

**1. Функции и структура на операционните системи, Понятие:** Комплексът от програмни модули, който създава първото (най-долно) ниво на абстракция в компютърната система е операционна система – ядро. **Функции на ОС:** Разширява възможностите на машината и управлява нейните ресурси. ОС реализира абстрактни понятия и операции за работа с тях - реализира системните извиквания (system calls, системни примитиви, системни функции). Системните примитиви са програмен интерфейс на ОС и са входове в ядрото. Абстрактните понятия са 3, включват: **-Процес:** (process, задача, task) е програма в хода на нейното изпълнение; **-Файл:** осигурява унифицирани операции за вход и изход, т.е. операции, които не зависят от входно-изходните устройства, съхраняващи данните. **-Каталог:** (directory) осигурява удобна организация на файловете. **-Ресурси:** ОС трябва ефективно да управлява ресурсите на машината, като ги разпределя между много програми или потребители (процеси), състезаващи се за правото да ги използват. Разпределяне във времето: Процесите използват ресурса последователно един след друг. Разпределяне в пространството: Ресурса е разделен на части и всеки процес получава част от него. (оперативна памет, дискова памет) **Структури:** Съществуват два подхода за структуриране на ядрото:**-Монолитна структура:** ядрото представлява един изпълним файл, който съдържа всички модули – всеки един модул има строго определен интерфейс – например в UNIX и LINUX**-Иерархична (слоеста) структура:** ядрото се разделя на слоеве, като всеки слой осигурява операции, чрез които по-горният слой се обръща към него – например в MINIX. Смисълът е, че всеки слой осигурява функции или операции чрез които този слой комуникира с по-горния слой. Най-общата схема на една компютърна система е следната:

|                                 |
|---------------------------------|
| приложни системи                |
| командни интерпретатори (shell) |
| обслужващи програми             |
| на sysadmin                     |
| системи за програмиране         |
| ОС – ядро                       |
| хардвер                         |

ОС. Следващият слой е от четири части, които са неразличими за ОС-ядрото. Първите три слоя се различават по кръга от потребители, които ги използват. Обслужващите програми са предназначени за обикновените потребители – например текстови редактори, програми за обработка на файлове и др. **-Програмите на sysadmin** осигуряват средства за настройка и проверка на състоянието на ОС. Системите за програмиране са инструменти за създаване на софтуер – среди за програмиране, свързващи редактори и др.

**-Командният интерпретатор** е свързващо звено в слоя – той реализира диалога с потребителя. Той приема и изпълнява заявки от потребителя, чрез които необходимите случаи се обръща към останалите три компоненти на слоя. Интерфейсът за комуникацията между командния интерпретатор и потребителя се осигурява с **команден език**.

По-подробно ядрото на **MINIX** се структурира по следния начин:

|                                     |          |                |     |
|-------------------------------------|----------|----------------|-----|
| shell                               | init     | vi             | ... |
| file system                         |          | memory manager |     |
| disk task                           | tty task | clock task     | ... |
| микроядро (управление на процесите) |          |                |     |

Най-долният слой управление на процесите изгражда абстрактният процес и осигурява механизъм за комуникация между различните процеси – той работи по същия начин както ядрото на UNIX. Следващият слой представлява различни входно-изходни задачи за всяка от които има определен драйвер, който се изпълнява като отделен процес. Под terminal се разбира например монитор или клавиатура. Чрез по-горния слой се реализират системните примитиви. Частта file system реализира системни примитиви с файлове, а останалите системни примитиви се реализират от memory manager. Основната идея при такова разпояване е част от функциите да се изнесат извън ядрото на ОС и да се реализират като независими процеси.

**Класификация на операционните системи: -Еднопотребителски:** може да се използва само от един потребител.**-Многопотребителски:** може да се използва едновременно от няколко потребители.

Друга класификация е следната:**-Еднопроцесни:** в даден момент ядрото поддържа работата само на един процес.**-Многопроцесни:** ядрото поддържа в един момент работата на няколко процеса - един потребител може да изпълнява няколко процеса в даден момент. Естествено многопотребителските операционни системи трябва да са многопроцесни.

**3. Основни команди в LINUX, Команди: -За помощ: man [име на команда]** – за какво служи командата която ни интересува; **Echo: echo [-n] [string]** – извежда на std. изход своя вход. Може да се използва с променливи по следния начин: **echo \$име на променлива** – извежда значението на променливата. По отношение на аргументите на echo важат правилата за екраниране на метасимволи, чрез използване на кавички; **-За работа с файлове: cat [списък от файлове]** – извежда на std. изход съдържанието аргументите си или std. вход последователно. Командите **more** и **less** имат аналогичен синтаксис и действие. При more на всеки 24 реда потребителя се подканва да въведе интервал. Това е полезно, ако изведеният текст е повече от един екран. За разлика от more, където може да се придвижваме само напред в извеждания текст, при less е възможно връщане назад; **rm [-f] [списък от файлове и/или каталози]** – Тя изтрива указаните файлове. Ако към някой файл има твърда връзка, то се намалява неговия брояч на твърдите връзки без да се изтрива файла. По подразбиране преди изтриването на всеки файл се извежда съобщение за потвърждение, то може да се избегне с опция **-f** (от force). -г трие всичко в поддървото (ако няма -г и не е празна директорията не я трие); **cp [-f] [failn1 failn2] или [списък от файлове] [каталог]** – При първия вариант тя прави копие на failn1 с име failn2. Ако failn2 съществува, то се извежда подканващо съобщение дали да се презапише файла. Възможно е това съобщение да не се извежда, ако използваме ключов аргумент **-f**. При втория вариант, командата прави копия на файловете в указания каталог, при това със същите имена; **mv [failn1 failn2] или [списък от файлове] [каталог]** – като ср но мести; **ln [-s] [failn1 failn2]** – създава твърда връзка между failn1 и failn2; **-За работа с директори: mkdir [-p] [списък от каталози]** – създава каталози с указаните имена в текущия каталог, ако файловете с такива имена не съществуват. Чрез ключов аргумент **-p** (от mode) може да се задава код на защита за новосъздадените каталози; **rmdir [списък от каталози]** – изтрива указаните каталози от текущия каталог. Изтриването е успешно, само ако каталозите са празни; **ls [-laR] [списък от файлове и/или каталози]** – аргументите може да са относнителни или абсолютни имена на файлове или на каталози. Ако даден аргумент е файл, тя извежда името на файла (ако съществувал), ако е каталог извежда неговото съдържание (файловете в каталогa). По-подробна информация (кода на защита, размер и т.н.) за файловете може да се види с опция **-l**. При преглеждане на каталози, опцията **-R** (от recursive) означава рекурсивно обхождане на подкаталозите на указания каталог; **-За достъп: chown [-R] [име на потребител/група] [списък от файлове и/или каталози]** – сменя достъпа до файла/каталогa; **chmod [код за достъп] [списък от файлове и/или каталози]; -За манипулация на текстови файлове: vi [файл]; sort [списък от файлове]** – слива съдържанието на указаните файлове, след това сортира лексикографски полученото по редове и най-накрая извежда сортираната последователност от редове на std. изход; **uniq [списък от файлове]** – Слеща файловете и премахва повтарящи се редове; **cut [-cM-N] [списък от файлове]** – извежда определена част от всеки ред на всеки от указаните файлове. Коя част се извежда се задава с опция **-c** със значение числов интервал със следния формат N-M. N задава първата позиция, M задава последната позиция на байтовете за извеждане от всеки ред; **grep [шаблон] [списък от файлове]** – Тя търси шаблонът във всеки от указаните файлове. Всеки път когато командата открие шаблона в някой от файловете, тя извежда на стандартния изход целия ред от файла, в който е намерен шаблона. В шаблона могат да се използват следните символи: -всички символи, които нямат специално значение, например латинските букви и цифрите; **символът " "** означава кой да е символ; **"последователност** означава символ, ограден в [...]" означава точно един от тези символи; последователност от символи, оградени в [...] означава само един от тези символи; **символът "\*"**  означава начало на ред, **символът "\$"**  означава край на ред; **ws [списък от файлове]** – Тя извежда в стандартния изход броя на байтовете (символите), думите и редовете във всички указани файлове поотделно. Възможно е да се извежда само част от информацията, като се задават опции **-c, -i** или **-l**. Те съответстват на брой байтове, брой думи и брой редове; **file failn** – извежда информация за файл; **-Други: who** – извежда кои потребители в момента са в сесия; **ps** – Тя извежда информация за съществуващите процеси в системата. По подразбиране ps връща информация само за процесите, които принадлежат на потребителя, изпълняващ ps. С опция **-A** се извеждат абсолютно всички процеси; **sleep N** – изчака N секунди; **ty** – името на терминала; **cc** – компилира файл, с -o се задава конкретно име на новия.

**Допълнително:** Самият shell прави анализ на аргументите и ги предава към съответната процедура за изпълнение на командата. В някои случаи shell прави преобразуване на аргументите – така нареченото **разширение**. Ако в аргументите shell срещне специални символи (**метасимволи**), shell ги замества с това, което означава преди да ги предаде на командата.Аргументите се отделят от името на командата с помощта на разделител. Ако аргументите са повече от един, те също се отделят един от друг с разделители. В LINUX единственият разделител, който може да се използва е един или няколко интервала. Аргументите биват два вида: **-Позиционните аргументи** са основни аргументи за командата и мястото, където се поставят те е съществено. **-Ключовите аргументи** (опции, флагове) носят своето име при задаването си. Те могат да се поставят на произволно място в списъка с аргументите.

**2. Командни езици. Принципи на действието на командния интерпретатор, Исканения към командните езици: Дружелюбеност; Изразителност:** Командният език има характеристики на език за програмиране: променливи, управляващи команди (оператори).На командния език се пишат програми – shell script, shell файл, команден файл, командна процедура. Командната процедура се интерпретира от програмата-shell; **Лесно разширение:** В командния език лесно се добавят нови команди.

**Видове команди:** Командите се делят на два вида според реализацията – вътрешни и външни. **Вътрешните команди** се реализират от програмния код на командния интерпретатор, докато **външните команди** са извън този код. На всяка външна команда съответства файл. В зависимост от съдържанието на този файл външните команди се поделят на два вида: **1)** Файлът съдържа изпълним код – например в MSDOS такива файлове имат разширение .exe или .com; **2)** Файлът съдържа командна процедура, т.е. програма на команден език – в MSDOS това са .bat файловете. В UNIX/LINUX няма специални разширения, подобно на MSDOS, чрез които да се различават описаните два вида файлове. **Друга класификация** на командите е според действието – обикновени и управляващи. Пример за обикновени команди са копиране на файл, преименуване на файл, извеждане на точно време и дата и т.н. Управлящите команди още се наричат оператори. Те са специални конструкции на командния език чрез които могат да се създават по-сложни командни процедури – например условен оператор, оператори за цикъл, оператор за модифициран избор и т.н. ...Наличието на такива оператори е причина командните езици да се разглеждат и като езици за програмиране. В различните операционни системи някои обикновени команди се реализират като вътрешни, други като външни. Управлящите команди, обаче, са винаги вътрешни.

**Принцип на действието:** Първото нещо което извършва **login** shell процесът е да **инициализира** средата в която ще се работи – това се осъществява от една или повече командни процедури, които се наричат **profile** процедури. В MSDOS подобна инициализация се осъществява от командната процедура във файла autoexec.bat. По-нататък се извежда комбинация от символи, наречени **prompt** – по същество това е подканващо съобщение на login shell, което означава че той е готов да изпълнява нова команда. Например в bash shell тези символи са \$ за обикновен потребител или # за администратор. В UNIX/LINUX има единствен потребител, който е администратор и неговото име е root. Съществуват начини за промяна на std. prompt от самия потребител. След появяване на prompt потребителят въвежда инструкция на съответния команден език и shell изпълнява командата. След това отново се извежда prompt и т.н. Този цикъл може да се приключи с помощта на специална команда **logout**, която прекратява login shell процесът. В много shell-ове се използва и команда **exit**, която, за разлика от logout, освен за прекъсване на login shell процеса може да се използва за прекъсване и на sub shell процеси. Цялата работа която системата изпълнява от стартирането на login shell процесът до неговото прекъсване (чрез logout или exit) наричаме **сесия**. Така в рамките на една сесия потребителят може да изпълнява различни команди.

**1)** while(не е край на сесия) { извежда покана; четe команда; if(командата е вътрешна) вътрешна\_команда(); else { по името на командата намира файла; if(файлът съдържа командна процедура) превключва входа на текущия процес-shell от файла; else { създава процес за изпълнението код във файла; if(режима не е фонов) wait(); } } } Командната процедура може да смени обкръжението на процеса-shell. Не може асинхронно (фоново) изпълнение на командна процедура. По-икономичен е. **2)** while(не е край на сесия) { извежда покана; четe команда; if(командата е вътрешна) вътрешна\_команда(); else { по името на командата намира файла; if (файлът съдържа командна процедура) създава процес-subshell, който чете входа от файла; else създава процес за изпълнениия код във файла; if (режима не е фонов) wait(); } } Командната процедура не може да смени обкръжението на текущия процес-shell. Може асинхронно изпълнение на всяка външна команда. Не е така икономичен. Рекурсивни командни процедури.

**Допълнително:** Съвременните командни езици осигуряват **диалогов** режим на работа -потребителят може да се намесва в процеса на изпълнение на заданията. Възможно е да се работи в така наречения **фонов** или **асинхронен** режим (background), при който процесите се изпълняват независимо от диалога на командния интерпретатор с потребителя. В UNIX/LINUX командният интерпретатор се нарича shell. Съществуват различни версии shell-ове за UNIX/LINUX – B shell, C shell, Korn shell, Bash shell. Bash shell се появява заедно с LINUX. За ядрото на ОС командният интерпретатор е обикновена програма. Файлът, който съдържа изпълнимият код на командния интерпретатор в LINUX се нарича **bash**. В ОС MSDOS този файл се нарича command.com. Общото между всички командни интерпретатори е, че осигуряват интерактивен режим на работа с потребителя. Командните езици могат да се разглеждат като езици за програмиране от високо ниво – на тях могат да се пишат програми, наречени командни процедури. В UNIX/LINUX се наричат shell file или shell script. В MSDOS командните процедури са във файловете с разширение .bat. Съществено е, че командните процедури не се компилират, а се интерпретират в съответствие с интерактивния режим на работа с потребителя. Съвременните командни езици осигуряват достъпен интерфейс и са лесни за усвояване от потребителите.

**4. Програмиране на команден език в UNIX и LINUX, Метасимволи:** Метасимволите са символи, които имат специално значение за shell. Когато се срещнат в определена конструкция на командния език, те се интерпретират по специален начин. Под екраниране на метасимволи се разбира отнемане на специалното им значение. Групи метасимволи: за имена на файлове; за пренасочване на вход/изход и конвейер; за режима на изп. на ком.; за групиране на ком.; за променливи; за аритмет. разширение.

**Генериране на списък от имена на файлове:** Ако в конструкция на команден език се среща дума, която би трябвало да е име на файл, но съдържа един или повече от трите специални символа **" , ? , [...] "** тя се разглежда като шаблон. Shell интерпретира този шаблон като го разширява до списък от файловете в заданията (текущия по подразбиране) каталог, които отговарят на шаблона. При сравняване на файл с шаблон специалните символи имат следната семантика: **\*** означава произволен брой произволни символи, **?** означава точно един на брой произволен символ, **[низ]** означава точно един измежду символите в низа, **[\*...]** означава точно един символ, който не е измежду символите в низа. Единственото изключение от тези правила е, че **" ? " и ? "**  могат да съответстват на **" "** в началото на низ. В UNIX/LINUX е прието файловете, чиито имена започват с **" "** да се считат за скрити.

**Пренасочване на вход и изход:** Много от командите извеждат данни в терминала (монитора) или въвеждат данни от терминала (клавиатурата). По-общо всеки процес получава със своето създаване три стандартно отворени файлове – файл за **стандартен вход**, файл за **стандартен изход** и файл за **стандартен изход за грешки**. Във файла за стандартен изход за грешки се извеждат съобщения за грешки по време на изпълнението на командите. Стандартно трите файла са свързани с терминала – стандартният вход е свързан с клавиатурата, а стандартният изход и стандартният изход за грешки са свързани с монитора. Под **пренасочване** разбираме свързване на трите стандартни файла с други файлове (обикновени файлове на диска) или свързване на тези файлове с други устройства или терминали. За всеки файл, който се отваря от някакъв процес ядрото на операционната система създава **файлов дескриптор** – идентификатор (цяло неотрицателно число), който се подава на процеса и се използва от системните примитиви при работа с файла. Първите три файлови дескриптора 0, 1, 2 стандартно са свързани съответно с горно изброените. **-Пренасочвания:** на stdin **команда < файл**; на stdout **команда > файл** – По този начин тя се извежда в указания файл. Ако този файл не съществува, той ще бъде създаден, ако съществува съдържанието му ще се загуби. Във втория случай е възможно да се запази съдържанието на файла по следния начин: **команда >> файл**. По-общо конструкцията **n< файл, n> файл**, където n е файлов дескриптор отвара файлът за четене и го свързва с дескриптор n.

**Конвейер:** команда1 | команда2 | [команда3 ...] Пренасочва стандартния изход на команда1 към стандартния вход на команда2, стандартния изход на команда2 към стандартния вход на команда3 и т.н. В UNIX/LINUX това се реализира с конкурентни процеси, които се изпълняват едновременно и използват програмния канал (**pipe**) за комуникация помежду си.

**Фонов режим на изпълнение:** При **фонов режим (background)**, след като се стартира изпълнението на дадена външна команда интерпретаторът не я изчака да завърши, затова този режим се нарича още **асинхронен**. По подразбиране режимът на изпълнение е привилегирован, а за да изпълним една команда асинхронно използваме **&** по следния начин: **команда&**. Ако командата използва стандартен вход, то той трябва да бъде пренасочен. Същото се препоръчва и за стандартния изход и изхода за грешки. За входа изискването е съществено, тъй като изпълнението на командата може да бъде блокирано.

**Код на завършване:** Всяка команда при завършването си изработва код на завършване (exit status) и го връща към shell. Код 0 означава нормално (успешно) завършване на командата. Код различен от 0 (число от 1 до 255) означава грешка или изключителна ситуация (неуспех). Кодът се използва в условните конструк-ции на командния език.

**Списък от команди:** Възможно е няколко команди да се обединяват в списъци: **ком1; ком2 [ком3; ...]** Shell изпълнява командите от списъка последователно. За разлика от конвейера, в последователния списък между командите няма логическа връзка. Списъкът за асинхронно изпълнение има следния синтаксис: **ком1 & ком2 [ком3 & ...]** & Shell стартира ком1 асинхронно, след това ком2 асинхронно и т.н. Ако в реда има знак &, тогава Shell е асинхронен с всички команди. В противен случай Shell се синхронизира с последната команда в списъка. Shell поддържа два вида списъци за условно изпълнение на команди: **ком1 && ком2** – изпълнява се ком1 и ако тя издаде код на завършване равен на 0, се изпълнява ком2, в противен случай не се изпълнява ком2; **ком1 || ком2** – изпълнява се ком1, и ако тя издаде код на завършване, различен от 0, се изпълнява ком2, в противен случай не се изпълнява ком2. Конструкцията (**списък от команди**) има две приложения. **1)** е за обединяване на изходите на командите от списъка. **2)** е за приоритет на дадени операции за списъци пред други. По подразбиране операцияте за списък за условно изпълнение имат приоритет пред операцияте за списък за последователно и за асинхронно изпълнение. (**списък от команди**) се изпълнява по следния начин: текущият shell процес създава нов sub shell процес и този нов процес изпълнява командите в списъка. Смыслът на тази конструкция е, че командите променят обкръжението на shell процеса, а ние искаме тези промени да не засягат текущия shell процес. В обкръжението на един shell процес се включва, например, текущият каталог – ядрото поддържа независим текущ каталог за всеки shell процес

**5. Программиране на команден език в UNIX и LINUX.** Присвояване на значение на променлива: Промениливите в shell имат име, значение и евентуално атрибут. Името е символен низ от букви, цифри и символа за подчертаване, като започва с буква или символа " ". Значението е символен низ. Декларирането не е задължително и може да става при присвояване на значение. Промениливите са част от обкръжението на процес shell. Променилива се изключва от обкръжението на процес shell с командата unset. Присвояването има следния синтаксис: **име\_на\_променлива=значение**. Значението е символен низ, който е ограден или не е ограден в кавички ("...", "..."). Кавичките са метасимволи за задаване на границите на значението и те не се включват в него. Ако в значението присъстват метасимволи, тогава кавичките имат следното действие: -единичните кавички отменят действието на всички метасимволи в значението без себе си; -двоините кавички отменят действието на всички метасимволи без \$, %, \, " , ' .

**Заместване на променлива:** При използването на променлива тя се замества с нейното значение. Използват се следните две конструкции: **\$име\_на\_променлива** и **\${име\_на\_променлива}**. Ако променливата е обявена и двете конструкции се заменят с нейното значение, ако не е обявена се заменят с празен низ. Името на променливата се огражда в {...}, ако е необходимо явно да се посочи на shell къде е края на името на променливата. В по-новите shell има конструкции за условно заместване на променлива с нейното значение. **\$име\_на\_променлива:==[+/? значение]** – ако променливата е неопределена или има значение празен низ: (==) тогава на нея се присвоява указаното значение след което цялата конструкция се замества с това значение. (+=) цялата конструкция се замества с указаното значение, иначе конструкцията се замества със значението на променливата. (+) нищо не се замества, иначе цялата конструкция се замества с указаното значение (?) се изписва значението на изхода за std. грешка; **\$име\_на\_променлива:отместване[:брой]** – отместване и брой са цели положителни числа. Конструкцията се заменя с подниз от значението на променливата. Отместването задава първия символ на подниза, като номерацията започва от 0, а броят задава дължината на подниза. Ако броят не е допустим (отместване+брой > дължината на значението на променливата) или не присъства, тогава поднизът достига до последния символ на значението.

**Команди за променливи: read** списък **с имена на променливи** – чете един ред от стандартния вход и го разделя на думи, като за разделител между думите се приема интервал. Аргументите на read са позиционни – на указаните променливи последователно се присвояват прочетените думи. Ако броят на думите е по-малък от броя на променливите, на последните променливи се присвоява празен низ. Обратно, ако броят на променливите е по-малък от броя на думите, то на последната променлива се присвоява целият остатък от прочетената ред: **echo *hi* *hi* *hi* ...** – извежда на стандартния изход своите аргументи. Може да се използва за променливи по следния начин: **echo \$име\_на\_променлива** – извежда значението на променливата. По отношение на аргументите на echo важат правилата за екраниране на метасимволи, чрез използване на кавички; **set** – аргументи извежда информация (име и значение) за всички променливи в текущия shell процес; **unset** списък **с имена на променливи** – изключва указаните променливи от обкръжението на текущия shell процес; **declare [-x] (списък с имена на променливи)** -x атрибут export; -t атрибут readonly; -i атрибут integer (с + става логическо ограничение, по закони на ДеМорган).

**Системни променливи: HOME** – Пълно име на началния каталог на потребителя; **PATH** – Списък от каталози, в които shell търси външни команди; **PS1** – Първична покана (prompt) на shell; **PS2** – Вторична покана на shell стандартно е "> "; **UID** (User id) – Вътрешен идентификатор на потребителя (readonly); **IFS** (Internal Field Separator) – Символите, които се разглеждат като разделители на думи при word splitting и командата read. Стандартно са " <space><tab><newline> "; **PWD** – Име на текущия каталог; **OLDPWD** – Име на предишния текущ каталог; **USER** – Име на потребителя; **SHELL** – Име на потребителския login shell; **BASH** – Име на файла с изпълнимия код на bash, стандартно е /bin/bash; **BASH VERSION** – Версия на bash

**Инициализиращи команди процедури (profiles):** В термините на UNIX/LINUX тези процедури се наричат **profile**. При стартиране на всеки login shell процес се изпълнява главния profile с пълно име /etc/profile, ако той съществува. Самият файл е текстов и съдържа командна процедура. След изпълнение на главния profile се стартира \$HOME/.bash\_profile. Той настройва сесиата на конкретния потребител, който е стартирал login shell процеса. В UNIX името на този profile е \$HOME/.profile. Когато се създава интерактивен shell процес, който не е login shell се изпълнява \$HOME/.bashrc. За неинтерактивни процеси името на съответния profile се намира в променливата BASH\_ENV в обкръжението на текущия shell процес.

**Заместване на изхода:** Заместването на изхода (command substitution) позволява обръщението към командата да се замести с нейния изход. Синтаксис: **"команда"**. Заместването се осъществява по следния начин: изпълнява се командата и след това цялата конструкция се замества със стандартния изход на командата.

**Аритметични изчисления: expr** [израз] – пресмята изрзаа и извежда резултата на стандартния изход. В изрзаа могат да се използват цифрови нивозе, значения на променливи с метасимвол \$, операциите за изчисление +, -, \*, /, %, операциите за сравнение <, <=, >, >=, =, !=, скобите ( ) и др. Всички операции и операнди в изрзаа са отделни аргументи на командата и затова те трябва да са разделени с интервали. Освен това всички операнди, които са метасимволи трябва явно да се екранират (\*, (, <, >) **Аритметично разширение: \$[израз]** и **\$((израз))** – изчислява се цялостния израз и резултата замества конструкцията. Изразът се конструира от цели числа, променливи, аритметичните операции и скоби. Операциите и приоритета им е както в езика C.

съвпадне с никой от шаблоните. Шаблонът \* може да се използва като else, тъй като той се разширява до произволна дума.

**Функции:** Shell позволява да се дефинират функции за изпълняване на поредица от команди. Функциите приличат на командни процедури, но има една съществена разлика – те се изпълняват в рамките на shell, в който са извикани, т.е. за тях не се създава нов sub shell процес. **[function] име\_на\_функция {** **списък от команди;** **}** Извикване на функция: **име\_на\_функция арг1 арг2 ...** Когато изпълнението на функцията приключи управлението се връща на мястото, където функцията е била извикана и всички специални вътрешни променливи възстановяват предишните си стойности. Кодът на завършване на една функция е кодът на завършване на последната изпълнена команда в тялото на функцията, освен ако не се използва команда **return** [0-255] Тази команда може да се използва само в тялото на функцията. Всички променливи и функции, определени в обкръжението на текущия shell процес могат да се използват в тялото на функцията. Командата **local** има аналогичен синтаксис и изпълнение на **declare** и се използва в тялото на функцията за дефиниране на локални променливи. Тези променливи са достъпни само в тялото на функцията.

**Екраниране:** Метасимволът \ отменя специалното значение на непосредствено следващия символ. Ако \ се постави в края на реда, се отменя специалното значение на Enter и изписването на командата продължава на следващия ред. "..." отменят специалното значение на всички метасимволи, разположени между тях, с изключение на \$ във всичките му форми, ` , \ и " . ' ' отменят специалното значение на всички метасимволи, разположени между тях.

**Заместване и изпълнение на команда:** Ще разгледаме какво се крие зад реда **прочитане\_на\_команда()**; в принципа на действие на shell: 1) извършва синтактичен анализ на командата, независимо от самата команда; 2) извършва разширяване в следния ред: заместване на променливите, заместване на изхода, аритметично разширяване, заместване на метасимволи за генериране на списък от имена на файлове; 3) организира пренасочването на входа и/или изхода, ако това е указано в командния ред; 4) резултатът от разширяването се интерпретира по следния начин: първата дума е името на командата, останалите думи са фактическите параметри; **Съществуването на командата** се проверява в следния ред: 1) търси се функция; 2) търси се вътрешна команда; 3) търси се външна команда в каталозите, указани в системната променлива PATH; Това означава, че функциите покриват вътрешните и външните команди при съвпадение на имената. Тези правила не вадат, ако в командата се задава абсолютно или относително име на файл, тогава се преминава направо към 3.

**7. Логическа структура на файлова система, Имена на файлове:** Името на файл е низ от символи с определена максимална дължина. Името на файл може да се състои от две части, разделени с определен символ. Втората част на името (разширение) носи информация за типа или формата на данните във файла. **Типове файлове:** 1) обикновен файл (regular file) 2) специален файл (character special device file, block special device file) 3) каталог (справочник, directory) 4) символна връзка (symbolic link) 5) механизми за комуникация – програман канал (pipe), FIFO файл.

**Кака е вътрешната структура на файл:** В съвременните операционни системи файлът се разглежда като последователност от байтове. Това опростява структурата на файла и съответно опростява системните примитиви за работа с файлове. Най-съществено е, че такава структура може да се приложи към всички изброени по-горе типове файлове. Избраната вътрешна структура на файл влияе на: системните примитиви read и write и избора на типовете файлове.

**Атрибути на файл:** Информацията, която се съхранява за файла, освен името и данните? 1) размер в байтове, записи, блокове 2) дата и време на създаване, последен достъп, последно изменение и др. 3) собственик, права на достъп, пароли за достъп 4) флагове - read only, hidden, system, archive, secure deletion, undelete, immutable, append only, compress file.

**Организация на файловата система:** Еднокаталогова файлова система: на всеки диск има един каталог, съдържащ информация за всички файлове на диска. В по-новите операционни системи се изгражда йерархична организация на файловата система (може да са с фиксиран брой нива или с произволен). Имената на файловете трябва да са уникални само в рамките на един каталог.

**Пълно име на файл:** Трябва да има начин за унифициране на името на всеки файл. Абсолютно пълно име (absolute path name) съответства на единствения път в дървото от корена до файла. /home; Относително пълно име (relative path name) съответства на пътя в дървото от текущия в даден момент каталог до файла.

**Сравнение на UNIX/LINUX и MSDOS: UNIX, LINUX:** Имена на файловете до 14с или 255с, може с няколко разширения. Прави се разлика между малки и главни букви. Типовете са обикновени, каталози, специални и др., като имената им са вградени в структурата на ФС. Системите имат единна йерархия за всички файлове и за всички устройства. Специалните файлове са разположени в каталог с име dev. Поддържа се текущ каталог за всеки отделен процес. Името на главния каталог е /. **MSDOS:** Имената са до 8с върва част и до 3с разширение, без разлика между малки и главни букви. Типовете файлове са обикновени, каталози, специални, но имената на специалните не са вградени в структурата на ФС. За всяко дисково устройство се изгражда собствена йерархична структура със собствен корен. Въвжеждат се имена на различните устройства и се поддържа текущо устройство. Текущ каталог се поддържа само за отделните устройства, но не и за всички процеси. Името на главния каталог е \.

**Командна процедура и аргументи:** Текстов файл (shell script), съдържащ конструкции на командния език. Изпълнението на командните процедури се осъществява от sub shell процес, породен от текущия shell процес, в който са извикани. Когато приключи изпълнението си, всяка командна процедура издава код на завършване. Този код явно може да бъде зададен с помощта на командата exit със следния синтаксис: **exit** [0-255]. В командните процедури могат да се поставят коментари със символа #. Всички символи след символа # се игнорират от командния интерпретатор. **Изпълнение:** Командните процедури се изпълняват по два начина: 1) Командната процедура се изпълнява като външна команда. За целта потребителят трябва да има правата да може да изпълнява съответният файл. За да може да се изпълнява файлът с командната процедура само със задаване на неговото име, той трябва да е поместен в каталозите, указани в системната променлива PATH. В противен случай трябва да се задава негово пълно име. 2) Чрез командата **bash [опции] [име на файл]** Той се използва ако потребителят няма права да изпълнява файла с командната процедура. **Позиционни аргументи:** Имената им са 0, 1, 2, ... Присвояват се съответно с елементите от списъка подаден след процедурата. Ако името на аргумента е съставено от повече от една цифра, то трябва да се огради в {...}. \$# – брой аргументи, предадени на shell без нулевия; \$\*, \$@ – всички аргументи, предадени на shell без нулевия, разделени с интервали. \$? – код на завършване на последната изпълнена команда в привилегирован режим. \$\$ – идентификатор (pid) на процеса shell. \$! – Идентификатор (pid) на последния изпълнен фонов процес. **shift** [n] – има отношение към аргументите на една командна процедура. \$2→\$1, \$3→\$2, и т.н. Значението в \$1 се губи.

## 6. Программиране на команден език в UNIX и LINUX.

**Изполнен оператор: Синтаксис:**

```
if списък от команди 1
then списък от команди 2
[elif списък от команди 3
then списък от команди 4]
...
[else списък от команди N]
fi
```

Ако списъкът команди в if или elif върне код на завършване 0, условието е истина, а ако върне код различен от 0 е лъжа. Кодът на завършване на оператора if съвпада с кода на завършване на последната изпълнена команда, в списъците които не са условия. Ако не е изпълнена нито една такава команда, кодът на завършване на if е 0.

**Команди test: test условие или [условие]** – проверява условието и връща код на завършване 0, ако условието е изпълнено. В противен случай връща код на завършване различен от 0. Условието е логически израз. Има три типа условия – условия за файлове, условия за нивозе и условия за числа. ! усл – отрицание на усл; усл1 –a усл2 – логическо "и"; усл1 –o усл2 – логическо "или". При тези конструкции не могат да се използват скоби и условията се оценяват последователно отляво надясно.

**Условията за файлове** се използват за проверка на различни характеристики на файловете: -e файл – изпълнено е, ако файлът съществува; -f файл – изпълнено е, ако файлът е обикновен; -d файл – изпълнено е, ако файлът е каталог; -c файл – изпълнено е, ако файлът е символен; -b файл – изпълнено е, ако файлът е блоков; -s файл – изпълнено е, ако файлът не е празен; -t файл – изпълнено е, ако потребителят има права да чете от файла; -w файл – изпълнено е, ако потребителят има права да пише във файла; -x файл – изпълнено е, ако потребителят има права да изпълнява файла; файл1 –nt файл2 – изпълнено е, ако файл1 е по-нов (newer than) от файл2 в съответствие с датата на последната модификация; файл1 –ot файл2 – изпълнено е, ако файл1 е по-стар (older than) от файл2 в съответствие с датата на последната модификация. **Условията за нивозе** се използват за сравняване на нивозе: нюз1 = нюз2 – изпълнено е, ако нюз1 и нюз2 съвпадат; нюз1 != нюз2 – изпълнено е, ако нюз1 и нюз2 не съвпадат; нюз1 < нюз2 – изпълнено е, ако нюз1 е лексикографски по-напред от нюз2; нюз1 > нюз2 – обратното на <; -n нюз – изпълнено е, ако нюзът не е празен; -z нюз – изпълнено е, ако нюзът е празен. **Условията за числа** се използват за срав на числа: n1 –eq n2; n1 –ne n2; n1 –gt n2; n1 –ge n2; n1 –lt n2; n1 –le n2. **Разширена команда test: [[условие]]** В условието са добавени скоби и нови операции в стила на езика C.

**Оператори за цикъл: Синтаксис на while, until и for:**

```
while списък от команди 1
do списък от команди 2
done
```

```
until списък от команди 1
do списък от команди 2
done
```

```
for име на променлива [in списък]
do
done
```

Кодът на завършване на while съвпада с кода на завършване на последната команда, изпълнена в тялото.

Ако не се изпълни нито една команда в тялото, кодът на завършване на while е 0. while работи докато условието връща код 0 а until докато е 1. Операторът for е още една форма за присвояване на значение на променлива, не е необходимо тази променлива да е определена преди това. Броят на итерациите съвпада с броя на думите във фразата in след разширението. Кодът на завършване на оператора for съвпада с кода на завършване на последната команда, изпълнена в тялото на цикъла. Ако такава команда няма, т.е. ако тялото не се изпълни нито веднъж, кодът на завършване на for е 0. break [N] – може да се използва само в тялото на цикъл и води до завършване на изпълнението на N на брой вложени цикъла откъдето навън, като броеното започва от цикъла, в който е поместен break. Кодът на завършване на break е 0, ако break е в тялото на цикъл, else 1. continue [N] – сходно на break, но не прекъсва а продължава.

**Оператор case: Синтаксис:**

```
case дума in
шаблон[шаблон1...])
[шаблон[шаблон1...]) списък от команди;
...
esac
```

Думата се сравнява последователно с шаблоните. При първото съвпадение се изпълнява съответният списък от команди и изпълнението на case приключва. За един списък от команди може да има много шаблони, разделени с |. Кодът на завършване на case е кодът на завършване на последната команда, изпълнена в някой от списъците от команди или 0, ако такава команда не се изпълни, т.е. ако думата не

**8.Системни примитиви за работа с файлове - open, close, creat, read, write, lseek, stat.** Основни понятия: **Файлов дескриптор** (file descriptor): Неотрицателно цяло число, което е идентификатор на отворен файл. **Текуща позиция във файл** (file offset, file pointer): Определя позицията във файла, от която ще бъде четено или записано. **Режим на отваряне:** Опред. начина на достъп до файла чрез съответния файлов дескриптор.

**Системни примитиви за обикновени файлове:** I) **Създаване на файл: int creat(const char \*filename, mode\_t mode);** II) **Отваряне на файл: int open(const char \*filename, int oflag |, mode\_t mode);** Връщат файлов дескриптор на отворения файл, -1 при грешка. **Флагове:** O\_RDONLY, O\_WRONLY, O\_RDWR, O\_CREAT, O\_EXCL, O\_TRUNC, O\_APPEND, O\_SYNC. **Open** може да се използва както за отваряне на съществуващ файл, така и за създаване на нов файл. С **open** могат да се отварят и специални файлове, но могат да се създават само обикновени файлове. Аргументът **filename** задава името на файла – абсолютно, собствено или относително спрямо текущия каталог в текущия процес. Аргументът **flag** е цяло число, което се интерпретира като флагове, задаващи режима на отваряне. Аргументът **mode** се използва само при създаване на файл и той задава код на защита. **Алгоритъм на open:** 1) По името на файла се организира търсене по каталозите и се намира индексния описател на файла. 2) Индексният описател на файла се зарежда в таблицата на индексните описатели, ако вече не е там. 3) Добавя се нов запис в таблицата на отворените файлове, в който се зарежда режима на отваряне от аргумента flag, установява се текуща позиция 0 и брояч на указателите 1. 4) Разпределя се първият свободен запис от таблицата на файловете дескриптори на текущия процес и той се свързва със запис от таблицата на отворените файлове. III) **Затваряне на файл: int close(int fd);** Връща 0 при успех, -1 при грешка. **Алгоритъм на close:** 1) Намалява се с 1 броят на указателите в запис за файла в таблицата на отворените файлове. 2) Ако новото значение на брояча е по-голямо от 0, то се освобождава записът в таблицата на файловете дескриптори и close завършва. 3) Ако новото значение на брояча е 0, то се намалява с 1 броят на отварянията в запис за файла в таблицата на индексните описатели и се освобождават записът в таблицата на отворените файлове и записът в таблицата на файловете дескриптори. 4) Ако новото значение на брояча на отварянията е 0, то индексният описател на файла се записва обратно на диска, ако е бил изменен и се освобождава записът в таблицата на индексните описатели. IV) **Четене и писане във файл: ssize\_t read(int fd, void \*buffer, size\_t nbytes);** Аргументът fd задава файлов дескриптор на отворен файл, от който ще се чете. Аргументът буf задава адрес в паметта, където ще се записват прочетените данни. Аргументът count задава брой байтове, които ще се прочетат. Връща действителния брой прочетени байта, 0 при EOF, -1 при грешка. **ssize\_t write(int fd, void \*buffer, size\_t nbytes);** Аргументът fd задава файлов дескриптор на отворен файл, в който ще се пише. Аргументът буf задава адрес в паметта, откъдето ще се взимат данните за писане. Аргументът count задава брой байтове, които ще се записват във файла. При успех връща броя на действително записаните байтове. При неуспех write връща -1. След изпълнението на write текущата позиция се увеличава с броя на действително записаните байтове. V) **Позициониране във файл: off\_t lseek(int fd, off\_t offset, int flag);** Аргументът fd е файлов дескриптор на отворен файл. Аргументът offset задава отместването, с което ще се промени текущата позиция в брой байтове. Аргументът flag определя откъде ще се отчита отместването. Flag може да приема следните стойности (в скоби са посочени еквивалентни символни константи): 0 (SEEK\_SET) – отместването се отчита относно началото на файла, т.е. новата стойност на текущата позиция е offset; 1 (SEEK\_CUR) – отместването се отчита относно текущата позиция във файла, т.е. към текущата позиция се прибавя offset; 2 (SEEK\_END) – отместването се отчита относно края на файла, т.е. новата стойност на текущата позиция е размерът на файла, прибавен към offset. VI) **Информация за файл: int stat(const char \* filename, struct stat\* sbuf); int fstat(int fd, struct stat\* sbuf);** Аргументът filename задава име на файл. Информацията за файла се връща в аргумента sbuf. Структурата stat е описана в заглавния файл stat.h и нейните полета включват всички атрибути на файла от индексния описател, номер на индексния описател и др. При успех stat връща 0, при неуспех връща -1.



**9.Физическа организация на файловете системи.Системни структури-информация за свободната памет и за паметта разпределена за файлове. Основни цели: ефективност**(бърз достъп до файловете, ефентивно ползване на дисковата памет), **надежност**(устойчивост при конкурентен достъп, устойчивост при сривове), **разширяемост**(новости в хардуера и софтуера да не затрудняват Ф.С.).

**Стратегии за управление на дисковото пространство:-Проблеми:** **1.**кога се разпределя дискова памет-еднократно - при създаване на файла (статично);заделя се нова порция при нарастване на файла (динамично). **2.**колко непрекъснати области ще заема един файл-една непрекъсната област-такава че да побере всичките му данни;много (несъседни) области. **3.**каква да е единичната за разпределение на дисковата памет-дали да е с променлив размер или не;колко да е голяма.

**-Стратегии:** **1.** статично и непрекъснато.Проблеми-при нарастване на файл; фрагментация на свободната дискова памет. **2.**динамично и поблоково с фиксиран размер на блока.Проблеми-важен е въпроса за големината на блоковете които се разпределят. Ако са малки се пести дискова памет, но се получава фрагментация. Ако са големи пък се губи дискова памет.(ползва се в наши дни).

**Системни структури съдържащи инфо. за разпределението на дисковата памет:** При работата на ОС са необходими структури, които съдържат информация за текущото разпределение на дисковата памет. Тези структури играят и основна роля в алгоритмите които ОС използва за заделяне и освобождаване на дискова памет. Основните такива структури са: за свободните блокове;за блоковете разпределени за даден файл;за общи параметри на файловата система.

**Информация за свободните блокове:** -Свързан списък на свободните блокове-Тук всеки свободен блок съдържа адреса на следващ свободен блок. Адреса на първия свободен блок се съдържа в друга структура. Когато се задела памет се задела първия свободен блок, а адреса на следващия се записва в другата структура като адрес на първи свободен блок. Реализирана по този начин структурата е ненадеждна (защото при срив се къса цялата верига) и неефективна.(XINU); -**Свързан списък от блокове с номера на свободни блокове**-Имаме няколко блока, които съдържат като информация в себе си номерата на свободни блокове. Всеки от тези няколко блокове (без последния), сочи към следващ блок с номера на свободни блокове. Този реализация е по-надеждна и по - ефективна от предишната. (UNIX System 5); -**Карта/таблица (bitmap)** - Използва се масив с брой елементи = броя блокове във файловата система. Всеки елемент на масива отговаря за един блок. Съседните елементи на масива отговарят за физически съседни блокове. Когато даден елемент (най често бит) е вдигнат това означава, че съответния му блок е свободен. Основно предимство на тази реализация е, че може да се отчита физическото съседство на блоковете.

**Информация за разпределените блокове:**Структура, която съдържа информация кои блокове принадлежат на даден файл и в каква последователност.Тук също имае варианти на реализация: -**Свързан списък на блоковете на файла**-Всеки блок принадлежащ на даден файл съдържа и адреса на следващия блок принадлежащ на файла. Последния такъв блок съдържа някакъв маркер за край на файла (EOF). Друга структура пък (най-често каталог) съдържа информация за първия блок принадлежащ на файла. Поради това че веригата е пръсната, тази реализация е ненадеждна при сривове. Освен това е и неефективна, поради факта, че за да се прочете произволен блок от даден файл трябва да се обходи голяма част от веригата. Основния проблем е, че се смесват данни и адреси (във всеки блок се съдържа освен данни на самия файл и служебни данни - адреса на слеващия блок); -**Карта таблица**- Масив с брой елементи = броя елементи на блоковете в диска. Всеки елемент отговаря за съответния си блок. Всеки елемент съдържа номера на следващия блок за даден файл (= индекс в масива). Отделна структура пък съдържа имената на файловете и първия свободен блок за даден файл. Реализацията се използва в MS DOS и структурата се нарича FAT (File allocation table); -**Индекс**- Всеки файл си има собствен индекс-списък на блоковете разпределени за този файл. Възможнож са няколко варианта за конкретна реализация: -Свързан списък с индексни блокове(XINU) ; -Дърво (UNIX, LINUX, MINIX) -В+ дърво(OS/2-HPFS).

**Структури за общите параметри на файловата система:**Служат за съхраняване на важна информация за файловата система на глобално ниво-размер на файловата система; размер на блок; размери и адреси на области, съдържащи системни структури; общ брой свободни блокове;

**Реализация на каталог:**Каталогът осигурява връзка между името на файла и данните и атрибутите му.Каталогът реализира логическата структура на ФС. Всеки каталог съдържа записи за:файловете и каталозите, на които той е родителски каталог;-два std. записа: "-;"-описва самия каталог; "-;"-описва родителския му каталог.Наредба на записите в каталога-хронологична и сортирани по името на файла.

**11. Физическа организация на файловата система в LINUX:** Цялото дисково пространство се разделя на еднакви блокове, с размер S. Стойността на S се задава при създаването на файловата система и може да приема стойности от 1024, 2048 или 4096 байта. Блокът е най-малката единица, с която файлт може да работи. Това означава, че даден блок, дори и непълен, в даден момент може да се използва само от един файл. Първият (най-велят) блок от дисковото пространство се нарича **boot блок** и служи за първоначално зареждане на системата. Останалите блокове се обединяват в еднакви по размер **групи блокове** - всяка група съдържа част от ФС и копие на глобалните системни структури(Суперблок, Описатели на групи, Битова карта на блоковете, Битова карта на I-nodes, Таблица на I-nodes, Област с данни). **Битова карта на блоковете:** Описват свободните ресурси - блокове и индексни описатели в групата. Значение 0 означава свободен, а 1 използван блок или i-node. Размерът на групите се избира така, че всяка битова карта да заема един блок. Така ако размерът на блока е 1024 байта и битовата карта на блоковете заема един блок то в нея има 8192 бита. Следователно броят на блоковете в групата = 8192 и имаме 8192 \* 1KB = 8192 KB = 8 MB. Т.e във всяка група можем да поберем 8MB информация. **Битова карта на I-nodes:** Битовата карта на I-nodes също заема един блок. На всеки бит съответства най-много един I-node от таблицата. Тук е възможно да съществуват битове, които нямат съответствие с I-node, защото броят на I-nodes е <= от броя на блоковете. Последните 1024 - K бита остават неизползвани. Ако даден бит от картата е 0, то съответния му I-node е зает , ако е 1, то той е свободен. Тъй като всеки файл заема най-малко един блок, то обикновено броят на необходимите I-nodes в системата е много по-малък от броя на блоковете. Затова е решено при създаването на файловата система ръчно да се задава броят на I-nodes в системата. По подразбиране за всеки 4096 байта се задела един I-node. Интересен е факта, че дори да има свободно място на диска то ако е изчерпан броят на I-nodes не можем да създадем нов файл. **Таблица на I-nodes:** Нека при формиране на системата е зададен брой на I-nodes = M. И нека диска е разделен на N групи. Тогава таблицата на I-nodes във всяка група съдържа M/N последователно разположени I-nodes. **Индексни описатели:** Всеки I-node може да сочи към блокове от произволна група. Индексната област във всяка група съдържа част от индексните описатели на файловата система.Те са направени така, че I-node по възможност първо да използва блокове от неговата си група. Всеки I-node има размер 128 байта и съдържа следните елементи: -адресните полета са 12+1+1+1 по 4 байта; -освен атрибутите на файл от Unix са добавени: -размер на файла в брой блокове по 512 байта: -още едно поле за дата и време; -флагове: immutable, append only, synchronous write, secure deletion, undelete. ; -др. **Описатели на групи:** с всяка група се свързва структура наречена описател на групата. В тази структура основи са следните полета: - указател към блока с битовата карта на блоковете в групата; -указател към блока с битовата карта на I-nodes в групата; -указател към първия блок на таблицата с I-nodes в групата; -променлива показваща броя на свободните блокове в групата; -променлива показваща броя на свободните I-nodes в групата; -променлива показваща броя на свободните директории в групата. Описателите за всички групи се записват по ред на номерата един след друг и образуват структура наречена описатели на групи. Тя заема един блок и нейно копие се намира във всяка група. **Суперблок:** Суперблока е структура, която съдържа глобална информация за системата. Някой от по-важните полета са: -брой на I-nodes в системата; -брой на блоковете системата; -брой на свободните I-nodes системата; -брой на свободните блокове системата; -големина на блока системата. Някой от полетата са променливи, други като големината на блока са константи, които се фиксират при създаването на файловата система. Суперблока заема един блок и има копие във всяка група. **Директории:** Директоритите представляват файлове, които са с определен формат. Всеки запис се отнася за файл от директорията. Записите са с променлива дължина но са кратни на 4. Всеки запис включва следните неща: -номер на I-node за съответния файл - 4 байта; -дължина на записа - 2 байта; -дължина на името на файла 2 байта; -име на файла до 255(байта) символа (масив от симв., като се допълва до кратност на 4). Директорията съдържа 2 стандартни записа: . и .. които са синоними на самата директория и на родителската директория. При изтриване на файл от каталог i-node = 0,len(на предния запис) += тоя който тримем. **Бързи символни връзки:** Ако във файл от тип символна връзка броят на символите указващи пътя до файл заема по-малко от 64 байта, то се задела само I-node (не се задела блок). **Новото в LINUX в сравнение с UNIX:** -използвани са битови карти при управление на ресурсите - блокове и индексни описатели(отчита се съседство); - разделяне на дисковото пространство на групи блокове – по висока степен на локалност на файловете, системните стуктури са близи до обектите, които описват; -няколко копия на системните структури, съдържа-щи информация критична за ФС; -бързи символни връзки; -по-надеждна и по-ефективна ФС.

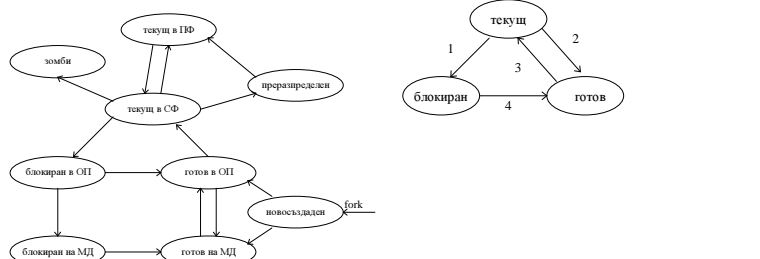
**10.Физическа организация на файловата система в UNIX:** Дисковото пространство се разглежда като последователност от еднакви блокове. Размерът им се определя при създаването на файловата система и може да заема стойности от 512KB, 1024KB, 2048KB. Диска е разделен на 4 области:Boot block – програма за зареждане на ОС, super block – общи параметри на ФС, индексна област и област за данни – блокове с данни на файлове и каталози, косвени блокове, блокове от списъка на свободните блокове и свободни блокове. Boot блока и супер блока заемат по един блок и се намират съответно на първа и втора позиция от началото на файловата система. След тях е индексната област, а след нея областта с данни. Индексната област се състои от последователност от индексни описатели, които имат еднаква структура и всеки от тях заема по 64B. Размерът на индексната област зависи от това колко броя индексни описатели съдържа тя. Броят на индексните описатели се определя при създаване на файловата система и не може да бъде променян по-късно. Ето защо иранциата между индексната област и областта данни се фиксира при създаването на файловата система и остава непроменена през цялото съществуване на файловата система. Предназначението на индексните описатели е да съдържат информация за файловете. Всеки индексен описател може да описва най-много един файл. Поради тази причина максималния брой файлове, които може да има във файловата система = броя на индексните описатели. Тази особеност може да доведе до парадокса на диска да има още свободно пространство, т.e. областта за данни да не е пълна, но да не може да се създават повече файлове, защото няма свободни индексни описатели.

**В индексния описател се пази следната информация за даден файл: -mode-** тип на файла и код на защита; **-nlin-** брой на твърдите връзки; **-uid-** собственик; **-gid-** група; **-size-** размер; **-atime-** дата и време на последен достъп; **-mtime-** дата и време на последно изменение; **-ctime-** дата и време на изменение на i-node; -addr- 13 адреса на блока; **Код на защита:** в младшите 12 бита на **mode.** Той определя правата на различните потребители за достъп до файла. **Класове потребители:** -администратор(root); -собственик- потребител, чийто идентификатор е в полето uid на i-node; -група- потребители, които не са собственик на файла, но принадлежат на групата в полето gid на i-node; -други– потребители, които не са в първите класове (nwx – всеки). **Типове достъп:** -r- да четем от файла или каталога; -w- да пишем във файл, за каталог – да създаваме или унищожаваме файлове в него; -x- да извикваме файл за изпълнение, за каталог – търсене на файлове в него и позициониране в каталога; -sticky bit- за каталог е за допълнителна защита при унищожаване на файлове в него. **Супер блок:** размер на файловата система - максимален номер на блок; размер на индексната област; общ брой свободни блокове; общ брой свободни индексни указатели; списък от номера на свободни блокове; списък от номера на свободни индексни блокове; индекса на следващия свободен блок в масива с номера на свободни индексни блокове; индекса на следващия свободен индексен описател в масива с номера на свободни индексни описатели. **Алгоритъм за заделянето на нов свободен блок:** Ако в списъка от номера на свободни блокове в суперблока има повече от един номер, то се задела последния в списъка ( най-десния ), и указателя минава една позиция наляво. Ако е последният то номера се тълкува като номер на блок съдържащ номера на свободни блокове. В този случай номерата от този блок се копират в списъка от суперблока, а самият блок се задела за необходимата операция. **Алгоритъм за освобождаване на блок:** Ако списъка в суперблока е пълен то номерата от него се преместват в новоосвободения блок, като при това списъкът в суперблока остава празен, на първо място в него се записва номера на освободения преди малко и вече запълнен с номера на свободни блокове блок. **Каталози:** Каталогът има същата структура като обикновено файл със следните разлики: -в индексния му описател като тип файл е записано - каталог; -записите във файл от тип каталог са еднакви, с фиксирана дължина и всеки от тях съдържа две полета: номер на индексен описател и име на файл. **Твърди връзки (hard link):** Твърдите връзки са няколко записа в каталози, които съдържат един и същи номер на i-node.(ln) **Символни връзки (symbolic link):** Символна връзка е тип файл, който сочи към друг файл (съдържанието му е името на другия файл).(ln -s).

**12. Физическа организация на файловата система в MSDOS:** Дисковото пространство е разделено на 5 области: - **boot сектор**; - **FAT**; - **FAT**; -**Коренен каталог**; -**Данни**. **Boot сектора:** съдържа програмата за зареждане на ОС + информация за някой параметри на файловата система (рамера и големината на единичната за разпределение на дисковата памет - клъстера, размер на корения каталог). **FAT (File allocation table):** Съдържа информация за свободните дискови блокове (клъстери) и за блоковете разпределени за всеки файл. Всеки елемент на FAT съответства на един клъстер. По този начин всеки елемент отговаря за състоянието на съответния му клъстер. Ако даден клъстер е свободен, то в елемента на FAT е записан код за празен клъстер - 0. Ако в елемента на FAT е записана стойност K, където K е от 2 до N (където N е броя на елементите във FAT), то значи съответния клъстер е разпределен за някой файл и следващия клъстер от този файл е с номер K. Ако в елемент на FAT е записан код FFF, то съответния му клъстер е последния клъстер принадлежащ на даден файл. Първите 2 елемента на FAT са заделени за служебни нужди и затова номерата на клъстерите започват да се броят от 2 и съответно в елементите във FAT не можем да записваме стойности по-малки от 2. Поради важността и на диска се съдържаът 2 еднакви копия на тази структура. Така при евентуален срив имаме резервно копие на FAT. **Област с данни:** Тук се намират клъстерите които се заделят за потребителските програми и данни. **Каталози:** В DOS каталозите са тип файлове. Изключение прави корения каталог, който е изнесен в отделна структура. Всички каталози съдържат записи с фиксиран размер 32 байта – всеки описващ един файл или подкаталог: - име на файла - 8 байта; - разширение на името - 3 байта; - флагове - 1 байт (Всеки бит отговаря за различен флаг - read only, hidden, system, archive, флаг за това дали файла е каталог, дали е етикет на диска и др.); - час на създаване или последно изменение - 2 байта; - дата на създаване или последно изменение - 2 байта; - номер на първи клъстер на файла - 2 байта; - размер на файла в брой байтове - 4 байта. Всеки запис отговаря за 1 файл и съдържа описаните полета. **Коренен каталог:** Кореният каталог е с размер, фиксиран при форматирането, и не съдържа стандартните записи, но съдържа запис за етикет на тома. Различава се по това, че не е файл, а е изнесен в отделна структура. По тази причина имаме ограничен брой на записите в корения каталог и съответно ограничение за броя на файловете и директориите които се намират в него. **Развитие на файловата система FAT:** -FAT12 (4096 клъстера), FAT16 (65536 клъстера) и FAT32 (2<sup>28</sup> клъстера); - Допълнителни атрибути в резервираното място - дата и време на създаване, дата на последен достъп, още 2 байта за номер на първи клъстер; -Дълги имена на файлове (до 255 символа) в Unicode – постига се чрез няколко последователни записа за файл в каталог, един основен с име във формат "xxxxxx-1.yyy" и атрибути, няколко допълнителни с дългото име. Структура на запис с част от дълго име:



От 14-та тема:



**13. Физическа организация на файловата система NTFS:** Единица за разпределение и адресиране на дисковото пространство в NTFS е кластер - 1KB, 2KB или 4KB. Адресът на кластер относно началото на тома се нарича Logical Cluster Number, а адресът в рамките на определен файл се нарича Virtual Cluster Number. NTFS е на принципа всичко на диска е файл: и данните и метаданните (системните структури) се съхраняват в тома като файлове, т.е. на всеки том има обикновени файлове, каталози и системни файлове. Това позволява динамично разпределение на дискова памет при нарастване на метаданните, без да са необходими фиксирани области въхру диска за тях. Главният системен файл е MFT (Master File Table). Разпределение на дисковото пространство в една файлова система: -boot file; -MFT; -MFT зона; -свободно пространство; -други системни файлове; -свободно пространство; -копие на boot file. **Файлтът MFT** е индекс на всички файлове на тома. Съдържа записи по 1KB и всеки файл на тома е описан чрез един запис, включително и MFT. Освен MFT има и други файлове с метаданни, които имена започват със символа \$ и са описани в първите записи на MFT. Системните файлове са към средата на тома. В началото на тома е boot файла. Скрипо в края на тома е копие на boot файла. За да се намали фрагментирането на MFT файла, се поддържа буфер от свободно дисково пространство - MFT зона. Размерът на MFT зоната се намалява наполовина винаги когато останалата част от тома се запълни. **MFT файл:** Всеки файл на тома е описан в поне един запис на MFT файла. Индексът (номерът) на началния MFT запис за всеки файл се използва като идентификатор на файла във файловата система. Първите записи са резервирани за системните файлове:(име на файл/ индекс/ описание): Mft0/MFT файл; \$MftMirr/Копие на първите записи от MFT файла; \$LogFile2/Журнал при поддръжане на транзакции; \$Volume /3/Описание на тома; \$AttrDef4/Дефиниции на атрибутите; \$VКоренен каталог на тома; \$BitMap6/Битова карта на тома; \$Boot7/Boot сектори на тома; \$BadClus8/Списък на лошите кластери.

**Атрибути на файла:** Всеки файл се съхранява като последователност от двоични „атрибути/значение“. Един от атрибутите са данните на файла (unpamed data attribute). Други атрибути са име на файл, стандартна информация и други. Всеки атрибут се съхранява като отделен поток от байтове. Обикновен файл може да има и други атрибути данни, наричани named data attribute. Това променя представата ни за файл, като една последователност от байтове, т.е. файлът може да има няколко независими потока данни. Типовете атрибути(общо-14)(име на тип атрибут/описание): \$FILENAME/ Името на файла, един файл може да има няколко имена, при твърди връзки или ако се генерира кратко име в MSDOS стил; \$STANDARD INFORMATION/атрибути на файл, като флагове, време и дата на създаване и последно изменение, брой твърди връзки; \$DATA/данните на обикновен файл, всеки файл има един неименован атрибут данни и може да има допълнителни именувани атрибути данни; \$INDEXROOT, \$INDEXALLOCATION, \$BITMAP/при атрибути използвани при реализацията на каталозите; \$ATTRIBUTELIST/този атрибут се използва, когато за файл има повече от един запис в MFT, съдържа списък от атрибутите на файла и индексите на записите; \$VOLUME NAME/тези атрибути се използват само в системния файл \$Volume; \$VOLUMEINFORMATION/те съхраняват информация за тома. Атрибутите биват **резидентни** или **нерезидентни**. **Резидентен** е атрибут, който се съхранява изцяло в MFT записа. Някои атрибути са винаги резидентни, напр., \$FILE NAME, \$STANDARD INFORMATION, \$INDEX ROOT. Ако значението на атрибут, като данните на голям файл, не може да се съхрани в MFT записа, то за него се разпределят кластери извън MFT записа. Такива атрибути се наричат **нерезидентни**. Файловата система решава как да съхранява един атрибут. Нерезидентни могат да бъдат само атрибути, чиито значения могат да нарастват, например \$DATA. **MFT запис:** Всеки MFT запис съдържа заглавие на записа (record header) и атрибути на файла. Всеки атрибут се съхранява като заглавие на атрибута (attribute header) и данни (значение). Заглавието на атрибута съдържа код на типа, име, флагове на атрибута и информация за разположението на данните му. Един атрибут е резидентен ако данните му се поместват в един запис заедно със заглавието на всички атрибути на файла. Ако един атрибут не е резидентен, заглавието му (което е винаги резидентно) съдържа информация за кластерите, разпределени за данните му. Адресната информация се съхранява в последователност от описания на екстенсти (run/extent entry). Всеки **екстенст** е непрекъсната последователност от кластери, разпределени за данните на съответния атрибут и се описва от адрес на началния кластер и дължина (VCN, LCN, брой кластери). Ако един файл не може да се опише в един MFT запис, то се разпределят допълнителни записи. В основния (първи) запис има атрибут \$ATTRIBUTE LIST, съдържащ указатели към допълнителните записи (код на типа и номер на MFT записа). **Каталози:** Каталогът съдържа записи с променлива дължина, всеки от които съответства на файл или подкаталог, в съответния каталог. Всеки запис съдържа името на файла и индекса на основния MFT запис на файла, както и копие на стандартната информация на файла. Това дублиране на стандартната информация изисква две операции писане при изменението ѝ, но ускорява извездането на справки за съдържанието на каталога. Записите в каталога са сортирани по името на файла и се съхраняват в структура В дърво. Ако каталогът е малък, всичките му записи се съхраняват в атрибута \$INDEXROOT, който е резидентен, т.е. целият каталог се намира в MFT записа си. Когато каталогът стане голям, за него се разпределят екстенсти с размер 4KB, наречени индексни буфери. Атрибутът \$INDEX ROOT и тези екстенсти са организирани в В дърво. В този случай каталогът има и атрибут \$INDEXALLOCATION, който съхранява адресна информация за разположението на екстенстите-индексни буфери. Атрибутът \$BITMAP е битова карта за използването на кластерите в индексните буфери.

работоспособен: - **зомби (zombie)** - процесът е изпълнил системния примитив exit и вече не съществува, съхранява се информация за него, която може да бъде предадена на процеса-баща.(диаграмите след 12-та тема)

**15. Контекст на процес:** Всичко, което реализира един процес, се нарича контекст на процес.

**Части на контекста:** 1) Потребителска част – образ на процеса, отговаря на тази част от процеса, която е в потребителска фаза 2) Машина (регистрова) част – съдържанието на машинните регистри – броячът на командите, статус регистърът, общите регистри и др. 3) Системна част – структури в пространството на ядрото, описващи процеса: таблица на процесите, потребителска област (User area или U area), стек на ядрото и динамична част на контекста.

**Образ на процеса:** В UNIX системите образът се разделя на логически единици, наречени **региони**: 1) Код (text) – съдържа машинните команди, изпълнявани от процеса в потр. фаза. 2) Данни (data) – съдържа глобалните данни, с които процесът работи в потребителска фаза - инициализирани и неинициализирани. 3) Стек (stack) – чрез него се реализира обръщението към потребителски функции. Стек от слове - по един слой за всяка извикана функция, която още не е изпълнила getutm. Съдържа данни, локални за функцията.

**Таблица на регионите:** Логическите единици на които се дели образа на процеса се наричат региони (regions).Регионът е обект за защита; Регионът е обект за съвместно използване от процесите - един регион може да е общ за няколко процеса. **Таблица на регионите:** Съдържа информация за всички активни региони в системата.1) тип на региона - код, данни, стек и др. 2) размер на региона в байтове 3) адрес в паметта 4) флагове за състояние - заключен, зарежда се в паметта и др. 5)брой процеси, използващи региона. **Таблица region:** (Per process region table) Всеки процес има своя таблица region 1) тип на достъп на процеса достъп до региона – read-only (само за четене), read-write (четене и писане), read-execute (четене и изпълнение); например регионът за код обикновено е read-execute, регионите за данни и стек са read-write, регионът обща памет може да е read-write или read-only в различните процеси 2) виртуален адрес на региона в рамките на образа на процеса 3) указател към запис от таблицата на регионите.

**Таблица на процесите:** Таблицата на процесите е глобална структура, която описва всички процеси в системата. За всеки процес в таблицата има запис, който съдържа най-важните характеристики на процеса. Такава структура се поддържа от всяка операционна система. В този смисъл получаваме още едно определение за процес – **процес** е обект, за който има запис в таблицата на процесите. 1) идентификатор на процеса (**pid**) 2) идентификатор на процеса-баща (**ppid** - parent pid) 3) идентификатор на група процеси (**pgid** - pid на процеса-лидер на групата) 4) идентификатор на сесия (**sid** - pid на процеса-лидер на сесията) 5) състояние на процеса 6) събитие, настъпването на което процесът чак в състояние блокиран 7) полета, осигуряващи достъп до образа на процеса и U area 8) полета, определящи приоритета на процеса при планиране 9) полета, съхраняващи времена - използването от процеса време на ЦП в системна и потребителска фаза и др 10) код на завършване на процеса. Следата от процеса, когато той е в състояние **зомби** е единствено запис в таблицата на процесите. От него бащата на процеса може да получи информация за сина си – код на завършване и др.

**Потребителска област(U area):** Съдържа данни за процеса, които са необходими и достъпни за ядрото само когато той е в състояние текущ. 1) указател към записът на таблицата на процесите 2) файлови дескриптори 3) текущ каталог на процеса 4) управляващ терминал на процеса 5) маска при създаване на файлове (заредена с **umask**) 6) реален потребителски идентификатор (**ruid**) 7) ефективен потребителски идентификатор (**euid**) 8) реален идентификатор на потребителска група (**rgid**) 9) ефективен идентификатор на потребителска група (**egid**) 10) параметри на текущия систем примитив и върнати от него стойности.

**Стека на ядрото и динамична част:** Когато един процес работи в системна фаза е необходим стек на ядрото, чрез който да се реализират обръщения към функции в ядрото. Освен това, когато процесът преминава от текуща в системна фаза трябва да се запомни неговата машинна част – състоянието на регистрите. Така за всеки процес контекстът на процеса включва стек на ядрото и област за съхранение на регистрите. Възможно е докато процесът работи в системна фаза, той да премине в нова системна фаза, ако настъпи прекъсване с по-висок приоритет – преход от състояние текущ в системна фаза в същото състояние, който не е изобразен по-горе в диаграмата на преходите. За да се реализира това контекстът на процеса включва така наречената динамична част, която най-общо представлява стек от слове. Във всеки слой се съхранява състоянието на регистрите преди последното прекъсване и стек на ядрото, който се използва при обработката на последното прекъсване. Процес, който работи в системна фаза винаги се изпълнява в контекста на слоя, който е във връзка на стека. Слойт, който е в дъното на стека съдържа съхранените регистри от потребителската фаза на процеса. Когато процесът работи в потребителска фаза, динамичната част на неговия контекст е празна. Останалата част на контекста още се нарича статична част.

**Превключване на контекста:** **Превключване на контекст се прави когато:** 1) Текущият процес се блокира. 2) Текущият процес е изпълнил системния примитив exit и минава в състояние зомби. 3) Текущият процес е завършил системната си фаза и ще трябва да се върне в потребителска фаза, но планировчикът решава да го свали. **Стъпките при превключване на контекста:** 1) Решава дали да прави превключване на контекста. 2) Съхранява контекста на "стария" процес (push в динамичната част на контекста му). 3) Избира "най-подходящия" процес за текущ, използвайки алгоритъма за планиране. 4) Възстановява контекста на "новия" процес (форс от динамичната част на контекста му).

**14. Процеси. Модел на процесите. Състояние. Диаграма на преходите:** В операционните системи понятието процес е централно. Съвременните компютри са способни да изпълняват няколко операции едновременно - **мултипрограмиране**. В ОП са заредени няколко програми и макар, че ЦП във всеки един момент изпълнява само една команда, той може да се превключва от изпълнение на една програма към друга много бързо. Мултипрограмирането се въвежда с цел по-ефективното използване на ресурсите на компютъра. Създава впечатление за едновременно изпълнение на няколко програми. Истината е, че ЦП изпълнява няколко последователни дейности, като за тази цел трябва да се съхранява информация за тези дейности, за да е възможно да се възстановява изпълнението на прекъсната дейност. **МОДЕЛ НА ПРОЦЕСИТЕ:** В такъв един модел се въвежда понятието за обозначаване на дейността по изпълнение на програмата(и) за определен потребител. В различни операционни системи са използвани понятията **задание (job), задача (task), процес (process)**. Най-краткото определение за **процес** е **програма в хода на нейното изпълнение**. За разлика от програмата, която е нещо статично - файл записан на диска и съдържащ изпълним код, процесът е дейност. В понятието процес освен програмата се включват и текущите стойности на регистъра PC, на другите регистри, на програмните променливи, състоянието на отворените файлове... В мултипроцесна ОС всякия софтуер работещ на компютъра е организиран в процеси.ЦП винаги изпълнява. Когато операционната система поддържа едновременото съществуване на няколко процеса се казва, че е многопроцесна - UNIX, MINIX и LINUX. **ИЕРАРХИЯ НА ПРОЦЕСИТЕ:** Операционна система, която реализира абстракцията процес, трябва да предоставя възможност за създаване на процеси и за унищожаване на процеси, а също така и начин за идентифициране на процес.В UNIX, LINUX и MINIX процес се създава със системния примитив fork. В тези системи най-точното определение за **процес** е **обект, който се създава от fork**. Когато един процес изпълни fork ядрото създава нов процес, който е почти точно негово копие. Първият процес се нарича процес-баща, а новият е процес-син. След fork процесът-баща продължава изпълнението си паралелно с новия процес-син. Един процес може едновременно да има няколко процеса-синове. Процесът-син също може да изпълни fork и да създаде свой процес-син. Всички едновременно съществуващи процеси са свързани в йерархия. Всеки процес се идентифицира чрез уникален номер, наричан **pid (process identifier)**, който освен, че е уникален за процеса и не се променя. UNIX/LINUX - преди първият процес няма друг. Програмата за начало зареждане, която е в boot блока, зарежда ядрото в паметта и предава управлението на модул start, който инициализира структурите в ядрото (системните таблици) и ръчно създава процес с pid 0. От тук нататък ядрото продължава да работи като процес с pid 0 и създава нормално с fork процес с pid 1, в който пуска за изпълнение програмата init. След това процес 0 създава няколко други процеси, които се наричат системни процеси (kernel processes), и самия той става системен процес.Процес init е нормално създаден процес и затова ядрото го счита за корен на дървото на процесите. Той се грижи за инициализация на процесите. Когато процес init зароботи, чете файл /etc/inittab и създава процеси. Например, за всеки терминал в системата init създава процес, в който пуска за изпълнение специална програма - getty в повечето версии на UNIX (mingetty в LINUX). getty чака вход от съответния терминал и когато той постъпи приема, че потребител иска вход в системата. Тогава getty в процеса се сменя с login, която извършва идентификация на потребителя и при успех се сменя с програмата shell за съответния потребител (или поражда нов процес за програмата shell). В живота на един процес могат последователно да се сменяят различни програми, а също така няколко едновременно съществуващи процеса могат да изпълняват една и съща програма, но те са различни обекти за ядрото.След като процесът init приключи с инициализацията на процесите, той изпълнява безкраен цикъл, в който чака завършване на свой процес-син и изпълнява довършителни дейности. **СЪСТОЯНИЕ НА ПРОЦЕС:** В многопроцесните операционни системи едновременно съществуват много процеси, а ЦП е един и във всеки един момент може да изпълнява само един от тези процеси. За този процес казваме, че се намира в състояние текущ (running). Останалите процеси са в някакво друго състояние. Най-опростеният модел на процесите включва три вида състояния: - **текущ (running)** - ЦП изпълнява команди на процеса; - **готов (ready)** - процесът може да продължи изпълнението си, ако му се предостави ЦП; - **блокиран (blocked)** - процесът чаква настъпването на някакво събитие, много често завършването на I/O операция. Преход 1 се случва, когато процесът открие, че трябва да чака някакво събитие. Преход 4 се извършва когато настъпи събитието, чакано от процеса. Преходи 2 и 3 се управляват от scheduler-а. Когато ЦП се освободи, планировчикът избира един от готовите процеси за текущ – преход 3. Преход 2 означава насилствено отнемане на ЦП от процеса - ако ОС реализира този преход - **планиране с преразпределение** (preemptive scheduling), иначе **е планиране без преразпределение. Моделът на процесите в UNIX System V включва девет състояния** : - **текущ в потребителска фаза (user running)** - ЦП изпълнява команди от user програма свързана с процеса; - **текущ в системна фаза (kernel running)** - ЦП изпълнява команди от ядрото, т.е. от името на процеса работят модули на ядрото; - **готов в паметта (ready in memory)**; - **блокиран в паметта (blocked in memory)** - процесът чаква настъпването на някакво събитие и се намира в паметта; - **готов на диска (ready, swapped)** - процесът е готов за изпълнение, но планировчикът трябва да го зареди в паметта преди да може да бъде избран за текущ; - **блокиран на диска (blocked, swapped)** - процесът е блокиран и планировчикът го е изхвърлил от паметта в специална област на диска - свързан област, която представлява разширение на паметта, за да освободи място за други процеси; - **преразпределен (preempted)** - процесът е бил на път да се върне в състояние 1, след състояние 2, но планировчикът му е отнел ЦП насилствено, за да го предостави на друг процес; - **новосъздаден (created)** - това е началното състояние, в което процес влиза в системата. Той е почти създаден, но още не е напълно

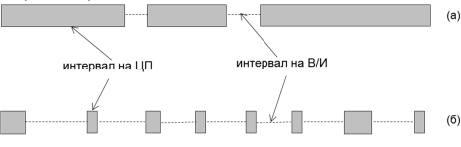
**16. Системни примитиви за процеси - fork, exit, wait, exec, getpid, getppid.** Създаване на процес: **pid\_t fork(void);** В процеса-баща функцията връща pid на процеса-син или -1 при грешка; В процеса-син връща 0. **Алгоритъм на fork:** 1) Определя уникален pid за новия процес и създава запис в таблицата на процесите (група и сесия от процеса-баща, състояние "новосъздаден"). 2) Създава U area на новия процес - копие на тази на бащата (файлови дескриптори, текущ каталог, управляващ терминал, маска, потребителски идентификатори - euid, uid, egid и rgid). 3) Създава образ на новия процес - копие на образа на процеса-баща. 4) Създава динамичната част от контекста на новия процес: Слой 1 - копие на слой 1 от контекста на бащата и Слой 2 - съхранените регистри от слой 1, като регистър PC е изменен така, че синът да започне изпълнението си в fork от стълка 7. 5) Изменя състоянието на процеса-син в "готов в паметта". 6) В процеса-баща връща pid на новосъздадения процес-син. 7) В процеса-син, връща 0. **Общото между процесите баща и син:** изпълняват една и съща програма; Процесът-син наследява от бащата файловите дескриптори; имат един и същ текущ каталог, управляващ терминал и група процеси и се; имат еднакви права. **Завършване на процес: void exit(int status); Алгоритъм на exit:** 1) Изпълнява close за всички отворени файлове и освобождава текущия каталог на процеса. 2) Освобождава паметта, заемана от образа на процеса и от U area. 3) Сменя състоянието на проц в "зомби". Записва кода на завършване в таблицата на процесите. 4) Урежда изключването на проц от йерархията на процесите: Изпраща сигнал "death of child" на процеса-баща на завършващия процес; Ако проц има синове, то техен баща става процесът init и ако някой от тези синове е зомби изпраща сигнал "death of child" на процеса init. **Изчакване завършването на процеса-син: pid\_t wait(int \*status);** При успех функцията връща pid на завършилия син, а чрез аргумента **status** кода му на завършване. **Алгоритъм на wait:** 1) Ако процесът няма синове, връща -1. 2) Ако процесът има син в състояние зомби (синът вече е изпълнил exit) освобождава записа му от таблицата на процесите, като взема кода на завършване и неговия pid и ги връща. 3) Ако процесът има синове, но никой от тях не е зомби, той се блокира, като чака сигнал "death of child". Когато получи такъв сигнал процесът продължава работ в точка 2; **pid\_t waitpid(pid\_t pid, int \*status int options);** В зависимост от **pid** чак: **pid == -1** първия син, който завърши; **pid > 0** син с идентификатор pid; **pid == 0** първия син от същата група процеси; **pid < -1** първия син от група процеси с идентификатор |pid| **Aко options** е WNOHANG, процесът не се блокира ако синът не е завършил, а функцията връща 0. **Изпълнение на програма: int exec(const char \*name, const char \*arg[] , const char \*arg1[]... 0); int execp(const char \*name, const char \*arg[] , const char \*arg1[] , const char \*arg1[]... 0); int execl(const char \*name, char \*argv[]); int execevp(const char \*name, char \*argv[]);** **name** - името на файл, съдържащ програма в изпълним код; **arg0, arg1, ...** или **argv[]** – указатели към аргументите, които ще бъдат предадени на функцията main на новата програма. Връщат –1 при грешка, нищо при успех. **Алгоритъм на exec:** 1) Намира файла, чието име е в аргумента **name** и проверява дали процесът има право за изпълнение на този файл. 2) Проверява дали файлът съдържа изпълним код. 3) освобождава паметта, заемана от стария образ на процеса. 4) Създава нов образ на процеса, използвайки изпълнимия код във файла с име **name** и копира аргументе на ехес в новия потребителски стек. 5) Изменя значенията на някои регистри в областта за съхранение регистри в слой 1 от динамичната част на контекста (PC, указател на стек). Така, когато процесът се върне в потребителска фаза ще заработи от началото на функцията main на новия образ. 6) Ако програмата е set-UID прави съответните промени на потребителските идентификатори на процеса. **Информация за процес: pid\_t getpid(void);** – Връща идентификатора на текущия процес. **pid\_t getppid(void);** – Връща идентификатора на процеса-баща. **Потребителски идентификатор на процес:** С всеки процес се свързват два потребителски идентификатора - реален (**ruid**) и ефективен (**euid**). Процесите-синове ги наследяват от своите бащи. Реалният е идентификаторът на потребителя, който е създал процеса. Ефективният е идентификаторът на потребителя, според който се определят правата на процеса при работа с файлове и при изпращане на сигнали. Всъщност за всеки процес има още един потр идентификатор **sid**, който се използва при различни схеми за промяна на потр идент-ри. Промяна на потр-идент-ри на процес може да се осъществи по два начина: 1) когато то изпълнява системния примитив ехес и новата програма е setuid, т.е. файлът, съдържащ програмата има код на защита, в който бита SUID е вдигнат. Тогава **euid** се променя със собственика на файла, който съдържа програмата. Този начин на промяна се използва за временно повишаване на правата на потребителя за да могат те да изпълняват някои команди. 2) чрез системния примитив **int setuid(uid\_t uid);** Ако текущият **euid** на процеса е root, то се променят **ruid, euid и sid** със стойността на аргумента uid. Ако текущият **euid** на процеса не е root, то **euid** се променя с аргумента uid, само ако uid = **ruid** или uid = **sid**. При успех **setuid** връща 0, при неуспех -1. Системните примитиви **getuid и geteuid** се използват за извличане на стойността на потребителските идентификатори на процес. Те имат следните прототипи: **uid\_t getuid ( ); uid\_t geteuid ( ).** Getuid връща реалният потребителски идентификатор на процеса, **geteuid** връща ефективният потребителски идентификатор на процеса. И двата системни примитива винаги завършват успешно. **17. Взаимно изключване. Алгоритъм на Декер. Алгоритъм на Питерсен:** В този раздел ще разгледаме проблемите при междупроцесни комуникации (Interprocess Communication или IPC) и някои механизми за решаването им. Проблемите при комуникации между процеси имат два аспекта. Първият е **предаване на информация между процесите**. Другият е свързан със **съгласуване на действието на процесите**, които работят асинхронно, така че да се гарантира правилното им взаимодействие. **Най-простият начин, по който два или повече процеса могат да взаимодействат е да се конкурират за достъп до общ ресурс:** Например, два процеса P и Q четат и пишат в обща променлива брояч counter, като всеки увеличава променливата. Нека значението на counter е 7 и достъпът на двата процеса до нея се извърши в следния ред: 1.(P) Чете counter в локална променлива br; 2.(Q) Чете counter в локална



променлива pb; 3.(Q) pb = pb + 2; 4.(Q) Записва pb в counter; 5.(P) pa = pa + 1; 6.(P) Записва pa в counter. При тази последователност на изпълнение на двата процеса резултатът в counter ще е неправилен - 8, а не 10, тъй като изменението на процеса Q ще се загуби. Такава ситуация, при която два или повече процеса четат и пишат в обща памет и крайният резултат зависи от реда, в който работят процесите се нарича **състезание (race condition)**. Как да се избегне състезанието? Решението на този проблем се нарича **взаимно изключване (mutual exclusion)**, т.е. по такъв начин да се организира работата на двата (или повече) процеса, че когато един от тях осъществява достъп до общия ресурс (изпълнява трите стъпки в примера) за другия (другите) да се изключи възможността да прави същото. **Друг по-сложен начин на взаимодействие на два или повече процеса е когато те извършват обща работа:** Съществува няколко класически модела на взаимодействието на процеси, извършващи обща работа, например: **1.производител – потребител:** Всеки елемент от данни трябва да е подаден на потребителя и обработен точно веднъж. Понеже процесите работят асинхронно е необходимо механизъм за тяхната синхронизация. Ако за извършването на тази задача се използва обща памет, то механизъмът за синхронизация се състои в това, да се забрани на производителя да пише в пълен буфер и на читателя да чете от празен буфер; **2.читатели – писатели:** Тук имаме няколко процеса които пишат в дадена база данни и няколко които четат от нея. Ако няма синхронизация би настъпил хаос. Решението на проблема е следното: Или много читатели имат достъп до базата едновременно или само един писател; **3.задача за обядващите философи;** **4.клиент – сървър.** Това, което е необходимо за коректното взаимодействие на процесите (освен евентуално взаимно изключване), се нарича **синхронизация (synchronization)**. **Механизми за комуникация между процесите:** - **Общата памет** е най-бързия механизъм, защото използва машинните команди, но е и най- несигурния. Това е така, защото не се осигурява никаква синхронизация; - **Семафорите** са механизъм предложен от Дейкстра, като механизъм допълващ общата памет. Този механизъм решава проблема със същата обща памет и синхронизацията на процесите; **Съобщения** - най-безопасния механизъм, тъй като има вградени елементи за избягване на състезанието и осигуряване на синхронизацията. **Взаимно изключване:** Проблемът за избягване на състезанието е бил формулиран от Е. Дейкстра (E.Dijkstra) чрез термина **критичен участък (critical section)**. Част от кода на процеса реализира вътрешни изчисления, които не могат да доведат да състезание. В друга част от кода си процесът осъществява достъп до обща памет или върши неща, които могат да доведат до състезание. Този част от програмата ги наричаме критичен участък и ще казваме, че процес е в критичния си участък, ако е започнал и не е завършил изпълнението му, независимо от състоянието си. За избягване на състезанието и коректното взаимодействие на конкуриращите се процеси трябва да са изпълнени следните условия: 1.Бъв всеки един момент най-много един процес може да се намира в критичния си участък (взаимно изключване); 2.Никой процес да не остава в критичния си участък безкрайно дълго; 3. Никой процес, намиращ се вън от критичния си участък, да не пречи на друг процес да влезе в своя критичен участък; 4.Решението не бива да се основава на предположения за относителните скорости на процесите. **Алгоритъм с редуване на процесите:** #define TRUE 1 shared int turn=0; P0(){while(TRUE){while (turn != 0);critical\_section0();turn = 1;noncritical\_section0();}} P1(){}.Този алгоритъм използва една обща променлива turn, чийто значение е номер на процес, който е на ред да влезе в критичния си участък. В този алгоритъм и останалите в раздела общата памет ще записваме като деклариране на променлива с думата shared. Двата процеса строго се редуват. Недостатък на това решение е нарушаване на изискване 3. Ако един от процесите е по-бавен това ще пречи на другия процес да влиза по-често в критичния си участък, или ако един от процесите завърши другият повече няма въобще да може да влиза в критичния си участък. **Алгоритъм на Декер:** #define FALSE 0 #define TRUE 1 shared int wants0=FALSE, wants1=FALSE; shared int turn=0; P0(){ while (TRUE) { wants0 = TRUE; while (wants1) if (turn == 1) { wants0 = FALSE; while (turn == 1); wants0 = TRUE; } critical\_section0(); turn = 1; wants0 = FALSE; noncritical\_section0(); }}. Този алгоритъм използва три общи променливи за осигуряване на взаимно изключване. Променливите wants0 и wants1 са флагове за всеки от процесите. Флаг FALSE означава, че съответният процес не е в критичния си участък и не иска вход. Когато процес иска вход в критичния си участък, той вдига флага си (TRUE). И този алгоритъм използва променлива turn, чийто значение е номер на процес, който е на ред да влезе в критичния си участък, но двата процеса не се редуват строго. Променливата се използва когато и двата процеса желаят вход в критичните си участъци, за да разреши конфликта. **Алгоритъм на Питерсон:** shared int turn=0; shared int interested[2] = {FALSE, FALSE}; enter\_region(int process){int other; other = 1 - process; interested[process] = TRUE; turn = process; while (turn==process &&interested[other]==TRUE); } leave\_region(int process) { interested[process] = FALSE; } P0(){ while (TRUE) { enter\_region(0); critical\_section0(); leave\_region(0); noncritical\_section0(); }} Алгоритъмът на Питерсон също използва три общи променливи за осигуряване на взаимно изключване на два процеса. Променливите interested[2] също са флагове на процесите. И тук се използва променлива turn, чийто значение е номер на процес, който е на ред да влезе в критичния си участък. Но наистин, по който променливата turn се изменя и проверява, тук е по-различен. **Основният недостатък на двата алгоритъма;** че за осигуряване на взаимното изключване се използва активно чакане (busy waiting). Всеки от процесите, който желае да влезе в критичния си участък изпълнява цикъл, в който непрекъснато проверява дали това е възможно, докато стане възможно. Много по-естествено и ефективно би било, когато процес не може да влезе критичния си участък да бъде блокиран и когато влизането стане възможно операционната система да го събуди. Освен това и двата алгоритъма реализират взаимно изключване на два процеса. Алгоритъм за взаимно изключване на n процеса, известен като Bakery algorithm, е бил предложен от Лампорт но е по-сложен.

масив от семафори s [N], по един за всеки философ, по който той се блокира когато трябва да чакат освобождаването на вилиците. #define N 5 #define TRUE 1 #define LEFT (-1)%N #define RIGHT (+1)%N #define THINKING 0 #define HUNGRY 1 #define EATING 2 shared int state[N]={0,0,0,0,0}; semaphore mutex=1; semaphore s[N]={0,0,0,0, 0}; philosopher(int i) { while(TRUE) { think(); take\_forks(i); eat(); put\_forks(i);} take\_forks(int i){ P(mutex); state[i] = HUNGRY; test(i); V(mutex); P(s[i]); } put\_forks(int i){ P(mutex); test(LEFT); test(RIGHT); V(mutex); } V(mutex); } if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {state[i] = EATING; V(s[i]);}} При използването на обща памет и семафори за междупроцесни комуникации отговорността за правилното взаимодействие на процесите е на програмиста. Поради това този метод не е безопасен. Грешки в програмите могат да доведат до дедлок, състезание и други форми на непредсказуемо и невъзпроизводимо поведение. **19. Съобщения.** Между процесите се установява комуникационна връзка. Процесите обменят съобщения чрез примитиви:send(destination, message) и receive(source, message). За да се осъществи предаване на едно съобщение между два процеса, е необходимо единият процес да изпълни send, а другият receive. При конкретна реализация на механизъм за съобщения му процеси трябва да се решат следните въпроси относно логическите характеристики на комуникационната връзка: **1)** Как се установява комуникационна връзка му процесите? **2)** Може ли една ком.връзка да свързва повече от два процеса? **3)** Колко ком.връзки може да има м/у два процеса? **4)** Еднопосочна или двупосочна е ком.връзка? **5)** Какво е капацитет на ком.връзка? Размерът и структурата на съобщенията касаят физическата организация на връзката. **6)** Има ли някакви изисквания за размер и/или структура на съобщенията предавани по определена комуникационна връзка? **Адресиране на съобщенията: Директна комуникация:** При този метод destination и source са идентификатори на процеси.За да се предаде съобщение от процес P към процес Q, процесът P трябва да изпълни send (Q, message), а процесът Q трябва да изпълни receive (P,message). Отговорите: **1)** комуникационната връзка се създава автоматично м/у двата процеса, но всеки от тях трябва да знае идентификатора на другия; **2)** ком.връзка може да се създаде само м/у два процеса; **3)** между два процеса може да съществува точно една ком.връзка; **4)** ком.връзка е двупосочна – например, възможно е изпращане на потвърждение за получено съобщение: процес P - send(Q, message), receive (P, message); процес Q - receive (Q, message), send (P, "acknowledgement"). Асиметричен вариант на директна комуникация - receive(ANY, message); **Косвена комуникация:** Въвежда се нов обект, наричан пощенска кутия (mailbox), опашка на съобщенията (message queue) или порт (port). Съобщения се изпращат в и получават от специален обект: пощенска кутия,опашка на съобщенията. Процес P - send(A, message). Процес Q - receive(A, message); **1)** Комуникационната връзка се установява между процесите когато те използват обща пощенска кутия. **2)** Тя може да свързва и повече от два процеса **3)** Между два процеса може да съществува повече от една ком. връзка. **4)** Ком.връзка може да е еднопосочна или двупосочна. В този случай освен примитивите send и receive трябва да има и примитив за създаване на пощенска кутия. Една възможна реализация е собственик на пощенската кутия да става процесът, който я създаде. Всеки друг процес, който знае името на пощенската кутия и на който собственикът е дал права, може да я използва. Унищожаването на пощенска кутия може да се реализира чрез системен примитив, извикван явно от собственика ѝ. Друга възможност е процесите да могат да ползват обща пощенска кутия чрез механизма за създаване на процеси и когато последният процес, ползващ определена пощенска кутия, завърши тя да се унищожава автоматично от системата. **Буфериране на съобщенията:(5 и 6) Съобщения без буфериране:** Процесите изпращач и получател се синхронизират в момента на предаване на всяко съобщение.Не може да има изчакане. Предаване на съобщения по метода на **рандеву.** **Съобщения с автоматично буфериране:** Определен брой (обем) изпратени и още неполучени съобщения временно се съхраняват в буфер на комуникационната връзка;След като изпълни send, изпращачът не може да знае дали съобщението му е получено. **Примери за реализация на съобщения: UNIX System V:** Косвена комуникация - опашка на съобщенията; Автоматично буфериране: Структура на съобщение struct msgbuf { long mtype; char mtext[N]; }; **1)** Опашка на съобщения се създава с примитив **msgget**. **2)** Процесът-собствник на опашката определя правата на другите процеси. **3)** Опашка се унищожава явно с примитив **msgctl**. **4)** Съобщение се изпраща или получава с примитивите **msgsnd** и **msgrcv**. **5)** Няколко потока за предаване на съобщения в рамките на една опашка (двупосочна комуникационна връзка). **MINIX:** Директна комуникация; Без буфериране; Съобщения с фиксирана дължина: **UNIX, MINIX, LINUX:** неименован програман канал (unnamed pipe или pipe) - за комуникация между родствени процеси; именован програман канал (named pipe или FIFO файл) - за комуникация между независими процеси. Като механизъм за комуникация те са еднакви. Реализират се като тип файл, който се различава от обикновените файлове и има следните особености - за четене и писане в него се използват системните примитиви read и write, но дисциплината е FIFO. Програмният канал може да се разглежда като механизъм на съобщения с косвена адресация и автоматично буфериране. Разликата е, че в програмния канал няма граници между съобщенията, т.е. процес P може да запише едно съобщение от 1000 байта, а процес Q да го прочете като 10 съобщения от по 100 байта или обратно.

**20. Семафори. Взаимно изключване чрез семафори. Синхронизация чрез семафори:** механизъм за междупроцесни комуникации(Дейкстра) **семафори (semaphores)**. С всеки семафор се свързва цяла променлива - брояч и списък на чакащи в състояние блокиран процес. Значението на брояча е неотрицателно число. За семафорите Дейкстра определи три операции - инициализация, операции P и V. При инициализацията се създава нов обект семафор и се зарежда начално значение в брояча му. Операцията P проверява и намалява значението на брояча на семафора, ако това е възможно, в противен случай блокира процеса и го добавя в списъка на чакащи процеси. Операцията V събужда един блокиран по семафора процес, ако има такива, а в противен случай увеличава брояча на семафора. **ВЗАИМНО ИЗКЛЮЧВАНЕ ЧРЕЗ СЕМАФОР:** За осигуряване на взаимното изключване на произволен брой процеси е необходим един семафор, инициализиран с 1. Освен това всеки от процесите трябва да зареди критичния си участък с операциите P и V над този семафор. Такъв семафор се нарича двоичен (тука се пише още простия код). **СИНХРОНИЗАЦИЯ ЧРЕЗ СЕМАФОРИ:** Съществува няколко **класически задачи за междупроцесни комуникации**. Сега ще разгледаме решаването на някои от тези задачи чрез механизмите обща памет и семафори. Пример: имаме два процеса P1 и P2, които работят асинхронно и искаме операторите S1 в процеса P1 да се изпълнят преди S2 в P2. Решението е следното: **semaphore s=0; P1(){ ... S1; V(s); ... } P2(){ ... P(s); S2; ... } **Производител – Потребител:** Задача: Производител-V-UNIX и LINUX: когато комуникалният интерпретатор изпълнява конвейер, who I wc -l, той решава задачата Производител-Потребител. Има два процеса, които взаимодействат, като извършват обща работа. Процесът-производител (who) произвежда данни, които се предават на процеса-потребител (wc), който ги използва. Командните интерпретатори в UNIX и LINUX решават тази задача чрез програман канал. Тук ще разгледаме решение, в което използваме обща памет за предаване на данните от производителя към потребителя и семафори за синхронизация. Предполагаме, че двата процеса използват общ буфер, който може да поеме N елемента. Производителът записва в буфера всеки произведен от него елемент, а потребителят чете от буфера елементите, за да ги обработи. Двата процеса работят едновременно, с различни и неизвесни относителни скорости. Следователно, задачата за синхронизация се състои в това да не се позволи на производителя да пише в пълен буфер, да не се позволи на потребителя да чете от празен буфер и всеки произведен елемент да бъде обработен точно един път. Освен това трябва да се осигури и взаимно изключване при достъп до общия буфер. Решението на Дейкстра използва три семафора: mutex за взаимното изключване, empty и full за синхронизацията. Броячът на empty съдържа броя на свободните места в буфера и по него производителът ще се блокира когато няма място в буфера. Семафорът full ще брой запълнените места в буфера и по него потребителят ще се блокира когато в буфера няма данни.#define N 100 #define TRUE 1 shared buffer buf; semaphore mutex = 1; semaphore empty = N; semaphore full =0; producer(){int item; while (TRUE){ produce\_item(&item); **P(empty); P(mutex);** insert\_item(&item); **V(mutex); V(full);}** consumer(){ int item; while (TRUE){ **P(full); P(mutex);** remove\_item(&item); **V(mutex); V(empty);** consume\_item(&item); }}. **Читатели – Писатели:** Тази задача е модел на достъп до обща база данни от много конкурентни процеси, които се делят на два вида. Единият вид са процеси-читатели, които само четат данните в базата, а другият вид са процеси-писатели, които изменят по някакъв начин данните в базата. Искаме във всеки момент достъп до базата данни да могат да осъществяват или много процеси- читатели или един процес-писател. Тези задачи за синхронизация е решена от Куртоа, Хейманс и Парнас чрез една обща променлива брояч и два двоични семафора. #define TRUE 1 shared int rcount=0; semaphore mutex=1;(достъп до count) semaphore Pdb=1; reader(i){ while(TRUE) { **P(mutex); rcount = rcount + 1; if (rcount == 1) { P(db); V(mutex);** read\_data\_base(); **P(mutex); rcount = rcount - 1; if (rcount == 0) V(db); V(mutex); }** writer(i){ while (TRUE) { collect\_data(); **P(db);** write\_data\_base(); **V(db);}** Ако никой от процесите не осъществява достъп до базата данни, то двата семафора са с 1 и rcount е 0, тогава първият процес, който се появи ще изпълни P(db) и ще затвори този семафор. Ако това е читател, то rcount ще стане 1 и следващите читатели само ще увеличават rcount и ще започват да четат базата данни. Когато читател завърши четенето той намалява rcount, а последният читател изпълнява V (db) и ще отвори семафора db. Ако първият процес, получил достъп до базата данни е писател, то първият читател ще се блокира по семафора db, следващите читатели ще се блокират по mutex, а следващите писатели ще се блокират по db. Когато писателят завърши писането той ще изпълни V (db) и с това ще събуди един процес, чакащ за достъп до базата данни. Ако това е първият чакащ читател, той ще изпълни V (mutex) и ще събуди следващия чакащ читател и т.н. докато всички читатели бъдат събудени. Недостатък на това решение е, че то дава преимущество на читателите, което в една реална база данни не е добра стратегия. **Задача за обядващите философи:** Задачата е модел на поведението на процеси, които се състезават за монополен достъп до ограничен брой ресурси. Дейкстра - чрез семафори и обща памет. Пет философа седят около кръгла маса, като пред всеки от тях има чиния със слатети, а между всеки две чинии има само по една вилица. Животът на всеки философ представлява цикъл, в който той размислява, в резултат на което огладнява и се опитва да вземе двете вилици около своята чиния. Ако успее известно време се храни, а след това връща вилиците. Задачата е да се напише програма, която да прави това, което се очаква от философа. #define N 5 philosopher(int i){ while(TRUE){ think(i); takefork(i); takefork((i+1)%N); eat(i); putfork(i); putfork((i+1)%N);}} Грешката в това решение е, че може да доведе до дедлок. Ако всичките пет философа вземат едновременно левите си вилици. fix) След think ( ) се направи критичен участък. Храни се най-много един философ(пощо). Решението на Дейкстра използва общ масив state [N], като всеки елемент описва състоянието на съответния философ. Възможните състояния са: мисли, гладен е (опитва се да вземе двете вилици) и храни се. Достъпът до общия масив се регулира от двоичен семафор mutex. Освен него се използва**

**20. Планиране на процесите. Дисциплини на процесите. Нива на планиране:** При наличие на много заявки за използване на определен ресурс, вземането на решение коя да бъде удовлетворена се нарича **планиране**. Този част от ядрото, която избира най-подходящия процес за текущ и решава колко дълго ще работи се нарича **планировчик (scheduler)**.**1)** Планиране на високо ниво(на заданията)-Решава кое от чакащите задания да бъде заредено в системата и да се създадат задачи (процеси) за него **2)** Планиране на ниско ниво-Планировчикът на това ниво определя кой от готовите процеси да бъде избран за текущ и колко време да работи **3)** Планиране на междинно ниво (на свопинга) - Планировчикът управлява извървянето на процеси на диска в свопинг областта и обратното им връщане в ОП. **Цели на планиране:** **1)** Справедливост-времето на ЦП да се разпределя справедливо между процесите.**2)** Баланс-пълно натоварване на всички ресурси на системата **3)** Ефективност-по-голяма част от времето на ЦП да се използва за изпълнение на потребителски процеси, а не за служебни цели. **4)** Време за отговор (response time)-времето за обслужване на една потребителска заявка. т.е. времето от въвеждане на данни или команда от потребителя до получаване на поредния отговор (важно при интерактивни процеси) **5)** Време за чакане (waiting time)-общото време, през което процесът чаква обслужване в състоянието готов. Този чел е важна при пакетни процеси (процеси, изпълнявани във пакетен или фонов режим). Там не е важно как процесът работи във времето, а да се намали общото време, през което той е в системата. **6)** Предсказуемост-да се гарантира завършването на всеки процес, т.е. да не се случва безкрайно отлагане. **Видове процеси:** Работата на всеки процес е последователност от редуации се интервали с различни дължини: **1)** интервал, в който процесът изпълнява команди на ЦП (CPU burst) **2)** интервал, в който процесът чаква В/И (I/O burst).(a) Процес ограничен от ЦП (CPU bound)(b) Процес ограничен от Вход/Изход (I/O bound)  **Дисциплини на планиране:** **“Когато работи планировчикът? 1)** Когато текущият процес завърши, като изпълни exit. **2)** Когато текущият процес извика примитив, изискващ блокиране като read, write, wait или друг. Ако планировчикът работи само в тези два случая, се казва че реализира **планиране без преразпределение** (nonpreemptive scheduling). **3)** След обработката на апаратно прекъсване, преди процесът да се върне в потребителска фаза.Ако планировчикът работи и в този случай, се казва че реализира **планиране с преразпределение** (preemptive scheduling).**1)** FCFS (First-Come-First-Served) Процесите се обслужват в реда на появяването им. Има една опашка на готовите процеси, в която новите процеси се добавят в края. Същото се прави и за процесите, които са били блокирани, след като бъдат деблокирани. Когато работи планировчикът избира първия процес в опашката и го изключва от нея. Всеки избран процес работи докато се блокира или завърши, т.е. това е дисциплина **без преразпределение и без приоритети** на процесите. Основното преимущество е, че това е лесна за разбиране и реализация дисциплина. Освен това е справедлива, в смисъл че предоставя на всички процеси еднакви услуги, т.е. еднакво средно време за чакане. Недостатъците са повече: Не е приложима в интерактивни ОС, защото няма преразпределение. Кратките процеси чакат толкова колкото и дългите, което някой ще каже, че не е справедливо. **2)** SJF (Shortest-Job-First)-без преразпределение,приоритетна дисциплина – приоритетът е обратно пропорционален на очакваното време за изпълнение на процеса Точното време за изпълнение е неизвестно предварително, затова се използва очаквано време, задавано от потребителя. Това е един от недостатъците, тъй като потребителят сам определя приоритета на процесите си. Неприложима е за интерактивни процеси, тъй като се иска оценка за необходимото време за изпълнение. **3)** SRT (Shortest-Remaining-Time)-сперразпределение,приоритетна дисциплина - приоритетът е обратно пропорционален на оставащо време = очаквано време - получено време .Кратките процеси се обслужват с предимство, следователно се намалява тяхното време за чакане;Не е приложима в интерактивни ОС, защото се иска оценка на очакваното време. Общ недостатък на последните две дисциплини е задържането на дългите процеси, което може да доведе до безкрайното им отлагане. **4)** RR (Round-Robin) или циклична дисциплина:Всички готови процеси са подредени в опашка по реда на появяването им (създаване или деблокиране).Планировчикът избира първия процес от опашката и му определя интервал време - квант (quantum), през който той може да използва ЦП;Ако при изтичане на кванта процесът не е завършил или не се е блокирал, то му се отнема ЦП и се поставя в края на опашката на готовите процеси (с преразпределение).Дава предимство на кратките процеси без да дискриминира дългите;Осигурява приемливо време за отговор в интерактивните ОС за процесите ограничени от В/И;Проста за реализация е;Справедлива е - всички процеси са равнопранви. **Дисциплини с няколко опашки:** Процесите да се класифицират в няколко класи;За всеки клас да се поддържа опашка на готовите процеси;Всяка опашка има свой приоритет и може да се обслужва с различна дисциплина;Когато ЦП се освободи планировчикът избира процес от непразната опашка с най-висок приоритет и го обслужва според дисциплината на опашката му.Как процесите се разпределят по опашките? Връзката между процес и опашка е статична в MINIX(определен процес винаги попада в една и съща опашка когато е готов); Опашки с обратна връзка-в UNIX. Дисциплината се адаптира към поведението на процеса като повишава

или намалява приоритета му. **Планиране на процесите в UNIX:** Алгоритъмът за планиране на процесите в UNIX използва няколко опашки с обратна връзка. Всяка опашка има свързан с нея приоритет. Приоритетите са цели числа, като по-малко значение означава по-висок приоритет. Диапазонът на приоритетите се дели на два непресичащи се класа: потребителски приоритети и системни приоритети, които са по-високи от първите. Когато процес минава в състояние блокиран, му се дава съответен системен приоритет. Когато процес се връща от система в потребителска фаза му се дава потребителски приоритет, защото може да е бил блокиран и приоритетът му е системен.При обработка на прекъсване от апаратния таймер на всяка секунда се преизчисляват всички потребителски приоритети по формулата:  $priority = CPU/2 + base + nice$ (има картинка)

**Планиране на процесите в MINIX:** Алгоритъмът за планиране на процесите в MINIX използва три опашки. Всяка опашка има свързан с нея приоритет. В опашката с най-висок приоритет попадат един специален вид системни процеси, наречени входно-изходни задачи. Във всяка В/И задача работи част от операционната система, която реализира драйвер на някой тип В/И устройство. Затова тази опашка се обслужва по дисциплината FCFS. В средната опашка се нареждат така наречените процеси сървери. Те са два: FS (File Server) и MM (Memory Manager). Те реализират системните примитиви на операционната система, съответно FS - за работа с файлове и MM - за управление на процеси. Затова и тази опашка се обслужва по дисциплината FCFS. В опашката с най-нисък приоритет са потребителските процеси, които се обслужват с дисциплина RR с квант 100 msec. В/И задача или процес-сървер никога не се свалят от ЦП, независимо колко дълго са работили.

**21. Deadlock, Понятие:** Множество от два или повече процеса са в дедлок, ако всички те са в състояние блокиран и всеки чака настъпването на събитие, което може да бъде предизвикано само от друг процес в множеството. -Събитието е предоставяне на ресурс; Типове ресурси – всеки от тях може да има няколко идентични екземпляра;Ресурси могат да са: апаратни устройства – печатачо устройство, лентово устройство и др. Или данни – запис в системна структура, файл или част от файл и др. Стъпките при работа на процес с ресурс: 1) заявка за ресурс (request) 2) използване на ресурс (use) 3) освобождаване на ресурса (release)

**Условия за Deadlock: 1)** Взаимно изключване (Mutual exclusion)В системата има поне един ресурс, който трябва да се използва монополено. **2)** Очакване на допълнителен ресурс (Hold and Wait) Процесите могат да получават ресурси на части, като съществува поне един процес, който задържа получени ресурси и чака предоставяне на допълнителен ресурс. **3)** Непреразпределение (No preemption) Системата не отнема наситеното ресурс, предоставен на процес.(Процестъ го освобождава.) **4)** Кръгово чакане (Circular wait) - Съществува множество от процеси  $\{p_1, p_2, \dots, p_k\}$ , такива че  $p_1$  чака ресурс държан от  $p_2$ ,  $p_2$  чака ресурс държан от  $p_3$  и т.н.  $p_k$  чака ресурс държан от  $p_1$ .

**Граф на разпределение на ресурсите:** Двуделен ориентиран граф  $G = \{V, E\}$ : **1)** Множеството на върховете  $V$  е обединение на две непресичащи се подмножества:  $\{r_1, r_2, \dots, r_n\}$  – множество на процесите  $\{f_1, f_2, \dots, f_m\}$  – множество на типовете ресурси **2)** Множеството на ребрата  $E$  включва два вида ребра:  $(p_i, r_j)$  – означава, че процес  $p_i$  иска ресурс  $r_j$  (request edge) ;  $(r_j, p_i)$  – означава, че един екземпляр от ресурс  $r_j$  е даден на процес  $p_i$  (assignment edge) Ако в графа няма цикъл, то в системата няма дедлок.Ако графът съдържа цикъл, то може да има дедлок. Наличието на цикъл в графа е необходимо условие за дедлок.

**Предотвратяване на Deadlock: Стратегии на Хавендер** Ако едно от необходимите условия не е изпълнено дедлок е принципно невъзможен.**1)** Взаимно изключване -Никой ресурс никога не се предоставя за монополено използване. **2)** Очакване на допълнителен ресурс-Всеки процес трябва да иска всичките необходими му ресурси наведнъж и преди започване на работа, и те да му бъдат предоставени преди началото на изпълнение. Процес може да иска ресурси и след започване на работа, само когато не задържа никакви други ресурси. **3)** Непреразпределение-Ако процес, който е получил и държи никакви ресурси, поиска допълнителни и системата не може да му ги предостави веднага, то той бива блокиран и всички дадени му до момента ресурси му се отнемат. Процестъ ще продължи изпълнението си, когато системата може да му даде всичките ресурси – новите (поисканите) и старите (отнетите му). **4)** Кръгово чакане -Въвежда се наредба на всички типове ресурси - определяме функция  $F: R \rightarrow N$ .Ако първоначално процес е получил ресурси от тип  $r_i$ , то след това може да иска само ресурси от тип  $r_i$ , където  $F(r_i) > F(r_j)$ .Когато процес иска ресурс от тип  $r_i$ , той трябва да е освободил всички ресурси от тип  $r_i$ , за които  $F(r_i) \geq F(r_j)$ .

**Заобикаляне на Дедлок: Алгоритъм на банкера:** Всеки процес трябва да даде предварителна информация за максималния брой екземпляри от всеки тип ресурс, които може да поиска. Състоянието на разпределение на ресурсите се описва чрез:-Брой свободни ресурси;-Максимален брой ресурси за всеки процес;-Брой ресурси дадени на всеки процес;-Брой ресурси, които процесът може още да поиска.Казваме, че системата се намира в **надеждно състояние (safe state)** на разпределение на ресурсите, ако съществува поне една последователна наредба на процесите  $\langle p_1, p_2, \dots, p_n \rangle$ , за която е изпълнено следното: нуждите на всеки процес  $p_i$  могат да се удовлетворят от свободните в момента ресурси и ресурсите държани от процесите  $p_j$ , където  $j < i$ . Следователно, от текущото разпределение на ресурсите съществува някаква последователност от други състояния, в която системата може да удовлетвори максималните потребности на всеки процес и той след време да завърши. Когато за текущото разпределение на ресурсите не съществува нито една такава последователност се казва, че състоянието е ненадеждно. **Алгоритъмът:**Анализира всяка заявка за ресурс и ако има свободни ресурси и новото състояние остава надеждно я удовлетворява.В противен случай процесът се блокира и ще бъде събуден когато системата е в състояние да удовлетвори заявката му.

**22. Нишки, Понятие:** Нишката представява част от процес, но не са процеси. Моделът на процесите се базира на две независими концепции:-групиране на ресурсите - процесът има различни ресурси;-изпълнение на програма - процесът е последователност на изпълнение на команди (thread of execution или накратко само thread). Тези два аспекта на процеса могат да бъдат разделени и така се появява понятието нишка. Процестъ се използва за групиране на ресурсите, а нишките са обектите, изпълнявани от ЦП. Всяка последователност на изпълнение в рамките на процес се нарича нишка, т.е. нишката е част от процес, която има собствен набор от регистри и стек.Идеята на въвеждането на понятието нишка е процесът да може да има няколко последователности на управление, т.е. да е конкурентен, а не последователен процес.Процес,включващ няколко нишки-**multithreaded process**. Различните нишки в един процес не са така независими, както производителност и яснота на проекта ни.

**Елементи на нишка:** Регистри – РС и др.,Стек,Състояние – текуща, готова, блокирана, зомби.

**Елементи на процес:** Адресно пространство,Глобални променливи, Отворени файлове,Текущ каталог.

**Реализация на нишки: В потребителското пространство:** Всеки процес има собствена таблица на нишките,run-time системата реализира всички операции с нишки,run-time системата се намира и работи в потребителското пространство. Няма прекъсване и вход в ядрото при операции с нишки.Проблеми-Ако при изпълнението на примитив,който блокира процеса(като read, write...) се наложи, се блокира целият процес, а не само нишката, извикала примитива.-противоречие на основната идея за използване на нишки в приложениа, а именно когато една нишка в процеса се блокира друга негова нишка да продължи работа. Друг проблем се проявява при планиране на нишки. Когато една нишка започне работа тя ще работи докато доброволно не освободи ЦП. Това е така, защото планировчикът на нишките не може да работи докато управлението не се предаде в run-time системата. Следователно, планирането на нишките в един процес е без преразпределение.

**В пространството на ядрото:** В ядрото има една глобална таблица на нишките и таблица на процесите. В този случай в потребителското пространство няма run-time система и Таблица на нишките. Всички операции с нишки са реализирани като системни примитиви (изискват прекъсване и вход в ядро). Когато нишка се блокира, ядрото може да избере друга нишка от същия или от друг процес.Следователно, системните примитиви като read не създават проблеми. Недостатък на този метод е, че операции с нишки стават по-бавни (включват прекъсване, съхранение на контекста и възстановяването му).

**Основни операции с POSIX нишки: Създаване на нишка:** Първата - главна нишка на процес се създава автоматично. Друга нишка се създава чрез функцията: `#include <pthread.h>`  
`int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);`

`thread` - идентификатор на създадената нишка; `attr` - атрибути на нишката: **1)** тип на нишката - joinable / detached Тип joinable означава, че друга нишка в процеса може да се синхронизира с момента на завършването ѝ, т.е. след завършване на нишката тя не изчезва веднага (подобно на състояние зомби при процеси). Тип detached означава, че при завършване на нишката веднага се освобождават всички ресурси, заемани от нея, т.е. тя изчезва в момента на завършване. По премълчаване нишката се създава joinable **2)** дисциплина и параметри при планиране на нишки; `start_routine` - началната функция на нишката; `arg` - аргумент, предаван на функцията `start_routine`

**Завършване на нишка:** Нишка завършва когато: Изпълни return от `start_routine`; Извика явно `pthread_exit`; Друга нишка я прекрати чрез `pthread_cancel`

`void pthread_exit(void *retval);`Аргументът `retval` е код на завършване, който нишката изработва. Той е предназначен за всяка друга нишка на процеса, която изпълни `pthread_join`. Но ако нишката е от тип detached, то след `pthread_exit` от нея не остава никаква следа; следователно и кода не се съхранява. Няма връщане от тази функция.**Изчакване завършването на нишка:** `int pthread_join(pthread_t thread, void **value_ptr);` `thread` - идентификатор на нишка, чието завършване се чака; `value_ptr` - код на завършване,върнат от нишката `thread`. Всяка нишка в процес може да се синхронизира със завършването на всяка друга нишка в процеса. Нишката `thread` трябва да е от тип joinable. За всяка нишка в процеса може да се изпълни най-много един `pthread_join`.