device belonging to that driver.
4. After the handling procedure is executed, the processor is assigned to one of the prepared processes (it can be the same one that was interrupted).

As written above, the (more complex) driver can be split into two parts – an upper critical one, which must be done now, and a lower less critical one, which “can wait”. The lower part can be implemented in several different ways. The most common are the follows:

- *tasklet* (time criticality somewhere between handler and normal process, priority slightly lower than handler), runs as *software interrupt* on the same processor as the original interrupt,
- *work queue* (priority lower, similar to normal processes), runs in the context of a kernel thread, is normally scheduled on any processor,
- execution *within a system call*, that is, in the context of a normal process (no matter the speed).

 The basis of the interrupt table is *Interrupt Descriptor Table* (IDT, it is fully in the control of the operating system), the same as the MS-DOS interrupt vector table – for each interrupt there is all the necessary information, but there is a bit more information. For each interrupt, we keep track of the handler addresses (including the necessary data, it’s a *concatenated list*, each entry has a pointer to the next), as well as status information, statistical information, and a spinlock to provide sequential calls to the handlers.


This is a concatenated list because multiple registrations (multiple registered devices) can be associated with a single IRQ – sharing – and it is necessary to have handling routines and other information stored for all of these registrations.

 **Windows.** In Windows in general, much of what was written above about Linux applies to interrupts, but with some differences.

When sharing interrupts, it is also necessary to keep a full list of records for each IRQ, and if an interrupt comes with a specific IRQ, one specific (correct) handler must be started. While in UNIX systems, the handlers registered to a given IRQ are run sequentially, and each must test whether or not the interrupt came from it, in Windows it is instead a query is sent over the bus to determine which device actually sent the signal, and only that single handler is started.

8.4 Running Non-Native Applications

Non-native applications are applications intended for a different operating system.


 The programs we can use for this purpose can be divided into several groups:


- *virtual machine* – simulates a computer (it can be a completely different HW platform than the one on which the system runs “in real”), we can have any number of other operating systems installed on this computer that we have a license for,

- *operating system emulator* – simulates a particular operating system,
- *subsystem* to run applications of another operating system.

8.4.1 Virtual Machine

Each of these programs has its own typical features, for example there are emulators used purely to emulate a specific hardware platform (for Amiga, ZX Spectrum, PowerPC, etc. – used mainly by programmers of these devices, game consoles, etc.) or allowing to choose between several platforms (installation of a new platform is then done by installing the appropriate module), or with the choice of HW platform we choose the operating system (on some platforms “not to choose from”, for example with Amiga).

 Some products support *paravirtualization*. The emulator does not need to virtualize the hardware, but only creates a communication interface within the host system that translates the hardware requests from the guest (internal) operating system into requests that the actual computer hardware understands. This technology requires direct support in the host operating system and must be added by modifying the kernel. In the case of open-source operating systems, this is not a problem, but only Microsoft’s virtualization solutions can run paravirtualization on Windows as the host system.

 Current processors include *hardware support for virtualization* (but its support is also required for other hardware, especially motherboards and network cards). If the virtualization solution runs on top of a processor that supports virtualization, the difference in response between the real (host) and virtualized operating system is smaller. If the internal operating system is 64-bit, then hardware support is practically necessary.


Hardware support must be enabled in the BIOS/UEFI. If the virtual machine refuses to run a 64-bit guest system, the reason may be that the hardware support for virtualization is disabled (it can happen, for example, after some Windows update).

The most well-known solutions:

- *VMWare Workstation*, *VMWare Player* runs on both Windows and Linux (host systems), virtually any can be installed inside as guest systems. It is one of the best and most popular universal simulators, commercial (there is a freeware version for personal non-commercial use).

VMWare’s products have traditionally been at the forefront of development in this area – this is where virtualization began.

- *VirtualBox* is a free product from Sun, running on Windows, Linux and macOS. Windows, Linux and various UNIX systems can run inside.
- *MS Virtual PC* is distributed by Microsoft (it was originally developed by another company, Microsoft bought this company), it runs only under Windows, commercial. In the latest versions, only running various versions of Windows as guest (internal systems) is officially supported, unofficially Linux can be installed on this product, but at your own risk. The configuration options are rather sub-average, and even surprisingly does not support Direct3D.
- *Xen* takes the hardware platform from the computer it runs on, otherwise we can install any operating systems. Freely distributable, for Linux and some other UNIX systems.
- *Qemu* is fast and easy to use, while being able to emulate different hardware architectures. It runs on Linux, Windows and macOS and is freely available.

 **Hypervisor.** Especially in data centers, we can see full (native) virtualization. This means that the lowest layer of the whole system (closest to the hardware) is not the kernel of an operating system, but there is *hypervisor* – a thin layer that is basically similar to the kernel. None of the installed operating systems are preferred (at least for most of these solutions).

The virtual machines and their specific operating systems then run on top of the hypervisor. The hypervisor is actually the interface that provides transparency to the operating systems. All hardware requests from the operating systems are routed to the hypervisor, and the systems enclosed in the virtual machines do not interact with each other and do not “see” each other (but can communicate with each other over the virtualized network).

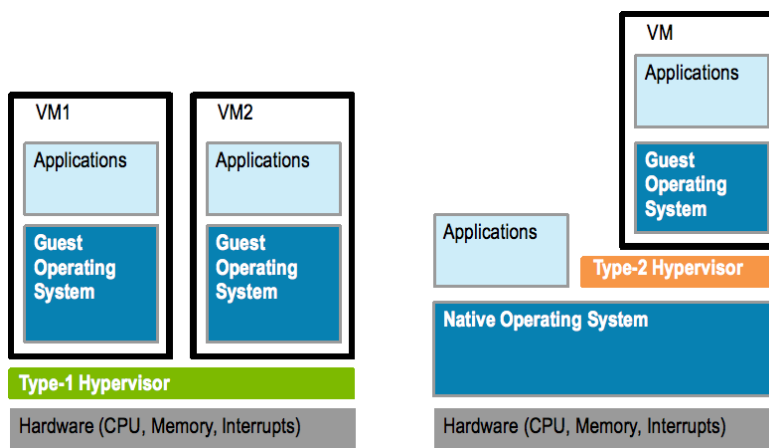


Figure 8.6: Type-1 and type-2 hypervisor²

Type-1 hypervisor is just a native hypervisor. Type-2 hypervisor is an application-level virtualization solution installed on an operating system (e.g. VirtualBox).

Products offering a native hypervisor are

- VMWare ESXi Server,
- Citrix XenServer,
- Microsoft Hyper-V for Windows,
- KVM as the kernel module for Linux.

8.4.2 Operating System Emulators and Subsystems

These programs simulate the running of a specific operating system, so we do not have to install the new operating system itself.

If an operating system with everything (almost) is emulated, we can work with everything in this operating system, including configuration. However, if it is a subsystem, the purpose is primarily to be able to run non-native applications.

We only install applications that we want to run virtually, using the tools provided by the emulator (subsystem). We must already have a license for the applications if the license terms require it (EULA, etc.).

²From: <https://microkerneldude.files.wordpress.com/2012/01/type1-vs-2.png>

Examples of emulators and subsystems:

- *Wine* is actually a recursive abbreviation of “Wine Is Not Emulator”. The authors used this name to emphasize that they did not intend to emulate Windows, but only to allow programs written for Windows to run on UNIX systems. It is a custom implementation of the Win API (the interface, the translation layer between the application and the kernel of the real operating system).


The Wine developers’ site has an extensive list of programs, possible problems when running them through Wine, and solutions. Unfortunately, some programs cannot be run this way or have unrecoverable problems (but these are exceptions). The programs, and their individual versions, are sorted into groups according to the difficulty of running them in Wine – platinum, gold, silver, bronze and others.

- *CygWin* is similar to Wine, but works as a subsystem in Windows to run Linux applications (a set of libraries plus an API translation mechanism). It is freely available and is often used even when we want to use Linux tools on Windows (including the shell).

CygWin includes a lot of different tools (word processing, programming tools, networking, even a graphical environment, and many more), and during installation we decide which of these tools to install.


In Windows 10 and 11 we can run *WSL* (Windows Subsystem for Linux), which is an emulation of several specific Linux distributions. Unlike other virtualization solutions, it is a module in the kernel, which means better system throughput, but on the other hand more risk for the kernel.

8.4.3 Server and Desktop Virtualization

 **Server Virtualization.** Server virtualization has already been written about here. This is basically full (native) virtualization, where the *native hypervisor* is underneath the entire system kernel (actually usually several different operating system kernels). Only the hypervisor has direct access to the hardware and resource allocation mechanism. The kernels of the virtualized operating systems run on top of the hypervisor.

Thanks to the fact that only the hypervisor has a gateway to access the resources it allocates to each OS, the individual OSES do not restrict each other, “do not know about each other”. They usually even run at the same time (we have more than one processor on the server), and a huge advantage is the ability to run applications native to different OSES without interfering with each other.

Products: *VMWare* ESXi Server (also *VMWare* vSphere and other related products), *Citrix* XenServer, *Microsoft* Hyper-V.

 **Desktop Virtualization.** The datastore stores generic or personalized desktop images (full OS and application installations). The user has either a thin client or any computer with the appropriate software (dedicated software or a specified web client, depending on the solution). The user “logs” into the corporate network, working remotely with “his” desktop.


There are two variants – centralized and distributed. In the centralized variant, the data center processor works primarily (hypervisor principle), only the interface is on the client side. In the second case, the client works on a full-fledged computer, where it runs its desktop image (images can also be distributed on removable media).


The purpose of desktop virtualization is primarily the possibility of simple mass management of desktops, the ability to run the same desktop on different devices according to the current position, and with remote access also the ability to work from home on the same desktop.


Again, we encounter mainly VMWare, Citrix, Microsoft products.

Chapter 9

Storage Media


 *Quick preview:* This chapter is devoted to block devices, primarily storage media. We will cover their structure, directories, and file systems intended for Windows and Linux.

 *Keywords:* Storage media, partition, MBR, GPT, directory, file system, attribute, journaling, FAT, NTFS, exFAT, VFS, ext4, virtual file systems, UDF.

 *Objectives:* The goal of this chapter is to get an overview of data organization on storage, especially various file systems.

9.1 Disks


Block devices, including disks, are characterized by the fact that data blocks are transferred, meta-information is required (data is more complex), block devices are the random-access devices, and the seek operation can be used.

 Disks can be *attached* in these two ways:

- host-attached storage (local),
- network-attached storage (over network, NAS).

The interfaces used for local disks and SSDs are SATA, eSATA, M.2 (or older mSATA), USB, Thunderbolt, SAS. The interfaces used for NAS devices are simply the network interfaces (Ethernet, Wi-fi, etc.), and it can be possible to transfer (local) disk operations over network: e.g. SCSI operations can be transmitted over network using the iSCSI interface, or the Fibre Channel technology can be used. FC is more expensive, but with better throughput, and iSCSI can be implemented in software (Linux has the iSCSI driver installed, and it is possible to load it into Windows as well).

9.1.1 Disk Format

 A disk can be divided into several *partitions*. A partition can be set as active, usually when an operating system is installed on it. We have a file system on each partition, such as NTFS (Windows), ext4fs (Linux), HFS+ or newly APFS (Apple MacOS), ZFS (BSD systems, Solaris), swap (Linux for a swap partition).

There are two common disk formats: MBR and GPT.

- *MBR* (Master Boot Record) disks have a maximum of 4 primary partitions, one of which can be extended. Any number of logical disks can be chained in an extended partition (but e.g. Windows cannot be installed on a logical disk, it must be on the primary partition).
- *GPT* (GUID Partition Table) disks can contain up to 128 partitions, no extended partitions are used.

MBR disk structure. Only numbers that fit in 32 bits can be stored in address fields in structures on a MBR disk – if we use the LBA addressing (see below). For this reason, the maximum partition size is only about 2 TiB, and the partition start address must also fit in 32 bits (i.e. the last partition on the disk must start at about 2 TiB and its size is not more than 2 TiB, the total size 4 TiB).

Figure 9.1 shows common structure of a MBR disk where the third primary partition is extended.

MBR is the main sector of the disk. Here we can find the main boot record (BIOS instructions that tell a CPU what should happen when an operating system has to be booted), and the partition table. The main boot record detects which partition is marked as *active*, and then attempts to boot the operating system from this partition (starts the partition’s boot program). If multiple operating systems are installed, there is a *boot manager* in the main boot record, this program allows to choose one of multiple installed operating systems. If only one operating system is installed, the first part of the boot loader of such operating system is here.

The *Partition Table* occupies 64 B on the disk. It has four entries (one for each primary partition

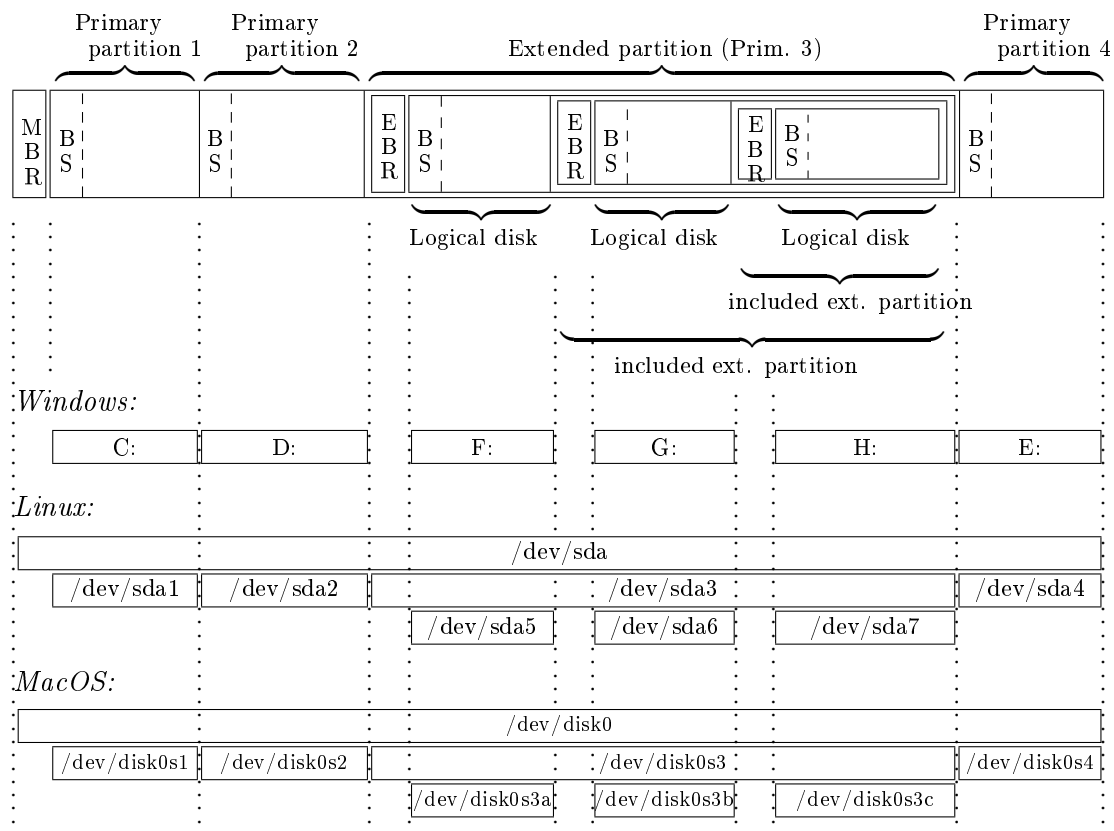


Figure 9.1: MBR disk structure and notation in various operating systems

– the extended partition is also considered primary), For each entry, the following information is about the corresponding primary partition:

- if this partition is active (hexadecimal number 0×80) or not (the number 0),
- location of the boot sector of the partition (the start address of the partition),
- type of the partition (extended, or what file system is present, each file system has its identification number),
- other metrics (the ending address of the partition, number of sectors from MBR till begin of the partition, length of the partition in sectors).

BS (Boot Sector) is the boot sector of the partition. If an operating system is installed in the partition, the main part of the boot loader of the system is here.

EBR (Extended Boot Record) is similar to MBR, there is the extended partition table here with two entries: one for the included logical disk and one for either second included logical disk, or for another included extended partition. The included extended partition can either contain one logical disk, or be divided into two logical disks, or be divided between one logical disk and one included extended partition, etc.

Each logical disk has its own boot sector.



Remark

Figure 9.1 shows names of the special files of the partitions in Linux and MacOS. BSD systems and Solaris use a bit more complicated hierarchy, with one additional level (it is defined for the ZFS file system):


- the special files for disks are denoted by `/dev/ad0`, `/dev/ad1,...` for SATA disks, `/dev/da0`, `/dev/da1,...` for SCSI/SAS and USB disks,
- each disk is divided into slices, the slices for the first disk (SATA) are denoted by `/dev/ad0s1`, `/dev/ad0s2,...`, e.g. the second slice from the fourth SAS disk is `/dev/da3s2`,
- each slice is divided into partitions, the special files for partitions have a distinctive letter in addition to the special files of slices, e.g. `/dev/ad0s2a` is the special file of the first (“a”) partition at the second (“s2”) slice of the first (“ad0”) SATA disk.



Additional information

- <http://linuxbsdos.com/2014/11/08/a-beginners-guide-to-disks-and-disk-partitions-in-linux/>
- <https://www.freebsd.org/doc/handbook/disk-organization.html>
- <https://developer.ibm.com/tutorials/l-lpic1-102-1/>




 **GPT disk structure.** GPT is part of the UEFI standard from Intel. This is a 64-bit concept, giving more options for addressing (that is, partitions can be very large). GPT only uses LBA addressing (CHS is not supported at all), see below.

We can have up to 128 partitions on one GPT disk. One partition can be up to 9.4×10^{21} B = 8 ZiB, but so large partitions are not supported by operating systems (i.e. an operating system would have a problem with using and addressing this partition), for example, Windows can handle no more than 18EB partition.

Protective MBR (1 sektor)
the rest of GPT header (33 sectors)
partitions
copy of the GPT header (34 sectors)

Table 9.1: Basic structure of a GPT disk

The structure of the GPT disk is shown in Figure 9.1. The GPT table takes up 34 sectors, one of them is called *Protective MBR* and its purpose is to ensure compatibility with older non-GPT systems. An operating system not supporting GPT considers such a disk to be a single-partition MBR disk with unreadable data. The rest of the GPT header is the primary GPT table.

 The *primary GPT table* contains the following information:

- GPT version, length of the header,
- checksum of the header (this field is counted as with zero value),
- LBA address of this and the backup GPT header (the backup header is at the end of the disk),
- LBA address of the first partition, and the last LBA address usable for partitions (i.e. the last sector before the backup GPT table),
- GUID number of the disk,
- information about partitions (the array of entries with information about the particular partitions, before these entries we can find information about this array – the number of entries, etc.).


In the array of entries with information about partitions, there are *entries* with the following information:

- the GUID number of the type of the partition, then the GUID number of the partition (16 B each),
- LBA addresses of the begin and end of the partition (8 B each),
- attributes (8 B), e.g. read-only, system, hidden,
- name of the partition (72 B).


The partitions follow, and the last part of the disk is the backup GPT header (copy of the primary GPT header).


9.1.2 Addressing

Each sector of a magnetic disk has its own address, and there are two types of addressing: CHS and LBA.


 *Cylinder-Head-Sector* (CHS) is old addressing scheme based on the disk structure. An address of a sector is a vector of three numbers – the number of the cylinder where the sector is located in, the number of the platter (= the read-write head for this platter; the first two numbers take the track) and the number of the sector at this track. The cylinder closest to the edge is numbered by 0.


This addressing scheme is not used now, it has one significant disadvantage: we have three numbers, thus three limits resulting from the way in which the BIOS, controllers, and interfaces use and store these three numbers.

 The old IDE (PATA) allowed to address only 512 B, with the 512B sector length. Its successor, EIDE, allows to address 1024 B. The next generation, XCHS (eXtended CHS), is able to address 8 GB.

 *Logical Block Addressing (LBA)* is strictly linear addressing scheme, it does not take into account disk geometry. The sectors are numbered from 0, we start at the track closest the edge.

There are some limits set for this platform. The oldest LBA used 28 bits per address (2^{28} sectors, 128 GB), later versions use 48 bits per address (2^{48} sectors), and it is sufficient for the current large disks.

 For MBR disks, the both addressing types can be used, GPT disk are used only with the LBA addressing type.

 **Remark**

MBR and GPT format is not “definitive”. The MBR disk can be converted to GPT disk, the transformation consists in creating the appropriate metadata structures on the disk (GPT table, etc.). If nothing is installed on the disk yet, we can boot into a live Linux distribution (i.e. the system is running from removable media, not from hard disk), and we will execute this change with the command `sgdisk -g /dev/sda` (if the given disk has the special file `/dev/sda`)


But there are also non-destructive methods, for example, a newer builds of Windows should handle this change from the graphical interface (in the Disk Management tool, `diskmgmt.msc`).



9.1.3 Scheduling

An I/O request contains the following information:

- whether it is an input or output operation,
- the source/target address of the place to read/write, either LBA or CHS,
- the number of sectors to transfer,
- the corresponding target/source address in memory for the read/write operation.

 The LBA address is converted to the CHS vector, the cylinder part of the address is the most important for the following algorithms.

FCFS. Suppose that the sequence of I/O requests contains the following cylinder addresses:

85, 170, 25, 130, 10, 140, 60.

and the initial cylinder number is 35. The first disk scheduling method uses a simple queue – First-Come First-Served.

The final number of tracked cylinders is a sum of (absolute values of) differences between the neighbor numbers of cylinders, including the initial cylinder:

$$|85 - 35| + |170 - 85| + |25 - 170| + |130 - 25| + |10 - 130| + |140 - 10| + |60 - 140| = 715$$

If the given sequence of I/O requests is fulfilled in the original order, it means a lot of head seeking through many tracks. It would be desirable to rearrange the I/O requirements so that the heads arm

does not have to alternate in both directions (among lower and higher cylinder addresses), or otherwise optimize the heads path.

SSTF. the Shortest-Seek-Time First (SSTF) algorithm favors requests requiring the shortest heads seek from the current position.

If the current position is 35, the nearest request is for the address 25. Thus the third request is moved to the head of the queue, and it is first fulfilled. So, the entire queue is rearranged, and if a new request comes into the queue, it is put in the same meaning. Our queue is rearranged:

25, 10, 60, 85, 130, 140, 170.

The final seek time is 185 clusters passed.

The disadvantage of this method is that the order in the queue can change very dynamically (whenever a new request comes in) and, moreover, requests for farther places may get stuck in the queue when requests for near addresses are constantly outdated.

If the same address is in the queue twice, the order of these addresses has to be kept.

SCAN. The heads start at the lowest cylinder address and fulfill the requests from the lowest address till the highest address (it scans the addresses in their order, not in the order of requests). When reaching the highest address, the direction is reversed, and the heads scan the addresses from the highest till the lowest address.

For our example, the final order is

60, 85, 130, 140, 170, (max), 25, 10.

(the start position is 35).

C-SCAN. The Circular SCAN algorithm is similar to the previous algorithm, but it does not change directions. If the heads reach the highest cylinder address it immediately moves to the lowest address, and the following “circle” begins.


For our example, the final order for the initial position 35) is


60, 85, 130, 140, 170, (max), 0, 10, 25.

LOOK and C-LOOK. The previous two algorithms really reach the outer limits (the zero cylinder and the maximal number cylinder), but it is not effective, these edge addresses are not used for reading/writing. The optimized forms do not reach the edge addresses, they turn back earlier. For example, the LOOK algorithm for our sequence:

60, 85, 130, 140, 170, 25, 10.

9.2 File Systems

 A *volume* is a part of a disk in which we create a file system. It can be one partition, or the entire disk (e.g. USB flash disk), or several partitions from one or more disks (thus we can use a logical volume, associating multiple partitions, e.g. with using LVM – Logical Volume Manager).


 A *file system* is set of methods and data structures by which the operating system maintains file records. File systems are simple databases that allow access to specific data, sorting (to directories), and keeping track of those data.

9.2.1 File organization and Access mechanism


Suppose that we have a file containing a sequence of records.

 **File Access.** Files can be accessed in one of the following ways:

- sequential access – the records are accessed sequentially, starting on the beginning of the file,
- random access – the records are accessed randomly, without relations between neighbors, and without passing the previous records,
- index access – we have an index = sequence of pointers to various records, this index is sorted by a certain criterion.

 **File organization.** There are several methods of file organization (the way data is stored in a file):


- *sequential* – the records are stored in a particular order, sorted using a key field, the records are accessed in sequence (we have to pass the previous records), this method is suitable for magnetic tapes,
- *serial* – the sequence of the records is as the original, no sort operation is performed, the records are accessed in sequence,
- *random* (direct) – the records are stored in random locations with no relationship between neighbor records, so they are accessed directly without passing the previous records, e.g. optical disks are of this type, this method can be used for magnetic and optical media,
- indexed-sequential file organization – the records are stored in the sequential order, they are accessed sequentially but using an index.

 **File attributes.** Besides data, each file needs some metadata, e.g. name, creator/owner, security information for access protection, flags (read-only flag, system flag, hidden flag, archive flag, . . .), timestamps (time of creation, time of last modification, etc.), current size, . . .

 **File operations.** We can manipulate with files using the following operations:


- Create – the file is created with no data,
- Delete,
- Open, Close,
- Read, Write, or Get, Put,
- Append – we can add data to the end of the file,
- Seek – carry the file pointer to a specified place inside file,
- Get Attributes, Set Attributes,
- Rename,
- Lock, Unlock.

9.2.2 Directories

 Files are organized in *directories*. Directories are containers for files and other (nested) directories – subdirectories.

Usually, a directory contains information about the files that are embedded in it, including their physical location on the disk (address), hence the name. Because a directory is actually a collection

of file data, in many operating systems it is also transparently understood as a file, albeit with special meaning.

 Directories form a structure that takes on varying degrees of complexity. The directory that contains everything else on the media is called *root*. We distinguish the following types of directory structures: *Single-level directory* – only one directory exists, all files are contained in it. This concept was used by the CP/M system.

Two-level directory – the root directory contains directories, but these directories can contain only files, no nested directories. The directories are used for users separation, each user has its own directory.


Tree-structured directory – directories can be nested, each directory can contain subdirectories, the entire structure is a tree with one root. This structure is used by Windows.

Acyclic-graph directory – in addition to the tree structure, it adds the ability to have files and some directories stored in multiple directories, so more than one path can lead to some items. It is necessary to ensure acyclicity so that the search algorithm does not loop around.

An item (file or subdirectory) is physically only at one address that can be listed in multiple directories. The main advantage is easy access to the same file from different directories.


It is used in UNIX file systems.

General graph directory – there may be more than one path to items, and cycles are allowed, as opposed to the previous solution. This type is used only as a virtual superstructure for simpler structures. For example, it may be a symbolic link system in UNIX file systems or Windows shortcuts. These links are short files with information about the actual address of the item and possibly other information.

 The acyclic directory structure may cause the *retention of acyclic graph problem* while adding new addresses to directories. This can be addressed in several ways.

The easiest way is to limit multiple addressing to files (to exclude directories) – there may be only a file in multiple directories, not a directory (that is, when creating alternative paths, they can only lead to common files, not directories), or we can add some directories with special significance to this mechanism.

For example, in UNIX systems, an alternative paths can be created only for files, not for directories, but the implicit directories *.* (pointer to the current directory) and *..* (pointer to the parent directory) exist as the alternative paths in each directory. The searching algorithm ignores “dot” paths, thus it does not loop.

 Furthermore, in this acyclic structure, when deleting items, it is necessary to ensure non-existence of orphans without any location. UNIX systems use this solution:

Each file has a counter that captures the number of links to that entry. When deleting an item, its counter is decremented by 1. If, after this reduction, it is 0, the item is physically deleted, otherwise it is left.

9.2.3 File Sharing

Each file has its owner (for common files and directories, the owner is mainly the creator, for system files, it is a system account or a main administrator). The owner usually has maximal access privileges.

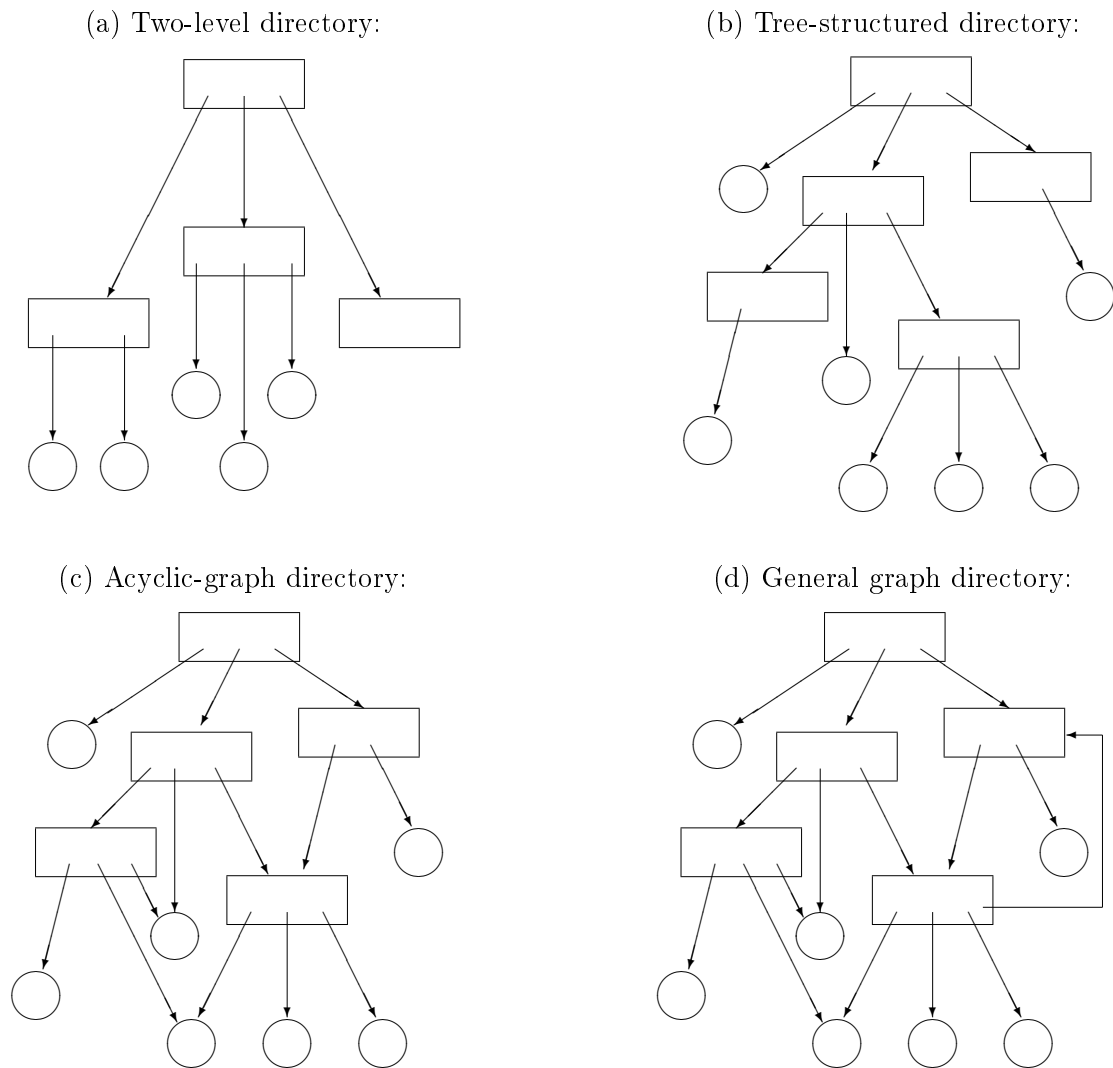


Figure 9.2: Directory structures


Then, other users and groups of users can have assigned certain permissions, so access to such file is shared among multiple users.

Some file systems can be handled remotely. For UNIX systems, it is the NFS file system, supporting file sharing over computer network.

In Windows we can use CIFS for sharing in the local network. CIFS is implementation of the SMB specification, and another implementation is Samba, supported in UNIX systems, so it is possible to share files, printers, etc. in a local network among computers with various operating systems.

9.2.4 Journaling File Systems

A journaling file system is a file system that retains information about ongoing operations.

 Each operation (such as saving a file), called a *transaction*, is subdivided into atomic sub-operations, and whenever a sub-operation is performed, the record is written to a special file or other structure, which is usually called a *journal* (or a file system log file). When a sequence of sub-operations of a transaction is written to the journal, this sequence of sub-operations is progressively performed, and

each sub-operation is noted to have been performed.

After all the sub-operations of the operation have been performed, the corresponding sequence of records is deleted, i.e. the journal contains only information about operations in progress.

If there is a power outage, the working process is killed, or the file system is otherwise interrupted, all journal entries are rolled back in the reverse order, so any unfinished operations are canceled. The goal is to bring the file system back into a consistent state.

More advanced journaling also stores the data blocks to work with, not just operations. It means that the log file acts as a buffer in which the result of the performed operation is completed by sub-operations, and when the result is complete in the buffer, it is written directly to disk in a single request.

Examples of journaling file systems:

- for Windows: NTFS,
- for Linux: ext3fs, ext4fs, XFS, JFS, ReiserFS and others,
- for MacOS: HFS+, APFS,
- for other UNIX systems: ZFS (does not use a traditional journal, the consistent state is ensured in another way).


Some of these file systems log only metadata (such as NTFS, and it is default for ext3 and ext4, but can be changed to full journaling).

9.3 Windows File Systems


 Files are stored in *clusters*, one cluster decays over several sectors (4, 8, 16, . . . sectors per cluster).

9.3.1 Older Windows File Systems

FAT stands for File Allocation Table, the system is based on the file and directory location of the table at the beginning of the partition. First we look at the structure of the simpler variant (FAT16) and then we'll show what also works in the newer FAT32.

 **FAT16.** Typically, the cluster length for very small partitions is 2 sectors (1 KB), with the capacity increasing, this value is significantly higher, determined by partition size. The FAT16 file system partition structure is as follows:

- *boot sektor*
- *FAT* (File Allocation Table)
- the second *FAT* (copy), if the primary *FAT* is not correct
- *root* (main directory of the partition)
- *clusters* of the remaining directories, numbered from 1, each cluster has its own small entry in the *FAT* table

 **Contents of the *FAT* table.** Individual data area clusters are numbered, *FAT* contains one 2B record per cluster (hence *FAT* 16, 2B = 16 bit, but not all possible values are used for cluster numbers, some are reserved and represent special codes for example for a failed cluster).

The contents of the table entries determine what we find in the respective cluster. If the cluster is free, there is a 0x0000, a defective – the number 0xFFFF (this number is written here by disk surface inspection programs).

If a part of a file or directory is stored in the cluster, the corresponding table entry identifies the next cluster (i.e., concatenation for the cluster in which the file continues). If this is the last cluster of a file or directory (and therefore no cluster does not follow), there is a number `0xFFFF` in the FAT entry.



Example

The file begins at the cluster `0x0021`, follows at the clusters `0x0027`, `0x0025`, `0x0026`, `0x0029`. The cluster `0x0022` is damaged, the remaining clusters up to `0x002A` are free. The FAT table from the entry `0x21` till `0x2A` has the following contents:

Index	0021	0022	0023	0024	0025	0026	0027	0028	0029	002A
Contents	0027	FFF7	0000	0000	0026	0029	0025	0000	FFFF	0000

Table 9.2: Example of the FAT table contents



To load a particular file (or directory), first of all we need to know the cluster number on which it starts. In the FAT table entry for this cluster, we find out on which cluster it continues, in its entry we find the next cluster number, etc.


Chaining clusters can be an advantage (the organization doesn't take up too much space on the partition), but also a disadvantage (damaging one FAT entry results in losing the rest of the file).

Data area. Under this term we will understand everything that is behind FAT tables, i.e. root and clusters. Root contains links to directories, directories can contain links to other directories or links to files by tree, root can also contain files. Root can also contain an item of type *label*, which represents the partition name.

While a regular file contains any data, a directory consists of 32B *directory entries* describing the files and subdirectories for that directory, the following information being recorded in the entries:

- file or subfolder name (8 B),
- file extension (3 B),
if the entry is for the partition label, this name occupies all the previous 11 bytes,
- attributes (1 B), the particular bits are `xxADLSHR` where
 - x no used,
 - A for archiving,
 - D directory,
 - L label,
 - S system file or directory,
 - H hidden,
 - R read-only.
- creation time and date and last access date (3+2+2 B),
- the number of the first cluster of this file or directory (2 B) the label entry has this value set to 0,
- last change time and date – writing to file or changing the directory structure (2+2 B),
- length of the file (number of bytes), not used for labels and directories (set to 0).

If the first byte of a directory entry is set to 0, it means the end of this directory (similar to NULL in dynamic data structures), this entry does not represent any file/subdirectory.

 **VFAT** VFAT is an abbreviation of Virtual FAT and is a FAT16 add-on that adds long file name support to the properties of this file system (also applies to longer file extensions such as .html) and the ability to use some other characters (such as national characters) in names, or spaces. It is a virtual driver through which communication with the FAT16 system goes. This term is also referred to as the FAT32 system, which has similar characteristics, but internally, without the need for a superstructure.

The name of the file or directory in VFAT up to 255 characters, some sources indicate that this length is including the file path. A limitation is necessary because the file name (more or less often including the file path) is used as a parameter of many programming functions, it must fit in the storage space defined by the data type. Long name support itself is implemented in such a way that the following entry in the directory is used for a longer name.

 There are four types of directory entries:

- files,
- subdirectories (subfolders),
- volume label,
- entry for the extended name of a file or directory.

The entry for the extended name of a file or directory has a specific form. Length is the same as for others (32 B) but it contains some additional parameters and 13 Unicode symbols (i.e. each in 2 B) for the extended name. As with files, these entries are concatenated (in the FAT table), the next entry in the string contains an additional 13 characters, ...


In the original file or directory entry, the file name in the 8.3 form is derived from the long conversion name (deleting spaces and others in DOS, or replacing them, the end is cut off and replaced by identification files or directories with the same abbreviated name.


 **Remark**

In the newer Windows file systems including NTFS, the short name form 8.3 is sometimes very useful. If the file entry is corrupted or created in another operating system (MacOS), no operation with the full name is allowed, but the short name access is often possible. For example, if we want to delete this corrupted file, we can display the short name:

```
dir /x
```

(it is in the column before the last one), and the short name can be used e.g. as the parameter of the `del` command.


 **FAT32** takes all VFAT properties, including long file name support. We can define different cluster sizes according to the Windows version:

 Thus, the FAT table contains cluster entries as 32-bit numbers. When it comes to the records that determine which cluster the file or directory continues, in fact they are stored in 28 bits of this number, the rest is again reserved for special codes. For example:

- 0x00000000, 0x10000000, 0xF0000000 mean that the cluster is free (lower 28 bits are set to 0),
- 0x0FFFFFFF7 damaged cluster,
- 0xFFFFFFFF the last cluster of file or directory.

Volume length	Smallest cluster length
512 MB – 8 GB	4 KB
8 GB – 16 GB	8 KB
16 GB – 32 GB	16 KB
32 GB – 2 TB	32 KB = 16 sectors

Table 9.3: The smallest cluster length for FAT32

 The structure of directory entries is similar to FAT16, except for one thing – the number of the first cluster of a file or directory is stored here, and for the backward compatibility, it is in the same place as in FAT16, but the field is only 16 bits long, which is little. Therefore, an additional 16-bit field is added a few bytes further, so we have 32 bits in total, but in two different places. The top 16 bits are in the same place as in FAT16 at offset 0x14, the bottom 16 bits at offset 0x1a.

 **Remark**

What happens when we delete a file in FAT32? In the entry for this file, the character 0xE5 is written in place of the first character of the file name (or 0x05: 0xE5 is commonly used in character sets used in Japan), and the field holding the 16 upper bits of the number of the first cluster is zeroed (the field for the lower 16 bits remains). The following file cluster numbers in the FAT table remain until they are needed to store new files.

When we want to recover a deleted file, we will find all the directory entries where the first byte of the file name field is 0xE5. The lower 16 bits of the first cluster file number are present in the found entry, but the upper 16 bits must be estimated. The recovery algorithms are based on a file extension that specifies the format, browsing the contents of all clusters with the corresponding lower 16 bits, and trying to select a cluster that could be the first cluster of that file. For example, .PNG images have the string “PNG” in their contents from the second till the fourth byte.

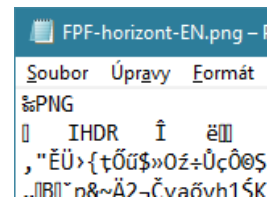


Figure 9.3: First few bytes of each PNG file


Because the first character of the file name was overwritten with 0xE5, it is not possible to restore the first letter of the short file name (but can usually be found from an entry for a long name, if any).

 **Additional information**

- <https://www.win.tue.nl/~aeb/linux/fs/fat/fat-1.html>
- <https://www.cnwrecovery.com/html/fat32.html>




9.3.2 NTFS

 NTFS (New Technology File System) is a journaling file system developed for Windows NT series. The main requirement in its development was to ensure greater data security, especially the ability to define access rights for users.


In the NTFS file system, we have the ability to control file and folder access by defining access

rights for different users and groups. Each file or folder has its Access Control List (ACL) with a list of users and groups and their access privileges.

 **Properties.** NTFS was much inspired by UNIX file systems and took over much of their properties. For example:

- everything is a file, including metadata structures,
- access permissions for users and groups, but different from those in UNIX systems,
- names of files are in UNICODE (the UTF-8 set),
- using a journal,
- support for hardlinks and softlinks,
- sparse files – files with empty areas with no information, e.g. databases, disk images where much bytes are zero; only the parts of the file that carry the information occupy disk space.

The hardlinks can be created by the `fsutil hardlink create` command in the Command line (with higher access privileges), or in code by the `CreateHardLink` system call, and there is a counter of hardlinks for each file, increasing by this command or system call. They can lead to common files, not to directories.

 **Structure.** The length of clusters, as in FAT systems, is derived from the volume size (up to 64 KB, i.e. 32 sectors).

Size of volume	Size of cluster
512 MB nebo méně	512 B
512 MB – 1 GB	1 KB
1 GB – 2 GB	2 KB
2 GB nebo více	4 KB

Table 9.4: Cluster size for NTFS

Several metadata structures are used inside the NTFS volume, the main important structures are the following:

`$BOOT` the boot record for this volume.

`$MFT` (Master File Table) – records for all files (including the files with metadata), each record is usually about 1 KiB long (can be longer). For each file, there is a list of file names (for all hard links leading to this file), a timestamp of file creation, last modification, last access and last modification of the `$MFT` record, attributes, security settings (the security descriptor), file position in the disk, etc.

`$MFTMIRR` – the mirror of `$MFT`, only its first four records.

`$LOGFILE` – the journal file for the transaction information.

`$BITMAP` – the array of bits, each cluster has one bit in this array. If the cluster is free (not used), the corresponding bit is 0, if the cluster is used, the bit is set to 1.


`$BADCLUS` – the similar array as `$BITMAP`, but determining the bad (damaged) clusters.

`$QUOTA` – information about user quotas.


These special files are invisible in common file managers.

NTFS prevents fragmentation by searching for the best fitting sequence of clusters (similar to the BestFit algorithm for memory), so that fragmentation occurs only when it is there too little free space on the partition (there is no “appropriately large” sequence of clusters) or when a file is extended and the number of needed clusters is growing. The main problem would be fragmentation of \$MFT; NTFS solves it by leaving some clusters free around the \$MFT, reserving them to the \$MFT.

NTFS, in its default form, reduces system throughput. It doesn't matter on faster computers, but otherwise there are ways to speed up its work. NTFS even updates the last access date and time while browsing the directory structure. This can be turned off by finding the value `NtfsDisableLastAccessUpdate` in the registry database and changing it to 1. This adjustment is also very suitable for SSDs.

 **Alternate Data Streams.** In the NTFS file system, we can associate a stream of data (or multiple streams) with each file (and folder). Physically, each file contains at least one (main) stream that is named by the file name. Other streams are named only by an additional string and are accessible through the colon, they are called the *alternate data streams* (ADS).

We can create an alternate stream either in process, or by redirecting (for example, we can redirect the output of the `echo` command).

 **Example**


We will create a text file whose main stream will remain empty, and store two other streams.

```
echo Hello > d:\file.txt:firststring    the first string is created,
echo Bye > d:\file.txt:secondstring    the second string is created,
more < file.txt:firststring            the first ADS is displayed,
more < file.txt:secondstring           the second ADS is displayed,
type file.txt                          the regular contents of the file is written, but the main stream is empty – so, this
                                         output is empty too,
dir file.txt                            information about our file, its length is 0,
dir /r file.txt                          the option /r can display all streams, including ADS.
```

If the file is copied to another volume and another file system (FAT32, etc.), the alternate streams are not copied.



9.3.3 exFAT

 exFAT (Extended FAT) is somewhat different from other Microsoft file systems. It is optimized for USB flash drives and SD cards. It can be said that its properties are somewhere between FAT32 and NTFS.

It is significantly simpler than NTFS (also faster) and writes fewer metadata to the media, thus less wears on the flash chip (we know that flash memory has a limited lifetime in terms of maximum write count).

Compared to FAT32, exFAT is able to store larger files, the size of a volume (partition) may be larger, also the cluster size (which is related). The specification also includes ACL support, but does not apply to all supported operating systems. While it supports transactions (journaling), it is only when this feature is implemented by the device manufacturer.

As with FAT32, there are also FAT tables (in similar terms – clusters chaining), but there are additional structures. Like NTFS, there is also a structure that records free clusters (it is not in FAT32).

It is a proprietary file system. This results in limited support in some operating systems. In general, exFAT is supported in Windows from Vista SP1 and later. MacOS supports exFAT since version 10.6.5. There is a FUSE driver for Linux as well.

9.3.4 Protection

Access is permitted or denied depending on the applicant and the requested operation (read, write, execute, append, delete, . . .). The permissions are usually stored as ACLs (Access Control List) specifying the user or group name and the corresponding privileges (access rights).

In Windows, each file and directory (folder) has its security descriptor. Users and groups are identified by their SID (Security ID), and the security descriptor holds SIDs and the relevant permissions, as we can see in Figure 9.4.

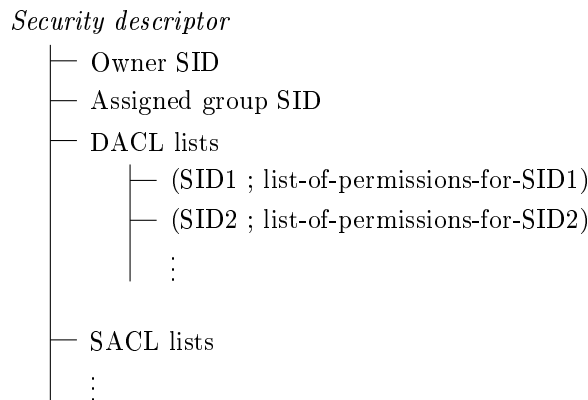



Figure 9.4: Simplified structure of a security descriptor


The permissions are either full access (F), or reading (R), writing (W), changing (C), none (N), or special privileges for special types of items (for example, listing the contents of a folder for folders). DACL lists are used to determine access privileges, SACL lists are used to determine accounting (access logging, etc.).

 In addition, for folders, we can specify which nested objects will inherit the set permissions (current folder, subfolders, files). Rights inheritance is determined as a combination of options

- (OI) – this folder and files inside
- (CI) – this folder and subfolders inside
- (IO) – exclude this folder (used in combination with the previous strings)

Possible combinations of the previous inheritance settings are in Table 9.5.

The privileges are set either in the graphical environment, or using the `cacls` command.

 The `CACLS` command displays and sets access privileges for the NTFS file system.

`cacls file` displays privileges (access lists) for the given file

`cacls *` for all files and directories in the current folder

`cacls file /g patrick:r` assigns the “read” privilege to the given file for the user patrick (all previously set privileges are rewritten!)

String	Meaning
<i>empty string</i>	only this folder, no inheritance
(OI)	this folder and files inside
(CI)	this folder and subfolders inside
(OI)(CI)	this folder, files and subfolders inside (full inheritance)
(OI)(CI)(IO)	files and subfolders inside, but not the folder
(OI)(IO)	files inside, not the folder, not subfolders
(CI)(IO)	subfolders inside, not the folder, not files inside

Table 9.5: Inheritance of privileges in ACLs

`cacls file /e /g patrick:w` similar, but the previously set privileges remain

`cacls *.docx /t /e /g patrick:w` the given user will have the write privilege to all .docx files in the current folder and its subfolders, recursively, the previous privileges remain

`cacls abc.docx /r patrick` removes all privileges to the given file for the given user, the user can have some privileges as a member of some group if any privileges are defined for that group

`cacls abc.doc /d patrick` sets the privilege “none”, access for this user to the given file is disabled


**Example**

We will play with access privileges. We move to the disk on which we have write rights and create two nested directories. We will also create a new user and add or remove access permissions.

```
md example    the first directory created
cd example    move into the directory
md insidedir  create the subdirectory
cd..         move one level up (all previous commands can be performed by one command: md example\insidedir)
net user newuser somepassword /add    create new user
net user newuser    display properties of the new user, including its groups
cacls example    display privileges of the new directory, there is an entry for “everyone” (F = full)
cacls example /e /d newuser    the new user is not able to use this directory anyway
cacls example /e /g newuser:r    we will set the read privilege
cacls example    display the privileges
cacls example\inside    is the recursion used?
cacls example /e /r newuser    we removed the entry with the user entry from the ACL of the security descriptor
cacls example    what is the result?
cacls example /t /e /d newuser    similar, but with recursion (/t)
cacls example /t /e /g newuser:r    recursively add the read privilege
cacls example    what is the result?
cacls example /d newuser    ups, we forgot /e, all the privileges (ACLs) are rewritten, we can verify it by the cacls example command
```

```
cd example    the message "Access denied" is possibly displayed, no privileges are defined
cacls example /e /g everyone:f    mistake repaired (access for setting privileges is allowed)
rd /s example    clean folders, recursively (/s)
net user newuser /delete    clean users list
```



 The newer versions of Windows allow to use the command `icacls`, originally intended for servers. Some of the parameters are similar to `cacls` (e.g. we use `/t` for recursion), but in general it has a different syntax.

Example

The following command

```
icacls directory /grant:r newUser:r /t
```

adds read permissions to the user `newUser` (“r” after `/grant` means adding permissions without overriding the ones already defined, only the second “r” after the user is a read permission), the last parameter means recursive inheritance.

In addition to regular permissions, we can also work with special permissions:

```
icacls someFile /grant *S-1-1-0:(d,wdac)
```


The *Everyone* group (by default has SID S-1-1-0) is assigned special delete permissions (D, delete) and the ability to write to lists specifying access permissions to the specified object (file) – WDAC.

We can also determine inheritance in detail:

```
icacls someDirectory /grant:r someUser:(OI)(CI)r /T
```

(We have specified full inheritance, we can use any combination of 9.5. The string (OI)(CI) means that no inheritance is to be used (do not propagate inherit).



 In PowerShell we can use cmdlets `get-acl` and `set-acl`.

Example


ACLs can be listed as follows:

```
get-acl c:\windows\system32 | fl
```

```
get-acl ~ | fl
```



9.3.5 Handling Partitions and File Systems

 DISKPART is a program to work with partitions for Windows since Vista and Server 2008. It is an interactive text console and can be used in both interactive and non-interactive modes.

This command allows us to work with disks and their partitions at an advanced level, including dynamic volumes and dynamic partitions. Typical tasks are to find disk or partition information, create or delete a partition, create or extend a dynamic volume, work with RAID (mirror), mount, or unmount a partition.

Some commands are designed to work with a specific disk or partition. Before using such commands, we must first select the disk or partition with the `select` command, the selected disk is called “focused”.

**Example**

We will show the basic usage of `diskpart`.

```

diskpart   we started the program, we are in interactive mode, then we enter the commands for this
              program (the prompt is now in the form diskpart>)
list disk   the output is the list of all disks (including removable media) with information, each disk
              has its identification number (from 0)
select disk 1  the disk #1 is selected, this disk is “focused”
list partition list of all partitions of the focused disk
select partition 3 the partition #3 is selected, focused
delete partition the focused partition is deleted
create partition primary size=18000 a new primary partition is created, with the size 18 000 MB
assign letter=e the letter is assigned, it is possible to mount the created partition into a directory
              similarly as in Linux, the command is assign mount=...)
detail disk detailed information about the focused disk
exit

```

If we don't work in interactive mode, we use a slightly different syntax. Internal commands are entered using program parameters, for example

```

diskpart /delete \Device\HardDisk0\Partition1 the disk #0 is selected and the partition #1 is
              deleted
diskpart /add \Device\HardDisk0 15000 a new partition is created
diskpart /delete F: the partition with the assigned letter F: is deleted

```

**Additional information**

<http://support.microsoft.com/kb/300415>



FSUTIL is intended to manage volumes with the NTFS file system, but it is able to handle FAT32 and exFAT too. It is necessary to have higher privileges to use this program. This command has several subcommands (key words):

```
fsutil fsinfo
```

writes information about disk volumes and file system:

```
fsutil fsinfo drives list of mounted volumes, including removable media
```

```
fsutil fsinfo drivetype d: information (drive type) about volume with the assigned letter
D:
```

```
fsinfo volumeinfo d: more detailed information (name, serial number, file system, etc.)
```

```
fsutil fsinfo ntfsinfo d: if the NTFS file system is inside, detailed information about this
file system is displayed (number of sectors, number of clusters, addresses of some metadata,
etc.)
```

```
fsutil fsinfo statistics d: statistical information about the volume (assuming the NTFS
file system) such as the number of reads, the number of writes, etc.
```

fsutil file

working with particular files (create, find, etc.)

`fsutil file createnew d:\somefile.txt 10000` the new file is created, its length is 10 000 B
`fsutil file findbysid smith d:\somedir` searching a file with the owner `smith`, in the directory `d:\somedir` (this command can be used if the user quotas are set)

fsutil volume

basic management of a volume – free space detection and disconnection for the selected volume

`fsutil volume diskfree d:` free space detection
`fsutil volume dismount d:` disconnection of the volume

fsutil behavior

display or change file system behavior (only for NTFS)

`fsutil behavior query disablelastaccess` we will see if timestamp is enabled or disabled for each access (whether the last access time entry changes each time a file is accessed, including read-only access)
`fsutil behavior set disablelastaccess 1` setting the timestamp for each access is disabled (it is important especially for SSDs)

There are more items that can be detected or set up in this way. The command always contains either the expression `query` or `set`.

fsutil dirty

works with the *dirty* flag

`fsutil dirty query d:` is the *dirty* flag set?
`fsutil dirty set d:` sets the *dirty* flag, this volume will be checked during the next system boot

fsutil hardlink

working with hard links

`fsutil hardlink create d:\somedir\newfile.dat c:\someoldpath\oldfile.dat` creates a hard link `newfile.dat` leading to the file `oldfile.dat`


fsutil quota

working with quotas

Quotas set each user a limit on the amount of space they can use for their own files. Usually there is a certain threshold that can cause an “alarm”, however, the user may still just a little exceed

`fsutil quota query d:` displays setting of created quotas
`fsutil quota violations` displays breaking of set quotas
`fsutil quota track d:` allows following the quotas
`fsutil quota enforce d:` allows enforcing the quotas
`fsutil quota disable d:` disables quotas
`fsutil quota modify d: 1024 10000000 smith` creates a new quota or changes the existing one for the given user, the first number is threshold, the second parameter is allowed space for the stated user

Other useful commands working with storage media, for easier access to partitions:


 `subst` creates virtual volumes (we specify a letter that will be a substitution for a long name of a folder)

`subst G: Z:\dir1\dir2\dir3` creates a virtual volume with the `G:` letter assigned, pointing at the given directory

`copy somefile.xxx G:` this letter can be used as other volume letters, including the tasks in GUI

`subst G: /d` deletes the binding between the letter and the directory

After restart, this binding is deleted as well.

 `mountvol` creates or destroys a mount point or an assigned letter for a volume

`mountvol` displays information about volumes (the both mounted and unmounted), the letter or mount point is displayed for the mounted volumes

`mountvol c:\media\firstvolume /L` finds out what is attached to the specified directory (if it is a mount point for volumes)

`mountvol c: /L` the same for volumes with an assigned letter

`mountvol c:\media\firstvolume \\volumeGUID` mounting of volume to the given mount point (directory), the directory must exist

`mountvol c:\media\firstvolume /D` the binding is deleted

The volumes we want to mount are labeled as follows:

`\\?\Volume{disk_GUID}\`

where GUID (Global Unique Identifier) is the device identifier, e.g.

`\\?\Volume{865f1ff0-a863-11d9-9f02-806d6172696f}\`

The `mountvol` command works close to kernel than the `subst` command. As a result, we need higher permissions, the drive labeling is not intuitive, and `mountvol` does not “see” the virtual volumes created by `subst`.

9.3.6 Comparison of Windows File Systems


For the volume size and the maximum possible file size, see Table 9.6.


	Max. volume size	Number of clusters	Max. objects in root	Max. file length	Max. number of files
FAT16	2 (4 v NT) GB	max. 2^{16}	512	4 GB bez 1 B	2^{16}
FAT32	512 MB – 2 TB (XP: up to 32 GB)	min. 2^{16}	65 534	2^{32} B minus 1 B	nearly 2^{32}
NTFS	256 TB minus 64 KB (for 64KB cluster) 16 TB minus 4 KB (for 4KB cluster)	$2^{64} - 1$ (XP: $2^{32} - 1$)	not def.	2^{64} B minus 1 KB (XP: 2^{44} B minus 64 KB)	$2^{32} - 1$
exFAT	128 PB	cca 2^{32}	not def.	16 EB	not def.


Table 9.6: Comparison of Windows file systems

9.4 Linux File Systems

9.4.1 VFS

 Linux works with virtual file system (VFS) (Virtual File System), through which all “real” filesystems are accessible. It is a kernel module through which all disk service calls go, covering file systems on all volumes (including removable media) and, if necessary, passing requirements to the particular filesystem being worked on. Through VFS, a user also uniquely accesses all devices, and everything is included in a single root directory structure.

 If we want to use a volume, we must mount it to VFS either in the graphical interface or in the console with the command `mount`. The system partition is already connected at system startup, so we don't have to worry about it, other volumes on hard disks usually also be connected automatically (depending on distribution). Removable media is needed to mount, in GUI it is usually resolved automatically.

 The most used file system for common volumes in Linux is `ext4fs`, but we can meet various file systems:

- ReiserFS was popular in the German distros (Open SUSE, etc.), now we can meet it rarely
- XFS, JFS, . . . file systems for servers
- SquashFS file system for removable media with Linux installed (Live distributions)
- `vfat` for FAT32 (most USB flash disks) or FAT16 with VFAT
- `ntfs` compatible with NTFS, usually the `ntfs-3g` driver or similar
- `iso9660` for CDs, the same as CDFS for Windows
- UDF for DVDs
- `procfs`, `sysfs`, `tmpfs`, `ramfs`, `devfs`, `udev` virtual file systems for various purposes, connected to VFS
- FUSE module for file systems in user space
- NFS network file system
- `swap` file system for the swap partition

 **Remark**


At UNIX systems, we say that “everything is a file”, so directories, devices, sockets, etc. are handled as files.



9.4.2 File Database Structure

All UNIX/Linux native file systems use the same type of the file database structure, originating from the first UNIX file system.

Each volume with this file systems is divided into *blocks* (the same as clusters in Windows) whose size can be predetermined (usually 4096 B).

 **i-node.** Each file has its information structure called *i-node*, and its number, unique in the file system (volume), called *i-node number*.

The i-node contains important file information (owner ID, associated group ID, ACLs, file length, number of hard links, create time, last write time, last access time, . . .), and 15 links to blocks. From these links

- 12 blocks store file data (level 1),
- 13th block may contain links to blocks that store file data (level 2),
- 14th block may contain links to blocks containing links to blocks that store file data (level 3),
- 15th block may contain links to blocks containing links to blocks containing links to blocks that store file data (level 4).

The file uses blocks only till the level that is sufficient. Figure 9.5 shows a shortened structure of links in an i-node.

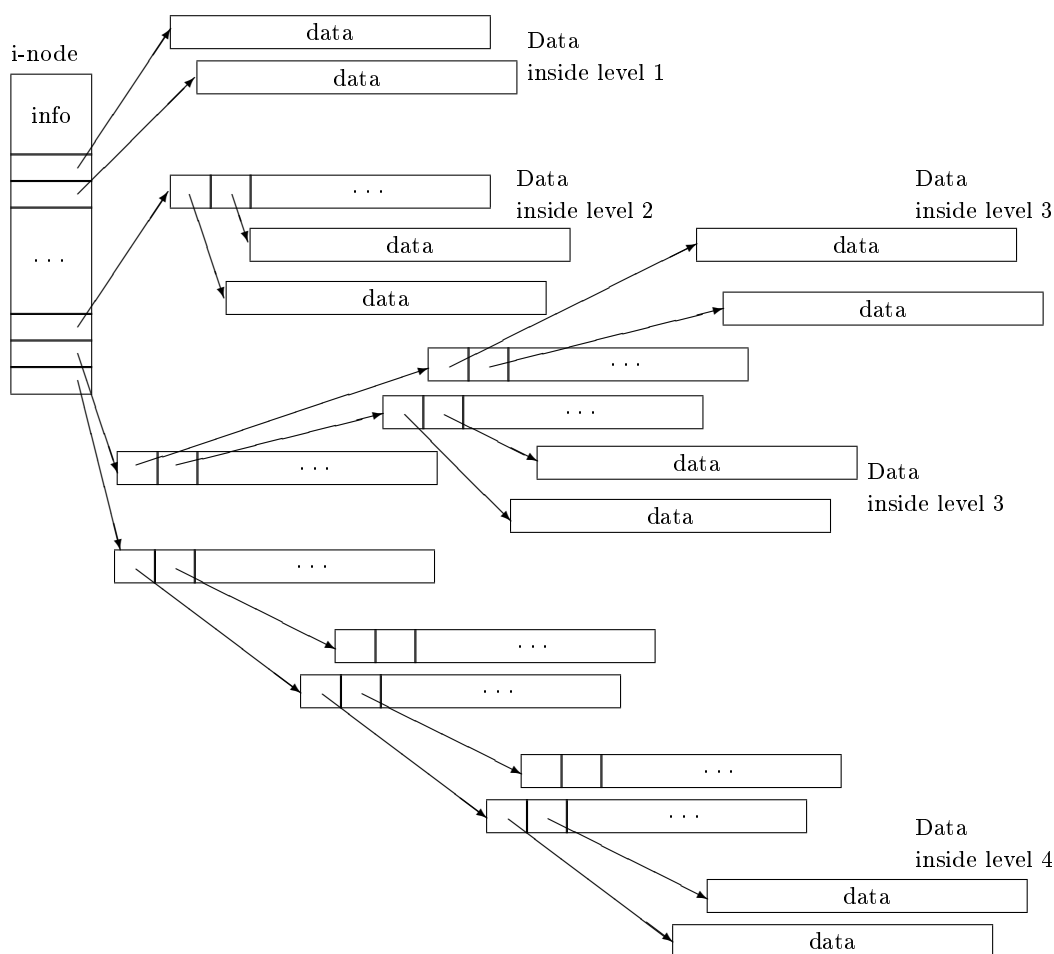


Figure 9.5: i-node structure



Example


Suppose that address length is 32 bits (4 B), the block length is 1024 B (1 KiB) for simplicity. Let us look to the possible limits.

- level 1 is sufficient for files up to 12 288 B ($12 \cdot 1024$ B), i.e. 12 KiB, only 1–12 blocks are allocated,
- level 2 is sufficient for files up to 12 KiB + $256 \cdot 1024$ B = 268 KiB,
- level 3 is sufficient for files up to 268 KiB + $256 \cdot 256 \cdot 1024$ B = 65 804 KB = 64 MB + 268 KB,

- level 4 is sufficient for files up to $65\,804\text{ KiB} + 256 \cdot 256 \cdot 256 \cdot 1024\text{ B} = 16\text{ GB} + 64\text{ MB} + 268\text{ KB}$.

This example is only illustrative, in fact, this structure is a bit more complicated and of course it can be (and probably will be) a larger block size than 1024 B.



 **Volume structure.** Each volume is divided into blocks. The first of these blocks, the *bootblock*, contains a boot loader for the system volume.

Other blocks are in *groups of blocks*. Each group contains a special block, called *superblock*, with information about the file system as a whole (for example, the size of the file system, the number of i-nodes – see below, number of blocks, ...), the group description block follows, then the other management blocks with the usage of blocks and i-nodes, i-node table, and then data blocks.

The fact that important system information is present in each group, and thus actually backed up, not only makes the system work more efficient, but also makes it safer.

Table 9.7 shows a simple structure with the block length 1024 B.


Begin (block #)	Number of blocks	Comment
0	1	boot block
group of blocks 0		
1	1	superblock
2	1	group description
3	1	the bitmap of the used blocks of the group (1 bit for each block; free blocks have = 0)
4	1	the bitmap of the used i-nodes of the group, the bit of a particular i-node can be found by its index in the i-node table
5	214	i-node table; the i-node structures are placed here, and each i-node is uniquely identified by index in this table
219	7974	blocks wit data
group of blocks 1		
8193	1	superblock – backup
8194	1	group description
8195	1	the bitmap of the used blocks of the group
8196	1	the bitmap of the used i-nodes of the group
8197	214	i-node table
8408	7974	blocks wit data
group of blocks 2		
16385	1	superblock – backup
16386	1	group description
...		

Table 9.7: Volume structure

As can be seen, some parts of a group of blocks are optional (including the “bitmaps”, they only fasten searching). Whether a part is present in a group and at where can be found for a volume, it can

be found in the description of the group of blocks (after the super block), and the position of all groups can be found in the superblock.

The *Free space* is recorded in a chain list whose structure is similar to i-nodes. In one of the blocks of a block group, there is an array whose elements refer to free blocks; if the number of these blocks is above the array's capacity, then one record of that array points to a block that contains links to free blocks, ...

 **Directory structure.** Every directory can contain files or other directories. Directories are files that contain a list of records. Each record contains an i-node number, record length, file name, and file length. The records are of variable length to use long file names – if we had a fixed length of record, there would be a lot of free space in the memory. Directories, like any other structure on a volume, are also seen as a file, so they have their own i-nodes and can be spread across multiple blocks just like other files.

9.4.3 Hard Links and Soft Links


All common UNIX file systems allow use of hard links and symbolic (soft) links.

For a *hard link*, several file names can be associated with a single i-node and thus all lead to the same physical file. In each i-node, there is information about the number of hard links, and this number is decremented when the file is deleted in one of the positions (thus, one hard link is destroyed); the file is physically deleted only when this number drops to 0, i.e. when all links are already deleted. All hard links leading to one file are of equal importance, none of them is the main one.

Hard links have some limitations, which are primarily intended to ensure that a cycle does not occur in the directory structure graph: a hard link must not point to a directory other than itself and the parent directory (that is, the allowed links to directories are `.` and `..`), it must also not point to objects that are in another file system (such as other volumes).

Soft links contain location of a file to which they refer. The advantage is the removal of limits imposed on hard links, the soft link can point to any node in the directory structure, including nodes inside other file systems.

Their disadvantage is that when the source of the soft link is deleted, using the corresponding soft link will throw an error.

 The both types of links can be created in GUI, and there is one command for the text mode to work with them:

```
ln sourcefile [targetfile | target_directory]
    creates a hard link, this link increases the number of hard links in the i-node structure of the file
ln /dir1/dir2/file.pdf .    creates a hard link in the current working directory (but the "dot"
    is not necessary; when no target directory is set, the current directory is used as the target)
ln first.txt second.txt third.txt /home/someuser    three hard links are created inside the
    home directory of the stated user
ln first.txt second.txt third.txt ~    the created hard links are in the home directory of the
    writing user
ln -s sourcefile [target]
    creates a soft link
```

`ln -s /etc/X11/xinit/xinitrc.xfce ~/.xinitrc` it changes the desktop environment (GUI) automatically started after the X Window start, to XFce environment (the script `.xinitrc` in the home directory of a user determines the desktop environment for this user, and the file `xinitrc.xfce` is the start file for XFce)



Example

We will create a new file `test.xxx` in the home directory and then a hard link and a soft link to this file.

```
sarka@sarka-Vostro ~$ touch test.xxx
sarka@sarka-Vostro ~$ ln test.xxx testhard.xxx
sarka@sarka-Vostro ~$ ln -s test.xxx testsym.xxx
```

The i-node numbers are displayed when using the `-i` option:

```
sarka@sarka-Vostro ~$ ls -li test*.*

1841021 -rw-r--r-- 2 sarka sarka 0 Apr 15 13:15 testhard.xxx
1843248 lrwxrwxrwx 1 sarka sarka 8 Apr 15 13:16 testsym.xxx -> test.xxx
1841021 -rw-r--r-- 2 sarka sarka 0 Apr 15 13:15 test.xxx
```

The first column is the i-node number. As we can see, the hard link has the same i-node number as the original file, the soft link has different i-node number. The length of the hard link is the same as original (the `touch` command creates an empty file), but the soft link has the length 8 Bytes.


The second column begins with one symbol determining the type of file. For common files, the symbol is “-”, for soft links, the symbol is “l”.

The third column holds number of hard links, the `testhard.xxx` file has the same value as the original file, because this number is stored in the i-node (and the both files have the same i-node).


The names of these files are different (may be the same if they are in different directories), because this information is not in the i-node, it is in the directory record (different for various paths to file = hard links).




9.4.4 File Systems of the Type `ext x fs`

 **ext2fs** All that was written above applies to `ext2fs`. This file system is reported to be usable for partitions of up to 4 TiB. It supports long file names (up to 255 characters, but this limit can be moved further if needed). However, this file system is practically no longer in use today, using its successors `ext3fs`, `ext4fs`.

It only makes sense where speed is more important than maintaining data consistency when it changes, because it is a bit faster than `ext3fs` (that is, for directories whose content is changed rarely but often accessed or accessed for a long time interval, such as the `/boot` partition). The reason for the greater speed is that no journal file is used.

 **ext3fs** is an `ext2fs` enhancement, it is 32-bit file system. It is backward compatible (more precisely: compatible in both directions), preserves all `ext2` structures, but is also a journaling file system. If we have an `ext2` file system on the volume, just create a journaling file and when rebooting the system, we


can mount the partition as `ext3`, and vice versa, if we have a volume defined as `ext3`, we can mount it as `ext2` at the next system boot.


 **ext4fs** is the newer version, it is 64-bit file system. It is also backward compatible with certain limitations. Among other things, `ext3` has the following features:

- the limits valid for `ext3fs` are increased for 64-bit addresses (thus, for 64-bit operating systems),
- the time stamps in the journal file are more accurate (1 ns),
- is more friendly to flash memory (including SSDs),
- it is possible to use *extents* – extent is a collection of multiple consecutive blocks; instead of a pointer to a data block, a pointer to extent can be used (resulting in the ability to store larger files with less fragmentation).

The `ext4fs` can be mounted as `ext3fs`, if no extents are used.


9.4.5 Other Journaling File Systems

 **ReiserFS** is a journalizing file system based on the ballanced tree, which speeds up work with a large number of files in a directory. Another great feature is that it is possible to store a few small files (or the rests of large files that do not fit in whole blocks) into one block, so there are no unreachable holes in this file system. The disadvantage of this property is the ability to reduce system performance, which partially improves the file system with various techniques used in database systems. However, it is a good choice for a system where we work primarily with very small files.

 **XFS** is a journalizing file system where only metadata are journaling. This increases the file system throughput, but it is also the reason that this file system is unsuitable for deployment on machines with frequently modified data. XFS is very suitable for those servers where data is mainly read and not often modified.


It is a 64-bit file system, optimized for working with large files, while working with small files is not optimal.

It has many interesting features, one of them is the “realtime subvolume”, which allows processes to reserve an access band at a certain width (Bytes per second) to a file. This is very practical, for example, when working with multimedia, where we need constant and fast access to a file (such as video).

 **BtrFS** (B-tree File System) is one of the latest file systems by Oracle, designed especially for servers running on Linux. While it is still in development, it is already part of the Linux kernel of some distributions.

Compared to ordinary Linux filesystems, the support for features that are valued mainly on servers is added – management without unmounting (including defragmentation, balancing – this also suggests a name, quotas, etc.), creating a snapshot image without unmounting (using redundant file creation capability), which can be used to create backups, native RAID 0, 1 and 10 support, use of checksums, transparent compression, etc. Other features are planned, including encryption. The I/O operation is optimized using a B-tree, after which it is also named.

BtrFS is considered an alternative to ZFS from Sun (ZFS is distributed under the CDFS license, which is incompatible with the GNU GPL, so ZFS support cannot be implemented directly into the Linux kernel).

 **SquashFS** is a compressed read-only file system, it compresses all – the files, directories, i-nodes with keeping good access times. It is used mainly for Live Linux distributions installed at removable media (Live distros run from removable media, not from a harddisk volume), or it is also intended for disks with backups.

9.4.6 Comparison of Linux File Systems

It cannot be said which of these filesystems is better or worse, each has its advantages and disadvantages. Journaling may be an advantage, but it may not (or might) reduce system performance.

In the following table, there is a comparison of the systems according to various criteria:


	ext2fs	ext3fs	ext4fs	ReiserFS	XFS
Maximum volume size	4 TiB	4 TiB	1 EiB	16 TiB *)	18*210 PiB
Block size	1–4 KiB	1–4 KiB	4 KiB	up to 64 KiB	512 B – 64 KiB
Maximum file size	2 GiB	2 GiB	16 TiB	up to 210 PiB *)	9*210 PiB

*) Depending the file system version.

Table 9.8: Comparison of some Linux file systems

9.4.7 Virtual File Systems

In Linux, as well as in other UNIX systems, virtual (pseudo-) file systems are also used. Besides the VFS file system, we can meet the following virtual file systems:

 **procfs** makes system and process runtime information available (used in Linux). Some files can be written to change system behavior at runtime. It does not match any physical data medium, it is mounted to the `/proc` directory. Its subdirectories whose names are PIDs of all running processes provide runtime information about processes with the PID in the directory name. E.g. these pseudo-files and directories can be found inside subdirectories for processes:

- **exe** the soft link leading to the executable file – the image of the process,
- **cmdline** each string of command line of the process, including options and parameters,
- **status** the file with detailed information about a process (state, memory usage, number of threads, affinity – allowed processes, etc.),
- **net** directory with information about communication of the process through network.

In the case of the whole system (i.e. directly in the `/proc` directory), we also find the file **cmdline**, which contains the command that started the system (including parameters) and also the subdirectory **net** with network information. There are other interesting items in the system part of the filesystem, such as:

- **cpuinfo** the processor information,
- **meminfo** memory usage information,
- **modules** the list of modules loaded in kernel (drivers, antivirus, network protocols, codecs, etc.),
- **version** the Linux kernel version and the distro version,
- **ioports**, **iomem**, **dma**, **interrupts** information about low-level hardware communication, firmware addresses for devices, DMA channels, interrupts, etc.

**Remark**

Do not attempt to open files from the `/proc` directory in a text editor. Although they seem as text files (not all, for example, `exe` is a soft link), note that they are not physical files – they are only interfaces. Text editors have a habit of locking the file open, but it is not possible in the `proc` file system. So you only get an error message. However, in text mode, it is not a problem to use the command command e.g. `cat /proc/cpuinfo`.



sysfs is based on a similar principle to `procfs`, used to make device information available (in Linux), it contains a device run-time information. It accesses data in the `/sys` directory.



devfs and udev are virtual file systems that manage special device files stored in the `/dev`. The Linux kernel uses the `udev` module, and in the later versions, this module also manages the `sysfs` subsystem drivers. The older static `devfs` is still used in MacOS.

These two virtual file systems maintain the device special files structure inside the `/dev` directory. Each device (including virtual devices) is identified by its special file in the user space, and by the vector of two numbers – the major and minor number. The kernel works with the numbers only, users work with special files.

**Example**

The following command lists all the information:

```
ls -l /dev

drwxr-xr-x 2 root root      420 Apr 15 17:07 block
lrwxrwxrwx 1 root root         3 Apr 15 17:07 dvd -> sr0
crw-rw-rw- 1 root root    1, 3 Apr 15 17:07 null
crw-rw-rw- 1 root root    1, 8 Apr 15 17:07 random
brw-rw---- 1 root disk    8, 0 Apr 15 17:07 sda
brw-rw---- 1 root disk    8, 1 Apr 15 17:07 sda1
brw-rw---- 1 root disk    8, 2 Apr 15 17:07 sda2
brw-rw---- 1 root disk    8, 16 Apr 15 17:07 sdb
brw-rw---- 1 root cdrom 11, 0 Apr 15 17:07 sr0
crw-rw-rw- 1 root tty     5, 0 Apr 15 17:07 tty
crw--w---- 1 root tty     4, 0 Apr 15 17:07 tty0
crw--w---- 1 root tty     4, 1 Apr 15 17:07 tty1
crw-rw-rw- 1 root root    1, 5 Apr 15 17:07 zero
```

(the output is shortened). The first letter in each row is the file type. The letter “d” indicates a directory, the letter “l” indicates the soft link. These rows have only one number inside the 5th column (420 and 3). This number is the file length in Bytes.

The letters “b” and “c” mean block and character devices. These rows have two numbers inside the 5th column, e.g. the `null` device has 1, 3 here. These two numbers represent the major and minor number of the device.

As we can see, all listed virtual devices (`null`, `random`, `zero`) have the same major number, because the number 1 represents the class of virtual devices. The block devices being disks and partitions (`sda`, `sdb`, `sda1`, ...) have the major number 8, and various minor numbers.


When listing the contents of the `block` directory, we find that it contains soft links to special files of all block devices named with the device numbers (`major_number:minor_number`).

```
ls -l /dev/block

lrwxrwxrwx 1 root root 6 Apr 15 17:07 11:0 -> ../sr0
...
lrwxrwxrwx 1 root root 6 Apr 15 17:07 8:0 -> ../sda
lrwxrwxrwx 1 root root 7 Apr 15 17:07 8:1 -> ../sda1
lrwxrwxrwx 1 root root 6 Apr 15 17:07 8:16 -> ../sdb
...
```


Similar output can be obtained using the command `lsblk` (listing information about all block devices).




 **ramfs, tmpfs:** the `ramfs` file system is used to implement RAMdisk (i.e. part of the memory will be used in the same way as a partition; the advantage is great speed, the disadvantage is that content is not preserved after shutdown or reboot). The `tmpfs` is something similar – it creates a simulated temporary data partition in memory (the advantage is that for temporary data does not matter when it is lost after shutting down or rebooting).


The `tmpfs` is also used to implement the `/run` system. This directory is used during the system start, when no partitions are mounted (even the main partitions), and the system or processes need to write data somewhere.

9.4.8 The `fstab` File

 The `/etc/fstab` file (file systems table) contains the list of the *mountable file systems* – belonging to volumes, mainly mounted during the booting process, and possibly those that can be mounted later (e.g. removable media or the volumes of another operating systems). If the item in this file is tagged for automatic mounting, the volume is mounted automatically during the system boot process.

 The `/etc/mtab` file (mounted file systems table) contains the list of the *currently mounted file systems*. It is much longer, because all virtual file systems and the removable media are present in this file.

First, we look at the syntax of entries in the `/etc/fstab` file. Each row corresponds to one volume (or removable media), the syntax is the following:

 `volume mount_point file_system parameters dump FS_check`

A row consists of:

1. `volume`: determines the device to mount. It can be a special file, or the UUID number (Universally Unique ID, a type of GUID).
2. `mount_point`: the directory specified as the mount point. the system partition (where Linux is installed) has its mount point named “/”, another important directories may be in their own volume and so they have the own row in this file. The removable media usually use a directory inside `/media` or `/mnt`.

The directory listed here must exist.

3. `file_system`: determines the file system of the volume, e.g. `ext4fs`, `udf`, `vfat`,... , for the swap partition there is a value `swap` here.

If the value `auto` is here, it means that the file system is automatically recognized during mounting. It is common for removable media.

4. **Parameters.** Each file system has its own parameters, some of them are in all common file systems. The most important are:

- **ro** (read-only), **rw** (writing is allowed),
- **user** (mounting is allowed to each user, but unmounting can be performed only by the same user), **users** (mounting by everybody, unmounting by everybody),
- **noauto** (this volume is not mounted automatically during booting process, but the mounting process must be carried out manually),
- **exec** (permission to run a program on this volume), **noexec** (prohibition to run a program on this volume, highly recommended for data volumes),
- **nodev** (no files will be handled as special files of devices; for security reasons, it is recommended for all partitions where there no special files supposed here),
- **sync** (this volume is immediately synchronized, without cache),
- **acl,user_xattr** (ACLs are used for this volume),
- **noatime** (the last access time stamp is not updated, it can speed up work with the volume and it is better for SSDs).

Several parameters are intended only for selected file systems. Details can be found in manual pages – `man fstab`, `man mount`.

5. **dump.** If there is the digit 1, this volume is backed up.

6. **check.** If 1 or 2 is here, this volume is checked at system startup by the `fsck` program. The value 1 means higher priority, it is used for the system volume.

The columns in `/etc/mtab` are similar, only partitions (volumes) are represented by special files, not UUIDs, and `tab` is not used as separator, but space, and the default options and parameters are added.



Example

The `/etc/fstab` file may look like this (using tabs to align it into columns):

```
/dev/sda6      swap          swap          defaults      0 0
/dev/sda7      /             ext3          acl,user_xattr 1 1
/dev/sda8      /home        ext3          acl,user_xattr 1 2
/dev/sda9      /usr         ext3          acl,user_xattr 1 2
/dev/sda2      /mnt/winC    ntfs-3g      users,gid=users,umask=133,dm=022 0 0
/dev/sda5      /mnt/winD    ntfs-3g      users,gid=users,umask=133,dm=022 0 0
proc          /proc        proc          defaults      0 0
sysfs         /sys         sysfs        noauto        0 0
debugfs       /sys/kernel/debug debugfs      noauto        0 0
usbfs         /proc/bus/usb usbfs         noauto        0 0
devpts        /dev/pts     devpts       mode=0620,gid=5 0 0
```

In newer distributions, we are more likely to see the UUID instead of the device file (the first column).

The following contents is in `/etc/mtab`, corresponding to the above stated `/etc/fstab`:


```
/dev/sda7 / ext3 rw,acl,user_xattr 0 0
/proc /proc proc rw 0 0
sysfs /sys sysfs rw 0 0
debugfs /sys/kernel/debug debugfs rw 0 0
udev /dev tmpfs rw 0 0
devpts /dev/pts devpts rw,mode=0620,gid=5 0 0
/dev/sda8 /home ext3 rw,acl,user_xattr 0 0
/dev/sda9 /usr ext3 rw,acl,user_xattr 0 0
/dev/sda2 /mnt/winC fuseblk rw,noexec,nosuid,nodev,allow_other,default_permissions,blksize=4096 0 0
/dev/sda5 /mnt/winD fuseblk rw,noexec,nosuid,nodev,allow_other,default_permissions,blksize=4096 0 0
```



```
fusectl /sys/fs/fuse/connections fusectl rw 0 0
securityfs /sys/kernel/security securityfs rw 0 0
none /proc/sys/fs/binfmt_misc binfmt_misc rw 0 0
/dev/sdb1 /media/disk vfat rw,nosuid,nodev,noatime,flush,uid=1000,utf8,shortname=lower 0 0
```

The last row is for the currently connected removable media (USB flash disk with the FAT32/VFAT file system).



 We use the `mount` and `umount` commands to mount and unmount a partition or removable media (which is usually a single partition).

Example

Examples of mounting and unmounting filesystems (partitions):

`mount` (with no options) lists all mounted volumes, similar as `cat /etc/mtab`

`mount /mnt/winD` if this mount point is in `/etc/fstab`, the mount point name is enough to mount this volume

`mount -o ro /mnt/winD` the same, but the volume is mounted for read-only access

`mount -l -t fstype` lists all volumes with the given file system type

`mount -t vfat /dev/sdb2 /media/flash` we mount the USB flash disk with the VFAT file system (FAT32), we enter the both the special file and the mount point

`umount /mnt/winD` unmounting the volume with the given mount point

`mount -t ntfs-3g /dev/sda1 /mnt/winC` the mount point is not in `/etc/fstab`, and it is a Windows partition with the NTFS file system, we use the `ntfs-3g` driver

`ntfs-3g /dev/sda1 /mnt/winC` the same result, we use the control program `pro` the given driver




Example

Let us return to the UUIDs used to mark partitions instead of special files. If we are not sure which special file UUID belongs to, we can display the relationships with the command `blkid`. Its output is similar to the following:

```
...
/dev/sda4: UUID="d4f25949-0d2..." TYPE="ext4"
/dev/sda5: LABEL="DATA" UUID="1F0801CAA09..." TYPE="ntfs"
...
/dev/sdb1: LABEL="TRANSCEND" UUID="B49B-633..." TYPE="vfat"
```



In ordinary Linux distributions, we do not have to worry about connecting removable media, they are connected automatically. Other operating system partitions (such as Windows) are usually not automatically connected, but there is a different problem: Windows 10 usually uses a “fast startup” mode, ie the kernel only hibernates when shutting down (so you must really restart it for updates). However, the hibernation system makes its partitions inaccessible to any other system, so it cannot be mounted in Linux.

 **Mounting disk images.** We do not only need to mount (physical) media, we can also mount disk images, such as ISO, DD or IMG. Then they can be accessed in the same way as any other media.

**Example**

A disk image (more exactly a partition image) can be mounted by creating a loop to the given media:

```
mkdir /media/MyMountDir ..... creating a mount point
mount -o loop somefile.iso /media/MyMountDir ..... creating a loop
```

And now, this media can be accessed in the same way as another (real) partitions:

```
ls /media/MyMountDir
cd /media/MyMountDir
```

**9.4.9 Handling Partitions and File Systems**

Partitions can be managed either in GUI, or using `fdisk` (for MBR disks), `gdisk` (for GPT disks) or similar program. These two programs are interactive (various keys of keyboard have some meaning), `sfdisk` and `sgdisk` are fully command-line tools (using parameters of commands), `cmdisk` and `cgdisk` run in the pseudographic mode with menus.

**Example**

The `fdisk` command needs higher access privileges. It can list information about disks, or go to the interactive mode.

```
fdisk -l /dev/sda lists info about the given disk (its partitions, including file systems)
```

```
fdisk -l lists info about all available disks
```

```
fdisk /dev/sda going over to the interactive mode, we want to configure the disk specified in the
parameter
```

In the interactive mode, we use the “letter commands” listed in Table 9.9.





Key	Meaning
	help
	lists the disk partitions
	creates a new partition
	deletes the selected partition

Table 9.9: Keyboard commands for the interactive mode of `fdisk`

The `M` key is the most important: if we don’t know what to do, we simply press the `M` key.



The `df` command (Disk Free) lists free space on individual volumes. Using options, we can determine in more detail what and how to list.

```
df basic information
```

```
df -T also the file system type (if FUSE is used, we get only the information “fuseblk”)
```

```
df -Th in addition, the free space is displayed in the human readable format (kB, MB, GB, etc.)
```

```
df -i number of i-nodes instead “common” units (corresponding to the number of files and directories)
```

```
df -a all mounted file systems, including most virtual file systems
```

**Example**

The `df -ah` command has the following output (laptop with dualboot Windows 7 and SUSE Linux, old installation):

Filesystem	Size	Used	Avail	Use%	Mounded on
/dev/sda7	20G	3.8G	16G	20%	/
/proc	0	0	0	-	/proc
sysfs	0	0	0	-	/sys
debugfs	0	0	0	-	/sys/kernel/debug
udev	2.0G	192K	2.0G	1%	/dev
devpts	0	0	0	-	/dev/pts
/dev/sda8	40G	188M	38G	1%	/home
/dev/sda9	60G	5.1G	51G	10%	/usr
/dev/sda2	117G	30G	87G	26%	/mnt/winC
/dev/sda5	168G	256M	168G	1%	/mnt/winD
fusectl	0	0	0	-	/sys/fs/fuse/connections
securityfs	0	0	0	-	/sys/kernel/security
none	0	0	0	-	/proc/sys/fs/binfmt_misc



The `du` command (Disk Usage) lists the occupied space for a particular file/directory, recursively.

`du` lists only directories in the current directory, recursively, occupied space is in the number of blocks

`du -a` similar, but including files

`du -h somefile.pdf` occupied space for the given file, in the human readable unit

`du -h `ls *.txt`` for all text files in the current directory (`du` is not able to work with filters)



the `lsof` command (LiSt Open Files) lists open files, and processes working with these files.

`lsof` all open files, this output is very long

`lsof > ~/listofopenfiles.txt` redirection to the given file

`lsof | grep "\.so[0-9\.]$" > ~/used-libraries.txt` list of all used libraries (the `.SO` files are for Linux the same as `.DLL` for Windows)

`lsof ~` list of processes working with our home directory

`lsof -u someuser` list of files used by processes of the given user (the user can be entered by its UID or its name)

`lsof | grep pipe` list of all processes communicating through an anonymous pipe

`lsof | grep socket` the same for anonymous sockets

Bibliography

- [Silberschatz2013] SILBERSCHATZ, Abraham, Peter B. GALVIN and Greg GAGNE. *Operating system concepts*. Ninth edition. Hoboken, NJ: Wiley, [2013]. ISBN 978-1-118-06333-0.
WWW: <http://iips.icci.edu.iq/images/exam/Abraham-Silberschatz-Operating-System-Concepts—9th2012.12.pdf>
- [Tanenbaum2006] TANENBAUM, Andrew S. and Albert S. WOODHULL. *Operating systems: design and implementation*. 3rd ed. Upper Saddle River, N.J.: Pearson/Prentice Hall, c2006. ISBN 978-0-13-142938-3.
WWW: https://mcdu.files.wordpress.com/2017/03/tanenbaum_woodhull_operating-systems-design-implementation-3rd-edition.pdf
- [Deitel2004] DEITEL, H.M., P.J. DEITEL and D.R. CHOFFNES. *Operating Systems*. Third Edition. Upper Saddle River, NJ: Pearson, Prentice Hall, 2004. ISBN 0-13-124696-8.
WWW: <http://202.74.245.22:8080/xmlui/bitstream/handle/123456789/629/Operating%20systems%20%28Deitel%29%20%283rd%20edition%29%281%29.pdf>
- [Fox2015] FOX, Richard. *LINUX with operating system concepts*. Boca Raton: CRC Press, [2015]. ISBN 978-1-4822-3589-0.
Most pages on: <https://books.google.cz/books?id=VG8LBAAAQBAJ&printsec=frontcover>
- [Bovet2006] BOVET, Daniel P. and Marco CESATI. *Understanding the Linux kernel*. 3rd ed. Sebastopol, CA: O'Reilly, c2006. ISBN 978-0-596-00565-8. WWW: <https://books.google.cz/books?id=h0lltXyJ8aIC&printsec=frontcover>
- [ProcLib] *Process Library* [online]. [2019-01-25]. WWW: <https://www.processlibrary.com>
- [TLDLP] *The Linux Documentation Project* [online]. [2019-01-25]. WWW: <http://www.tldp.org>
- [Kernel.org] *The Linux Kernel Archives* [online]. [2019-01-25]. WWW: <https://www.kernel.org/>
- [WinDocs] *Windows Documentation: Microsoft Docs* [online]. [2019-02-07].
WWW: <https://docs.microsoft.com/en-us/windows/>
- [AVTest] *AV-TEST: Antivirus and Security Software and Antimalware Reviews* [online]. [2019-01-25].
WWW: <https://www.av-test.org/>

- [AVComparatives] *AV-Comparatives: Independent Tests of Anti-virus Software* [online]. [2019-01-25].
WWW: <https://www.av-comparatives.org/>
- [Sysinternals] *Windows Sysinternals* [online]. [2019-01-25]. WWW: <https://sysinternals.com>
- [Holmes2013] HOLMES, Lee. *Windows PowerShell cookbook: the complete guide to scripting Microsoft's command shell*. 3rd ed. Farnham: O'Reilly, 2013. ISBN 14-493-2068-6.
WWW: http://rsmt.it.fmi.uni-sofia.bg/books/windows_powershell_cookbook_3rd_edition.pdf
- [Negus2015] NEGUS, Chris. *Linux bible*. Ninth edition. Indianapolis, Indiana: John Wiley, 2015. ISBN 978-111-8999-875.
Most pages on: <https://books.google.cz/books?id=c-zGBwAAQBAJ&printsec=frontcover>
- [McHoes2011] MCHOES, Ann McIver a Ida M. FLYNN. *Understanding operating systems*. 8th edition. Clifton Park, NY: Cengage Learning, 2011. ISBN 978-1-4390-7920-1.
WWW: <http://160592857366.free.fr/joe/ebooks/ShareData/Understanding%20Operating%20Systems%20e%20By%20Ann%20McIver%20McHoes%20and%20Ida%20M.%20Flynn.pdf>
- [Panek2017] PANEK, William. *Mcsa: windows 10 complete study guide*. Indianapolis, IN: John Wiley, 2017. ISBN 978-1-119-38496-0.
Most pages on: <https://books.google.cz/books?id=U80yDAAAQBAJ&printsec=frontcover>
- [Sarwar2016] SARWAR, Syed Mansoor and Robert KORETSKY. *UNIX: the textbook*. Third edition. Boca Raton: Taylor, 2016. ISBN 978-1-4822-33-582.
Most pages on: <https://books.google.cz/books?id=NzuLDQAAQBAJ&printsec=frontcover>