

Operating Systems

Dan Cosma

1

Preliminaries

2

Audience and impact

3

Who is this course for

→ *3rd year undergraduate students
and all other people interested in
understanding the main concepts
involved in the modern operating
systems*

4

Objectives and targeted abilities

- ▷ using operating systems *at a professional software engineering level*
- ▷ understanding the fundamental concepts fondamentale associated with operating systems, *with the explicit focus on developing modern software applications*
- ▷ using in programs, at an advance level, the services provided by the operating system and the associated libraries
 - ↳ *system programming*

5

and, in other words, this course may help building a solid career

→ → →

6



7

Operating Systems course structure

8

Course

- **14 weeks, 2 hours each**

- presenting the fundamental issues, explanations, examples

- **interactivity**

- discussions, analyses, problems, answers -- please do ask questions!

- **feedback**

- suggestions, observations, complaints -- all are welcome (honestly!) — they can lead to a better course

9

Laboratory

- **individual lab assignments**

- capture the essential practical aspects,
 - introduce fundamental concepts for software development,
 - help you become professional software engineers

- **interactivity**

- discussions, analyses, problems, answers -- please do ask questions!

- **feedback**

- suggestions, observations, complaints -- all are welcome (honestly!) — they can lead to better communication and better lab support

10

Evaluation

- **3 tests at the lab**

- small yet complete programs, that require about an hour of work
 - test the student's abilities regarding the system programming and/or advanced OS usage, as they are developed at the time of the test
 - define the lab grade (weighted mean of the test grades)
 - count as 35% of the final grade

- **exam**

- a set of questions evaluating the understanding of the studied concepts and techniques
 - a practical component, evaluating the abilities/knowledge gained during the labs
 - 65% of the final grade

- **feedback**

- suggestions, observations, complaints -- all are welcome — they can lead to better evaluation methods and the detection of errors or problems

11

Feedback

- **e-mail: dan dot cosma at cs dot upt dot ro**

- **during the lectures or the lab classes**

12

Resources

- Course site

- integrated in our "LOOSE" software engineering portal
- <http://loose.upt.ro/~oose/pmwiki.php/OS/OperatingSystems>

- Lab site

- all the necessary lab materials
- same address as the course

13

Bibliography

1. W.R.Stevens, S.A.Rago, *Advanced Programming in the UNIX Environment, Third Edition*; Addison Wesley, 2013
2. W. Stallings, *Operating Systems: Internals and Design Principles, 7th edition*, Prentice Hall, 2011
3. Eric S. Raymond: *The Art of UNIX Programming*, Addison-Wesley, 2003
4. A. Robbins: *UNIX in a Nutshell, Fourth Edition*; O'Reilly, 2005. Ioan Jurca: *Programarea de sistem in UNIX*, Editura de Vest, Timisoara. 2005
6. A. S. Tannenbaum: *Modern Operating Systems, 2nd Edition*, Prentice Hall, 2001

14

1. Introduction

15

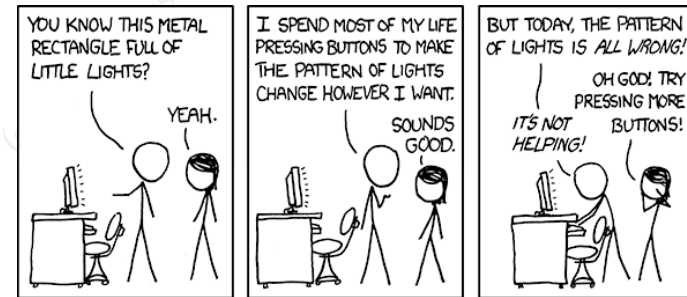
What is an operating system?

16

Computers are complex machines...



17

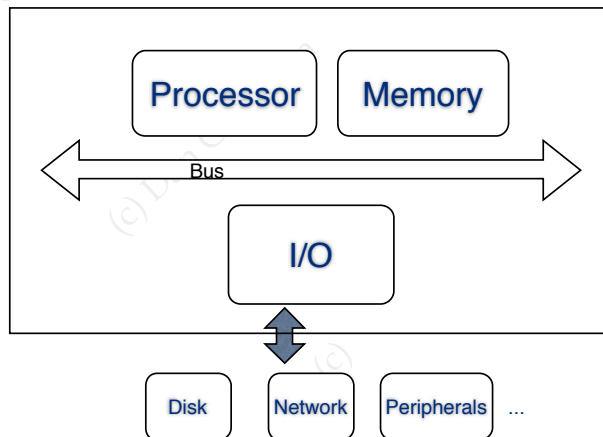


<http://xkcd.com/722/>

... which have to be used efficiently

18

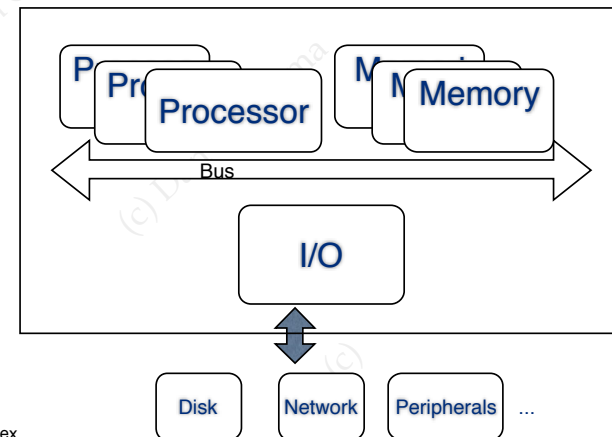
Computer



(the "classic" architecture)

19

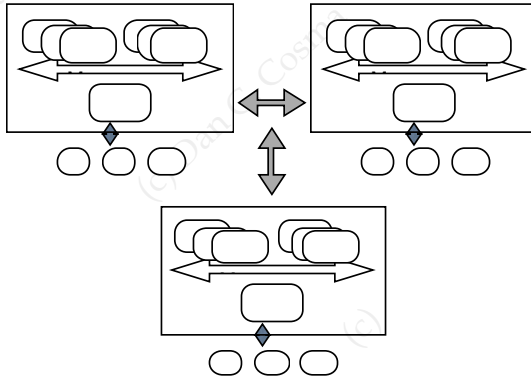
Computer



(a more complex architecture)

20

Computer



(an even more complex architecture)

21

Computer



(... anything else)

22



... we need mechanisms to make them more approachable
 → in usage
 → in software development

23

Operating System

A set of programs that:

- manage the hardware resources
- create high-level abstractions for resources
- control the execution of applications
- provide an interface to the applications
- provide an interface to the user

depending on the different variants of systems, some of the above roles may be assigned to applications, and lay outside the OS (e.g., the user interface)

24

Operating system goals [2]

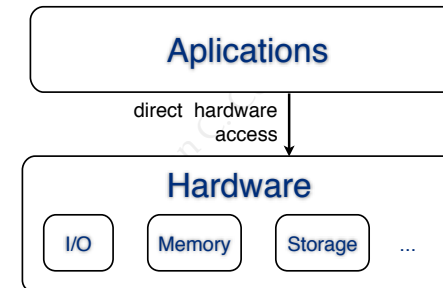
- **Ease of use**
→ to facilitate the access to resources
- **Efficiency**
→ in how resources are used and managed
- **Ability to evolve**
→ the capacity of adding new functionalities, without affecting the services provided by the OS

[2] W. Stallings, *Operating Systems: Internals and Design Principles*, 7th edition, Prentice Hall, 2011

25

Ease of use. The layered architecture of the software in a computer system

What if...?



26

Ease of use. The layered architecture of the software in a computer system

Direct hardware access

- extremely complex programs
- code duplication (and more) in different apps (e.g., handling data storage formats on a disk)
- too strong dependency on hardware devices
- lack of portability
- vulnerability to the system's evolution in time
- ...

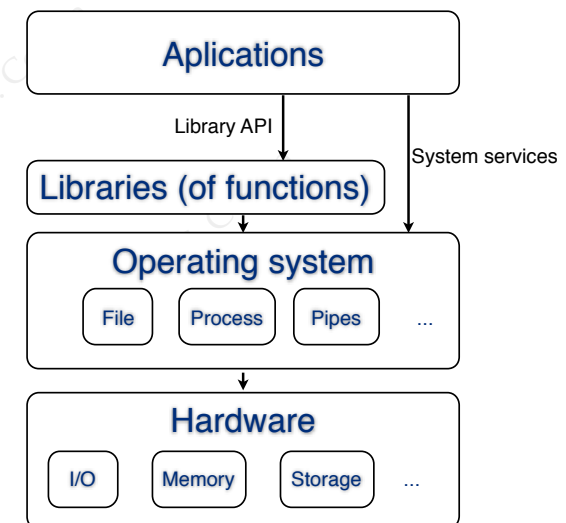
```

-004031C0 837E4000    cmp [esi+48], 00000000
-004031C2 0F4E702000   je 00403000
-004031D4 23FF        xor edi, edi
-004031D6 8D450C      lea eax, [ebp-24]
-004031D9 50          push eax
-004031DB 8B4DF0      mov ecx, [ebp-10]
-004031DD 51          push ecx
-004031DE FF4554      inc [esi+54]
-004031E1 8B4DE8      mov ecx, [ebp-18]
-004031E4 E8B7530000   call 004085A0
-004031E9 097E68      cmp [esi+68], edi
-004031EC 7470       je 00403260
-004031EE 6A00040000   push 00000400
-004031F3 00854FEFFFF lea eax, [ebp+FFFFFF54]
-004031F9 50          push eax
-004031FA 8B4DF0      mov ecx, [ebp-10]
-004031FD 51          push ecx
-004031FE 8B4DE8      mov ecx, [ebp-18]
-00403201 E86A560000   call 00408870
-00403206 85C0       test eax, eax
-00403209 7465       je 00403270

```

27

Ease of use. The layered architecture of the software in a computer system



28

Operating system services

- Program execution and control
- Memory management
- Access to I/O devices
- Simplified and controlled access to data (files etc.)
- Error detection and handling
- Software development tools
- Security, monitoring, synchronization etc.

Using the resources

- efficient management of the processor time
 - through algorithms that properly schedule the entities/programs that run in parallel or sequentially in the system
- efficient management of I/O
 - caching, managing the resources that abstract devices, etc.
- efficient memory management
 - freeing unused memory, swapping, virtual memory management, etc.
- efficient communication between programs
 - fast mechanisms for synchronous and asynchronous communication
- ...

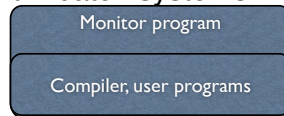
The operating system needs to evolve when

- changes in hardware occur
 - e.g., adding a device (disk, memory stick, printer, ...)
- new services are needed
 - e.g., new network protocols, user-centric improvements, new data storage formats, etc.
- errors must be corrected
 - e.g., solving security problems, fixing bugs
- optimizations are needed
 - examples: faster disk access algorithms, faster UI response, etc.

Modern operating systems natively include advanced mechanisms of update, upgrade, and software package management

A short history of operating systems

- **<1955: Mainframe, no operating system**
→ used sequentially (once at a time), programs read from punch cards or magnetic tapes
- **'50-'60: Mainframe with "batch systems"**
→ the monitor, an "OS" that permanently resides in the memory, allowing the user to launch "jobs"
- **Ca. 1955: Mainframe with dedicated OS-es**



33

- **1969-1971: UNIX**

- Ken Thompson (Bell Labs) starts to work on a new OS, after Bell Laboratories withdraws from the Multics project
- Co-author: Dennis Ritchie, who will also create the C language (1971-1973), to use it to write UNIX (the first UNIX version was written in assembly, and the application in an interpreted language called "B")
- First UNIX version: running on PDP 7 (DEC), 1969-1970 ("UNICS" - "UNiplexed Information and Computing Service")
- starts to be distributed freely, at the source code level

- **1971: UNIX on PDP 11**

- used for text processing within the Patent Department at Bell Labs

- **1972: UNIX reports 10 installations**

- its free distribution makes it extremely popular in industry and academia
- native multitasking, multiuser

34

- **~1971 - the 80's: The Home Computer "revolution"**
→ mass production of microprocessors leads to the first personal computers being introduced
→ the "operating systems": a BASIC interpreter stored in ROM, capable of running programs, providing a simple command-based user interface
→ applications: games, programming languages, interpreters, compilers
→ examples: Apple II, ZX Spectrum, Commodore 64, HC-85 (RO), Tim-S (RO, TM)
- **~1974: CP/M**
→ "Control Program/Monitor" -> "Control Program for Microcomputers"
→ used in business environments, education, microcomputers
→ approx. 5 commands, unifies the services provided to programs (for portability)
→ examples of computers: Altair 8080, Amstrad PCW, CUB-Z (RO)
- **1977: BSD**
→ "Berkeley Software Distribution" / "Berkeley UNIX"
→ developed at University of California, Berkeley, derived from the UNIX sources from Bell Labs
→ today, one of the main open source operating systems

35

- **1980: MS-DOS**

- IBM contracts Microsoft to write an OS for the new personal computers (PC) developed by the company, after a similar discussion with the CP/M creator fails; Microsoft retains the right to sell the MS-DOS system separately from the hardware
- based on QDOS ("Quick and Dirty OS"), a CP/M clone, developed by Tim Paterson, in 6 weeks, for the company he was employed at
- QDOS bought by Bill Gates (Microsoft) with 50 000 \$; the deal with IBM was kept secret by Microsoft at the buying time
- after a year, Tim Paterson is employed by Microsoft
- mono-tasking, CP/M-inspired command-line interface
- first PC generations lacked the hardware capability of running UNIX

- **1980-1990: "The UNIX Wars"**^[3]

- the period when UNIX is exploited commercially
- different UNIX versions successfully compete on the market
- TCP/IP is developed and is adopted by UNIX, first at Berkeley
- selling UNIX eliminates the free circulation of its source code, with a side effect: the vitality of its development is reduced
- different attempts of porting UNIX on i386 fail

^[3] Eric S. Raymond: The Art of UNIX Programming, Addison-Wesley, 2003

36

- **1983: Richard Stallman starts the GNU project**
 - with the goal of creating a free UNIX
 - introduces the GNU General Public License
 - although the resulting UNIX kernel (Hurd) is unsuccessful, the GNU project becomes one of the main promoters of the open source movement
- **1984: Apple Macintosh “System Software”**
 - ran on Apple Macintosh 128K (the first Apple computer)
 - will be later rename to Mac OS
 - is the OS that popularized the idea graphical interface
- **1985: Windows 1.0**
 - a graphical user interface for MS-DOS
 - announced in 1983, closely resembled semăna the Apple Macintosh UI; at launch it was shown in a modified form
- **1990: Windows 3.0**
 - the first significant success of Windows
 - partial multitasking (cooperative), virtual memory (i386)
 - important versions: Windows 3.1, Windows 3.11 for Workgroups

37

- **August 1991: The first Linux version**
 - developed by Linus Torvalds, then student at Helsinki University (Finland)
 - implements the UNIX-specific standards
 - open source, becomes very popular, quickly develops as a strong, mature OS
- **1992: BSD is ported to i386**
- **1993: Windows NT**
 - new system version, different from the other Windows systems
 - native multitasking, multiuser
 - the first complete 32 bits OS; nowadays, also comes in 64 bits versions
 - this is the system that will eventually become the modern Windows (XP, 2000, Vista, 7, 8)
- **1999: Apple OS X**
 - based on UNIX
 - its kernel (Darwin) will also be used on the mobile versions (iOS)
- ...

38

Types, variants, versions of operating systems

39



40



41

Types of operating systems

Purpose

- server operating systems
 - UNIX (e.g. Solaris), Linux, BSD, Windows Server, OS X Server
- desktop operating systems
 - Windows, Linux, BSD, OS X, Chrome OS
- mobile operating systems
 - Android, iOS, Windows 8, Symbian, Bada, BlackBerry OS, Palm OS
- embedded operating systems
 - OpenWRT (Linux), Windows CE, LynxOS
- network operating systems
 - Novell NetWare, JunOS (Juniper), Cisco IOS

42

Types of operating systems

Kernel origin

- UNIX / Linux
 - Solaris, HP-UX, BSD, OpenBSD, FreeBSD, Linux (toate variantele), Android, OS X, OS X Server, iOS, webOS, Chrome OS, Tizen, openWRT, Firefox OS etc.
- Windows NT
 - Windows NT, 2000, XP, Vista, 7, 8 (including the Server versions), Windows Phone 8
- Windows
 - Windows 95, Windows 98, Windows Millennium
- other proprietary kernels
 - Symbian, Palm OS, ...

43

Types of operating systems

Licensing model

- proprietary operating systems
 - the various UNIX variants (e.g. Solaris), Windows, OS X, BlackBerry OS
- open source operating systems
 - Linux, BSD
- open source with proprietary components
 - Android, Tizen (Samsung, Intel, Linux Foundation), webOS (Palm→HP→LG)
- proprietary operating systems using open source components
 - OS X, iOS (open source "Darwin" kernel, derived from BSD)

44

Licensing models

Software license

- ➔ *A legal instrument (contract) describing and imposing the terms related to the way a software product can be used, modified, and/or distributed*

45

Licensing models

- **Free/Open source**
 - ➔ allows that the code source, concept and design to be freely used, modified and published or shared (some terms apply)
 - “copyleft” open source
 - ex: GNU General Public License (GPL)
 - unlimited freedom for usage, study, change and redistribution, as long as the redistribution does not introduce additional restrictions to GPL (e.g., it doesn't make the code proprietary)
 - permissive open source
 - ex: BSD License
 - unlimited freedom for usage and study, freedom of change. , change. The redistribution terms are more relaxed, do not impose keeping the completely open character of the software.
- **Proprietary / closed source**
 - ➔ a limited number of copies can be used according to an EULA (End-User Licence Agreement)
 - ➔ the company retains source code ownership; seldom permits redistribution

46

Operating system versions

- **Windows**
 - 32 bit: Windows NT, 95, 98, Millenium, XP
 - 32/64 bit: Windows XP, Vista, 7, 8
- **Linux distributions**
 - are operating systems packaged as complete solutions; they include a Linux kernel and a vast suite of applications
 - the majority are free / open source, some are commercial, others provide payed technical support
 - include complex software package management, are easily extensible and upgradable
 - examples: Slackware, openSUSE, Debian, Fedora, Ubuntu, Mandriva, Mint Linux, CentOS, RedHat, Arch Linux, ...
- **Apple Mac OS X**
 - certified as UNIX, with an open source kernel (Darwin) derived from BSD
 - 10.4: "Tiger", 10.5: "Leopard", 10.6: "Snow Leopard", 10.7: "Lion", 10.8: "Mountain Lion", 10.9: "Mavericks", 10.10: "Yosemite"

47

Operating System Architectures - an Introduction

48

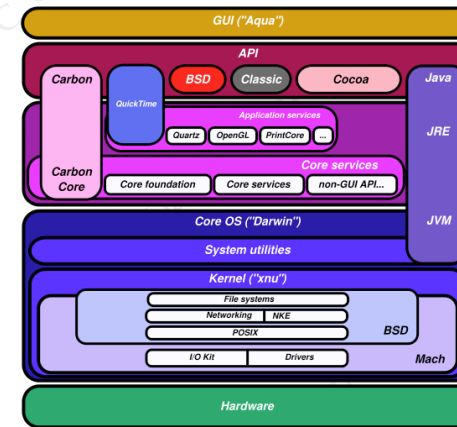
OS components

- **Kernel**
 - provides the main functionality of the OS
 - its size is very dependent on the actual OS architecture and type
- **Functional subsystems**
 - other OS components, having various purposes
 - may include system commands and utilities, APIs, specialized libraries, system services implemented outside the kernel etc.

The various architectures define different functional relations between the kernel and the other OS components

49

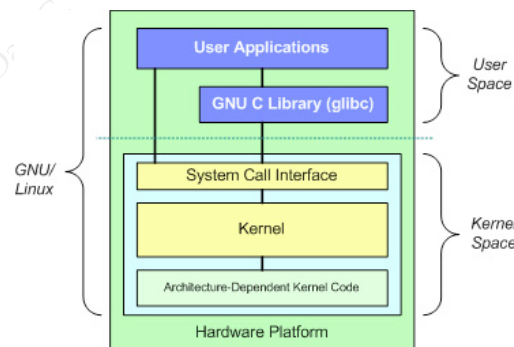
Example: Mac OS X architecture



(c) Wikimedia Commons, <http://commons.wikimedia.org/wiki/File:MacOSXArchitecture.svg>

50

Example: Linux architecture



source and (c): <http://www.ibm.com/developerworks/library/l-linux-kernel/>

51

Memory spaces

Modern operating system define two virtual memory spaces

- **Kernel space**
 - the memory space used by the kernel and the majority of drivers
- **User space**
 - used by user applications, utilities, commands, some OS-specific services or drivers

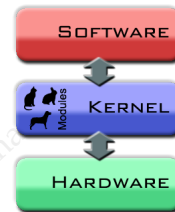
This separation enables accurate privilege-based control, protection and security

52

Types of kernels

- **Monolithic**

- all services run in the same memory space as the main kernel thread
- may involve dynamically loadable modules (Linux)
- advantages: direct access to hardware, fast communication inside the kernel, easier to implement
- disadvantages: strong dependencies between the kernel components, difficult maintenance
- example: Linux



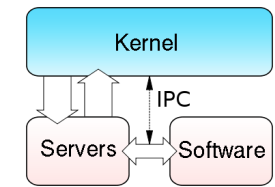
source: Wikipedia

53

Types of kernels

- **Microkernel**

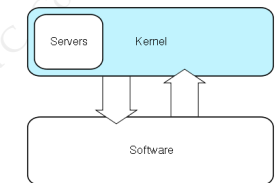
- a set of "server" components, built around a minimal kernel
- the kernel only provides the most basic services: inter-process communication (IPC), memory management, process management
- the "servers" implements all the other system-specific services and are placed in different memory spaces than the kernel
- advantages: flexibility, easy maintenance, minimal dependencies
- disadvantages: lower performance (because of the intense inter-server communication) larger memory needs, harder to debug
- example: QNX



source: Wikipedia

- **Hybrid kernels**

- similar with microkernel, but include more services implemented directly by the kernel, to improve performance
- Examples: OS X (Darwin), Windows NT



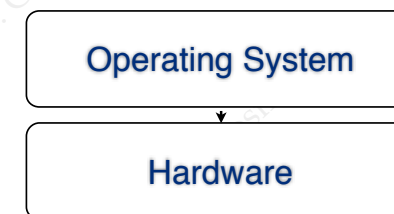
source: Wikipedia

54

Virtual Machines

55

- **The traditional architecture**



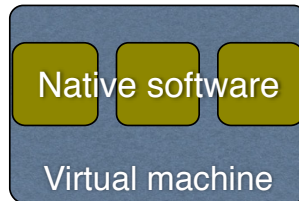
- **Disadvantages**

- only one OS at a time
- applications must be ported to several OS-es

- **Solution: virtualization**

56

- **Virtual Machine (VM)**
a software that simulates a complete hardware system (computer), providing a virtual environment for the programs to run in

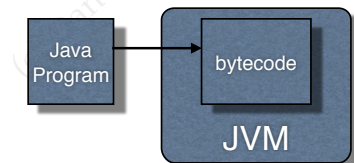
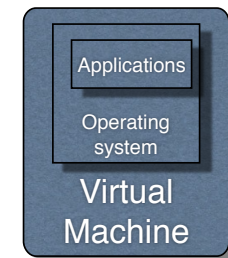


- **Characteristics:**
 - the virtual machine can run native software, for which a separate hardware would have been needed
 - one computer can run several virtual machines at the same time, each having its own distinct architecture
 - the programs installed inside the virtual machine run as if on real hardware, and are completely isolated from the host system and from the other virtual machines

57

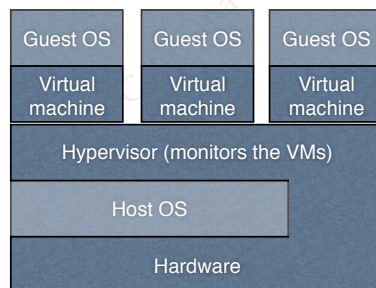
Types of virtual machines

- **System VM**
 - simulates a complete hardware, usually an existing, real, computer system
- **Process VM**
 - provides a virtual execution environment for running programs written in a specific programming languages
 - the VM is developed only for running the byte code of these applications, it doesn't simulate a real system
 - provides portability to the programs written in that specific language
 - example: Java Virtual Machine (JVM)



58

The general architecture



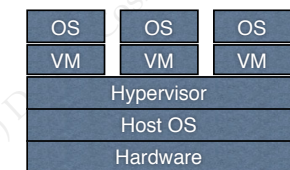
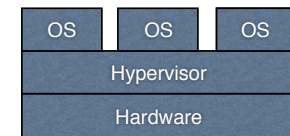
59

Virtualization techniques

A. Native virtualization

→ virtualizes the particular hardware it runs on

- **Type 1 Hypervisor (native)***
 - directly accesses the hardware
- **Type 2 Hypervisor***
 - runs over a conventional OS



*After Gerald J. Popek and Robert P. Goldberg: "Formal Requirements for Virtualizable Third Generation Architectures", 1974

60

Virtualization techniques

B. Architectural emulation

- virtualizes a foreign hardware architecture
- the virtual architecture doesn't necessarily have a correspondent in real-life hardware

C. Operating system-level virtualization

- a technology that virtualizes *servers* within the OS
- the OS kernel provides several distinct user spaces which are available to the user as distinct servers
- the distinct spaces are separated from each other and do not interact
- not all OS-specific services are provided to the virtual servers
- cannot host other OS-es than the real (host) one

61

2. Advanced OS usage

62

Operating Systems “Design Philosophy”

63

Design-time goals

- **Target**
 - general use, specific use (desktop, server, mobile, embedded), tipul de utilizator (avansat, novice, consumer)
- **Interaction**
 - processing (jobs, multiuser), user interface type (command-line, graphical), main user interaction paradigm (direct commands, touch gestures voice, windows and buttons,...) etc.
- **Philosophy**
 - basic principles that shape the system, applicable to the entire system, regardless of the other goals

64

The UNIX Philosophy

- **Modularity and interconnectivity**
→ writing software components that are easy to connect to each other: the output of any program can be the input for another program
- **Clarity and simplicity**
→ simple and clear applications are preferred to unnecessarily complex programs
- **Well-defined and focused purpose**
→ a program must do one single thing, and do it well

Elaborate and diverse behaviors can and should be achieved by freely combining simple, focused and interconnectable components, thus avoiding unnecessary complexity

65

Why UNIX ?



66

UNIX commands

67

The UNIX command-line interface

Expressive

- enables the user to directly and accurately define her needs
- uses simple, focused concepts, with well-defined goals and function

Powerful

- a great variety of commands are available
- command behavior can be tuned extensively (arguments, configuration files, environment variables)
- maximum flexibility by freely combining existing commands to achieve new functionality

Adaptable

- configurable, can be used by persons with various levels of experience: while it doesn't complicate the interaction, it does not impose artificial limitations

Independent

- does not depend on special hardware features: e.g., the same powerful functionality can be accessed both locally, and from remote locations

68

The command interpreter (the shell)

= An interactive program presented to the user

- provides a command-line interface
- allows command executions, command interaction, control
- there are several interpreters in UNIX, selectable by the users
- users can start as many shells they want (in separate windows, in separate text-mode consoles, etc.)
- provides programming-like facilities (scripting)
- examples: sh, csh, tcsh, bash

Terminology:

- *shell* (in a larger sense) - any user interface
- *shell* - the command interpreter
- *shell script* - a program written using the syntax and semantics recognized by the shell, using commands to do various tasks

69

Types of commands

- Internal commands

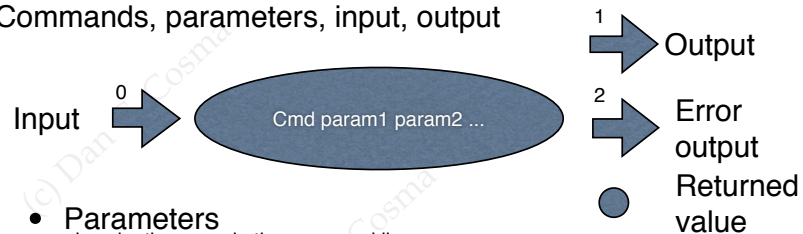
- interpreted directly by the shell
- examples: cd, break, fg, bg, source, eval, exec, exit

- External commands

- independent executables, existent as separate programs on the disk; this includes the OS-specific commands, and the "OS-specific" installed applications
- examples of external UNIX commands: ls, man, cat, cut, ps, top

70

Commands, parameters, input, output



- Parameters

- given by the users in the command line
- words separated by spaces

- Input

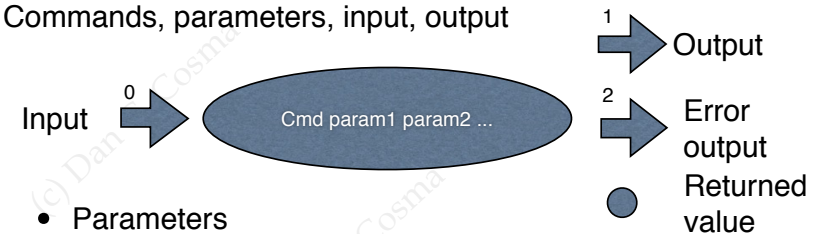
- default input: the keyboard
- can be redirected from files, or can be generated by other programs
- standard descriptor: 0 ("stdin")

- Output

- default: the current shell window, the screen
- can be redirected to files, or sent to other commands
- standard descriptors: 1 ("stdout") and 2 ("stderr")

71


Commands, parameters, input, output



- Parameters

```
ls -al . | grep ".gmail-"
```

- Input

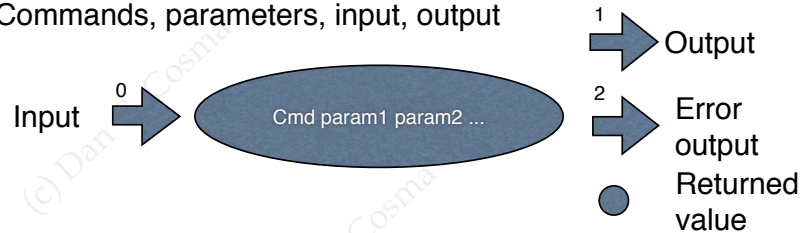


```
$ grep "abc"
abc 123
abc 123
asdf 234
123 abc 5678
123 abc 5678
```

- Output

```
$ ls -al .
drwx----- 28 root root 4096 Oct 4 15:35 .
drwxr-xr-x 25 root root 4096 Jul 10 13:35 ..
-rw----- 1 root root 8902 Oct 9 16:29 .bash_history
-rw-r--r-- 1 root root 570 Jan 31 2010 .bashrc
drwxr-xr-x 2 root root 724096 Jul 5 18:55 Desktop
```

Commands, parameters, input, output



- **Returned value**
 - At termination, any program returns an integer value to the operating system
 - C: `exit(value)`; or `return value;` in `main()`
- **Convention:**
 - 0: the program ended correctly
 - $\neq 0$: the program ended with error (and the value is the error code)
- **In the UNIX command line:**

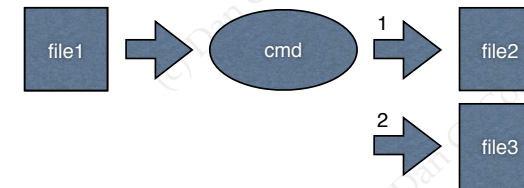
```
$ test 1 -eq 2
$ echo $?
1
```

73

Input and output redirection

- **Redirection**

- `cmd < file1`
- `cmd > file2`
- `cmd >> file2`
- `cmd 2> file3`



74

Chaining commands

- **UNIX commands are interconnectable**
 - input and output: usually text
 - the output of a program can become the input to another program
- **Chaining commands**
 - the "pipe" operator is used: `|`
 - `cmd1 | cmd2 | cmd3 ...`
 - the commands start in parallel



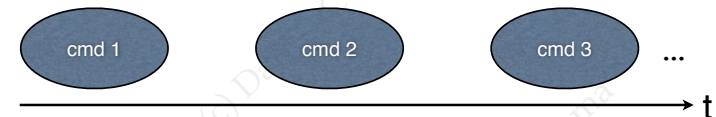
`ls -al . | grep ".gmail-"`

75

Chaining commands

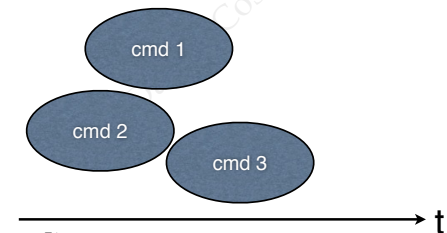
- **Sequential execution**

- `cmd1 ; cmd2 ; cmd3` ▷ Returned value: the value returned by the last command
- `cmd1 || cmd2 || cmd3` ▷ Returned value: logical OR
- `cmd1 && cmd2 && cmd3` ▷ Returned value: logical AND



- **Parallel execution**

- `cmd1 & cmd2 & cmd3 &`



76

Examples of UNIX commands

```
man [options] [section] command
pwd
cd directory
ls [-adgilrst] file ...
mv file1 file2
cp file1 file2
sort
du
df
who
ps
```

77

Examples of redirection and chaining

```
sort < fis1 > fis2

ls -l | grep student | wc -l > fis3

ls -a | grep ".qmail-" | grep -v ".qmail-ac:" | grep ":" | less

tar czf - /etc/ /var/named/ /service /var/lib/qmail /var/qmail
| ssh subspace.cs.upt.ro "dd of=/disk2/bf2-bak-20130710.tar.gz"

/opt/bin/mediaclient --cat /dev/dvb/adapter0/dvr0
| /Applications/VLC.app/Contents/MacOS/VLC file:///dev/stdin

JARS=`find ${ANT_LIB} -name '*jar' | while read JAR_FILE; do
echo -n ":$JAR_FILE"; done`
```

78

UNIX shell scripts

- *shell script* - a **program** written using the syntax and semantics recognized by the **command interpreter**, using OS-specific commands to do various tasks

79

80

command interpreter

81

bash

82

bash

- “Bourne-again Shell” (wordplay derived from the name Bourne Shell)
- the free and modern replacement of one of the traditional UNIX shells (Bourne Shell -- *sh*)
- diverse facilities of command processing and programming: control structures, wildcarding, pipe, command substitution, iteration, condition evaluations, command history, autocompletion in the command line, etc.
- the syntax is a superset of the *sh*-specific syntax, extended and improved
- present in basically all current UNIX versions/distributions

83

bash

- “Bourne-again Shell” (wordplay derived from the name Bourne Shell)
- the free and modern replacement of one of the traditional UNIX shells (Bourne Shell -- *sh*)
- diverse facilities of command processing and programming: control structures, wildcarding, pipe, command substitution, iteration, condition evaluations, command history, autocompletion in the command line, etc.
- the syntax is a superset of the *sh*-specific syntax, extended and improved
- present in basically all current UNIX versions/distributions

other shells: csh, ksh, tcsh, ...

84

bash → Environment variables

85

→ Environment variables

86

Environment variables

- variables usable in the command line and scripts
- their type is always string

- Assignment
`VARIABLE=value`
- Getting the value
`$VARIABLE`

```
$ VAR1=abcd
$ echo $VAR1
abcd
```

87

- Concatenation

```
$ VAR1=abcd
$ VAR2=x
$ echo $VAR1$VAR2
abcdx
$ echo 123$VAR1
123abcd
$ echo ${VAR1}123
abcd123
$ VAR3=y${VAR1}123; echo $VAR3
yabcd123
```

88

- Quotation

```
$ VAR1="ab cd"
$ VAR2=x
$ echo $VAR1
ab cd
$ echo "123 $VAR2"
123 x
$ echo '123 $VAR2'
123 $VAR2

$ echo \ $VAR1
$VAR1
$ echo \ $ $VAR1
$ab cd
```

89

Predefined environment variables

PWD - current (working) directory
HOME - current user's home directory
PATH - command search path
 example: /bin:/usr/bin:/usr/local/bin
PS1 - first prompt (printed by the shell before the command line)
PS2 - secondary prompt
UID - the ID of the current user
HOSTNAME - the name of the computer
 ...

```
mycomputer:~ janedoe$ echo $PS1
\h:\W \u\ $
mycomputer:~ janedoe$ echo $PS2
>
mycomputer:~ janedoe$ echo #HOME
/home/janedoe
```

90

Special variables

*****, **@** - the command line parameters
- the number of command line parameters
? - the status of the last run command
0 - the name of the current script or the current shell
1, 2, ... - the *n*th parameter in the command line
 ...

```
myscript.sh:
echo $#
echo $@

$ sh myscript.sh 1a 2 b 3
4
1a 2 b 3
$ sh myscript.sh 1a "2 b" 3
3
1a 2 b 3
```

91

Path name expansion

→ after it separates the words in the command line, bash searches each word for the occurrence of the following characters: *****, **?** and **[** .
 → if found, interprets the respective words as patterns which it expands as file names, as follows:

***** - any string, including the null string
? - exactly one character
[...] - any of the characters between the square brackets

```
$ ls *.txt
a.txt abctxt txt
$ ls abc?l
abcx1 abcd1 abcl1
$ ls ab[12]x
ab1x ab2x
```

92

Brace expansion

- resembles path expansion, but the resulting names do not necessarily need to represent existing file names
- syntax:
 - prefix{expression}postfix
 - the expression can be made of words separated by commas, or of interval specifiers (..)

```
$ echo a{x,y}
ax ay
$ echo a{x..z}
ax ay az
$ mkdir a{1,2,3}x
$ (will create the directories: a1x, a2x, a3x)
```

93

Tilde expansion (“~”)

- `~name` is interpreted as the home directory of the user `name`
- `~` is interpreted as the home directory of the current user

```
$ echo ~
/home/janedoe
$ echo ~gregory
/home/gregory
```

94

Command substitution

- in a syntactic construct in the form ``string`` or `$(string)`, the shell considers `string` a command
- the command is executed, and its output will be used for replacing the entire syntactic construct
- note: the delimiters are backquotes: ```, not quotes

```
$ echo `pwd`
/home/janedoe/alx
$ A=`ls -a`
$ echo $A
. .. abc myscript.sh xyz
```

95

Input and output redirection

- the general construct for output redirection (e.g., given after a command):
`[n]>word`
`[n]>>word`
 - redirects the file descriptor `n` to file `word`; if `n` is not specified, the descriptor is redirected to the standard output
 - if `>>` is used, the output of the command will be *appended* at the end of the file (the file is not overwritten)
- the general construct for redirecting the input:
`[n]<word`
 - redirects the file descriptor `n` to read from the file `word`; if `n` is missing, the redirection is done for the standard input

```
$ cat x 1>>fisier
```

96

“Here Documents”

```
<<word
  here-document
delimiter
```

- the interpreter will take **word** as a delimiter
- the text in the *here-document* lines is sent to the command input
- the delimiter marks the end of the input
- if **word** contains single/double quote characters (“ or ’), they are ignored, but inside the here-document no variable expansion or command substitution is done

```
cat <<SFARSIT
abc
123
x
SFARSIT
abc
123
x
```

97

Other bash features

- aliases (comenzile interne alias, noalias)
- functions
- array variables
- command autocompletion (the TAB key)
- command history (the .bash_history file)
- default scripts started at login, logout and when bash starts (.bash_profile, .bash_logout, .bashrc)
- ...

98

Conditions

- the internal commands **test** and **[**
- used for composing logical expressions
- the conditional command returned can be 0 (**true**) or 1 (**false**), so that is easily integrated with the conditional specifiers (such as if)
- examples of parameters for **test** and **[**:
 - e file - true if file exists
 - d file - true if file exists and is a directory
 - f file - true if file exists and is regular
 - string1 == string2 - true if string1 is identical with string2
 - string1 != string2 - true if string1 differs from string2
 - string1 < string2 - true if string1 is before string2, alphabetically
 - string1 > string2 - true if string1 is after string2, alphabetically
 - arg1 operator arg2 - true if arg1 and arg 2 are in the relation specified by the operator
 - where operator: -eq (equal), -ne (not equal), -lt (lower than), -gt (greater than), -ge (greater or equal), -le (lower or equal)

```
$ [ 1 -eq 2 ]
$ echo $?
1
```

99

Control structures

```
for name [ in word ] ; do list ; done
```

```
for (( expr1 ; expr2 ; expr3 )) ; do list ; done
→ expr1 - arithmetic expressions
→ first expr1 is evaluated; then expr2 is evaluated at each iteration until reaches zero; if expr2≠0 list is executed, and expr3 is evaluated
```

```
case word in [ ([ pattern [ | pattern ] ... )
list ;; ] ... esac
```

```
if list; then list; [ elif list; then list; ] ...
[ else list; ] fi
```

```
while list; do list; done
```

```
until list; do list; done
```

100

Examples

```
for (( i=1; i<=4; i++ ))
do
    echo $i
done
```

```
for i
do
    echo $i
done
```

```
i=0
while [ -f .dotask ]
do
    (( i++ ))
    echo Starting task: $i
    /usr/local/bin/myprogram --start
done
```

```
$ sh script
1
2
3
4
```

```
$ sh script a b c
a
b
c
```

```
$ sh script
Starting task: 1
Starting task: 2
Starting task: 3
Starting task: 4
Starting task: 5
...
```

101

Examples

```
case "$1" in
    start)
        /usr/local/myservice -d
        ;;
    stop)
        /usr/local/myservice -x
        ;;
    status)
        if /usr/local/myservice --isrunning 2>/dev/null
        then
            echo Service is running
        else
            echo Service not started
        fi
        ;;
    restart)
        stop
        start
        ;;
    *)
        echo $"Usage: $0 {start|stop|restart}"
        exit 1
esac
```

```
$ sh service.sh status
Service is running
$ sh service.sh reboot
Usage: service.sh {start|
stop|restart}
```

102

Functions

Recursive function

```
#!/bin/sh

fact()
{
    if [ $1 -gt 1 ]
    then
        i=`expr $1 - 1`
        j=`fact $i`
        k=`expr $1 \* $j`
        echo $k
    else
        echo 1
    fi
}

read -p "Numar:" x
fact $x
```

The name of the shell that will execute this script

First function parameter

103

Example

```
for i in episode-S02E*avi
do
    nr=`ls "$i" | cut -c 13-14`
    fn=${i%.*}
    mv episode-subtitle-en-2x${nr}*srt "$fn".srt
    echo $nr $fn
done
```

104

Regular expressions in UNIX

105

Regular expression

An expression made of a sequence of characters describing a pattern used when searching text.

106

UNIX

Many UNIX commands accept regular expressions, in two formats:

- POSIX basic (BRE)
- POSIX extended (ERE)

107

POSIX regular expressions

. - matches a single character
Example: a.x smatches abx, aax, acx, etc.

[] - matches a single character of those specified between the brackets

Examples: [abc] matches a, b or c
[a..x] - any character between a and x

[^] - matches a single character except for those specified between the brackets

Examples: [^abc] matches any character except a, b or c
[^a..x] - any character, except those between a and x

^ - matches the start position (usually in a line)
Example: ^a means "the a character at the beginning of the line"

\$ - matches the end position (usually in a line)
Example: a\$ means "the a character at the end of the line"

108

POSIX regular expressions

() - defines a subexpression

The value that matches the pattern between brackets can be later referenced using `\n` (where n is a number). In the "basic" syntax (BRE), the parentheses must be quoted: `\(\)`.

\n - reference to a subexpression

Refers the n -th subexpression designated by parentheses, where $n \in [1, 9]$.

Describes an expression (string) made of zero or more occurrences of the character that precedes the `*`.

Examples:

`ab*x` matches `ax`, `abx`, `abbbbx`, etc.

`[abc]*` matches the null string, `a`, `aa`, `aaaa`, `b`, `bbb`, `ab`, `ba`, `abcc`, `abc`, `aabbcc`, `aabbcca`, etc.

{m,n}

Describes an expression made of a minimum of m and a maximum of n occurrences of the preceding character. In the "basic" mode (BRE), the braces must be quoted: `\{` and `\}`.

109

POSIX regular expressions

+

Only available in the extended mode (ERE). Describes an expression made of one or many occurrences of the preceding character.

?

Only available in the extended mode (ERE). Describes an expression made of zero or one occurrence of the preceding character.

Note: in the extended mode references to subexpressions (`\n`) are not available, and a quotation using `\` will simply mean the next character as it is (`\(` means the character `(`).

110

Exemple

<code>[cm]asa</code>	<code>casa masa</code>
<code>^c?are</code>	<code>care are (strictly at the beginning)</code>
<code>.are</code>	<code>sare mare tare xare fare ...</code>
<code>[ab]*re</code>	<code>re are aabre abbre aaarea aabbre baare ...</code>
<code>a+re</code>	<code>are aare aaare ... (but not "re")</code>
<code>episode\ [123]x.*</code>	<code>episode 1x01 - The Super Hero</code> <code>episode 1x02 - The Hero Cries</code> <code>episode 3x22 - Hero No More</code> <code>...</code>

111

Commands that recognize regular expressions

grep, sed, awk

Usually, to enter the extended mode (ERE), commands need a specific option (e.g., `-E`).

grep -E pattern
egrep pattern

`ls -l | grep -E ^Fisierul\ meu.*txt$`

`Fisierul meu cu scrisori.txt`
`Fisierul meu preferat.txt`
`Fisierul meu cu txt`
`...`

112

3. File systems

113

Programming with files

114

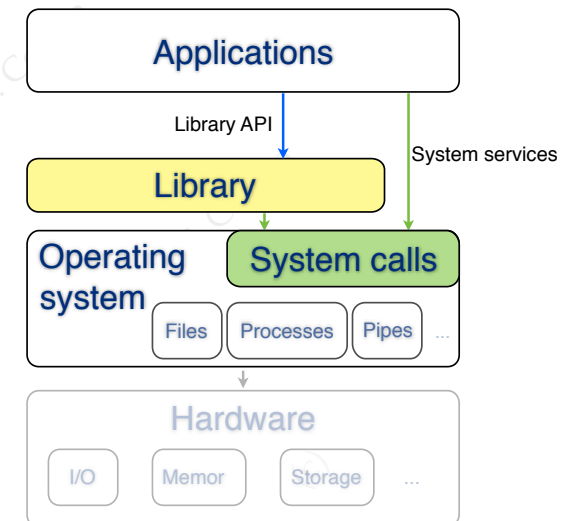
```
if((fd1=open(argv[1], O_RDONLY))<0)
{
    printf("Error opening input file\n");
    exit(2);
}
if((fd2=open(argv[2], O_WRONLY | O_CREAT |
O_EXCL, S_IRWXU)) < 0)
{
    printf("Error creating destination file
\n");
    exit(3);
}

while((n = read(fd1, &c, sizeof(char))) > 0)
{
    if(write(fd2, &c, n) < 0)
    {
        printf("Error writing to file\n");
        exit(4);
    }
}

if(n < 0)
{
    printf("Error reading from file\n");
    exit(5);
}

close(fd1);
```

115



116

Lecția de engleză

117

Lecția de engleză. Și de română

118

Lecția de engleză. Și de română

Library - noun,

: a place in which literary, musical, artistic, or reference materials (as books, manuscripts, recordings, or films) **are kept for use but not for sale**

: a collection resembling or suggesting a library



The Free Merriam-Webster dictionary, www.m-w.com

Librărie - substantiv,

: Magazin în care se vând cărți.

Dicționarul explicativ al limbii române, ediția 1998

119

Lecția de engleză. Și de română

Library - noun,

: a place in which literary, musical, artistic, or reference materials (as books, manuscripts, recordings, or films) **are kept for use but not for sale**

: a collection resembling or suggesting a library



The Free Merriam-Webster dictionary, www.m-w.com

~~**Librărie** - substantiv,~~

~~: Magazin în care se vând cărți.~~

Biblioteca - substantiv,

: Instituție care colecționează cărți, periodice etc. spre a le pune în mod organizat la dispoziția cititorilor

: Colecție de cărți, periodice, foi volante, imprimate etc.

Dicționarul explicativ al limbii române, ediția 1998

120

Lecția de engleză. Și de română

Library - noun,

: a place in which literary, musical, artistic, or reference materials (as books, manuscripts, recordings, or films) **are kept for use but not for sale**

: a collection resembling or suggesting a library

The Free Merriam-Webster dictionary, www.m-w.com



Biblioteca - substantiv,

: Instituție care colecționează cărți, periodice etc. spre a le pune în mod organizat la dispoziția cititorilor

: Colecție de cărți, periodice, foi volante, imprimate etc.

Dicționarul explicativ al limbii române, ediția 1998

121

Lecția de engleză. Și de română

Bookstore - noun,

: a place of business where books are the main item **offered for sale** — called also bookshop

The Free Merriam-Webster dictionary, www.m-w.com



Librărie - substantiv,

: Magazin în care se vând cărți.

Dicționarul explicativ al limbii române, ediția 1998

122

Library ≠ **Librărie**

Library = **Biblioteca**

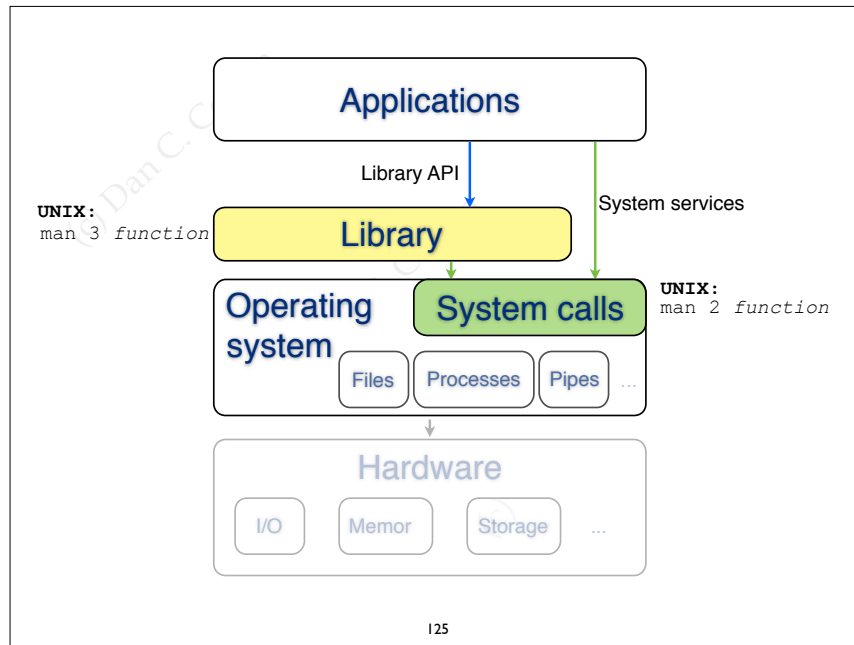
123

Library ≠ **Librărie** ← “False Friend”

Library = **Biblioteca**

Concluzie: nu vă ajutați singuri să deveniți ridicoli... ;)

124



System calls for working with files

```
int open(const char *pathname, int oflag, [, mode_t mode]);
```

- pathname** - file name
- oflag** - opening flags. It is a set of bits. `fcntl.h` defines constants that can be combined with "T". Examples:
 - `O_RDONLY` - open for reading only
 - `O_WRONLY` - open for writing
 - `O_RDWR` - open for reading and writing
 - `O_APPEND` - open for appending at the end of the file
 - `O_CREAT` - create the file if it does not exist; with this option, **open** must also receive the parameter *mode*.
 - `O_EXCL` - "exclusive" file creation: if `O_CREAT` is used and the file already exists, **open** will return error
 - `O_TRUNC` - if file exists, it is truncated
- mode** - only when creating a file - the access rights for the file. Constants:
 - `S_IRUSR` - read for the owner (*user*)
 - `S_IWUSR` - write for the owner (*user*)
 - `S_IXUSR` - execute for the owner (*user*)
 - `S_IRGRP` - read for the group that owns the file
 - `S_IWGRP` - write for the group
 - `S_IXGRP` - execute for the group
 - `S_IROTH` - read for others
 - `S_IWOTH` - write for others
 - `S_IXOTH` - execute for others

Returns a **file descriptor**

```
int creat (const char *pathname, mode_t mode);
int close (int filedes);
```

126

System calls for working with files

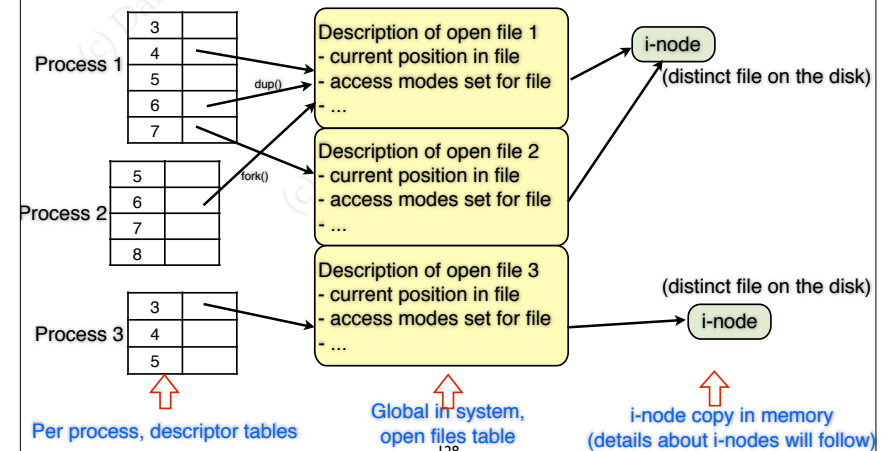
Standard file descriptors:

```
STDIN_FILENO
STDOUT_FILENO
STDERR_FILENO
```

127

Managing open files in UNIX

- file descriptors: numbers between 1..n, n depends on the system
- an initialized descriptor references an open file; once closed, the descriptor is reused
- each process owns a table containing the open descriptors

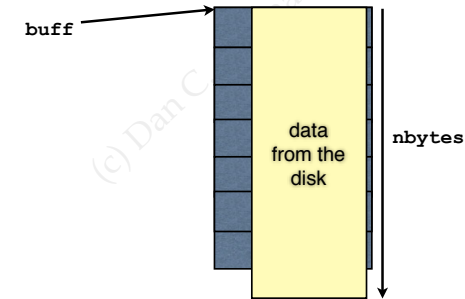


- a call to open() creates
- a new descriptor
 - a new open file description

129

```
ssize_t read(int fd, void *buff, size_t nbytes)
```

- reads **exactly** *nbytes* bytes from the current position
- stores the bytes in the memory area **referenced** by *buff*
- returns the number of bytes **actually** read (0 at the end of file) or -1 if error

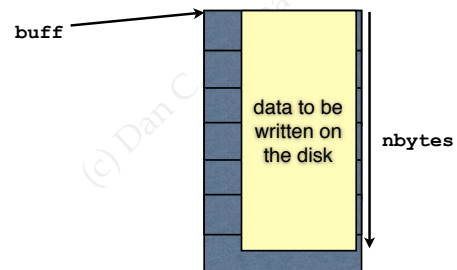


What happens if the buffer is not correctly allocated?

130

```
ssize_t write(int fd, void *buff, size_t nbytes)
```

- writes on the disk **exactly the first** *nbytes* bytes in buffer
- gets them from the memory area **indicated** by *buff*



What happens if the buffer is not correctly allocated?

131

```
off_t lseek(int fd, off_t offset, int pos)
```

Sets the offset of the file descriptor at *offset*, as follows:

- if *pos* = SEEK_SET, the positioning is calculated relatively to the start of the file
- if *pos* = SEEK_CUR, the positioning is relative to the current position
- if *pos* = SEEK_END, the positioning is relative to the end of the file

```
int mkdir(const char *pathname, mode_t mode)
```

```
int rmdir(const char *pathname)
```

132

Library functions for working with files

```
FILE *fopen(const char *filename, const char *mode);

int fclose(FILE *stream);

int fprintf(FILE *stream, const char *format, ...);

int fscanf(FILE *stream, const char *format, ...);

size_t fread(void *ptr, size_t size, size_t nmemb, FILE
*stream);
reads from the file indicated by stream a number of nmemb elements, each having
the size size, and puts them in the memory area indicated by ptr.

size_t fwrite( void *ptr, size_t size, size_t nmemb, FILE
*stream);
writes to the file indicated by stream a number of nmemb elements, each having
the size size, read from the memory area indicated by ptr.
```

133

Library functions for working with files

Standard file descriptors:

```
stdin
stdout
stderr
```

134

Finding out file properties (system calls)

```
int stat(const char *file_name, struct stat *buf);

int fstat(int filedes, struct stat *buf);

int lstat(const char *file_name, struct stat *buf);

struct stat
{
    dev_t      st_dev;      /* device */
    ino_t      st_ino;      /* inode */
    umode_t    st_mode;     /* protection */
    nlink_t    st_nlink;    /* number of hard links */
    uid_t      st_uid;      /* user ID of owner */
    gid_t      st_gid;      /* group ID of owner */
    dev_t      st_rdev;     /* device type (if inode device) */
    off_t      st_size;     /* total size, in bytes */
    unsigned long st_blksize; /* blocksizes for filesystem I/O */
    unsigned long st_blocks; /* number of blocks allocated */
    time_t     st_atime;    /* time of last access */
    time_t     st_mtime;    /* time of last modification */
    time_t     st_ctime;    /* time of last change */
};
```

135

Library functions for working with directories

```
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
void rewinddir(DIR *dir);
int closedir(DIR *dir);

/*Linux*/
struct dirent {
    ino_t      d_ino;      /* inode number */
    off_t      d_off;      /* offset to the next dirent */
    unsigned short d_reclen; /* length of this record */
    unsigned char d_type;   /* type of file; not supported
                             by all file system types */
    char       d_name[256]; /* filename */
};
```

136

- `int link(const char *oldpath, const char *newpath);` - creeaza legaturi fixe spre fisiere
- `int symlink(const char *oldpath, const char *newpath);` - creeaza legaturi simbolice spre fisiere sau directoare
- `int unlink(const char *pathname);` - sterge o intrare in director (legatura, fisier sau director)
- `int rename(const char *oldpath, const char *newpath);` - redenumire / mutare de fisiere
- `int rmdir(const char *pathname);` - stergere de directoare
- `int chdir(const char *path);` - schimbarea directorului curent
- `char *getcwd(char *buf, size_t size);` - determinarea directorului curent

137

An example

- program that copies a file having the name specified as the first argument in the command line to another file, also specified in the command line
- error messages are printed when necessary

138

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

void usage(char *name)
{
    printf("Usage: %s <source> <destination>\n", name);
}

int main(int argc, char *argv[])
{
    int fd1, fd2;
    int n;
    char c;

    /** Check command line args */
    if(argc!=3)
    {
        usage(argv[0]);
        exit(1);
    }
}
```

139

```
/** Open files */
if((fd1=open(argv[1], O_RDONLY))<0)
{
    printf("Error opening input file\n");
    exit(2);
}
if((fd2=open(argv[2], O_WRONLY | O_CREAT | O_EXCL, S_IRWXU)) < 0)
{
    printf("Error creating destination file\n");
    exit(3);
}

while((n = read(fd1, &c, sizeof(char))) > 0)
{
    if(write(fd2, &c, n) < 0)
    {
        printf("Error writing to file\n");
        exit(4);
    }
}

if(n < 0)
{
    printf("Error reading from file\n");
    exit(5);
}

close(fd1);
close(fd2);

return 0;
}
```

140

```

/** Open files */
if((fd1=open(argv[1], O_RDONLY))<0)
{
    printf("Error opening input file\n");
    exit(2);
}
if((fd2=open(argv[2], O_WRONLY | O_CREAT | O_EXCL, S_IRWXU)) < 0)
{
    printf("Error creating destination file\n");
    exit(3);
}

while((n = read(fd1, &c, sizeof(char))) > 0)
{
    if(write(fd2, &c, n) < 0)
    {
        printf("Error writing to file\n");
        exit(4);
    }
}

if(n < 0)
{
    printf("Error reading from file\n");
    exit(5);
}

close(fd1);
close(fd2);

return 0;
}

```

141

A small change in the program...

142

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

#define BUFSIZE 4096

void usage(char *name)
{
    printf("Usage: %s <source> <destination>\n", name);
}

int main(int argc, char *argv[])
{
    int fd1, fd2;
    int n;
    char buf[BUFSIZE];

    /** Check command line args */
    if(argc!=3)
    {
        usage(argv[0]);
        exit(1);
    }
}

```

143

```

/** Open files */
if((fd1=open(argv[1], O_RDONLY))<0)
{
    printf("Error opening input file\n");
    exit(2);
}
if((fd2=open(argv[2], O_WRONLY | O_CREAT | O_EXCL, S_IRWXU)) < 0)
{
    printf("Error creating destination file\n");
    exit(3);
}

while((n = read(fd1, buf, BUFSIZE)) > 0)
{
    if(write(fd2, buf, n) < 0)
    {
        printf("Error writing to file\n");
        exit(4);
    }
}

if(n < 0)
{
    printf("Error reading from file\n");
    exit(5);
}

close(fd1);
close(fd2);

return 0;
}

```

144


```
date; ./copyfile2 beethoven-symph-5-1.wav b.wav; date
```

File size	Buffer size (bytes)	Copy time
74 MB	1	6 minutes 30 seconds
74 MB	100	3 seconds
74 MB	4096	1 second

Notă: Moreover, even simply calling a function takes time. Do not abuse.

145

About some small C questions. And common sense...

```
read(fd1, buf, BUFSIZE);

read(fd1, buf, sizeof(buf));

read(fd1, buf, strlen(buf));

int v;
...
read(fd1, &v, sizeof(int));
```

```
char buf[BUFSIZE];
char *buf;
```



```
...
/*read file from disk*/
read(fd1, buff, file_size);
...
```



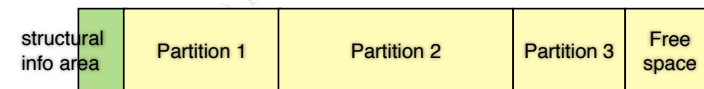
146

File systems

147

Storing data on a disk

- At a logical level, a disk is made of a set of sectors
- Sector size is fixed and depends on the disk type. Example: "regular" hard disk: 512 bytes, newer hard disk: 4096 bytes
- A disk can be [partitioned](#)



Example of partitioned disk

148

Partitioning schemes

- Differ with the type of disk, computer, OS, etc.
- Most popular partitioning schemes:
 - MBR (Master Boot Record)
 - the classic scheme, used on most current PCs
 - GPT (GUID Partition Table)
 - the partitioning scheme for PCs, more flexible

149

MBR

Boot sector

- a region on a disk, usually at the start of the disk, containing, among others, an executable code that can be started by the computer's firmware at the initialization time
- the executable code will load a specific program on the disk, usually a program that starts the OS installed on that disk

Master Boot Record

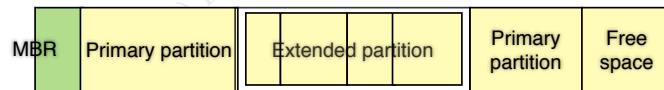
- a special type of boot sector, specific to IBM-PC-compatible computers (even current PCs)
- contains, among the loader program, informations about the disk partitions

150

MBR

Partitioning scheme

- The information about partitions is located in a [partition table](#) in the MBR
- The partition table consists of 4 entries, therefore a maximum of 4 partitions can be defined. These partitions are called [primary partitions](#)
- A partition can be designated as [extended](#), in which case it will contain other partitions



Example of a MBR partitioned disk

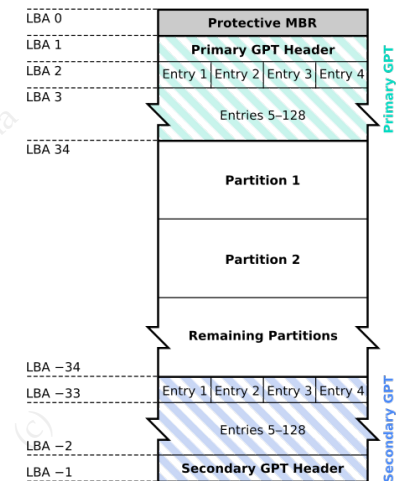
151

GPT

Description

- Part of the UEFI standard (United Extensible Firmware Interface), aimed at replacing PC BIOS
- Uses GUIDs (Globally unique identifier) for disk and partition type identification, in order to avoid duplicates
- Allows creating an arbitrary number of partitions (only dependent on the space reserved for the partition table)

GUID Partition Table Scheme



source: Wikipedia

152

File system

The logical way of organizing data on a physical or virtual support, for storage and data access

- Several file systems can be installed on a same computer, for instance on distinct disks or partitions
- Operating systems usually are accompanied by specific file systems
- Some operating systems recognize several file system types, even if they were not developed for the respective OS
- A file system describes both the data structures involved in data storage, and the way the data is accessed (*e.g.*, as a tree of files and directories)are)

153

Examples of file systems

FAT16 (File Allocation Table, 16 bit)

- Specific to MS-DOS, Windows. File names have maximum 8+3 characters. Maximum volume (partition) size: 2 GB / 4GB (Win NT)

FAT32

- Windows. Longer file names. Maximum volume size: 8 GB. Maximum file size: 4 GB.

NTFS

- Windows NT and successors. Maximum file size: 16 TB (\leq Win7), 256 TB (Win8). Journalling file systems.

HFS Plus

- Specific to OS X. Maximum volume size: 8 EB*. Maximum file size: 8 EB*. Journalling file system.

* $1 \text{ exabyte} = 10^{18} \text{ bytes} = 10^9 \text{ gigabytes}$

154

Examples of file systems

Ext2 - Second Extended File System

- Specific to Linux. Maximum volume size: 2-32 TB. Maximum file size: 16 Gb - 2 TB

Ext3 - Third Extended File System

- Specific to Linux. Maximum volume size: 2-32 TB. Maximum file size: 16 Gb - 2 TB. Journalling file system.

Ext4 - Fourth Extended File System

- Specific to Linux. Maximum volume size: 1 EB*. Maximum file size: 16 TB. Journalling file system.

* $1 \text{ exabyte} = 10^{18} \text{ bytes} = 10^9 \text{ gigabytes}$

155

File system support

Windows

- FAT, NTFS, exFAT

Linux

- Tens of file systems. Examples: ext2, ext3, ext4, XFS, FAT, NTFS, HFS+, JFFS, JFFS2 (Journalling Flash File System)

OS X

- HFS+, UFS, FAT, NTFS (read only)

156

→ Several file systems can be installed on a same computer,
for instance on distinct disks or partitions

```
# fdisk /dev/sda
```

The number of cylinders for this disk is set to 30401.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
1) software that runs at boot time (e.g., old versions of LILO)
2) booting and partitioning software from other OSs
(e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): p

Disk /dev/sda: 250.0 GB, 250059350016 bytes
255 heads, 63 sectors/track, 30401 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk identifier: 0xd42ad42a

Device	Boot	Start	End	Blocks	Id	System
/dev/sda1	*	1	4476	35953438+	7	HPFS/NTFS
/dev/sda2		29095	30400	10490445	7	HPFS/NTFS
/dev/sda3		4477	29094	197744085	5	Extended
/dev/sda5		4477	6428	15679408+	83	Linux
/dev/sda6		6429	6695	2144646	82	Linux swap / Solaris
/dev/sda7		6696	29094	179919936	83	Linux

157

The hierarchical organization of a file system

- Tree / trees of directories and files

→ UNIX: a single tree (a single root directory: /)

→ DOS/Windows: several trees (a root dir for each disk/partition: A:\, B:\, C:\, D:\, ... “\” means the root directory of the current disk)

- File reference

→ **absolute path names:**

UNIX: /usr/bin/ls, /home/jane/myscript, /jome/jane/my\ files/file1

Windows: C:\Windows\wordpad.exe, “D:\games\My Super Game”

→ **path names relative to the current directory:**

UNIX: myscript, “my files/file1”, .ssh/known_hosts

Windows: wordpad.exe, “My Super Game\startgame.exe”

158

Common (almost “standard”) directories in UNIX

- / - the root directory
- /bin - essential commands that are needed even when only the root file system is mounted
- /dev - devices
- /etc - system configuration files
- /home - user home directories
- /lib - essential libraries and kernel modules
- /opt - directories for additional applications
- /sbin - system executables (exclusively for administration uses)
- /tmp - temporary files and directories
- /usr - the root of an important subtree with system-wide purposes
- /usr/X11 - the X11 windowing system
- /usr/X11R6 - the X11R6 windowing system
- /usr/bin - utilities, commands that can be called by users
- /usr/lib - programming libraries
- /usr/local - local applications
- /usr/local/bin - local binaries
- /usr/share - architecture-independent data
- /var - variable data files: e-mail, logs, caches, etc.

159

Common directories in Windows (NT, ..., 8):

- C:\, D:\ - root dirs
- C:\Windows - OS files
- C:\Windows\System - 16 bits system and library files
- C:\Windows\System32 - 32 or 64 bits system and library files
- C:\Windows\SysWOW64 - 32 bits system and library files for running 32 bits applications when the system is 64 bits (WOW = Windows on Windows)
- C:\Documents and Settings - user home directories (NT, 2000, XP)
- C:\Users - user home directories (Vista, 7, 8)
- C:\Temp, C:\Windows\Temp - temporary files and directories

160

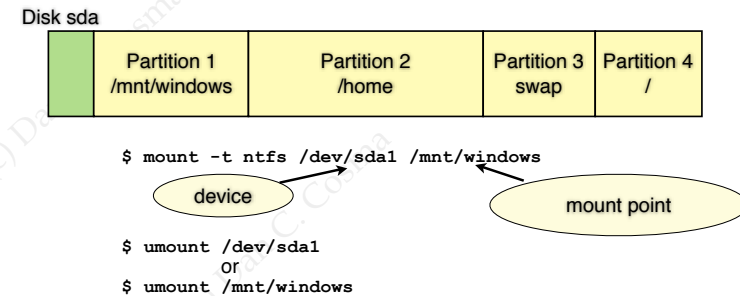
Mounting a file system (UNIX)

- Several file systems can coexist on the same computer
 - on distinct disks, partitions, storage devices, in memory, etc.
- There is a single root directory

- ⇒ A file system can be mounted in any existing directory
- ⇒ The first mounted system is the **root file system**. It is automatically mounted at boot time, in the / directory
- ⇒ The root file system must contain all the necessary files and directories for running the OS

161

Example of mounting and unmounting



The /etc/fstab config file

- specifies file systems having predefined mount points
- if not otherwise specified, they will be automatically mounted at boot time

/dev/sda3	swap	swap	defaults	0 0
/dev/sda4	/	ext3	acl,user_xattr	1 1
/dev/sda2	/home	ext3	acl,user_xattr	1 2
/dev/sda1	/mnt/windows	ntfs	user,noauto	0 0

162

Users and access rights

- Users
 - UNIX accepts multiple users on the same system
 - each user has a name and a user identifier (uid)
 - each user has a home directory, which she owns
 - basic user configuration files: /etc/passwd and /etc/shadow
- Groups
 - users are organized in groups
 - a user can belong to more than one group
 - each group has a name and a group identifier (gid)
 - group configuration file: /etc/group

163

- Access rights are set for each file, for three **categories of users**
 - **file owner**: **user, u** -- the owner of the file; usually, the user that has created the file (but it can also be a different user)
 - **the group that owns the file**: **group, g** -- each file can be owned by a group; by default, it is the owner's group, but it can be changed
 - **all other users**: **others, o**
- There are **three types of rights** for files:
 - **read, r** -- the content of the file can be read
 - **write, w** -- the content of the file can be modified
 - **execute, x** -- the file can be executed; for directories, shows that the directory can be entered
- Combining the above, 9 access rights can be specified, using **9 file mode bits**:

r w x	r w x	r w x
user	group	others

164

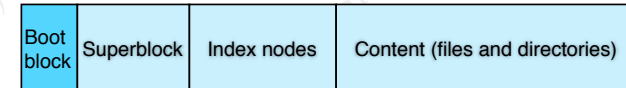
Examples. The chmod, chown, and chgrp commands

```
$ ls -l
total 4
-rw-r--r-- 1 danc users 0 2013-10-23 23:54 file1
-rw-r--r-- 1 danc users 5 2013-10-23 23:54 file2.txt
$ chmod a+x file1 ; ls -l
total 4
-rwxr--xr-x 1 danc users 0 2013-10-23 23:54 file1
-rw-r--r-- 1 danc users 5 2013-10-23 23:54 file2.txt
$ chmod g-rw file1 ; ls -l
total 4
-rwx--xr-x 1 danc users 0 2013-10-23 23:54 file1
-rw-r--r-- 1 danc users 5 2013-10-23 23:54 file2.txt
$ chmod 766 file2.txt ; ls -l
total 4
-rwx--xr-x 1 danc users 0 2013-10-23 23:54 file1
-rwxrw-rw- 1 danc users 5 2013-10-23 23:54 file2.txt
$ chown jane.users file1 ; ls -l
total 4
-rwx--xr-x 1 jane users 0 2013-10-23 23:54 file1
-rwxrw-rw- 1 danc users 5 2013-10-23 23:54 file2.txt
$ chgrp staff file2.txt ; ls -l
total 4
-rwx--xr-x 1 jane users 0 2013-10-23 23:54 file1
-rwxrw-rw- 1 danc staff 5 2013-10-23 23:54 file2.txt
```

165

The structure of a UNIX file system

⇒ A UNIX partition can contain:

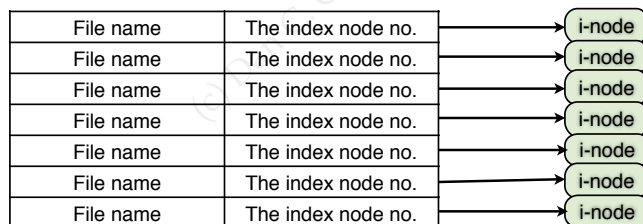


- **The boot block** - programs that load the UNIX operating system.
- **Superblock** - general information about the file system: the start of the next areas on the disk, the start of the free blocks.
- **Index nodes area** - contains an entry for each *file* (in a larger sense) in the partition
- The last area stores the actual data (files and directories).

166

⇒ Files and directories are stored in a tree-like structure

⇒ A directory is a special file - a table where each entry describes a file in the respective directory:



167

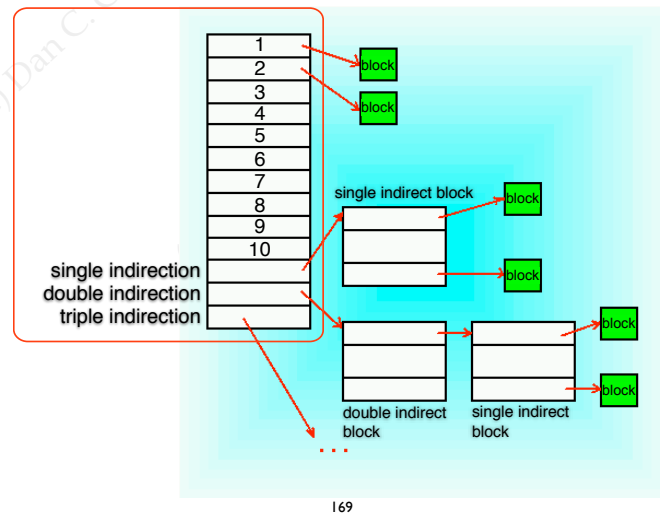
⇒ The index node (i-node)

→ stores information about a physical file

- **user ID: uid (user-id).** The ID of the file's owner
- **group ID**
- **access rights.** Three types of rights (*r-read, w-write, x-execute*) grouped in three categories:
 - *user* - owner rights
 - *group* - the rights for the users belonging to the owner group
 - *others* - all other users
- **last access time**
- **last modification time**
- **last i-node modification time**
- **file type.** Types of files: regular (-), directories (d), peripherals (c), etc.
- **file size (in bytes)**
- **link count.** The number of hard links that point to this file. EUsed when removing the file.
- **the list of data blocks** for the file

168

- **the list of data blocks** for the file



➡ The index node (i-node)

→ stores information about a physical file

- **user ID:** *uid (user-id).* The ID of the file's owner
- **group ID**
- **access rights.** Three types of rights (*r-read, w-write, x-execute*) grouped in three categories:
 - *user* - owner rights
 - *group* - the rights for the users belonging to the owner group
 - *others* - all other users
- **last access time**
- **last modification time**
- **last i-node modification time**
- **file type.** Types of files: regular (-), directories (d), peripherals (c), etc.
- **file size (in bytes)**
- **link count.** The number of hard links that point to this file. EUUsed when removing the file.
- **the list of data blocks** for the file

170

File types in UNIX

- **File**
 - the file is used in UNIX as a general, unifying concept that represents various logical and physical resources
- **Types of files**
 - regular file
 - directory
 - symbolic link
 - FIFO (named pipe)
 - socket
 - character device
 - block device
 - ... (depending on the UNIX variant several other file types may exist)

171

Devices (peripherals)

➡ Represented by files in the /dev directory

→ /dev/sda, /dev/sdb ...

→ /dev/sda1, /dev/sda2, /dev/disk/by-id/scsi-SATA_ST3250820AS_9QE499JB-part5

→ /dev/cdrom

→ /dev/dvdrw

```
dd if=/dev/sdb2 of=backup-partition2.img bs=1024
```

```
strings /dev/sda3 > strings_on_attacked_rootpartition.txt
```

172

Virtual devices

→ /dev/random /dev/urandom

→ /dev/null

→ /dev/zero

→ /dev/full

```
sh myscript >/dev/null 2>/dev/null
```

```
dd if=/dev/zero of=foobar count=1024 bs=1024
```

173

Links

→ The UNIX file system allows creating links to files

≈ alternative names for the same file

→ Two types of links:

→ Hard links

→ Symbolic links

174

Hard links

→ several directory entries referring the **same** i-node
 → references are possible only within the same file system (partition)

→ cannot point to directories

→ to avoid circular dependencies; exception: newer HFS+ versions (OS X), don't used by the automatic backup system

→ the i-node stores a **link count**

→ used when creating and removing the file; file is deleted only when link count = 1

Directory1

a.txt	i-node number	→	i-node
abc.sh	i-node number	→	i-node
script.sh	i-node number	→	i-node

Directory2

fisier1	i-node number	→	i-node
abc.sh	i-node number	→	i-node
mailsystem.log	i-node number	→	i-node

175

Hard links

```
$ touch abc
$ ls -l
total 0
-rw-r--r-- 1 danc staff 0 23 Oct 17:24 abc
$ ln abc abc1
$ ls -l
total 0
-rw-r--r-- 2 danc staff 0 23 Oct 17:24 abc
-rw-r--r-- 2 danc staff 0 23 Oct 17:24 abc1
$ ln abc xyz
$ ls -l
total 0
-rw-r--r-- 3 danc staff 0 23 Oct 17:24 abc
-rw-r--r-- 3 danc staff 0 23 Oct 17:24 abc1
-rw-r--r-- 3 danc staff 0 23 Oct 17:24 xyz
$ rm abc
$ ls -l
total 0
-rw-r--r-- 2 danc staff 0 23 Oct 17:24 abc1
-rw-r--r-- 2 danc staff 0 23 Oct 17:24 xyz
```

176

Symbolic links

- a special type of file, referring an existing **file**
- can point to files in other file systems (partitions)
- can point to directories
- the symbolic link file has its own i-node and occupies space on the disk
- if the referred file is removed, the symlink will still exist but it will point to an invalid location

177

Symbolic links

```
$ echo "text" > abc
$ cat abc
text
$ ln -s abc link1
$ cat link1
text
$ ls -l
total 8
-rw-r--r-- 1 danc  staff  0 23 Oct 17:34 abc
lrwxr-xr-x 1 danc  staff  3 23 Oct 17:34 link1 -> abc
$ ln -s abc link2
$ ls -l
total 16
-rw-r--r-- 1 danc  staff  0 23 Oct 17:34 abc
lrwxr-xr-x 1 danc  staff  3 23 Oct 17:34 link1 -> abc
lrwxr-xr-x 1 danc  staff  3 23 Oct 17:34 link2 -> abc
$ rm abc
$ ls -l
total 16
lrwxr-xr-x 1 danc  staff  3 23 Oct 17:34 link1 -> abc
lrwxr-xr-x 1 danc  staff  3 23 Oct 17:34 link2 -> abc
```

178

Again, a bit of programming...

179

A program

- **recursively** scans a directory given as a command-line argument
- prints
 - for symbolic links: name and the referred path
 - other files: name
 - if file is executable, appends * to its name
 - indents the printing according to the current tree depth

180

```

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <string.h>
#include <errno.h>
#include <dirent.h>
#include <unistd.h>
#include <sys/stat.h>
#include <limits.h>

void parcurge(char *nume_dir, int nivel)
{
    DIR *dir;
    struct dirent *in;
    char *nume;
    struct stat info;
    char cale[PATH_MAX], cale_link[PATH_MAX + 1], spatii[PATH_MAX];
    int n;

    memset(spatii, ' ', 2*nivel);
    spatii[2*nivel]='\0';

    if(!(dir = opendir(nume_dir)))
    {
        printf("%s: ", nume_dir); fflush(stdout);
        perror("opendir");
        exit(1);
    }

```

181

```

printf("%sDIR %s:\n", spatii, nume_dir);

while((in = readdir(dir))>0)
{
    nume = in->d_name;
    if(strcmp(nume, ".") == 0 || strcmp(nume, "..")==0)
        continue;

    sprintf(cale, "%s/%s", nume_dir, nume);
    snprintf(cale, sizeof(cale), "%s/%s", nume_dir, nume);

    if(lstat(cale, &info)<0)
    {
        printf("%s: ", cale); fflush(stdout);
        perror("error at lstat");
        exit(1);
    }

    if(S_ISDIR(info.st_mode))
        parcurge(cale, nivel + 1);
    else
    if(S_ISLNK(info.st_mode))
    {
        n = readlink(cale, cale_link, sizeof(cale_link));
        cale_link[n]='\0';
        printf("%s %s -> %s\n", spatii, cale, cale_link);
    }
    else

```

182

```

        {
            printf("%s %s", spatii, cale);
            if(info.st_mode & S_IXUSR || info.st_mode & S_IXGRP ||
info.st_mode & S_IXOTH)
                printf("*");
            printf("\n");
        }
        closedir(dir);
    }

    int main(int argc, char *argv[])
    {
        if(argc != 2)
        {
            printf("Usage: %s directory\n", argv[0]);
            exit(1);
        }

        parcurge(argv[1], 0);

        return 0;
    }

```

183

```

$ ./rd /etc
DIR /etc:
/etc/AFP.conf
/etc/afpovertcp.cfg
/etc/aliases -> postfix/aliases
/etc/aliases.db
DIR /etc/apache2:
DIR /etc/apache2/extra:
/etc/apache2/extra/httpd-autoindex.conf
/etc/apache2/extra/httpd-dav.conf
/etc/apache2/extra/httpd-default.conf
/etc/apache2/extra/httpd-info.conf
/etc/apache2/extra/httpd-languages.conf
/etc/apache2/extra/httpd-manual.conf
/etc/apache2/extra/httpd-mpm.conf
/etc/apache2/extra/httpd-multilang-errordoc.conf
/etc/apache2/extra/httpd-ssl.conf
/etc/apache2/extra/httpd-userdir.conf
/etc/apache2/extra/httpd-vhosts.conf
/etc/apache2/httpd.conf
/etc/apache2/httpd.conf~previous
/etc/apache2/magic

```

184

How would you address the following tasks?

- find the file size
- only print files to which hard links were created
- find out the owner user rights
- find out the owner ID
- modify the access rights
- remove the file
- find out info about a file pointed to by a symbolic link

How do you find out...

- what header files to include (#include)?
- what does a system call return?
- what are the C macros for finding out the file type?

185

4. Processes

186

Main concepts

187

Multitasking

= the ability of doing several tasks at the same time

188

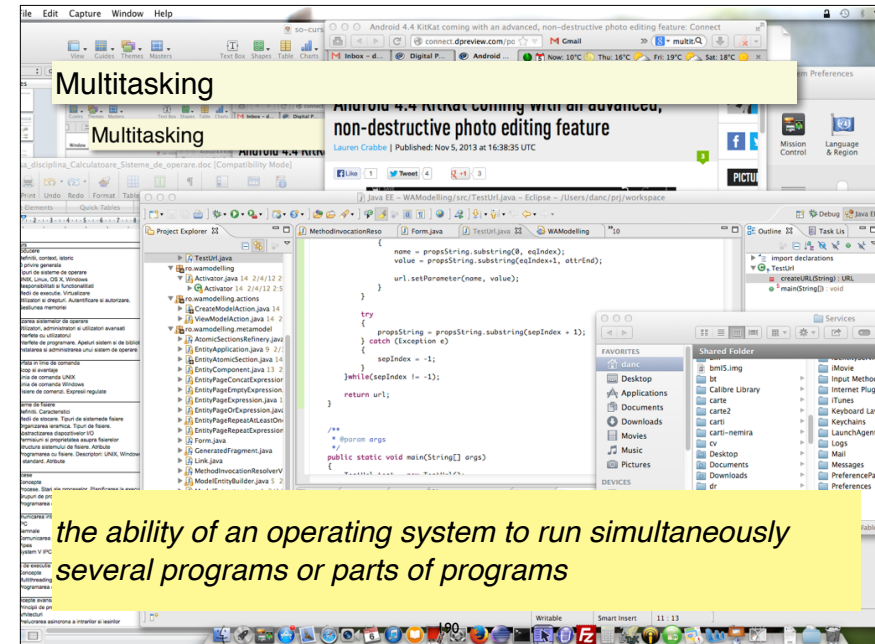
Multitasking

= the ability of doing several tasks at the same time



the ability of an operating system to run simultaneously several programs or parts of programs

189



the ability of an operating system to run simultaneously several programs or parts of programs

the ability of an operating system to run simultaneously several programs or parts of programs

191

the ability of an operating system to run simultaneously several programs or parts of programs

192

programs parts of programs

193

Processes

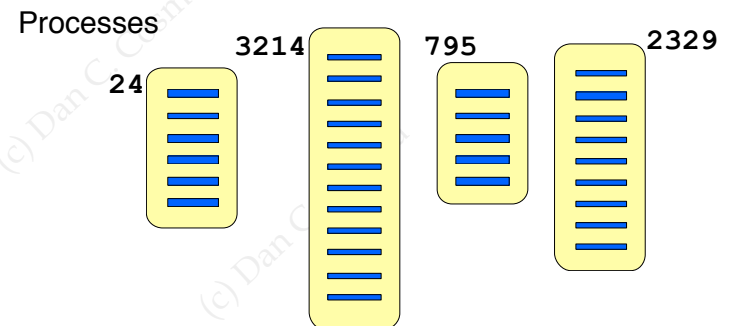
194

Processes

Process = the basic concept used by the OS for modeling concurrent software entities

Process = a program or a part of a program running under the supervision of the operating system

195

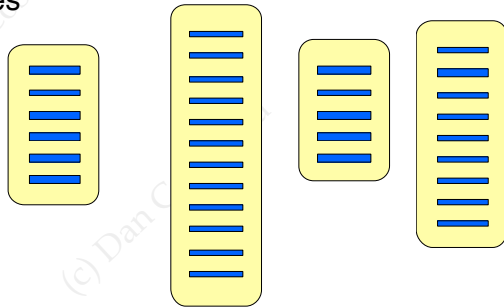


Processes are identified by numbers ([process identifier - PID](#))

At any time, only one process is assigned a given ID, but IDs are reused after the processes end

196

Processes



At any given time, in an OS can run

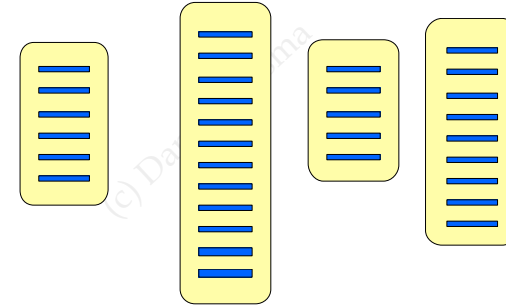
several processes

- system processes
- user processes

197

several processes

One processor ?
N processors ?

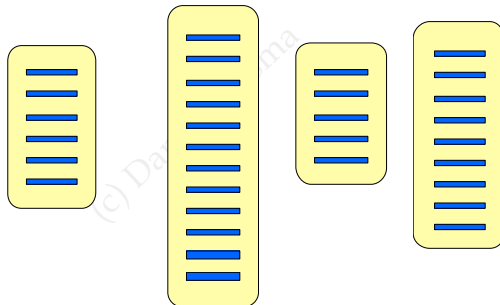


198

several processes

One processor ?
N processors ?

How?!



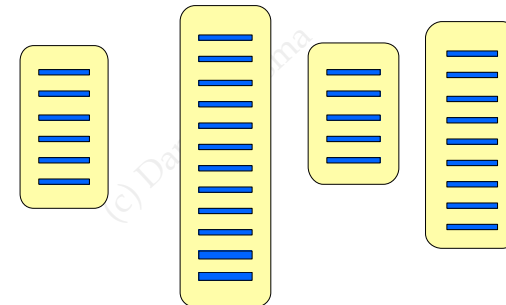
199

several processes

One processor ?
N processors ?

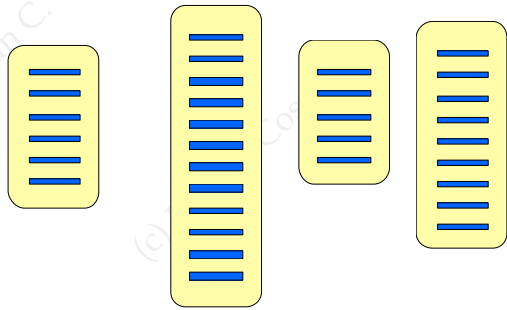
How?!

Process Scheduling



200

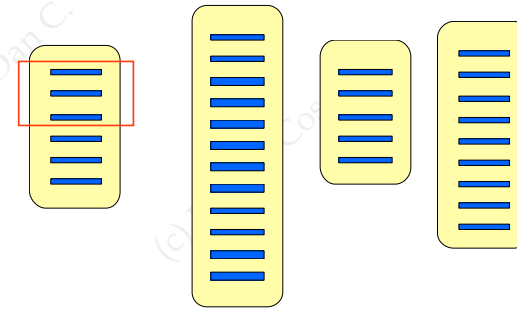
Process Scheduling



time

201

Process Scheduling

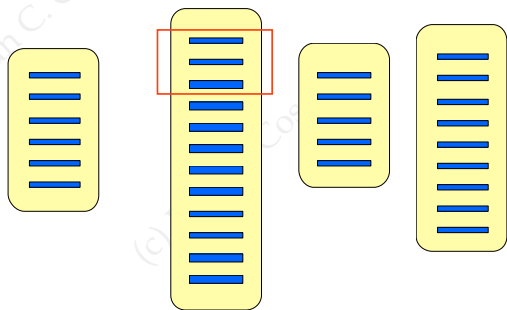


t_1

time

202

Process Scheduling

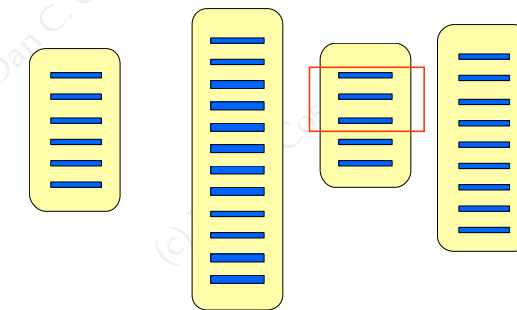


t_2

time

203

Process Scheduling

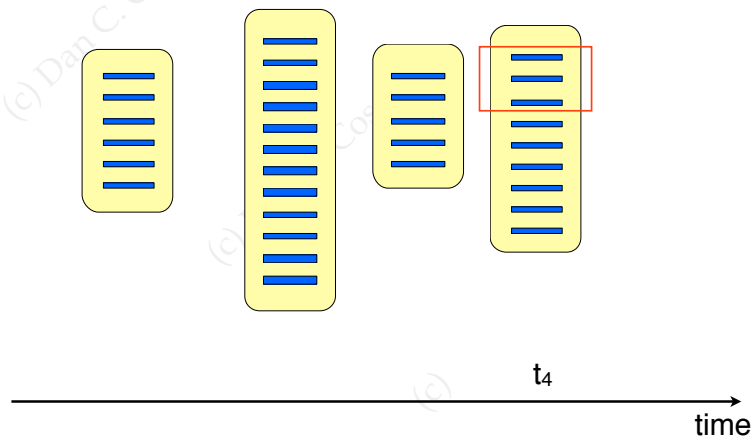


t_3

time

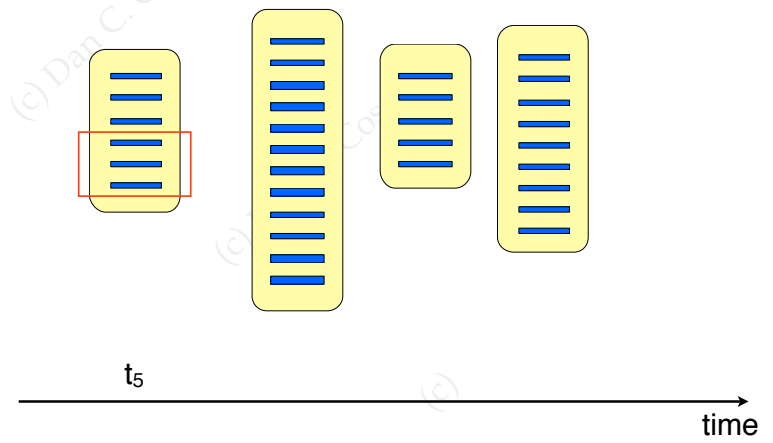
204

Process Scheduling



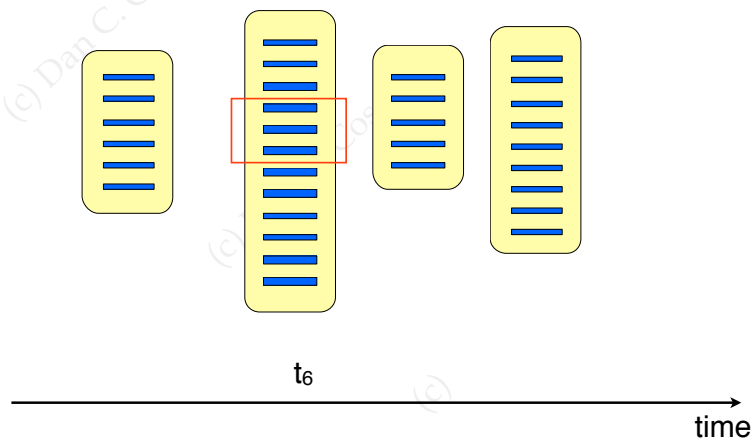
205

Process Scheduling



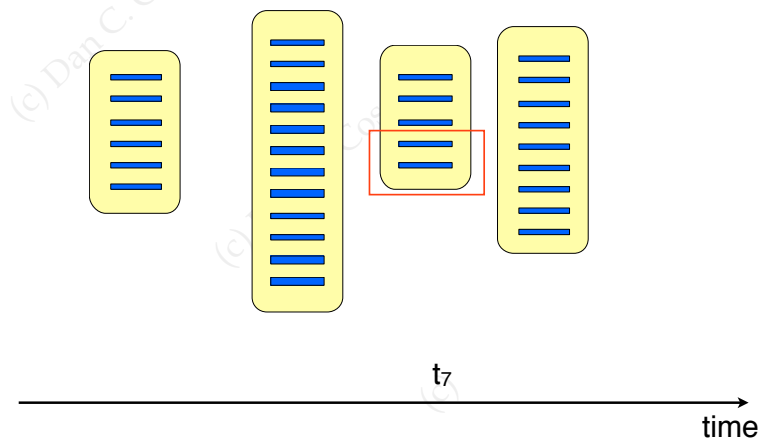
206

Process Scheduling



207

Process Scheduling



208

...

209

→ process scheduling algorithms

- implemented within the OS kernel which consequently becomes a process *dispatcher*
- can use various strategies: "round-robin", priority-based, etc.

210

*The process execution is coordinated by the operating system, which is responsible for managing the entire process **life cycle***

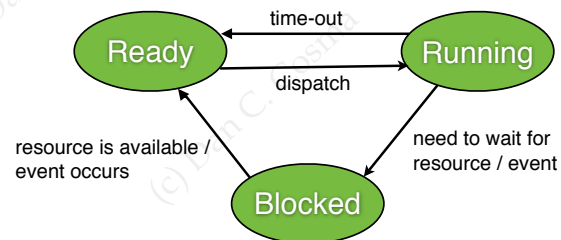
211

Process states

During its life, a process can be in one of the following **main states**:

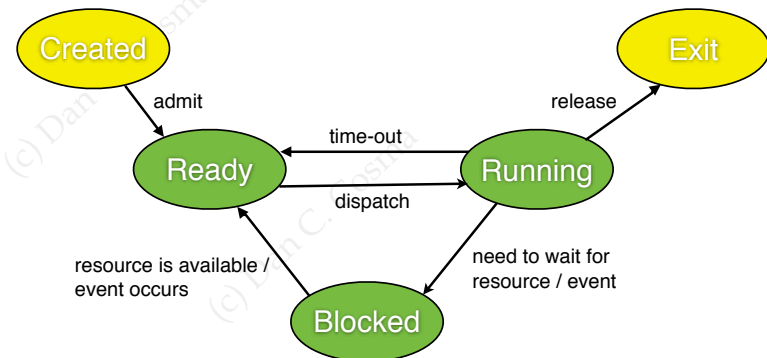
- Ready to run (Ready, Runnable)
→ the process can be run, but it is not its time yet
- In execution (Running)
→ the process runs
- Blocked (Blocked / Waiting)
→ the process is blocked waiting resources or events (example: input from keyboard, from a file, etc.)

212



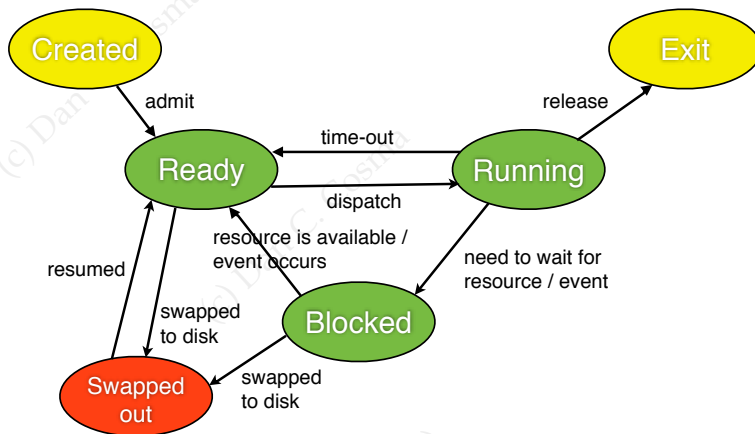
După: W. Stallings, *Operating Systems: Internals and Design Principles*, 7th edition, Prentice Hall, 2011

213



După: W. Stallings, *Operating Systems: Internals and Design Principles*, 7th edition, Prentice Hall, 2011

214



După: W. Stallings, *Operating Systems: Internals and Design Principles*, 7th edition, Prentice Hall, 2011

215

Process states (another perspective)

- extras from the `ps` man page in Linux -

PROCESS STATE CODES

Here are the different values that the `s`, `stat` and `state` output specifiers (header "STAT" or "S") will display to describe the state of a process.

D Uninterruptible sleep (usually IO)
 R Running or runnable (on run queue)
 S Interruptible sleep (waiting for an event to complete)
 T Stopped, either by a job control signal or because it is being traced.
 W paging (not valid since the 2.6.xx kernel)
 X dead (should never be seen)
 Z Defunct ("zombie") process, terminated but not reaped by its parent.

216

The `ps` command (Process Status)

→ prints the process list and process information

```
$ ps aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1   0.0  0.1 39348  4460 ?        Ss   Nov06   0:02 /sbin/init
root         2   0.0  0.0      0     0 ?        S    Nov06   0:00 [kthreadd]
root         3   0.0  0.0      0     0 ?        S    Nov06   0:00 [ksoftirqd/0]
root         6   0.0  0.0      0     0 ?        S    Nov06   0:00 [migration/0]
root         7   0.0  0.0      0     0 ?        SN   Nov06   0:00 [rcu0]
root        45   0.0  0.0      0     0 ?        S    Nov06   0:00 [scsi_eh_4]
root       1433   0.0  0.0 21052   612 ?        Ss   Nov06   0:00 /sbin/rpcbind
root       1435   0.0  0.0 62216  2192 ?        Ss   Nov06   0:02 /usr/sbin/nmbd -D -s /etc/samba/smb.conf
danc      2185   0.0  0.0 23256   704 ?        Ss   Nov06   0:00 /usr/bin/gpg-agent --sh --daemon --write-env-file /home/d
danc      2186   0.0  0.0 18568   528 ?        Ss   Nov06   0:00 /usr/bin/ssh-agent /etc/X11/xinit/xinitrc
danc      2198   0.0  0.0 20092   876 ?        S    Nov06   0:00 dbus-launch --sh-syntax --exit-with-session
danc      2199   0.0  0.0 22944  1820 ?        Ss   Nov06   0:00 /bin/dbus-daemon --fork --print-pid 5 --print-address 7 -
root      2318   0.0  0.1 164652 4440 ?        Sl   Nov06   0:00 /usr/lib/upower/upowerd
danc      2353   0.0  0.2 134304 7048 ?        S    Nov06   0:00 /usr/lib/xfce4/panel/wrapper /usr/lib64/xfce4/panel/plugi
danc      2361   0.1  0.3 112224 8904 ?        S    Nov06   0:07 /usr/lib/xfce4/panel-plugins/xfce4-orsageclock-plugin 1 2
danc      2363   0.0  0.1 146904 3012 ?        Sl   Nov06   0:00 /usr/lib/gvfs/gvfs-afc-volume-monitor
danc      2367   0.0  0.0 58124  2624 ?        S    Nov06   0:00 /usr/lib/gvfs/gvfs-gphoto2-volume-monitor
danc      2380   0.0  0.1 48512  4092 ?        S    Nov06   0:00 /usr/lib/GConf/2/gconfd-2
danc      2387   0.0  0.1 58796 3880 ?        S    Nov06   0:00 /usr/lib/gvfs/gvfsd-trash --spawner :1.10 /org/gtk/gvfs/e
danc      2392   0.0  0.0 42512  2520 ?        S    Nov06   0:00 /usr/lib/gvfs/gvfsd-burn --spawner :1.10 /org/gtk/gvfs/ex
danc      2396   0.5  1.3 524352 37188 ?        Sl   Nov06   0:35 /usr/bin/python -OO /usr/bin/gmixer -d
root      3092   0.0  0.1 90004 3800 ?        Ss   Nov06   0:00 sshd: danc [priv]
danc      3100   0.0  0.0 90004  2092 ?        S    Nov06   0:00 sshd: danc@pts/0
danc      3101   0.0  0.1 20772 3508 pts/0    Ss   Nov06   0:00 -bash
root      3186   0.0  0.0      0     0 ?        S    Nov06   0:00 [flush-8:16]
danc      5611   1.2  0.0 35708 1936 ?        SN   00:01   0:06 grav -root
danc      6242   0.0  0.0 13252 1148 pts/0    R+   00:09   0:00 ps aux
```

this is only a part of the `ps` output

217

Data structures for processes

218

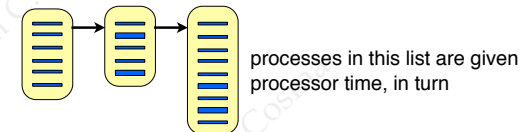
In order to manage processes, the OS maintains dedicated data structures.

Switching from one process to another (context switching, process switching) implies significant costs in time and resources

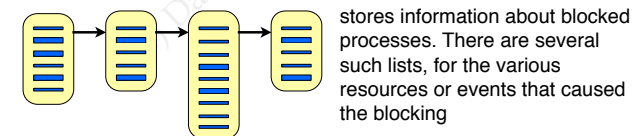
219

Data structures (examples)

List of ready processes



List of blocked processes



When a process is unblocked, it is moved to the ready process list

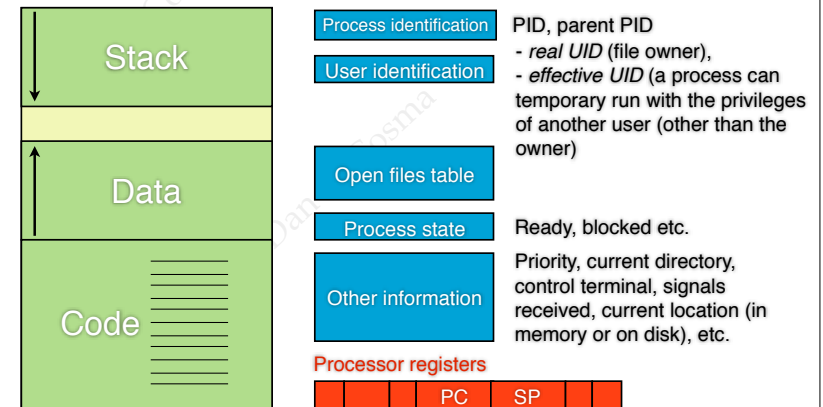
...

220

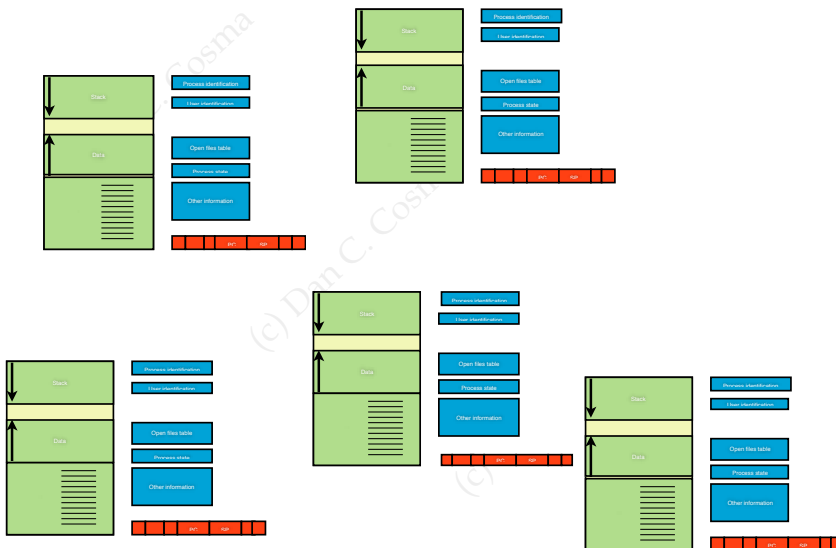
Each process is allocated separate memory areas and control/management

When a process changes state, the information about the process is saved; process memory areas can also be saved on disk if necessary

221



222



223

Creating processes

224

Creating processes

Any process can create a new process

- the created process is called child process
- the creator process is called parent process

→ This is the only way of generating new processes

- thus, each process will have a parent process
- a tree describing the parent-child relationships is therefore created throughout the system

When the system starts, an initial process is automatically created: the **init process**

- init has PID = 1
- it is the root of the entire process tree

225

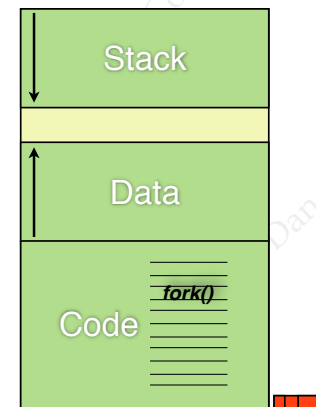
The `fork()` system function

→ creates a child process

```
#include <unistd.h>
```

```
pid_t fork(void);
```

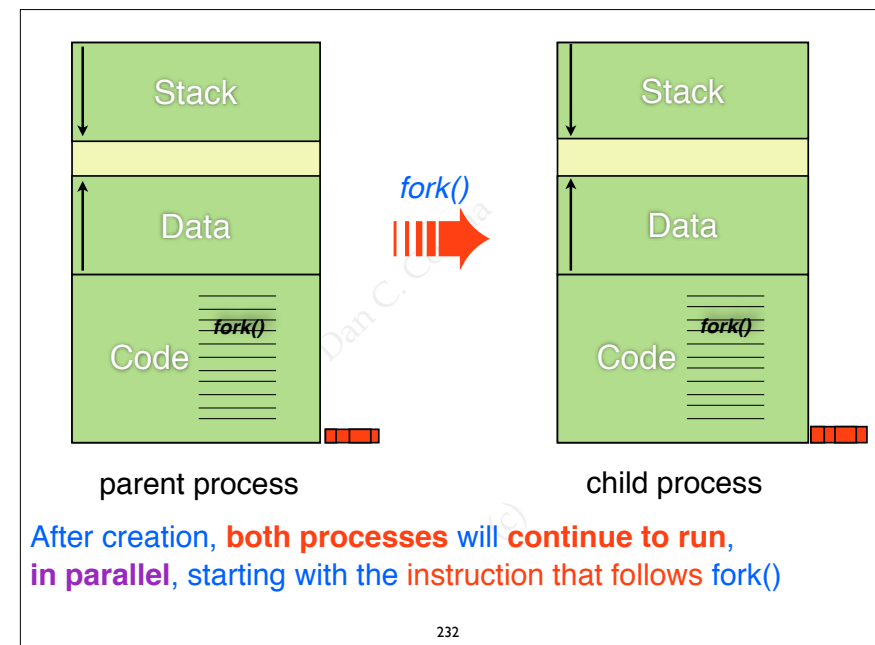
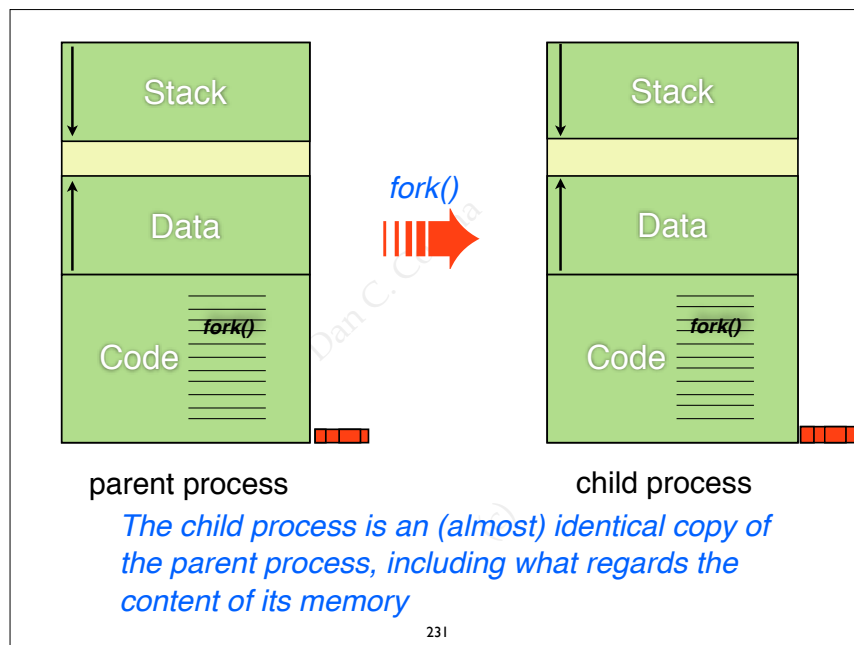
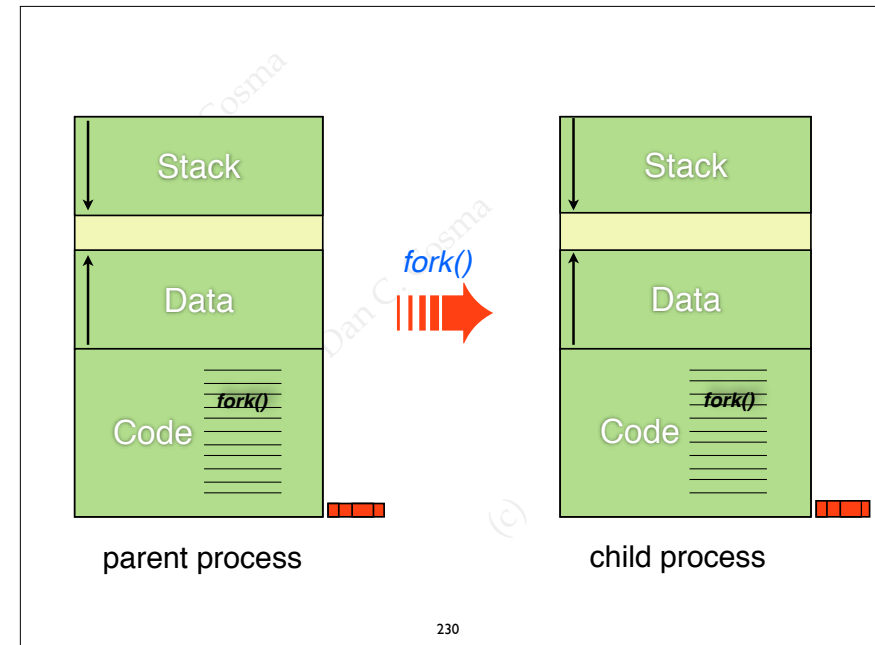
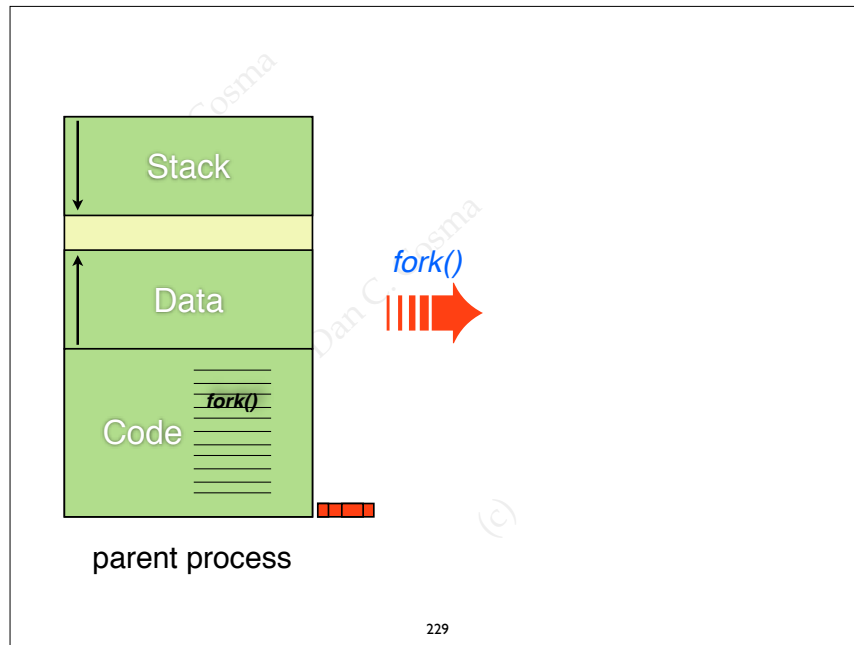
226



parent process

228

227



After creation, **both processes will continue to run, in parallel**, starting with the instruction that follows `fork()`

233

After process creation

234

After process creation

Two identical yet independent processes exist

→ they have separate memory areas, stacks, registers, etc.

The child process inherits from its parent

→ all data (global variables), having the values available in parent immediately before `fork()`

→ current program counter, call stack, local variables

→ open files table: all files open in parent before `fork()` will be accessible and usable by the child process

235

After process creation

Two identical yet independent processes exist

→ they have separate memory areas, stacks, registers, etc.

The child process inherits from its parent

→ all data (global variables), having the values available in parent immediately before `fork()`

→ current program counter, call stack, local variables

→ open files table: all files open in parent before `fork()` will be accessible and usable by the child process

What is the difference ?

236

After process creation

What is the difference ?

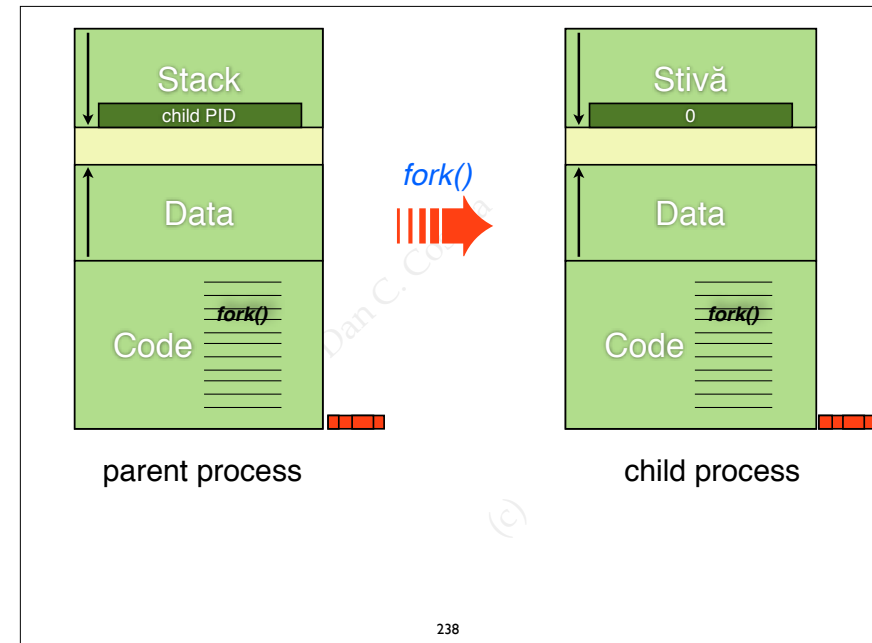
fork() returns different values in parent and child:

→ in the child process, returns 0

→ in parent returns the PID of the newly created child process

On error, fork() returns -1 and does not create a new process

237



238

Therefore, a program can do as follows:

```
...
if( ( pid=fork() ) < 0)
{
    perror("Eroare");
    exit(1);
}
if(pid==0)
{
    /* codul fiului */
    ...
    exit(0);
}
/* codul parintelui */
...
```

⇒ the parent code and the child code will behave differently

239



What is *the effect* of the following code?

```
...
fork();
printf("a");
...
```

240



What is *the effect* of the following code?

```
...
int i;
for(i=0; i<=10; i++)
    fork();
...
```



241

The `exec...()` calls

- useful for implementing completely distinct processes (not process copies)
- usually called immediately after `fork()`, in the child process
- `exec...()` loads a program from the disk, and uses it to overwrite the current process, wiping out its memory areas (code, data, ...)

242

The `exec...()` calls

Notes

- the loaded will start with its first instruction (e.g., with its `main()` function)
- most of the process attributes are preserved
 - process identifier (PID)
 - the parent-child relationship (Parent PID - PPID),
 - pending signals, time remaining to alarm
 - open files and file redirections (except for files opened by specifying the `FD_CLOEXEC` flag)
 - real UID, control terminal, current directory, root directory, priority etc.
- if successful, `exec` does not return (cannot return, as it was overwritten)
- on error, the function returns (and the return value is -1)
- `fork()` and `exec...()` combined provide flexibility in process creating

243

The `exec...()` functions

```
#include <unistd.h>

extern char **environ;

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execlx(const char *path, const char *arg,
    ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
    char *const envp[]);
```

of the above, only `execve()` is a system call, the rest are library functions

244

exec[l][p][v][e]

program arguments are given as a NULL-terminated list

program arguments are given as a NULL-terminated vector

the program is looked for in the paths specified by \$PATH

the environment of the program is given as a parameter to exec

245

Example: print the environment variables

```
#include <stdio.h>
extern char **environ;

int main(int argc, char *argv[])
{
    char **p;
    p = environ;
    while(*p)
    {
        printf("%s\n", *p);
        p++;
    }
    printf("\n\n-----\n\n");

    /**
     * the same code can be written as follows:
     */
    for(p=environ; *p; p++)
        printf("%s\n", *p);
}
```

246

Example

```
int main(int argc, char *argv[])
{
    pid_t pid;
    if((pid=fork())<0)
    {
        printf("Eroare la fork\n");
        exit(1);
    }
    if(pid==0) /* child process */
    {
        execlp("ls","ls","-l",NULL); /* process will run
                                         the ls command */
        printf("Eroare la exec\n");
        /* If execlp returned, the program
           could not be launched */
    }
    else /* parent process */
    {
        printf("Proces parinte\n");
        exit(0);
    }
}
```

247

Getting the process return value

● Return value

• Return value

→ At termination, any program **process** returns an integer value to the operating system
 → C: `exit(valoare)`; or `return valoare`; in `main()`

• Convention:

→ 0: process ended correctly
 → ≠0: process ended with error (and the value is the error code)

248

- A parent process must read the values returned by its child processes

- the termination status of the child process is thus verified
- processes for which the parent hasn't read the value (yet) are stored by the system even after termination, as "zombie processes"
- processes whose parent ends without reading the return value are adopted by the *init* process.

- Reading the return value

- any process can call the `wait()` and `waitpid()` functions to read the status returned by one of its child processes

249

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);

pid_t waitpid(pid_t pid, int *status, int options);
```

- `wait()`

- blocks until one (any) of the child process ends
- fills the status with information about the ended child process, including the return value
- to read the information stored in status specific macros are available:
 - `WIFEXITED(status)`
 - returns true if the child ended normally, i.e., by calling `exit()` or by returning a value in `main()`
 - `WEXITSTATUS(status)`
 - the status returned by the child process terminat

- `waitpid()`

- like `wait()` but waits for a specific child process, identified by its PID

250

The `exit()` library function

```
#include <stdlib.h>
```

```
void exit(int status);
```

- ends the current process and returns the value given as argument
- before termination all open files are closed, including the streams specific to the stdio library (FILE *)
- at termination, calls the functions previously installed by calls to `atexit()` or `on_exit()`

The `_exit()` system call

```
#include <unistd.h>
```

```
void _exit(int status);
```

- ends the current process and returns the value given as an argument
- closes all open files, **without** closing the streams specific to the stdio library (FILE *) This means no streams are flushed, data can be lost.
- **no** functions like `atexit()` or `on_exit()` are called

251

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
void process(char chr, int n){
    int i;
    for(i=0; i<=n-1; i++){
        printf("%c", chr);
    }
}
int main(){
    pid_t pid; int status;

    if((pid=fork())<0){
        printf("Error creating child process\n"); exit(1);
    }
    if(pid==0) /* procesul fiu */ {
        process('c', 2000);
        exit(0);
    }
    /* procesul părinte */
    process('p', 3000);
    wait(&status);
    if(WIFEXITED(status))
        printf("\nChild ended with code %d\n", WEXITSTATUS(status));
    else
        printf("\nChild ended abnormally\n");
}
```

252

Question

Program that starts two commands:

a) in parallel

b) sequential



253

a)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    pid_t pid1, pid2, wpid;
    char *arg1[]={ "echo", "a", "b", "c", NULL };
    char *arg2[]={ "ls", "-l", ".", NULL };
    int i, status;

    if((pid1=fork())<0)
    {
        printf("Eroare la fork\n");
        exit(1);
    }
    if(pid1==0) /* procesul fiu 1 */
    {
        execvp("echo", arg1);
        printf("Eroare la exec\n");
        exit(2);
    }
    /* procesul parinte */
    if((pid2=fork())<0)
    {
        printf("Eroare la fork\n");
        exit(1);
    }
    if(pid2==0) /* procesul fiu 2 */
    {
        execvp("ls", arg2);
        printf("Eroare la exec\n");
        exit(2);
    }
    /* din nou procesul parinte */
    for (i=1; i<=2; i++)
    {
        wpid = wait(&status);
        if(WIFEXITED(status))
            printf("\nChild %d ended with code %d\n", wpid, WEXITSTATUS(status));
        else
            printf("\nChild %d ended abnormally\n", wpid);
    }
}
```

254

a)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    pid_t pid1, pid2, wpid;
    char *arg1[]={ "echo", "a", "b", "c", NULL };
    char *arg2[]={ "ls", "-l", ".", NULL };
    int i, status;

    if((pid1=fork())<0)
    {
        printf("Eroare la fork\n");
        exit(1);
    }
    if(pid1==0) /* procesul fiu 1 */
    {
        execvp("echo", arg1);
        printf("Eroare la exec\n");
        exit(2);
    }
    /* procesul parinte */
    if((pid2=fork())<0)
    {
        printf("Eroare la fork\n");
        exit(1);
    }
    if(pid2==0) /* procesul fiu 2 */
    {
        execvp("ls", arg2);
        printf("Eroare la exec\n");
        exit(2);
    }
    /* din nou procesul parinte */
    for (i=1; i<=2; i++)
    {
        wpid = wait(&status);
        if(WIFEXITED(status))
            printf("\nChild %d ended with code %d\n", wpid, WEXITSTATUS(status));
        else
            printf("\nChild %d ended abnormally\n", wpid);
    }
}
```

code duplication?!

255

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    pid_t pid[2], wpid;

    char *arg1[]={ "echo", "a", "b", "c", NULL };
    char *arg2[]={ "ls", "-l", ".", NULL };
    int i, status;

    char ** param[2];

    param[0] = arg1;
    param[1] = arg2;

    for(i=0; i<2; i++)
    {
        if((pid[i]=fork())<0)
        {
            printf("Eroare la fork\n");
            exit(1);
        }
        if(pid[i]==0) /* procesul fiu i */
        {
            execvp(param[i][0], param[i]);
            printf("Eroare la exec\n");
            exit(2);
        }
    }

    /* procesul parinte */
    printf("Processes started:\n");
    for(i=0; i<2; i++)
        printf("%d ", pid[i]);
    printf("\n");
    for (i=1; i<=2; i++)
    {
        wpid = wait(&status);
        if(WIFEXITED(status))
            printf("\nChild %d ended with code %d\n", wpid, WEXITSTATUS(status));
        else
            printf("\nChild %d ended abnormally\n", wpid);
    }
}
```

256

b)

homework...

257

Finding out the IDs of the current and parent process

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

Launching a system command

```
#include <stdlib.h>

int system(const char *command);
```

- uses fork and exec launch a command in a separate process that executes `/bin/sh -c command`
- waits for the command termination and returns its exit status

258

5. Signals

259

Main concepts

260

Signal

= a software-level interruption, used for modeling asynchronous events
→ signals are sent to processes
→ sources of signals: processes, the operating system (and may also be caused by hardware events)

261

UNIX signals have identifiers and names derived from the event they model. Examples (*man 7 signal, Linux*):

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard (CTRL C)
SIGQUIT	3	Core	Quit from keyboard (CTRL \)
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

262

A process can specify the actions to be taken upon receiving a signal:

- ignore the signal

- there are signals that cannot be ignored: SIGKILL, SIGSTOP
- ignoring signals that were caused by hardware can lead to undefined behaviors

- handle the signal

- the program must define signal handler for the target process
- the handler function must be registered to the kernel through calls like signal() or sigaction()
- when the signal arrives, the kernel will interrupt the process, and call the handler; after the handler ends, the process will resume at the point it was interrupted
- a signal occurrence can lead blocking system calls to be unblocked(example: read). In this case, the respective call will return an error code (-1), and the errno variable will be set to EINTR

- accept the default behavior for the signal

- for most signals, this means the termination of the process

263

The kill, killall commands

kill -SIGNAL PID

→ sends a signal to a process

killall -SIGNAL command

→ sends a signal to all processes that run a given command

→ If the signal is not specified, SIGTERM is generated

Examples:

```
kill -SIGUSR1 2346
killall -9 java
```

264

The signal() system call

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

265

The signal() system call

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

pointer to a function that
receives an int parameter

the returned value:
pointer to a function that
receives an int parameter

- specifies, for the current process, the way it will react to a signal occurrence or installs a signal handler function
- the sig parameter is the number of the signal
- the func parameter
 - is a pointer to the signal handling function;
 - can also take the following values:
 - SIG_IGN : signal will be ignored
 - SIG_DFL : reset to the default behavior for the respective signal
- the function returns the old value of the handler function (can also be one of SIG_IGN, SIG_DFL) or SIG_ERR if an error occurred.

266

After the signal handler is installed, any such signal sent to the process that installed the handler will lead to the asynchronous execution of the handler function.

The handler function will be given (as an argument) the number of the signal that occurred. A function can be installed as a handler for multiple signals, therefore this parameter is useful for implementing different behaviors for different signals.

267

Example

A program made of two processes, parent and child. The parent (process a) counts continuously starting with zero, until it is interrupted by the user. The interruption is done by generating the SIGINT signal, explicitly (using the kill command) or implicitly (pressing Ctrl-C in the terminal on which the program runs in foreground). To properly view the results, the process calls usleep() at each step, generating a delay of about 1000 microseconds.

268

```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <sys/wait.h>

pid_t child_pid = 0;
int n = 0;

void process_a_ends(int sig)
{
    int status;

    if (kill(child_pid, SIGUSR2) < 0)
    {
        printf("Error sending SIGUSR2 to child\n");
        exit(2);
    }

    /* waiting for the child to end */

    wait(&status);
    printf("Child ended with code %d\n", WEXITSTATUS(status));

    printf("Process a ends.\n");
    exit(0);
}

```

269

```

void process_a()
{
    int i;

    if (signal(SIGINT, process_a_ends) == SIG_ERR)
    {
        printf("Error setting handler for SIGTERM\n");
        exit(1);
    }
    for (i = 0;;i++)
    {
        usleep(1000);
        if (i%10 == 0)
        if (kill(child_pid, SIGUSR1) < 0)
        {
            printf("Error sending SIGUSR1 to child\n");
            exit(2);
        }
    }
}

void process_b_writes(int sig)
{
    printf("Process b received SIGUSR1: %d\n", ++n);
}

```

270

```

void process_b_ends(int sig)
{
    printf("Process b ends.\n");
    exit(0);
}

void process_b()
{
    /* Ignoring SIGINT. Process b will end only when receives SIGUSR2 */
    if (signal(SIGINT, SIG_IGN) == SIG_ERR) {
        printf("Error ignoring SIGINT in process b\n");
        exit(3);
    }
    /* Setting the signal handlers */
    if (signal(SIGUSR1, process_b_writes) == SIG_ERR) {
        printf("Error setting handler for SIGUSR1\n");
        exit(4);
    }
    if (signal(SIGUSR2, process_b_ends) == SIG_ERR) {
        printf("Error setting handler for SIGUSR2\n");
        exit(5);
    }

    /* Infinite loop; process b only responds to signals */
    while(1)
    {
    }
}

```

271

```

int main()
{
    /* First, ignore the user signals, to prevent interrupting the
    child process before setting the appropriate handlers */
    signal(SIGUSR1, SIG_IGN);
    signal(SIGUSR2, SIG_IGN);

    /* Creating the child process. A global variable is used to store
    the child process ID in order to be able to use it from the signal
    handlers */
    if ((child_pid = fork()) < 0) {
        printf("Error creating child process\n");
        exit(1);
    }
    if (child_pid == 0) { /* child process */
        process_b();
        exit(0);
    }
    else /* parent process */
    {
        process_a();
    }
    /* this is still the parent code */
    return 0;
}

```

272

The sigaction() system call

→ *recommended to be used instead of signal()*

```
#include <signal.h>
```

```
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
```

handler

handler, if the 3 parameter version is preferred (SA_SIGINFO set in sa_flags)

signals that must be blocked during the handler execution (bit mask, the sigsetops have to be used)

various options for the sigaction() call (for instance to control the behavior upon signal receipt)

unused (old)

273

- the **sa_flags** field (examples of options):

SA_NOCLDSTOP - if signum is SIGCHLD, the process will not get a SIGCHLD signal when the child process is suspended (for example using SIGSTOP), SIGCHLD will only be generated when the child process ends;

SA_NOMASK sau **SA_NODEFER** - the respective signal will not be automatically included in sa_mask (the default setting is to prevent the occurrence of a signal when executing the handler for the same signal);

SA_SIGINFO - specified when sa_sigaction is to be used instead of sa_handler.

274

The sigprocmask() system call

```
#include <signal.h>
```

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

→ reads or changes the bit mask specifying the blocked signals for the calling thread

how:

SIG_BLOCK - the signal set specified in the set argument is added to the current set of blocked signals

SIG_UNBLOCK - the signal set specified in the set argument is removed from the current set of blocked signals

SIG_SETMASK - the current set of blocked signals is replaced with the set specified in the set argument

set: the set of signals use by the call according to the how option

oldset: if not NULL, a pointer where the old set of blocked signal will be stored

275

Other functions

```
#include <sys/types.h>
#include <signal.h>
```

```
int kill(pid_t pid, int sig);
```

→ Sends the signal sig to the process pid

```
#include <signal.h>
```

```
int raise(int sig);
```

→ Sends the signal sig to the current process

```
#include <unistd.h>
```

```
unsigned int alarm(unsigned int seconds);
```

→ Installs an alarm; after *seconds* seconds, a SIGALRM signal will be generate to the current process

276

6. Pipes

277

Creating and using pipes

278

Pipe

- a UNIX inter-process communication primitive
- describes a data channel two processes can use to send data to each other

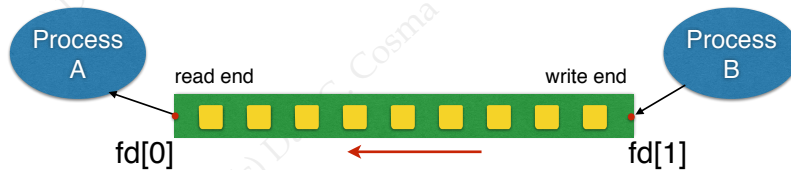
279

Pipe



280

Pipe



- The read and write ends are modeled as *file descriptors*
- A pipe is *unidirectional*

281

The pipe() system call

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

- Creates a pipe
- fills in the array given as argument with the pipe's descriptors:
 - pipefd[0]: read
 - pipefd[1]: write
- once the pipe is created, the current process will be able to read and write from/to the pipe, using read() and write()

282

- once the pipe is created, the current process will be able to read and write from/to the pipe, using read() and write()



283

- once the pipe is created, the current process will be able to read and write from/to the pipe, using read() and write()

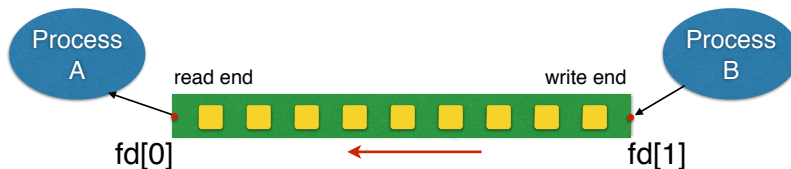
- The two ends of the pipe are handled like file descriptors
- File descriptors are *inherited* by the child process after fork()

A pipe is also inherited by a process from its parent if properly initialized before fork()

284

How to use pipe()

- the pipe is created by a process
- the process calls fork(): the child process will inherit the pipe, therefore will be able to use its descriptors (to read and write)
- the two processes (parent, child) agree on how the pipe will be used: one process writes, the other reads
- the agreement: **each process closes the unused descriptor**



285

```
...
int pfd[2];
int pid;
...
if(pipe(pfd)<0)
{ printf("Eroare la crearea pipe-ului\n"); exit(1); }
...
if((pid=fork())<0)
{ printf("Eroare la fork\n"); exit(1); }

if(pid==0) /* child process */
{
    close(pfd[0]); /*closes the read descriptor => process writes */
    ...
    write(pfd[1],buff,len); /* writing to the pipe */
    ...
    close(pfd[1]); /* at the end, closes the used descriptor, too */
    exit(0);
}
else /* parent process */
{
    close(pfd[1]); /* closes the write descriptor => process reads*/
    ...
    read(pfd[0],buff,len); /* reading from the pipe */
    ...
    close(pfd[0]); /* at the end, closes the used descriptor, too */
    exit(0);
}
```

286

Notes

- If a process reads from a pipe for which the write end is closed, read() will return 0
- If a process writes in a pipe for which the read end is closed, write() will fail, as follows:
 - the respective process will receive the SIGPIPE signal
 - if the process doesn't handle, block or ignore SIGPIPE, the process will be terminated; otherwise, the value returned by write() will be -1, and *errno* will be set to EPIPE

287

Notes

- Pipes can be inherited **by several processes** (all the process subtree of the process that created the pipe)
- A process **must always close all pipe descriptors it does not use**. If the process doesn't use the pipe at all, it must close both its descriptors
- Important *recommendation*: process/processes that write to pipe and the process/processes that read **must clearly agree, at each moment in time, the size and meaning of the exchanged packages of data**

Example: process A sends n bytes; the reading process B must read (wait for) exactly n bytes: no less, no more.

288

Example: process A sends n bytes; the reading process B must read (wait for) exactly n bytes: no less, no more.

Explanation:

- if process B reads (waits for) more than n bytes, the `read()` call may block waiting for data that will never come (if A does not send anymore) or will read data from a different (future) "transmission" that does not belong to the current exchange
- if B reads less than n bytes, then the data read at the current step will be incomplete, and unread data will remain in the pipe; this data will *probably* be mistakenly read in the future at a step that is meant to exchange a different set of data than the current one
- such protocol errors *may* lead to one or more processes become blocked (waiting forever data that will never come)
- of course, there may be cases when this recommendation can be ignored: nevertheless, you must know precisely what you're doing

289

Why unused pipe ends must always be closed ?

Explanation:

- Say process A is parent for B, A created the pipe before `fork()`
- A reads from pipe, B writes to pipe
- A "forgets" to close the write descriptor
- A reads data from the pipe in a loop

```
while ((n=read(pfd[0], buff, no_of_bytes))>0) {  
    ...  
}
```

- At some point, B ends its data transmission



A locks forever in `read()`, because `read()` will never return zero ("end of file") while there still is a process that, theoretically could write data to the pipe (has the pipe write descriptor open): this process is process A itself

290

Remember!

- create the pipe **before** `fork()`
- close all unused descriptors, for all pipes that are visible in the respective process
- do close the used ends, immediately after they are not needed (**why?**)
- define a precise communication protocol between the reader and writer processes (the number of bytes read by a reader must be exactly the same as the number of bytes written by its peer at the same moment)

291

File descriptor duplication and redirection

292

Duplicating a file descriptor

```
#include <unistd.h>
```

```
int dup(int oldfd);
```

- duplicates oldfd, creating a new descriptor which will point to the same file; the new descriptor is returned by the function call
- both descriptors will share the current file offset, open flags, etc.
- the new descriptor will always be the lower unused descriptor available

293

Example

```
...  
fd=open("Fisier.txt", O_WRONLY);  
...  
fd1=dup(fd);  
...  
write(fd1, "Un text", 8);  
...
```

294

Duplicating file descriptors with dup2()

```
#include <unistd.h>
```

```
int dup2(int oldfd, int newfd);
```

- duplicates oldfd, creating a new descriptor which will point to the same file;
- the new descriptor **will have the value given by the newfd** argument
- if newfd was already used for an open file, the file is closed, then the descriptor is given the new meaning
- returns the newly allocated descriptor (newfd)

295

Redirecting file descriptors

- setting a new meaning for an existing file descriptor, to point to a different file than the one it initially designated
- it is a particular case of duplication
- can be done by duplicating the descriptor that points the new file, while making sure that the descriptor value obtained through duplication is precisely the one of the descriptor that needs to be redirected

For example, a duplication that ensures that descriptor 1 corresponds to a file on the disk, effectively represents the redirection of the standard output

→ all calls that write "to the standard output" (example: printf) really write to the descriptor having the value 1 (STDOUT_FILENO); redirecting this descriptor, the effect of all these functions will be visible in the target file of the redirection

296

Example of redirecting the standard output

```
...
fd=open("Fisier.txt", O_WRONLY);
...
if((newfd=dup2(fd,1))<0)
{
    printf("Eroare la dup2\n");
    exit(1);
}
...
printf("ABCD");
...
```

297

Again, about pipe...

→ The two ends of a pipe are modeled as file descriptors

⇒ they can be used in duplications or redirections

→ for example, we can redirect

- the standard output: to the write end of a pipe
- the standard input: from the read end of a pipe

298

Example

→ connecting two processes through a pipe; one process runs (exec) a program from the disk

299

Example

```
void main()
{
    int pfd[2];
    int pid;
    FILE *stream;

    ...
    if(pipe(pfd)<0)
    {
        printf("Eroare la crearea pipe-ului\n");
        exit(1);
    }
    ...
    if((pid=fork())<0)
    {
        printf("Eroare la fork\n");
        exit(1);
    }
}
```

300

```

if(pid==0) /* child process */
{
    close(pfd[0]); /* closes the read descriptor; */
    /* process writes to pipe */
    ...
    dup2(pfd[1],1); /* redirects standard output to pipe*/
    ...
    execlp("ls","ls","-l",NULL); /* process runs ls*/
    printf("Eroare la exec\n");
}
else /* parent process */
{
    close(pfd[1]); /* closes the write descriptor; */
    /* process reads from the pipe */
    ...
    stream=fdopen(pfd[0],"r");
    /* opens a stream (FILE *) for the read descriptor */
    while(...)
    { ...
        fscanf(stream,"%s",string);
        /* reads from the pipe, using the associated stream */
        ...
    }
    ...
    close(pfd[0]); /* at the end, also closes the used descriptor */
    exit(0);
}
}

```

301

Note: the example uses fdopen()

```

#include <stdio.h>

FILE *fdopen(int fd, const char *mode);

```

→ associates stream of type FILE * (managed by the stdio library) to an open file designated by the *fd* integer (system-calls-specific) file descriptor
 → at the enf, the file must be closed with fclose(), and NOT with close() (to let the stdio library make the necessary cleanup, such as emptying the memory buffers to the disk). fclose() calls close() in its implementation.
 → the *mode* options must be compatible with the mode specified when opening the *fd* descriptor

302

Note

→ the pipe is the communication primitive used by the shell when chaining commands separated by the ' | ' operator

Exercise: write a simplified version of a program that provides the same effect as the following line:

prog1 | prog2

303

```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>

/* parent */
if((pid_b=fork())<0)
{
    printf("Eroare la fork\n");
    exit(1);
}

if(pid_b==0) /* b */
{
    close(pfd[1]);
    dup2(pfd[0],0);
    execlp(argv[2], argv[2], NULL);
    printf("Eroare la exec\n");
    exit(1);
}

/* b */
close(pfd[0]);
close(pfd[1]);

/* Parent process reads the results */
int status;
waitpid(pid_a, &status, 0);
waitpid(pid_b, &status, 0);

/* a simplified version of getting the return status */
if(WIFEXITED(status))
    return WEXITSTATUS(status);
else
    return 1;
return 0;
}

int pfd[2];

int main(int argc, char *argv[])
{
    int pid_a, pid_b;
    if(argc != 3)
    {
        printf("Utilizare: %s prog1 prog2\n", argv[0]);
        exit(1);
    }

    if(pipe(pfd)<0)
    {
        printf("Eroare la crearea pipe-ului\n");
        exit(1);
    }

    if((pid_a=fork())<0)
    {
        printf("Eroare la fork\n");
        exit(1);
    }

    if(pid_a==0) /* a */
    {
        close(pfd[0]);
        dup2(pfd[1],1);
        execlp(argv[1], argv[1], NULL);
        printf("Eroare la exec\n");
        exit(1);
    }
}

```

304

Named pipes

305

Named pipes

- pipes that can be explicitly created from the command line or programs, while associating names to them
- they are *visible in the file system* as special files, to which the normal read and write operations can be done
- reading and writing is done following the FIFO mechanism
- named pipes can be used explicitly, for instance in scripts, to communicate between processes, commands, to replace temporary files, etc.

Note: the pipes we previously discussed (those created calling pipe()) are called anonymous pipes (in contrast to named pipes)

306

Creating named pipes

```
mkfifo [-m mode] name
```

where mode: the access rights to the special FIFO file to be created

```
mkfifo --mode=0766 ~/tmp_pipe
```



```
#include <sys/types.h>
#include <sys/stat.h>
```

```
int mkfifo(const char *pathname, mode_t mode);
```

307

Example

```
$ mkfifo pipe1
$ wc -l < pipe1 > result.txt &
[1] 768
$ ls -l > pipe1
[1]+  Done                  wc -l < pipe1 > result.txt
$ cat result.txt
28
```

308

7. Threads

309

Multitasking

= the ability of doing several tasks at the same time

310

Multitasking

= the ability of doing several tasks at the same time

→ Processes

311

Multitasking

= the ability of doing several tasks at the same time

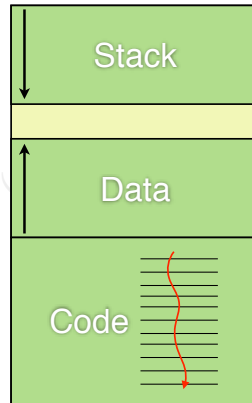
→ Processes

→ Threads

312

Thread

= a sequential execution inside a process



313

Several threads can exist inside the same process

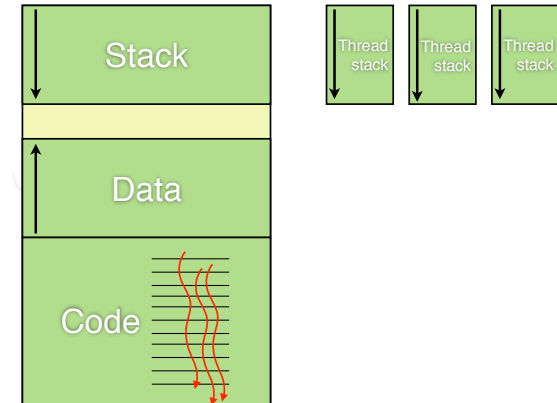
→ they run **in parallel** and execute different or even the same code

→ share the data area of the process,

but have separate stacks

← Consequences?

When created, a process has a single thread (main thread)



314

Advantages against processes

- Managing threads requires less resources
- Context switching (switching from one thread to another) is faster
- Threads can easily communicate to each other using the shared memory

315

Notes:

→ At creation time, a process is made of a single thread.

→ All threads inside a process run in parallel.

→ A **process** ends:

- when its main thread ends
- if a thread calls `exit()`
- when the `main()` function ends (therefore the main thread ends)
- if the process receives an un-handled signal

...

→ If a process made of multiple threads ends, **all its threads end**.

→ as they share the same data area, threads in a same process **share all global variables**. Local variables and function arguments are not shared, as the stacks are separate for each thread.

→ Many system and library calls have effect on the entire process, **consequently they will affect all its threads**, regardless of the thread that calls them. Example: the `sleep()` function.

316

Remark: the **pthread library** is used for working with threads.

→ usually, it is not linked automatically by gcc to the object code of the program, therefore this must be explicitly asked for (option `-lpthread`). In newer UNIX and gcc versions, thread support is directly included in the glibc library, and can be activated using the option `-pthread`

→ functions in this library usually return 0 when they ended correctly, and an error code otherwise

actually, the link editor

317

Thread identification

→ unique identifiers inside the process

```
#include <pthread.h>
```

```
pthread_t pthread_self(void);
```

→ gets the current thread ID

→ the actual definition of the pthread_t type depends on the implementation, it may be a data structure

318

Creating threads

```
#include <pthread.h>
```

```
int  
pthread_create(pthread_t *restrict thread,  
               const pthread_attr_t *restrict attr,  
               void *(*start_routine)(void *),  
               void *restrict arg);
```

address at which the function will store the ID of the newly created thread

attributes for creation (or NULL for the default attributes)

argument passed to the thread function

thread main function (thread body); obviously, this function may in turn call other functions in the program

→ creates a thread which will start immediately by calling the start_routine function with the argument arg.

Note: the **restrict** keyword tells that, for the entire life of the p pointer, only p or a pointer expressed directly using p (such as p + 1) is the only pointer that indicates the respective memory area. This information is used by the compiler for optimizations, and its validity through the code must be ensured by the programmer.

319

Other functions

```
#include <pthread.h>
```

```
int pthread_join(pthread_t thread, void **value_ptr);
```

→ waits for the thread thread to end, then gets its return value and writes it at the address value_ptr. The value_ptr argument can be NULL, if the return value is not needed.

→ can be called by any thread in the process

→ if the caller tries to wait for itself, or a similar circular dependency is detected, the function returns an error code

```
#include <pthread.h>
```

```
void pthread_exit(void *value_ptr);
```

→ ends the current thread, setting its return value to value_ptr.

320

Other functions

```
#include <pthread.h>
```

```
pthread_t pthread_self(void);
```

→ returns the current thread ID

```
#include <pthread.h>
```

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

→ returns non-zero if t1 and t2 represent the same thread, otherwise return zero.

321

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_code(void *arg)
{
    int i;
    for(i=0; i<1000; i++)
        printf("%s", (char *)arg);
    printf("\n");

    return (void *)((char *)arg) - 'A' + 1;
}

int main(int argc, char *argv[])
{
    pthread_t th1, th2;
    void *ret1, *ret2;

    pthread_create(&th1, NULL, thread_code, (void*) "A");
    pthread_create(&th2, NULL, thread_code, (void*) "B");

    printf("Threads created.\n");

    pthread_join(th1, &ret1);
    pthread_join(th2, &ret2);

    printf("Thread 1 ends returning: %d.\n", (int)ret1);
    printf("Thread 2 ends returning: %d.\n", (int)ret2);
    exit(0);

    return 0;
}
```

322

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
```

Another example. Is there a mistake?

```
void *thread_code(void *arg)
{
    int i;
    for(i=0; i<1000; i++)
        printf("%c", *((char *)arg) );
    printf("\n");

    return (void *)((char *)arg) - 'A' + 1;
}

int main(int argc, char *argv[])
{
    pthread_t th1, th2;
    void *ret1, *ret2;
    char c;

    c='A';
    pthread_create(&th1, NULL, thread_code, &c);
    c='B';
    pthread_create(&th2, NULL, thread_code, &c);

    printf("Threads created.\n");
    pthread_join(th1, &ret1);
    pthread_join(th2, &ret2);
    printf("Thread 1 ends returning: %d.\n", (int)ret1);
    printf("Thread 2 ends returning: %d.\n", (int)ret2);
    exit(0);

    return 0;
}
```



323

What effect has the following code:

```
for (i=0; i<100; i++)
    pthread_create(&th[i], NULL, thread_code, NULL);
```



324

Types of threads

- *joinable*
 - the value returned at termination can be read by another thread
 - the resources allocated for the thread are not released until another thread calls `join()` for it
- *detached*
 - cannot be joined by other threads
 - the resources allocated for the thread are released immediately when the thread ends

In most implementations, `pthread_create()` creates by default *joinable* threads

325

“Detached” threads

→ a *joinable* thread can be transformed in a *detached* one using:

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

Example:

```
pthread_detach(pthread_self());
```

→ a thread can be directly created as detached by setting attributes in the *attr* parameter of `pthread_create()`.

326

Setting attributes for `pthread_create()`

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);

int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int detachstate);

int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate);
```

```
int pthread_attr_destroy(pthread_attr_t *attr);
```

Notă: other attributes exist, but are outside the scope of this discussion

Steps:

1. Initialize an attribute variable with `pthread_attr_init()`
2. Set the desired attribute using the corresponding

`pthread_attr_set...()` call

3. After the thread was created, free the resources created for the attribute variable by calling `pthread_attr_destroy()`.

327

Example

```
...
pthread_attr_t attr;
...
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
...
pthread_create(...);
...
pthread_attr_destroy(&attr);
...
```

328

Notes

- Once a thread was marked as *detached*, it cannot be made *joinable*
- Setting the detached attribute only refers to the way system resources are allocated for those threads.
- Detached* threads **do not** remain in the system after the process ends
- For each thread created in a process (except the main one) either *pthread_join()*, or *pthread_detach()* must be called

329

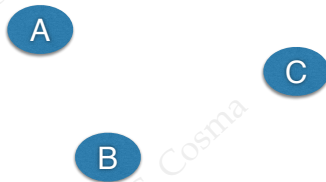
Revisiting pipes: an example

- program made of 3 processes, called in the command line:

program n file

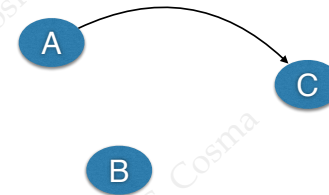
- process A: uses an external command to print the last *n* lines in file, and sends them to process C
- process B: sends 100 random numbers to process C then receives and prints the results from C
- process C receives, in turn, data from A and B, and:
 - counts lower case characters from A
 - finds the maximum even number received from B
 - sends the results to B, as soon as they are available

330



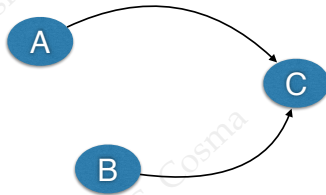
- process A: uses an external command to print the last *n* lines in file, and sends them to process C
- process B: sends 100 random numbers to process C then receives and prints the results from C
- process C receives, in turn, data from A and B, and:
 - counts lower case characters from A
 - finds the maximum even number received from B
 - sends the results to B, as soon as they are available

331



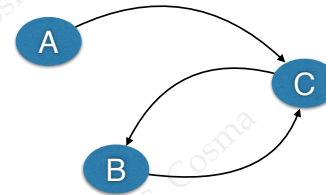
- process A: uses an external command to print the last *n* lines in file, and sends them to process C
- process B: sends 100 random numbers to process C then receives and prints the results from C
- process C receives, in turn, data from A and B, and:
 - counts lower case characters from A
 - finds the maximum even number received from B
 - sends the results to B, as soon as they are available

332



- process A: uses an external command to print the last n lines in file, and sends them to process C
- process B: sends 100 random numbers to process C then receives and prints the results from C
- process C receives, in turn, data from A and B, and:
 - counts lower case characters from A
 - finds the maximum even number received from B
 - sends the results to B, as soon as they are available

333



- process A: uses an external command to print the last n lines in file, and sends them to process C
- process B: sends 100 random numbers to process C then receives and prints the results from C
- process C receives, in turn, data from A and B, and:
 - counts lower case characters from A
 - finds the maximum even number received from B
 - sends the results to B, as soon as they are available

334

```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <time.h>

int pipe_ac[2], pipe_bc[2], results_pipe[2];

enum processes { COUNTER, MAXRANDOM };
struct result_data {
    enum processes type;
    long data;
};

void process_c()
{
    int count = 0, n1, n2, index;
    int num;
    long max = LONG_MIN;
    int end1=0, end2=0;
    char c;
    struct result_data res;

    close(pipe_ac[1]);
    close(pipe_bc[1]);
    close(results_pipe[0]);

    while(1) {
        if(!end1) {
            if((n1=read(pipe_ac[0], &c, sizeof(char)))>0) {
                printf("Eroare la citire din pipe a-c\n");
                exit(1);
            }
            if(n1 > 0) {
                if(islower(c)) {
                    count++;
                }
            }
            res.type = COUNTER;
            res.data = count;

            if(write(results_pipe[1], &res, sizeof(struct result_data))>0) {
                printf("Eroare la scriere in pipe rezultate\n");
                exit(1);
            }
            end1=1;
        }

        if(!end2) {
            if((n2=read(pipe_bc[0], &num, sizeof(long)))>0) {
                printf("Eroare la citire din pipe b-c\n");
                exit(1);
            }
            if(n2 > 0) {
                if(num % 2 == 0) {
                    if(num >= max) {
                        max = num;
                    }
                }
            }
            res.type = MAXRANDOM;
            res.data = max;

            if(write(results_pipe[1], &res, sizeof(struct result_data))>0) {
                printf("Eroare la scriere in pipe rezultate\n");
                exit(1);
            }
            end2=1;
        }
    }
}

int main(int argc, char *argv[])
{
    int pid_a, pid_c;

    if(argc != 3) {
        printf("Utilizare: %s nr_linii fisier\n", argv[0]);
        exit(1);
    }

    if(pipe(results_pipe)>0) {
        printf("Eroare la crearea pipe-ului\n");
        exit(1);
    }

    if(pipe(pipe_ac)>0) {
        printf("Eroare la crearea pipe-ului a-c\n");
        exit(1);
    }

    if(pipe(pipe_bc)>0) {
        printf("Eroare la crearea pipe-ului b-c\n");
        exit(1);
    }

    if(pid_a = fork()) {
        printf("Eroare la fork\n");
        exit(1);
    }

    if(pid_a == 0) {
        close(pipe_ac[0]);
        close(pipe_bc[0]);
        close(results_pipe[1]);
        dup2(pipe_ac[1], 1);
        execp("tail", "tail", "-n", argv[1], argv[2], NULL);
        printf("Eroare la exec\n");
        exit(1);
    }

    if(pid_c = fork()) {
        printf("Eroare la fork\n");
        exit(1);
    }

    if(pid_c == 0) {
        close(pipe_bc[0]);
        close(pipe_ac[0]);
        close(results_pipe[1]);
        dup2(pipe_bc[1], 1);
        mod normal ar trebui facuta aici si verificarea starilor de return
        return 0;
    }

    wait(&); wait(&); // preluarea starii fiilor - in mod normal ar trebui facuta aici si verificarea starilor de return
}

```

335

How would you solve the following problem?

→ Program made of several processes.

→ A variable number of “producer” processes that generate, concurrently, data. Producers are of different, clearly defined types (categories).

→ A number of “consumer” processes, equal with the number of producer types, each consumer being therefore responsible for a single category of producers.

→ Data from producers must be sent only to the consumers responsible for the respective category.

336

8. Advanced concepts

337

Rights, users, identifiers

338

Identifiers associated with any process

- **real User ID, real Group ID**: the real owner of the process
- **effective User ID, effective Group ID**: the user/group on behalf on which the process runs (can be different from the real one)
- **supplementary Group IDs**: groups the user belongs to~
- **saved set user-ID, saved-set-group-ID**: copies of the IDs, saved by exec

339

When a program on a disk is launched

- the file on the disk has an owner user and an owner group
- usually, the effective UID/GID are equal to the real UID/GID of the current process (the process that launches the program)
- among the modes (rights, etc.) set for a file on a disk there are two special flags:
 - **set-user-ID (SETUID)**: if set for a program, the program will be executed by setting its effective UID to the UID of the **file owner** (instead of the UID of the launching process)
 - **set-group-ID (SETGID)**: if set for a program, the program will be executed by setting its effective GID to the GID of the **owner group of the file** (instead of the launching program's GID)

340

Example:

→ program that runs as root, although it was launched by a regular user (example from Linux):

```
> ls -l /usr/bin/passwd  
-rwsr-xr-x 1 root shadow 81792 oct 29 2011 /usr/bin/passwd
```

How to set SETUID and SETGID:

```
> chmod 4766 fichier > chmod u+s fichier SETUID
```

```
> chmod 2766 fichier > chmod g+s fichier SETGID
```

```
> chmod 6766 fichier > chmod ug+s fichier both
```

341

Changing the real/effective identity (UID, GID) of a process

It is needed when

- process needs more rights in order to do privileged operations
- process reduces its own privileges to prevent the access to certain resources

Adopting the **minimal privileges** strategy is recommended: a process should always retain only the minimal set of rights needed for accomplishing its job

Changing identity is governed by **strict rules**.

342

The setuid, setgid functions

```
#include <sys/types.h>  
#include <unistd.h>
```

```
int setuid(uid_t uid);  
int setgid(gid_t gid);
```

Rules:

1. If process has root privileges (superuser):
 $\text{real UID} \leftarrow \text{uid}, \text{effective UID} \leftarrow \text{uid}, \text{saved SETUID} \leftarrow \text{uid}$
2. If process is not root AND
(uid == **real UID** OR uid == **saved SETUID**):
 $\text{effective UID} \leftarrow \text{uid}$
3. Else, functions return -1 and set errno to EPERM

(likewise for groups, with setgid())

343

Remarks:

- only a process with root privileges can change real UID/GID
- when the root-privileged process uses setuid, setgid, all the three types of identifiers are changed, therefore, that process cannot regain the root privileges in the future.
 - useful when a privileged program (example: login) launches a user program, which is never allowed to run in a privileged state
 - in fact, root does not have any other reason for calling setuid, except to permanently reduce privileges
 - if a temporary privilege reduction is needed, other function can be used (seteuid)

The exec functions

- a) if for the executable file the SUID flag is set (activated):
exec sets effective UID to the file owner UID-ul
- b) always, exec saves effective UID in saved SUID;

Task b) is done after a) (if it is the case), therefore the saved id is the one got from the executable file

344

The seteuid, setegid functions

```
#include <sys/types.h>
#include <unistd.h>

int seteuid(uid_t euid);
int setegid(gid_t egid);
```

Sets only effective UID/GID, even if the process is privileged.

A process without root privileges can only set the attribute on the real UID/GID or saved SUID/SGID values already associated to the process.

345

Example*

The **at** command, which schedules the execution of programs in the future.

Security problems:

- **at** must run with the privileges specific to the user, as long as possible
- it must access system configuration files, therefore at a point it will need higher privileges
- the program that will be launched will have to run exclusively with the rights of the user that scheduled it

There are two components of this system:

- **the at command**, used for setting the schedule for programs
- **the atd service** (runs in background), which actually launches the programs at the scheduled times

*After: W.R.Stevens, S.A.Rago, *Advanced Programming in the UNIX Environment, Third Edition*; Addison Wesley, 2013

346

In some systems (e.g.: Linux 3.1) the file `/usr/bin/at` has SETUID set, and the owner is root. The following steps are done:

1. When starting **at**, because SETUID is set for root, the process attributes are:
 - real UID == the UID of the user who started **at**
 - effective UID = root
 - saved SUID = root
2. At start, **at** reduces its privileges to run as the user who started it. To do this, **at** calls `seteuid()`. Consequently:
 - real UID == the UID of the user who started **at**
 - effective UID = the UID of the user who started **at**
 - saved SUID = root
3. After a while, **at** needs higher privileges. Calls `seteuid()` to regain root privileges. It is allowed to, because root was saved in saved SUID (this case shows the utility of saved SUID):
 - real UID == the UID of the user who started **at**
 - effective UID = root
 - saved SUID = root

347

4. After it finished accessing the configuration files, returns to the privileges of the user that started it, calling `seteuid()` again:
 - real UID == the UID of the user who started **at**
 - effective UID = the UID of the user who started **at**
 - saved SUID = root
5. The **atd** service is a program that runs in the system with root privileges. When it prepares to start the program scheduled by the user, it must ensure the program runs strictly with the user rights.

To launch the program, **atd** calls `fork()`. Then, the child process calls `setuid(the_uid_of_the_user_who_scheduled_the_program)`. As the process has root privileges, all the three types of UIDs are set to the user's UID. The child process then runs the scheduled program.

- real UID == the UID of the user who started **at**
- effective UID = the UID of the user who started **at**
- saved SUID = the UID of the user who started **at**

348

File rights

349

The process rights for accessing files

When a process tries to create/modify/read/delete a file, the following checks are done, in order*:

- **if effective UID** of the process is root, access is allowed;
- **else, if effective UID == file owner ID**, access is allowed only **if** the permission bits corresponding to the operation are set (those belonging to the “user” category), **else** access is denied;
- **else, if effective GID** or one of the **supplementary GIDs** is equal to the **GID of the file**, access is allowed only **if** the group permission bits corresponding to the operation are set, **else** access is denied;
- **else, if** the bits corresponding to the “other” category are set, access is allowed, **else** access is denied;

* First matching rules applies, the rest are ignored.

350

The owner of a newly created file

When a file is created:

- **the owner (UID) of the file** is set as being the **effective UID** of the process that creates the file
- **the owner group (GID) of the file** is set depending on the UNIX version or on the options specified when mounting the file system.

It can be one of the:

- **effective GID** of the process that creates the file
- **the GID of the directory** where the file is created (Linux: makes it so if the directory has the SETGID flag set)

351

The fcntl function

352

The fcntl function

```
#include <unistd.h>
#include <fcntl.h>
```

```
int fcntl(int fd, int cmd, ... /* arg */ );
```

Applies various commands on the fd descriptor. The return value and the arguments depend on the specific commands. Returns -1 on error.

commands (the cmd argument):

- F_DUPFD (long) — duplicates fd and returns the new descriptor, which will be the lowest available (unopened) descriptor greater than arg (arg is considered of type long)
- F_DUPFD_CLOEXEC (long) — duplicates fd (as above) and sets the FD_CLOEXEC flag. (close on exec, i.e., the descriptor will be closed at exec)
- F_GETFD (void) — reads the descriptor flags; arg is ignored; for now, only the flag FD_CLOEXEC is defined
- F_SETFD (long) — sets the descriptor flags to the value given in arg
- F_GETFL (void) — gets the status flags of the file; the same ones used with open(): O_RDONLY, O_RDWR, etc.
- F_SETFL (long) sets descriptor status flags; only some of them can be modified, in Linux they are O_APPEND, O_ASYNC, O_DIRECT, O_NOATIME, O_NONBLOCK
- F_GETLK, F_SETLK, F_SETLKW: acquiring, testing, releasing of locks for portions of files (outside the scope of this course, details in bibliography)

353

Non-blocking I/O

354

Non-blocking input-output operations

Some I/O operations that usually imply blocking (examples: read, write) can be performed without blocking (ex: read tries to read, but if no data is available it doesn't wait anymore)

Two ways of doing it:

- With open(), the flag O_NONBLOCK is explicitly set
- With fcntl, by adding the O_NONBLOCK flag for an already open descriptor

355

Non-blocking input-output operations

Effect:

- operations will be performed without blocking, thus the respective calls will return immediately
- the operations (e.g.: read) can return error, but setting errno to EAGAIN — this means the operation did not succeed right away, but it is able to continue (for read(): no data was available at that time)

356

Example

```
int oldflags;
...
if ((oldflags = fcntl(fd, F_GETFL, 0)) < 0)
{ printf("Error at fcntl\n"); exit(1); }
if (fcntl(fd, F_SETFL, oldflags | O_NONBLOCK) < 0)
{ printf("Error at fcntl\n"); exit(1); }
...
while(1)
{
    do_stuff();

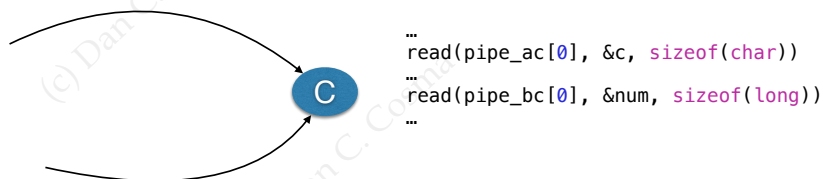
    size = read(fd, buff, expected_size);
    if(size < 0)
    {
        if(errno == EAGAIN)
            continue; /* or do other stuff */
        else
            { printf("Error at read\n"); exit(1); }
    }
    else
        if(size == 0) /* end of file */
            break;
}
```

357

I/O Multiplexing

358

Multiplexing input-output operations



Problems?

359

The *select()* function

Monitors sets of descriptors blocking itself until at least one of them becomes ready for the i/o operation

```
#include <sys/select.h>

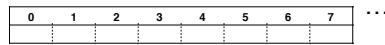
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

- readfds, writefds, exceptfds = the monitored sets of descriptors
 - readfds: for reading, writefds: for writing, exceptfds: exceptions*
 - can be NULL if the respective operation doesn't need to be monitored
- timeout: the maximum time to wait (NULL = unlimited)
- nfds: the maximum value (integer) of the monitored descriptors, plus one (must be calculated)

*exceptional conditions — for now the only exceptional condition is the existence of "out of band" data on a socket; outside the scope of this course

360

The set can be pictured as a bit array, one for each possible descriptor



Sets of descriptors are filled using specific macros. First `FD_ZERO` must be called, then the needed descriptors are set (added) with `FD_SET`

```
void FD_CLR(int fd, fd_set *set); - clears all descriptors
int FD_ISSET(int fd, fd_set *set); - verifies if a descriptor is in set
void FD_SET(int fd, fd_set *set); - sets (adds) a descriptor to set
void FD_ZERO(fd_set *set); - initializes the descriptor set
```

361

When `select()` returns because one or more descriptors have become ready, it will reinitialize the three sets, adding (setting) only the descriptors that have become ready

The return value of `select()`:

- 1: error or a signal occurred (in which case `errno` is set to `EINTR`)
- 0: `select` returned because the timeout elapsed
- > 0: success, returns the total number of ready descriptors, which will be available in the three sets

362

A descriptor is considered ready, depending on the set it is part of, as follows:

- *readfds*: a subsequent call to `read()` on that descriptor will not block (data is available)
- *writefds*: a `write()` to the descriptor will not block
- *exceptfds*: an exceptional condition occurred for that descriptor

Other remarks:

- for regular files, the descriptors are always considered ready
- at "end of file", the descriptor is considered ready

363

Example

```
int nfds = 1 + (pipe_ac[0]>pipebc[0] ? pipe_ac[0] : pipe_bc[0]);
...
int over=0, n;
while (1) {
    ...
    FD_ZERO(&readfds);
    FD_SET(pipe_ac[0], &readfds);
    FD_SET(pipe_bc[0], &readfds);

    if((nready = select(nfds, &readfds, NULL, NULL, NULL))<0)
        { printf("Eroare\n"); exit(1); }

    if(FD_ISSET(pipe_ac[0], &readfds))
    {
        n = read(pipe_ac[0], &c, sizeof(char));
        if(n==0) over++; /* end of data */
        ...
    }
    if(FD_ISSET(pipe_bc[0], &readfds))
    {
        n = read(pipe_bc[0], &num, sizeof(long));
        if(n==0) over++; /* end of data */
        ...
    }
    if(over == 2)
        break;
    ...
}
```

364

Groups of processes, jobs

365

Groups of processes

= a set of one or more processes, usually associated with a same *job*.

Any process can belong to a process group.

Any process group has an identifier (GID). The process with the identifier equal to the group is considered the “group leader”.

After `fork()`, the child process inherits the GID of its parent (belongs to the same group)

366

Finding out the group ID:

```
#include <unistd.h>

pid_t getpgid(pid_t pid);
    dacă pid==0, returns the group of the pid process

pid_t getpgrp(void);
    returns the group of the current process
```

367

Jobs

A *job* is a process group which can be controlled through the framework provided by the command interpreter. Not all shells support job control.

A process group is usually created when pipelining commands, launching background commands, etc. Example: the following lines create 2 process groups and, implicitly, 2 jobs:

```
$ ls -l | grep abc &
[1] 1165
$ ls &
[2] 1166
(after pressing ENTER)
[1]   Exit 1
[2]+ Done

ls -l | grep abc
ls
```

368

Job control

`$ program`

CTRL+Z: suspends the foreground job (SIGTSTP)

CTRL+C: ends (SIGINT) the foreground job

CTRL+\: ends (SIGQUIT) the foreground job

`$ bg %1` sends in background and resumes the suspended job no. 1

`$ bg` sends in background and resumes the last suspended job

`$ fg %1` brings in foreground job 1

369

Creating a group and adding processes to groups

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

Sets the group for a given process.

- if `pid == 0`, sets the group for the current process
- if `pid == pgid`, the `pid` process becomes the leader of a new group
- if `pgid == 0`, the group of the `pid` process will be made equal with `pid` (`pid` process becomes the leader of a new group)
- else, `pgid` must be an existing group, in the same session with the current one; the process is moved to that group

A process can call `setpgid` only for itself and for any of its children that have not yet called `exec`

370

A process can send a signal to an entire group of processes.

```
#include <sys/types.h>
#include <signal.h>
```

```
int kill(pid_t pid, int sig);
```

- `pid > 0`: Sends signal `sig` to process `pid`
- `pid = 0`: Sends signal `sig` to the current group of processes
- `pid < 0`: Sends signal `sig` to the `-pid` group of processes

371

Sessions

372

Session

= a set of one or more process groups

A session has at most one *control terminal*, which is a device capable of displaying data and providing input (keyboard)

A control terminal can be associated to a session when the “session leader” process is created, only if that terminal was not already associated. The way a terminal is requested differs from one UNIX variant to another. At login, a terminal is automatically associated.

373

If a session has a control terminal:

- Only one group of processes is the **foreground group**. Only this group will be able to read from the terminal (keyboard). All processes in this group will be affected by CTRL-C, CTRL-\ (will receive the corresponding signals)
- All other process groups are **background groups**. A read() from the terminal made by a process in a background group will suspend the group (the group will receive SIGTSTP)

374

Creating a new session

```
#include <unistd.h>

pid_t setsid(void);
```

If the caller process is NOT a group leader, a new session is created:

- the session will NOT have a control terminal
- the process becomes “session leader”
- the process becomes the leader of a new process group, the first in the session
- if the process had a control terminal before the call, its connexion with it is lost

If the caller process is already a group leader, the call returns error (-1).

375