

# Общи приказки

## Операционни системи, ФМИ, 2022/2023

## Книга

- Tanenbaum, Andrew S. Modern Operating Systems. Pearson, 2014  
(4th edition)

## Операционна система: мотивация

- Преносимост (portability) на програмите върху различен хардуер
- Интерфейс, предоставящ възможност за изпълнение на отделни програми
- Многозадачност
  - едновременно за потребителя изпълнение на отделни програми
- Многопотребителност
  - използване на една и съща система от няколко потребителя; изолация
- Автоматизация

## Операционна система: абстракции

- Процеси
- Комуникационни канали
- Споделена памет (краткотрайна и дълготрайна)
- Пространство от имена, индексиращо обектите, за които ОС предоставя абстракции

## Структура на една система (в общия случай)

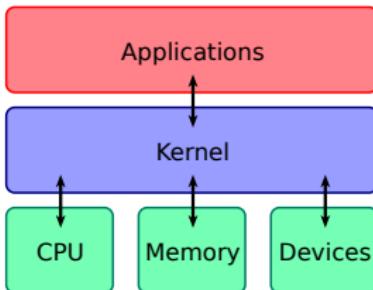


Figure 1: Приложения, ядро, устройства

- Ядро
- Библиотеки, позволяващи лесен достъп до абстракциите, предоставяни от ядрото
- Конфигурационни данни
- Основни инструменти
- Допълнителни инструменти / приложения

## Хляб и масло

- Език за програмиране
- Компилатор
- Програма / машинен код
- Изпълним файл
- Изпълнение на програма

## Процеси

- Основна абстракция, предоставяна от една ОС
- Приемаме, че един процес абстрагира последователно изпълнение на конкретна програма
- Всеки процес има контекст
  - памет, която може да достъпва
  - отворени файлове
  - права за достъп
  - ...
- ОС позволява процесите да **комуникират** помежду си и с външния свят

## Нишки

- Някои ОС позволяват няколко паралелни нишки на изпълнение в рамките на един процес
  - нишките споделят общ контекст на процеса
  - могат да се изпълняват едновременно, все едно са отделни процеси
- Няма да говорим за тях засега - приемаме, че всеки процес съответства на изпълнението на една програма на един процесор

## Илюзията за едновременност

- Еднозадачни операционни системи
  - опашки от задачи
- Многозадачни операционни системи
  - в общия случай процесорите са по-малко от процесите
  - времеделене и смяна на контекста
  - cooperative and preemptive scheduling

## Драйвери

- Процеси (microkernel) или части от ядрото (monolithic kernel)
- ОС предоставя начин обикновените процеси да комуникират с драйверите
- Драйверите комуникират с (външни) устройства

## Какво трябва да сте запомнили за “операционна система”

- Предоставя абстракции
  - процеси: работещи програми с контекст
  - комуникационни канали между процеси
  - споделена памет / файлове
  - пространство от имена, рефериращи обектите
- Абстрагира хардуера, което позволява преносимост (portability)
- Може да поддържа конкурентно изпълнение на процеси чрез времеделене

## Какво следва?

- Досега обяснихме по много абстрактен начин какво очакваме от една операционна система
- Сега ще преминем към по-конкретни понятия, свързани с операционните системи, базирани на UNIX

# UNIX, GNU, Linux

## Операционни системи, ФМИ, 2022/2023

## UNIX, история

- Многозадачна и многопотребителна операционна система
- Разработена през 1969 в Bell Labs
  - Brian Kernighan
  - Dennis Ritchie
  - Ken Thompson
- По-късно пренаписана от Асемблер на C

# Основни принципи на UNIX

- Глобално пространство от обекти, наречени “файлове”
  - всеки обект в операционната система се манипулира като файл
  - дървото от файлове което ще наричаме “абстрактна файлова система”
- Много на брой малки програми
  - всяка прави едно нещо добре
  - работят върху текст и се конфигурират с текст
  - UNIX Philosophy
- Комбиниране на (малки) програми за решаване на сложни задачи
  - паралелно-работещи процеси, които комуникират чрез текстови потоци

# Richard Stallman



Figure 1: Richard Stallman

# Richard Stallman, GNU, FSF, GPL

- GNU (GNU's Not Unix), 1983
- Free Software Foundation, 1985
- GNU General Public License (GPL)
  - Задължава софтуера да идва заедно с кода си до потребителите
  - Заразен (софтуер, надграждащ GPL софтуер, също трябва да е GPL)
- Free as in Freedom<sup>1</sup> <sup>2</sup>
  - <http://oreilly.com/openbook/freedom/>

---

<sup>1</sup>Williams, Sam. Free as in Freedom: Richard Stallman's Crusade for Free Software. O'Reilly, 2002

<sup>2</sup>Williams, Sam. Free as in freedom (2.0): Richard Stallman and the free software revolution. Boston: Free Software Foundation, 2010

## GPL freedoms

- ① The freedom to run the program, for any purpose.
- ② The freedom to study how the program works, and change it so it does your computing as you wish. Access to the source code is a precondition for this.
- ③ The freedom to redistribute copies so you can help your neighbor.
- ④ The freedom to distribute copies of your modified versions to others. By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.

## GNU tools

- gcc, gdb, emacs - open source toolchain, състоящ се от компилатор, дебъгер и текстов редактор, които позволяват да създаваме софтуер
- програми за работа с файлове и с операционната система
- много програми за работа с текст и за решаване на всекидневни задачи

- Linux kernel
  - създаден от Linus Torvalds
  - вдъхновен от Minix<sup>3</sup>
  - до голяма степен съвместим с интерфейсите на ниско ниво на UNIX
  - лицензиран под GPL
- GNU/Linux - комбинация от *Linux kernel* и *GNU utilities*<sup>4</sup>
  - тази комбинация ще я наричаме просто “Linux”<sup>5</sup>
- Други комбинации, които са не-толкова разпространени
  - GNU/Hurd
  - GNU/kFreeBSD

---

<sup>3</sup>образователна операционна система, създадена от Andrew Tanenbaum (да, авторът на учебника)

<sup>4</sup>съкупността от инструменти и библиотеки (utilities) извън ядрото още се нарича “userland”

<sup>5</sup>за голямо разочарование на Stallman

# Linux<sup>7</sup> дистрибуции

- Linux идва като комплект, който наричаме “дистрибуция”
- Всяка дистрибуция се отличава със свой начин да *пакетира* софтуер
- Всяка библиотека/инструмент/приложение обикновено<sup>6</sup> идва в *пакет*, който може да се инсталира от потребителя
- Различни идеологии при различните дистрибуции
- Но в крайна сметка разликите са в пакетната система
  - всяка дистрибуция има леко различен набор от пакети
  - някои дистрибуции са “стабилни” и обновяват основната версия на пакетите си по-рядко
  - в различните дистрибуции софтуерът може да е пакетиран така, че да се конфигурира по различен начин

---

<sup>6</sup> примери за дистрибуции с нестандартна пакетна система: buildroot, yocto, nixos, guix

<sup>7</sup> actually, it's called GNU/Linux, not Linux 😊

# Популярни Linux дистрибуции

- Основно за потребителски компютър
  - Debian
  - Ubuntu
  - Linux Mint
  - Fedora
  - Manjaro
- Основно за сървъри
  - Debian
  - Red Hat Enterprise Linux (RHEL)
  - CentOS (Community ENTerprise Operating System)
- За телефони
  - PostMarketOS
  - Ubuntu Phone
- По-затворени
  - Android
  - ChromeOS
- За всичко (много конфигурираме)
  - Arch Linux
  - Gentoo
  - NixOS
- Non-linux
  - FreeBSD, OpenBSD, NetBSD
  - MacOS
  - SerenityOS

## GNU/Linux Distribution Timeline

- Картинка, качена в Wikimedia
- Картинката е направена, използвайки  
<https://github.com/FabioLolix/LinuxTimeline>

# Терминали, сесии и как да оцелеем в Linux

Операционни системи, ФМИ, 2022/2023

## Терминал (абстрактен)

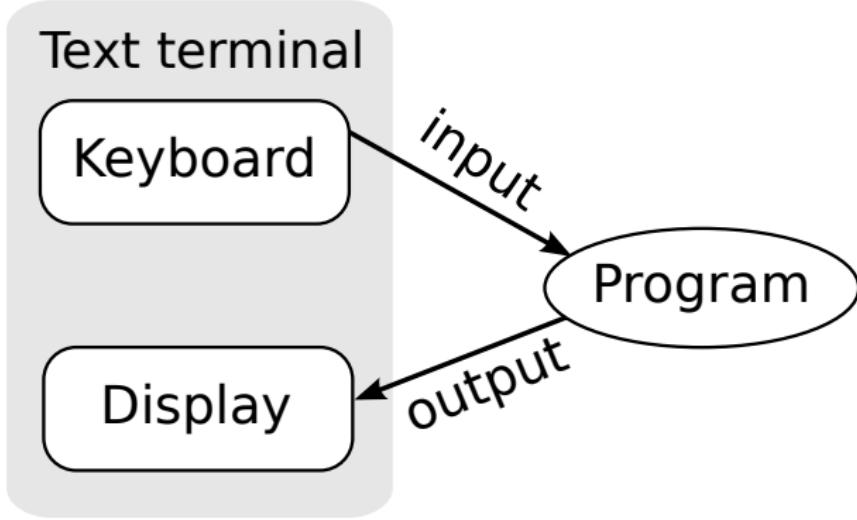


Figure 1: абстракция за терминал

- Терминал - *нешто*, позволяващо (текстова) комуникация между потребител и програма

## Терминал (физически)



Figure 2: Терминал “VT100”, произведен през 1978

- Физическо устройство
- Свързва се към компютър с кабел
  - комуникацията по най-често става по сериен протокол (RS232)
- Вход (клавиатура) и изход (екран)
- ОС има драйвер, който управлява съответния модел терминал

## Терминал (виртуален)



Figure 3: Графичен виртуален терминал

## Терминал (виртуален)

- Програма, която симулира физически терминал
- Позволява текстов вход/изход през не-текстови устройства (монитори)
- TTY - виртуален терминал, емулиран от kernel-a (CTRL-ALT-Fn)
- PTY Графични терминали (alacritty, gnome-terminal, urxvt, xterm)
- PTY Отдалечени терминали
  - telnet - отдалечен терминал през мрежа
  - OpenSSH - криптиран отдалечен терминал през мрежа
  - PuTTY - за Windows използва SSH протокола
  - JuiceSSH - за Android, използва SSH протокола
- PTY терминал-в-терминал
  - screen
  - tmux

## Графичен и текстов интерфейс

- На този курс говорим за текстови интерфейси, защото те позволяват прецизен контрол на ОС и лесна автоматизация
- Освен тях са доста разпространени и графични интерфейси
- Има и други видове интерфейс (напр. гласов, пригоден за незрящи)
- Всеки от тези видове интерфейси има своите входни и изходни устройства

## Потребители и сесии

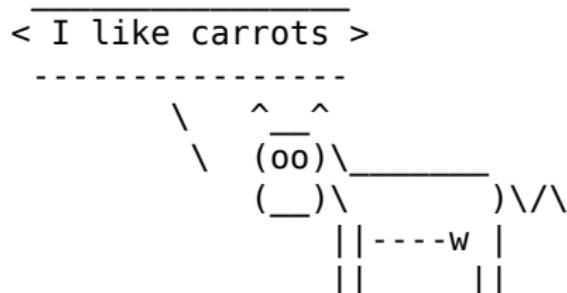
- Казахме, че Linux (и генерално UNIX-базираните системи) са **многопотребителни**
- При инициализиране на терминал, се изпълнява програма, наречена *Login*, която установява кой е текущият потребител
  - Обикновено “начинът за установяване” е питане за име и парола
  - След login, се изпълнява програма, която наричаме *Shell*

## Shell

- Програма, която предоставя интерактивен команден интерфейс през терминала
- Shell-ът позволява на потребителя лесно да изпълнява други програми и да командва операционната система
- Съществуват различни shell програми и използването на конкретен shell е въпрос на избор

## Интерпретиране на команди

```
human@astero:~$ cowsay I like carrots
```



```
human@astero:~$
```

## Интерпретиране на команди

```
human@astero:~$ cowsay I like carrots
```

- Prompt - `human@astero:~$`
  - Текст, който shell-ът отпечатва, когато очаква да въведем команда
  - При повечето shell-ове, prompt-ът за обикновен потребител завършва с “\$”.
  - Дава никаква допълнителна информация (текущ потребител, име на машината ...)
- Име на команда - `cowsay`
- Аргументи (параметри) на командата - разделени с интервали (whitespace)

## Интерпретиране на команди

- Когато някъде видите символа “\$” пред някакъв текст, вероятно се има предвид, че това е shell команда

```
$ cowsay Cookies are tastier than carrots
```

## Ехо и крави

- До тук видяхме командата `$ cowsay`, която рисува крава, казваща аргументите си
- Командата `$ echo <аргументи>` изписва аргументите си, без да рисува крава

## Аргументи

- Произволен брой интервали<sup>1</sup> отделят аргументите на команда

```
$ echo I have      many      spaces  
I have many spaces
```

- Ако искаме да подадем текст, съдържащ интервали, като един аргумент, използваме кавички (quoting):

```
$ echo 'I have      many      spaces'  
I have      many      spaces
```

- Някои символи, например скоби, имат специално значение за shell-а, и ако искаме да са буквална част от аргумент, също трябва да използваме кавички:

```
$ echo 'I like cookies (chocolate ones)'  
I like cookies (chocolate ones)
```

- Двойните кавички ("") работят по подобен начин като единичните (''), но за тях ще говорим по-късно.

---

<sup>1</sup>и произволен друг whitespace

## Опции

- Повечето команди по конвенция приемат “опции”, които започват с тире. Опциите обикновено променят поведението на командата.

```
human@astero:~$ cowsay -f vader 'I am your father'
```

< I am your father >

## Cowth Vader

## Много shell-ове

Съществуват най-различни възможни shell програми

- Thompson Shell (`sh`) – Ken Thompson, 1971, AT&T
- Bourne Shell (`sh`) – Stephen Bourne, 1977, AT&T
- C Shell (`csh`) – Bill Joy, 1978, BSD
- Korn Shell (`ksh`) – David Korn, 1983, AT&T
- Enhanced C Shell (`tcsh`) – Ken Greer, 1975 - 1983, CMU
- Bourne Again Shell (`bash`) – Brian Fox, 1989, GNU
- Z Shell (`zsh`) – Paul Falstad, 1990, Princeton
- Debian Almquist shell (`dash`) – port of NetBSD `ash` to Linux by Herbert Xu 1997, renamed `dash` 2002
- fish (`fish`) – a smart and user-friendly command line shell, 2005

## Много shell-ове

- В този курс ползваме bash

## Навигация на командния ред

- Стрелките нагоре/надолу навигират историята от команди
- `Ctrl-R` търси в историята
- Клавишът `Tab` довършва частично-написана команда/аргумент (autocomplete)
- `Ctrl-C` прекратява текущата команда

## Изчистване на терминала

- `$ clear` - изчиства терминала, като запазва scrollback-а
- `$ reset` - изчиства терминала, scrollback-а и всички настройки на терминала

## Помощ (опции на команди)

- Много команди имат аргументи `--help` или `-h`, които ги карат да отпечатват помошно съобщение

## Помощ (man)

- \$ man <тема> дава информация по дадена тема - man-страница (manpage)
- Навигира се със стрелки, излиза се с клавиша Q
- Пробвайте \$ man cowsay, \$ man man
- Различни секции (shell командите са в секция 1)
  - \$ man 1 printf, \$ man 3 printf
  - когато искаме да реферираме специфична ман страница, ще пишем така: printf(1), printf(3), cowsay(1)
- \$ apropos <низ> - търси в кратките описания на всички man страници
- \$ whatis <низ> - връща всички секции, в които се среща дадена man страница, и кратки описания на съответните теми

# Потребители

## Операционни системи, ФМИ, 2022/2023

## Потребители

- Всеки потребител си има User ID (UID) и основно Group ID (GID)
- Освен основната си група, потребителите могат да имат и произволен брой други групи
- Текущият потребител си има *активна* група, която може да е някоя от групите, на които е член
  - обикновено активната група съвпада с основната
  - `$ newgrp <група>` - смяна на активна група в рамките на текущата сесия
- `$ id` дава информация за текущия потребител
  - `$ id -u` - само UID-то
- `$ whoami` отпечатва името на текущия потребител
- `$ who` отпечатва всички логнати потребители

## Конфигурация на потребители

- Файлът `/etc/passwd` съдържа информация за всички потребители
- Файлът `/etc/group` съдържа информация за всички групи
- `passwd(5)` и `group(5)`, за да разберете повече<sup>1</sup>

---

<sup>1</sup> подсещам, че записът `passwd(5)` ви подканва да изпълните командата `$ man 1 passwd`

## Супер потребител

- Потребителят с UID 0 (който винаги се казва `root`) се нарича *супер потребител (superuser)*
- Има право да прави и достъпва всичко

## Смяна на потребител

- \$ su <потребител> - изпълнява *login shell* от името на подадения потребител
- \$ sudo <команда> - изпълнява подадената команда от името на root
  - \$ sudo -u <потребител> <команда> - изпълнява подадената команда от името на подадения потребител
- Тези команди работят само ако вашият потребител има право да ги използва
  - потребителите, които използвате на упражнения по ОС, нямат такова право - не се опитвайте да използвате тези команди

# Абстрактна файлова система

## Операционни системи, ФМИ, 2022/2023

## Абстрактна файлова система

- В UNIX-подобните операционни системи съществува **едно** пространство от имена
- Ще го наричаме “абстрактна файлова система” или просто “файловата система”
- Възлите в това дърво наричаме “файлове”
  - вътрешните възли наричаме “директории” (които също са “файлове”)
- “файл” е произволен споделен (между процесите) обект
  - “обикновен файл” е дълготраен обект, съдържащ масив от данни
  - другите обекти също са “файлове”

## Пътища (абсолютни)

- \$ pwd - отпечатва текущата директория
- \$ cd <path> - премества текущата директория в дадената
- \$ cd /home/students/student - абсолютните пътища започват с “/” и еднозначно определят някой обект
- \$ ls - отпечатва списък със всички обекти в текущата директория
  - може да приеме като аргумент име на директория
  - предназначена за четене от **хора**
  - \$ ls -a - показва и файловете, чиито имена започват с “.”  
*(скрити файлове)*
  - \$ ls -l - показва повече информация

## Пътища (относителни)

- Път, който **не** започва с “/”, е *относителен* и интерпретацията му зависи от текущата директория
- <абсолютен път> = <текуща директория>/<относителен път>
- При текуща директория /home/baba, абсолютният път foo/bar/baz води до /home/baba/foo/bar/baz

## Команди за работа с пътища

- `$ realpath <път>` - превръща произволен път в абсолютен път
- `$ basename <път>` - връща базовото име на файл (символите след последната наклонена черта)
- `$ dirname <път>` - връща пътя на директорията, в която е файл (символите преди последната наклонена черта)

## Home директория

- (Почти) всеки потребител има директория, която наричаме *home* директория
- Всеки потребител има пълни права върху файловете в home директорията си
- Съкратен запис за home директория: ~ и ~<потребител>
  - ~ води до home директорията на текущия потребител, например /home/human
  - ~ivan води до home директорията на потребителя ivan
- Home директорията на някой потребител не винаги се намира в /home

## Навигация нагоре

- Във всяка директория винаги има две служебни директории:
  - . - води към същата директория
  - .. - води към по-горната директория
- ../../ води 2 директории нагоре от текущата
- /home/students/student/.. води в /home/students
- /home/baba/foo/bar/../../../../../qux/.../dyado/. води в /home/dyado

# Filesystem Hierarchy Standard (FHS)

- <https://refspecs.linuxfoundation.org/fhs.shtml>
- Стандарт, който назава кои от системните файлове в коя директория да седят

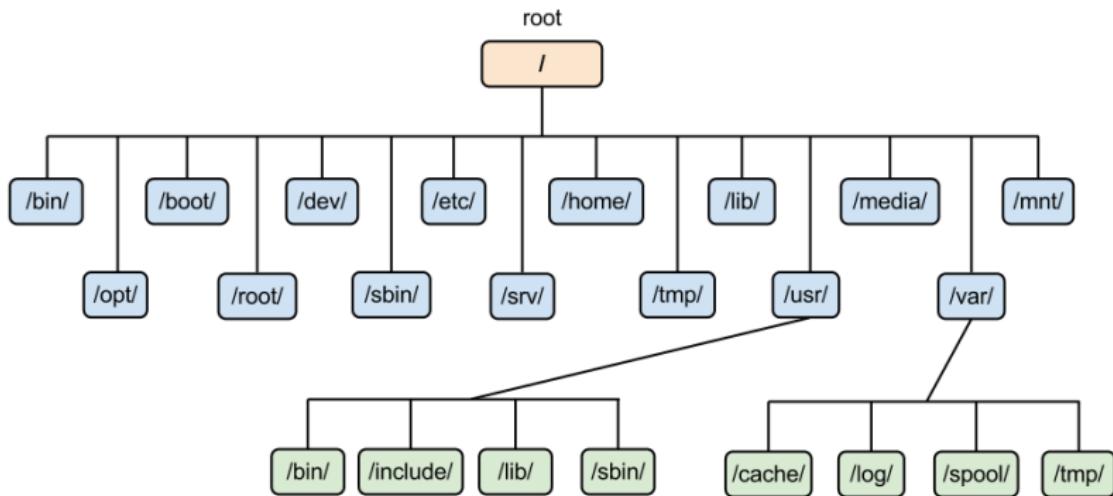


Figure 1: Директорийна структура спрямо FHS

## Още няколко команди

- \$ touch <път> - създава празен файл
- \$ mkdir <път> - създава празна директория
  - \$ mkdir -p - създава и всички директории по пътя
- \$ cp <път-от-къде> <път-къде> - копира файл
  - \$ cp -r - копира директория
- \$ mv <път-от-къде> <път-къде> - преименува/премества файл
- \$ rm <път> - изтрива файл
  - '\$ rmdir - изтрива празна директория
  - \$ rm -r <път> - изтрива директория и всичко в нея

Дати и часове  
Операционни системи, ФМИ, 2022/2023

## Работа с дати и часове

- Един от най-сложните проблеми на човечеството
- Използваме библиотеки и команди, които някой друг е написал (никога не си пишем сами)

## date

Командата `date` без аргументи може да се използва за да видим текущото време<sup>1</sup>

```
$ date  
Sun 26 Feb 18:35:56 CET 2023
```

---

<sup>1</sup>в локалната времева зона

## date

date може да изписва времето в произволен формат, който задаваме с опцията +<форматен низ>

```
$ date +'%Y-%m-%d %H:%M:%S'  
2023-02-26 18:53:44
```

```
$ date +'%H:%M'  
18:53
```

```
$ date +'%H:%M:%S.%N'  
18:53:44.647864108
```

## date

date може да превежда време от един формат в друг (-d)

```
$ date -d '26 Feb 1998 13:59'  
Thu 26 Feb 13:59:00 CET 1998
```

```
$ date -d 'Yesterday'  
Sat 25 Feb 19:01:03 CET 2023
```

```
$ date -d 'Tomorrow 15:00' +'%Y-%m-%d %H:%M'  
2023-02-27 15:00
```

```
$ date -d '15:00 EET'  
Sun 26 Feb 14:00:00 CET 2023
```

## UNIX timestamp

- *UNIX timestamp* е представяне на момент от времето като брой секунди, изминали от 1970-01-01 00:00 UTC.
- Това представяне е удобно за подреждане на моменти от време.
- Linux ядрото използва UNIX timestamp-ове за да представя повечето времена

## Работа с UNIX timestamps

date може да покаже UNIX timestamp с форматирация низ ‘%s’:

```
$ date +'%s'  
1677434831
```

Ако искаме повече точност, може да използваме и %N  
(наносекунди):

```
$ date +'%s.%N'  
1677434888.017353335
```

date може и да чете UNIX timestamp така:

```
$ date -d '@1677434888'  
Sun 26 Feb 19:08:08 CET 2023
```

```
$ date -d '@0'  
Thu 1 Jan 01:00:00 CET 1970
```

Птички и пчелички  
Операционни системи, ФМИ, 2022/2023

## Стоп

Преди да продължим с тази лекция, всеки трябва да е напълно запознат със следните понятия:

- Битове и байтове
- Запис на числа с различни бази (бройни системи)
  - Двоичен запис
  - Осмичен запис
  - Шестнадесетичен запис
- Младша (least-significant) и старша (most-significant) цифра
- Побитови логически операции (AND, OR, XOR)
- Представяне на число чрез двоични данни
  - Little-endian и Big-endian стойности

# Работа с файлове в UNIX

## Операционни системи, ФМИ, 2022/2023

## Типове файлове

За тези вече говорихме

- [ - ]: Обикновен файл
- [d]: Директория

За тези ще говорим след две минути

- [c]: Character device
- [b]: Block device

За тези ще говорим скоро

- [l]: Символна връзка (symlink)
- [p]: Pipe

До тези може и да не стигнем

- [s]: Unix domain socket

## Специални файлове

- Обикновените файлове, директориите и някои други типове файлове заемат физическо място върху някаква памет (най-често дълготрайна)
- Други типове файлове, напр. символните и блоковите устройства, просто реферират някаква структура от данни в ядрото, която абстрактира съответния обект

## Устройства

- Директорията `/dev` съдържа обекти, рефериращи физически устройства
- Character devices (напр. `/dev/tty*`) - входно-изходни устройства, работещи със символи
- Block devices (напр. `/dev/sd*`) - устройства за съхранение, представлящи масив от байтове
  - разделени на блокове - малка единица за четене/писане

## Сглобяване на файловата система

- Казахме, че всички обекти, които наричаме *файлове*, са организирани в **едно** дърво
- Същевременно искаме да можем да организираме данните си във файлове, върху повече от едно дискови устройства
- Как съчетаваме тези две неща?

## Сглобяване на файловата система

NAME	TYPE	MOUNTPOINT
/dev/sda	mbr	
└/dev/sdal	dm_crypt	
└/dev/mapper/sdal_crypt	lvm	
├/dev/mapper/stg-swap	swap	[SWAP]
├/dev/mapper/stg-root	ext4	/
└/dev/mapper/stg-home	ext4	/home
/dev/sdb	gpt	
└/dev/sdb1	btrfs	/var
└/dev/sdb2	btrfs	/srv/games
	devtmpfs	/dev
	tmpfs	/tmp

- Върху блоковите устройства можем да имаме структури от данни, които улесняват разделянето и владеенето им
- Пак получаваме блокови устройства, образувайки дърво
- Върху листата на това дърво разполагаме конкретни *файлови системи*, които монтираме на клоните на дървото на абстрактната файла система
- За повече информация - `mount(8)`

## Файлови системи и Файлови системи

Има няколко понятия, които наричаме “файлова система”:

- Абстрактната файлова система, която е едно дърво, което обединява всички конкретни и виртуални файлови системи
- Конкретна файлова система, разположена на дисково устройство, която е структура от данни
  - може да бъде монтирана върху директория от абстрактната файлова система
  - напр. ext4, exfat, ntfs, btrfs
- Конкретна виртуална файлова система, която съдържа специални обекти, управлявани от операционната система
  - може да бъде монтирана върху директория от абстрактната файлова система
  - напр. /dev

## Номер на инод

- Всеки файл в рамките на една (конкретна) файлова система е номериран с число, което наричаме *номер на инод* (*inode number*)
- Може да видите тези номера с командалата `$ ls -l -i`
- Името е такова по исторически причини: някога единствените файлови системи, които са се използвали под UNIX/Linux са използвали структура от данни, наречена инод<sup>1</sup>

---

<sup>1</sup>такива файлови системи са ext2, ext3, ext4; по-нататък в курса ще говорим за имплементацията им

## Директории

- За операционната система всяка директория е *файл*, който съдържа таблица с имена на файлове, намиращи се в нея, и съответните им номера на иноди

## Атрибути на файловете

Всеки файл, файловата система пази списък от атрибути:

- Тип
- Времена
  - време на последна модификация (`mtime`)
  - време на последен достъп (`atime`)
  - време на последна смяна на атрибути (`ctime`)
- Собственост и права за достъп (за тях ще говорим след малко)
- Размер
- Брой линкове (колко различни имена има този файл)

## Блокове

- Файловите системи (обикновено) делят дисковото пространство на *блокове*
  - Блоковете на файловата система е хубаво да съвпадат с блоковете на блоковото устройство, върху което е разположена тя
- Всички блокове са с фиксиран еднакъв размер - например 4kb

## Атрибути на файловете

- Командите `$ stat <файл>` и `$ ls -l` показват информация за атрибутите на файловете

```
$ stat /tmp/foo
  File: /tmp/foo
  Size: 37704      Blocks: 80          IO Block: 4096   regular file
Device: 0,43      Inode: 28           Links: 1
Access: (0644/-rw-r--r--)
    Uid: ( 1000/    human)  Gid: ( 1000/    human)
Access: 2023-02-23 09:54:04.773861779 +0200
Modify: 2023-02-23 15:21:13.785684395 +0200
Change: 2023-02-23 15:21:13.785684395 +0200
 Birth: 2023-02-23 09:54:04.773861779 +0200
```

## Още малко за stat

- Опцията `-c` (или `--format`) може да се използва за да изолираме специфичен атрибут на файла
- `$ stat -c '%s' <файл>` - точен размер на файла в байтове
- `$ stat -c '%U'` - име на потребителя-собственик на файла
- `$ stat -c '%Y'` - момент на последна модификация на файла (UNIX timestamp)
- За синтаксиса на форматирация низ и за други атрибути, вижте `stat(1)`

## Собственост

- Всеки файл има потребител-собственик и група-собственик
- Можете да ги видите в изходите на `$ stat` и `$ ls -l`
- Променят се с командата `$ chown`:
  - `$ chown <потребител>:<група> <файл>` - променя и двете
  - `$ chown <потребител> <файл>` - променя само потребителя
  - `$ chown :<група> <файл>` - променя само групата<sup>2</sup>

---

<sup>2</sup>има и команда `$ chgrp`, която също променя само групата

## Права за достъп

- Всеки файл има множество от права за достъп (UNIX permissions set)
- 9 възможни елемента на множеството:  
 $\{ \text{User, Group, Others} \} \times \{ \text{Read, Write, Execute} \}$

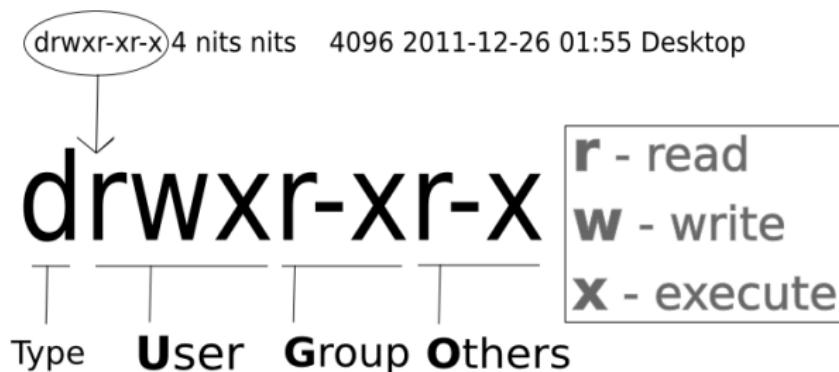


Figure 1: Права за достъп във формата на `ls -l`

## Права за достъп

символ право	обект	описание
r	read	file      може да отвори файл за четене
r	read	dir      може да види обектите в директория <sup>3</sup>
w	write	file      може да променя файл
w	write	dir      може да добавя/манипулира/трие обектите в директория
x	execute	file      може да изпълни файл <sup>4</sup>
x	execute	dir      може да се позиционира в директория и да адресира обектите в нея

<sup>3</sup>директорията е просто файл - за да видим какво има вътре, трябва да прочетем съдържанието ѝ

<sup>4</sup>файлът трябва да е програма под никаква форма - по-нататък ще говорим за това

## Промяна на права

- Можете да промените правата за достъп на файл с командата  
  \$ chmod <права> <файл>
- Правата може да са в няколко различни формата:
  - \$ chmod u=rwx,g=rx,o=- конкретни права
  - \$ chmod a=rw - същото като \$ chmod u=rw,g=rw,o=rw
  - \$ chmod g+x - добавя право на изпълнение за групата-собственик
  - \$ chmod u-w - премахва право на запис за потребителя-собственик
  - \$ chmod 755 - конкретни права в осмичен вид - ще обясним след малко
- \$ chmod -R - работи рекурсивно за дадената директория и нейните поддиректории

## Права за достъп (осмичен вид)

- Удобно е да представим 9-те бита за права като 3 групи по 3 бита
- Всяка група от 3 бита можем да я запишем като цифра
- Това е същото като да мислим за едно 9-битово число, записано в осмична бройна система

## Права за достъп (осмичен вид)

- Основни елементи:
  - r - 100b - 4 – Read
  - w - 010b - 2 – Write
  - x - 001b - 1 – Execute
- Комбинираме ги с побитово “или”:
  - $r \mid w = 6$
  - $r \mid x = 5$
- Примерни множества от права:
  - 755: u=rwx, g=rx, o=rx
  - 750: u=rwx, g=rx, o=
  - 644: u=rw, g=r, o=r
  - 600: u=rw

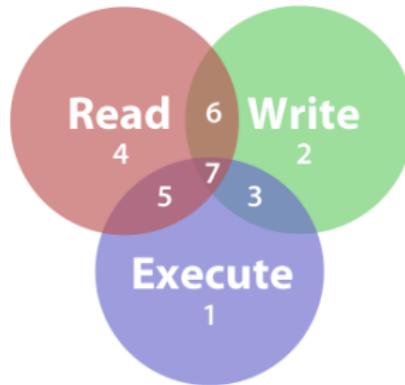


Figure 2: Представяне на правата като 3-битова цифра

## Специални права

- Въщност, освен 9-те елемента на множеството от права, които разглеждахме дотук, има още 3
- You were all deceived, for another permissions set had been forged
- Цялата информация за правата се кодира в 12 бита

## Специални права (1/3)

SUID (set UID upon execution) \* за изпълними файлове кара процесът, създаден при изпълнение на файла да е от името на потребителя-собственик на файла вместо на текущия потребител

- за директории не прави нищо

## Специални права (2/3)

SGID (set GID upon execution) \* за изпълнени файлове кара процесът, създаден при изпълнение на файла да е от името на групата-собственик на файла вместо на активната група

- за директории кара новите файлове, създандени в директорията да имат група-собственик като нея, вместо като активната група

## Специални права (3/3)

Sticky bit \* за файлове не прави нищо

- за директории кара операциите върху имената в директорията (изтриване, преименуване...) да зависят от правата на съответния файл вместо тези на директорията

## Creation mask

- Правата по подразбиране са:
  - 777 (a=rwx) за директории
  - 666 (a=rw) за файлове<sup>5</sup>
- С командата `$ umask <маска>` можем да зададем множество от права, които се **премахват** от тези по подразбиране при създаване на нов обект
- С командата `$ umask` без аргументи можем да видим текущата маска
- `final_perms := default_perms & ~umask`
- Обикновено шелът е настроен да задава автоматично `$ umask 022`

---

<sup>5</sup>TODO: paste youtube link to Iron Maiden - The Number of the Beast

## Твърди връзки (hardlinks)

- Обикновените файлове могат да имат повече от едно име
- Нищо не пречи на две имена в някакви директории да бъдат асоциирани с един и същи номер на инод
- В този контекст имената ги наричаме “твърди връзки”
- Само в рамките на една файлова система

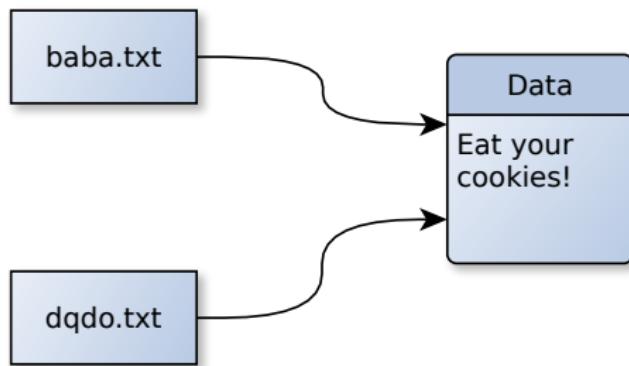


Figure 3: Твърда връзка

## Твърди връзки (hardlinks)

- Създават се с командата `$ ln <нов път> <текущ път>`
- `ls -l`, `stat` и `find` могат да ни кажат колко линка (имена) има някой файл (тази информация се пази при атрибутите на файла)

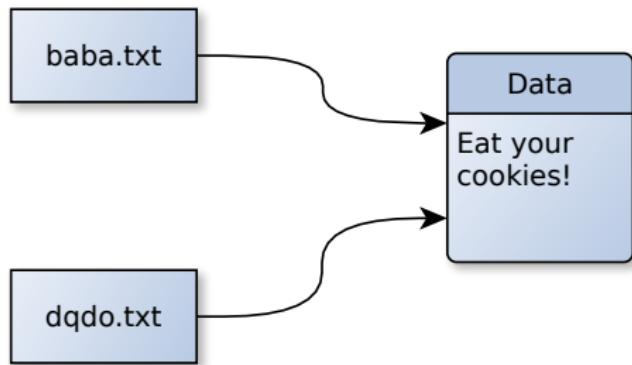
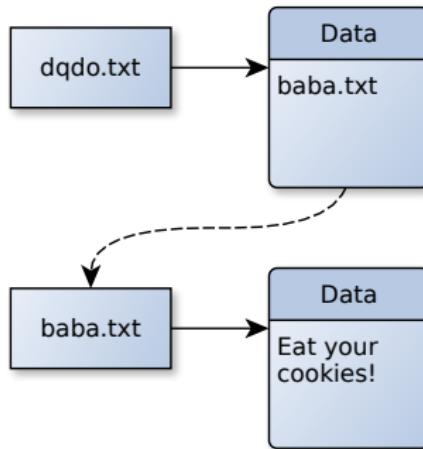


Figure 4: Твърда връзка

## Символни връзки (symlinks)

- Символната връзка е специален файл, съдържащ път до друг файл
- Автоматично се *dereferencirat* от операционната система
- Нямат някои ограничения, които твърдите връзки имат:
  - могат да сочат към директории
  - могат да сочат към файлове на други файлови системи
  - могат да сочат към несъществуващи файлове (broken symlink)<sup>6</sup>
- Създават се с команда `$ ln -s <нов път> <текущ път>`



## Относителни символни връзки

- Може да имаме символна връзка с относителен път
- `$ ln -s ../pesho/bar /home/gosho/baz` - символна връзка, която се назава `/home/gosho/baz` и сочи към `/home/pesho/bar`
- Пътят е относителен спрямо местоположението на самия symlink

## Команди за работа с пътища (поведение върху символни връзки)

- `$ realpath <път>` - превръща произволен път в абсолютен път
  - дереферира всякакви символни връзки по пътя
- `$ basename <път>` - връща базовото име на файл
  - работи върху пътя като низ, не се интересува от символни връзки
- `$ dirname <път>` - връща пътя на директорията, в която е файл
  - работи върху пътя като низ, не се интересува от символни връзки
- `$ readlink <път>` - дереферира конкретна подадена връзка

## Поведение на други команди спрямо символни връзки

- Повечето команди просто дереферират символните връзки
- Някои команди, напр. `$ stat`, не дереферират символни връзки по подразбиране, но имат опция, която ги кара да ги дереферират

```
$ stat /usr/bin/sh
  File: /usr/bin/sh -> bash
  Size: 4           Blocks: 8
  IO Block: 4096   symbolic link
```

```
$ stat -L /usr/bin/sh
  File: /usr/bin/sh
  Size: 1071664    Blocks: 2096
  IO Block: 4096   regular file
```

...

## Заето и свободно място на файловите системи

- `$ df <път>` - отпечатва статистика за заето и свободно място на файловата система, в която е подаденият път (или на всички файлови системи, ако не е подаден път)
  - `-h, --human-readable`
  - `-H, --si`
  - `-i, --inodes`
  - `-T, --print-type`
- `$ du <път>` - рекурсивно изписва заетото място на поддelenата директория и нейните поддиректории
  - `-h, --human-readable`
  - `--si`
  - `-s, --summarize` - показва само едно число, общото заето място
  - `-a` - показва информация и за обикновени файлове, не само за директории

Търсене из файловата система  
Операционни системи, ФМИ, 2022/2023

## Командата find

- Командата `$ find <директория>` изписва на нови редове имената на всички файлове в подадената директория, и в нейните поддиректории
- Редът на изведените файлове не е определен<sup>1</sup>

```
$ find ~/texts  
/home/human/texts/foo.txt  
/home/human/texts/sacred_texts  
/home/human/texts/sacred_texts/yoda.txt  
/home/human/texts/bar.txt
```

---

<sup>1</sup>зависи от имплементацията на файловата система

## Командата `find`

- За разлика от `ls`, изходът на `find` е четим и от програми, не само от хора
- Ползвайте `ls` в конзолата; за реални проблеми и задачи, ползвайте само `find`

## Използване на find

- `find` може много неща - винаги гледайте `find(1)`
- `$ find [опции] [директория] [фильтри] [действия]`
  - глобални *опции*, които променят начина, по който открива файлове<sup>2</sup>
  - *фильтри*, които филтрират файловете в подадения път
  - *действия*, които казват на `find` какво да направи с откритите файлове

---

<sup>2</sup>глобалните опции ще ги ползвате рядко

## Филтри на find

- type - ограничава файловете по тип
  - -type f - само обикновени файлове
  - -type d - само директории
  - -type l - само символни връзки

## Филтри на find

-name - ограничава файловете по име

- `-name foo.txt` - файлове, които се казват `foo.txt`
- `-iname bar.txt` - файлове, които се казват `bar.txt`, без да гледаме разликата между главни и малки букви
- поддържа и *globs* - за това ще говорим малко по-късно

## Филтри на find

- `-maxdepth <число>, -mindepth <число>` - ограничаваме се само до файлове на зададените дълбочини от дървото, спрямо подадения път

## Филтри на find

- Има много други филтри - за права, за време на създаване и др.
- Гледайте `find(1)`

## Комбиниране на филтри на find

- можем да комбинираме по няколко филтъра наведнъж
- `-o, -not, '(' , ')'` - логически операции<sup>3</sup>

```
$ find /etc -type f -name nginx.conf
```

```
$ find /etc -type d -o -name bar
```

```
$ find . '(' -type f -name foo.conf ')' -o -name bar
```

---

<sup>3</sup> скобите са в кавички, защото иначе имат специално значение за shell-а - за това ще говорим малко по-късно

## Действия на find

- **-print** - отпечатва имената на файловете на нови редове (това е действието по подразбиране)
- **-ls** - отпечатва данни за файловете във формат, подобен на `ls -l`
- **-delete** - изтрива всички открити файлове
- **-exec <команда> ;** - изпълнява подадената команда за всеки файл, заменяйки низа {} в командата с името на файла  
`$ find /home/human -type f -exec cat {} ';'`

# Още функционалности на shell-а

## Операционни системи, ФМИ, 2022/2023

## Генериране на аргументи

- Много команди приемат произволен брой аргументи
- Shell-ът предоставя начини да генерираме списъци от аргументи по някакви критерии

## Генериране на аргументи по имена на файлове: globbing

Използвайки специални за shell-а символи, можем да подадем всички имена на съществуващи файлове, които съответстват на някакъв шаблон (globbing pattern)

- ? - съответства на един произволен символ
- \* (wildcard) - съответства на произволен брой произволни символи
- [abc] - множество от символи, заградени в квадратни скоби, съответства на точно един от изброените символи

## Globbing (примери)

- `cat /tmp/*.txt`
- `echo foo_???.png`
- `ls /[du]*`
- `ls /[a-z]*` - в множествата можем да имаме и интервали от символи

## Специални символи и екраниране

- Както видяхме, някои символи (например \*) имат специални значения за shell-а и не се подават директно като аргумент
- Може да искаме да подадем някой специален символ като част от аргумент на команда, без да изпълняваме специалното му значение - това се нарича *екраниране (escaping)*

## Екраниране на специални символи

- Когато искаме да екранираме някой символ, можем:
  - Да заградим негово обкръжение в кавички
  - Да сложим символа \ пред него
- `cat *.txt` - прочети всички файлове, чиито имена завършват `c.txt`
- `cat '* .txt'` - прочети файлът, който се назова `*.txt` (звездичка-точка-txt)
- `cat \*.txt` - същото като горното
- `find . -type f -exec cat {} \;` - символът ; е част от синтаксиса на `find`, и затова го екранираме

## Пренасяне на команди

- Можем да екранираме символа за нов ред, като сложим \ пред него
- Можем да се възползваме от това, за да пренасяме команди на нов ред

```
$ cat file_1 file_2 \
    file_3 file_4 file_5
```

## Единични и двойни кавички

- Единичните кавички екранират всички символи освен себе си
- Двойните кавички екранират всички символи освен себе си и символа \$
  - след малко ще говорим за специалното значение на \$

## Аргументи, съдържащи кавички

- Ако искаме да подадем аргумент, съдържащ единична кавичка, трябва да го оградим в двойни кавички:

```
$ echo "I'm trapped"  
I'm trapped
```

- Как подаваме аргумент, съдържащ буквално и двата вида кавички?

```
$ echo 'A single quote (') and a double quote ("')  
A single quote (') and a double quote ("")
```

```
$ echo "A single quote ('') and a double quote (\"\"\")"  
A single quote (') and a double quote (")
```

## Globbing при find

- Опциите `-name` и `-iname` на команда `find` могат да приемат `glob`
- Нужно е globbing шаблоните, които подаваме на `find`, да бъдат екранирани, за да може да се интерпретират от `find`, а не от shell-a

## Globbing при find

- `find ~ -type f -name '*.txt'`
- `find /home -mindepth 1 -maxdepth 1 -iname '*ivan*'`

## Brace expansion

- Синтаксисът {<низ1>,<низ2>,<низ3>,...} позволява да генерираме аргументи с декартово произведение на низове
- За разлика от globs, няма общо с имена на файлове (но може да се комбинира с globs)

```
$ echo {foo,bar,baz}  
foo bar baz
```

```
$ echo my_{foo,bar,baz}.txt  
my_foo.txt my_bar.txt my_baz.txt
```

```
$ echo {foo,bar,baz}_{1,2}  
foo_1 foo_2 bar_1 bar_2 baz_1 baz_2
```

```
$ echo {foo,bar,baz}{1,}  
fool foo bar1 bar baz1 baz
```

## Brace expansion

```
$ mv -v foo{,.txt}  
foo -> foo.txt
```

## Brace expansion

Brace expansion и globbing могат да се комбинират, като първо се изпълнява brace expansion:

```
$ cat /home/{pesho,gosho}/documents/*.txt
```

Данни във файлове  
Операционни системи, ФМИ, 2022/2023

## Данни във файлове

- В UNIX-базираните операционни системи, данните на файловете са масив от байтове
- Съществуват и други начини да се организират данните
  - Някои операционни системи<sup>1</sup> организират данните като последователност от записи

---

<sup>1</sup>операционните системи на някои mainframe-и

## Интерпретация на данните

- *'s all bytes, man*
- The data is in the eye of the beholder
- Всичко се свежда до това как интерпретираме байтовете
- Две основни категории
  - текстови файлове (plain text)<sup>2</sup>
  - binary файлове<sup>3</sup>

---

<sup>2</sup>[http://www.linfo.org/plain\\_text.html](http://www.linfo.org/plain_text.html)

<sup>3</sup>[http://www.linfo.org/binary\\_file.html](http://www.linfo.org/binary_file.html)

## Преглеждане на байтовете

- С командата `$ xxd <файл>` можете да прегледате съдържанието на подаден файл байт по байт

```
$ xxd /tmp/foo.bin
```

```
00000000: 7ec8 a5b2 8353 7bf0 f61e 3196 a0ea 1ef6 ~....S{....1....  
00000010: 3bf9 385b a784 c231 b020 5a78 9bdd 9191 ;.8[....1. Zx....
```

```
$ xxd /tmp/covid.txt
```

```
00000000: 446f 776e 6c6f 6164 206f 7572 2063 6f6d Download our com  
00000010: 706c 6574 652c 2064 6169 6c79 2075 7064 plete, daily upd  
00000020: 6174 6564 204f 7572 2057 6f72 6c64 2069 ated Our World i  
00000030: 6e20 4461 7461 2043 4f56 4944 2d31 3920 n Data COVID-19  
00000040: 6461 7461 7365 742e 0a dataset..
```

- Три колонки: отместване, байтове, интерпретация като текст
- Данните се изписват в шестнадесетична бройна система
  - Това позволява всеки байт да е точно два символа ( $16^2 = 2^8$ )

## Текстови файлове

- Данните представляват поредица от символи
- Всеки символ се кодира като един или повече байтове

## Кодиране на символи

- Може да измислим много начини да кодираме символи като байтове
- ASCII<sup>4</sup> - утвърден стандарт за кодиране на 127 на брой основни символи (латиница, цифри, контролни символи...) в 7 бита
- По-късно се появява нуждата за кодиране на символи извън ASCII - тогава осмият бит влиза в употреба
- Различни *кодови таблици* (ISO/IEC 8859-1, ISO/IEC 8859-15, Windows-1251, KOI8-U...)
  - всеки байт съответства на един символ
  - първите 127 символа са фиксираны на ASCII
  - останалите символи са различни в зависимост от кодовата таблица

---

<sup>4</sup>American Standard Code for Information Interchange

## След кодовите таблици: Unicode

- Unicode е стандарт, обединяващ всякакви символи, които биха ни трябвали
- 1112064 възможни кодирания (code points): от U+0000 до U+10FFFF
- UTF-8 - ползва от 1 до 4 байта на символ, напълно обратно съвместим с ASCII
- UTF-16 - ползва 2 или 4 байта на символ
- UTF-32 - ползва 4 байта на символ
- Модерният софтуер за Linux обикновено ползва UTF-8

## Редове

- Полезно е да мислим за отделни “редове” в текста
- Това го кодираме, като всеки ред го завършваме със *символ за нов ред*
- Различни стандарти:
  - LF (0x0a) - Linux
  - CR (0x0d) - MacOS
  - CRLF (0x0d0a) - Windows/DOS
- Символите CR и LF са част от ASCII

## Преглеждане на текстови файлове

- \$ cat <файл> - изписва съдържанието на файл в конзолата

## Преглеждане на текстови файлове

- \$ `head <файл>` - изписва първите 10 реда от файл в конзолата
  - \$ `head -n <N> <файл>` - изписва първите `<N>` реда от файла
  - \$ `head -n -<N> <файл>` - изписва всички без последните `<N>` реда от файла
- \$ `tail <файл>` - изписва последните 10 реда от файл в конзолата
  - \$ `tail -n <N> <файл>` - изписва последните `<N>` реда от файла
  - \$ `tail -n +<N> <файл>` - изписва редовете от файла, започвайки от `<N>`-тия
  - \$ `tail -f <файл>` - изписва последните редове от файла и чака, изписвайки нови редове, добавени след това

## Преглеждане на текстови файлове

- `$ less <файл>` - pager за файлове (програма, която показва само една страница от текста наведнъж на екрана) - работи по същия начин, както `$ man`, но за файлове

## Формати, базирани на текст

- Текстовите файлове могат да се използват за кодиране на произволни машинно-четими данни
- Таблици (пример: /etc/passwd - вижте passwd(5))
- Формати, базирани на текст със специфична структура (plain text email, Org-Mode, Markdown, ...)
- Формати за структурирани данни (JSON, XML, ...)
  - Конкретни формати, базирани на горните (GPX, SVG, DOCX/XLSX<sup>5</sup>, ...)

---

<sup>5</sup>въщност, тези са компресиран текст, но все пак текст

## Не-текстови (бинарни) файлове

- Всеки файл, който не съдържа текст, кодиран по начините, които описахме, наричаме бинарен / двоичен / binary файл

## Формати, не-базирани на текст

Много формати не кодират данните си като текст

- Текстът заема повече място<sup>6</sup>
- Тъй като заема повече място, отнема повече време да се обработи

---

<sup>6</sup>текстът има по-ниска ентропия

## Формати, не-базирани на текст

Използването на текстови формати става все по-разпространено

- След *компресиране*, текстът става сравним по размер с еквивалентен, добре оптимизиран не-текстов формат
- Мястото за съхранение и процесорното време са евтини
- Работата с текстови формати е по-лесна за хора, а човешкото време е скъпо

## Формати, не-базирани на текст

Все пак, бинарните формати са по-смислени, когато имаме нужда от високо ниво на компресия и производителност

- Изображения (JPG, PNG, ...)
- Audio/video (flac, mp3, mkv, ...)
- Изпълними файлове (ELF, exe, ...)
- Архиви (tar, zip, ...)

## Кодиране на числа в бинарни файлове

- Много често ще се налага да работим с бинарни данни, които кодират (цели) числа
- Важно е да преговорите начините за кодиране на число
- Little-endian и big-endian подредба на байтовете
- Числа със знак - two's complement кодиране

## Преглеждане на бинарни данни

- \$ `xxd` - универсален инструмент, ползвайте го много - вече говорихме за него
- \$ `strings <файл>` - отпечатва всички последователности от символи от файла, които могат да се интерпретират като текст

## В какъв формат е този файл?

- Командата `$ file <файл>` показва информация за формата на подадения файл
- Има база от формати и парчета данни, които често се срещат в началото на файл от съответния формат (`/usr/share/file/magic.mgc`)
- Не я интересува името на файла

## Разширения

- По конвенция, имената на някои файлове завършват с *разширение*: няколко символа, предшествани от точка
- Разширението подсказва формата:
  - `baba.jpg` вероятно е във формат JPEG
  - `dyado.pdf` вероятно е във формат PDF
- Linux не се интересува от разширенията
  - но някои приложения по някога се интересуват

# Компресиране и архивиране

## Операционни системи, ФМИ, 2022/2023

# Компресиране и архивиране

- **Компресиране** - Алгоритми за кодиране, които (се опитват да) намалят обема на данните
  - compress (Lempel-Ziv)
  - gzip (DEFLATE)
  - bzip2
  - xz (lzma)
  - brotli
  - zstd
- **Архивиране** - Файлови формати, които опаковат няколко файла в един
  - cpio
  - tar (tarball)
- Някои формати комбинират архивиране и компресиране в едно
  - rar
  - 7zip
  - zip

## Команди за архивиране

- `$ tar -c -f baba.tar cookies.txt banitsa.txt`
  - създава файл `baba.tar`, който съдържа файловете `cookies.txt` и `banitsa.txt`
- `$ tar -x -f foo.tar`
  - разархивира архива `foo.tar` в текущата директория

## Команди за архивиране

- \$ tar -cf project.tar project - създава архив `project.tar`, който съдържа цялата директория `project`
- По конвенция е добре архивите да съдържат единствено една директория, която се назова по същия начин като архива
  - всеки път, когато пратите на някого архив без директория в него, 128 пингвина изчезват безследно

## Команди за архивиране

- Опцията `-v` кара `tar` да изписва имената на файловете вътре
- Опцията `-t` кара `tar` да не прави нищо с архива
- Ако ги комбинираме, може да видим списък с файловете в архив

## Команди за компресиране

- `$ gzip <файл>` - създава `<файл>.gz`, компресираайки `<файл>` с алгоритъма DEFLATE, и изтрива оригиналния файл<sup>1</sup>
- `$ gzip -d <файл>.gz` - създава `<файл>`, декомпресираайки и изтривайки `<файл>.gz`
- `-k` - кара `gzip` да не трне оригиналния файл
- Повечето команди за компресия имат подобен интерфейс, включително опциите `-d` и `-k`
  - `zstd`, `xz`, `bzip2`

---

<sup>1</sup>`gzip` е пример за команда, която взима предвид разширението

## Формати за компресиране - кой да ползваме?

- gzip го има навсякъде, бърз е, компресира задоволително добре
- xz е много бавен, но компресира уникално добре
- zstd е state of the art, много бърз и компресира много добре, обаче не го има навсякъде

# Формати за компресиране - кой да ползваме?

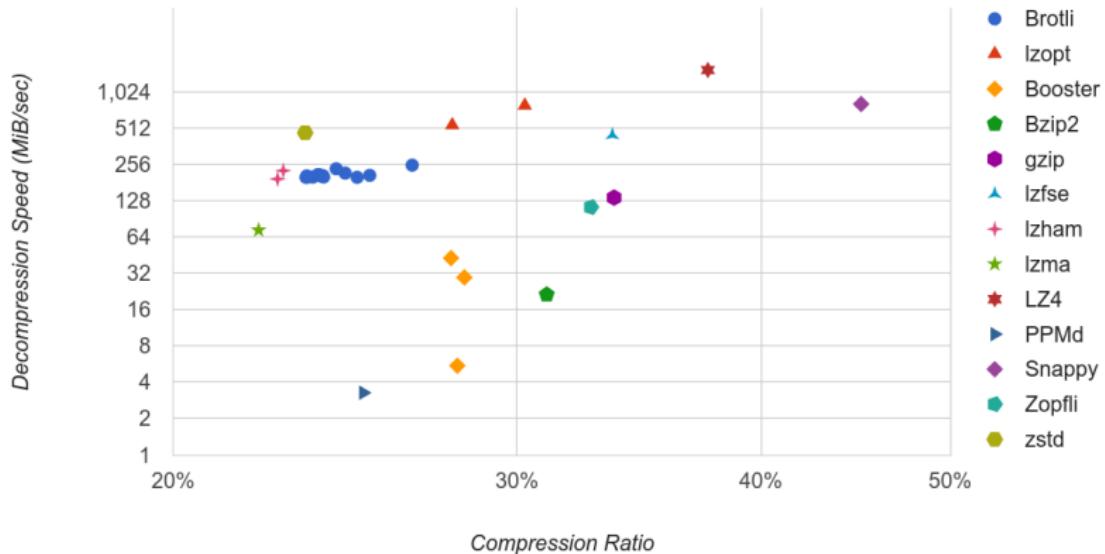


Figure 1: <https://pages.di.unipi.it/farruggia/dcb/>

## Компресиране и архивиране

- Командата `tar` може да направи двете едно след друго вместо нас
- `$ tar --zstd -cf project.tar.zst project` - създава zstd архив `project.tar.zst`, който съдържа директорията `project`
- `$ tar --zstd -cf project.tar.gz project` - създава zstd архив `project.tar.gz` (**лошо!**)
- `$ tar --gzip -cf project.tar.gz project` - създава gzip архив `project.tar.gz`
- `$ tar -caf project.tar.zst project` - създава zstd архив `project.tar.zst`
  - -а измисля формата в зависимост от разширението

## Разархивиране и разкомпресиране

- Командата `tar` може да направи двете едно след друго вместо нас
- `$ tar -xf foo.zst` разархивира архива `foo.zst` в текущата директория
  - `-x` работи за произволни (компресирани) архиви, които `tar` поддържа

## Компресиране и архивиране

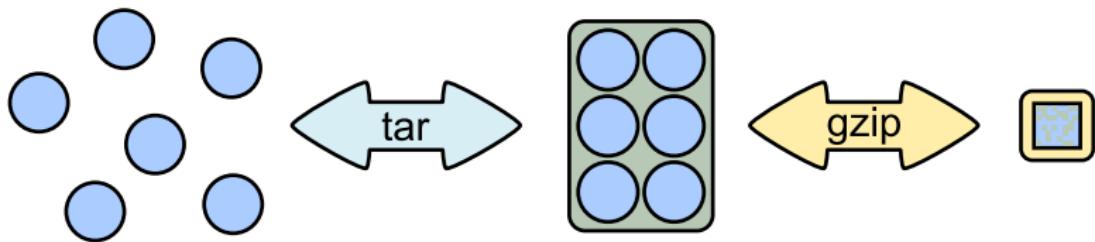


Figure 2: Процес на компресиране и архивиране

Тръби и потоци  
Операционни системи, ФМИ, 2022/2023

## Стандартни потоци

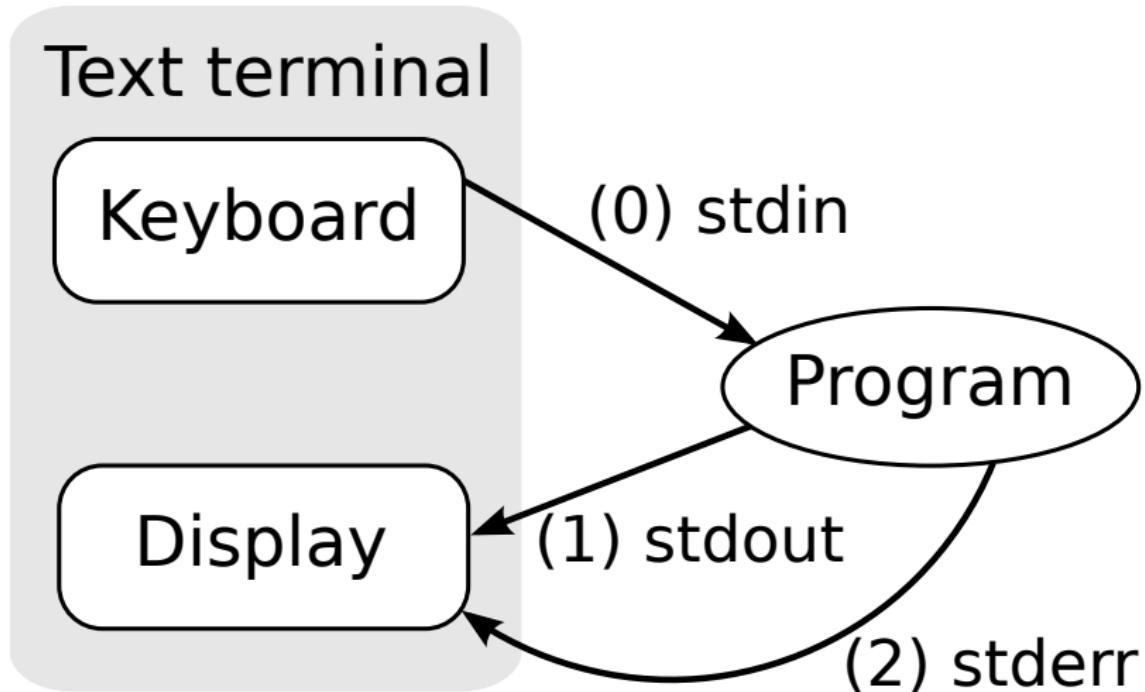


Figure 1: Стандартни потоци

## Стандартни потоци

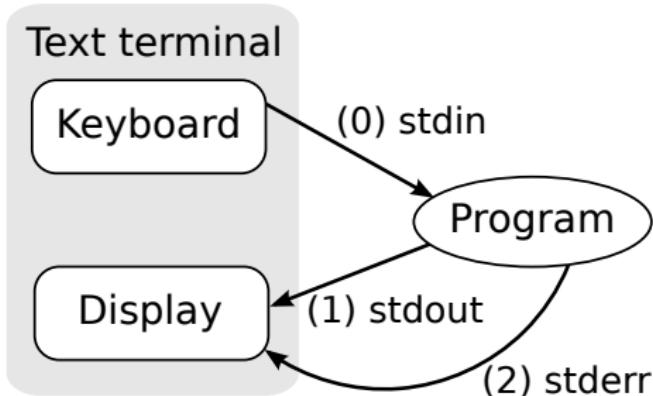


Figure 2: Стандартни потоци

- **stdin** (стандартен вход) - програмата чете входни данни от него
- **stdout** (стандартен изход) - програмата записва изходни данни в него
- **stderr** (стандартен изход за грешки) - програмата изписва съобщения към потребителя в него

## Command substitution

С конструкцията \$(<команда>) може да вложим изхода на една команда в друга команда

- Вложената команда се изпълнява и \$(...) се заменя с цялото съдържание на стандартния ѝ изход
- Ако искаме резултатът от изпълнението на вложената команда да се интерпретира като един цял низ, вместо да се раздели по интервали, трябва да използваме кавички
  - Това е случай, в който ни трябват двойни кавички вместо единични: не искаме да екранираме значението на \$

## Command substitution

- echo "Часът в момента е \$(date +'%H:%M:%S')"
- find / -type f -user \$(whoami)

## Пренасочване

- По подразбиране `stdin` е закачен към “клавиатурата” на терминала, а `stdout` и `stderr` са закачени към “екрана” на терминала
- Shell-ът позволява да ги откачим от терминала и да ги закачим към файлове

## Пренасочване

- `$ my_cmd > my_file`
  - пренасочва `stdout` на `my_cmd` във файла `my_file`
- `$ my_cmd 1> my_file`
  - също пренасочва `stdout` на `my_cmd` във файла `my_file`
- `$ my_cmd 2> my_file`
  - пренасочва `stderr` на `my_cmd` във файла `my_file`
- `$ my_cmd &> my_file`
  - пренасочва `stdout` и `stderr` на `my_cmd` във файла `my_file`
- `$ my_cmd < my_file`
  - пренасочва файла `my_file` към `stdin` на `my_cmd`

## Пренасочване

- `$ date > ~/time_now.txt`
- `$ cowsay < ~/time_now.txt`
- `$ find / -type f -name nginx.conf 2>/dev/null`
  - така игнорираме грешките, които `find` изписва на stderr
  - виртуалният файл `/dev/null` игнорира всички данни, пратени в него
- `$ find / -name nginx.conf 2>/dev/null > results.txt`

## Пренасочване на поток в поток

Можем да пренасочим един поток в друг така:

- `my_cmd 2>&1` - данните, които командата извежда на stderr отиват в stdout и се смесват с данните, които извежда на stdout
- `my_cmd 2>&1 1>/dev/null` - stdout се пренасочва в /dev/null а stderr се пренасочва в stdout
- `my_cmd 1>/dev/null 2>&1` - stderr се пренасочва в stdout, който *вече* е пренасочен в /dev/null: и двете отиват в /dev/null
- `my_cmd 3>&1 1>&2 2>&3` - разменяме stdout и stderr

## Пренасочване с append

- По подразбиране, пренасочването с > презаписва файла, ако той съществува
- Ако използваме >> вместо >, пренасочването става след края на файла, ако той съществува

```
$ echo foo > file
$ echo bar >> file
$ rm -f /etc/passwd 2>> file
$ cat file
foo
bar
rm: cannot remove '/etc/passwd': Permission denied
```

## Тръби

- Тръбите са примитиви на операционната система, които позволяват еднопосочна комуникация между процеси
- Shell-ът ни позволява да използваме тръби, пренасочвайки изхода на една команда във входа на друга
- Командите се изпълняват в паралелно работещи процеси
- Тръбите позволяват да комбинираме прости програми за да решаваме сложни задачи

## Тръби

- `$ echo "my name is $(whoami)" | cowsay`
  - кравата казва резултата от команда
- `$ find /tmp -type f | head -n 5`
  - показваме само 5 от файловете в /tmp
- `$ find /tmp -type f 2>&1 1>/dev/null | head > errors.txt`
  - записваме първите 10 грешки, изведени от find във файл, и игнорираме изведените имена на файлове

# Работа с текст

## Операционни системи, ФМИ, 2022/2023

## Текстът е хубаво нещо

- Текстът се чете и пише лесно от хора и програми
- Можем да комбинираме програми, които работят върху текст, използвайки тръби
  - Така може да решаваме сложни задачи с прости инструменти

## Комбиниране на данните във файлове

- `$ cat <файл1> <файл2> ...` - конкатенира съдържанията на подадените файлове
- `$ paste <файл1> <файл2> ...` - отпечатва редовете на файловете един до друг паралелно, така че да образуват колонки
  - разделителят по подразбиране е TAB
  - можем да го сменим с `-d`

## Комбиниране на данните във файлове

```
$ cat f1
foo
bar
baz
$ cat f2
baba
dyado
$ cat f1 f2
foo
bar
baz
baba
dyado
$ paste f1 f2
foo baba
bar dyado
baz
```

## Броене на текст

Командата \$ wc може да брои:

- байтове (\$ wc -m)
- символи (\$ wc -c)
- редове (\$ wc -l)
- думи (\$ wc -w)

## Броене на текст

- \$ wc -l /etc/passwd - брой редове в /etc/passwd
- \$ who | wc -l - брой логнати потребители

## Броене на текст

```
$ echo 'foo bar' | wc -c  
8
```

```
$ echo 'foo bar' | wc -m  
8
```

```
$ echo 'баба' | wc -c  
9
```

```
$ echo 'баба' | wc -m  
5
```

## Операции със символи

Командата `$ tr` чете текст от `stdin`, прави някаква операция със символите в него и извежда резултата на `stdout`.

```
$ echo 'foo' | tr f b  
boo
```

```
$ echo 1337 | tr 0123456789 abcdefghij  
bddh
```

```
$ echo 'i am screaming' | tr a-z A-Z  
I AM SCREAMING
```

## Операции със символи

Командата `$ tr` може и да трябва символи:

```
$ echo 'socialism CAPITALISM' | tr -d a-z  
CAPITALISM
```

## Операции със символи

Командата `$ tr` може и да “смачква” (squeeze) поредици от еднакви символи до един:

```
$ who  
human    tty1          2023-03-02 17:41
```

```
$ who | tr -s ' '  
human tty1 2023-03-02 17:41
```

## Изрязване на колонки

Командата `$ cut` разглежда текста ред по ред и отрязва специфични колонки.

- Много е удобна за ситуации, в които имаме структуриран текст във вид на таблица

## cut (1/2)

- \$ cat some\_file | cut -c 2-5 - изрежи всеки ред от втория до петия символ
- \$ cat some\_file | cut -c 2- - изрежи всеки ред от втория символ нататък
- \$ cat some\_file | cut -c -8 - изрежи всеки ред от началото до осмия символ

## cut (2/2)

- `$ cat some_file | cut -d ' ' -f 2` - изрежи втората колонка, приемайки, че колонките са разделени с интервал
- `$ cat some_file | cut -d ' ' -f 2,5` - изрежи втората и петата колонка, приемайки, че колонките са разделени с интервал
- `$ cat some_file | cut -d ' ' -f 1-3` - изрежи първата, втората и третата колонка, приемайки, че колонките са разделени с интервал

## Сортиране на текст

Командата `$ sort` сортира редовете в подадения ѝ текст.

- `$ cat some_file | sort` - сортирай редовете по азбучен ред
- `$ cat some_file | sort -r` - сортирай редовете по азбучен ред, наобратно
- `$ cat some_file | sort -n` - приеми, че всеки ред започва с число, и ги сортирай по числата
- `$ cat some_file | sort -h` - приеми, че всеки ред започва с число със SI суфикс (k, m, g), и ги сортирай по числата

## Сортиране на текст

- \$ cat some\_file | sort -k 2,5 - сортирай първо по втората колонка и после по петата
- \$ cat some\_file | sort -t ',' -k 2,5 - сортирай първо по втората колонка и после по петата, приемайки, че колонките са разделени със запетайки
- \$ cat some\_file | sort -r -k 2 - сортирай по втората колонка, на обратно

## Уникални редове

Командата `$ uniq` свежда групи от последователни еднакви редове до едно копие на съответния ред:

```
$ cat some_file
foo bar
foo bar
baz qux
baz qux
baz qux
foo bar
baba dyado
```

```
$ cat some_file | uniq
foo bar
baz qux
foo bar
baba dyado
```

## Уникални редове

Командите `$ uniq` и `$ sort` могат да се използват заедно, ако искаме да видим всички уникални редове:

```
$ cat some_file
foo bar
foo bar
baz qux
baz qux
baz qux
foo bar
baba dyado
```

```
$ cat some_file | sort | uniq
baba dyado
baz qux
foo bar
```

## Уникални редове

\$ uniq -c показва брой срещания на всеки ред, в отделна колонка:

```
$ cat some_file
foo bar
foo bar
baz qux
baz qux
baz qux
foo bar
baba dyado
```

```
$ cat some_file | uniq -c
2      foo bar
3      baz qux
1      foo bar
1      baba dyado
```

## Уникални редове

Как можем да подредим редовете по брой срещания?

```
$ cat some_file
foo bar
foo bar
baz qux
baz qux
baz qux
baz qux
foo bar
baba dyado
```

```
$ cat some_file | sort | uniq -c | sort -n
1      baba dyado
3      foo bar
4      baz qux
```

## comm

Командата `$ comm <файл1> <файл2>` извежда 3 колонки, съдържащи:

- Редовете, които се срещат само в първия файл
- Редовете, които се срещат само във втория файл
- Редовете, които се срещат и в двета файла

`$ comm` работи **само върху сортирани файлове**

## comm

```
$ cat file_1
```

```
bar
```

```
baz
```

```
foo
```

```
qux
```

```
$ cat file_2
```

```
abc
```

```
baz
```

```
qux
```

```
$ comm file_1 file_2
```

```
abc
```

<-- само във файл 2

```
bar
```

<-- само във файл 1

```
baz
```

<-- и в двата файла

```
foo
```

<-- само във файл 1

```
qux
```

<-- и в двата файла

## comm

Опциите -1, -2 и -3 на `$ comm` **премахват** съответната колонка от изхода:

- `$ comm -1 -2 file_1 file_2` - извежда редовете, които се срещат и в двета файла
- `$ comm -1 -3 file_1 file_2` - извежда редовете, които се срещат само във втория файл
- `$ comm -3 file_1 file_2` - извежда редовете, които се срещат в един от файловете, но не и в двета едновременно

## join

Командата `$ join`, която също **работи само върху сортирани файлове**, съединява редовете в двата файла по общи стойности в дадена колонка.

Може да я използвате, когато искате да комбинирате данни от отделни източници за едни и същи обекти.

За информация как се ползва, вижте `join(1)`

# Търсене в текст с grep

## Операционни системи, ФМИ, 2022/2023

## Търсене в текст

Командата `$ grep` може да се използва за търсене в текст:

- `$ grep <низ> <файл>` извежда само редовете от файла, които съдържат подадения низ
- `$ cat my_file | grep <низ>` - когато не е подаден файл като аргумент, `$ grep` чете от `stdin`

## Опции на grep

- Опцията `-n` кара grep да отпечатва и номера на редовете
- Опциите `-B` (before), `-A` (after) и `-C` (context) карат grep да отпечатва зададен брой редове преди/след тези, които съдържат търсения низ
  - `$ cat my_file | grep -B 2 pesho` - за всеки ред, съдържащ “pesho”, отпечатва него и предните два
  - `$ cat my_file | grep -A 3 gosho` - за всеки ред, съдържащ “gosho”, отпечатва него и следващите три
  - `$ cat my_file | grep -C 3 penka` - за всеки ред, съдържащ “penka”, отпечатва него, предните три и следващите три

## Опции на grep

- Опцията `-i` кара grep да не зачита разликата между главни и малки букви (ignore case)
- Опцията `-v` кара grep да инвертира търсенето (намира редовете, които **не** съдържат търсения низ)

## Опции на grep

- grep може да приеме файл като аргумент, вместо да чете текст от `stdin`:
  - '\$ grep 'baba tonka' some\_file'

## Опции на grep

- \$ grep -r <низ> <директория> търси файлове, съдържащи подадения низ, рекурсивно в подадената директория

## Опции на grep

- Преди малко ви излъгах: grep не търси редове, съдържащи подадения низ
- Въщност, grep търси редовете, които отговарят на подадения *регулярен израз*
  - след малко ще обясним как работят регулярните изрази
- Опцията -F кара grep да търси редовете, буквално съдържащи подадения низ
  - този път не ви лъжа

Регулярни изрази  
Операционни системи, ФМИ, 2022/2023

## Регулярни изрази

- Както казахме, grep търси редове в текст, отговарящи на някакъв *регулярен израз*
- Други команди, за които ще говорим, също могат да работят с регулярни изрази (sed, awk, perl ...)

## Регулярни изрази в математиката

Да припомним математическата дефиниция на регулярен израз:

- Имаме азбука на езика и азбука на специалните символи<sup>1</sup>
- Всеки символ от езика е регулярен израз
- ‘0 е регулярен израз (конкатенация)
- ( $\langle \text{регулярен израз} \rangle | \langle \text{регулярен израз} \rangle$ ) е регулярен израз (“или”)
- ( $\langle \text{регулярен израз} \rangle ^*$ ) е регулярен израз (звезда на Клини)

---

<sup>1</sup>специалните символи в този случай са скоби, права черта, звезда

## Регулярни изрази в програмирането (regex)

- Basic regex
  - По подразбиране grep използва такива изрази
  - Различен синтаксис за различните имплементации на grep (GNU grep, BSD grep)
  - Доста малко възможности
- Extended regex
  - grep -E използва extended regex
  - egrep е същото като grep -E
  - Синтаксисът за регулярни изрази, който ще покажем сега, е именно Extended. За разлики между различните варианти, вижте regex(7)
- Perl-compatible regex (PCRE)
  - grep -P използва PCRE
  - Най-пълният синтаксис

## Регулярни изрази в програмирането (regex)

- Вижте `regex(7)`

## Regex syntax (1/13): не-специални символи

- Всяка буква, цифра, интервал и друг не-специален<sup>2</sup> символ match-ва себе си
- grep -E 'baba' търси низа "baba" (защото буквата a match-ва a, а буквата b match-ва b)

---

<sup>2</sup>след малко ще изброим специалните символи и ще кажем какви са специалните им значения

## Regex syntax (2/13): escape codes

- Някои по-странны символи можем да ги “опишем” в регулярен израз, използвайки съответния *escape code*:
  - \t - таб
  - \n - символ за нов ред / line feed (LF)
  - \r - carriage return (CR)
  - \x42 - конкретен символ, изразен чрез неговия код в шестнадесетична бройна система (hex)

## Regex syntax (3/13): точка

- Точката match-ва един произволен символ
- grep 'ba..' match-ва низа "ba" последван от два произволни символа

## Regex syntax (4/13): котви

- Някои специални конструкции са *котви* (*anchors*) и не match-ват конкретен символ, а вместо това фиксират позиция:
  - \$ фиксира *край на ред*
  - ^ фиксира *начало на ред*
  - \> фиксира *край на дума*
  - \< фиксира *начало на дума*
  - \b фиксира *начало или край на дума*

## Regex syntax (5/13): класове

Множество от символи, заградени в квадратни скоби, наричаме **символен клас** (*character class*)

- Целият клас match-ва **един** произволен символ от множеството
- [ foabx] match-ва един от символите f, o, a, b или x
- [a-zA-Z] match-ва произволна латинска буква
- [^a-zA-Z] match-ва произволен символ, който не е латинска буква
  - Символът ^ в началото на символен клас го инвертира
  - Да не се бърка с другото значение на ^ извън символен клас (начало на ред)

## Regex syntax (6/13): класове

Съществуват предефинирани подмножества на символни класове:

- \w вътре в клас е равносилно на a-zA-Z0-9\_
- [:alnum:] вътре в клас изброява всякакви букви и цифри
- Други такива:
  - [:alnum:] [:alpha:] [:cntrl:] [:digit:]
  - [:lower:] [:punct:] [:space:] [:upper:]
- grep -E '[[[:alpha:]]\_.]' match-ва един символ, който е буква, добра черта или точка
  - забележете двойните квадратни скоби

## Regex syntax (7/13): “или”

Операторът | се интерпретира като “или” на два регулярни израза:

- grep -E 'lift|elevator' - match-ва редове, които съдържат “lift” **или** “elevator”

## Regex syntax (8/13): атоми и скоби

- Всяка от конструкциите, match-ващи един символ, които изброихме досега, образува един *атом*
- Можем да групирате няколко атома в *група*, която е еквивалентна на един атом, като ги заградим в скоби
- Операторът “скоби” е с по-голям приоритет от оператора “или”:
  - `grep 'foo (bar|baz) qux'` match-ва “foo bar qux” или “foo baz qux”
- След малко ще говорим за няколко оператора, за които това е полезно

## Regex syntax (9/13): quantifiers

Съществуват **количествени оператори** (quantifiers), които модифицират броя срещания на атома преди тях:

- <атом><sup>\*</sup> означава произволен брой (включително 0) срещания на дадения атом
- <атом><sup>+</sup> означава произволен брой (1 или повече) срещания на дадения атом
- <атом><sup>?</sup> означава 0 или 1 срещания на дадения атом
- <атом><sup>{3,5}</sup> означава 3, 4 или 5 срещания на дадения атом
- <атом><sup>{5,}</sup> означава 3 или повече срещания на дадения атом
- <атом><sup>{,42}</sup> означава най-много 42 срещания на дадения атом

## Regex syntax (10/13): quantifiers (примери)

Няколко примера, използващи файла /usr/share/dict/words, който съдържа много думи на английски език:

- \$ grep -E '^@[stu].{14}\$' /usr/share/dict/words
  - редове, започващи с някоя от буквите s, t или u, дълги общо 15 символа
- \$ grep -E '^@[aeiou].{9}ion\$' /usr/share/dict/words
  - редове, започващи с гласна буква, дълги общо 13 символа и завършващи на “ion”
- \$ grep -E '^c.{15,}\$' /usr/share/dict/words
  - редове, започващи с “c”, дълги над 16 символа
- \$ grep -E '^n.{6,10}c\$' /usr/share/dict/words
  - редове, започващи с “n”, завършващи с “c” и дълги от 8 до 12 символа

## Regex syntax (11/13): quantifiers (примери)

- `grep -E '(ba)+'` match-ва низовете “ba”, “baba”, “bababa” и т.н.

## Regex syntax (12/13): рефериране на групи

- Досега видяхме, че скобите групират текста, match-нат от някакъв израз, в един атом
- Освен това, групираният текст може да се реферира на друго място в израза, използвайки конструкцията \<номер>
  - номерата на групите се индексират спрямо позицията на лявата скоба в израза
  - група 0 е целият match-нат текст
  - група 1 е текстът, match-нат от израза, ограден от най-левата лява скоба
  - група 2 е текстът, match-нат от израза, ограден от втората най-левая скоба
  - т.н.
- grep '([aouei]).\1' match-ва гласна буква, последвана от произволен символ и след това **същата** гласна буква

## Regex syntax (13/13): екраниране

Всички специални символи могат да се *екранират* с “\”

- \ ( match-ва лява скоба
- \\* match-ва звездичка

И т.н.

# Трансформиране на текст със sed

## Операционни системи, ФМИ, 2022/2023

## sed

- Командата `$ sed` (Stream EDitor) чете текст, променя го по някакво правило, и го извежда
- Има собствен език за изразяване на трансформации (изразите, написани на този език, ще наричаме *sed изрази*)
- Най-често ще използваме трансформацията “замяна на текст” (*substitute*):

```
$ echo 'the quick brown fox' | sed 's/quick/fast/'  
the fast brown fox
```

- `sed` е много мощен и ще покрием много малка част от възможностите му: за повече, вижте `sed(1)`

## sed: оператори при замяна

Операторът `/g` на края на израз за замяна кара замяната да се извърши за всички срещания в рамките на всеки ред, а не само за първото:

```
$ echo 'quick, quick, quick!' | sed 's/quick/fast/'  
fast, quick, quick!
```

```
$ echo 'quick, quick, quick!' | sed 's/quick/fast/g'  
fast, fast, fast!
```

## sed: екраниране и разделители

sed поддържа различни разделители, в случаите, в които искаме да заменяме низове, съдържащи специални символи:

```
$ echo 'my home dir is /home/human' \
    | sed 's:home/human:home/baba:g'
my home dir is /home/baba
```

Също така, можем да екранираме разделителя с \:

```
$ echo 'my home dir is /home/human' \
    | sed 's/home\/human/home\/baba/g'
my home dir is /home/baba
```

## sed: regex

sed поддържа регулярни изрази и рефериране на групи (backreferences):

```
$ cat file
Parenthesis allow you to store matched
patterns.
```

```
$ cat file | sed -E 's/[^aouei]{2,}/_/g'
Pare_esi_a_o_ou_o_ore_a_ed
pa_e_
```

```
$ cat file | sed -E 's/(.)\1/\[\1\1\]/g'
Parenthesis a[ll]ow you to store matched
pa[tt]erns.
```

Както при grep, опцията -E кара sed да използва extended regex синтаксиса вместо POSIX regex.

## sed: inplace

- Опцията `-i` на `sed` позволява да модифицираме файл `in-place`
- `sed -i 's/baba/dyado/g' foo.txt` – заменя всички срещания на `baba` с `dyado`, презаписвайки `foo.txt`
- `sed -i 's/baba/dyado/g' /etc/*.conf` – заменя всички срещания на `baba` с `dyado`, презаписвайки всички файлове в `/etc`, чиито имена завършват с `.conf`

# Обработка на структуриран текст с awk

## Операционни системи, ФМИ, 2022/2023

## awk

- awk е език за програмиране, специализиран за обработка на табличен текст<sup>1</sup>
- Името идва от първите букви на имената на тримата създатели
- Командата awk приема скрипт, написан на езика awk, и го изпълнява върху текста, получен на `stdin`
  - `$ awk -f <скрипт>` чете awk-скрипт от файл вместо от аргумент

---

<sup>1</sup>Aho, Alfred V., Brian W. Kernighan, and Peter J. Weinberger. The AWK programming language. Addison-Wesley Longman Publishing Co., Inc., 1987

## awk скриптове

```
$ cat /etc/passwd | awk -F ':' '$1 == "ivan" { print $5 }'  
Ivan Ivanov
```

- Тази команда изписва съдържанието на петата колонка от редовете на `/etc/passwd`, чиято първа колонка е `ivan`.
- Опцията `-F` позволява да изберем разделител за колонки
  - по подразбиране, колонките се разделят по произволно количество whitespace

## awk скриптове

- awk скриптовете имат следната структура:

```
<тест> { <действие>; <действие> ... }  
<тест> { <действие>; <действие> ... }  
...
```

- За всеки ред от входа се изпълняват действията, чиито тестове са успешни

## BEGIN и END

- BEGIN { <действие> ... }
  - изпълнява се в началото на цялото изпълнение
- END { <действие> ... }
  - изпълнява се в края на цялото изпълнение

## awk променливи

- Всеки awk скрипт може да ползва някои вградени променливи:
  - \$0 - съдържанието на текущия ред
  - \$1, \$2, \$3, ... - съдържанието на първата, втората и т.н. колонка от текущия ред
  - NF - брой колонки в текущия ред
  - NR - брой редове, прочетени досега (т.е. номер на текущия ред)
- Освен това, можем да използваме потребителски променливи
- С опцията ‘-v =’ можем да подадем глобална променлива от shell-а на awk скрипта

## Пример

Да преброим всички редове в `/etc/services`, които съдържат името на даден сървис в първата колонка, подавайки името на сървиса (напр. “asp”) като променлива:

```
$ cat /etc/services awk -v service='asp' \
'BEGIN { i = 0 } $1 == service { i += 1 } END { print i }'
2
```

## Регулярни изрази в awk

- Операторът ~ /<регулярен израз>/ е тест за регулярен израз
- \$ who | awk '\$1 ~ /^s[0-9]\*\$/ ' - отпечатва редовете от изхода на \$ who, които имат низ, състоящ се от s, последвано от произволен брой цифри, в първата колонка

## awk съкратен запис

- тестът може да липсва - тогава се подразбира `true`
  - `'awk '{ print $3 }'` изписва третата колонка от всеки ред
- действието може да липсва - тогава се подразбира `print $0`
  - `awk '$1 == "ivan"'` изписва всички редове, чиято първа колонка е `ivan`

## Форматиран изход в awk

- Освен отпечатване на променливи с `print <променлива>`, можем да отпечатваме форматиран изход с `printf()`, която работи като съответната функция в C

```
$ who | awk '  
    BEGIN      { i = 0; all = 0; }  
                { all += 1 }  
    $2 ~ /^pts/ { i += 1 }  
    END        {  
        printf(  
            "%.*f%% of users are logged in from PTS\n",  
            (i/all)*100  
        )  
    } '
```

33.33% of users are logged in from PTS

## Още awk

- Вижте awk(1)

Текстови редактори  
Операционни системи, ФМИ, 2022/2023

## Интерактивни текстови редактори

- Досега показвахме начини за автоматична обработка на текст
- Понякога се налага да редактираме текст *интерактивно*
- Това го правим с помощта програма, наречена *интерактивен текстов редактор*, или накратко, *текстов редактор*

## Видове текстови редактори

- Спрямо вид интерфейс
  - Терминални
  - Графични
  - Специални (напр. за незрящи)
  - Web-базирани
- Спрямо начин на редактиране
  - Редактиращи по един ред наведнъж - приличащи на редактора `ed`
  - Базирани на режими (modal) - приличащи на редактора `vi`
  - Приличащи на редактора `emacs`
  - Приличащи на `pico/nano`, Microsoft Notepad, ...

## Текстови редактори в Linux

- Повечето Linux системи идват с инсталриран терминален текстов редактор, подобен на `vim`.
  - В този курс ще се научите да използвате `vim`.
- Можете да инсталрирате и използвате и много други терминални текстови редактори (`nano/pico`, `emacs`, `neovim`)
- Терминалните текстови редактори позволяват работа с текст през отдалечени терминали, свързани по всевъзможен начин със системата, включително много бавна връзка

## vi/vim

- Редакторът `vi` е създаден през 1976 за UNIX
- Редакторът `vim` (Vi IMproved) използва подобен интерфейс но има доста повече възможности
- В последно време има много fork-ове и варианти на подобни редактори (напр. neovim)

## vim

- \$ vim <име на файл> - стартира редактора vim върху някакъв файл

## `vim` - режими

Във всеки един момент редакторът е в един даден **режим**:

- **Normal mode** - клавиатурата извършва навигация, движение на курсора, манипулиране на текст
- **Insert mode** - клавиатурата въвежда символи на мястото, където е курсорът
- **Command mode** - клавиатурата въвежда команда в полето за команди
- и други, за които няма да говорим сега

## `vim` - движение и Normal Mode

- Клавишът `ESC` превключва редактора в Normal mode
- Докато сме в Normal Mode, можем да движим курсора
  - `h` го движи наляво
  - `j` го движи надолу
  - `k` го движи нагоре
  - `l` го движи надясно
  - `w` го движи с една дума напред
  - `b` го движи с една дума назад
  - `gg` го движи до началото на буфера
  - `G` го движи до края на буфера
  - ...

## vim - Insert mode

- Има няколко начина да се влезе в Insert mode от Normal mode
  - i влиза в Insert mode на текущата позиция
  - a влиза в Insert mode след текущата позиция
  - o влиза в Insert mode на следващия ред
  - ...
- Докато сме в Insert mode можем да въвеждаме текст
- ESC ни връща обратно в Normal mode

## vim - команди

- С клавиша : можем да преминем от Normal mode в команден режим
- vim поддържа много команди, които правят различни неща
- :w записва текущия файл
- :q излиза от редактора
- :q! излиза без да пита
- :wq - командите могат да се комбинират

## `vim` - още информация

- Командата `:help` във `vim`
- `$ vimtutor` - команда, която ви учи как да използвате `vim`
- Много информация в интернет

# Работа с процеси

## Операционни системи, ФМИ, 2022/2023

## Работа с процеси

- Ще си говорим за управление на процеси в UNIX, по-конкретно Linux
- В началото на курса загатнахме няколко понятия, свързани с процеси
  - Понятието *процес* абстрагира *работеща програма*
  - Операционната система извършва *времеделене* (preemptive scheduling) между работещите процеси

## Вилици

- При стартиране на операционната система, тя създава един процес, който изпълнява програма, наречена `init`
  - има различни програми, които могат да изпълняват ролята на `init`
- Всички други процеси се стартират с механизъм, наречен `fork`
  - при `fork` на процес, той се разделя на две копия: родител и дете
  - контекстът се *копира*, а не се *споделя*<sup>1</sup>
- `init` програмата има ролята да стартира всички други, програми, които са нужни за работата на операционната система, използвайки `fork`
- Всички процеси образуват дърво с корен `init`

---

<sup>1</sup>въщност, паметта се копира семантично, но реално се извършва CoW

## Още вилици

По-нататък в курса ще говорим за подробности относно стартиране на процеси

- Ще видим как една програма може да поиска операционната система да `fork`-не нейния процес
- Ще видим как shell-ът стартира програми и ще се научим как да пишем програми, които стартират други програми

Но преди това, да видим как да работим с процесите, които вече съществуват

## Контекст на процес

Всеки процес има *контекст*, част от който са някои важни атрибути:

- PID - номер на процеса, по който можем да го идентифицираме
  - Процесът `init` има PID=1
- PPID - номер на процеса родител на дадения
  - Процесът `init` има PPID=0
- Nice value - стойност, по която се определя приоритета на процеса
  - по-високо nice value означава по-нисък приоритет
  - между -20 и 19, като за повечето процеси е 0
- Security context (ще говорим за него след малко)
- Памет, която процесът може да достъпва
- Отворени файлове (*файлови дескриптори*)
- И други

## Security context

Всеки процес има няколко атрибути, свързани с потребители и изолация:

- UID, GID - потребител и група, собственици на процеса
- Effective UID, Effective GID - потребител и група, с които процесът се представя, когато се опитва да работи с някакъв обект
  - обикновено ефективните и реалните потребител и група съвпадат
  - някои специални програми се възползват от това, че могат да бъдат различни
  - при изпълнение на програма със SUID/SGID бит биха били различни
- и други...

## Състояния на процесите

В даден момент един процес може да е в едно от няколко състояния:

- R – running - в опашката с работещи процеси
- D – uninterruptible sleep - чака събитие (обикновено вход/изход на данни) и не може да бъде спрян
- S – interruptible sleep - чака събитие и може да бъде спрян
- T – stopped (job control) - паузиран от потребител
- t – stopped by debugger - паузиран от дебъгер
- Z – defunct (“zombie”) - приключи и чака родителят му да обработи резултата
- и няколко други, които не са важни в момента

## Преглеждане на информация за процесите

- Директорията /proc съдържа информация за всички процеси
- За всеки процес има директория /proc/<PID>, в която има виртуални файлове, съдържащи информация за този процес
- Има няколко команди - инструменти, които четат информацията в /proc и я показват в удобен вид
  - \$ ps - показва информация за дадени процеси в момента на извикване
  - \$ top - показва информация за процесите в интерактивен режим
  - \$ pstree - рисува дърво с процесите

## Малко повече за ps

- По подразбиране изписва данни за процесите в текущия терминал
- \$ ps -e изписва данни за *всички* процеси
- \$ ps -u <потребител> изписва данни за процесите на дадения потребител

## Малко повече за ps

```
$ ps  
 PID TTY      TIME CMD  
14928 pts/4    00:00:00 bash  
15353 pts/4    00:00:00 ps
```

```
$ ps -e | head -n 5  
 PID TTY      TIME CMD  
 1 ?    00:00:14 init  
 2 ?    00:00:00 kthreadd  
 3 ?    00:00:00 rcu_gp  
 4 ?    00:00:00 rcu_par_gp
```

```
$ ps -u pesho  
 PID TTY      TIME CMD  
7557 ?    00:00:00 sshd  
7558 pts/6    00:00:00 bash
```

## Малко повече за ps

- Опцията -f казва на ps да изписва повече информация
  - \$ ps -ef - показва много информация за всички процеси
- С опцията -o можем да включим специфични колонки в изхода на ps
  - \$ ps -u pesho -o pid,ppid,cmd - за всеки процес на потребителя pesho, изписва неговия PID, родителски PID и команда

## Малко повече за ps

- Можем да именуваме колонките

```
$ ps -e -o user=account,pid=process_id | head -n 4
account  process_id
root          1
root          2
root          3
```

- Можем и да скрием хедъра

```
$ ps -e -o user=,pid= | head -n 3
root          1
root          2
root          3
```

## Сигнали

На всеки процес може да му бъде изпратено специално съобщение, наречено **сигнал**

- Има много (около 30) сигнала, които могат да бъдат използвани
- Ще разгледаме няколко, които са по-важни
- Когато получи сигнал, с процес може да се случи едно от няколко неща
  - нищо
  - да бъде прекратен (убит)
  - да бъде паузиран / продължен
  - да бъде уведомен за сигнала

## Сигнали

Сигнал	Описание	default
SIGHUP(1)	праща се на процес, когато затворим терминала	kill
SIGINT(2)	праща се при Ctrl-C	kill
SIGKILL(9)	убива процеса директно; не може да се маскира	kill!
SIGSEGV(11)	праща се при непозволен достъп до памет	kill
SIGTERM(15)	убива процеса, но може да се маскира	kill
SIGSTOP(19)	паузира процеса	stop/pause!
SIGCONT(18)	продължава паузиран процес	continue!

- Някои сигнали могат да се “маскират”
  - процесът може да поиска да не се изпълни поведението по подразбиране на сигнала
  - вместо това, процесът просто бива уведомен за сигнала и може да направи нещо друго
- Някои сигнали не могат да се маскират
  - те са отбелязани с ! в таблицата

## Пращане на сигнали

- \$ kill -<СИГНАЛ> <PID> праща дадения сигнал на дадения процес
  - \$ kill -TERM 42 - праща сигнал SIGTERM на процес със номер 42
- \$ killall -<СИГНАЛ> <име> праща дадения сигнал на всички процеси с даденото име
  - \$ kill -INT bash праща сигнал SIGINT на всички процеси bash
- Ctrl-C в терминала праща SIGINT на текущия процес, а Ctrl-Z праща SIGSTOP
  - за Ctrl-Z и SIGSTOP ще говорим след малко
- Ако не подадем конкретен сигнал, се праща SIGTERM

## Job control

- В една терминална сесия може да имаме няколко работещи процеса едновременно
- Три възможни състояния на всеки процес
  - Foreground (работи и клавиатурата е закачена към `stdin` на процеса)
  - Background (работи във фонов режим, `stdin` е откачен)
  - Stopped (паузиран във фонов режим)
- Освен по своя PID, процесите в една терминална сесия се идентифицират и по число, наречено *Job ID*

## Job control

- Командата `$ jobs` показва всички процеси в текущата сесия и техните job ID-та
- Конструкцията `$ <команда> &` пуска команда във фонов режим
  - командата си работи в отделен процес, а през това време можем да продължим да използваме shell-а
- `Ctrl-Z` паузира текущо-работещия процес
  - използва SIGSTOP
- Командата `$ fg <job id>` закача фонов процес към терминала
  - ако е бил паузиран, го стартира
- Командата `$ bg <job id>` стартира паузиран процес във фонов режим

# Изпълнение на програми

## Операционни системи, ФМИ, 2022/2023

## Изпълнение на команда

Когато пишем команда<sup>1</sup> в shell-а, се случват няколко неща:

- ➊ Shell-ът намира файла, в който се намира програмата, изпълняваща дадената команда
  - след малко ще обясним как се случва това “намиране”
  - с командата `$ which <команда>` можем да разберем коя е програмата, асоциирана с дадена команда
  - например, програмата на командата `$ whoami` може да е файлът `/usr/bin/whoami`
- ➋ Стартира се процес, в който да се изпълни въпросната програма
- ➌ Програмата се изпълнява от операционната система
  - по-нататък ще видим как точно става това

---

<sup>1</sup>тук се има предвид команда, която не е вградена в самия shell

## Изпълнение на програма от файл

Вместо да пишем име на команда, shell-ът позволява директно да изпълним програма от даден изпълним файл

- Това става, като напишем пълен път до програмата
- \$ /usr/bin/whoami изпълнява директно командата whoami
- Трябва да имаме execute permission върху изпълнимия файл

## Интерпретатори

При изпълнение на програма, операционната система намира *интерпретатор*, който да я изпълни. Можем да разгранишим два вида такива интерпретатори:

- Интерпретатори за програми, компилирани за текущата машина
  - Повечето програми в света на Linux са във формат ELF, и “интерпретаторът” е библиотека, наречена `ld-linux-x86-64.so`<sup>2</sup>
  - Интерпретаторът се грижи да зареди нужните библиотеки (ако има такива<sup>3</sup>) и да “скочи” на функцията `main` в кода на програмата
- “Истински” интерпретатори, които четат подадената програма и я изпълняват
  - Програмите от този вид ще наричаме *скриптове*

---

<sup>2</sup>тук вместо x86-64 може да стои произволна друга архитектура

<sup>3</sup>потърсете информация за “static linking” и “dynamic linking”

## Shebang

Обикновено скриптовете се изпълняват, като ги подадем като аргумент на съответния интерпретатор.

- Ако скриптът започва със специален низ от вида `#!<интерпретатор>`, операционната система автоматично ще използва този интерпретатор при опит за изпълнение на файла
- Например, ако скриптът `foo.sh` започва с `#!/bin/bash`, операционната система ще изпълни `/bin/bash` с аргумент `foo.sh`

## Shebang

Можем да пишем скриптове за произволен интерпретатор:

```
$ cat ./filter.awk
#!/usr/bin/awk -f

{ print $1 }

$ who | ./filter.awk
pesho
gosho
ivan
```

Изпълнението на файла `./filter.awk` е същото като да изпълним `/usr/bin/awk -f ./filter.awk`

## Shebang

- В този курс ще пишем скриптове, които се интерпретират от shell-a Bash
- Всеки скрипт ще започва с `#!/bin/bash`
- По конвенция, имената на скриптовете имат разширение `.sh`

# Shell скриптове

## Операционни системи, ФМИ, 2022/2023

## Shell Скриптове

- Както казахме в началото на курса, *автоматизацията и комбинирането на програми*, за да решим сложни задачи, са централни идеи в света на UNIX
- В тази връзка, говорихме за pipeline-ове
- Сега ще говорим за *Shell скриптове*, които много подпомагат автоматизацията

## Shell скриптове

- Скриптът е текстов файл, в който има изредени команди
- Започва със shebang, за да може да се изпълнява лесно
- Shell-ът изпълнява командите последователно

```
$ cat foo.sh  
#!/bin/bash
```

```
echo "These are the files in your home directory:  
find ~ -maxdepth 1 -mindepth 1 -printf "%f\n"
```

```
$ ./foo.sh  
Documents  
Pictures  
super_secret.txt  
Downloads  
...
```

## Коментари

- Всеки ред, който започва с # е коментар и се игнорира
- shebang-ът също е коментар - shell-ът го игнорира, когато изпълнява самият скрипт

## Изпълнение на shell скриптове

- Изпълнение в отделен процес, възползвайки се от shebang

```
$ /path/to/script
```

- Изпълнение в отделен процес, викайки интерпретатора директно

```
$ bash /path/to/script
```

- Изпълнение в процеса на текущия шел (по-късно ще видим това защо е полезно)

```
$ source /path/to/script
```

```
$ . /path/to/script
```

# Конфигуриране на shell-a

## Операционни системи, ФМИ, 2022/2023

## Смяна на shell-a

- Всеки потребител има асоцииран shell по подразбиране
  - тази информация се пази във файла /etc/passwd
- Винаги можем да изпълним друг shell
  - например, ако в момента се намираме в bash shell, можем да влезем в zsh shell, ако изпълним команда `zsh`
- Можем да сменим shell-а по подразбиране с команда `$ chsh`

bash

Нещата, които ще говорим от тук нататък, са специфични за  
shell-a bash

## Alias

С командата \$ alias <име>=<команда> можем да създадем кратко име за често използвани команди:

```
$ alias home_usage="df -h ~ | awk 'NR > 1 { print $5 }'"  
$ home_usage  
44%
```

С командата \$ unalias <име> можем да изтрием alias.

## Постоянна конфигурация

- Конфигурациите на shell-а, като например alias-ите, действат само в рамките на текущата сесия
- За да позволи конфигурации, които остават за постоянно, bash чете няколко скрипта при всяко стартиране
  - `~/.bash_profile`, `~/.profile` или `/etc/profile` - при login
  - `~/.bashrc` или `/etc/bashrc` - при всяко стартиране на shell
  - има и други
- Ако искаме някоя команда да се изпълнява при стартиране на всеки shell, можем да я сложим в скрипта `~/.bashrc`
- Ако искаме някоя команда да се изпълнява при всеки логин, може да я сложим в `~/.profile`

# Задълбаване в shell-a

## Операционни системи, ФМИ, 2022/2023

## Задълбаване в shell-а

- Сега ще говорим за някои вградени възможности на shell-а
- Макар и всяка от тях да може да се използва и в командния ред, ние ще ги използваме главно в контекста на shell скриптове

## Променливи

- Всеки работещ shell има таблица с променливи
- Стойността на всяка променлива е низ<sup>1</sup>
- С командата `$ set` можем да видим всички променливи

---

<sup>1</sup>bash is stringly-typed

## Променливи

- Можем да зададем стойност на променлива с <име на променлива>=<стойност>
- Можем да реферираме променлива с \${<име на променлива>}
  - работи по подобен начин като command substitution
  - можем да използваме и по-кратката конструкция \${име на променлива} (без къдрави скоби), но тя е по-опасна

```
$ name="Ivan Ivanov"
```

```
$ echo "my name is ${name}"  
my name is Ivan Ivanov
```

- Обърнете внимание на кавичките: почти винаги искаме да реферираме към променлива в контекст, заграден от двойни кавички (зашо?)

## Environment променливи

- Променливите са два вида - обикновени променливи и *environment* променливи
- Обикновените променливи са само за текущия shell
- Environment променливите се наследяват от всички процеси, деца на текущия shell
  - Всъщност, environment променливите са част от контекста на всеки процес, и се копират при `fork`

## Environment променливи

- Командата `$ env` извежда цялата таблица с environment променливи на текущия shell
- `$ export <име на променлива>=<стойност>` задава стойност на environment променлива
- Командата `$ env <var1>=<val1> <var2>=<val2> ... <команда>` може да се използва за да стартираме дадена команда с модифицирани environment променливи
  - Опцията `-i` игнорира всички съществуващи environment променливи и стартира командата само с тези, подадени директно на `env`
- Много настройки на shell-а и на различни програми се правят чрез environment променливи

## Променливата \${PATH}

- При извикване на команда, shell-ът търси изпълним файл със съответното име в директориите, изброени в променливата \${PATH}
- Директориите се разделят със символа :
- Ако добавите някаква директория в \${PATH}, всички изпълними файлове в нея ще могат да се изпълняват като команди, без задаване на пълен път

## Още няколко стандартни environment променливи

- \${SHELL} – пълен път до текущия shell
- \${USER} – потребителско име на текущия потребител
- \${HOME} – home директория на текущия потребител
- \${PS1} – низ, управляващ prompt-а
  - пробвайте да го смените
- \${PATH} – пътища за търсене на команди
- \${PWD} – текуща директория
- \${IFS} – Internal Field Separator
  - ще говорим за него след малко
- \${EDITOR} – текстов редактор по подразбиране

## Параметри на скриптовете

- Когато изпълняваме скрипт, можем да му подадем параметри, както на всяка друга команда: `$ /path/to/my_script.sh param_1 param_2 ...`
- Вътре в скрипта можем да достъпим параметрите от променливите  `${1}, ${2}, ${3} ...`
- Променливата  `${0}` съдържа името (пътя) на самия скрипт
- Променливата  `${#}` съдържа броя параметри

## Параметри на скриптовете

```
$ cat print_args.sh
#!/bin/bash
echo "num args: ${#}"
echo "arg 0: ${0}"
echo "arg 1: ${1}"
echo "arg 2: ${2}"
echo "arg 3: ${3}"

$ ./print_args foo bar
num args: 2
arg 0: ./print_args
arg 1: foo
arg 2: bar
```

## Параметри на скриптовете

- Променливата \${@} съдържа всички параметри
  - има специално поведение в кавички
  - "\${@}" се разпакетира до "\${1}" "\${2}" "\${3}" ...
- Променливата \${\*} също съдържа всички параметри
  - няма допълнително специално поведение
  - "\${\*}" се разпакетира до "\${1} \${2} \${3} ..."
- От двете, почти винаги искате да използвате \${\*} (защо?)

## Изписване на текст на stdout

- Командата `$ echo` изписва аргументите си
- `$ echo -n` не изписва символ за нов ред накрая
- `$ echo -e` приема интерпретира специални последователности
  - `\n` означава нов ред
  - `\t` означава таб
  - за други такива, вижте `echo(1)`

## Четене на данни от stdin

- С вградената конструкция \$ read <име на променлива> чете данни от stdin

```
$ cat foo.sh
#!/bin/bash

echo -n "What's your name? "
read name
echo "Hello, ${name}"
```

## Четене на данни от stdin

- С вградената конструкция `$ read -p <prompt>` показва някакъв prompt на потребителя преди да чака за вход:

```
$ cat foo.sh  
#!/bin/bash
```

```
read -p "What's your name?" name  
echo "Hello, ${name}"
```

## Четене на данни от stdin

- `read` може да прочете вход в няколко променливи:

```
$ cat add.sh  
#!/bin/bash
```

```
read a b  
echo "${a} plus ${b} is $(( a + b ))"
```

```
$ echo '42 26' | ./add.sh  
42 plus 26 is 69
```

## IFS + read

- Входът, който `read` и някои други конструкции обработват, се разделя на **думи**
  - границите между думите се определят от символите в съдържанието на променливата IFS
  - можем да я променим, ако искаме текстът да се разделя по друг символ
  - **интересно:** всъщност, IFS управлява и символите, по които се разделят аргументите на произволни команди
- Това е полезно, ако искаме да обработим текст, съдържащ полета, разделени с някакъв символ

## IFS + read

```
$ cat add.sh
#!/bin/bash

old_IFS="${IFS}"
IFS=':'
read a b
IFS="${old_IFS}"

echo "${a} plus ${b} is $(( a + b ))"

$ echo '42:26' | ./add.sh
42 plus 26 is 69
```

## read на повече от един ред

- read спира да чете, когато види символ за нов ред:

```
$ cat foo.sh  
#!/bin/bash
```

```
read a  
read b
```

```
echo "a: ${a}"  
echo "b: ${b}"
```

```
$ echo -e 'baba\ndyado' | ./foo.sh  
a: baba  
b: dyado
```

- Опцията `-d <символ>` на `read` го кара да спира да чете, когато види дадения символ (вместо нов ред)
  - това ще ни е полезно малко по-късно

## Аритметика с цели числа

- Вградената конструкция `$(( <израз> ))` пресмята произволен аритметичен израз и се заменя с резултата
- В израза могат да участват променливи

```
$ cat foo.sh  
#!/bin/bash
```

`a=42`

`b=26`

```
echo "result: $(( a + b + 1 ))"
```

```
$ ./foo.sh  
result: 69
```

## Аритметика с външни команди

- Можем да използваме външни команди за аритметика
- Например, команда `$ bc` чете израз от `stdin` и връща резултата на `stdout`. Също така, поддържа десетични дроби:

```
$ echo '3 + 5' | bc  
8
```

```
$ echo '3 + 2.5' | bc  
5.5
```

## Аритметика с външни команди

Пример с използване на bc заедно с променливи:

```
$ cat foo.sh
#!/bin/bash

a=42.5
b=26
result=$(echo "${a} + ${b}" | bc)

echo "result: ${result}"

$ ./foo.sh
69.5
```

## Аритметика с външни команди

- Има и други команди / езици за програмиране, които можете да използвате за аритметика
  - perl(1)
  - expr(1)
- Няма да говорим за тях сега

## Exit status

- При приключване, всеки процес предава на родителя си число, наречено *exit status*
- Съответно всяка команда в shell-а има exit status
- По конвенция, exit status, равен на 0, означава успех, а всякакъв друг означава неуспех
- Променливата \${?} съдържа exit status-а от предната команда

## Exit status

```
#!/bin/bash
```

```
grep -q '^ivan:' /etc/passwd  
echo "status: ${?}"
```

- grep има exit status 0, ако е намерил редове, отговарящи на условието
- Опцията -q на grep го кара да не извежда нищо

## Поведение при exit status

- По подразбиране, exit status-ът на поредица от команди, отделени с пайпове, е този на последната команда
- Можем да конфигурираме shell-а така, такава поредица да завършва неуспешно ако **която и да е** от командите е неуспешна
  - това се прави с `$ set -o pipefail`, например в началото на скрипта
- Можем да конфигурираме shell-а да прекратява скрипта в момента, в който някоя команда е неуспешна
  - това се прави с `$ set -e`
- Можем да конфигурираме shell-а да прекратява скрипта при рефериране на несъществуваща променлива
  - това се прави с `$ set -u`

## Поведение при exit status

- Можем да комбинираме тези неща:

```
#!/bin/bash
set -euo pipefail
```

...

- По този начин може да сме сигурни, че скриптовете ни ще се прекратяват при грешка
- Недостатъкът е, че ако искаме да проверим дали дадена команда е завършила с неуспех, тя задължително трябва да е част от логически израз или `if`
  - това ще видим как става след малко

## Команди във фонов режим (част 2)

- Вече знаем, че символът & в края на команда я пуска във фонов режим
- След пускане на команда във фонов режим, променливата \${!} съдържа PID-а на пуснатия процес
- Командата \$ wait <pid> изчаква процеса с даден PID да завърши, и приключва с неговия exit status

```
#!/bin/bash
cp /foo/large_file /bar/large_file &
cp_pid="${!}"

echo "started copying large file"
wait "${cp_pid}"
echo "finished copying large file with status ${?}"
```

## Изпълнение на команди последователно

- Команди, разделени със символ за нов ред, се изпълняват една след друга
  - това вече го знаем - така работят скриптовете
- Символът ; работи по същия начин като символа за нов ред - позволява да изпълним няколко команди на един ред

```
$ echo foo; echo bar
foo
bar
```

## Блокове

- Конструкцията { <команда 1>; <команда 2>; ... <команда n>; } се нарича **блок** и третира поредицата от команди като една команда

```
$ { echo 'The time is:'; date; } | tr a-z A-Z
THE TIME IS: SUN 12 MAR 17:42:08 EET 2023
```

## Subshells

- Конструкцията ( <команда 1>; <команда 2>; ... <команда n>; ) се нарича *subshell* и много прилича на блок
  - също както блоковете, е начин да третираме поредица от команди като една команда
  - разликата е, че командите се изпълняват в отделен shell от текущия

```
$ ( echo 'The time is:'; date; ) | tr a-z A-Z
THE TIME IS: SUN 12 MAR 17:42:08 EET 2023
```

## Subshell

- Тъй като subshell-ът е отделен shell, той наследява **копия** на променливите от оригиналния shell, които можем да променяме без тези промени да се отразят на оригиналната променлива
- Ето примера с IFS от по-рано, но вместо да запазваме старото състояние на IFS, използваме subshell:

```
$ cat add.sh
#!/bin/bash

(
    IFS=':'
    read a b
    echo "${a} plus ${b} is $(( a + b ))"
)

$ echo '42:26' | ./add.sh
42 plus 26 is 69
```

## Subshell

- Друг начин да направим subshell, е да използваме bash като външна команда:

```
$ bash -c <низ, съдържащ команди>
```

- Това е удобно, когато искаме да подадем няколко команди на команда, приемаща команда като аргумент:

```
find /etc -type f -exec bash -c \
'echo "reading {}"; cat {}' \;
```

## Process substitution

- От по-рано знаем конструкцията *command substitution* – `$(<команда>)`, която се заменя с низ, който е съдържанието на `stdout` на командата
- Сега ще покажем още две конструкции, наречени *process substitution*
  - `>(<команда>)` се заменя с низ, който е **име на виртуален файл**, представляващ тръба, закачена към `stdin` на командата
  - `<(<команда>)` се заменя с низ, който е **име на виртуален файл**, представляващ тръба, закачена към `stdout` на командата

## Process substitution

Така можем да вземем имената на потребителите, които **не са** логнати в момента:

```
$ comm -2 -3 <(cat /etc/passwd | cut -d : -f 1 | sort) \
      <(who | cut -d ' ' -f 1 | sort)
pesho
gosho
```

Въобще, резултата от тази конструкция е просто име на файл:

```
$ echo <(whoami)
/dev/fd/63
```

## Process substitution, pipes and redirection

- С process substitution може да емулираме оператора pipe:

```
$ whoami | tr a-z A-Z  
IVAN
```

```
$ tr a-z A-Z < <(whoami)  
IVAN
```

- Каква е (функционалната) разлика?

- при обикновени команди - никаква!

## Process substitution, pipes and redirection

```
$ whoami | read name  
$ echo "name: ${name}"  
name:  
  
$ read name < <(whoami)  
$ echo "name: ${name}"  
name: ivan
```

- Каква е (функционалната) разлика?
  - при обикновени команди - никаква!
  - при вградени конструкции като `read` има значение, защото дясната страна на оператора `|` се изпълнява в subshell

“НЕ”

- ! <команда> изпълнява командата и
  - има status 0, ако командала е имала не-нулев статус
  - има status 1, ако командала е имала нулев статус

## “И” и “ИЛИ”

- <команда 1> || <команда 2> има семантика на “ИЛИ”
  - ако команда 1 е била успешна, целият израз е успешен и команда 2 **не се** изпълнява
  - ако команда 1 е била неуспешна, се изпълнява команда 2 и се гледа нейният статус
- <команда 1> && <команда 2> има семантика на “И”
  - ако команда 1 е била успешна, целият израз е успешен и команда 2 **не се** изпълнява
  - ако команда 1 е била неуспешна, се изпълнява команда 2 и се гледа нейният статус
- Ляво-асоциативни са

## “И” и “ИЛИ”

Поради лявата асоциативност, тези две команди правят едно и също:

```
$ grep '^ivan:' /etc/passwd && echo "Ivan is here" \
               || echo "Ivan is not here"
Ivan is here
```

```
$ { grep '^ivan:' /etc/passwd && echo "Ivan is here"; } \
  || echo "Ivan is not here"
Ivan is here
```

- Конструкцията `[[ <израз> ]]` пресмята логически израз и завършва със status 0, ако резултатът е истина, или със status 1, ако е лъжа
- Исторически е била имплементирана като команда `$ test`, а по-късно като конструкция `[ <израз> ]`
  - може да видите тези алтернативни конструкции в някои скриптове
  - ние ще използваме `[[ <израз> ]]`

- Сравнение на числа (-eq, -gt, -lt, -le, -ge)

```
$ [[ 42 -eq 26 ]] && echo yes || echo no  
no
```

```
$ [[ 42 -gt 26 ]] && echo yes || echo no  
yes
```

- Лексикографско сравнение на низове (==, <, >)

```
$ [[ "$(whoami)" == "ivan" ]] && echo yes || echo no  
yes
```

```
$ [[ "foo" > "bar" ]] && echo yes || echo no  
yes
```

- Операции за низове
  - [[ -n <низ> ]] проверява дали низ е не-празен
  - [[ -z <низ> ]] проверява дали низ е празен
- Операции за файлове
  - [[ -f <име> ]] проверява дали обект с такова име съществува и е файл
  - [[ -d <име> ]] проверява дали обект с такова име съществува и е директория
  - [[ -r <име> ]] проверява дали можем да четем файл
  - [[ -w <име> ]] проверява дали можем да пишем файл

## Test

```
#!/bin/bash

[[ "$(whoami)" != "root" ]] && {
    echo "This script must only be run as root!" >&2
    exit 1
}
```

## If

- Конструкцията `if` във `bash` изглежда така:

```
if <команда>; then
    <блок от команди>
else
    <блок от команди>
fi
```

- Можем и да изпуснем `else`-блока:

```
if <команда>; then
    <блок от команди>
fi
```

- Exit status-ът на команда-условие определя дали да се изпълни `then`-блока или `else`-блока

## Блокове и ;

Да припомним, че символът за нов ред и ; са взаимно заменяеми - следните 3 конструкции са еднакви:

```
if grep -q '^ivan:' /etc/passwd  
then  
    echo yes  
else  
    echo no  
fi
```

```
if grep -q '^ivan:' /etc/passwd; then  
    echo yes  
else  
    echo no  
fi
```

```
if grep -q '^ivan:'; then echo yes; else echo no; fi
```

## If + test

Често се случва да комбинираме конструкциите If и Test:

```
#!/bin/bash

read -p "Are you sure you want to continue? (yes/no)" answer

if [[ "${answer}" != "yes" ]]; then
    echo "Coward!"
else
    rm -rf /*
fi
```

## Case

Конструкцията case в bash изглежда така:

```
case <низ> in
    <glob1>
        <блок>
        ;;
    <glob2>
        <блок>
        ;;
    ...
    <globN>
        <блок>
        ;;
esac
```

## Case

```
case "${1}" in
    start)
        echo "starting service"
        ;;
    stop)
        echo "stopping service"
        ;;
    restart)
        echo "restarting service"
        ;;
*)
    echo "Usage: ${0} (start|stop|restart)"
    exit 1
    ;;
esac
```

## For

Конструкцията **for** в bash изглежда така:

```
for <име на променлива> in <аргументи>; do  
    <блок>  
done
```

For

```
$ cat foo.sh
#!/bin/bash

for name in 'Pesho' 'Gosho' 'Penka'; do
    echo "${name} is awesome!"
done
```

```
$ ./foo.sh
Pesho is awesome!
Gosho is awesome!
Penka is awesome!
```

## For: аргументи

Дойде време за пример с \${@}:

```
$ cat foo.sh
#!/bin/bash

for name in "${@}"; do
    echo "${name} is awesome!"
done
```

```
$ ./foo.sh 'Pesho Peshkov' 'Gosho Goshkov' 'Penka Penkova'
Pesho Peshkov is awesome!
Gosho Goshkov is awesome!
Penka Penkova is awesome!
```

## For: аргументи

Кавичките имат значение:

```
$ cat foo.sh
#!/bin/bash

for name in ${@}; do
    echo "${name} is awesome!"
done
```

```
$ ./foo.sh 'Pesho Peshkov' 'Gosho Goshkov' 'Penka Penkova'
Pesho is awesome!
Peshkov is awesome!
Gosho is awesome!
Goshkov is awesome!
Penka is awesome!
Penkova is awesome!
```

## For: аргументи

Както казахме, поведението в кавички на \${\*} е различно от това на \${@}:

```
$ cat foo.sh
#!/bin/bash

for name in "${*}"; do
    echo "${name} is awesome!"
done
```

```
$ ./foo.sh 'Pesho Peshkov' 'Gosho Goshkov' 'Penka Penkova'
Pesho Peshkov Gosho Goshkov Penka Penkova is awesome!
```

## Разделяне на думи при for

- Аргументите на for се разделят по същия начин както при read
- Този скрипт ще обходи полетата на реда от /etc/passwd, относящ се за потребителя ivan, последователно:

```
#!/bin/bash

old_IFS="${IFS}"
IFS=':'
for field in $(grep '^ivan:' /etc/passwd); do
    echo "field: ${field}"
done
IFS="${old_IFS}"
```

## seq

Командата `$ seq <начало> <край>` изписва всички числа в дадения интервал:

```
$ seq 1 5
1
2
3
4
5
```

## seq

\$ seq се комбинира добре с for:

```
$ cat foo.sh
#!/bin/bash

for i in $(seq 1 5); do
    echo "i: ${i}"
done
```

```
$ ./foo.sh
i: 1
i: 2
i: 3
i: 4
i: 5
```

## While

Конструкцията `while` в `bash` изглежда така:

```
while <команда>; do
    <блок>
done
```

Блокът се изпълнява множество пъти, докато командалата завърши с неуспех

## While

```
#!/bin/bash
while ps -e -o cmd= | grep -q firefox; do
    sleep 1
done
echo "firefox has died"
```

## While + read

Можем да се възползваме от факта, че `read` чете по един ред наведнъж, и да пренасочим изхода на някаква команда в целия блок на `while`:

```
#!/bin/bash
while read filename; do
    echo "found file: ${filename}"
done < <(find ~ -type f)
```

Това също работи, но има уловка:

```
#!/bin/bash
find ~ -type f | while read filename; do
    echo "found file: ${filename}"
done
```

## While + read (pedantic)

Може да имаме име на файл, съдържащо символ за нов ред. За да се предпазим от тези ситуации, когато обхождаме файлове, можем да използваме тази конструкция:

```
#!/bin/bash
while read -d $'\0' filename; do
    echo "found file: ${filename}"
done < <(find ~ -type f -print0)
```

- Опцията `-print0` на `find` го кара да изписва имената разделени с **нулев символ** вместо символ за нов ред
- Опцията `-d` на `read` му казва до какъв символ да спира да чете
- Конструкцията `$ '<низ>'` интерпретира специални последователности в низа (в случая заменя `\0` с нулев символ)

## `break, continue`

- Командите `$ break` и `$ continue` могат да се използват в тялото на цикъл
  - правят това, което предполагате

## Функции

- Можем да обявяваме *функции* в bash със следната конструкция:

```
function <име на функция> {  
    <тело>  
}
```

- Телото на функцията може да вижда параметрите, с които е извикана, чрез променливите \${@}, \${\*}, \${#}, \${1}, \${2}, ...
- Можем да създаваме локални променливи в телото на функция така:

```
local <име>=<стойност>
```

## Функции

```
$ cat foo.sh
#!/bin/bash

function greet {
    local name="${1}"
    echo "Hello, ${name}"
}

greet "$(whoami)"

$ ./foo.sh
Hello, ivan
```

# Полезности

## Операционни системи, ФМИ, 2022/2023

## xargs

- Командата `xargs` чете записи от `stdin` и извиква някаква команда, подавайки ѝ ги като аргументи
- `$ who | cut -d ' ' -f 1 | sort | uniq | xargs touch -` създава файлове в текущата директория с имената на потребителите, логнати в момента
- `$ ps -u ivan -o pid= | xargs kill -TERM` - убива всички процеси на потребителя `ivan` със сигнал `TERM`
- Вижте `xargs(1)` за повече информация - има много възможности!

## Именовани тръби

- Командата `mkfifo <път>` създава виртуален файл, наречен *именована тръба*

```
$ mkfifo /tmp/my_pipe
$ echo "my name is $(whoami)" > /tmp/my_pipe &
$ cat /tmp/my_pipe
my name is ivan
```

## Временни файлове

- Понякога, в рамките на скрипт, ни се налага да използваме временен файл
- Командата `$ mktemp` създава временен файл и изписва името му
  - `$ mktemp -d` създава директория
- Добре е да създаваме временните файлове само с `$ mktemp`, и да ги изтриваме след ползването им

## Още команди

- `cmp(1)` - проверява дали два файла са еднакви
- `sha256sum(1), md5sum(1)` - алгоритми за пресмятане на хеш
- `diff(1)` - сравнява текстови файлове ред по ред
- `printf(1)` - форматиран изход
- `pwgen(1)` – генерира пароли

# Програми на C, използващи системни извиквания

## Операционни системи, ФМИ, 2022/2023

## Увод

В този раздел ще пишем на С.

Неща, които трябва да знаете, преди да започнем:

- Синтаксис на С
- Как компилираме С код
- Как пускаме програми на С

Езикът С и системни извиквания  
Операционни системи, ФМИ, 2022/2023

## Системни извиквания

- Дотук знаем, че операционната система имплементира комуникация между процеси и връзка на процесите с външния свят
- Връзката между един процес и (ядрото на) операционната система се извършва чрез операции, наречени *системни извиквания* (system calls или syscalls)
- В този раздел ще се научим да пишем програми, които директно използват системните извиквания на операционната система
  - Това се нарича “*системно програмиране*”

## Как работят системните извиквания

- От гледна точка на програмиста, системните извиквания са “*просто*” библиотечни функции, които може да извика
- Специалното на тези функции е, че вместо да изпълняват код като част от програмата, *казват* на ядрото да изпълни съответната операция и чакат резултат
- Програмата заспива докато ядрото не стане готово с изпълнението на системното извикване, и когато се събуди, получава резултат
- Тъй като тези функции “обивват” системните извиквания, ще ги наричаме *syscall wrappers*

## Как работят системните извиквания

- Най-често използваната С библиотека, имплементираща syscall wrappers, е **glibc**
  - glibc е “стандартна библиотека” за С, която освен syscall wrappers имплементира и всички функции от С стандарта
  - Има и други такива библиотеки, напр. musl
- Съответните библиотеки съдържат хедъри със стандартизиирани имена (`fcntl.h`, `unistd.h`, ...), в които са дефинирани syscall wrapper-ите

## Какво реално прави един syscall wrapper?

- Копира аргументите на системното извикване и неговия номер в специфични процесорни регистри
- Изпълнява процесорна инструкция, която предизвика хардуерно прекъсване
  - При стартиране на операционната система, тя е конфигурирала процесора при такова прекъсване да скочи на специфично място в кода на ядрото, където ще се прочетат въпросните аргументи и номер на системно извикване, и то ще се обработи
  - След това ядрото записва резултата от системното извикване в специфичен процесорен регистър и скача обратно в кода на syscall wrapper-а
- Вади резултата от въпросния процесорен регистър и го връща

## Какво реално прави един syscall wrapper?

- Когато имам време, на този слайд ще сложа картинка

## Защо C?

- Езикът, на който е написан Linux (и други UNIX-и) е C
- По тази причина, системните извиквания използват подреждане и формат на данните, съобразени с ABI<sup>1</sup> на C
- Затова повечето библиотеки, имплементиращи syscall wrappers, са написани на C
  - Нищо не ни пречи да използваме системни извиквания директно и от друг език - пример за език, който има собствена имплементация на syscall wrappers е Go
  - На практика почти всички останали езици извикват C код през FFI, който от своя страна вика syscall wrappers от библиотека като glibc, вместо да ги имплементират сами

---

<sup>1</sup>ABI: Application Binary Interface

## Пример за системни извиквания на С

Системните извиквания `getuid` и `geteuid` връщат реалното и ефективното UID на процеса, изпълняващ програмата.

```
#include <unistd.h>      // getuid, geteuid
#include <sys/types.h>    // uid_t
#include <stdio.h>         // printf

int main(void)
{
    uid_t me = getuid();
    uid_t pretending = geteuid();
    printf("uid: %d euid: %d\n", me, pretending);
    return 0;                // exit status 0
}
```

## [man syscall](#)

В този раздел ще са ни полезни следните секции от man страници:

- Секция 2: системни извиквания
  - `getuid(2)`
  - `geteuid(2)`
- Секция 3: библиотечни функции в C
  - `printf(3)`

## Exit status

Exit status-ът на една програма е стойността, върната от функцията `main()`:

```
int main(void)
{
    return 42;           // exit status 42
}
```

## Exit status

Системното извикване `_exit(2)` прекратява изпълнението на процеса, независимо от текущата функция:

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

void foo(void) {
    _exit(42);          // exit status 42
}

int main(void)
{
    foo();
}
```

## Exit status

Библиотечната функция `exit(3)` вътрешно вика `_exit(2)`, но финализира и някои други неща преди това.

Използвайте нея, когато искате да прекратите програмата.

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

void foo(void) {
    exit(42);           // exit status 42
}

int main(void)
{
    foo();
}
```

# Обработване на грешки при системни извиквания

Операционни системи, ФМИ, 2022/2023

## Резултат от системни извиквания

- По конвенция, повечето системни извиквания връщат резултат от числов тип, който е отрицателно число, ако извикването е било неуспешно
- За пример ще използваме `open(2)` - системно извикване, отварящо файл

## Резултат от системни извиквания

```
int open(const char *pathname, int flags)
```

- първият аргумент е път до файл
- вторият аргумент е множество от опции, задаващи режима на отваряне на файл
- резултатът при успех е положително число - номер на *файлов дескриптор* - след малко ще говорим за него
- засега важното е, че резултатът от `open()` е -1, ако отварянето на файл е било неуспешно

## Грешки при системни извиквания

```
#include <fcntl.h> // open
#include <stdio.h> // printf
#include <stdlib.h> // exit

int main() {
    int result = open("/tmp/some_file", O_RDONLY);
    if (result < 0) {
        printf("opening /tmp/some_file failed\n");
        exit(1);
    }

    printf("opened /tmp/some_file successfully\n");
}
```

## `errno`

- Когато някое системно извикване е неуспешно, ни се иска начин да разберем какво не е било наред
- Затова, при неуспех системните извиквания записват число (код на грешка) в **глобалната променлива `errno`**
  - Вижте `errno(3)` за повече информация

## errno

Можем да използваме `errno`, за да разберем каква е била грешката:

```
#include <fcntl.h> // open
#include <stdio.h> // printf
#include <errno.h> // errno
#include <stdlib.h> // exit

int main() {
    int result = open("/tmp/some_file", O_RDONLY);
    if (result < 0) {
        switch (errno) {
            case 2: printf("no such file\n"); break;
            case 13: printf("permission denied\n"); break;
            // ...
        }
        exit(1);
    }

    printf("opened /tmp/some_file successfully\n");
}
```

errno

По-добре е да ползваме константите, дефинирани в `errno.h`, вместо чисти номера на грешки:

```
#include <fcntl.h> // open
#include <stdio.h> // printf
#include <errno.h> // errno
#include <stdlib.h> // exit

int main() {
    int result = open("/tmp/some_file", O_RDONLY);
    if (result < 0) {
        switch (errno) {
            case ENOENT: printf("no such file\n"); break;
            case EACCES: printf("permission denied\n"); break;
            // ...
        }
        exit(1);
    }

    printf("opened /tmp/some_file successfully\n");
}
```

## err.h

Най-добрият вариант е да използвате функцията `err()`, която изписва форматирано съобщение за грешка (вътрешно гледа променливата `errno`).

Първият аргумент на `err()` е exit status, с който да прекрати програмата.

```
#include <fcntl.h> // open
#include <stdio.h> // printf
#include <err.h> // err

int main() {
    int result = open("/tmp/some_file", O_RDONLY);
    if (result < 0) {
        err(1, "could not open file");
    }

    printf("opened /tmp/some_file successfully\n");
}
```

## err.h

```
#include <fcntl.h> // open
#include <stdio.h> // printf
#include <err.h> // err

int main() {
    int result = open("/tmp/some_file", O_RDONLY);
    if (result < 0) {
        err(1, "could not open file");
    }

    printf("opened /tmp/some_file successfully\n");
}
```

Например при грешка ENOENT, тази програма ще изведе съобщение, изглеждащо така:

could not open file: No such file or directory

## err.h

Вторият и следващите аргументи на `err()` задават форматин низ (както при `printf()`):

```
#include <fcntl.h> // open
#include <stdio.h> // printf
#include <err.h>   // err

int main() {
    const char filename[] = "/tmp/some_file";
    int result = open(filename, O_RDONLY);
    if (result < 0) {
        err(1, "could not open file %s", filename);
    }

    printf("opened %s successfully\n", filename);
}
```

Например при грешка ENOENT, тази програма ще изведе съобщение, изглеждащо така:

```
could not open file /tmp/some_file: No such file or directory
```

Всъщност, `err.h` задава 4 полезни функции:

- `void err(int eval, const char* fmt, ...)`
  - изписва съобщението, което сме подали
  - изписва грешката от `errno`
  - прекратява програмата с подадения статус
- `void errx(int eval, const char* fmt, ...)`
  - изписва съобщението, което сме подали
  - **не** изписва грешката от `errno`
  - прекратява програмата с подадения статус
- `void warn(const char* fmt, ...)`
  - изписва съобщението, което сме подали
  - изписва грешката от `errno`
  - **не** прекратява програмата
- `void warnx(const char* fmt, ...)`
  - изписва съобщението, което сме подали
  - **не** изписва грешката от `errno`
  - **не** прекратява програмата

Ще срещнете и четирите вида ситуации, в които е подходяща съответната функция. Ползвайте ги!

# Файлови дескриптори

## Операционни системи, ФМИ, 2022/2023

## Файлови дескриптори

- Споменахме, че системното извикване `open()` връща число, което наричаме *номер на файлов дескриптор*
- При отваряне на файл, ядрото създава системна структура, наречена *файлов дескриптор*, която съдържа:
  - Указател към самия файл
  - Текуща позиция (индекс на байт) във файла
  - И други
- За всеки процес ядрото алокира масив от файлови дескриптори: номерът на файлов дескриптор, върнат от `open()`, е индекс в този масив.

## Опции на open()

- open() може да приема 2 или 3 аргумента:

```
int open(const char *pathname, int flags);
```

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

- Аргументите на open() са:

- Път до файл
- Множество от опции
- Права за достъп (mode) - в осмична бройна система

## Опции на open()

Вторият аргумент на open( ) е битова маска от опции.

- Комбинираме опциите с “побитово ИЛИ”
- Някои опции, които ще ни трябват, са:
  - O\_WRONLY, O\_RDONLY - отваряне за писане или за четене
  - O\_RDWR - отваряне за четене и писане едновременно
  - O\_CREAT - ако файлът не съществува, го създава преди да го отвори
  - O\_TRUNC - ако файлът съществува, зачиства съдържанието му преди да го отвори
  - O\_APPEND - ако файлът съществува, началната позиция е в края му вместо в началото

## Опции на open()

Третият аргумент на open() задава права за достъп на файла, ако го създаваме сега

- Има смисъл само с O\_CREAT
- Единият вариант е да зададем правата директно като число

```
int fd = open(  
    "/tmp/some_file",  
    O_WRONLY|O_CREAT|O_TRUNC,  
    0644  
);
```

- Другият вариант е да ползваме побитови константи, дефинирани в стандартната библиотека

```
int fd = open(  
    "/tmp/some_file",  
    O_WRONLY|O_CREAT|O_TRUNC,  
    S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH  
);
```

- За повече информация, вижте open(2)

## Опции на open(): примери

- Отваряне на файл за четене

```
int fd = open("/tmp/some_file", O_RDONLY);
```

- Отваряне на файл за писане, създавайки го с права 644 ако не съществува и зачиствайки го, ако съществува

```
int fd = open(  
    "/tmp/some_file",  
    O_WRONLY|O_CREAT|O_TRUNC,  
    0644  
) ;
```

- Отваряне на файл за писане, създавайки го с права 644 ако не съществува и заставайки накрая му, ако съществува

```
int fd = open(  
    "/tmp/some_file",  
    O_WRONLY|O_CREAT|O_APPEND,  
    0644  
) ;
```

## Затваряне на отворен файл

- Когато един процес умре, всички негови файлови дескриптори се затварят автоматично
- Можем ръчно да затворим файлов дескриптор, използвайки системното извикване `close(2)`

```
#include <fcntl.h> // open, close
#include <err.h>   // err

int main() {
    const char filename[] = "/tmp/some_file";
    int fd = open(filename, O_RDONLY);
    if (fd < 0) {
        err(1, "could not open file %s", filename);
    }

    // ...

    if (close(fd) < 0) {
        err(1, "could not close file %s", filename);
    }
}
```

## Затваряне на отворен файл

- Добре е отворените файлове да се затварят веднага, щом сме приключили да работим с тях

## Четене от файл

- Можем да четем от отворен файл със системното извикване `read(2)`:

```
char buf[20];
int num_bytes = read(fd, buf, 20);
if (num_bytes < 0) {
    err(1, "could not read data");
}
```

- Аргументите на `read()` са:
  - Номер на файлов дескриптор, от който четем
  - Указател към памет, в която искаме да се запишат прочетените данни
  - Максимална дължина на четене
- Резултатът от `read()` е броят реално прочетени байтове

## Четене от файл

- При успешно изпълнение на `read()`, текущата позиция на файловия дескриптор се придвижва напред с броя успешно прочетени байтове
- Това означава, че всяко следващо викане на `read()` чете нови данни от файла
- Ако `read()` прочете 0 байта, това означава, че сме стигнали края на файла

## Четене от файл

```
int fd = open("/tmp/some_file", O_RDONLY);
if (fd < 0) {
    err(1, "could not open file");
}

char buf[20];
int num_bytes = read(fd, buf, 20);
while (num_bytes > 0) {
    // < do something with the data in buf >
    num_bytes = read(fd, buf, 20);
}
if (num_bytes < 0) {
    err(1, "could not read from file");
}

if (close(fd) < 0) {
    err(1, "could not close file");
}
```

## Четене от файл

```
int fd = open("/tmp/some_file", O_RDONLY);
if (fd < 0) {
    err(1, "could not open file");
}

char buf[4096];
int num_bytes;
while ((num_bytes = read(fd, buf, sizeof(buf))) > 0) {
    // < do something with the data in buf >
}
if (num_bytes < 0) {
    err(1, "could not read from file");
}

if (close(fd) < 0) {
    err(1, "could not close file");
}
```

## Четене от файл

```
int fd = open("/tmp/some_file", O_RDONLY);
if (fd < 0) {
    err(1, "could not open file");
}

char c;
int num_bytes;
while ((num_bytes = read(fd, &c, 1)) > 0) {
    // < do something with the character in c >
}
if (num_bytes < 0) {
    err(1, "could not read from file");
}

if (close(fd) < 0) {
    err(1, "could not close file");
}
```

## Писане във файл

Писането във файл е аналогично на четенето: чрез системното извикване `write(2)`:

```
char buf[] = "Hello world!\n";
int num_bytes = write(fd, buf, 13);
if (num_bytes < 0) {
    err(1, "could not write data");
}
if (num_bytes != 13) {
    errx(1, "could not write data all at once");
}
```

- Аргументите на `write()` са:
  - Номер на файлов дескриптор, в който пишем
  - Указател към памет, от която да се прочетат данни
  - Брой байтове (размер на данните), които искаме да запишем
- Резултатът от `write()` е броят реално записани байтове

## Записване на текстови низове от паметта в текстов файл

- При записване на текстови низове, трябва да внимаваме да не запишем терминрация нулев символ във файла
  - Функцията `strlen(3)` е полезна: връща размера на низа, без да включва терминиращата нула

```
char buf[] = "Hello world!\n";
int num_bytes = write(fd, buf, strlen(buf));
if (num_bytes < 0) {
    err(1, "could not write data");
}
if (num_bytes != 13) {
    errx(1, "could not write data all at once");
}
```

## Файлови дескриптори на стандартни потоци

- Стандартните потоци `stdin`, `stdout` и `stderr` по подразбиране съществуват при създаване на процес
  - Файлов дескриптор 0 е `stdin`
  - Файлов дескриптор 1 е `stdout`
  - Файлов дескриптор 2 е `stderr`

## Файлови дескриптори на стандартни потоци

```
char name_buf[512];
const char prompt[]    = "What's your name? ";
const char hello[]     = "Hello, ";
const char end[]       = "!\\n";

int write_result = write(1, prompt, strlen(prompt));
if (write_result < 0) { err(1, "could not write prompt"); }

int name_len = read(0, name_buf, strlen(name_buf));
if (name_len < 0) { err(1, "could not read name"); }

write_result = write(1, hello, strlen(hello));
if (write_result < 0) { err(1, "could not write hello"); }

write_result = write(1, name_buf, strlen(name_buf));
if (write_result < 0) { err(1, "could not write name"); }

write_result = write(1, end, strlen(end));
if (write_result < 0) { err(1, "could not write end"); }
```

## Преместване на текущата позиция във файл

- Досега видяхме, че `read()` и `write()` местят текущата позиция *напред*
- Със системното извикване `lseek(2)` можем да преместим текущата позиция на *произволно място* във файла

```
off_t lseek(int fd, off_t offset, int whence);
```

## Преместване на текущата позиция във файл

```
off_t lseek(int fd, off_t offset, int whence);
```

- Аргументите на `lseek()` са:
  - Файлов дескриптор
  - Отместване
  - Интерпретация на отместването
- Резултатът от `lseek()` е новата абсолютна позиция
- Възможните интерпретации на отместването (`whence`) са:
  - SEEK\_SET: абсолютно отместване
  - SEEK\_CUR: относително отместване спрямо текущата позиция
  - SEEK\_END: относително отместване спрямо края на файла

## Преместване на текущата позиция във файл

- Скачане в началото на файла:

```
int new_pos = lseek(fd, 0, SEEK_SET);
if (new_pos < 0) {
    err(1, "could not go to start of file");
}
```

- Скачане на позиция 42 във файла:

```
int new_pos = lseek(fd, 42, SEEK_SET);
if (new_pos < 0) {
    err(1, "could not go byte 42");
}
```

- Скачане с 5 байта назад:

```
int new_pos = lseek(fd, -5, SEEK_CUR);
if (new_pos < 0) {
    err(1, "could not jump 5 bytes backwards");
}
```

## Преместване на текущата позиция във файл

- Скачане 2 байта преди края на файл:

```
int new_pos = lseek(fd, -2, SEEK_END);
if (new_pos < 0) {
    err(1, "could not jump to 2 bytes before end");
}
```

- Скачане 42 байта след края на файл:

- Това работи само ако можем да пишем във файла
- Файлът пораства с толкова, колкото е нужно

```
int new_pos = lseek(fd, 42, SEEK_END);
if (new_pos < 0) {
    err(1, "could not jump to 42 bytes after end");
}
```

# Четене и писане на двоични данни от паметта във файлове

Операционни системи, ФМИ, 2022/2023

## Форматиран и неформатиран вход/изход

- Мислено можем да разделим подходите за вход/изход на две категории:
  - Форматиран вход/изход
  - Неформатиран вход/изход

## Неформатиран вход/изход

- Когато говорим за *неформатиран* вход/изход, имаме предвид, че програмата чете и пише *данни* във формат, който не може да се интерпретира като текст
- Числата най-често ги представяме по същия начин, както са представени в паметта
- Системните извиквания `read()` и `write()` могат да се използват за неформатиран вход/изход на данни в паметта, без промяна на тяхната структура

## Неформатиран вход/изход

```
void write_number(int fd, uint16_t num) {
    write(fd, &num, sizeof(num));
}

uint16_t read_number(int fd) {
    uint16_t num;
    read(fd, &num, sizeof(num));
    return num;
}
```

## Форматиран вход/изход

- Когато говорим за *форматиран* вход/изход, имаме предвид, че програмата чете и пише *текст*, предназначен за четене от хора
- Нищо не пречи текстът да е и машинно четим
  - това го обсъдихме по-нашироко в темата “Данни във файлове”
- Числата са *форматирани* като последователности от цифри (текст)

## Форматиран изход на числа

```
int n_digits(uint16_t num) {
    if (num == 0) { return 1; }
    int result = 0;
    for (; num != 0; num /= 10) { result++; }
    return result;
}
void num_to_text(uint16_t num, char* buf) {
    buf[n_digits(num)] = '\0';
    for (int i = n_digits(num) - 1; i >= 0; i--) {
        buf[i] = '0' + (num % 10);
        num /= 10;
    }
}
void print_number(int fd, uint16_t num) {
    char num_text[6];
    num_to_text(num, num_text);
    write(fd, num_text, n_digits(num));
}
```

## Форматиран изход на числа

- Можем да използваме вградената функция `snprintf(3)` за да форматираме числа като текст:

```
void print_number(int fd, uint16_t num) {
    char num_text[6];
    snprintf(num_text, sizeof(num_text), "%d", num);
    write(fd, num_text, n_digits(num));
}
```

## Работим само с файлови дескриптори

- В стандартната библиотека на С има абстракция за работа с файлове, наречена FILE\*, която обвива системните извиквания за работа с входно-изходни операции във функции от по-високо ниво
- Повечето такива функции са в `<stdio.h>`
- **Ние няма да ползваме тези функции**, а ще работим директно с файловите дескриптори
- Единствените функции от `<stdio.h>`, които ще си позволим, са:
  - `dprintf()`, за форматиран изход (само в случаите, в които искаме да форматираме число)
  - `snprintf()`, за генериране на форматиран низ

# Информация за файловете чрез stat

## Операционни системи, ФМИ, 2022/2023

## Информация за файловете чрез stat

- Системното извикване `stat(2)` дава достъп до метаданните на файла (командата `stat(1)` използва това системно извикване)
- Първият аргумент е път до файл, а вторият е указател към структура от тип `struct stat`, която е дефинирана в стандартната библиотека.

## struct stat - stat(3type)

```
struct stat {  
    dev_t      st_dev;      // ID of device containing file  
    ino_t      st_ino;      // Inode number  
    mode_t     st_mode;     // File type and mode  
    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 ID (if special file)  
    off_t      st_size;     // Total size, in bytes  
    blksize_t  st_blksize;  // Block size for filesystem I/O  
    blkcnt_t   st_blocks;   // Number of 512 B blocks allocated  
  
    struct timespec st_atim; // Time of last access  
    struct timespec st_mtim; // Time of last modification  
    struct timespec st_ctim; // Time of last status change  
};
```

stat

```
struct stat info;
int result = stat("/tmp/foo.txt", &info);
if (result < 0) { err(1, "could not stat /tmp/foo.txt") };

dprintf(1, "owner UID: %d\n", info.st_uid);
dprintf(1, "owner GID: %d\n", info.st_gid);
dprintf(1, "size: %d bytes\n", info.st_size);
```

## stat

- Алтернативният вариант `fstat()` приема файлов дескриптор като първи аргумент вместо път
- Може да го ползвате за вече отворени файлове

# Изпълняване на програми с ехес

## Операционни системи, ФМИ, 2022/2023

## Изпълняване на програми с exec

- Фамилията от системни извиквания exec(3) се използва, за да изпълним външна програма в текущия процес
  - Различни варианти на извикване - execl(), execlp(), execvp(), execle(), execve()...
- При успешно изпълнение на exec(), програмата на текущия процес се **заменя** с дадената

## Изпълняване на програми с exec

```
int main(void) {
    int result = execl(
        "/usr/bin/cat",           // executable
        "cat", "/etc/issue",      // arguments
        (char*)NULL               // sentinel
    );
    if (result < 0) {
        err(1, "could not exec");
    }
    dprintf(1, "you will never read this\n");
}
```

## exec\*p - търсене в \$PATH

Вариантите exec\*p използват environment променливата PATH за да търсят изпълнимия файл:

```
int main(void) {
    int result = execlp(
        "cat",                  // executable
        "cat", "/etc/issue",    // arguments
        (char*)NULL             // sentinel
    );

    if (result < 0) {
        err(1, "could not exec");
    }

    dprintf(1, "you will never read this\n");
}
```

## execv\* - масив от аргументи

Вариантите execv\* приемат масив от аргументи:

```
int main(int argc, char* argv[]) {
    if (argc > 9) {
        errx(1, "cannot work with more than 8 arguments");
    }

    char* command_args[10];
    char cat[] = "cat";
    command_args[0] = &cat;

    for (int i = 1; i < argc; i++) {
        command_args[i] = argv[i];
    }
    command_args[argc] = NULL;

    execvp("cat", command_args);
    err(1, "could not exec cat");
}
```

# Създаване на процеси

## Операционни системи, ФМИ, 2022/2023

## Създаване на процеси

- В UNIX света създаването на процеси става чрез системното извикване `fork(2)`.
- При извикване на `fork()`, текущият процес се клонира на *родител и дете*
  - Семантично, цялата памет на процеса се **копира**
  - Реално копието се извършва чрез *copy-on-write* (CoW)
- Родителят и детето използват **отделни** региони във физическата памет и не могат да достъпват паметта по между си
- Изпълнението на програмата в процеса-детете продължава от мястото, където е извикан `fork()`
- Отделните процеси работят конкурентно и не се изчакват

## Създаване на процеси

- Стойността, върната от `fork()`, е различна при родителя и детето:
  - В детето, `fork()` връща 0
  - В родителя, `fork()` връща число, по-голямо от 0: pid-а на детето

## Създаване на процеси

```
pid_t pid = fork();
if (pid < 0) {
    err(1, "could not fork");
}

if (pid > 0) {
    dprintf(1, "I am your father\n");
} else {
    dprintf(1, "Noooooooo!\n");
}
```

## Създаване на процеси

```
pid_t pid = fork();
if (pid < 0) {
    err(1, "could not fork");
}

if (pid > 0) {
    dprintf(1, "I am the parent\n");
    dprintf(1, "The child's pid is %d\n", pid);
} else {
    dprintf(1, "I am the child\n");
}

dprintf(1, "I am both\n");
```

## pids

- Както видяхме, PID-ът на процеса-дете се връща от `fork()`
- `getpid(2)` и `getppid(2)` връщат PID-а на текущия процес и на неговия родител:

```
pid_t my_pid = getpid();
pid_t parent_pid = getppid();

dprintf(
    1,
    "My pid is %d and my parent's pid is %d\n",
    my_pid, parent_pid
);
```

## Изчакване с `wait()`

- Системното извикване `wait(2)` блокира, докато някое дете на текущия процес умре
  - Аргументът му е указател, сочещ към променлива, в която `wait()` ще запише статуса на завършилoto дете
  - Стойността, върната от `wait()` е PID-а на детето

## Изчакване с wait()

- Всъщност, статусът, който `wait()` записва в аргумента си, кодира малко повече информация освен exit status-а на процеса-дете
- Например, можем да разберем дали процесът е бил убит или е завършил нормално
  - Макрото `WFEXITED(status)` проверява дали статусът е такъв на нормално-завършил процес
- Можем и да извлечем истинският exit status на процеса
  - Макрото `WEXITSTATUS(status)` извлича exit status-а
- За по-подробна информация, вижте `wait(2)`

## Изчакване с wait()

```
for (int i = 0; i < num_tasks; i++) {
    pid_t child_pid = fork();
    if (child_pid < 0) { err(1, "could not fork"); }
    if (child_pid == 0) {
        do_task(i);
        exit(0);      // The child does its work and exits
    }
}

for (int i = 0; i < num_tasks; i++) {
    int status;
    pid_t child_pid = wait(&status);
    if (child_pid < 0) { err(1, "could not wait for child");
    if (!WIFEXITED(status)) {
        warnx(1, "a task failed: child was killed!");
    } else if (WEXITSTATUS(status) != 0) {
        warnx(1, "a task failed (exit status != 0)!");
    }
}

dprintf(1, "all tasks completed successfully\n");
```

## Изчакване с `wait()`

- Със системното извикване `waitpid(2)` можем да изчакаме завършването на процес със конкретен PID
- Има и малко повече възможности от `wait()`
  - Може да провери дали процес е завършил, без да блокира
  - Може да чака за цяла група процеси
- За информация как да го ползвате, вижте `man страницата`.

## Наследяване на средата при fork()

- Процесът-дете наследява цялата среда на родителя си
  - Потребител (EUID, UID)
  - Права
  - Environment променливи
  - Отворени файлови дескриптори

## Наследяване на файлови дескриптори

```
int fd = open(
    "/tmp/test.txt",
    O_WRONLY|O_CREAT|O_TRUNC,
    0666
);
if (fd < 0) { err(1, "could not open file"); }

pid_t pid = fork();
if (pid < 0) { err(1, "could not fork"); }

for (int i = 0; i < 1000; i++) {
    if (pid == 0) {
        write(fd, "foo\n", 4);
    } else {
        write(fd, "bar\n", 4);
    }
}

close(fd);
```

## Наследяване на файлови дескриптори

- В предния пример получихме файл, в който имаме 1000 реда “foo” и 1000 реда “bar”, в произволен ред
- Двата процеса имат достъп до един и същ файлов дескриптор
- Процесите се състезават, кой от тях да запише своя текст и да премести указателя на файловия дескриптор напред
- В следващата тема ще се възползваме от наследяване на файлови дескриптори, за да правим по-интересни неща

Тръби и водопроводчици  
Операционни системи, ФМИ, 2022/2023

## pipe

- Системното извикване `pipe(2)` създава *тръба*
  - Тръбата е структура в ядрото, имплементираща FIFO опашка
- Взаимодействаме с тръбата през два файлови дескриптора:
  - `pipe()` приема като аргумент масив от 2 елемента, в който да запише номерата на двата файлови дескриптора
  - Дескриптор за четене (индекс 0)
  - Дескриптор за писане (индекс 1)

## pipe

```
int pfd[2];
if (pipe(pfd) < 0) {
    err(1, "could not create pipe");
}

pid_t pid = fork();
if (pid < 0) { err(1, "could not fork"); }
if (pid == 0) {
    close(pfd[0]);

    write(pfd[1], "foo\n", 4);

    close(pfd[1]);
    exit(0);
} else {
    close(pfd[1]);

    char buf[20];
    read(pfd[0], buf, 20);
    // ... do something with data
}
```

## pipe

- Тръбите са удобен метод за комуникация между процеси
- При четене от тръбата, текущият процес блокира докато някой друг не запише данни в тръбата
- Когато всички краища за писане се затворят, краищата за четене получават “край на файл” (EOF) и четенето от тях вече не блокира
- Нужно е всеки процес да затваря краищата на тръбата, които не ползва
  - В противен случай може да се получи deadlock
  - Например, може процесът, който пише данни, да е приключил, но процесът, който чете данни, да чака блокиран до безкрай, защото не е затворил своя край за писане

## Копиране на файлови дескриптори

- Можем да копираме файлови дескриптори със системните извиквания `dup()` и `dup2()`: вижте `dup(2)`
- `int dup(int oldfd)` - копира подадения файлов дескриптор с номер `oldfd` на първия свободен номер, и връща новия номер
- `int dup2(int oldfd, int newfd)` - копира подадения файлов дескриптор с номер `oldfd` като нов файлов дескриптор с номер `newfd` и връща `newfd`
  - Ако файлов дескриптор с номер `newfd` е съществувал, `dup2()` го затваря преди да направи копието
- Тези системни извиквания са много полезни, ако искаме да имплементираме пренасочване на стандартните потоци

## Пренасочване

Пренасочване на изхода на команда към файл:

```
int fd = open(
    "/tmp/test.txt",
    O_WRONLY|O_CREAT|O_TRUNC,
    0666
);
if (fd < 0) { err(1, "could not open file"); }

int result = dup2(fd, 1); // replace stdout by fd
if (result < 0) { err(1, "could not dup"); }

execlp("ps", "ps", "-e", (char*)NULL);
err(1, "could not exec");
```

## Пренасочване: pipe + dup

Пренасочване на текст към `stdin` на команда:

```
int pfd[2];
if (pipe(pfd) < 0) {
    err(1, "could not create pipe");
}

pid_t pid = fork();
if (pid < 0) { err(1, "could not fork"); }
if (pid == 0) {
    close(pfd[0]);

    write(pfd[1], "foo\n", 4);
    close(pfd[1]);
    exit(0);
} else {
    close(pfd[1]);

    dup2(pfd[0], 0); // replace stdin by pfd[0]
    execvp("wc", {"wc", "-m", (char*)NULL});
    err(1, "could not exec wc");
}
```

## pipe + dup

Пренасочване на изхода на една команда към входа на друга:

```
int pfd[2];
if (pipe(pfd) < 0) {
    err(1, "could not create pipe");
}

pid_t pid = fork();
if (pid < 0) { err(1, "could not fork"); }
if (pid == 0) {
    close(pfd[0]);
    dup2(pfd[1], 1); // replace stdout by pfd[1]
    execlp("ps", "ps", "-e", (char*)NULL);
    err(1, "could not exec ps");
} else {
    close(pfd[1]);
    dup2(pfd[0], 0); // replace stdin by pfd[0]
    execlp("grep", "grep", "firefox", (char*)NULL);
    err(1, "could not exec grep");
}
```

## pipe + dup

- Всъщност, shell-ът имплементира операторите >, >>, < и | точно така: чрез pipe() и dup().
- It's all just system calls!