

```
int pid = fork();
    if(pid > 0) {
        printPIDs("PARENT");
        wait(&child_status);
    } else if(!pid) {
        printPIDs("CHILD");
        exit(0);
    } else if(pid == -1) {
        printf("ERROR");
        return EXIT_FAILURE;
    }
```

LINUX SYSTEM PROGRAMMING



+IBA= 

Table of Contents

Въведение	1.1
Част 1 - Основи на Линукс	1.2
Какво е системно програмиране?	1.2.1
Работна среда	1.2.2
Отдалечен достъп	1.2.3
Как да получите помощ в Линукс?	1.2.4
Файлова система	1.2.5
Трансфер на файлове	1.2.6
Процеси в Линукс	1.2.7
Потребители и групи	1.2.8
Файлови разрешения	1.2.9
Управление на потребители и групи	1.2.10
Стандартни потоци	1.2.11
Пренасочване и тръбопроводи	1.2.12
Текстовият редактор Nano	1.2.13
Упражнение върху основи на Линукс	1.2.14
Част 2 – Програмиране	1.3
Компилиране	1.3.1
Компилятор	1.3.2
Изходен програмен код	1.3.3
Компилирайте в асемблер	1.3.4
Компилирайте до обектен файл	1.3.5
Компилирайте до изпълнима програма	1.3.6
Стартирайте изпълнимата програма	1.3.7
Библиотеки	1.3.8
Архиватор	1.3.9
Създаване на обектните файл	1.3.10
Създаване на статична и динамична библиотеки	1.3.11
Програма за намиране сумата на числа	1.3.12
Дебъгване	1.3.13
Упражнение върху тема програмиране	1.3.14
Част 3 – Файлова система и файлове	1.4
Файлове	1.4.1
Файлови системи и именовани пространства	1.4.2
Работа с файлове	1.4.3
Буферирани срещу небуферирани потоци	1.4.4

Библиотека за работа с файлове	1.4.5
Отваряне и затваряне на файл	1.4.6
Четене на съдържанието на файл	1.4.7
Четене и отпечатване на файл	1.4.8
Четене и писане на файл	1.4.9
Запис на изречения във файл	1.4.10
Търсене в файлове и откъслечни файлове	1.4.11
Направете файл с дупка	1.4.12
Заклучване на файлове	1.4.13
Заклучи и пиши там	1.4.14
Упражнение върху работа с файлове	1.4.15
Част 4 – Процеси	1.5
Процеси	1.5.1
Управление на процесите	1.5.2
Методи за работа с процеси	1.5.3
Изпълнение на команда	1.5.4
Показване на изходния код на файла	1.5.5
Стартиране на дъщерен процес	1.5.6
Отпечатване на идентификаторите на процесите	1.5.7
Изчакване и прекратяване на дъщерен процес	1.5.8
Изпълнение и отпечатване на състоянието	1.5.9
Зомбита и проста обработка на сигнали	1.5.10
Игра на зомбита	1.5.11
Упражнение върху процеси	1.5.12
Част 5 – Комуникация между процеси	1.6
Методи за комуникация между процеси	1.6.1
Сигнали	1.6.2
Най-важните сигнали	1.6.3
Управление на сигналите	1.6.4
Обработка на сигнал	1.6.5
Изчакване на сигнали	1.6.6
Разглеждане на сигнали	1.6.7
Показване на информация за сигналите	1.6.8
Reentrancy	1.6.9
Анонимни и наименувани тръби	1.6.10
Работа с тръби	1.6.11
Използване на анонимни тръби	1.6.12
Използване на именувани тръби	1.6.13

Упражнение върху комуникация между процеси	1.6.14
Част 6 - Синхронизация на комуникацията между процесите	1.7
Опашка за съобщения	1.7.1
Библиотеки за работа със съобщения	1.7.2
Определяне на общите данни	1.7.3
Сървър за съобщения	1.7.4
Клиент за съобщения	1.7.5
Споделена памет	1.7.6
Библиотеки за работа със споделена памет	1.7.7
Определяне на общите данни	1.7.8
Сървър за памет	1.7.9
Клиент за памет	1.7.10
Семафори	1.7.11
Библиотеки за работа със семафори	1.7.12
Определяне на общите данни	1.7.13
Семафор Сървър	1.7.14
Семафор Клиент	1.7.15
Упражнение за синхронизация на комуникация между процеси	
Част 7 – Сокети	1.8 1.7.16
Сокетите в Линукс	1.8.1
Работа със сокети в C	1.8.2
Използване на Unix сокети	1.8.3
Файл сокет сървър	1.8.4
Файл сокет клиент	1.8.5
Двойка сокети	1.8.6
Пример за двойка сокети	1.8.7
Мрежови сокети	1.8.8
Мрежов сокет сървър	1.8.9
Мрежов сокет клиент	1.8.10
Упражнение върху сокети	1.8.11
Част 8 – Нишки	1.9
Въведение в нишките	1.9.1
Библиотека за работа с нишки	1.9.2
Функции за управление на нишки	1.9.3
Създаване на нишки	1.9.4
Финализиране на нишки	1.9.5
Съединяване на нишки	1.9.6
Пример за съединяване на нишки	1.9.7

Функции за прекратяване на нишки	1.9.8
Типове при прекратяване на нишки	1.9.9
Пример за прекратяване на нишки	1.9.10
Упражнение върху нишки	1.9.11
Част 9 - Синхронизация на нишки	1.10
Изход от нишка	1.10.1
Управление на изход от нишка	1.10.2
Пример за изход от нишка	1.10.3
Защо е необходима синхронизация?	1.10.4
Механизми за синхронизация	1.10.5
Мютекси	1.10.6
Кога е необходимо заключване?	1.10.7
Типична употреба на мютекси	1.10.8
Проблеми при състезателни условия	1.10.9
Безопасен за нишките код	1.10.10
Състояние на мъртва хватка	1.10.11
Създаване и унищожаване на мютекси	1.10.12
Заключване и отключване на мютекси	1.10.13
Пример за синхронизиране посредством използване на мютекс	
Синхронизиране със семафори	1.10.15 1.10.14
Пример за синхронизиране посредством използване на семафор	
Упражнение за синхронизация на нишки	1.10.17 1.10.16
Част 10 – Демони	1.11
Какво са демоните?	1.11.1
Скелет на демон	1.11.2
Чат демон	1.11.3

Linux System Programming

Настоящият проект **Linux System Programming** е съвместна инициатива на Бургаски Свободен Университет и IBA Bulgaria, като допълнителен ресурс към безплатният курс **Линукс системно програмиране**. Учебното пособие е налично за свободно четене под формата на [електронна книга](#). Учебните ресурси са налични за свободно изтегляне от [електронно хранилище](#). Всички материали се разпространяват под лиценз [CC-BY-NC-SA](#).

Формат	ISBN
PDF	978-619-7126-69-3
MOBI	978-619-7126-70-9
EPUB	978-619-7126-71-6

Автор

Димитър Минчев е университетски преподавател към Център по информатика и технически науки при Бургаски свободен университет. Подготвя студенти за [Републиканската студентска олимпиада по програмиране](#) в [Клуб по състезателно програмиране](#). Организира съревнование за разработка на настолни и мобилни приложения [ХАКАТОН @ БСУ](#). Създава уникалните [Академията за таланти по програмиране](#) и [Школа по роботика](#) за ученици от Бургас. Инициира ученическото състезание по програмиране [CODE@BURGAS](#). Участва в националната програма [Обучение за ИТ умения и кариера](#) на Министерството на образованието и науката.

Контакт
Служебен: тел. +359 56 900 477 и e-mail: mitko@bfu.bg
Личен: моб. +359 899 148 872 и e-mail: dimitar.minchev@gmail.com
Блог: http://www.minchev.eu

Част 1 - Основи на Линукс

- [Какво е системно програмиране?](#)
- [Работна среда](#)
- [Отдалечен достъп](#)
- [Как да получите помощ в Линукс?](#)
- [Файлове, директории и файлови системи](#)
- [Трансфер на файлове](#)
- [Процеси в Линукс](#)
- [Потребители и групи](#)
- [Файлови разрешения](#)
- [Управление на потребители и групи](#)
- [Стандартни потоци](#)
- [Пренасочване на потоци](#)
- [Текстовият редактор Nano](#)
- [Упражнение върху основи на Линукс](#)

Какво е системно програмиране?

Системното програмиране (Термин на Английски език: *System Programming*) е дисциплина, която се занимава с разработката на **системен софтуер** (Термин на Английски език: *System Software*). Характерно за системния софтуер е, че обикновено той взаимодейства пряко с хардуера и системните библиотеки на **ядрото** (Термин на Английски език: *Kernel*) на операционната система и служи за платформа на други приложения, които в повечето случаи се използват от крайния потребител или така наречен **приложен софтуер** (Термин на Английски език: *Applied Software*).



Операционните системи могат също да се разглеждат като системен софтуер, тъй като на практика те управляват всички други програми и ресурси. Други примери са **системни драйвери** (Термин на Английски език: *System Drivers*), **шелл** (Термин на Английски език: *Shell*), **текстов редактор** (Термин на Английски език: *Text editor*), **компилятор** (Термин на Английски език: *Compiler*) и **дебъгер** (Термин на Английски език: *Debugger*), **инструменти** (Термин на Английски език: *Tools*) и **демони** (Термин на Английски език: *Daemons*).

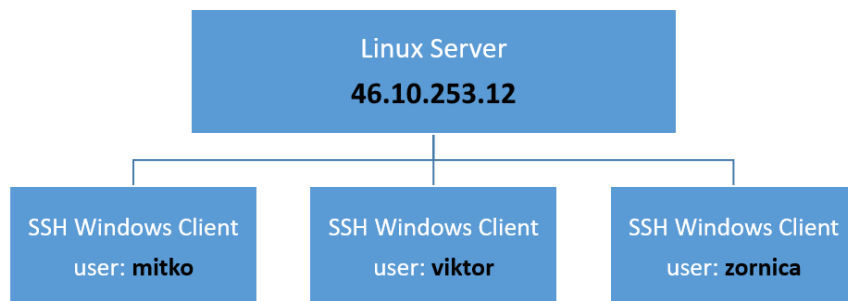
Понякога един програмен продукт може да се разглежда като приложен и системен софтуер. Системите за бази данни изпълняват системни функции за други програми, но предлагат и инструменти, използвани в ежедневната работа на системни оператори и анализатори на данни.

В Линукс, **системните инструменти** (Термин на Английски език: *System Tools*) обикновено се намират в директориите **/sbin** и **/usr/sbin**.

От **системните програмисти** (Термин на Английски език: *System Programmers*) се изисква да имат добра представа за хардуера и ОС, за да използват правилно и оптимално ресурсите на системата.

Работна среда

Работната среда се състои от един Линукс сървър с реален Интернет протокол адрес **46.10.253.12** и множество Windows клиенти. Достъпа на клиентите до сървъра се осъществява посредством [Telnet](#) използвайки [SSH](#).



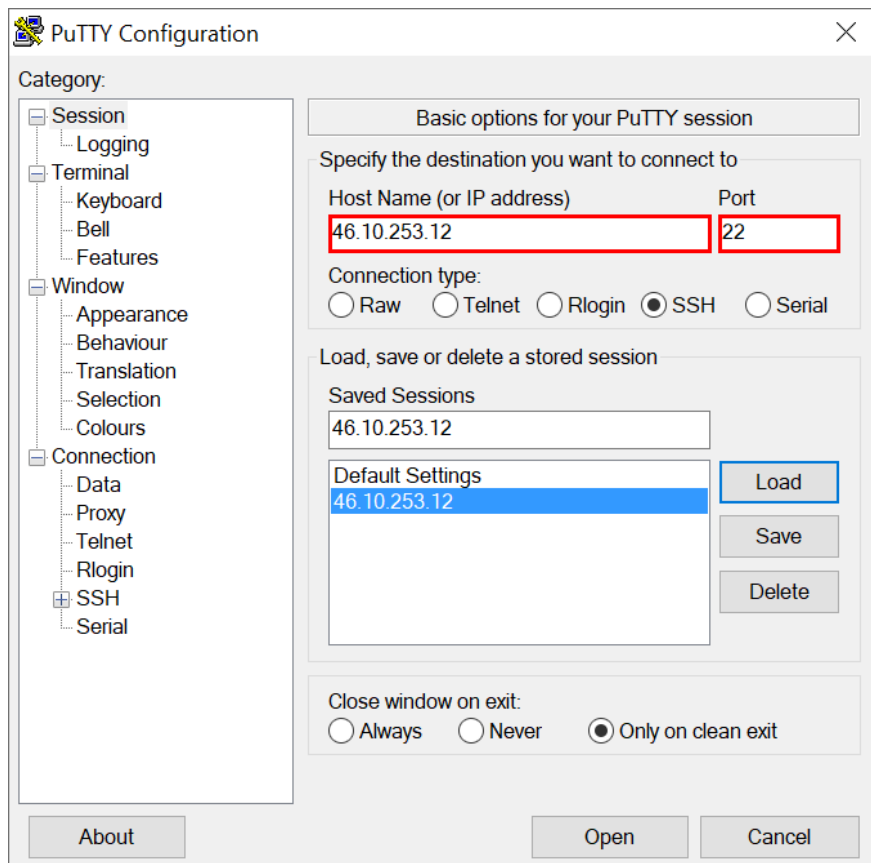
Telnet е разработен през 1969 г., като се започне с [RFC 15](#), разширен в [RFC 854](#) и стандартизиран като интернет стандарт [STD 8](#) на [Internet Engineering Task Force](#), един от първите интернет стандарти. Telnet осигурява достъп до интерфейс на командния ред на отдалечен хост до мрежови устройства и/или операционни системи. Поради сериозните опасения за сигурността при използването на Telnet през отворена мрежа като Интернет, използването му за тази цел значително намаля в полза на SSH.

SSH използва криптография с публичен ключ, за да удостовери отдалечения компютър и да му позволи да удостовери потребителя, ако е необходимо. Протоколът обикновено се използва за влизане в отдалечена машина и изпълнение на команди. Той може да прехвърля файлове, като използва свързаните протоколи за прехвърляне на файлове (SFTP) или защитени копия (SCP).

Системният софтуер включва **шелл** (Термин на Английски език: *Shell*), **текстов редактор** (Термин на Английски език: *Text editor*), **компилятор** (Термин на Английски език: *Compiler*) и **дебъгер** (Термин на Английски език: *Debugger*), **инструменти** (Термин на Английски език: *Tools*) и **демони** (Термин на Английски език: *Daemons*) на операционната система.

Отдалечен достъп

За осъществяване на отдалечен достъп до сървъра се използва програмата **PuTTY**, която се разпространява като софтуер с отворен код и може да бъде изтеглена безплатно от Интернет на адрес: <https://putty.org/>



Как да получите помощ в Линукс?

Повечето системни приложения и команди, работещи в конзолен режим на операционната система Линукс, имат съпътстваща документация във формата на страници от ръководства **manual pages**. За достъп до тях можете да използвате програмата **man**, със следния формат:

```
man <команда>
```

където <команда> е името на приложението, за което искате да получите информация. Ако съществува ръководство за избраната команда, man ще отпечата първата страница на екрана и ще предостави на потребителя средства за навигация в останалата част. Подробно описание на функциите, съответно можете да видите със следната команда:

```
man man
```

```
MAN (1)                                Manual pager utils                                MAN (1)
NAME
  man - an interface to the on-line reference manuals

SYNOPSIS
  man [-C file] [-d] [-D] [--warnings[=warnings]] [-R encoding] [-L locale] [-m system[,...]] [-M
  path] [-S list] [-e extension] [-i|-I] [--regex|--wildcard] [--names-only] [-a] [-u] [--no-subD
  pages] [-P pager] [-r prompt] [-7] [-E encoding] [--no-hyphenation] [--no-justification] [-p
  string] [-t] [-T[device]] [-H[browser]] [-X[dpi]] [-Z] [[section] page[.section] ...] ...
  man -k [apropos options] regexp ...
  man -K [-w|-W] [-S list] [-i|-I] [--regex] [section] term ...
  man -f [whatis options] page ...
  man -l [-C file] [-d] [-D] [--warnings[=warnings]] [-R encoding] [-L locale] [-P pager] [-r
  prompt] [-7] [-E encoding] [-p string] [-t] [-T[device]] [-H[browser]] [-X[dpi]] [-Z] file ...
  man -w|-W [-C file] [-d] [-D] page ...
  man -c [-C file] [-d] [-D] page ...
  man [-?V]

DESCRIPTION
  man is the system's manual pager. Each page argument given to man is normally the name of a
  program, utility or function. The manual page associated with each of these arguments is then
  found and displayed. A section, if provided, will direct man to look only in that section of
  Manual page man(1) line 1 (press h for help or q to quit)
```

По-важните клавиши, които можете да използвате при работа в програмата са **PgUp** за прелистване на страница нагоре, **PgDn** за прелистване на страница надолу и клавиша **q** за изход.

Файлове, директории и файлови системи

Файловете (*Термин на Английски език: Files*) са базово понятие във всички популярни операционни системи. Това напълно се отнася и за ОС Линукс, която се характеризира с това, че в нея “всичко е файл” (“everything is a file”). Файловете се групират в **директории** (*Термин на Английски език: Folders*), които от своя страна могат да съдържат други под-директории и по този начин се създава дървовидна структура.

Файловата система е начина за структуриране и организация на данните в компютъра. Тя представлява служебна таблица записана на диска, която операционната система използва, за да получи достъп до файловете. Записите в таблицата се наричат **inode** и накратко могат да се определят като сериен номер на файла.

Основни команди за работа с файловата система под Линукс са представени в таблицата по-долу:

Команда	Описание
pwd	Отпечатава името на текущата директория
ls	Списък на съдържанието на директорията
mkdir	Създава директория
cd	Смяна на работната директория
rmdir	Изтриване на директория
cp	Копиране на файлове и директории
mv	Преместване и/или преименуване на файлове
rm	Изтриване на файлове и/или директории
ln	Създава връзка между файлове
cat	Конкатенира файлове и отпечата на стандартния изход

Трансфер на файлове

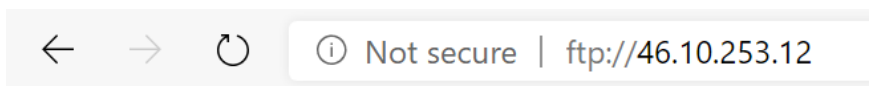
Операционната система предоставя различни средства за прехвърляне на информация между компютри. Обикновено се използва моделът "клиент-сървър", при който клиентите инициират връзки и заявки към сървър, който им отговаря.

Използваме протоколът за трансфер на файлове [File Transfer Protocol](#).


Стандартен мрежов протокол за прехвърляне на файлове между клиент и сървър в компютърна мрежа. Можете да копирате файлове локално във Вашата папка:

```
student@ubuntu:~$ pwd
/home/student
student@ubuntu:~$ ls
ftp
student@ubuntu:~$ ls ftp/
files
student@ubuntu:~$ █
```

Достъп до тези файлове можете да получите с всеки Интернет браузър, като се посетите адреса на работният сървър и се идентифицирате с потребителско име и парола за достъп:



Index of /

Name	Size	Date modified
 LSP.pdf	2.7 MB	3/16/20, 10:28:00 AM

Можете да управлявате файловете от Вашата папка с всеки един клиент поддържащ протокола FTP (Например: [FileZilla](#)).

Процеси в Линукс

Основната задача на операционните системи е да изпълняват компютърни програми. При стартирането на всяка програма се създават един или повече **процеси** (*Термин на Английски език: Processes*), които на практика представляват програмен код в изпълнение. Но освен статичен код, те също така се състоят от данни, ресурси, състояние и виртуализиран компютър.

В Линукс всеки процес се представя с уникален идентификатор, наречен **pid**. Когато няма активни процеси, операционната система фиктивно изпълнява специален **бездействащ процес** (*Термин на Английски език: Idle Process*), който има уникален идентификатор **pid = 0**.

Процесите могат да създават други процеси, като оригиналният процес се нарича **родител** (*Термин на Английски език: Parent*), а новосъздадения се нарича **дете** (*Термин на Английски език: Child*). По този начин се създават йерархични групи от процеси, които могат да комуникират и споделят ресурси помежду си.

Всеки процес е собственост на потребител и група, регистрирани в операционната система. В повечето случаи, това е потребителят, който е стартирал приложението и съответно групата, към която той принадлежи.

Потребители и групи

За да получи достъп до функциите на Линукс, всеки потребител трябва да премине през процес на идентификация наречен **login**, където се въвежда потребителско име и парола. След въвеждане на правилна комбинация, потребителят се регистрира в операционната система и се асоциира с уникален идентификатор наречен **uid**, който представлява цяло положително число. Тъй като имената също трябва да са уникални, в практиката те се използват по-често от **uid** за идентифициране на потребителите.

От своя страна, всеки потребител принадлежи към една или повече потребителски групи. По този начин едни и същи атрибути или права могат да се задават едновременно на множество потребители, в зависимост от тяхната роля в системата.

Авторизацията за достъп до ресурсите в Линукс се осигурява, чрез тази концепция за потребители и групи. Например, операциите, които може да извършвате с файловете в ОС, зависят от правата, които притежава Вашият потребител и групата, към която принадлежи.

Файлови права

Традиционните файлови системи поддържат три **режима** за използване на файловете: **четене** (Термин на Английски език: *Read*), **запис** (Термин на Английски език: *Write*) и **изпълнение** (Термин на Английски език: *Execute*).

От друга страна, всеки файл в Линукс принадлежи на **потребител** и **група**, като по подразбиране, това са създателя на файла и първичната му група. На базата на това са дефинирани три **нива** на правата за достъп до файлови операции: **собственик** (Термин на Английски език: *Owner*), **група** (Термин на Английски език: *Group*) и **публика** (Термин на Английски език: *Public*).

Чрез комбинирането на трите нива и трите режима се получават 9 възможни характеристики на файловете, които могат да се представят със следната битова маска (първият флаг показва дали полето описва директория или файл):

d	r	w	x	r	-	x	r	-	-
	READ	WRITE	EXEC	READ		EXEC	READ		
FILETYPE DIRECTORY	OWNER			GROUP			USER		
	4	2	1	4	0	1	4	0	0
	7			5			4		

На горния пример, притежателят на файла има пълни права над файла, членовете на групата на файла имат права за четене и изпълнение, а всички останали потребители имат само право за четене.

В конзолен режим на Линукс може да разгледаме правата на елементите в текущата директория с помощта на командата:

```
ls -la
```

```
sysadmin@ubuntu:~$ ls /usr/bin -al
total 103716
drwxr-xr-x  2 root  root    32768 Jun  1 15:29 .
drwxr-xr-x 10 root  root    4096 May  8 15:48 ..
-rwxr-xr-x  1 root  root   54772 Oct  4 2017 [
-rwxr-xr-x  1 root  root    96 Oct  4 2017 2to3-3.6
-rwxr-xr-x  1 root  root   22008 Sep 15 2017 aa-enabled
-rwxr-xr-x  1 root  root   22104 Sep 15 2017 aa-exec
-rwxr-xr-x  1 root  root   9984 Apr 28 2017 acpi_listen
-rwxr-xr-x  1 root  root   3452 Jul 20 2017 activate-global-python-argcomp
lete3
-rwxr-xr-x  1 root  root    6473 Oct 17 2017 add-apt-repository
-rwxr-xr-x  1 root  root   21916 Aug 14 2017 addpart
lrwxrwxrwx  1 root  root    24 Oct  1 2017 addr2line -> i686-linux-gnu-ad
dr2line
-rwxr-xr-x  1 root  root    2555 Oct 24 2017 apport-bug
-rwxr-xr-x  1 root  root   13348 May 11 01:27 apport-cli
lrwxrwxrwx  1 root  root    10 May 11 01:27 apport-collect -> apport-bug
-rwxr-xr-x  1 root  root   1849 May 11 01:27 apport-unpack
lrwxrwxrwx  1 root  root    6 Dec 13 2016 apropos -> whatis
-rwxr-xr-x  1 root  root   13732 Oct 26 2017 apt
```


В Линукс има дефинирани допълнителни специални режими за файловете и директориите. Един от тези режими е **SUID** (*Съкратено от: Set-User IDentification*) и в маската за правата се идентифицира със (s) вместо (x) на ниво собственик. Когато файл с изпълним код е в този режим и бъде изпълнен, то създадените процеси и ресурси ще принадлежат на собственика на файла, а не на потребителя, стартирал приложението.

Друг специален режим е **SGID** (*Съкратено от: Set-Group IDentification*) и се идентифицира със (s) вместо (x) на ниво група. Когато файл с изпълним код е в този режим и бъде изпълнен, то създадените процеси и ресурси ще принадлежат на групата на собственика на файла, а не на потребителя, стартирал приложението. Когато директория е в режим **SGID**, то създадените в нея файлове по подразбиране ще принадлежат на групата на горната директория.

Режимът **Sticky bit**, който се идентифицира със (s) вместо (x) на общо ниво, често се използва за споделени директории. Когато директория е в този режим, потребителите имат право да четат и изпълняват файлове на други потребители, но не могат да ги изтриват или преименуват.

Управление на потребители и групи

В таблицата по-долу са дадени команди от операционната система, които се използват за управление на потребители и групи:

Команда	Пояснение
id	отпечатва реални и ефективни потребителски и групови идентификатори
chmod	промяна на бита на файловия режим
umask	задаване на маска за създаване на файлов режим
chown	промяна на собственика на файла и групата
chgrp	промяна на собствеността на групата
passwd	промяна на потребителската парола

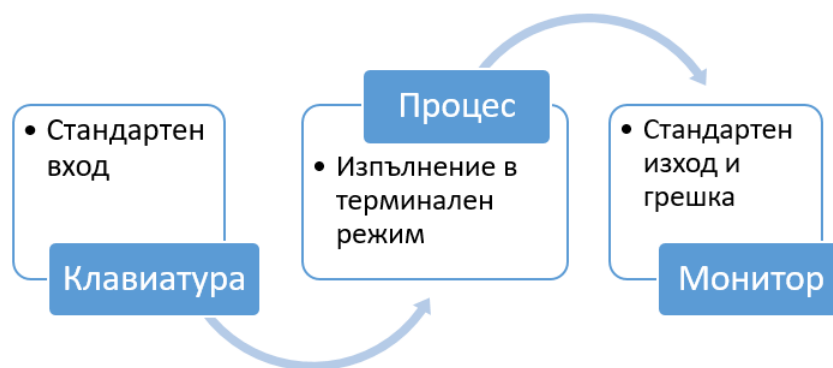
Стандартни потоци

В общия случай, компютърните програми получават входна информация (клавиатура, файл и т.н.), обработва я и извежда резултат, под формата на изходната информация (файлове, текст, изображение и т.н.). За стандартните конзолни приложения, входните данни се въвеждат от клавиатурата на компютъра, а изходните данни се извеждат на текстов екран.

В този модел на вход и изход на данните, не е известно предварително колко и какви данни ще бъдат получени от потребителите и съответно изведени като резултат. Такъв тип вход и изход се наричат текстови **потоци** (Термин на Английски език: *streams*).

Всяка програма, която изпълняваме на командния ред в Линукс, е свързана автоматично към три потока от данни:

поток	информация
STDIN (0)	Стандартен вход (данни, подадени в програмата, по подразбиране от клавиатурата)
STDOUT (1)	Стандартен изход (данни, отпечатани от програмата, по подразбиране към текстовия терминал)
STDERR (2)	Стандартна грешка (за съобщения за грешки, по подразбиране към текстовия терминал)



Пренасочване на потоци

Линукс поддържа различни средства за работа с входно-изходните потоци, например:

- Операторът **по-голямо** `>` пренасочва изхода на програмите към файл, вместо да бъдат отпечатани на екрана. Ако файлът вече съществува, съдържанието му ще бъде изтрито и версия с новото съдържание ще бъде запазена.
- Операторът **двойно по-голямо** `>>` пренасочва изхода и го добавя към файл, ако вече съществува.
- Операторът **конвейер** `|` пренасочва изхода на програмата отляво като вход на програмата отдясно.

```
sysadmin@ubuntu:~$ ls > file1.txt
sysadmin@ubuntu:~$ cat file1.txt
course
file1.txt
sysadmin@ubuntu:~$ ls >> file1.txt
sysadmin@ubuntu:~$ cat file1.txt
course
file1.txt
course
file1.txt
sysadmin@ubuntu:~$ ls | head -3 | tail -1
course
```

В горния пример създаваме текстов файл `file1.txt`, който съдържа списък с файловете в текущата директория. Забележете, че файлът, който създадохме, също е в списъка. Първо се създава файла, след това се изпълнява командата и изходът се запазва в него.

След това в същия файл отново добавяме листинга на текущата директория.

В третия пример извеждаме списъка с файлове в текущата директория, пренасочваме го към командата `head`, която взима първите три реда, които се пренасочват към командата `tail`, която оставя само последния ред.

Текстовият редактор nano

Текстовите файлове са често използвано средство за съхранение и обмен на информация в Линукс. Поради това, почти всяка една дистрибуция предлага средства за работа с тях. В нашата работна среда ще използваме текстовия редактор **nano**:

Някои от основните функции на **nano** са:

Файлов контрол

Клавиш	Пояснение
nano readme.txt	Отваря или създава файл readme.txt
Ctrl-o Y Enter	Запазва промените
Ctrl-r Alt-f	Отваря нов файл
Alt->	Превключва към следващ отворен файл
Alt-<	Превключва към предишен отворен файл
Ctrl-x	Изход от редактора

Навигация в съдържанието

Клавиш	Пояснение
Ctrl-a	Преместване в началото на текущия ред
Ctrl-e	Преместване в края на текущия ред
Ctrl-v	Прелистване страница надолу
Ctrl-y	Прелистване страница нагоре
Alt-\	Позициониране в началото на файла
Alt-/	Позициониране в края на файла
Alt-g	Позиционира към желан ред от файла

Копиране и вмъкване

Клавиш	Пояснение
Alt-a	Избор на блок за копиране или вмъкване
Alt-a Alt- ^	Копиране (Copy) на избрания блок в клипборда
Alt-a Ctrl-k	Изрязване(Cut) на избрания блок в клипборда
Ctrl-k	Изрязване(Cut) от позицията на курсора до края на реда
Ctrl-u	Вмъкване (Paste) съдържанието от клипборда на текущата позиция

Търсене и заместване

Клавиш	Пояснение
Ctrl-w	Търсене на текст
Alt-w	Повтаря последното търсене
Alt-r	Търсене и заместване

Полезна информация:

- [The Beginner's Guide to Nano, the Linux Command-Line Text Editor](#)
- [Nano text editor command cheatsheet](#)

Упражнение върху основи на Линукс

Упражнете материалите от темата, като реализирате представените по-долу задачи.

1. FTP

Посредством програмата **FileZilla** установете отдалечена връзка към работният сървър с адрес **46.10.253.12**. В домашната папка на потребителя качете файл **LSPp1.pdf**, който можете да изтеглите от Интернет на адрес https://github.com/dimitarminchev/LSP/presentations/01_Linux_Basics.pdf.

За потребителско име и парола за достъп използвайте тези предоставени от преподавателя. Алтернативно можете да влезете с потребител **students** и парола **password123**.

2. SSH

Посредством програмата **putty** установете отдалечена връзка към работният сървър с адрес **46.10.253.12**.

За потребителско име и парола за достъп използвайте тези предоставени от преподавателя. Алтернативно можете да влезете с потребител **students** и парола **password123**.

3. Terminal

1. Проверете дали файлът **LSPp1.pdf** присъства в домашната папка на потребителя.
2. Променете паролата си.
3. Във вашата домашна папка **~**, създайте нова папка **exercises** за упражненията и поддиректория в нея **day01** за тази лекция.
4. Отидете до последната папка **day01** и създайте файл **listing.txt** там, който съдържа списък на файловете от папката **/sbin**.
5. Отворете последния файл с текстовия редактор **nano** и добавете имената си в началото.
6. В домашната си директория създайте връзка към директорията **/home/students/**.

Пример

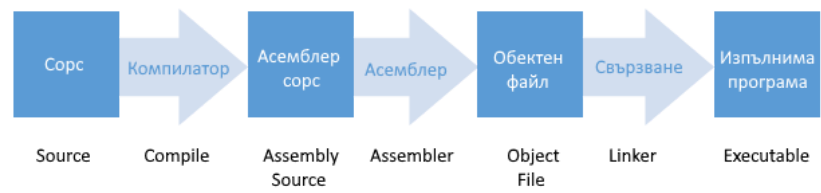
Какво е системно програмиране?

```
ls /home/students/LSPp1.pdf
passwd
cd ~
mkdir exercises
mkdir exercises/day01
cd exercises/day01/
ls -l /sbin/ > listing.txt
nano listing.txt
cd ~
ln -s /home/students/ shared
```


Част 2 – Програмиране

- [Компилиране](#)
- [Компилятор](#)
- [Изходен програмен код](#)
- [Компилирайте в асемблер](#)
- [Компилирайте до обектен файл](#)
- [Компилирайте до изпълнима програма](#)
- [Стартирайте изпълнимата програма](#)
- [Библиотеки](#)
- [Архиватор](#)
- [Създаване на обектните файлове](#)
- [Създаване на статична и динамична библиотеки](#)
- [Тестване на библиотека](#)
- [Дебъгване](#)
- [Упражнение върху тема програмиране](#)

Компилиране



Полезна информация: [Компилиране, асемблиране и свързване](#)

Компилятор

- Компиляторът GCC е разработен по проекта GNU и поддържа различни програмни езици.
- Първоначално е наречен GNU C Compiler, когато е работил само с програмния език C.
- Ще го използваме, за да направим нашите програми изпълними.

	
Developer(s)	GNU Project
Initial release	May 23, 1987; 30 years ago ^[1]
Stable release	8.1 ^[2] (6.x also supported) / May 2, 2018; 15 days ago
Repository	https://gcc.gnu.org/viewcvs/gcc/ 
Written in	C; and C++ since June 1, 2010; 7 years ago ^[3]
Operating system	Cross-platform
Platform	GNU
Type	Compiler
License	GNU GPL 3+ with GCC Runtime Library Exception ^[4]
Website	gcc.gnu.org 

Повече информация: https://en.wikipedia.org/wiki/GNU_Compiler_Collection

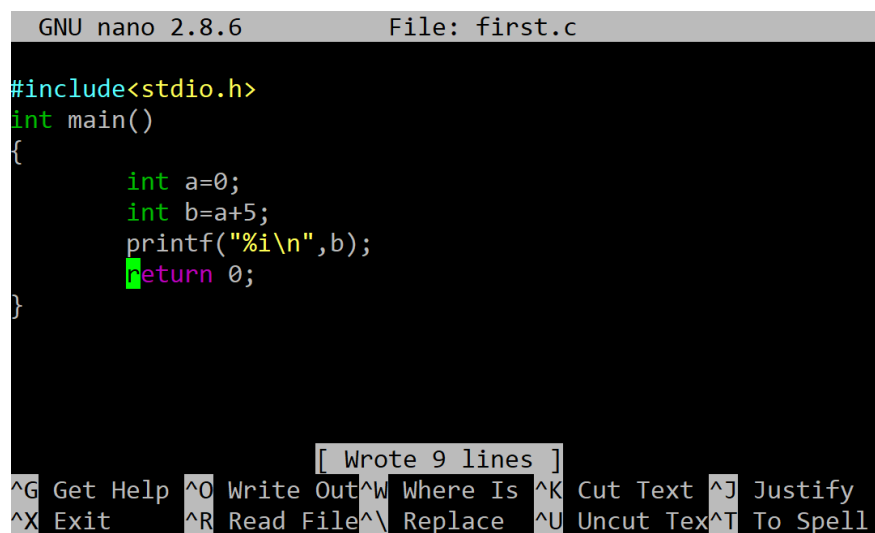
Изходен програмен код

Използвайки текстовия редактор **nano** напишете изходния програмен код във файл **first.c**

```
#include<stdio.h>

int main()
{
    int a=0;
    int b=a+5;
    printf("%i\n",b);
    return 0;
}
```

Изглед на изходния програмен код в текстовият редактор **nano**:



```
GNU nano 2.8.6      File: first.c

#include<stdio.h>
int main()
{
    int a=0;
    int b=a+5;
    printf("%i\n",b);
    return 0;
}

[ Wrote 9 lines ]
^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify
^X Exit      ^R Read File ^\ Replace  ^U Uncut Tex ^T To Spell
```

Компилирайте в асемблер

Използвайки компилатора **gcc** компилирайте в асемблер **assembly**, посредством изпълнението на следната команда:

```
gcc -S first.c -o first.s
```

Резултатът се записва във файл **first.s**, който трябва да изглежда по следния начин:

```
GNU nano 2.8.6      File: first.s
.
.file "first.c"
.section .rodata
.LC0:
.string "%i\n"
.text
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
leal 4(%esp), %ecx
.cfi_def_cfa 1, 0
andl $-16, %esp
pushl -4(%ecx)
pushl %ebp
.cfi_escape 0x10,0x5,0x2,0x75,0
movl %esp, %ebp
pushl %ebx
pushl %ecx
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell
```

Компилирайте до обектен файл

Използвайки компилатора **gcc** компилирайте до обектен файл **object**, посредством изпълнението на следната команда:

```
gcc -c first.c -o first.o
```

Резултатът се записва във файл **first.o**, който трябва да изглежда по следния начин:

```
GNU nano 2.8.6          File: first.o

^?ELF^A^A^A^@^@^@^@^@^@^@^@^@^A^@^C^@^A^@^@^@^@^@^@^@^@$
^@^D$^@GCC: (Ubuntu 7.2.0-8ubuntu3.2) 7.2.0^@T^@^@^@^@^@$
$P^@^@^@:^@^@^@^@^@^@^@^@^@^@P^@^@^@first.c^@main^@__x$
^M^@^@4^@^@^@    ^E^@^@<^@^@^@D^N^@^@    ^@^@^@B^B^@^@T^@^@^@$
^@^@^@D^@^@^@H^@^@^@A^@^@^@B^@^@^@^@^@^@^@^@^@^@^@\^A$

^G Get Help   ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify
^X Exit       ^R Read File ^\ Replace  ^U Uncut Tex ^T To Spell
```

Компилирайте до изпълнима програма

Използвайки компилатора **gcc** компилирайте до изпълнима програма **binary**, посредством изпълнението на следната команда:

```
gcc first.c -o first
```

Резултатът се записва във файл **first**, който трябва да изглежда по следния начин:

```
GNU nano 2.8.6      File: first
```

```
?ELF^A^A^A^@^@^@^@^@^@^@^@^C^@^C^@^A^@^@^@?^C^@^@4^@^@^@$  
^@^@^B^@^N^@^@^@^@^@^@^@^@?^^^@^@H^@^@^@?^^^@^@H^@^@^@?_ ^@$  
??^HPQ??P?]??E???' ^@^@^@^@U?S?SW?W?W?W?O^[ ^@^@??D??0^@^@^@$  
^@^A^[ ^C;8^@^@^@F^@^@^@????T^@^@^@????x^@^@^@Y?W?W?W?^@^@k$  
^@^@^@?^E^@^@^Y^@^@^@?^^^@^@[ ^@^@^@D^@^@^@Z^@^@^@?^^^@^@$  
^@^@^@?^@^@^K^@^@^@P^@^@^@U^@^@^@^@^@^@^@^C^@^@^@? ^@^$  
^@^@^@^@^@|^ ^C^@^@^@^@^@^@^C^@^K^@^@^@^@^@?^C^@^@^@^@^@^C$  
^@^@^@^@^@^@?^C^@^@^@^@^@^@^C^@^N^@^@^@^@^@?^E^@^@^@^@^@^C$
```

```
[ Read 8 lines (Converted from Mac format) ]
```

```
^G Get Help   ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify  
^X Exit       ^R Read File ^\ Replace  ^U Uncut Tex ^T To Spell
```

Стартирайте изпълнимата програма

За да стартирате изпълнимата програма използвайте следния команден ред:

```
./first
```

Резултатът от изпълнението трябва да изглежда по следния начин:

```
sysadmin@ubuntu:~/tmp$ ./first  
5  
sysadmin@ubuntu:~/tmp$ █
```


Библиотеки

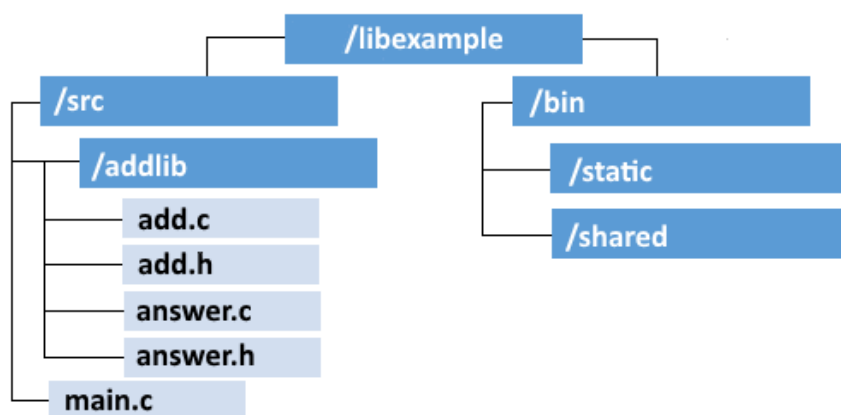
Библиотеките представляват множество компилирани обекти в един файл.

Техните преимущества са: **повторно използване на компоненти** (използване на една споделена библиотека заема по-малко място на диска), **управление на версиите** (стари и нови версии съжителстват едновременно на една Linux система), **компонентна специализация** (разработчиците могат да фокусират основната си компетентност в една библиотека).

Видовете библиотеки са: **статични** (обектен код в свързана библиотека, който става част от приложението) и **динамични** (споделени обекти, динамично свързвани по време на изпълнението).

Структура на библиотека

На фигурата по-долу е дадена примерна структура на библиотека:



Структура на директориите

Използвайте следните команди да създадете структурата на директориите:

```
mkdir libexample
mkdir libexample/src
mkdir libexample/src/addlib
mkdir libexample/bin
mkdir libexample/bin/static
mkdir libexample/bin/shared
```

Източник: [Creating a shared and static library with the gnu compiler gcc](#)

Файлове на библиотеката

В папка `libexample/src/addlib/` създайте файл `add.c` със следното съдържание:

Какво е системно програмиране?

```
#include <stdio.h>
int gSummand;
void setSummand(int summand) {
    gSummand = summand;
}
int add(int summand) {
    return gSummand + summand;
}
void __attribute__((constructor)) initLibrary(void) {
    printf("Library is initialized\n");
    gSummand = 0;
}
void __attribute__((destructor)) cleanUpLibrary(void) {
    printf("Library is exited\n");
}
```

В папка `libexample/src/addlib/` създайте файл `add.h` със следното съдържание:

```
void setSummand(int summand);
int add(int summand);
```

В папка `libexample/src/addlib/` създайте файл `answer.c` със следното съдържание:

```
#include "add.h"
int answer() {
    setSummand(20);
    return add(22); // 42 = 20 + 22
}
```

В папка `libexample/src/addlib/` създайте файл `answer.h` със следното съдържание:

```
int answer();
```

В папка `libexample/src/` създайте файл `main.c` със следното съдържание:

Какво е системно програмиране?

```
#include <stdio.h>
#include "addlib/add.h"
#include "addlib/answer.h"
int main(int argc, char* argv[])
{
    setSummand(5);
    printf("5 + 7 = %d\n", add(7));
    printf("And the answer is: %d\n", answer());
    return 0;
}
```

Архиватор

Архиваторът позволява: създаване, модифициране и извличане на архиви. С негова помощ можете да създавате библиотеки. Например следната команда създава библиотека:

```
ar rcs <library-name>.a <module-1>.o <module-2>.o ...
```

За да изведете списъка на файловете от библиотека използвайте следната команда:

```
ar -t <library-name>.a
```

Създаване на обектните файлове

За създаване на обектните файлове (директория `libexample`) използвайте следната команда:

```
gcc -c src/main.c -o bin/main.o
```

За създаване на обектните файлове предназначени за статична библиотека, използвайте:

```
gcc -c src/addlib/add.c -o bin/static/add.o  
gcc -c src/addlib/answer.c -o bin/static/answer.o
```

За създаване на обектните файлове предназначени за споделена библиотека, използвайте:

```
gcc -c -fPIC src/addlib/add.c -o bin/shared/add.o  
gcc -c -fPIC src/addlib/answer.c -o bin/shared/answer.o
```

Създаване на статична и динамична библиотеки

Комбинируйте файловете на обектите в една статична библиотека / архив, посредством командата:

```
ar rcs bin/static/libadd.a bin/static/add.o bin/static/answer.o
```

Свържете статично main.o с библиотеката, посредством командата:

```
gcc bin/main.o -Lbin/static -ladd -o bin/statically-linked
```

За споделена библиотека на GCC завършваща с .so вместо с .a, използвайте командата:

```
gcc -shared bin/shared/add.o bin/shared/answer.o -o bin/shared/libadd.so
```

Свържете се динамично със споделената библиотека, посредством командата:

```
gcc bin/main.o -Lbin/shared -ladd -o bin/use-shared-library
```

Преместете споделената библиотека в местоположението по-подразбиране:

```
sudo mv bin/shared/libadd.so /usr/lib  
sudo chmod u=rwx,go=rx /usr/lib/libadd.so
```

Използвайте споделената библиотека посредством **LD_LIBRARY_PATH**:

```
LD_LIBRARY_PATH=$(pwd)/bin/shared bin/use-shared-library
```

Тестване на библиотека

В папка `libexample/` създайте файл `test.c` със следното съдържание:

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

int main()
{
    // Load the dynamic library:
    void* lib_add = dlopen("bin/shared/libadd.so", RTLD.LAZY | RTLD_GLOBAL);

    // Declare the function pointers:
    void (*fptr_setSummand)(int);
    int (*fptr_add)(int);

    // Get the pointers to the functions within the library:
    fptr_setSummand = dlsym(lib_add, "setSummand");
    fptr_add = dlsym(lib_add, "add");

    // Call the function via the function pointer:
    fptr_setSummand(42);
    int result = fptr_add(7);
    printf("Result: %d\n", result);

    return 0;
}
```

Компилирайте и стартирайте тестовата програма използвайки командите:

```
gcc test.c -ldl -o test
./test
```

В резултат при успешно изпълнение трябва да видите следните три реда:

```
Library is initialized
Result: 49
Library is exited
```

Дебъгване

Преди да се запознаем със възможностите за дебъгване да напишем кратка програма за намиране на сума на числа: **suma.c**

Сумиране на числа

Инициализираме променлива **sum=0**. Отпечатваме стойността на **argc**.
Отпечатваме стойностите в **argv**. Броят на числата **num_count** е втората стойност в **argv**. **num_count** на брой пъти четем число от клавиатурата и го добавяме към сумата. Отпечатваме стойността на получената сума.

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

int main(int argc, char* argv[])
{
    int numbers_count = 0, sum = 0;
    char temp_str[50];
    printf("The value of argc: %d\n", argc);
    printf("The value(s) in argv:\n");
    for(int i = 0; i < argc; i++) {
        printf("> argv[%d]=%s\n", i, argv[i]);
    }
    numbers_count = atoi(argv[1]);
    printf("Enter %d numbers:\n", numbers_count);
    for(int i = 0; i < numbers_count; i++) {
        scanf("%s", temp_str);
        sum += atoi(temp_str);
    }
    printf("Total sum is %d\n", sum);
}
```

```
$ ./suma 4
The value of argc: 2
The value(s) in argv:
> argv[0]=./suma
> argv[1]=4
Enter 4 numbers:
1
2
3
4
Total sum is 10
```

Отстраняване на грешки

Какво е системно програмиране?

За да компилирате изходния код на програмата с включена възможност за дебъгване:

```
gcc suma.c -o suma
```

За да стартирате дебъгера:

```
gdb suma
```

За да поставите точка на прекъсване, където програмата ще спре изпълнението:

```
b [function name, line number]
```

За да стартирате програмата:

```
r [command line arguments]
```

В таблицата са дадени съкратените клавишни комбинации полезни при работа с дебъгера:

Клавиш	Информация
h	Помощ
n	Стъпка напред един блок код
s	Стъпка напред един ред код
p [variable]	Отпечатва стойността на променливата variable
info locals	Отпечатва стойностите на всички локални променливи
bt	Показва последователността на функциите, наречени до тази точка на изпълнение
q	Изход

Повече информация: [Introduction to GDB](#) и [Harvard University CS50](#)

Упражнение върху тема програмиране

Упражнете материалите от темата, като реализирате представените по-долу проекти.

Multi Add

Напишете програма **multi-add.c**, която взима едно или повече числа като входни аргументи, след това изчислява общата сума и отпечатва отговора на стандартния изход. Компилирайте програмата и я изпълнете. Компилирайте. Опитайте отстраняване на грешки посредством дебъгера.

Reverse Test

Напишете програма **reverse-test.c**, която чете низ от клавиатурата и го отпечатва наобратно. За да обърнете входния низ, трябва да използвате библиотеката **reverse**, която е предоставена в директорията **/day02/reverse/**. Функцията, която трябва да използвате, е:

```
void inplace_reverse(char * str)
```

Имате всички източници, за да можете да компилирате и свързвате библиотеката статично или можете да използвате споделената библиотека **reverse** от **/usr/lib/libreverse.so**. Най-добре е да опитате и двата подхода.

Част 3 – Файлова система и файлове

- Файлове
- Файлови системи и именовани пространства
- Работа с файлове
- Буферирани срещу небуферирани потоци
- Библиотека за работа с файлове
- Отваряне и затваряне на файл
- Четене на съдържанието на файл
- Четене и отпечатване на файл
- Четене и писане на файл
- Запис на изречения във файл
- Търсене в файлове и откъслечни файлове
- Направете файл с дупка
- Заклучване на файлове
- Заклучи и пиши там
- Упражнение върху работа с файлове

Файлове

Един от основните принципи в Unix и Linux гласи: **Всичко е файл!**.

Файловете с данни, директориите, информация за хардуерните устройства в системата, информация за процесите или важни конфигурационни параметри на системата са представени като файлове.

Съществуват следните видове файлове:

- **inode** - структура данни, която описва обект на файловата система.
- Файловете винаги се отварят от потребителското пространство с име.
- Двойката **filename** и **inode** се нарича връзка (link).
- **Редовни файлове** = байтове данни.
- **Директории** = картографиране между имена на файлове и inodes (връзки).
- **Твърди връзки** = множество връзки свързват различни имена със същия inode.
- **Символни връзки** = като обикновени файлове, които съдържат пълния път на свързаните файлове.
- **Специални файлове** = файлове с блокирани устройства, файлове със символни устройства, наречени тръби, досиета за Unix домейни.

Файлови системи и именовани пространства

Linux осигурява глобално и единно пространство от имена на файлове и директории с корен /. Файлова система е колекция от файлове и директории в официална и валидна йерархия. Файловите системи могат да бъдат добавени (монтирани) поотделно и премахнати (демонтирани) от глобалното пространство на имената на файлове и директории. Някои директории са специални, например **/dev** и **/proc**.

```
sysadmin@ubuntu:/dev$ ls
autofs          kmsg            sg0             tty30           tty61           ttyS5
block           kvm             sg1             tty31           tty62           ttyS6
bsg             lightnvm        shm             tty32           tty63           ttyS7
btrfs-control   log            snapshot        tty33           tty7            ttyS8
bus             loop0          snd             tty34           tty8            ttyS9
cdrom           loop1          sr0             tty35           tty9            ubuntu-vg
cdwr            loop2          stderr          tty36           ttyprintk       uhid
char            loop3          stdin           tty37           ttyS0           uinput
console         loop4          stdout          tty38           ttyS1           urandom
core            loop5          tty             tty39           ttyS10          userio
cpu             loop6          tty0            tty4            ttyS11          vcs
cpu_dma_latency loop7          tty1            tty40           ttyS12          vcs1
cuse            loop-control   tty10           tty41           ttyS13          vcs2
disk            mapper         tty11           tty42           ttyS14          vcs3
dm-0            mcelog         tty12           tty43           ttyS15          vcs4
```

```
sysadmin@ubuntu:/proc$ ls
1      18      25210   42      59      766      diskstats  net
10     18956   25211   43      6        774      dma         pagetypeinfo
1044   19      25212   431     60       78       driver      partitions
1047   2       25298   432     61       783      execdomains sched_debug
11     20      25299   437     62       7922     fb          schedstat
12     20693   25327   439     63       8        filesystems scsi
124    21      255     44      64       80       fs          self
125    21753   256     440     65       800      interrupts slabinfo
128    21816   26      441     66       81       iomem       softirqs
12841  21934   27      442     67       815      ioports     stat
129    22      27640   445     68       82       ipmi        swaps
13     23259   28      447     69       822      irq         sys
130    24      29998   448     7        855      kallsyms    sysrq-trigger
131    240     30      449     70       882      kcore       sysvipc
```

Структурата на директориите в Линукс е дефинирана със стандарт, който може да бъде намерен на <http://www.pathname.com/fhs/>.

Работа с файлове в С

Преди да може да се чете или пише файл, той трябва да бъде отворен. Ядрото поддържа списък на отворените файлове, наречен файлова таблица. Тази таблица се индексира чрез неотрицателни цели числа, известни като дескриптори на файловете (често съкратени `fds`). Всеки запис в списъка съдържа информация за отворен файл, включващ указател към копие в паметта на файла за поддръжка на `inode` и свързаните с него метаданни, като например позицията на файла и режимите за достъп. Както потребителското пространство, така и пространството на ядрото използват файловете дескриптори като уникални "бисквитки" по време на работа. Отварянето на файл връща дескриптор на файла, а последващите операции (четене, писане и т.н.) приемат описанието на файла като основен аргумент.

Основни методи за работа с файлове са представени в таблицата по-долу:

Метод	Пояснение
<code>fopen()</code>	Отваря файл и връща файлов дескриптор
<code>fread()</code>	Чете данни от файл
<code>fwrite()</code>	Записва данни във файл
<code>fflush()</code>	Изчиства потока
<code>close()</code>	Затваря файловия дескриптор
<code>fseek()</code>	Задава позицията на файловия дескриптор до дадена стойност
<code>fcntl()</code>	Манипулира файловия дескриптор (Например за заключване)
<code>errno</code>	Номер на последната грешка

Буферирани срещу небуферирани потоци

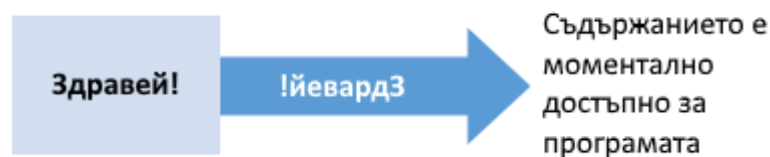
Потокът представлява предаване на данни от една страна към друга. Файлът представлява място за съхраняване на данни на диск. Файлът използва потоци за съхранение и зареждане на данни. Буферът се използва временно за съхраняване на данните от потока.

Знаците, написани или четени от **небуфериран поток** (unbuffered stream), се предават поотделно във или от файла възможно най-скоро. Знаци, написани или четени от **напълно буфериран поток** (fully buffered), се предават към или от файла в блокове с произволен размер. Знаците, написани на **линейно буфериран поток** (line buffered stream), се предават към файла в блокове, когато се появява нов ред.

Файлови операции на ниско ниво - небуфериран поток

Методите `open()`, `read()`, `write()`, `lseek()` са част от библиотеката `unistd.h`. Работят с файловия дескриптор (**INT**) и третираят входно/изходния поток като бинарни данни.

небуфериран



Файлови операции на ниско ниво - буфериран поток

Методите `fopen()`, `fread()`, `fwrite()`, `fseek()` са част от библиотеката `stdio.h`. Използват обектния тип **FILE** и третираят входно/изходния поток като текст. Операциите по четене/запис на акумулирания буфер може да бъде форсирани посредством метод `fflush()`.

буфериран



Библиотека за работа с файлове

Библиотеката **unistd.h** съдържа методи за работа с файлове:

```
#include<unistd.h>

int open(const char *name, int flags);

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

int close();
```

Параметъра флаг (**flag**) е бит маска определяща следното:

флаг	Информация
O_APPEND	Файлът ще бъде отворен в режим на добавяне.
O_ASYNC	SIGIO се генерира, когато може да се чете или да се записва.
O_CREAT	Ако файлът не съществува го създава.
O_DIRECT	Отваря файла за директен вход/изход.
O_DIRECTORY	Ако името не е директория, то дава грешка
O_EXCL	Ако O_CREAT и файлът съществува, то дава грешка.
O_LARGEFILE	Отваря файл по-голям от 2 ГБ
O_NOCTTY	Този флаг не се използва
O_NOFOLLOW	Ако пате е символна връзка, то дава грешка
O_NONBLOCK	Ако е възможно, отваря файла в неблокиращ режим.
O_SYNC	Файлът ще бъде отворен за синхронен Вход/Изход
O_TRUNC	Ако файлът съществува, той се съкращава до нулева дължина

В случай че е зададен флаг **O_CREAT**, то режима на работа е бит маска, както следва:

S_IRWXU	S_IXUSR	S_IWGRP	S_IROTH
S_IRUSR	S_IRWXG	S_IXGRP	S_IWOTH
S_IWUSR	S_IRGRP	S_IRWXO	S_IXOTH

Отваряне и затваряне на файл

- Инициализирайте променливите
- Отворете файла само за четене и получите файлов дескриптор
- Ако файловият дескриптор е -1, отпечатайте съобщение за грешка
- В противен случай, отпечатайте стойността на файловия дескриптор
- Затворете файла
- Ако затварянето е неуспешно, изведете съобщение за грешка

openclose.c

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

int main(int argc, char * argv[])
{
    int fd;
    fd = open ("/proc/self/environ", O_RDONLY);
    if (fd == -1) {
        printf("ERROR opening 'environ'!\n");
    } else {
        printf("File Descriptor of 'environ' = %d\n", fd);
    }

    // write/read commands to be added here
    if (close (fd) == -1)
        printf("ERROR closing the file!\n");
}
```

Четене на съдържанието на файл

Библиотеката **unistd.h** съдържа методи за работа с файлове:

```
#include<unistd.h>

ssize_t read(int fd, void *buf, size_t len);
```

- Всяко извикване чете **len** на брой байтове в **buf** от текущата позиция на файла, посочен в файловия дескриптор **fd**.
- При успех се връща броят байтове записани в **buf**.
- При грешка в резултат се връща -1, а информация за грешката се записва във **errno**.

Всъщност метода за четене може да доведе до много възможности:

- Връща стойност, равна на **len** и всички прочетени байтове са запазени в **buf**. Резултатите са както е предвидено.
- Връща стойност по-малка от **len**, но по-голяма от нула. Четените байтове се съхраняват в **buf**. Това може да се случи, защото сигналът прекъсва четенето в средата, в средата на четенето е възникнала грешка, повече от нула, но е налице по-малко от стойността на байтовете с леки байтове или EOF е достигнато, преди да бъдат прочетени байтове. Преиздаването на прочетеното (с съответно актуализираните стойности **buf** и **len**) ще прочете останалите байтове в останалата част от буфера или ще посочи причината за проблема.
- Връща стойност 0. Това показва **EOF**. Няма какво да чете.
- Извикването на метода блокира, тъй като няма налични данни. Това няма да се случи в режим на блокиране.
- Връща стойност -1 и грешката е зададена на **EINTR**. Това показва, че е получен сигнал преди четенето на байтове.
- Връща стойност -1, а грешката е зададена на **EAGAIN**. Това показва, че четенето би блокирало, защото понастоящем няма налични данни. Това се случва само в режим на блокиране.
- Връща стойност -1, а грешката е зададена на стойност, различна от **EINTR** или **EAGAIN**. Това показва по-сериозна грешка.

Четене и отпечатване на файл

Инициализирайте променливите. Отворете файла **readfile.c**, при проблем отпечатайте грешка. Докато четенето от файла връща дължина различна от 0. Ако дължината е -1 и грешката е **EINTR**, опитайте да прочетете отново. Ако дължината е -1 и грешката <> **EINTR**, изведете грешка и преустановете. В противен случай отпечатайте буфера. Затворете файла, отпечатайте грешка при проблем.

readfile.c

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#define BUF_SIZE 1000000
int main(int argc, char *argv[])
{
    int fd = open("./readfile.c", O_RDONLY);
    if (fd == -1)
    {
        printf("ERROR opening the file!\n");
        return -1;
    }
    ssize_t len;
    char buf[BUF_SIZE];
    while ((len = read(fd, buf, BUF_SIZE - 1)) != 0)
    {
        if (len == -1)
        {
            if (errno == EINTR) continue;
            printf("ERROR reading the file!\n");
            break;
        }
        printf("%s", buf);
    }
    if (close(fd) == -1)
    {
        printf("ERROR closing the file!\n");
        return -2;
    }
    return 0;
}
```

Четене и писане на файл

```
#include<unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

Библиотеката **unistd.h** съдържа методи за работа с файлове:

- Записва **count** на брой байтове, започващи от **buf** до текущата позиция във файла, посочен от файловия дескриптор **fd**.
- При успех се връща броят на записани байтове и се актуализира текущата позиция във файла.
- При грешка в резултат се връща -1, а информацията се записва във **errno**.

Note:

When a call to `write()` returns, the kernel has copied the data from the supplied buffer into a kernel buffer, but there is no guarantee that the data has been written out to its intended destination. Indeed, `write` calls return much too fast for that to be the case. The disparity in performance between processors and hard disks would make such behavior painfully obvious.

Instead, when a user-space application issues a `write()` system call, the Linux kernel performs a few checks, and then simply copies the data into a buffer. Later, in the background, the kernel gathers up all of the "dirty" buffers, sorts them optimally, and writes them out to disk (a process known as `writeback`). This allows `write` calls to occur lightning fast, returning almost immediately. It also allows the kernel to defer writes to more idle periods, and batch many writes together.

The delayed writes do not change POSIX semantics. For example, if a read is issued for a piece of just-written data that lives in a buffer and is not yet on disk, the request will be satisfied from the buffer, and not cause a read from the "stale" data on disk. This behavior actually improves performance, as the read is satisfied from an in-memory cache without having to go to disk. The read and write requests interleave as intended, and the results are as expected—that is, if the system does not crash before the data makes it to disk! Even though an application may believe that a write has occurred successfully, in this event, the data will never make it to disk.

Запис на изречения във файл

Създайте празен файл **sentences.txt**, в случай на неуспех отпечатайте грешка и излезте. Направете 100 пъти следното: извикайте **getSentence()**, за да получите нов текст в буфера и неговата дължина, запишете буфера във файла и запишете броя на байтове. Ако броят на записаните байтове е -1, отпечатайте грешка и излезте. Затворете файла, при неуспех изведете грешка.

writefile.c

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/time.h>
#include <string.h>
#define BUF_SIZE 1000000

long getMicrotime(){
    struct timeval currentTime;
    gettimeofday(&currentTime, NULL);
    return currentTime.tv_sec * (int)1e6 + currentTime.tv_usec;
}

int getSentence(char *buf) {
    sprintf(buf, "%ld> This is a new sentence.\n", getMicrotime());
    return strlen(buf);
}

int main(int argc, char *argv[])
{
    int fd;
    fd = open("./sentences.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd == -1)
    {
        printf("ERROR opening the file!\n");
        return -1;
    }
    char buf[BUF_SIZE];
    for (int i=0; i < 100; i++)
    {
        int text_len = getSentence(buf);
        ssize_t nr = write(fd, buf, text_len);
        if (nr == -1)
        {
            printf("ERROR writing to the file!\n");
            break;
        }
    }
    if (close(fd) == -1)
    {
        printf("ERROR closing the file!\n");
        return -2;
    }
    printf("SUCCESS!\n");
    return 0;
}
```

Търсене в файлове и откъслечни файлове

```
#include<stdio.h>

FILE *fopen(const char *pathname, const char *mode);

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

int fseek(FILE *stream, long offset, int whence);

int fclose(FILE *stream);
```

- Поведението на **fseek()** зависи от аргумента: **SEEK_CUR**, **SEEK_END** или **SEEK_SET**.
- Методът връща новата позиция на файла при успех.
- При грешка в резултат се връща -1, а информация за грешката се записва във **errno**.
- Търсене след края на файла и след това писане в него причинява дупки (**holes**) запълнени с нули.
- Файлове с дупки се наричат откъслечни файлове (**sparse files**).
- Дупките не заемат физически дисково пространство.
- **du** = оценка на използваното дисково пространство от файла.

Аргумент	Пояснение
SEEK_CUR	Взима се стойността на текущата файлова позиция (fd) плюс стойността POS , която може да бъде отрицателна, нулева или положителна. POS от нула връща стойността на текущата файлова позиция.
SEEK_END	Стойността на текущата файлова позиция (fd) става равна на дължината на файла плюс стойността POS , която може да бъде отрицателна, нулева или положителна. POS от нула задава изместване в края на файла.
SEEK_SET	Стойността на текущата файлова позиция (fd) става равна на стойността POS . POS от нула задава изместване в началото на файла.

Повече информация: [Sparse Files](#)

Направете файл с дупка

- Инициализирайте променливите
- Създайте файл, като използвате за името му първият аргумент, предаден на програмата
- Ако не се отвори, изведете грешка и излезте
- Запишете името на файла във файла
- Прескочете 16777216 байта напред във файла
- Запишете отново името на файла във файла
- Затворете файла

makeparse.c

```
#include <stdio.h>
#include <string.h>

#define BIG_SIZE 0x1000000

int main(int argc, char * argv[])
{
    FILE * f;
    f = fopen(argv[1], "w");
    if (f == NULL)
    {
        printf("ERROR creating file: %s", argv[1]);
        return 1;
    }
    fwrite(argv[1], 1, strlen(argv[1]), f);
    fseek(f, BIG_SIZE, SEEK_CUR);
    fwrite(argv[1], 1, strlen(argv[1]), f);
    fclose(f);
}
```


Заклучване на файлове

Използвайте методът **fcntl()**, който осигурява указател към структурата **flock**. Това извикване манипулира файловия дескриптор **fd**, в зависимост от командата **cmd**:

- За да заключите блок на файл, използвайте **F_SETLK**.
- Ако блокът вече е заключен, използвайте **F_GETLK**, за да получите информация за процеса на заключване.
- За да отключите блок на файл, използвайте **F_SETLK**, но задайте **flock.l_type = F_UNLCK**.

```
#include<stdio.h>
FILE *fopen(const char *pathname, const char *mode);
size_t fread(void *ptr, size_t size, size_t nmem, FILE *stream);
int fseek(FILE *stream, long offset, int whence);
int fclose(FILE * stream);
```

Структурата **flock**

```
struct flock {
    ...
    short l_type;    // F_RDLCK, F_WRLCK, F_UNLCK
    short l_whence;  // SEEK_SET, SEEK_CUR, SEEK_END

    off_t l_start;   // Starting offset for lock
    off_t l_len;     // Number of bytes to lock

    pid_t l_pid;     // PID of process blocking our lock
    ...
}
```

Заклучи и пиши там

Отворете за писане или създайте `./testlocks.txt`. Докато заключване на 64 байта от **offset** е неуспешно: получите и отпечатайте информация за процеса на заключване и преместете **offset** с още 64 байта. Преместете **offset** байта от началото на файла и запишете информацията за текущия процес. Изчакайте натискане на клавиш **Enter**. Отключете 64 заключени байта, изведете грешка при неуспех. Затворете файла.

lockfile.c

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

int main(int argc, char * argv[])
{
    char str[64];
    memset(str, 32, 64);
    struct flock fi;
    sprintf(str, "Stored by process %i", getpid());
    int fd = open("testlocks.txt", O_RDWR|O_CREAT);
    fi.l_type = F_WRLCK;
    fi.l_whence = SEEK_SET;
    fi.l_start = 0;
    fi.l_len = 64;
    int off = 0;
    while (fcntl(fd, F_SETLK, &fi) == -1)
    {
        fcntl(fd, F_GETLK, &fi);
        printf("bytes %i - %i blocked by process %i\n", off, off+64, fi.l_pid);
        off += 64;
        fi.l_start = off;
    }
    lseek(fd, off, SEEK_SET);
    write(fd, str, strlen(str));
    getchar();
    fi.l_type = F_UNLCK;
    if (fcntl(fd, F_SETLK, &fi) == -1) printf("ERROR while blocking!\n");
    close(fd);
}
```

Упражнение върху работа с файлове

Упражнете материалите от темата, като реализирате представените по-долу проекти.

File Manipulator

Напишете програма **file-manipulator.c**, която изисква 3 аргумента: **words_count**, **min_length**, **max_length**. Програмата трябва да генерира файл, наречен **file-manipulator.log**, който съдържа **words_count** на брой думи (разделени с интервали). За да генерирате думите създайте функция **random_word_generator** (функцията няма да генерира реални думи, а по-скоро произволни поредици от знаци):

```
char* random_word_generator(int min_length, int max_length);
```

След това програмата трябва да отпечата генерирания файл на екрана.

След като програмата създаде файла **file-manipulator.log** и преди да го запише, тя трябва да заключи първите 100 байта на файла. Когато програмата отпечата генерираните думи, трябва да изчака натискане на клавиша **Enter**, след което да отключи файла. Съответно, ако файлът вече е заключен от друг процес, програмата трябва да уведоми за това и да изчака натискане на клавиша **Enter**, преди да опита отново (за заключване, генериране и записване).

Компилирайте и изпълнете няколко пъти за тест.

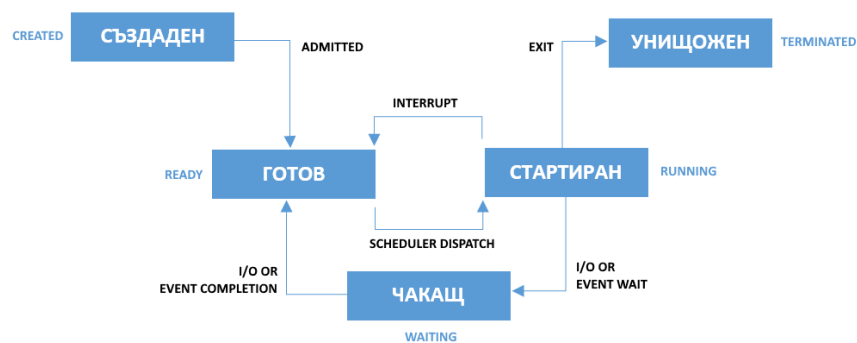
Част 4 – Процеси

- Процеси
- Управление на процесите
- Методи за работа с процеси
- Изпълнение на команда
- Показване на изходния код на файла
- Стартиране на дъщерен процес
- Отпечатване на идентификаторите на процесите
- Изчакване и прекратяване на дъщерен процес
- Изпълнение и отпечатване на състоянието
- Зомбита и проста обработка на сигнали
- Игра на зомбита
- Упражнение върху процеси

Процеси

- Процесите са обектен код в изпълнение: активни, живи, изпълняващи се програми.
- Процесите се състоят от данни (**data**), ресурси (**resources**), състояние (**state**) и виртуализиран компютър (**virtualized computer**).
- Всеки процес е представен от уникален идентификатор на процеса (**pid**).
- Процесът, който ядрото стартира, когато няма други изпълними процеси е процесът на празен ход (**idle process**), неговия идентификатор **pid = 0**.
- Процесът, който създава нов процес, е известен като родителски (**parent**), а новият процес е известен като дъщерен (**child**).
- Всеки процес е собственост на потребител (**user**) и група (**group**).

Възможните състояния на един процес са представени на фигурата по-долу:



Note:

If files are the most fundamental abstraction in a Unix system, processes are the second most fundamental. Processes are object code in execution: active, alive, running programs. But they're more than just object code—processes consist of data, resources, state, and a virtualized computer.

Processes begin life as executable object code, which is machine-runnable code in an executable format that the kernel understands (the format most common in Linux is ELF). The executable format contains metadata, and multiple sections of code and data. Sections are linear chunks of the object code that load into linear chunks of memory. All bytes in a section are treated the same, given the same permissions, and generally used for similar purposes.

The most important and common sections are the text section, the data section, and the bss section. The text section contains executable code and read-only data, such as constant variables, and is typically marked read-only and executable. The data section contains initialized data, such as C variables with defined values, and is typically marked readable and writable. The bss section contains uninitialized global data. Because the C standard dictates default values for C variables that are essentially all zeros, there is no need to store the zeros in the object code on disk. Instead, the object code can simply list the uninitialized variables in the bss section, and the kernel can map the zero page (a page of all zeros) over the section when it is loaded into memory. The bss section was conceived solely as an optimization for this purpose. The name is a historic relic; it stands for block started by symbol, or block storage segment. Other common sections in ELF executables are the absolute section (which contains nonrelocatable symbols) and the undefined section (a catchall).

A process is also associated with various system resources, which are arbitrated and managed by the kernel. Processes typically request and manipulate resources only through system calls.

Resources include timers, pending signals, open files, network connections, hardware, and IPC mechanisms. A process' resources, along with data and statistics related to the process, are stored inside the kernel in the process' process descriptor.

A process is a virtualization abstraction. The Linux kernel, supporting both preemptive multitasking and virtual memory, provides a process both a virtualized processor, and a virtualized view of memory. From the process' perspective, the view of the system is as though it alone were in control. That is, even though a given process may be scheduled alongside many other processes, it runs as though it has sole control of the system. The kernel seamlessly and transparently preempts and reschedules processes, sharing the system's processors among all running processes. Processes never know the difference.

Similarly, each process is afforded a single linear address space, as if it alone were in control of all of the memory in the system. Through virtual memory and paging, the kernel allows many processes to coexist on the

Какво е системно програмиране?

system, each operating in a different address space. The kernel manages this virtualization through hardware support provided by modern processors, allowing the operating system to concurrently manage the state of multiple independent processes.

Управление на процесите

В таблицата по-долу са представени команди за управление на процесите в операционна система Linux:

Команда	Информация
ps	моментна снимка на текущите процеси.
top	показва процесите на Linux.
kill	изпраща сигнал към процес.
pgrep, pkill	търсене или сигнал процеси въз основа на име и други атрибути.
killall	убива процесите по име.
Ctrl-C	убива процеса на преден план.
Ctrl-Z	прекъсва процеса на преден план.
jobs	показва статуса на задачите.
fg	връща прекъснатия процес на преден план.

Повече информация: <https://www.tecmint.com/linux-process-management/>

Пример:

```
$ pgrep -u tecmint top
$ kill 2308
$ pgrep -u tecmint top
$ pgrep -u tecmint glances
$ pkill glances
$ pgrep -u tecmint glances
```


Методи за работа с процеси

В таблицата по-долу са представени методи на C, които се използват при програмиране на процеси в операционна система Linux:

Команда	Информация
getpid()	върща идентификатора pid на извикващия процес.
getppid()	върща идентификатора pid на родителския процес.
exec(), execlp(), execl(), execv(), execvp(), execvpe()	изпълнение на команда.
fork()	създава дъщерен процес.
exit()	причинява нормално прекратяване на процеса.
wait(), waitpid(), waitid()	изчаква процеса да си промени статуса.

Изпълнение на команда

```
#include<unistd.h>

int execl (const char *path, const char *arg, ...);
int execlp (const char *file, const char *arg, ...);
int execl_e (const char *path, const char *arg, ..., char *const envp[]);
int execl_v (const char *path, char *const argv[]);
int execl_vp (const char *file, char *const argv[]);
int execl_ve (const char *filename, char *const argv[], char *const envp[]);
```

Полезни бележки:

- **l** и **v** очертават дали аргументите се предоставят чрез списък или масив (вектор).
- **p** означава, че пътеката на потребителя е търсена за даден файл. Командите, използващи вариантите **p**, могат да определят само име на файл, стига да са разположени в пътя на потребителя.
- **e** отбелязва, че за новия процес се предоставя и нова среда.

Стойности на грешките

При успех на системните извиквания резултат не се връща. При неуспех системните извиквания връщат -1 и задават грешка една от следните стойности:

Стойност	Информация
E2BIG	Общият брой байтове в предоставения аргумент (arg) или средата (envp) е твърде голям.
EACCESS	Процесът няма разрешение за търсене на компонент в пътя; пътя не е обикновен файл; целевият файл не е маркиран като изпълним; или файловата система, на която се намира пътя или файла не е монтиран като изпълним.
EFAULT	Даденият указател не е валиден.
EIO	Възникна грешка вход / изход на ниско ниво (това е лошо).
EISDIR	Крайният компонент в пътя или интерпретаторът е директория.
ELOOP	Системата е срещнала твърде много символични връзки в пътя.
EMFILE	Процесът на извикване достигна своята граница за отворените файлове.
ENFILE	Беше достигнато ограничението за отворените файлове в цялата система.
ENOENT	Целта на пътя или файла не съществува или необходимата споделена библиотека не съществува.
ENOEXEC	Целта на пътя или файла е невалидна програма или е предназначена за различна архитектура.
ENOMEM	Няма достатъчно памет на ядрото за изпълнение на нова програма.
ENOTDIR	Непълн компонент в пътя не е директория.
EPERM	Файловата система, на която се намира пътят или файла, е монтирана nosuid, потребителят не е root и пътя или файл има зададен suid или sgid бит.
ETXTBSY	Целта или файла е отворен за писане от друг процес.

Показване на изходния код на файла

- Инициализирайте променливите
- Изпълнете **nano show-source.c**
- Ако изпълнението не е успешно, изведете съобщение и излезте

show-source.c

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

int main(int argc, char * argv[])
{
    int ret = execl("/bin/nano", "nano", "show-source.c", NULL);
    if (ret == -1)
    {
        printf("Error running nano!\n");
        return EXIT_FAILURE; // -1
    }
    return EXIT_SUCCESS; // 0
}
```

Стартиране на дъщерен процес

```
#include<sys/types.h>
#include<unistd.h>
pid_t fork(void);
pid_t getpid(void);
pid_t getppid(void);
```

- Успешно извикване на **fork()** създава нов процес, идентичен почти във всички аспекти с извикващия го процес.
- Родителският процес получава идентификатора (**pid**) на дъщерния процес, а дъщерния процес получава нула.
- При грешка и дъщерния процес не е създаден, **fork()** връща -1 и грешката (**errno**) може да бъде:

Грешка	Пояснение
EAGAIN	Ядрото не успя да разпредели определени ресурси, като например нов идентификатор (pid) на процес.
ENOMEM	Няма достатъчно памет на ядрото за изпълнение на заявката.

Програмен фрагмен за стартиране на дъщерен процес:

fork.c

Какво е системно програмиране?

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
int main(int argc, char * argv[])
{
    pid_t pid = fork();
    if(pid == -1)
    {
        printf("Error forking a process!\n");
        return EXIT_FAILURE; // -1
    }
    else if(pid == 0)
    {
        printf("Child process!\n");
    }
    else
    {
        printf("Parent process!\n");
    }
    return EXIT_SUCCESS; // 0
}
```

Повече информация: [Linux Programming](#)

Записване на изречения във файл

- Инициализирайте функциите и променливите.
- Създайте дъщерен процес и запазете неговия идентификатор (**pid**).
- Ако стойността на **pid** е положителна, то работи родителски процес. Отпечатайте идентификатора на родителския процес и **ppid**.
- Ако стойността на **pid** е нула, то работи дъщерен процес. Отпечатайте идентификатора на дъщерния процес и **ppid**.
- Ако стойността на **pid** е -1, то е възникнала грешка. Изведете подходящо съобщение и излезте.

print-pids.c

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

void printPIDs(char* name)
{
    printf ("-----The %s process:\n", name);
    printf ("My pid=%d\n", getpid());
    printf ("Parent's pid=%d\n", getppid());
    printf ("-----\n");
}

int main(int argc, char* argv[])
{
    pid_t pid = fork();
    if(pid == -1)
    {
        printf("ERROR!\n");
        return EXIT_FAILURE;
    }
    else if(pid > 0)
    {
        printPIDs("PARENT");
    }
    else if (!pid) // pid==0
    {
        printPIDs("CHILD");
        exit(0);
    }
    return EXIT_SUCCESS;
}
```

Изчакване и прекратяване на дъщерен процес

```
#include<stdlib.h>
void exit(int status);
```

- Извикване на **exit()** изпълнява някои основни стъпки за изключване и след това инструктира ядрото да прекрати процеса.
- Параметърът на състоянието (**status**) се използва за обозначаване на изходното състояние на процеса.
- Когато процес завърши, ядрото изпраща сигнал **SIGCHLD** към родителят.

```
pid_t pid = fork();
if (pid > 0) {
    printPIDs("PARENT");
    wait(&child_status);
}
else if (!pid) {
    printPIDs("CHILD");
    exit(0);
}
else if (pid == -1) {
    printf("ERROR");
    return EXIT_FAILURE;
}
```

Преди да прекрати процеса, библиотеката C изпълнява следните стъпки:

- Извиква всички функции, регистрирани с **atexit()** или **on_exit()**, в обратен ред на тяхната регистрация.
- Изчиства всички стандартни отворени входни-изходни потоци.
- Премахва всички временни файлове, създадени с функцията **tmpfile()**.

Тези стъпки завършват цялата работа, която процесът трябва да направи в потребителското пространство, така че **exit()** извиква системното повикване **_exit()**, за да позволи на ядрото да се справи с останалата част от процеса на прекратяване:

```
#include <unistd.h>
void _exit (int status);
```

Процесът може да бъде прекратен, ако се изпрати сигнал, чието действие по подразбиране е да прекрати процеса. Такива сигнали са **SIGTERM** и **SIGKILL**.

Когато процес завърши, ядрото изпраща сигнал **SIGCHLD** към родителския процес. По подразбиране този сигнал се игнорира и родителят не предприема никакви действия. Процесите могат да изберат да обработват този сигнал, чрез системните извиквания **signal()** или **sigaction()**.

```
#include<sys/types.h>
#include<sys/wait.h>
pid_t wait(int status);
int WIFEXITED(status);
int WIFSIGNALED(status);
int WIFSTOPPED(status);
int WIFCONTINUED(status);
int WEXITSTATUS(status);
int WTERMSIG(status);
int WSTOPSIG(status);
int WCOREDUMP(status);
```

- **wait()** връща стойността на идентификатора (**pid**) на дъщерен процес или -1 при грешка.
- Ако никакъв дъщерен процес не е прекратен, извикването блокира, докато дъщерен процес приключи.
- **WIFEXITED** връща вярно (true), ако процесът завърши нормално.
- При нормално прекратяване **WEXITSTATUS** осигурява осем бита, които се предават на **_exit**.
- **WIFSIGNALED** връща вярно (true), ако сигнал е причинил прекратяването на процеса.
- В случай на прекратяване от сигнал **WTERMSIG** връща номера на този сигнал.
- В случай на прекратяване от сигнал, **WCOREDUMP** връща вярно (true), ако процесът натоварва ядро в отговор на получаването на сигнала.
- **WIFSTOPPED** и **WIFCONTINUED** връщат вярно (true), ако процесът е бил спрял или продължен.
- Ако **WIFSTOPPED** е вярно (true), **WSTOPSIG** предоставя номера на сигнала, който е спрял процеса.

При възникване на грешка има две възможни стойности:

Грешка	Информация
ECHILD	Извикваният процес няма дъщерни процеси.
EINTR	Извикването проклучи по-рано поради получен сигнал.

Изпълнение и отпечатване на състоянието

Проверете дали разполагаме с достатъчно на брой аргументи. Създайте дъщерен процес и запазете неговия идентификатор (pid). Родителският процес чака дъщерният процес и излиза при грешка. При успех се отпечатва състоянието на прекратяването. Дъщерният процес изпълнява командата в аргументите.

exec-stat.c

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
int main(int argc, char * argv[])
{
    int pid, status;
    if (argc < 2) {
        printf("Usage: %s command, [arg1 [arg2]...]\n", argv[0]);
        return EXIT_FAILURE;
    }
    printf("Starting %s...\n", argv[1]);
    pid = fork();
    if (pid == 0) {
        execvp(argv[1], &argv[1]);
        perror("execvp");
        return EXIT_FAILURE; // Never get there normally
    }
    else {
        if (wait(&status) == -1) {
            perror("wait");
            return EXIT_FAILURE;
        }
        if (WIFEXITED(status)) printf("Child terminated normally with exit code %d\n", WEXITSTATUS(status));
        if (WIFSIGNALED(status)) printf("Child was terminated by a signal %d\n", WTERMSIG(status));
        if (WCOREDUMP(status)) printf("Child dumped core\n");
        if (WIFSTOPPED(status)) printf("Child was stopped by a signal %d\n", WSTOPSIG(status));
    }
    return EXIT_SUCCESS;
}
```

Зомбита и проста обработка на сигнали

Когато дъщерен процес умре преди родителският процес, ядрото го поставя в специално състояние наречено зомби (**zombie**). Процесът в това състояние чака родителски процес да се допита до неговия статус и едва след това дъщерният процес преставя да съществува като зомби. Ако родителският процес никога не запита за състоянието на дъщерният процес, тогава зомбито се превръща в призрак (**ghost**), което е много лоша практика. Ако родителският процес завърши преди дъщерните му процеси, то техен родител става началният процес. Началният процес, от своя страна, периодически изчаква всички свои дъщерни процеси, като по този начин гарантира, че никое от тях няма да остане зомби.

Изчакване на процес:

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

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

Обработка на сигнали:

```
#include<signal.h>

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

struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);

    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
```

Какво се случва обаче, ако родителският процес умре преди дъщерният процес или ако той умре, преди да има възможност да изчака своите зомбита? Всеки път, когато даден процес се прекратява, ядрото на Линукс преглежда списъка на дъщерните процеси и ги възпроизвежда на началният процес (идентификатор на процеса `pid = 1`). Това гарантира, че никой процес няма да остане без родител. Началният процес, от своя страна, периодически изчаква всички свои дъщерни процеси, като по този начин гарантира, че никое от тях няма да остане зомби за твърде дълго време. Въпреки че това все още се счита за добра практика, тази предпазна мярка означава, че краткотрайните процеси не трябва да се притесняват прекомерно да чакат всичките си деца.

Игра на зомбита

Дефинирайте обработчик на сигнал за прекратяване на изпълнението. Инициализирайте променливи за използване на обработчик на сигнал. Задайте действие при **SIGCHLD**, изход при грешка. Направете 10 пъти следното: създайте дъщерен процес и запазете неговия идентификатор (**pid**). Родителският процес отпечатва идентификатора на дъщерния процес. Дъщерният процес отпечатва съобщение и излиза.

zombie-test.c

```
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

static void sigchld_hdl (int sig)
{
    while (waitpid(-1, NULL, WNOHANG) > 0) {}
}

int main (int argc, char *argv[])
{
    struct sigaction act;
    memset (&act, 0, sizeof(act));
    act.sa_handler = sigchld_hdl;
    if (sigaction(SIGCHLD, &act, 0)) {
        perror ("sigaction");
        return 1;
    }
    for (int i = 0; i < 10; i++) {
        int pid = fork();
        if (pid == 0) {
            printf("I will leave no zombie\n");
            exit(0);
        } else printf("Created a process with the PID %i\n", pid);
    }
    while (1) sleep(1);
    return 0;
}
```

Упражнение върху процеси

Упражнете материалите от темата, като реализирате представеният по-долу проект.

Temp File Generator

Напишете програма **temp-file-generator.c**, която изисква 1 аргумент: **files_count**. Програмата трябва да генерира **files_count** на брой файлове, наречени "**temp-file-NN.tmp**" (където номерът *NN* е число от 1 до *files_count*). Всеки файл е създаден от отделен процес и съдържа 1000 пъти идентификатора на съответния процес като текст (*pid = 1001 трябва да отнеме 4 байта на ID*). Родителският процес трябва да се уведомява след създаването на всеки файл.

Temp File Generator Plus

Напишете нова програма **temp-file-generator-plus.c**, като подобрите предишната програма, така че процесите да пишат само в един файл, където последователностите от идентификатори на процеси не се припокриват (имаме 1000 пъти *pid_file_1*, след това 1000 пъти *pid_file_2* ... след това 1000 пъти *pid_files_count*).

Част 5 – Комуникация между процеси

- Методи за комуникация между процеси
- Сигнали
- Най-важните сигнали
- Управление на сигналите
- Обработка на сигнал
- Изчакване на сигнали
- Разглеждане на сигнали
- Показване на информация за сигналите
- Reentrancy
- Анонимни и наименувани тръби
- Работа с тръби
- Използване на анонимни тръби
- Използване на именувани тръби
- Упражнение върху комуникация между процеси

01_ipc_methods

Структура	Информация
File	Запис запазен на диск или запис, синтезиран при поискване от файлов сървър, достъпен от множество процеси.
Signal	Системно съобщение, изпратено от един процес на друг, което обикновено не се използва за прехвърляне на данни, а вместо това се използва за отдалечено управление на процес.
Socket	Данните се изпращат през мрежов интерфейс или към различен процес на същия компютър или на друг компютър в мрежата. Поточно ориентирани (TCP, данни, записани чрез сокет и изискващи форматиране, за да се запазят границите на съобщенията) или по-рядко съобщения (UDP, SCTP).
Unix domain socket	Подобно на интернет сокет, но цялата комуникация се осъществява в рамките на ядрото. Домейн сокетите използват файловата система като адресно пространство. Процесите посочват домейн сокета като инопд и множество процеси могат да комуникират с един сокет.
Message queue	Поток от данни, подобен на сокета, но който обикновено запазва границите на съобщенията. Обикновено се изпълняват от операционната система и позволяват множество процеси да четат и записват в опашката на съобщенията, без да са директно свързани помежду си.
Pipe	Еднопосочен канал за данни. Данните, записани в края на запис на тръбата, се буферират от операционната система, докато се прочетат от четящия край на тръбата. Двупосочните потоци от данни между процесите могат да бъдат постигнати чрез създаване на две тръби, използващи стандартен вход и изход.
Named pipe	Тръба, изпълнявана чрез файл в файловата система, вместо стандартния вход и изход. Няколко процеса могат да четат и записват във файла като буфер за IPC данни.
Shared memory	На множество процеси се дава достъп до един и същ блок памет, който създава общ буфер, за да могат процесите да комуникират помежду си.

Повече информация: [Inter Process Communication](#)

Сигнали

Сигналите са механизъм за едностранни асинхронни известия. Сигналът може да бъде изпратен от ядрото до процес, от процес до друг процес или от процес до себе си. Сигналите обикновено предупреждават за процес за някое събитие, като например неизправност при сегментиране или потребител, който натиска клавишна комбинация **Ctrl-C**.

Линукс ядрото изпълнява около 30 сигнала (точният брой зависи от архитектурата).

Всеки сигнал е представен чрез цифрова константа и текстово име.

Например **SIGHUP**, използвано за сигнализиране, че е настъпило прекъсване на терминала, има стойност 1 на архитектурата i386.

С изключение на **SIGKILL** (което винаги прекратява процеса) и **SIGSTOP** (което винаги спира процеса) процесите могат да контролират какво се случва, когато получат сигнал.

Те могат да приемат действието по подразбиране, което може да бъде прекратяване на процеса, прекратяване на процеса и разтоварване на ядрото, спиране на процеса или нищо в зависимост от сигнала.

Алтернативно, процесите могат да избират изрично да игнорират или да обработват сигнали. Игнорираните сигнали мълчат.

Обработените сигнали причиняват изпълнението на функцията за обработка на сигнал от потребителя. Програмата преминава към тази функция веднага щом сигналът бъде приет и (когато обработващият сигнал се върне), управлението на програмата се възобновява при предишната прекъсната инструкция.

Какво е системно програмиране?

Най-важните сигнали

сигнал	информация	резултат
SIGHUP (1)	Контролният терминал на процеса е затворен (най-често потребителят е излязъл).	Terminate
SIGINT (2)	Потребителят използва комбинация за прекъсване Ctrl-C.	Terminate
SIGABRT (6)	Функцията abort() изпраща този сигнал до процеса, който го предизвиква. След това процесът се прекратява и се генерира основен файл.	Terminate with core dump
SIGKILL (9)	Този сигнал се изпраща от системното повикване kill(), той съществува, за да осигури сигурен начин за безусловното убиване на процес.	Terminate
SIGSEGV (11)	Този сигнал, чието име произтича от нарушаване на сегментирането, се изпраща в процес, когато се опитва невалиден достъп до паметта.	Terminate with core dump
SIGTERM (15)	Този сигнал се изпраща само чрез kill() и позволява на потребителя грациозно да прекрати процес (действие по подразбиране).	Terminate
SIGCHLD (17)	Когато процес завърши или спре, ядрото изпраща този сигнал към родителя. Обработващият този сигнал обикновено извиква wait(), за да определи идентификатора на дъщерния процес (pid) и кода за изход.	Ignored
SIGCONT (18)	Ядрото изпраща този сигнал на процес, когато процесът се възобновява след спиране (от SIGSTOP).	Ignored
SIGSTOP (19)	Този сигнал се изпраща само чрез kill(), той безусловно спира процес и не може да бъде уловена или пренебрегната.	Stop
SIGTSTP (20)	Ядрото изпраща този сигнал до всички процеси от групата на преден план, когато потребителят използва комбинация Ctrl-Z.	Stop

сигнал	информация	резултат
SIGIO (29)	Този сигнал се изпраща, когато се генерира асинхронно входно-изходно събитие.	Terminate
SIGUSR1(10), SIGUSR2(12)	Тези сигнали са налице за цели определени от потребителя; ядрото никога не ги използва. Процесите могат да използват SIGUSR1 и SIGUSR2.	Terminate

Signals in Linux

Execute following terminal command to view signal list:

```
kill -l
```

Result:

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL	5) SIGTRAP
6) SIGABRT	7) SIGBUS	8) SIGFPE	9) SIGKILL	10) SIGUSR1
11) SIGSEGV	12) SIGUSR2	13) SIGPIPE	14) SIGALRM	15) SIGTERM
16) SIGSTKFLT	17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO	30) SIGPWR
31) SIGSYS	34) SIGRTMIN	35) SIGRTMIN+1	36) SIGRTMIN+2	37) SIGRTMIN+3
38) SIGRTMIN+4	39) SIGRTMIN+5	40) SIGRTMIN+6	41) SIGRTMIN+7	42) SIGRTMIN+8
43) SIGRTMIN+9	44) SIGRTMIN+10	45) SIGRTMIN+11	46) SIGRTMIN+12	47) SIGRTMIN+13
48) SIGRTMIN+14	49) SIGRTMIN+15	50) SIGRTMAX-14	51) SIGRTMAX-13	52) SIGRTMAX-12
53) SIGRTMAX-11	54) SIGRTMAX-10	55) SIGRTMAX-9	56) SIGRTMAX-8	57) SIGRTMAX-7
58) SIGRTMAX-6	59) SIGRTMAX-5	60) SIGRTMAX-4	61) SIGRTMAX-3	62) SIGRTMAX-2
63) SIGRTMAX-1	64) SIGRTMAX			

https://dsa.cs.tsinghua.edu.cn/oj/static/unix_signal.html

Execute following terminal command to view signal(7) manual:

```
man 7 signal
```

<https://man7.org/linux/man-pages/man7/signal.7.html>

Управление на сигнали

метод	описание
sigemptyset()	инициализира празно множество от сигнали, като всички сигнали са изключени от множеството.
sigfillset()	инициализира пълно множество от сигнали, включващо всичките сигнали.
sigaddset()	добавя сигнал към множество.
sigdelset()	премахва сигнал от множество.
sigprocmask()	извлича и/или променя сигналната маска на извикващата нишка.
sigaction()	променя действието на сигнала.
sigwait()	чака сигнал.
strsignal()	върща низ описващ сигнала.

```
#include <signal.h>

int sigemptyset (sigset_t *set);
int sigaddset (sigset_t *set, int signo);
int sigprocmask (int how, const sigset_t *set, sigset_t *oldset);

int sigaction (int signo, const struct sigaction *act, struct sigaction *oldact);

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

sigemptyset() инициализира наборът от сигнали, зададен от комплекта, като го маркира празен (всички сигнали са изключени от комплекта).

sigaddset() добавя сигнал към комплекта от сигнали, зададен от комплекта, докато **sigdelset()** премахва знака от набора от сигнали, зададен от комплекта. И двете връщат 0 при успех, или -1 при грешка, в който случай грешката е настроена на код за грешка **EINVAL**, което означава, че знакът е невалиден идентификатор на сигнала.

Поведението на **sigprocmask()** зависи от стойността на едно от следните:

Поле	Описание
SIG_SETMASK	Задава се сигнална маска на извикващия процес.
SIG_BLOCK	Сигналите се добавят към сигналната маска на извикващият процес.
SIG_UNBLOCK	Сигналите се премахват от сигналната маска на извикващия процес.

Структура **sigaction**:

Поле	Описание
SA_HANDLER	Адрес на манипулатора SIG_IGN или SIG_DFL
SA_MASK	Сигнали за блокиране
SA_FLAGS	Допълнителни флагове, като SA_RESETHAND
SA_RESETHAND	Позволява режим "еднократно". Поведението на дадения сигнал се връща към стандартното, след като обработващият сигнал се върне.

Извикването на **sigaction()** променя поведението на сигнала, идентифициран от **signo**, което може да бъде всяка стойност освен тези, свързани със **SIGKILL** и **SIGSTOP**. Ако **act** не е NULL, системното извикване променя текущото поведение на сигнала, както е посочено в **act**. Ако **oldact** не е NULL, системното извикване запазва предишното поведение на даден сигнал (или текущия, ако **act** е NULL).

Обработка на сигнал

- Дефинирайте функцията за обработка на сигнали **SIGTERM**
- Иницирайте сигнал, като само **SIGHUP** е включен и блокирайте обработката на сигнали
- Задайте обработчик на събития за **SIGTERM**
- Отпечатайте идентификатора на процеса (**pid**)
- Влезте в безкраен цикъл

signal-handler.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

// Signal Handler
void signal_handler(int i)
{
    printf ("Terminating\n");
    exit(EXIT_SUCCESS); // 1
}

// Main Program
int main(int argc, char ** argv)
{
    // Signal Structure Initialization
    struct sigaction sa;
    sigset_t newset;
    sigemptyset(&newset);
    sa.sa_flags = 0;
    sa.sa_handler = signal_handler;
    sigaction(SIGINT, &sa, NULL); // Ctrl+C

    // Print
    printf("My pid is %i\n", getpid());
    printf("Waiting...\n");

    // Forever
    while(1) sleep(1);

    return EXIT_FAILURE; // -1
}
```

Изчакване на сигнали

```
#include<signal.h>

int sigwait(const sigset_t *set, int *sig);
```

Функцията **sigwait()** спира изпълнението на извикващата нишка, изчаквайки един от сигналите да стане текущ. Функцията приема сигнала (премахва го от чакащия списък със сигнали) и връща номера на сигнала в **sig**. При успех **sigwait()** връща 0, при грешка връща положително число, номер на грешката.

Пример за изчакване на сигнал

Дефинирайте функцията за обработка на сигнали **SIGTERM**. Иницирайте сигнал, като само **SIGHUP** е включен и блокирайте обработката на сигнали. Задайте обработчик на събития за **SIGTERM**. Отпечатайте идентификатора на процеса (**pid**). Изчакайте входящ сигнал **SIGHUP** и изведете подходящо съобщение. Ако възникне грешка преустановете програмата.

signal-wait.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

// Main program
int main(int argc, char ** argv)
{
    // Signal Structure Initialization
    int sig;
    struct sigaction sa;
    sigset_t newset;
    sigemptyset(&newset);
    sigaddset(&newset, SIGINT); // Ctr + C
    sigprocmask(SIG_BLOCK, &newset, 0);
    sigaction(SIGTERM, &sa, 0);

    // Print
    printf("My pid is %i\n", getpid());
    printf("Waiting...\n");

    // Forever
    while(!sigwait(&newset, &sig))
    {
        printf("SIGINT recieved\n");
        return EXIT_SUCCESS; // 0
    }

    return EXIT_FAILURE; // -1
}
```


Разглеждане на сигнали

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

char *strsignal(int sig);

extern const char * const sys_siglist[];

int sigfillset(sigset_t *set);

int sigdelset(sigset_t *set, int signum);
```

Извикване на **strsignal()** връща указател към описание на сигнала, даден от **signo**.

sys_siglist е масив от низове, съдържащи имената на сигналите, поддържани от системата, индексирани с номер на сигнала.

Показване на информация за сигналите

- Дефинирайте функцията за обработка на сигнали **SIGTERM**
- Иницирайте сигнално множество, като само **SIGTERM** е изключен
- Задайте обработчик на събития за **SIGTERM**
- Отпечатайте идентификатора на процеса (**pid**)
- Изчакайте и изведете подходящи съобщения за входящите сигнали
- Ако възникне грешка преустановете програмата

signal-info.c

```
#include<stdio.h>
#include<stdlib.h>
#include<signal.h>
#include<string.h>
#include<unistd.h>

void signal_handler(int sig)
{
    printf("Signal %i: %s\n", sig, strsignal(sig));
    exit(EXIT_SUCCESS); // 0
}

int main(int argc, char* argv[])
{
    int sig;
    struct sigaction sa;
    sigset_t sset;
    sigfillset(&sset);
    sigdelset(&sset, SIGTERM);
    sigprocmask(SIG_SETMASK, &sset, 0);
    sa.sa_handler = signal_handler;
    sigaction(SIGTERM, &sa, 0);

    printf("My pid is %i\n", getpid());

    while(!sigwait(&sset, &sig))
    {
        printf("Signal %i: %s\n", sig, strsignal(sig));
    }

    return EXIT_FAILURE;
}
```

Reentrancy

Когато ядрото изпрати сигнал, процесът обикновено изпълнява някакъв код. Това може да се случи в средата на важна за него операция, която ако бъде прекъсната, ще постави процеса в несъгласувано състояние. Например структурата данни с която процесът работи е само наполовина актуализирана или изчислението е изпълнено само частично. Процесът дори може да обработва друг сигнал.

Обработчика на сигнал (**signal handler**) не знае точно кой код се изпълнява от процеса в момента на поява на сигналът, това може да се случи по всяко време. Поради тази причина е много важно всеки обработчик на сигнал (**signal handler**), който напишете, да бъде много внимателен относно действията, които извършва и данните с които работи.

Обработчиците на сигнали (**signals handlers**), трябва да се погрижат да не правят предположения за това, какво прави процесът, когато бъде прекъснат. По-специално, те трябва да са предпазливи при модифициране на глобални и споделени данни. Като цяло, добра практика е обработчика на сигнал никога да не работи с глобални данни.

Какво ще кажете за системните извиквания или другите библиотечни функции? Какво ще стане, ако процесът е по средата на писане във файл или разпределяне на памет, и обработващ сигнал пише в същия файл или извика **malloc()**? Какво ще стане, ако при поява на сигнал, процесът е по средата на извикване на функция, която използва статичен буфер, например като **strsignal()**?

Поради всички изброени причини е много важно всеки обработчик на сигнал (**signal handler**) от процеса да е **reentrant** функция.

reentrant е функция която е безопасно да се извиква от самата нея (или едновременно от друга нишка в същия процес). За да се квалифицира като **reentrant**, функцията не трябва да манипулира статични данни, трябва да манипулира само данни, предоставени от стека, или данни, предоставени ѝ от извикващия функцията, и не трябва да изпълнява функция която не е **reentrant**.

Стандартните C функции, безопасни за употреба са:

accept, access, aio_error, aio_return, aio_suspend, alarm, bind, cfgetispeed, cfgetospeed, cfsetispeed, cfsetospeed, chdir, chmod, chown, clock_gettime, close, connect, creat, dup, dup2, execl, execve, _Exit & _exit, fchmod, fchown, fcntl, fdasync, fork, fpathconf, fstat, fsync, ftruncate, getegid, geteuid, getgid, getgroups, getpeername, getpgrp, getpid, getppid, getsockname, getsockopt, getuid, kill, link, listen, lseek, lstat, mkdir, mkfifo, open, pathconf, pause, pipe, poll, posix_trace_event, pselect, raise, read, readlink, recv, recvfrom, recvmsg, rename, rmdir, select, sem_post, send, sendmsg, sendto, setgid, setpgid, setsid, setsockopt, setuid, shutdown, sigaction, sigaddset, sigdelset, sigemptyset, sigfillset, sigismember, signal, sigpause, sigpending, sigprocmask, sigqueue, sigset, sigsuspend, sleep, socket, socketpair, stat, symlink, sysconf, tcdrain, tcflow, tcflush, tcgetattr, tcgetpgrp, tcsendbreak, tcsetattr, tcsetpgrp, time, timer_getoverrun, timer_gettime, timer_settime, times, umask, uname, unlink, utime, wait, waitpid, write.

Анонимни и наименувани тръби

Анонимни тръби

Анонимната тръба (*Термин на английски език: Anonymous pipe*) е обикновен **FIFO** комуникационен канал, който може да се използва за едноточна междупроцесна комуникация (IPC). Обикновено родителската програма отваря анонимни тръби и създава нов процес, който наследява другите краища на тръбите. Анонимната тръба трае само докато трае процесът.

Наименувани тръби

Наименуваната тръба (*Термин на английски език: Named pipe*) е продължение на концепцията за традиционните анонимни тръби. Наименуваната тръба може да бъде идентифицирана с име и да се покаже като файл в системата. Наименуваната тръба може да продължи да съществува толкова дълго, колкото компютърната система е включена, а не само по време живота на процеса. Наименуваната тръба може да бъде изтрита, ако вече не се използва.

Работа с тръби

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

pipe() - създава анонимна тръба.

- Масивът **pipefd** се използва за да върне два файлови дескриптора, сочещи към краищата на тръбата.
- **pipefd[0]** се отнася за четене в края на тръбата
- **pipefd[1]** се отнася за писане в края на тръбата.

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

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

mkfifo() - създава наименувана тръба (**FIFO**).

- **mode** указва правата за **FIFO** структурата. Той се модифицира от метода **umask** по обичайния начин: разрешенията на създадения файл са **(mode & ~umask)**.

Използване на анонимни тръби

- Отворете тръба (**pipe**) и запазете входно/изходните файлови дескриптори (**I/O FD**)
- Създайте дъщерен процес (**fork**)
- Родителски процес: затворете входният файлов дескриптор (**input/read**)
 - Запишете съобщение на изходният файлов дескриптор (**output/write**)
 - Затворете изходният файлов дескриптор (**output/write**)
- Дъщерен процес: затворете изходният файлов дескриптор (**output/write**)
 - Прочетете съобщение от входният файлов дескриптор (**input/read**)
 - Отпечатайте полученото съобщение
 - Затворете входният файлов дескриптор (**input/read**)

pipe-pass-message.c

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

int main(int argc, char ** argv)
{
    int pipefd[2];
    int pd = pipe(pipefd);
    if(pd == -1)
    {
        printf("Error pipe!\n");
        return EXIT_FAILURE;
    }

    int pid = fork();
    if(pid == -1)
    {
        printf("Error fork!\n");
        return EXIT_FAILURE;
    }

    if(pid > 0)
    {
        char* str = "Hello World!";
        printf("Parent pid %i\nSend: %s\n", getpid(), str);
        close(pipefd[0]);
        write(pipefd[1], (void*)str, strlen(str)+1);
        close(pipefd[1]);
    }
    else
    {
        printf("Child pid %i\nReceive: ", getpid());
        char buf[1024];
        int len;
        close(pipefd[1]);
        while((len = read(pipefd[0], buf, 1024)) != 0)
        {
            printf("%s", buf);
        }
        close(pipefd[0]);
        printf("\n");
    }

    return EXIT_SUCCESS;
}
```


Използване на именовани тръби

Представената двойка програми реализира чат базирано клиент/сървър приложение, което демонстрира използването на именовани тръби.

pipe-chat-server.c

Създайте файл на име **./pipe-chat-fifo**. Отворете за писане **./pipe-chat-fifo** в случай на грешка излезте. Стартирайте цикъл, който работи докато получим **"q"** от клавиатурата. Прочетете символ (**key**) и го запишете във файла. Ако **key** е Enter използвайте **flush**. Затворете и изтрийте файла.

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

#define FIFO_FILE "./pipe-chat-fifo"

int main(int argc, char ** argv)
{
    mkfifo(FIFO_FILE, 0600);
    FILE* f = fopen(FIFO_FILE, "w");
    if(f == NULL)
    {
        printf("Error open file!\n");
        return EXIT_FAILURE;
    }

    char key;
    do
    {
        key = getchar();
        fputc(key, f);
        if(key == 10) fflush(f);
    }
    while(key != 'q');

    fclose(f);
    unlink(FIFO_FILE);
    return EXIT_SUCCESS;
}
```

pipe-chat-client.c

Отворете за четене файл на име **./pipe-chat-fifo**. Стартирайте цикъл, който работи докато получим **"q"** от клавиатурата. Прочетете символ (**key**). Изведете получения символ в конзолата. Затворете и изтрийте файла.

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

#define FIFO_FILE "./pipe-chat-fifo"

int main(int argc, char ** argv)
{
    FILE* f = fopen(FIFO_FILE, "r");
    if(f == NULL)
    {
        printf("Error open file!\n");
        return EXIT_FAILURE;
    }

    char key;
    do
    {
        key = fgetc(f);
        putchar(key);
    }
    while(key != 'q');

    fclose(f);
    unlink(FIFO_FILE);
    return EXIT_SUCCESS;
}
```

Упражнение върху комуникация между процеси

Упражнете материалите от темата, като реализирате представените по-долу проекти.

Reverse Encryptor Decryptor

Нека дефинираме "Reverse Encryption" (**RE**) като техника за защита на текстови съобщения, при която текста на съобщенията се обръща наобратно, преди те да бъдат изпратени по-нататък. Напишете програма **reverse-encryptor-decryptor.c**, която има два процеса, комуникиращи посредством анонимна тръба (**anonymous pipe**).

За двата процеса да валидни следните правила:

- **Родителски процес** трябва да чете текст от клавиатурата, след това да приложи техниката RE и да изпрати кодираното съобщение на **Дъщерен процес**.
- **Дъщерен процес** трябва да изчака входящите съобщения, да отпечата кодираните съобщения на екрана, след това да ги декодира и да ги съхрани във файл **reverse-encryptor-decryptor.log**.

Encrypted Chat

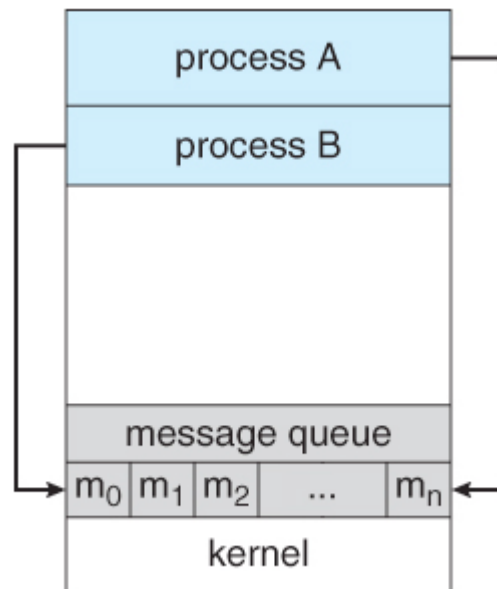
Напишете двойка програми за сървър (**encrypted-chat-server.c**) и клиент (**encrypted-chat-client.c**). Сървърът трябва да имитира поведението на **Родителски процес** от предишната задача, а клиента съответно на **Дъщерен процес**. Програмите трябва да използват наименувана тръба (**named pipe**) като метод за комуникация.

Част 6 - Синхронизация на комуникацията между процесите

- [Опашка за съобщения](#)
- [Библиотеки за работа със съобщения](#)
- [Определяне на общите данни](#)
- [Сървър за съобщения](#)
- [Клиент за съобщения](#)
- [Споделена памет](#)
- [Библиотеки за работа със споделена памет](#)
- [Определяне на общите данни](#)
- [Сървър за памет](#)
- [Клиент за памет](#)
- [Семафори](#)
- [Библиотеки за работа със семафори](#)
- [Определяне на общите данни](#)
- [Семафор Сървър](#)
- [Семафор Клиент](#)
- [Упражнение за синхронизация на комуникация между процеси](#)

Опашка за съобщения

Опашките за съобщения могат да бъдат описани като вътрешен свързан списък в адресното пространство на ядрото. Съобщенията могат да бъдат изпратени в опашката и извлечени от опашката по няколко различни начина. Всяка опашка за съобщения е уникално идентифицирана от IPC идентификатор.



Повече информация: [Message Queues](#)

Библиотеки за работа със съобщения

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg>

int msgget(key_t key, int msgflg);
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Системното извикване **msgget()** връща System V идентификатор на опашката за съобщения свързана със стойността на аргумента **key**.

Системните извиквания **msgsnd()** и **msgrcv()** се използват, съответно, за изпращане и получаване на съобщения от System V опашка за съобщения. Извикващият процес трябва да има разрешение за запис в опашката на съобщенията, за да изпрати съобщение и да има разрешение за четене, за да получи съобщение.

msgctl() и **cmd = IPC_RMID** премахва опашка на съобщенията.

Определяне на общи данни

- Определете **KEY** за уникалният идентификационен номер на опашката.
- Определете **MAXLEN** за максимална дължина на съобщението.
- Определете **msg_1_t** за клиентското съобщение.
- Определете **msg_1_2** за отговора от страна на сървъра.

message-types.h

```
#ifndef MSG_TYPES
#define MSG_TYPES

#define KEY 1274
#define MAXLEN 512

struct msg_1_t
{
    long mtype;
    int snd_pid;
    char body[MAXLEN];
};

struct msg_2_t
{
    long mtype;
    int snd_pid;
    int rcv_pid;
    char body[MAXLEN];
};

#endif
```

Сървър за съобщения

- Създайте опашка за съобщения с даден ключ **KEY**.
- Изчакайте съобщение от тип **msg_1_t** от опашката.
- Отпечатайте данните за получените съобщения.
- Подгответе и изпратете отговор от тип **msg_2_t**.
- Изчакайте потвърждение **msg_1_t**.
- Премахнете опашката за съобщения.

message-server.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#include "message-types.h"

int main(int argc, char ** argv)
{
    int msgid = msgget(KEY, 0777 | IPC_CREAT);
    if(msgid == -1)
    {
        printf("Error msgget.\n");
        return EXIT_FAILURE;
    }

    struct msg_1_t message1;
    msgrcv(msgid, &message1, sizeof(struct msg_1_t), 1, 0);
    printf("Client [pid = %i] send: %s\n", message1.snd_pid, message1.body);

    struct msg_2_t message2;
    message2.mtype = 2;
    message2.snd_pid = getpid();
    message2.rcv_pid = message1.snd_pid;
    strcpy(message2.body, "OK\n");
    msgsnd(msgid, &message2, sizeof(struct msg_2_t), 0);

    msgrcv(msgid, &message1, sizeof(struct msg_1_t), 1, 0);
    printf("Client [pid = %i] send: %s\n", message1.snd_pid, message1.body);

    msgctl(msgid, IPC_RMID, 0);
    return EXIT_SUCCESS;
}
```


Клиент за съобщения

- Отварете опашка за съобщения с даден ключ **KEY**.
- Изведете съобщение и излезте от програмата, в случай на неуспех.
- Прочетете тялото на съобщението от клавиатурата и задайте другите параметри на **msg_1_t**.
- Изпратете съобщение **msg_1_t** в опашката.
- Изчакайте **msg_2_t** и го отпечатайте.
- Изпратете потвърждение **msg_1_t**.

message-client.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include "message-types.h"

int main(int argc, char ** argv)
{
    int msgid = msgget(KEY, 0666);
    if(msgid == -1)
    {
        printf("Error msgget.\n");
        return EXIT_FAILURE;
    }

    int i = 0;
    struct msg_1_t message1;
    while( (i < (MAXLEN - 1)) &&
           ((message1.body[i++] = getchar()) != '\n') );
    message1.body[i] = '\0';
    message1.mtype = 1;
    message1.snd_pid = getpid();
    msgsnd(msgid, &message1, sizeof(struct msg_1_t), 0);
    printf("Sended: %s\n", message1.body);

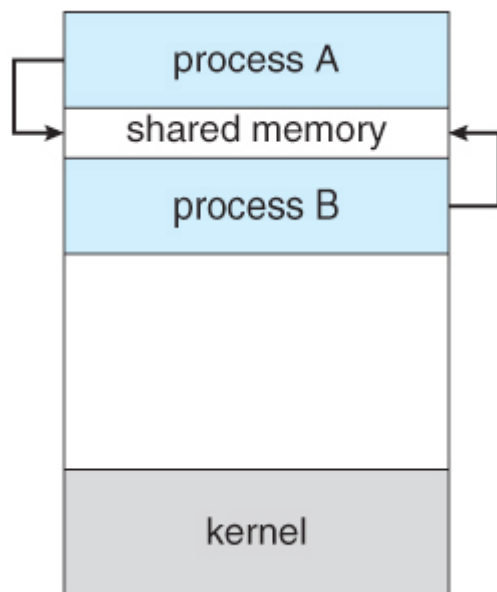
    struct msg_2_t message2;
    msgrcv(msgid, &message2, sizeof(struct msg_2_t), 2, 0);
    printf("Server [pid = %i] responded: %s\n", message2.snd_pid, message2.body);

    message1.mtype = 1;
    msgsnd(msgid, &message1, sizeof(struct msg_1_t), 0);

    return EXIT_SUCCESS;
}
```

Споделена памет

Споделената памет може да се опише като картографиране на област (сегмент) от паметта, която ще бъде картирана и споделена чрез повече от един процес. Това е най-бързата форма на между-процесна комуникация (IPC), тъй като няма посредничество - информацията се пренася директно от сегмент от паметта в адресното пространство на процеса на повикване. Сегментът може да бъде създаден от един процес и след това да бъде записван и четен от произволен брой процеси.



Повече информация: [Shared Memory](#)

Библиотеки за работа със споделена памет

```
#include<sys/types.h>
#include<sys/shm.h>
#include<sys/ipc.h>

key_t ftok(const char *pathname, int proj_id);
int shmget(key_t key, size_t size, int shmflg);
void *shmat(int shmid, const void *shaddr, int shmflg);
int shmdt(const void *shmaddr);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- **ftok()** = преобразува име на път и идентификатор на проект към ключ на System V IPC.
- **shmget()** = разпределя сегмент на System V споделена памет .
- **shmat()** = придава сегмента на System V споделена памет, идентифициран от **shmid*, в адресното пространство на извикващия процес.
- **shmdt()** = отделя сегмента на споделената памет, разположен на адреса, определен от **shmaddr** от адресното пространство на извикващия процес.
- **shmctl()** с cmd = IPC_RMID - маркира сегмента, който ще бъде унищожен.

Определяне на общи данни

Определете идентификационен номер на опашката **FTOK_FILE**.
Дефинирайте структурата на споделената памет **memory_block**:

- **server_lock** е 1, когато сървърът използва паметта
- **client_lock** е 1, когато клиентът използва паметта
- **turn** е 0, когато чакате клиентско съобщение
- **turn** е 1, когато чакате съобщение от сървъра
- **readlast** е 0, когато клиентът получи последно съобщение
- **readlast** е 1, когато сървър получи последно съобщение
- **string** държи текущото съобщение

shared-memory-types.h

```
#ifndef SHMEM_TYPES
#define SHMEM_TYPES

#define FTOK_FILE "./shared-memory-server"

#define MAX_LEN 512
#define FREE 1
#define BUSY 0
#define SERVER 1
#define CLIENT 0

struct memory_block
{
    int server_lock;
    int client_lock;
    int turn;
    int read_last;
    char string[MAX_LEN];
};

#endif
```

Сървър за памет

Създайте `./shared-memory-server`, излезте при грешка. Разпределете споделена памет. Конфигурирайте паметта на клиента и запишете съобщение **"Hello!"** в него. Докато текущото съобщение не е **"q"**. Заклучете паметта за сървъра, задайте ред на клиента. Изчакайте, ако клиентът използва паметта. Ако клиентът е обработил последното съобщение - посочете, че сървърът е обработил текущото съобщение, отпечатайте съобщението от клиента и върнете **"Ok!"** премахнете заключването на сървъра. Премахнете споделената памет.

shared-memory-server.c

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include "shared-memory-types.h"

int main(int argc, char ** argv)
{
    key_t key = ftok(FTOK_FILE, 1);
    if(key == -1)
    {
        printf("Error ftok.\n");
        return EXIT_FAILURE;
    }

    int shmid = shmget(key, sizeof(struct memory_block), 0666 | IPC_CREAT);
    if(shmid == -1)
    {
        printf("Error shmget.\n");
        return EXIT_FAILURE;
    }

    struct memory_block * mblock;
    mblock = (struct memory_block *) shmat(shmid, 0, 0);
    mblock->turn = CLIENT;
    mblock->server_lock = FREE;
    mblock->client_lock = FREE;
    mblock->read_last = SERVER;
    strcpy(mblock->string, "Hello");

    while(strcmp("q\n", mblock->string) != 0)
    {
        mblock->server_lock = BUSY;
        mblock->turn = CLIENT;
        while((mblock->client_lock == BUSY) &&
            (mblock->turn == CLIENT))
        {
            mblock->read_last = SERVER;
            printf("Sended: %s\n", mblock->string);
            if(strcmp("q\n", mblock->string) != 0)
                strcpy(mblock->string, "OK");
            mblock->server_lock = FREE;
        }
    }
}
```

Какво е системно програмиране?

```
printf("Server exit.\n");  
shmdt((void*)mblock);  
shmctl(shmid, IPC_RMID, 0);  
return EXIT_SUCCESS;  
}
```


Клиент за памет

Създайте `./shared-memory-server`, излезте при грешка. Разпределете споделената памет. Докато текущото съобщение не е "q". Заклучете паметта за клиента, задавайте ред на сървъра. Изчакайте, ако сървърът използва паметта. Ако сървърът е обработил последното съобщение - посочете, че клиентът е обработил текущото съобщение, отпечатайте съобщението от сървъра, прочетете ред от клавиатурата и премахнете заключването на клиента. Премахнете споделената памет.

`shared-memory-client.c`

```
#include<string.h>
#include<stdlib.h>
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include "shared-memory-types.h"

int main(int argc, char ** argv)
{
    key_t key = ftok(FTOK_FILE, 1);
    if(key == -1)
    {
        printf("Error ftok.\n");
        return EXIT_FAILURE;
    }

    int shmid = shmget(key, sizeof(struct memory_block), 0666);
    if(shmid == -1)
    {
        printf("Error shmget.\n");
        return EXIT_FAILURE;
    }

    struct memory_block * mb;
    mb = (struct memory_block *) shmat(shmid, 0, 0);

    while(strcmp("q\n", mb->string) != 0)
    {
        int i = 0;
        mb->client_lock = BUSY;
        mb->turn = SERVER;
        while((mb->server_lock == BUSY) &&
            (mb->turn == SERVER))
        if(mb->read_last == SERVER)
        {
            mb->read_last = CLIENT;
            printf("Received: %s\n", mb->string);
            while( (i < (MAX_LEN - 1)) &&
                ((mb->string[i++] = getchar()) != '\n') )
            mb->string[i] = '\0';
            mb->client_lock = FREE;
        }
    }

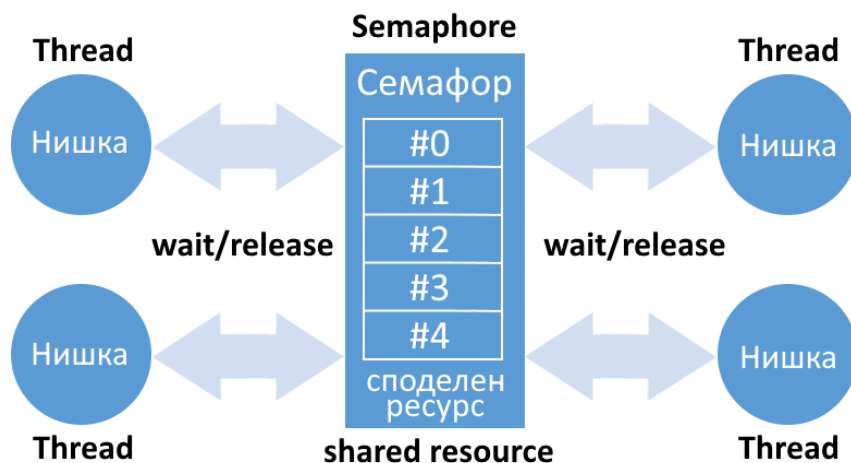
    printf("Client exit.\n");
    shmdt((void*)mb);
}
```

Какво е системно програмиране?

```
return EXIT_SUCCESS;  
}
```

Семафор

Семафорите са броячи, използвани за контрол на достъпа до споделени ресурси от множество процеси (броячи на ресурси). Използват се като блокиращ механизъм, за да се предотврати достъпът на процесите до конкретен ресурс, докато друг процес извършва операции с него. Броячът на ресурси намалява, когато процесът започне да използва ресурс и се увеличава обратно, когато ресурсът се освободи. Когато броячът = 0, ресурсът в момента не е налице.



Повече информация: [Semaphores](#)

Библиотеки за работа със семафори

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>

int semget(key_t key, int nsems, int semflg);
int semctl(int semid, int semnum, int cmd, ...);
int semop(int semid, struct sembuf *sops, size_t nsops);
```

- **semget()** - връща System V идентификатор на семафорно множество, свързано с аргумента **key**.
- **semctl()** с **cmd = SETVAL** - Задава стойността на **semval** да стане **arg.val** за **semnum**-тия семафор в множеството.
- **semctl()** с **cmd = IPC_RMID** - премахва семафора.
- **semop()** - изпълнява операции на избрани семафори от множеството, обозначени с **semid**.

Определяне на общи данни

Определяме заключващ файл **semaphore-server**. Дефинираме структурата за споделената памет **memory_block**. Променливата **string** съдържа текущото съобщение.

semaphore-types.h

```
#ifndef SEM_TYPES
#define SEM_TYPES

#define FTOK_FILE "./semaphore-server"

#define MAX_LEN 512

struct memory_block
{
    char string[MAX_LEN];
};

#endif
```

Семафорен сървър

- Задайте 2 семафора - първият показва, че сървърът трябва да чете, вторият за клиента
- Създайте **semaphore-server**, изход при грешка
- Разпределете и прикрепете споделената памет и напишете "Hello!"
- Освободете семафора на клиента
- Докато текущото съобщение не е "q":
- Поискайте сървърния семафорен ресурс
- Ако текущото съобщение не е "q", напишете "Ok!" в паметта и освободете клиента.
- Извадете и премахнете споделената памет и семафорите

semaphore-server.c

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/sem.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include"semaphore-types.h"

int main(int argc, char ** argv)
{
    key_t key = ftok(FTOK_FILE, 1);
    if (key == -1)
    {
        printf("Error ftok.\n");
        return EXIT_FAILURE;
    }

    int shmid = shmget(key, sizeof(struct memory_block), 0666 | IPC_CREAT);
    if(shmid == -1)
    {
        printf("Error shmget.\n");
        return EXIT_FAILURE;
    }

    int semid = semget(key, 3, 0666|IPC_CREAT);
    if(semid == -1)
    {
        printf("Error semget.\n");
        return EXIT_FAILURE;
    }

    struct sembuf buf[2];
    buf[0].sem_num = 0;
    buf[0].sem_flg = SEM_UNDO;
    buf[1].sem_num = 1;
    buf[1].sem_flg = SEM_UNDO;
    semctl(semid, 0, SETVAL, 0);

    struct memory_block * mb;
    mb = (struct memory_block *) shmat(shmid, 0, 0);
    strcpy(mb->string, "Hello!");
    buf[0].sem_op = -1;
    buf[1].sem_op = 1;
    semop(semid, (struct sembuf*) &buf[1], 1);

    while (strcmp("q\n", mb->string) != 0)
```



```
{
    semop(semid, (struct sembuf*) &buf, 1);
    printf("Client: %s\n", mb->string);
    if (strcmp("q\n", mb->string) != 0)
        strcpy(mb->string, "Ok!");
    buf[0].sem_op = -1;
    buf[1].sem_op = 1;
    semop(semid, (struct sembuf*) &buf[1], 1);
}

printf("Server exit.\n");
shmdt((void *) mb);
shmctl(shmid, IPC_RMID, 0);
semctl(semid, 2, IPC_RMID);
return EXIT_SUCCESS;
}
```

Семафор клиент

- Задайте 2 семафора - първият показва, че сървърът трябва да чете, вторият за клиента
- Създайте **semaphore-server**, изход при грешка
- Прикачете споделената памет
- Докато текущото съобщение не е "q":
- Поискайте семафорен ресурс на клиента и отпечатайте следващото съобщение
- Прочетете нов ред от клавиатурата в споделената памет
- Освободете семафора на сървъра
- Отделете споделената памет

semaphore-client.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "semaphore-types.h"

int main(int argc, char * argv[])
{
    key_t key = ftok(FTOK_FILE, 1);
    if (key == -1)
    {
        printf("Error ftok.\n");
        return EXIT_FAILURE;
    }

    int shmid = shmget(key, sizeof(struct memory_block), 0666);
    if (shmid == -1)
    {
        printf("Error shmget.\n");
        return EXIT_FAILURE;
    }

    int semid = semget(key, 2, 0666);
    if(semid == -1)
    {
        printf("Error semget.\n");
        return EXIT_FAILURE;
    }

    struct sembuf buf[2];
    buf[0].sem_num = 0;
    buf[0].sem_flg = SEM_UNDO;
    buf[1].sem_num = 1;
    buf[1].sem_flg = SEM_UNDO;
    buf[1].sem_op = -1;

    struct memory_block * mb;
    mb = (struct memory_block *) shmat(shmid, 0, 0);

    while (strcmp("q\n", mb->string) != 0)
    {
        semop(semid, (struct sembuf*) &buf[1], 1);
        printf("Server: %s\n", mb->string);
        int i = 0;
```

Какво е системно програмиране?

```
while ((i < (MAX_LEN - 1)) && ((mb->string[i++] = getchar()) != '\n') );
mb->string[i] = '\0';
buf[0].sem_op = 1;
buf[1].sem_op = -1;
semop(semid, (struct sembuf*) &buf, 1);
}

printf("Client exits\n");
shmdt((void *) mb);
return EXIT_SUCCESS;
}
```

Упражнение за синхронизация на комуникация между процеси

Упражнете материалите от темата, като реализирате представените по-долу проекти.

Double Shared Memory

Напишете двойка програми сървър и клиент (**double-memory-server.c** и **double-memory-client.c**), които споделят блокове памет помежду си. Сървърът да поддържа до 2 клиента едновременно, които не трябва да са в конфликт по време на работа. Сървърът инициализира споделената памет с поздравителни съобщения за клиентите си и ги чака те да променят със съобщенията си. Когато клиента напише ново съобщение, сървърът отговаря със същото съобщение, но с израз **Confirmed!** накрая.

Клиентът се стартира с един аргумент: **client_number**, който идентифицира клиента (1 или 2) за сървъра. Клиентът трябва непрекъснато да отпечатва съобщението от споделената памет, да чете ред от клавиатурата и да го записва в споделената памет. Програмата трябва да излезе, когато се въведе съобщение "q".

Double Dumper

Напишете програма **double-dumprer.c**, която в реално време показва промените в споделената памет от проекта **Double Shared Memory**. Може да се наложи да промените сървърните или клиентските програми, така че да поддържат тази функционалност.

Част 7 – Сокети

- [Сокетите в Линукс](#)
- [Работа със сокети в C](#)
- [Използване на Unix сокети](#)
- [Файл сокет сървър](#)
- [Файл сокет клиент](#)
- [Двойка сокети](#)
- [Пример за двойка сокети](#)
- [Мрежови сокети](#)
- [Мрежов сокет сървър](#)
- [Мрежов сокет клиент](#)
- [Упражнение върху сокети](#)

Сокетите в Линукс

Повечето комуникации между процесите използват модела клиент-сървър. Един от двата процеса, наречен **клиент** (Термин на Английски език: *client*), се свързва с другия процес наречен **сървър** (Термин на Английски език: *server*), обикновено с искане за информация. Всяка страна на тази вдупосочна комуникация може да бъде представена чрез **сокет** (Термин на Английски език: *socket*). Процесите клиент и сървър създават техен собствен сокет.

Видове сокети

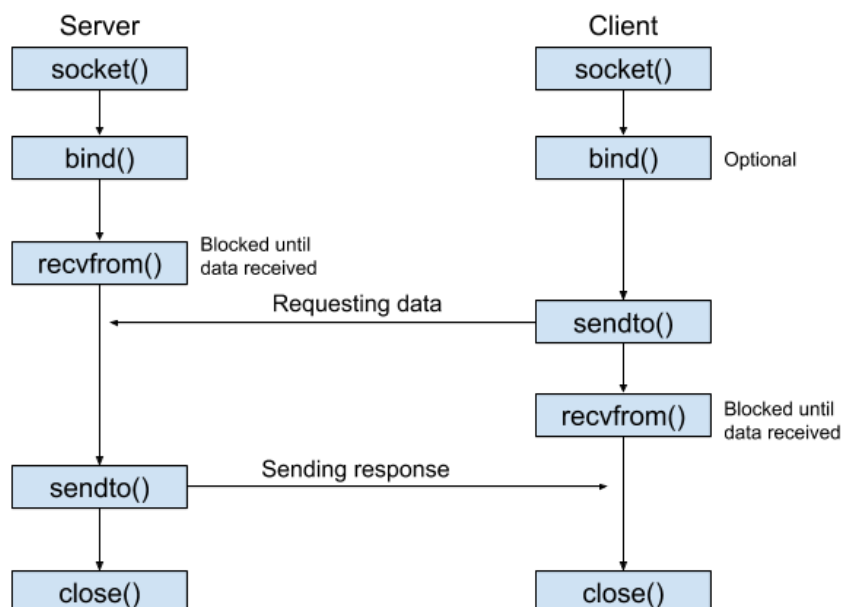
Unix domain socket е крайна точка за обмен на данни между процеси, изпълняващи една и съща хостова операционна система.

Network socket е крайна точка за изпращане или получаване на данни в рамките на хост от компютърната мрежа.

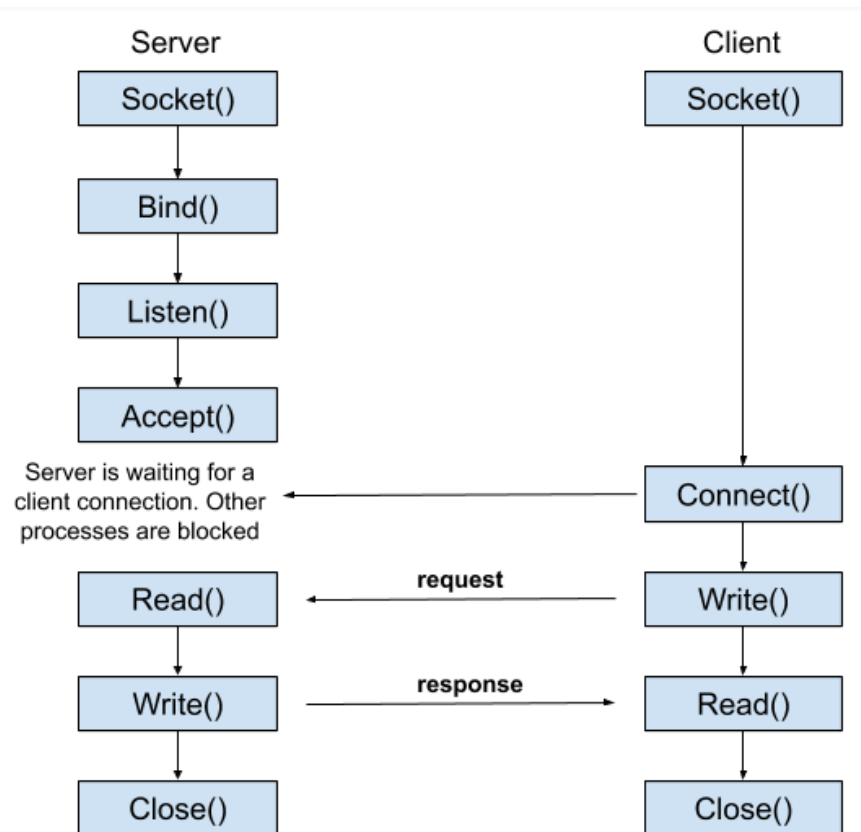
Типове мрежови сокети

Сокетите поддържат предаването на надежден **поток** (Термин на Английски език: *stream*) от байтове, както и подредено и ненадеждно предаване на **дейтаграми** (Термин на Английски език: *datagrams*). Поточните сокети разглеждат комуникациите като непрекъснат поток от символи, докато Дейтаграм сокетите трябва да четат целите съобщения наведнъж.

Комуникация посредством Дейтаграм сокет



Комуникация посредством Поточен сокет



Работа със сокети в С

метод	описание
socket ()	създава крайна точка за комуникация
bind ()	свързва име към сокет
recv (), recvfrom (), recvmsg ()	получава съобщение от сокет
send (), sendto (), sendmsg ()	изпраща съобщение към сокет
socketpair ()	създава двойка свързани сокета
listen ()	слуша за връзки към сокет
accept ()	приема връзка към сокет
connect ()	инициира връзка към сокет
htons ()	преобразува unsigned short integer в network byte order
gethostbyname ()	връща структура от тип hostent за даденото име на хост

Използване на Unix сокети

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

int socket(int domain, int type, int protocol);
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr
```

- **socket()** създава крайна точка за комуникация и връща дескриптор на файла, който се отнася до тази крайна точка.
 - домейнът (*Термин на Английски език: domain*) определя семейството протоколи, което ще се използва: **AF_UNIX**, **AF_LOCAL**, **AF_INET**.
 - Тип (*Термин на Английски език: type*) определя комуникационната семантика: **SOCK_STREAM** или **SOCK_DGRAM**.
- Когато се създава сокет със **socket()**, то той съществува в именованото пространство (адресното пространство), но няма зададен адрес.
- **bind()** присвоява адреса, посочен в **addr**, на сокета, посочен от файловият дескриптор **sockfd** върнат от метода **socket()**.
- **sockaddr** има следната структура:

```
struct sockaddr {
    sa_family_t sa_family;
    char sa_data[14];
}
```

Файл сокет сървър

Отворете Unix сокет за дейтаграми, при грешка изведете съобщение и излезте от програмата. Свържете сокета с файл **socket.file**, излезте при грешка. Вземете съобщение от сокета, излезте при грешка. Отпечатайте съобщението, затворете сокета и изтрийте сокет файла.

file-socket-server.c

```
#include<stdlib.h>
#include<unistd.h>
#include<stdio.h>
#include<string.h>
#include<errno.h>
#include<sys/types.h>
#include<sys/socket.h>

#define SOCK_NAME "socket.file"
#define BUF_SIZE 256

int main(int argc, char ** argv)
{
    int sock = socket(AF_UNIX, SOCK_DGRAM, 0);
    if(sock < 0)
    {
        printf("Error socket,\n");
        return EXIT_FAILURE;
    }

    struct sockaddr server, client;
    server.sa_family = AF_UNIX;
    strcpy(server.sa_data, SOCK_NAME);

    if(bind(sock, &server, strlen(server.sa_data) + sizeof(server.sa_family)) < 0)
    {
        printf("Error bind.\n");
        return EXIT_FAILURE;
    }

    char buf[BUF_SIZE];
    socklen_t len = sizeof(client);
    int bytes = recvfrom(sock, buf, sizeof(buf), 0, &client, &len);
    if(bytes < 0)
    {
        printf("Error recvfrom.\n");
        return EXIT_FAILURE;
    }
    buf[bytes] = '\0';
    client.sa_data[len] = 0;
    printf("Client: %s\n", buf);

    close(sock);
    unlink(SOCK_NAME);
    return EXIT_SUCCESS;
}
```

Файл сокет клиент

Отворете Unix сокет за дейтаграми, при грешка изведете съобщение и излезте от програмата. Настройте структурата за името на сървъра на сокета към **socket.file**. Изпратете съобщение до сървъра и затворете сокета.

file-socket-client.c

```
#include<unistd.h>
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<errno.h>
#include<sys/types.h>
#include<sys/socket.h>

#define SOCK_NAME "socket.file"
#define BUF_SIZE 256

int main(int argc, char ** argv)
{
    int sock = socket(AF_UNIX, SOCK_DGRAM, 0);
    if(sock < 0)
    {
        printf("Error socket.\n");
        return EXIT_FAILURE;
    }

    struct sockaddr server;
    server.sa_family = AF_UNIX;
    strcpy(server.sa_data, SOCK_NAME);

    char buf[BUF_SIZE];
    strcpy(buf, "HELLO");
    sendto(sock, buf, strlen(buf), 0, &server, strlen(server.sa_data) + sizeof(se

    close(sock);
    return EXIT_SUCCESS;
}
```

Двойка сокети

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

int socketpair(int domain, int type, int protocol, int sv[2]);
```

Извикването на метода **socketpair()** създава двойка свързани сокети в определения домейн (domain), от посочения тип (type) и използвайки по избор посочения протокол (protocol).

Файловите дескриптори, използвани при създаването на новите сокети, се връщат съответно в **sv[0]** и **sv[1]**.

При успех метода връща 0, при грешка -1, а информация за грешката се намира в **errno**.

Пример за двойка сокети

Отворете двойка сокети, при грешка изведете съобщение и излезте от програмата.

Създайте дъщерен процес, при грешка изведете съобщение и излезте от програмата.

Родителския процес: чака получаване на съобщение като чете от първият сокет, след което изпраща отговор с подбавашо обратно съобщение, като пише в първият сокет.

Дъщерния процес: изпраща съобщение като пише в втория сокет и чака получаване на обратно съобщение, като чете от втория сокет.

Накрая затворете двойката сокети.

socket-pair.c

```
#include<unistd.h>
#include<stdlib.h>
#include<stdio.h>
#include<errno.h>
#include<sys/types.h>
#include<sys/socket.h>

#define MSG_HELLO "HELLO"
#define MSG_OK "OK"
#define BUF_SIZE 256

int main(int argc, char ** argv)
{
    char buf[BUF_SIZE];

    int sockets[2];
    if(socketpair(AF_UNIX, SOCK_STREAM, 0, sockets) < 0)
    {
        printf("Error socketpair.\n");
        return EXIT_FAILURE;
    }

    int pid = fork();
    if(pid < 0)
    {
        printf("Error fork.\n");
        return EXIT_FAILURE;
    }
    if(pid > 0)
    {
        read(sockets[0], buf, sizeof(buf));
        write(sockets[0], MSG_OK, sizeof(MSG_OK));
        printf("Process (pid = %i) send message: %s\n", getpid(), buf);
    }
    if(pid == 0)
    {
        write(sockets[1], MSG_HELLO, sizeof(MSG_HELLO));
        read(sockets[1], buf, sizeof(buf));
        printf("Process (pid = %i) send message: %s\n", getpid(), buf);
    }

    close(sockets[0]);
    close(sockets[1]);
    return EXIT_SUCCESS;
}
```

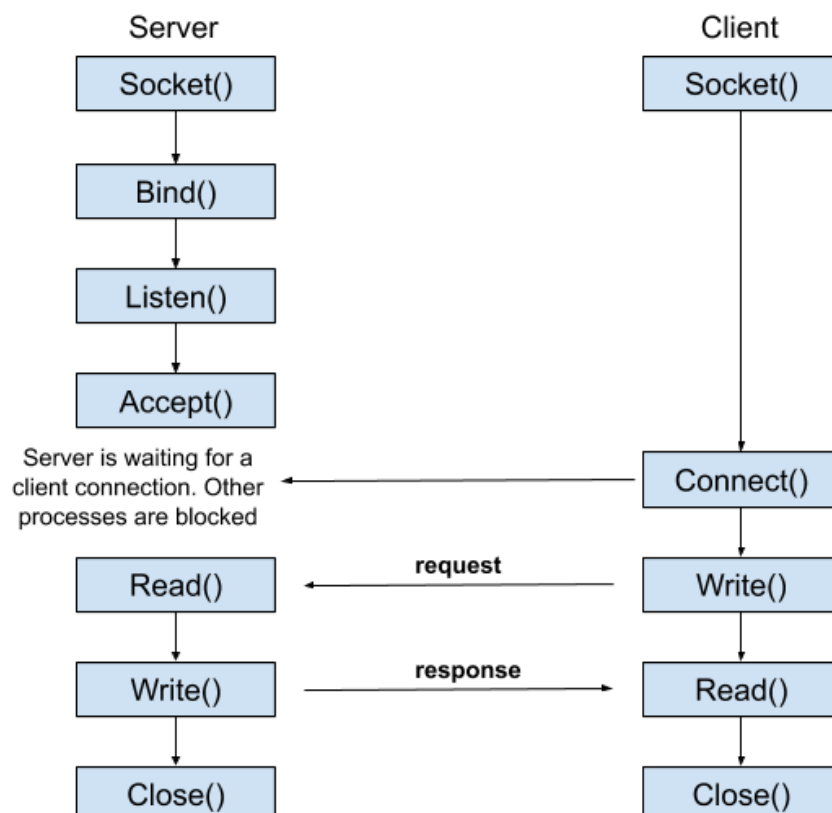

Мрежови сокети

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
extern int h_errno;

struct hostent *gethostbyname(const char *name);
uint16_t htons(uint16_t hostshort);
int listen(int sockfd, int backlog);
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

За да приемете връзките, се изпълняват следните стъпки:

1. Създава се сокет чрез **socket()**.
2. Сокета се свързва с локален адрес с помощта на **bind()**, така че другите сокети да могат да се свързват **connect()** с него.
3. Желанието за приемане на входящи връзки и ограничение броя на входящите връзки се определят от **listen()**.
4. Връзките се приемат с **accept()**.



Мрежов сокет сървър

Проверете входните аргументи, излезте при грешка. Отворете INET, STREAM сокет, излезте при грешка. Инициализирайте структурата **sockaddr_in** за сървъра. Свържете гнездото към порт, излезте при грешка. Започнете да слушате, разрешете само една връзка. Приемете нова връзка, излезте при грешка. Прочетете от сокета в буфер. Отпечатайте полученото съобщението от буфера. Изпратете "OK" в сокета. Затворете сокета.

network-socket-server.c

```
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define BUF_SIZE 256
int main(int argc, char ** argv)
{
    // Arguments Check!
    if (argc != 2)
    {
        printf("Usage: %s <port_number>\n", argv[0]);
        return EXIT_FAILURE; // return -1;
    }

    // Socket Create
    int sock_server = socket(AF_INET, SOCK_STREAM, 0);
    if (sock_server < 0)
    {
        printf("Socket Create Error: %d\n", errno);
        return EXIT_FAILURE; // return -1;
    }

    // Server Address
    struct sockaddr_in server_address;
    memset((char *) &server_address, 0, sizeof(server_address));
    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = htonl(INADDR_ANY);
    server_address.sin_port = htons(atoi(argv[1]));

    // Socket Bind
    if (bind(sock_server, (struct sockaddr *) &server_address, sizeof(server_address)) != 0)
    {
        printf("Socket Bind Error: %d\n", errno);
        return EXIT_FAILURE; // return -1;
    }

    // Socket Listen
    if ((listen(sock_server, 5)) != 0)
    {
        printf("Socket Listen Error: %d\n", errno);
        return EXIT_FAILURE; // return -1;
    }
}
```

```
// Client Address
struct sockaddr_in client_address;
int len = sizeof(client_address);
int sock_client = accept(sock_server, (struct sockaddr *) &client_address, &len);
if (sock_client < 0)
{
    printf("Accept Socket Error: %d\n", errno);
    return EXIT_FAILURE; // return -1;
}

// Socket Read/Write
char buf[BUF_SIZE];
memset(buf, 0, BUF_SIZE);
read(sock_client, buf, BUF_SIZE-1);
buf[BUF_SIZE] = '\0';
printf("Received message: %s\n", buf);
write(sock_client, "OK", 2);

close(sock_client);
close(sock_server);
return EXIT_SUCCESS; // return 0;
}
```

Мрежов сокет клиент

Проверете входните аргументи, излезте при грешка. Отворете INET, STREAM сокет, излезте при грешка. Вземете хост името сървъра, излезте при грешка. Конфигурирайте и свържете сокета към адреса на сървъра и порт, излезте при грешка. Изведете на екрана, че очаквате вход и прочетете нов текстов ред от клавиатурата. Изпратете въведения текст към сокета. Прочетете отговор от сокета и го отпечатайте. Затворете сокета.

network-socket-client.c

```
#include<string.h>
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>
#include<strings.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<netdb.h>
#define BUF_SIZE 256
int main(int argc, char ** argv)
{
    // Arguments Check!
    if (argc != 3)
    {
        printf("Usage: %s <ip_address> <port_number>\n", argv[0]);
        return EXIT_FAILURE; // return -1;
    }

    // Socket Create
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0)
    {
        printf("Socket Create Error: %d\n", errno);
        return EXIT_FAILURE; // return -1;
    }

    // Server IP Address
    struct in_addr addr;
    inet_aton(argv[1], &addr);
    struct hostent *server = gethostbyaddr(&addr, sizeof(addr), AF_INET);
    if (server == NULL)
    {
        printf("Host Not Found Error: %d\n", errno);
        return EXIT_FAILURE; // return -1;
    }

    // Server Address
    struct sockaddr_in serv_addr;
    memset((char *) &serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    strncpy((char *)&serv_addr.sin_addr.s_addr, (char *)server->h_addr, server->h_addr_len);
    serv_addr.sin_port = htons(atoi(argv[2]));

    // Socket Connect
```

```
if (connect(sock, (const struct sockaddr *) &serv_addr, sizeof(serv_addr)) <
{
    printf("Socket Connect Error: %d", errno);
    return EXIT_FAILURE; // -1
}

// Socket Write/Read
printf("Write the message and press [Enter] to send:\n");
char buf[BUF_SIZE];
memset(buf, 0, BUF_SIZE);
fgets(buf, BUF_SIZE-1, stdin);
write(sock, buf, strlen(buf));
memset(buf, 0, BUF_SIZE);
read(sock, buf, BUF_SIZE-1);
printf("Received: %s\n", buf);

close(sock);
return EXIT_SUCCESS; // return 0;
}
```

Упражнение

Упражнете материалите от темата, като реализирате представения по-долу проект.

Better Chat

Усъвършенствайте двойката програми за мрежов сокет сървър **network-socket-server.c** и клиент **network-socket-client.c**, като добавите следните подобрения:

- Сървърът трябва да поддържа няколко паралелни клиентски връзки.
- Разрешаване по няколко съобщения на разговор, докато клиентът не изпрати съобщение **!quit** на сървъра.

Част 8 – Нишки

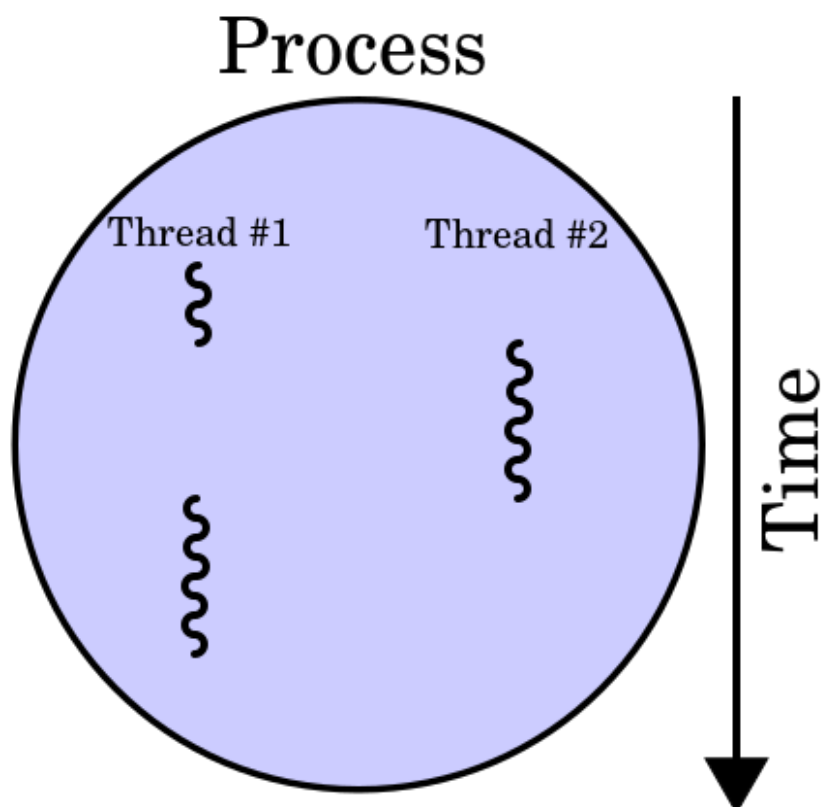
- [Въведение в нишките](#)
- [Библиотека за работа с нишки](#)
- [Функции за управление на нишки](#)
- [Създаване на нишки](#)
- [Финализиране на нишки](#)
- [Съединяване на нишки](#)
- [Пример за съединяване на нишки](#)
- [Функции за прекратяване на нишки](#)
- [Типове при прекратяване на нишки](#)
- [Пример за прекратяване на нишки](#)
- [Упражнение върху нишки](#)

Въведение в нишките

Нишка (*Термин на Английски език: thread*) в компютърните науки е съкращение от нишка за изпълнение. Нишките са начин програмата да се раздели на две или повече едновременно или псевдоедновременно изпълнявани задачи.

Нишките и процесите се различават при различните операционни системи, но като цяло нишка се съдържа вътре в процес и различните нишки в един и същи процес споделят едни и същи ресурси, докато различните процеси в една и съща многозадачна операционна система не.

Нишките консумират по-малко системни ресурси в сравнение с процесите.



Източник на изображението: [Wikipedia](#)

Библиотека за работа с нишки

Библиотеката **pthread** дефинира набор от типове, функции и константи на езика за програмиране C, които поддържат приложения с изисквания за множество потоци на контрол, наречени нишки, в рамките на един процес.

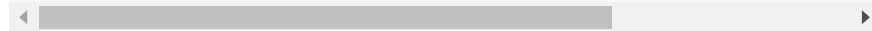
Има около 100 процедури за нишки, всички с префикс **pthread_** и те могат да бъдат категоризирани в четири групи:

Група	Описание
Thread management	Управление на нишки, създаване и присъединяване на нишки
Mutexes	Мютекси
Condition variables	Условни променливи
Synchronization	Синхронизация между нишки с помощта на заключване на четене/запис

Бележка: В тази тема ще се фокусираме само върху "Управление на нишки".

Функции за управление на нишки

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start
```



стартира нова нишка в извикващия процес

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

изчаква приключване на определена нишка

```
void pthread_exit(void *retval);
```

прекратява извикващата нишка и връща стойност чрез `retval`, която (ако нишката може да се присъедини) е достъпна за друга нишка в същия процес, който извиква **`pthread_join`**.

Създаване на нишки

Първоначално вашата програма **main()** се състои от една нишка по подразбиране. Всички други нишки трябва да бъдат изрично създадени от програмиста. **pthread_create** създава нова нишка и я прави изпълнима. Този метод може да бъде извикан произволен брой пъти от всяко място във вашия код.

pthread_create аргументи:

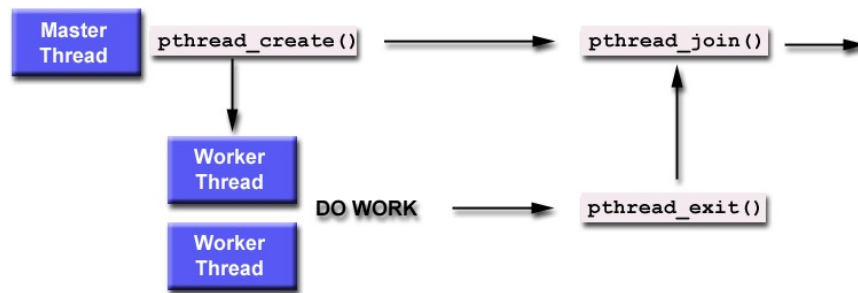
- **thread**: уникален идентификатор за новата нишка, върната от подпрограмата.
- **attr**: атрибутен обект, който може да се използва за задаване на атрибути на нишка. Можете да посочите обект с атрибути на нишка или NULL за стойностите по подразбиране.
- **start_routine**: процедура, която нишката ще изпълни, след като бъде създадена.
- **arg**: единичен аргумент, който може да бъде предаден на **start_routine**. Той трябва да се предава чрез препратка като указател на тип **void**. NULL може да се използва, ако не трябва да се предава аргумент.

Финализиране на нишки

Има няколко начина, по които една нишка може да бъде финализирана:

- Нишката приключва нормално и нейната работа е свършена.
- Нишката извиква **pthread_exit** независимо дали работата ѝ е свършена или не.
- Нишката е анулирана от друга нишка посредством **pthread_cancel**.
- Целият процес се прекратява поради извикване на **exec()** или **exit()**
- Ако **main()** завърши първи, без изрично да извика **pthread_exit**

Съединяване на нишки



`pthread_join()` блокира извикващата нишка, докато нишката с идентификатор **`threadid`** приключи.

Програмистът може да получи състоянието на прекратяване на нишката, ако указано извикване на **`pthread_exit()`**.

Пример за съединяване на нишки

- Дефинираме функция за генериране на пермутации
- Дефинираме функция при работа на нишките:
 - Разпределяме вектор с размер **arg**
 - Инициализираме вектора с първата пермутация и го отпечатваме
 - Докато има следваща пермутация: генерираме, отпечатваме и синхронизираме
 - Почистваме, като освобождаваме заетата от вектора памет и приключваме нишката
- В главната функция:
 - Стартираме функцията за нишка с размер 4, изход при грешка
 - Стартираме функцията за нишка с размер 3, изход при грешка
 - Изчакваме първата нишка и отпечатваме известие, изход при грешка
 - Изчакваме втората нишка и отпечатваме известие, изход при грешка
 - Отпечатваме **Job Done!** и приключваме

threads-join.c


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

// Print
void print(int *v, int n)
{
    for(int i=0;i<n;i++)
        printf("%i", v[i]);
        printf("\n");
}

// Swap
void swap(int *i, int *j)
{
    int temp = *i;
    *i = *j;
    *j = temp;
}

// Reverse
void reverse(int *v, int n)
{
    for(int i=0;i<(n/2); i++)
        swap(&v[i], &v[n-1-i]);
}

// Permute
int permute(int *v, int n)
{
    int i = n-1, j;
    while((i > 1) && (v[1] < v[i-1])) i--;
    if(v[i] > v[i-1])
    {
        j = n - 1;
        while(v[j] < v[i-1]) j--;
        swap(&v[j], &v[i-1]);
        reverse(&v[i], n-1);
        return 1;
    }
    return 0;
}

// Threads Function
void * func(void *arg)
{
    int i;
    int n = *(int*)arg; // n = 3 or 4
    int *v = malloc(sizeof(int)*n); // v[n]
    for(i=0;i<n;i++) v[i] = i+1; // 1,2 ... n
    print(v,n); // 123 or 1234
}
```

```
while(permute(v,n) != 0)
{
    print(v,n);
    sync();
}
free(v);
pthread_exit(arg);
}
// Main
int main(int argc, char *argv)
{
    void *ret;
    int result;
    pthread_t thread1, thread2;

    // thread one
    int n1 = 4;
    result = pthread_create(&thread1, NULL, func, &n1);
    if(result != 0)
    {
        printf("Error Creating Thread One!\n");
        return EXIT_FAILURE; // return -1;
    }

    // tread two
    int n2 = 3;
    result = pthread_create(&thread2, NULL, func, &n2);
    if(result != 0)
    {
        printf("Error Creating Thread Two!\n");
        return EXIT_FAILURE; // return -1;
    }

    // thread one result
    result = pthread_join(thread1, &ret);
    if(result != 0)
    {
        printf("Error Joining Thread One!\n");
        return EXIT_FAILURE; // return -1;
    }
    printf("Thread one finished with result: %i\n", *(int*)ret);

    // thread two result
    result = pthread_join(thread2, &ret);
    if(result != 0)
    {
        printf("Error Joining Thread Two!\n");
        return EXIT_FAILURE; // return -1;
    }
}
```

Какво е системно програмиране?

```
}  
printf("Thread two finished with result: %i\n", *(int*)ret);  
  
// Job Done  
printf("Job Done!\n");  
  
return EXIT_SUCCESS; // return 0  
}
```

Компилиране на програмата с изрична инструкция към компилатора да поддържа нишки:

```
gcc threads-join.c -o threads-join -lpthread
```

Функции за прекратяване на нишки

```
int pthread_cancel(pthread_t thread);
```

изпраща заявка за прекратяване на нишка **thread**.

```
int pthread_setcancelstate(int state, int *oldstate);
```

задава състояние за прекратяване на нишката посредством **state** на **PTHREAD_CANCEL_ENABLE** или **PTHREAD_CANCEL_DISABLE**

```
int pthread_setcanceltype(int type, int *oldtype);
```

задава тип за прекратяване на нишката посредством **type** на **PTHREAD_CANCEL_DEFERRED** или **PTHREAD_CANCEL_ASYNCHRONOUS**

Типове при прекратяване на нишки

Функцията `pthread_cancel()` изпраща заявка за прекратяване на нишка.

Дали и кога целевата нишка реагира на заявката за прекратяване зависи от два атрибута, които са под контрола на тази нишка: нейното състояние на отмяна (разрешено/деактивирано) и тип:

- **Deferred** (поведение по подразбиране) означава, че прекратяването ще бъде забавено, докато нишката не извика функция `pthread_testcancel()`.
- **Asynchronous** означава, че нишката може да бъде отменена по всяко време (обикновено незабавно, но системата не гарантира това).

Пример за прекратяване на нишки

- Създаваме нишка
- Опитваме да я прекратим
- Изчакваме я да приключи

```
pthread_create(&thread, NULL, thread_func, NULL);  
pthread_cancel(thread);  
pthread_join(thread, NULL);
```

- Маркираме нишката като неотменима
 - ... докато работи
 - ... и докато не разрешим прекратяване
 - ... и достигнем точка за прекратяване

```
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);  
...  
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);  
pthread_testcancel();
```

- Правим нишката отменима по всяко време
- Но засега я маркираме в неотменяемо състояние
 - ... докато работи
 - ... и докато не разрешим прекратяване
- Нишката се прекратяване автоматично, няма нужда да достигаме до точка за прекратяване

```
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);  
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);  
...  
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
```

threads-cancel.c

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

int i;
void * thread_func(void * arg)
{
    pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
    for(i=0; i<4; i++)
    {
        sleep(1);
        printf("I am running verry important process %i ...\n", i+1);
    }
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    pthread_testcancel();
    printf("YOU WILL NOT STOP ME!!!\n");
}

int main(int argc, char * argv[])
{
    pthread_t thread;
    pthread_create(&thread, NULL, thread_func, NULL);
    while(i < 1) sleep(1);
    pthread_cancel(thread);
    printf("Requsted to cancel the thread!\n");
    pthread_join(thread, NULL);
    printf("The thread is stopped!\n");
    return EXIT_SUCCESS; // 0
}
```

Компилиране на програмата с изрична инструкция към компилатора да поддържа нишки:

```
gcc threads-cancel.c -o threads-cancel -lpthread
```

Упражнение върху нишки

Упражнете материалите от темата, като реализирате представения по-долу проект.

ThreadedChat

Рефакторирайте двойката програми мрежов сървър/клиент от предишната тема със следните промени:

- Многопотребителската обработка в сървъра трябва да се извършва с помощта на нишки **threads**.
- Сървърът спира работа, когато някой от клиентите изпрати съобщение **!quitserver** към сървъра.

Част 9 - Синхронизация на нишки

- Изход от нишка
- Управление на изход от нишка
- Пример за изход от нишка
- Защо е необходима синхронизация?
- Механизми за синхронизация
- Мютекси
- Кога е необходимо заключване?
- Типична употреба на мютекси
- Проблеми при състезателни условия
- Безопасен за нишките код
- Състояние на мъртва хватка
- Създаване и унищожаване на мютекси
- Заключване и отключване на мютекси
- Пример за синхронизиране посредством използване на мютекс
- Синхронизиране със семафори
- Пример за синхронизиране посредством използване на семафор
- Упражнение за синхронизация на нишки

Изход от нишка

Нишка може да организира кои функции да бъдат извиквани при изход от нишката. Тези функции са известни още като **манипулатори за почистване на нишки** (*Термин на Английски език: Thread Cleanup Handlers*). Повече от един манипулатор за почистване може да бъде установен за всяка нишка. Манипулаторите се записват в **стек** (*Термин на Английски език: Stack*), което означава, че се изпълняват в обратен ред спрямо добавянето им.

Когато дадена нишка бъде анулирана или прекратена чрез извикване на **pthread_exit()**, всички подредени манипулатори за почистване се извеждат и изпълняват в обратен ред на реда, в който са били поставени в стека.

Управление на изход от нишка

```
#include<pthread.h>
void pthread_cleanup_push(void (*routine)(void *), void *arg);
void pthread_cleanup_pop(int execute);
```

- Функцията **pthread_cleanup_push()** добавя в горната част на стека съдържател на манипулатори за почистване на нишки. Когато манипулатора бъде извикан за изпълнение по-късно, `arg` променливата ще бъде подадена като аргумент.
- Функцията **pthread_cleanup_pop()** изважда от горната част на стека съдържател на манипулатори за почистване и по избор я изпълнява, ако изпълнението е различно от нула.

Пример за изход от нишка

- Дефинираме функция за изход, която освобождава заетата памет
- Дефинираме функция на нишката: Заклучване на отмяната на нишката
- Заемаме 1K памет и извеждаме съобщение
- Добавяме манипулатор за почистване при изход от нишката, предаващ заетата памет като параметър
- Отключваме отмяната на нишката
- Отпечатваме 5 съобщения за 5 секунди
- Извикваме и изпълняваме, манипулатора за почистване при изход от нишката

thread-clean-exit.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#define MEM_SIZE 1024 // 1K
// Thread Cleanup Handler Function
void thread_cleanup_handler(void *arg)
{
    free(arg);
    printf("Freed %i bytes accolated memory.\n", MEM_SIZE);
}
// Thread Handler Function
void *thread_handler(void * arg)
{
    // Critical Section
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);

    void *mem = malloc(MEM_SIZE);
    printf("Allocated %i bytes memory.\n", MEM_SIZE);

    // Attach Thread Cleanup Hanler
    pthread_cleanup_push(thread_cleanup_handler, mem);

    // End of Critical Section
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);

    for(int i=0; i<5; i++)
    {
        sleep(1); // sleep for 1 second
        printf("Important process %i is running...\n", i+1);
    }

    // Execute Thread Cleanup Handler
    pthread_cleanup_pop(1);
}
// Main Function
int main(int argc, char *argv[])
{
    pthread_t thread;
    int result = pthread_create(&thread, NULL, thread_handler, NULL);
    if(result == -1)
    {
        printf("Error creating thread!\n");
        return EXIT_FAILURE; // -1
    }
    pthread_join(thread, NULL);
}
```

Какво е системно програмиране?

```
return EXIT_SUCCESS; // 0  
}
```

Компилиране на програмата с изрична инструкция към компилатора да поддържа нишки:

```
gcc thread-clean-exit.c -o thread-clean-exit -lpthread
```

Защо е необходима синхронизация?

Синхронизирането на нишки (*Термин на Английски език: Thread synchronization*) е едновременното изпълнение на две или повече нишки, които споделят критични ресурси.

Нишките трябва да бъдат синхронизирани, за да се избегнат конфликти при използване на критични ресурси.

В противен случай могат да възникнат конфликти, когато паралелно работещи нишки се опитват да модифицират обща променлива по едно и също време.

Механизми за синхронизация

Библиотеката с нишки предоставя три механизма за синхронизация:

- **Мютекси** (*Термин на Английски език: Mutexes*). Взаимно заключване, блокиране на достъпа до променливи от други нишки. Това налага изключителен достъп от нишка до променлива или набор от променливи.
- **Съединения** (*Термин на Английски език: Joins*). Нишката изчаква, докато другите нишки приключат.
- **Условни променливи** (*Термин на Английски език: Condition Variables*). Докато мютексите прилагат синхронизация, като контролират достъпа на нишка до данни, условните променливи позволяват на нишките да се синхронизират въз основа на действителната стойност на данните.

Мютекси

Мютексите (*Термин на Английски език: Mutexes*) се използват за предотвратяване на несъответствия на данните, които могат да възникнат когато множество нишки работят с една и съща област на паметта, извършени по едно и също време, или за предотвратяване на **състояния на състезание**, когато се очаква определен ред при извършване на операциите с паметта.

Състояние на състезание (*Термин на английски език: Race Condition*) често възниква, когато две или повече нишки трябва да извършат операции в една и съща област на паметта, но резултатите от изчисленията зависят от реда, в който се изпълняват тези операции.

Мютексите се използват за сериализиране на споделени ресурси като памет. Всеки път, когато глобален ресурс е достъпен от повече от една нишка, ресурсът трябва да има мютекс, свързан с него.

Кога е необходимо заключване?

Thread 1	Thread 2	Balance
Read balance: \$1000		\$1000
	Read balance: \$1000	\$1000
	Deposit \$200	\$1000
Deposit \$200		\$1000
Update balance \$1000+\$200		\$1200
	Update balance \$1000+\$200	\$1200

В горния пример трябва да се използва мютекс за заключване на баланса (*Променлива: balance*), докато нишката използва този споделен ресурс от данни.

Thread 1	Thread 2	Balance
Read balance: \$1000		\$1000
Deposit \$200		\$1000
Update balance \$1000+\$200		\$1200
	Read balance: \$1200	\$1200
	Deposit \$200	\$1200
	Update balance \$1200+\$200	\$1400

В горния пример нишка 1 заключва ресурса за баланс (*Променлива: balance*) и нишка 2 е блокирана, докато мютексът не бъде освободен.

Типична употреба на мютекси

Типична последователност при използването на мютекс е както следва:

- Създайте и инициализирайте мютекс променлива
- Няколко нишки се опитват да заключат мютекса
- Само една успява и тази нишка е собственик на мютекса
- Нишката собственик на мютекса изпълнява някакъв набор от действия
- Собственикът отключва мютекса
- Друга нишка придобива мютекса и повтаря процеса
- Накрая мютексът се унищожава

Когато няколко нишки се състезават за мютекс, губещите блокират при това повикване, но е възможно да се използва метод **trylock()**, при който губещите нишки няма да бъдат блокирани.

Когато защитава споделените данни, всяка нишка, която трябва да използва мютекс, трябва да го направи. Например, ако 4 нишки актуализират едни и същи данни, но само една използва мютекс, данните все още могат да бъдат повредени.

Проблеми при състезателни условия

Състезателни условия (*Английски език: Race conditions*) въпреки че кодът може да се появи на екрана в реда, в който желаете да бъде изпълнен, нишките се планират от операционната система и се изпълняват на случаен принцип. Не може да се приеме, че нишките се изпълняват в реда, в който са създадени.

Възможно е също така те да се изпълняват с различна скорост. Когато нишките се изпълняват (надпреварват се да завършат), те могат да дадат неочаквани резултати.

Трябва да се използват **Мютекси** (*Термин на Английски език: Mutexes*) и **Съединения** (*Термин на Английски език: Joins*), за да се постигне предсказуем ред на изпълнение и резултат.

Безопасен за нишките код

Безопасен за нишките код (*Термин на Англиски език: Threads safe code*) означава, че нишките трябва да извикват само функции, които съдържат безопасен за изпълнение код.

Това означава, че няма статични или глобални променливи, които други нишки могат да забият или прочетат, ако се използва една нишка. Ако се използват статични или глобални променливи, трябва да се приложат мутекси или функциите да се пренапишат, за да се избегне използването на тези променливи.

В езика C локалните променливи се разпределят динамично върху стека. Следователно всяка функция, която не използва статични данни или други споделени ресурси, е безопасна за нишки.

Функциите, които не са безопасни за нишки, могат да се използват само от една нишка в даден момент в програмата и трябва да се гарантира уникалността на нишката.

Състояние на мъртва хватка

Състояние на мъртва хватка (*Термин на Английски език: Deadlock*) това състояние възниква, когато се използва мутекс и той не се освободи чрез отключване и остане заключен.

Това води до спиране на изпълнението на програмата за неопределено време. То може да бъде причинено и от недобре написано **application** или **joins**.

Бъдете внимателни, когато прилагате два или повече мутекса в една част от кода. Ако първият **pthread_mutex_lock** е приложен и вторият **pthread_mutex_lock** не успее поради прилагането на друга нишка на мутекс, първият мутекс може в крайна сметка да блокира всички други нишки от достъп до данни, включително нишката, която държи втория мутекс.

Нишките могат да чакат неограничено време ресурсът да се освободи, което води до задънена улица. Най-добре е да се тества и ако се появи неуспех, да се освободят ресурсите и да се спре, преди да се опита отново.

Създаване и унищожаване на мютекси

```
#include<pthread.h>

void pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)
void pthread_mutex_destroy(pthread_mutex_t *mutex)
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Променливите за мютекси трябва да бъдат декларирани с тип **pthread_mutex_t** и трябва да бъдат инициализирани, преди да могат да бъдат използвани. Съществуват два начина за инициализиране на променлива от тип мютекс:

- Статично, когато се декларират.
- Динамично, посредством **pthread_mutex_init()**. Този метод позволява задаването на атрибути за мютекс обекта.

Заклучване и отключване на мутекси

```
#include<pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex)
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

- **pthread_mutex_lock()** се използва от дадена нишка, за да получи заключване на посочената мютекс променлива. Ако мютексът вече е заключен от друга нишка, това извикване ще блокира извикващата нишка, докато мютексът не бъде отключен.
- **pthread_mutex_trylock()** ще се опита да заключи мютекс. Ако обаче мютексът вече е заключен, процедурата ще се върне незабавно с код за грешка заето (*Термин на Английски език: busy*). Тази процедура може да бъде полезна за предотвратяване на условия за възникване на мъртва хватка.
- **pthread_mutex_unlock()** ще отключи мютекс, ако бъде извикана от притежаващата го нишка. Извикването на тази процедура е необходимо, след като дадена нишка е приключила използването на защитени данни, ако други нишки трябва да получат мютекса за работата си със защитените данни.

Пример за синхронизиране посредством използване на мютекс

- Дефинираме променливи **mutex** и **balance**
- Дефинираме на функция за нишки, където заключваме mutex
- Добавяме сумата от депозита (*Променлива: deposit*) към баланса (*Променлива: balance*)
- Отпечатваме на баланса и отключваме на мютекса
- Създаваме на първа нишка с депозит от 200, извеждаме съобщение при неуспех
- Създаваме на втора нишка с депозит от 200, извеждаме съобщение при неуспех
- Изчакваме двете нишки да приключта своята работа

thread-sync-mutex.c

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

// Global Mutex
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

// Global Variable Balance
int balance = 1000;

// Thread Handler Function to Deposit Money into the Ballance
void *deposit(void *arg)
{
    // Critical Section
    pthread_mutex_lock(&mutex1);

    int deposit_amount = *(int *)arg;
    balance += deposit_amount;
    printf("Balance: $i\n", balance);

    pthread_mutex_unlock(&mutex1);
    // End of Critical Section
}

// Main Functoin
int main(int argc, char *argv[])
{
    int result;
    pthread_t thread1, thread2;
    int deposit_amount = 200;

    // Create Thread
    result = pthread_create(&thread1, NULL, &deposit, &deposit_amount);
    if(result == -1)
    {
        printf("Error creating thread one.\n");
        return EXIT_FAILURE; // -1
    }
    result = pthread_create(&thread2, NULL, &deposit, &deposit_amount);
    if(result == -1)
    {
        printf("Error creating thread two.\n");
        return EXIT_FAILURE; // -1
    }

    // Execute and join the results
    pthread_join(thread1, NULL);
```

Какво е системно програмиране?

```
pthread_join(thread2, NULL);  
  
return EXIT_SUCCESS; // 0  
}
```

Компилиране на програмата с изрична инструкция към компилатора да поддържа нишки:

```
gcc thread-sync-mutex.c -o thread-sync-mutex -lpthread
```

Синхронизиране със семафори

```
#include<semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);

int sem_wait(sem_t *sem);

int sem_post(sem_t *sem);

int sem_destroy(sem_t *sem);
```

- **sem_init()** инициализира неименован семафор на адреса, посочен от аргумента **sem**. Аргументът **value** указва началната стойност на семафора.
- **sem_wait()** декрементира (заклучва) семафора, посочен от аргумента **sem**. Ако в момента семафорът има стойност нула, извикването се блокира.
- **sem_post()** увеличава (отключва) семафора, посочен от аргумента **sem**.
- **sem_destroy()** унищожава неименован семафор на адреса, посочен от аргумента **sem**.

Пример за синхронизиране посредством използване на семафор

- Декларираме семафор (*Променлива: sem*)
- Декларираме баланс (*Променлива: balance*) и го инициираме с 1000
- Дефинираме процедура при работа на нишката, която обработва аргумента и освобождава семафора
- Инициираме семафора със стойност нула (*Означава: Зает*)
- Стартираме първата нишка и предаваме параметър за сума на депозита (*Променлива: deposit_amount*)
- Стартираме втората нишка и предаваме параметър за сума на депозита (*Променлива: deposit_amount*)
- Изчакаме двете нишки да приключат своята работа и унищожаваме семафора

thread-sync-semaphore.c

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

// Global Semaphore
sem_t sem;

// Global Variable Balance
int balance = 1000;

// Thread Handler Function to Deposit Money into the Ballance
void *deposit(void *arg)
{
    // Critical Section
    sem_post(&sem);

    int deposit_amount = *(int *)arg;
    balance += deposit_amount;
    printf("Balance: $%i\n", balance);

    sem_wait(&sem);
    // End of Critical Section
}

// Main Functoin
int main(int argc, char *argv[])
{
    int result;
    pthread_t thread1, thread2;
    int deposit_amount = 200;

    // Semaphore Initialization
    sem_init(&sem, 0, 0);

    // Create Thread
    result = pthread_create(&thread1, NULL, &deposit, &deposit_amount);
    if(result == -1)
    {
        printf("Error creating thread one.\n");
        return EXIT_FAILURE; // -1
    }
    result = pthread_create(&thread2, NULL, &deposit, &deposit_amount);
    if(result == -1)
    {
        printf("Error creating thread two.\n");
        return EXIT_FAILURE; // -1
    }
}
```

Какво е системно програмиране?

```
}

// Execute and join the results
pthread_join(thread1, NULL);
pthread_join(thread2, NULL);

// Semaphore Destroy
sem_destroy(&sem);

return EXIT_SUCCESS; // 0
}
```

Компилиране на програмата с изрична инструкция към компилатора да поддържа нишки:

```
gcc thread-sync-semaphore.c -o thread-sync-semaphore -lpthread
```

Упражнение за синхронизация на нишки

Упражнете материалите от темата, като реализирате представения по-долу проект.

AdvancedChat

Модифицирайте двойката програми мрежов сървър/клиент от предишната тема със следните промени:

- Сървърът трябва да поддържа допълнителен незадължителен аргумент **-p**, който, ако е наличен, определя, че комуникацията ще се осъществява в така наречения **любезен режим** (*Термин на Английски език: polite mode*). В този режим сървърът трябва да приема съобщенията на клиентите на кръгове: когато един клиент изпрати съобщение, следващото съобщение от същия клиент няма да бъде обработено, докато всички други свързани клиенти не изпратят своите съобщения. Ако е необходимо, можете да промените протокола за комуникация между клиентите и сървъра.
- Сървърът трябва да съхранява историята на чата във файл **chat.log**.

Част 10 – Демони

- [Какво са демоните?](#)
- [Скелет на демон](#)
- [Чат демон](#)

Какво са демоните?

Демон (Термин на Английски език: *Daemon*) е процес, който работи във фонов режим, без да се свързва с управляващ терминал.

Обикновено **демоните** се стартират по време на зареждане на операционната система, изпълняват се като `root` или друг специален потребител (например `apache` или `postfix`) и изпълняват задачи на системно ниво. По традиция името на демона често завършва на `d` (като `crond` и `sshd`), но това не е задължително или дори универсално.

Демонът има две общи изисквания:

- трябва да се изпълнява като дете на `init`;
- не трябва да е свързан с терминал.

Скелет на демон

Необходими стъпки за демонизиране на процес:

1. Разклонете родителския процес (*Използваме: `fork`*) и го оставете да се прекрати, ако разклоняването е успешно и тъй като родителският процес е прекратен, дъщерният процес сега работи във фонов режим.
2. Създайте нова сесия и получите идентификатор (*Използваме: `setsid`*)
Извикващият процес става лидер на новата сесия и лидер на групата процеси на новата група процеси. Процесът вече е отделен от управляващия го терминал (СТТУ).
3. Уловете сигналите, като ги игнорирате и/или обработвате.
4. Разклонете отново (*Използваме: `fork`*) и оставете родителския процес да се терминира, за да сте сигурни, че сте се отървали от водещия процес на сесията. (Само водещите сесии могат да получат отново ТТУ.)
5. Променете работната директория на демона (*Използваме: `chdir`*).
6. Променете маската на файловия режим в съответствие с нуждите на демона (*Използваме: `umask`*).
7. Затворете всички отворени файлови дескриптори, които могат да бъдат наследени от родителския процес (*Използваме: `close`*).

daemonize.c

```
/*
 * daemonize.c
 *
 * This example daemonizes a process, writes a few log messages,
 * sleeps 20 seconds and terminates afterwards.
 *
 * This is an answer to the stackoverflow question:
 * https://stackoverflow.com/questions/17954432/creating-a-daemon-in-linux/17954432
 * Fork this code: https://github.com/pasce/daemon-skeleton-linux-c
 */

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

static void skeleton_daemon()
{
    pid_t pid;

    /* Fork off the parent process */
    pid = fork();

    /* An error occurred */
    if (pid < 0)
        exit(EXIT_FAILURE);

    /* Success: Let the parent terminate */
    if (pid > 0)
        exit(EXIT_SUCCESS);

    /* On success: The child process becomes session leader */
    if (setsid() < 0)
        exit(EXIT_FAILURE);

    /* Catch, ignore and handle signals */
    //TODO: Implement a working signal handler */
    signal(SIGCHLD, SIG_IGN);
    signal(SIGHUP, SIG_IGN);

    /* Fork off for the second time*/
    pid = fork();

    /* An error occurred */
    if (pid < 0)
        exit(EXIT_FAILURE);
```

```
/* Success: Let the parent terminate */
if (pid > 0)
    exit(EXIT_SUCCESS);

/* Set new file permissions */
umask(0);

/* Change the working directory to the root directory */
/* or another appropriated directory */
chdir("/");

/* Close all open file descriptors */
int x;
for (x = sysconf(_SC_OPEN_MAX); x>=0; x--)
{
    close (x);
}

/* Open the log file */
openlog ("firstdaemon", LOG_PID, LOG_DAEMON);
}

int main()
{
    skeleton_daemon();

    while (1)
    {
        //TODO: Insert daemon code here.
        syslog (LOG_NOTICE, "First daemon started.");
        sleep (20);
        break;
    }

    syslog (LOG_NOTICE, "First daemon terminated.");
    closelog();

    return EXIT_SUCCESS;
}
```

Източници

1. [How to Create a Daemon in C?](#)
2. [Basic skeleton of a linux daemon written in C](#)

Чат демон

Настоящата програма има за цел да демонстрира чат клиент-вървър приложение, което използва сокети и работи на фонов режим.

chat-daemon.c

```
// Headers
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <syslog.h>
#include <signal.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

// Common Constants
#define MAX_CLIENTS 100
#define BUFFER_SZ 2048
#define NAME_LEN 32

// Common Variables
static int client_count = 0;
static int uid = 10;
int option = 1;
int listenfd = 0, connfd = 0;
struct sockaddr_in serv;
struct sockaddr_in client;
pthread_t tid;
pthread_mutex_t clients_mutex = PTHREAD_MUTEX_INITIALIZER;

// Structure of Client
typedef struct {
    struct sockaddr_in address;
    int sockfd;
    int uid;
    char name[NAME_LEN];
}
Client_t;

// Array of Clients
Client_t * clients[MAX_CLIENTS];

// Add Client
void add_client(Client_t * cl) {
    pthread_mutex_lock( & clients_mutex);
    for (int i = 0; i < MAX_CLIENTS; i++) {
        if (clients[i] == NULL) {
```

```

        clients[i] = cl;
        break;
    }
}
pthread_mutex_unlock( & clients_mutex);
}

// Remove Client
void remove_client(int uid) {
    pthread_mutex_lock( & clients_mutex);
    for (int i = 0; i < MAX_CLIENTS; i++) {
        if (clients[i] != NULL && clients[i] -> uid == uid) {
            clients[i] = NULL;
            break;
        }
    }
    pthread_mutex_unlock( & clients_mutex);
}

// Send Message
void send_message(char * msg, int uid) {
    pthread_mutex_lock( & clients_mutex);
    for (int i = 0; i < MAX_CLIENTS; i++) {
        if (clients[i] != NULL && clients[i] -> uid != uid) {
            int wrt_status = write(clients[i] -> sockfd, msg, strlen(msg));
            if (wrt_status < 0) break;
        }
    }
    pthread_mutex_unlock( & clients_mutex);
}

// Handle Client
void * handle_client(void * arg) {
    char buffer[BUFFER_SZ];
    char name[NAME_LEN];
    int leave_flag = 0;
    client_count++;

    Client_t * cli = (Client_t * ) arg;

    // Joined
    sprintf(name, "%s:%i", inet_ntoa(cli -> address.sin_addr), ntohs(cli -> address.sin_port));
    strcpy(cli -> name, name);
    sprintf(buffer, "%s joined!\n", cli -> name); // Write to buffer
    send_message(buffer, cli -> uid); // Print to all other clients
    bzero(buffer, BUFFER_SZ); // Clear buffer

    while (1) {

```



```
    if (leave_flag) break;

    int receive = recv(cli -> sockfd, buffer, BUFFER_SZ, 0);
    if (receive > 0) {
        // Message
        char buff[BUFFER_SZ];

        // Attempts to overcome limitations: https://developers.redhat.com/blog/2017/01/12/avoid-buffer-overflow-when-using-strncat/
        strncpy(buff, cli -> name, BUFFER_SZ - 1);
        buff[BUFFER_SZ - 1] = '\0';
        size_t n = strlen(buff);
        strncat(buff, "> ", BUFFER_SZ - n - 1);
        n = strlen(buff);
        strncat(buff, buffer, BUFFER_SZ - n - 1);

        // Print & Send
        send_message(buff, cli -> uid);
    } else if (receive == 0 || strcmp(buffer, "exit") == 0) {
        // Left
        sprintf(buffer, "%s left!\n", cli -> name);
        send_message(buffer, cli -> uid);
        leave_flag = 1;
    } else {
        leave_flag = 1;
    }
    bzero(buffer, BUFFER_SZ);
}
close(cli -> sockfd);
remove_client(cli -> uid);
free(cli);
client_count--;
pthread_detach(pthread_self());

return NULL;
}

// Daemonize
static void daemonize() {
    // Fork off the parent process
    pid_t pid = fork();

    // An error occurred
    if (pid < 0) exit(EXIT_FAILURE);

    // Success: Let the parent terminate
    if (pid > 0) exit(EXIT_SUCCESS);

    // On success: The child process becomes session leader
```

```
if (setsid() < 0) exit(EXIT_FAILURE);

// Catch, ignore and handle signals
signal(SIGCHLD, SIG_IGN);
signal(SIGHUP, SIG_IGN);

// Fork off for the second time
pid = fork();

// An error occurred
if (pid < 0) exit(EXIT_FAILURE);

// Success: Let the parent terminate
if (pid > 0) exit(EXIT_SUCCESS);

// Set new file permissions
umask(0);
int stdiofd = open("/dev/null", O_RDWR);
dup(stdiofd);
dup(stdiofd);

// Open the log file
openlog("chat-daemon", LOG_PID, LOG_DAEMON);
}

// Main Method
int main(int argc, char * argv[]) {
    // Arguments check
    if (argc != 2) {
        printf("Syntax: ./chat-daemon [port]\n");
        return EXIT_FAILURE;
    }

    // Socket settings
    listenfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    serv.sin_family = AF_INET;
    serv.sin_addr.s_addr = INADDR_ANY;
    serv.sin_port = htons(atoi(argv[1]));

    // Bind
    if (bind(listenfd, (struct sockaddr *) & serv, sizeof(serv)) < 0) {
        printf("ERROR: bind()");
        return EXIT_FAILURE;
    }

    // Listen
    if (listen(listenfd, 10) < 0) {
        printf("ERROR: listen()");
    }
}
```

```
    return EXIT_FAILURE;
}

// Daemonize
daemonize();

// Working
while (1) {
    socklen_t client_len = sizeof(client);
    connfd = accept(listenfd, (struct sockaddr *) & client, & client_len);

    // Check for max clients
    if (client_count + 1 == MAX_CLIENTS) {
        syslog(LOG_NOTICE, "Max clients connected! Connection rejected!\n");
        close(connfd);
        continue;
    }

    // Client settings
    Client_t * cli = (Client_t *) malloc(sizeof(Client_t));
    cli -> address = client;
    cli -> sockfd = connfd;
    cli -> uid = uid++;

    // Add client to queue
    add_client(cli);
    pthread_create( & tid, NULL, & handle_client, (void *) cli);

    // Reduce CPU usage
    sleep(1);
}

closelog();
return EXIT_SUCCESS;
}
```

Стартиране на чат демона

Server:

```
gcc chat-daemon.c -o chat-daemon -lpthread
./chat-daemon 5005
```

Clients:

```
nc 46.10.253.12 5005
```