Операционни системи

Софтуерно инженерство 2015/2016г.;

Теория за изпит

Преподавател: Георги Георгиев – Скелета

Изготвил: Ружа Боботанова

**1. Неформално описание на ОС**

Първият проект за реализиране на операционна система е дело на IBM. Компанията е била и продължава да бъде лидер в IT индустрията. Системата се е казвала OS/360. Тогава по нея са работели над 5000 програмиста и крайния резултат е бил с много големи и фатални грешки, които не дават очаквания резултат. Това е първата сериозна ОС и създаването е описано в книга.

След неуспехите от IBM, други компании от натрупания опит и неуспеха, натрупват знания, умения, математически изчисления, анализ, се създават действащи ОС. ОС имат за задача да поддържат математически изчисления, реална изчислителна система с много функционалности за олесняване на работата. За организацията на тази работа се нуждаем от софтуер, който да реализира тези идеи, а именно това е операционната система.

- преди са се писали на много стари езици

- в компютъра има много периферни устройства – не е само софтуър, има хардуер и т.н.

- първо се пишат програми, които работят с периферните устройства, защото винаги се налага работа с тях – четене на данни отвън, извеждане

- първата услуга, която възниква и се смята за част от ранните ОС – **да се направят единни инструменти за работа с хардуера или за цели класове от устройства – пр. за четене на магнитна лента, за четене на запис, за работа с диск и др.**

- да има готови инструменти за следващи операции, които да олесняват работата

- този тип дейност днес се нарича абстрахиране на хардуера, т.е. да заменим реалния хардуер, който използваме с виртуално устройство, което да е веднъж добре дефинирано и да ползваме само него, а самата работа с хардуера да се върши от софтуер, който е част от ОС.

|  |  |
| --- | --- |
| програмист | Когато програмистът пише програма, ОС му доставя необходимите ресурси, грижи се програмиста да получи необходимото, чрез **логически устройства** |
| ОС | Управител на ресурси; предоставя абстракции; има програма драйвер устройство – **предоставя абстрактния вид на информацията във вид на масив от байтове;** |
| хардуер | Физическия диск? Записва 0 и 1; има си специфики спрямо скоростта на главата на диска |

**2. Основни абстракции:**

**Файл** – абстракция от обособена информация, масив от байтове, има си име и може да е достъпен от другите агенти в системата

- **файл** - семантично обособени парчена информация, които логическата структура на входно-изходната информация се предоставя във вид на файл **–** това е друго ниво на абстракция, масив от байтове

- ако разглеждаме **ОС като удобство** – искаме удобствата да съществуват дълго във времето и да не зависят от конкретните агенти (хардуерни устройства). Това се постига като дисковете (хардуерите) се менят и излизат нови, но след като имаме логическото ниво на абстракция за диска е достатачно да сменим драйвера, за да използваме съществуващата информация. Върху тази абстракция се дефинират файлове, които са като управител на ресурсите, които вече чрез високите нива на абстракция ще дефинират как да се визуализират файловете и записаната информация в тях.

Важно е файловете да са именувани, за да може процесите ги използват да знаят как да ги търсят ( $ls )

**Процес** – абстрактен обект, който обработва информация за изчислителните/ аритметичните операции; Процесът е нова абстракния, която описва, предоставя ни виртуално изчислително устройство. В ранните ОС не може да се говори за процес. В съвремените ОС (като изчислителна система) на логическо ниво работят едновремнео няколко процеса. Идеята е да работят максимално, успоредно като всеки процес работи и реализира конкретна дейност на ОС. Отделните приложения са обособени логически изчислителни процеси. Процесът е реален изчислителен процес, който обработва информацията в реалната среда, като той не съществува изолирано, самостоятелно, той си взаимодейства с други обекти в информационанта среда – с други процеси!!! и файлове.

- процесът е изолирано

**-** за всеки процес е определено “капсуловано” пространство и няма достъп до останалите процеси; по този начин процесите не си пречат на изпълнението;

На това ниво на абстракция можем да разглеждаме цялата ОС на логическо ниво като процеси (вкл. И файловете), някои от които си говорят помежду си **– централна абстракция.**

Можем да декомпозираме работата на ОС. Тя е голяма сложна програма от много редове код и да ги разделим задачите на части като разделим нещата на логически свързани единици. Друга полза от централната абстракция е, че се абстрахираме от хардуера. Постигаме го, като всяка ОС си има собстевна техническа дефиниция на процеса – целите са да имаме ефективност при работата на програмите, в UNIX процеса си има параметри. Ако системата е представена като рам, всеки процес си има определено място:

|  |  |
| --- | --- |
| **RАМ** | - процесът е изолирано |
| …. | - за всеки процес е определено “капсуловано” пространство и няма достъп до останалите процеси; по този начин процесите не си пречат на изпълнението; |
| Процес 2 |  |
| Процес 1 |  |

Тук възниква другата абстракция – **изолация,** когатодейности, ресурси и изчисления имаме възможност от ОС да ги изолираме и защитим едни от други. Това ни помага примерно да не си пречат процесите, като един се повреди да не навреди на останалите процеси като примерно краде чужда памет за изпълнението. Тук ОС, за да реализира тези неща представя на логическо ниво ресурса така: ако ОС има пълен достъп до ресурсите, отделните приложения са парчета, които ползват части то ресурса и имат пълен контрол само на своите части от ресурса, а ОС има пълен върху всички.

Когато процесите си комуникират, те излизат от локалното си пространство, като не им даваме право сами да го правят, а да минават през ОС , която е като главен собственик на тези **комуникационни канали.**

Повечето от удобствата, котио една ОС поддържа са удобства за програмистите. Програмистът иска абстрактното понятие процес да е вечно и като напише една програма, която се изпълнява от този процес, той да е вечен и усилието да напише програмата да е само веднъж. Той също иска тази програма да може да се пусне и след 10год. Езиците за програмиране са абстрактни, не се влияят от хардуера, това са езици от високо ниво (с++, python и др.). Веднъж написана програмта на такав език, тя може да бъде превърната в процес като част от ОС..

За С++ с помощта на компилатор превръщаме програмата в изпълним файл и вече може да се пусне да работи като процес. Идеята на ОС е да можем да поддържаме такава дълготрайност на програмисткия труд, защото хардуерите се променят често и бързо. **Основаната цел на процеса е да поддържа тази вечност.**

За **комуникационните канали**:

- ако представим процесите като отделни изчислителни инструменти, които комуникират чрез прости методи за връзка, ние като дефинираме процес и начина му на вазимодеиствие с останалите, ОС ще поеме действието им по реализацията. Примерно **при писане на програма и изискване от нас да отворим файл, казваме на ОС намери ми Х файл и ми го отвори и по този начин ОС ни предоставя файлов дескриптор, едно неотрицателно число (номера на комуникационния канал, с който взаимодействаме, уникален идентификатор на отворения файл), след което ние пращаме или получаваме байт след байт информацията по ком. Канал.**

Пример: при комуникация с някой сървър (интернет страница или чат приложение) връзката, която се осъществява м/у тях всъщност са два комуникационни канала в двете посоки – това се нарича **socket.**

- за да може да се поддържа илюзията, че всички процеси работят едновремено, като се делят на малки интервали техните изпълнения; за превключване на процесите, съществува хардуерен елемент (**таймер**, вътрешен часовник – прекъсване, което е част от интегралната схема), който преценя дали даден процес работи и трябва ли да се прекрати изпълнението му.; всички прекъсвания са асоциирани с някое периферно устройство, кеото иска да приеме информация или да изведе такава.

- за да може процесите да ползват само свои ресурси, това може да се реализира по два начина;

1) апаратно – ако искаме да предоставим на процесите да работят на машинен език, за да не пречат на останалите, трябва да имаме такав режим (защителен режим на работа – парчетата от паметта, които процеса може да достъпва) – трябва да се реализира хардуерно;

2) ОС изолира процесите; когато процесора владее всички процеси и при неадекватно изпълнение на някои от тях, ОС може да прекрати неговото изпълнение, да го унищожи.

- **програмни интерфейси –** те са част то ОС и са инструменти за удобство на програмистите и потребителите. Това са командни интерпретатори (текстови консоли, графични потребителски интерфейси и APIs), които са виртуали екрани, които преимат входните с и данни от клавиатурата.

**3. Разпределено използване на ресурси**

- мултиплексиране

- разделяне на паметта

- разделяне на времето

**Мултиплексиране -** метода, по който се реализира **файловата система** Друго нещо, което искаме от една ОС да прави е да можем да дефинираме права, което е различно от разделянето на файловата система. Това е удобство, защото за да се предпази информацията, когато много хора ползват една ОС, и това се постига чрез **правата** на собственика и възможността да ги модифицираме, като това е едно от основните удобства, които ОС ни предоставя. Най-важната абстракция, която се е дефинирала е, че изчислителанта система трябва да може да съхранява и обработва информация.

- **абстракция** – ключова дума, която ОС дава като удобсвто за постигане на реализацията – чрез **мултиплексиране** – разделяме ресурса на много нови пространства от байтове, които са ни удобни за обработка

- интернета също може да се даде като пример за ресурс- многожество от компютри, с които можем да комуникираме; може да се разглеждат като процеси, които съсществуват дълго

Тук възниква другата абстракция – **изолация,** когатодейности, ресурси и изчисления имаме възможност от ОС да ги изолираме и защитим едни от други. Това ни помага примерно да не си пречат процесите, като един се повреди да не навреди на останалите процеси като примерно краде чужда памет за изпълнението. Тук ОС, за да реализира тези неща представя на логическо ниво ресурса така: ако ОС има пълен достъп до ресурсите, отделните приложения са парчета, които ползват части то ресурса и имат пълен контрол само на своите части от ресурса, а ОС има пълен върху всички.

- Ком. Канали се осъществяват от ОС, защото тя има достъп до останали процеси и ресурси, които може да имаме необходимост от тях

- друг тип **мултипликсиране –** съвремените машини имат многоядрени процесори, и по този начин ОС предоставят възможността да се изпълняват няколко процеса едновремено като всеки работи на отделоно ядро. Ако имаме един процесор и искаме изпълненеито на 3 просеца, то ще делим времето за изпълнението.

Стратегии за управление на дисковата памет:

Едната възможност е това да се прави статично (предварително) при създаването на файла, а другата е да се прави динамично при нарастване на файла.

**Статично и непрекъснато разпределение:**

- при създаването на файл се отделят н-байта от дисковото пространство; за целта

- за тази цел обаче, системата трябва да има предварителна информация за максималния размер на файла, който той може да достигне по време на съществуването си; тази информация трябва да се осигури от потребителя при създаването;

- **(+)** този метод дава висока производителност на системата при последователна обработка на файла, тъй като последователните байтове на файла са разположени в съседни сектори, писти и цилиндри на диска

- **(-)** може да възникне проблем при нарастване на файла, ако той надвиши разпределеното му пространство; възможни решения:

- да се задели нова област с по-голяма памет и да се премести файла там – може тази операция да се окаже много бавна и скъпа при големи файлове;

- да се направи компромис при искането за непрекъснатост на дисковата памет, т.е. файл да може да заема няколко, но малко на брой непрекъснати области, заявени от потребителя

- **(-)** фрагментация на свободната памет – съществъване на достатъчно свободни участъци дисковата памет, които обаче не са съседни и поради това не може да бъде удоволетворена заявка за първично или вторично разпределение на памет за файл. Причината за този проблем е изискването за непрекъснатост и нефиксирания размер на единицата за разпределение на дискова памет.

- това се е променило с развиването на технологиите и създаването на преносими дискове и други устройства за преносима информация.

**Динамично и поблоково разпределение**

- файлът се дели на части с фиксирана дължина (блокове), като при записването на файл не е задължително тези блокове да са физически съседи.

- **(+)** дава възможност дисковта памет да се разпределя, когато е необходимо – при нарастване на файла → разрешава се проблема с нарастването на файловете и фрагментацията на свободната памет

**4. Видове ОС по начина на разделяне и изолация на ресурсите**

- еднозначна, еднопотребителска

- многозначна

- разлика м/у времеделене и многозадачност

- многопотребителска

**Еднозадачната** ОС изпълнява само една задача в даден момент. **Еднопотребителските** ОС не различават потребителите, но не са непременно еднозадачни, тъй като няколко програми могат да се изпълняват в съчетание. Например MS DOS е еднопотребителска еднозадачна ОС.

**Многозадачната** изпълнява няколко задачи едновременно. Това се постига чрез разделение на времето на работа на процесора според инструкциите на специална подсистема (task scheduling subsystem). при разпределена многозадачност (pre-emptive multitasking) ОС отпуска на задачата определено време да ползва процесора. Ако тя не успее да приключи за това време, ОС я форсира да отстъпи процесора на следващата задача, която се нуждае от него. Такива ОС са Unix-базираните като Solaris, Linux.

При кооперативната многозадачност приложението, стартирано от ОС, използва 100% от процесора. В този случай, ако друга програма изиска процесорно време, то или няма да ѝ бъде предоставено, което ще доведе до нарушаване на функциите на това приложение и/или до терминирането му, или ще бъде предоставено след приключване на първото – такива са 16-битовите версии на Microsoft Windows.

**Многопотребителските** ОС разширяват концепцията за многозадачност, като различават потребителите по отношение ползването на процеси и ресурси, като например дисково пространство. Те планират ефикасното използване на ресурсите на системата и могат да съдържат специализиран софтуер за изчисление на процесорното време от много потребители, както и да отчитат използваната памет, ползване на принтер и други използвани ресурси.

**Времеделене** – при наличие на един процесор е необходимо да се работи в режим на времеделене, който нарязва на малки интервали от време, в които да се изпълняват отделните процеси, превключвани чрез таймера.

„Разделението по време“ (time sharing) позволява създаването на многопотребителски системи, в които централният процесор и блокът на оперативната памет обслужват много потребители. При това част от задачите (като въвеждане или редактиране на данни) могат да се изпълняват в диалогов режим чрез терминали, а други (като обемните изчисления) – в пакетен режим.

**5. Комуникация и общо ползване на ресурси**

Процесите в системата са организирани един вид в опашка. Ако процесите работят на различни ядра, те няма да могат да се изпълняват синхронно, по реда на постъпване в опашката.

**Race condition (съревнование за ресурси)**– проблем, при който няколко процеса използват общ ресурс. По време на работата и работата на повече от един процес, инвариантите на структурата опашка трябва да се създаде така че максимално да се изключат колизиите и това да се получи загуба на данни и да се оптимизира този алгоритъм. Трябва да имаме инструменти, с които да предотваим тези неща.

Важно е да се защитят ресурсите, за да може при наличие на прекъсвания и колизии да не се загуби информация. По тази причина поредицата от инструкции ще се извърши атомарно и по време на работата на някой процес няма да се намесват други процеси в нейната памет.

Друго решение, което изисква хардуерни инструменти за защита на ресурсите е да защитим самата структура. Отделя се специален бит в паметта на опашката, който опоменава с 0- за заета структура и 1- за свободна. Когато един процес започва работа с тази структура достъпва този бит, за да знае дали може да я резервира и да работи с нея.

! Важно е при конкуренцията за ресурса процеса на заемане да става с **атомарна инструкция**

Друга инструкция е swap – подобна на атомарната. За някой от регирстрие да размени със съдържанието на х ( swap R, x;).

В съвремените машини за многопроцесорните системи тази реализация е изключително трудна.

Пример: Имаме 4-ядрена система в 2-процесорна машина. Всеки процесор има свой кеш, както и отделно самата изчислетелна система може да има вънншен кеш, и отделно има РАМ. В този пример “х” може да се намира навсякъде. Като искаме процеса да е атомарен трябва да се изисква от всички компоненит да се откажат от “х” и това може да стане изключително бавно.

Следователно независимо от процесорната машина е важно да се използват атомарни процеси за защита на ресурсите.

Хардуерни инструменти за защита (lock) на ресурса

**Enable/disable interrupt за еднопроцесорна система -** Самите процедури за обработка на структурата да им осигурим изпълнението като единна инструкция неделима във времето. Инстукрцията блокира временно прекъсванията. Когато се влезе в режим на работата в ядрото, кода извиква процедура от ядрото да го изпълни и може да се сложи в опашката инструкция да спре и да пусне прекъсванията.

При еднопроцесорните системи се постига взаимно изключване просто като се изключат прекъсванията в критичния участък

**test and set -** Тази инструкция се използва, за да се пише в паметта и да се върне старата стойност като атомарна операция. ЦП може да използва test-and-set инструкция предложена от друг компонент,напривем двупортор RAM; ЦП може също да предложи test-and-set инструкция.

**atomic swap -** Атомарната инструкция се използва да се постигне синхронизация. Тя сравнява съдържанието на локация в паметта с дадена стойност и само ако са същите,модифицира съдържанието с новата стойност. Това е направено като единична атомарна операция.

**Spinlock** е ключ, който кара нишката,която се опитва да го вземе, да изчаква в цикъл,проверявайки постоянно дали ключа е свободен. Тъй като нишката остава активна, но не извърша смислена задача,това се нарича активно чакане.

**6. Синхронизация от високо ниво – семафор**

- приспиване и събуждане на процесите

Когато работим с общи ресурси и имаме р1 и р2 процесите имаме пример за изчакване:

р1

block

…. някакъв код за обработка, който се нарича критична секция; участък от кода където се обработва в режим на изключване; хубав е за бърза работа и е предсказуемо времето за изпълнение и изчакване

Unblock

това е хардуерен механизъм за изакване/приспиване

докато

р2

block – но в цикъл докато чака; хубаво е да бъде приспан за да не гупи

…. процесорно време докато чака

този процес е известен като spinlock – механизъм от ниско ниво – с изчакване и приспиване (р1 и р2)

- за да работи надежно тези механизми са в ядрото на ОС и при нужда процеса ги извиква

- Съществуват и други механизми за приспиване на процесите, които са механизми от високо ниво;

- Най-ранния механизъм (сега го наричат от средно ниво, но реално си е от механизъм от високо ниво) – СЕМАФОР (Semaphores) -

**Семафор –** целта му е да пропуска само определения брой от процеси;

Инициализира се с една **целочислена** **променлива**. Има брояч, който като брои и стане отрицателен, нишките които се изпълняват се блокират и не могат да продължат изпълненеито си докато друга нишка не увеличи семафора – **чакат в опашка**. Когато нишка увеличи семафора и ако има друга, която чака, то една от нишките трябва да изчака другата да се отблокира

Семафорът може да се разглежда като обект в програмирането. Има три процеса:

init(count) / increment() - за инициализиране на самия семафора

wait() - приспива процес и го поставя в режим на чакане

signal() - когато някой процес приключва увеличава брояча с 1;

**Силен семафор –** реда на събуждане е акто този на приспиване, т.е. се реализира FIFO; реализация със СД – опашка

- ако е със стек може да възникнат проблеми, понеже някой процес може много дълго време да е приспан

**Слаб семафор –** ако не е определен ред на събуждане; (live lock, starvation)

**7. Прости задачи за синхронизация чрез семафори**

**Signaling**

Като пример ще използваме два процеса и един семафор

|  |  |
| --- | --- |
| Процес А | Процес Б |
| а1 | b1 |
| а2 | b2 |
| S.signal | S.wait() |
| a3 | b3 |
| a4.. | b4... |

A1 < b3, b4

a2 < b3, b4

С ред 3 се осъществява синхронизираното изпълненеи на отделните редове от процесите. Това е проста задача за подреждане във времето – пример за времева рамка. Процес А се задължава да изпълни част от кода и след това Б

\*За **среща във времето** (**renedezvous**) – използват се два семафора (sA , sB) – всеки процес разрешава на друг да работи

|  |  |
| --- | --- |
| Процес А | Процес Б |
| а1 | б1 |
| а2 | б2 |
| sA.signal – razreshava na B da raboti → | sB.signal |
| sA.wait | sB.wait |
| ...а3 | б3... |

a3,4 > b1, b2

b3,4 > a1, a2

- тук има значение реда на signal / wait →

**Mutex –** гарантира, че само една нишка ще достигне споделоената променлива

- семафора се инициализира с 1 – когато Mutex=1 пъврата нишка, която се е добрала до него ще изпълни секцията си, но твората <0 и ще се блокира, докато първата достигнала не изпрати сигнал

пример:

A → mutex.init(1)

mutex.wait()

count ++

mutex.signal()

B → mutex.wait()

count ++ → critical section

mutex.signal()

**8. Още задачи за синхронизация чрез семафори**

Реализация на комунукационна тръба (**pipe**)

sA.init(1)

sB.init(0)

|  |  |
| --- | --- |
| Процес А | Процес Б |
| а1 | б1 |
| а2 | б2 |
| sB.wait | sA.wait → B ще примене и ще се намира м/у ред 3 и 4; А ще бъде приспан |
| a3 | B3 |
| sA.signal → ще позволи на В отново да премине през ред 3 | sB.signal → ще разреши на А да се събути |
| ...а3 | б3... |

Тук ще се получи цикличност на процесите.

Тази конструкция на семафорите може да се използва за реализация на комуникационна тръба (**pipe**), като пример е светофара. Преминавайки през сА ще се смени и през сБ сусхто. Преминаването на процесите трябва да се осъществява редувайки посоката си на изпулнениет →/ ← . Това точно се постига с добавянето на изпълнението на а3 и б3 м/у процесите wait/signal.

-б3 ще се изпълни първи и след което на А ще се приспят. При събуждане на А ще се изпълни а3.

- **(+)** това позволява редуване на общите ресурси от процесите А и В; това е реализация на **потребител/производител**

**pipe с буфер –** за редуване при използването на общи ресусри; двата процеса проверяват дали тръбата е празна, за да “оставят данни”;

- това може да се окаже неефективно, защото при цикличното завъртане на процесите и взаимното изчакване, ако единия процес е по-бърз от другия, когато има пакет но няма къде да го остави;

- приспиването и събуждането на процесите е скъпа операция и увеличава системното време, защото е необходимо да се превключва оперативната памет и процесорното време;

За да сведем до минимум загубите от ядро и сситемната памет е чрез синхронизиращите механизми (семафорите), използва се **буфериране**. При входно-изходни процеси ОС да задели малко повече памет, за да не се налага постоянно да се приспиват процеси. Това се постига с осигуряването на повече маса (памет).

Пример:

Имаме стая с няколко маси (н-брой). За бързодействие и съображение за ефективност е да се заделя сравнително малка бройка.

**- има два семафора**

1) процесите, коит опишат ако има празни маси

free → init(n)

2) процесите, които ще четат

used → init(0)

\* за защита от rase condition трябва да има защити:

m\_w.init(1) – mutex\_write ; ще пуска мах 1 байт за писане

m\_r .init(1) – мах 1 процес чете

А – процесите, които пишат

В – процесите, които четат

- можем да пишем в тръбата, когато масите < n

- можем да четем , когато масите > 0

организация за писането на байт:

- намалява свободните и увеличава заетите

put(bit)

1. да провери има ли къде да пише като премине през бариерата – free.wait()

2. ако има друг процес, който пише трябва: m\_w.wait()

=> в стаята сме

3. ако масива от масите то : p[j] ← bit

4. j ++

5. напускаме стаята и позволяваме други процеси да влизат и да пишат, т.е. маса J е запазена/пълна и увеличаваме j и сочи към следващата

m\_w.signal() -

6. used.signal() - увеличава заетите маси; казва се че има още една пълна маса; защитаваме да не се ползва от следващия читате;

организация за четене на байтове (аналогичен/симетричен)

- намалява заетите и учеличава свободните

get(bit)

1. проверява дали има байтове в тръбата и чака поява на пълна маса → used.wait()

2. m\_r.wait()

3. b ← p[i] – чете от маса I

4. i++ → увеличава свободните

5. m\_r.signal()

6. free.signal()

**9. Проблеми при некоректна синхронизация**

При използването на семафори, един от възможните проблеми, които мойже да възникне е подредбата на кода да вкара процесите в мъртва хватка ( **deadlock**). Това е пример, когато два процеса взаимно се блокират и няма кой да ги събуди. Това ще ги приспи до края на изпълнението на изчислителния процес.

|  |  |
| --- | --- |
| Процес А | Процес Б |
| а1 | б1 |
| а2 | б2 |
| sB. wait | sA.wait |
| sA.signal | sB.signal |
| ...а3 | б3... |

Самото понятие от Дийкстра е дефинирано по друг начин (края на 60-те години):

**Гладуването -** Този проблем се появява когато на процес не му бъде доставен достъп до нужните ресурси, без които този процес никога няма да може да завърши своята работа. Обикновено гладът е причинен от прилагането на твърде опростен алгоритъм за съставяне на разписания по които процесите се изпълняват. Този алгоритъм, който е част от ядрото (kernel), би трябвало да предоставя еднакво ресурси за всеки един от процесите, тъй че нито един от процесите в разписанието или разписанията да не остава без нужните ресурси.

**Livelock** е подобно на deadlock,обаче с разликата че състоянията на процесите,които участват в livelock-а постоянно се променят 1 с друг, като никой не прогресира.

Типочен пример е с 5 философа на кръглата маса, които ядат спагети, но имат сервирани само по един комплект прибори (отдясно). Ако секи посегне по едно и също време да яде и първо кън дясната вилица, то когато посегне към лявата тя вече ще е взета то съседа. Това ще достигне до липса на ресурси (нямат достъп до общ споделен ресурс), за да извършат процеса си.

Решение:

1. като видиш, че единия ресурс е зает, остави този който имаш и изчакай да се освободи заетия (за някакво определоено време- доакто мисли); когато посегне след известно време и двата ресурса ще са свободни и ще ги достъпи. Полуава се верига и се въртят почиващи и ядящи, като 1 и 3 ядат, 2 и 5 ядат, но 4 може винаги да гладува.

2. когато общите ресурси се номерират (вилиците); когато някой взима ресурс, първо се взима с по-малък номер, но тогава един ще има избор от 1 и 5 , като 1 ще е вече заето. Т.е. поне един от философите пак ще е в deadlock

**10. Процеси и тяхното упражление**

Видове процеси:

А – отнема **максимално време на ЦП**; да оптимизира предоставеното му време – има много изчисления

В – интерактивни; пъргави процеси – пр. Чат системите и всички приложения със зареждане на картинки и т.н. - идеята е да **минимизира празвните интервали** и процесорното време

С – **real time** – deadline (има краен срок на време за извършване на процеса) – процес, който го ползваме за управление на много важни неща – космически ракети, военни приложения, атомни централи, спирачки на автомобил и т.н.; да се намалят критичните точки; **управляване на бързопротичащи реални процеси чрез датчици;** не е важно палзите да са къски, изисква сама да си раздеял време на интервалите

Видове ОС според А, В, С -дали ги осигуряват:

- тези ОС, които използваме в ежедневието осигуряват А и В (смартфони, сървъри, лаптопи); те са **time sharing,** но не гарантират реалтайм

- С се използват в ракети, коли (промишлени контейнери)

1 – един процес (R ) ще бъде приспан, когато чака входно-изходна операция или чака време; wait() =0

2 – когато R работил много време и отнема процесорно време, той преминава в чакащ; условия за преминаване –когато много време е бил работещ (понеже всеки процес има квант време за работа и прекъсващия часовник го прехвърля в чакащ)

3 – обратно на 2) – когато някой чакащ процес за процесорно време се слага активен

4 – обратно на 1) – събуждане на приспан процес- пр. Signal(); когато процесора свърши работата си с ресурсите и позволи на други процеси да ги използват

\*тези стъпки се правят от ядрото

**Task scheduler –** това, което решава как да се превключва процеса (от чакащ в работещ) – осъществява се чрез хардуерния елемент **time interupt –** разделянето на времевите процеси

**11. Състояния на процеса и преходи между тях**

Най-краткото определение за **процес е програма в хода на нейното изпълнение**. Следователно, има връзка между понятията процес и програма, но те не са идентични. За разлика от програмата, която е нещо статично - файл записан на диска и съдържащ изпълним код, процесът е дейност.

**R** – работещ (заема процесор) – всички процеси, които НЕ чакат входно-изходна операция да настъпи, а си имат работа

**r** - активен (чака за процесорно време) run\_queue – нуждаят се от изчислителанта мощ

**S** – спящ (чака събитие, signal() от друг процес или драйвер) – **sleeping –** процеси, които работят, но чакат входно-изходна операция или да завърши операция и да дойде момент да ги събуди; нямат нужда от процесорно време; нямат нужда от изчислителен ресурс

**T** – чака настъпването на момент във времето, time\_queue; Процесът е зареден в главната памет и изчаква изпълнение върху ЦП. Модерните компютри имат възможността да подкарат множество различни програми или процеси по 1 и също време. Обаче ЦП може да обработва само по 1 процес. Процеси, които са готови за обработка са държани в опашка.

Диаграма на преходите:

**12. Управление на паметта**

- докато процес работи, използва само блок паммета за него (л**окалната си памет,** която му е отредена); за да го защитим трябва да има харуерен регистър, където е оказано в какъв диапазон от паметта може да се достъпва

- друг механизъм е ако този диапазон е sizе ор 0 до size

- в двата механизма е нужно хардуерния елемент да прави изчисление дали е в правилните рамки на паметта

**Йерархия на паметите- кеш, RAM, swap**

Паметта е разглеждаме като абстракция с реалността(паметта като единен блок, поредица от байтове) .При бърза работа с паметта-скъпо в енергиен план; те са енергозависими . Бързите памети - по нетрайни .

При междинният клас памети(динамични RAM чипове) – те са евтини, относително бързи и събират много информация, но ако не биват често обновявани, паметта се губи, защото се губи сигнала поради факта,че много малко електроника се влага в рам (енергозаивисми). Има различни видове памети в хардуера: Качества на паметите(енергозависимост, скорост на работа, брой презаписи)

• **Кеш**-скъпи, но бързи; за пресмятане да подходящи; поддържа данните на паметта; кеш е спомагателна памет за ускоряване обмена на данни между различните нива в йерархията на паметта. Ускоряването се постига чрез поддържане на копия от избрани части от данните върху носител с бързо действие, близко до това на горното ниво на паметта. Може да постигне различни степени на ускорение в зависимост от вида на обменяните данни, от алгоритъма за избор на данните за копиране и от съвместимостта помежду им. За да са по ефективни и по-ефикасни в употребата на данни, кешовете са сравнително малки

• **RAM**- по-голямата част от процеса е тук; това е паметта, която процесора директно ползва. В нея се разполагат стека, кода на програмата, данните(така бързо става четенето и писането на байтове)

• **Хард диска**-вечен, но те са механични устройства => тежък, хабят енергия; Флаш паметта-има деструктивност на паметта;броят презаписи е ограничен

- **Swap**- когато части от процеса са вкарани в по-бавни хардуерни ресурси; Това се нарича swapping-временно съхранение на рядко използваните процеси. Swap-памет върху диска, която е част от Ram-a

Кеш на централния процесор (CPU кеш), е кеш използвана от централният процесор (CPU) на компютъра, за да ускори времето за достъп до информация на RAM паметта. Повечето коммпютърни процесори имат няколко независими една от друга кеш памети.

Всички инструкции и адреси за запазване, които се ползват от процесора трябва да идват от RAM. Въпреки че RAM е много бърза, все пак значително време е нужно за ЦП да я достъпи. RAM паметта е много по-голяма от кеш паметта. Колкото е по-бърза паметта,толкова по-скъпа става тя.

Swap-ването е механизъм, чрез който процес може да бъде преместен временно от главната памет в резервен компонент(хард диск), и после да бъде върнат пак в паметта за да си продължи изпълнението.

Резервният компонент попринцип е хард диск, който има бърз достъп и е достатъчно голям, за да си набави копия на всички изображения на паметта за потребителите.

**Виртуална памет на процеса - функционално разделяне**

Виртуална памет — системна памет симулирана от операционната система и разположена на твърдия диск. Тя позволява да се прилага едно и също, непрекъснато адресиране на физически различни памети (участъци от твърдия диск). Например, при Windows ако няма достатъчно оперативна памет за изпълнение на програма или операция, операционната система използва виртуална памет. Тя се разделя на различни статични парчета:

o Статична част (read only) от кода-частта която го описва, тя се изчислява колко е дълга при създаване на процеса, използва се и от други процеси(споделена област-така се пести памет):

- Има блок с фиксирани размери, където се разполага кода на програмата, памет, която няма да се променя,записана е самата програма, изчислява се размера предварително при стартирането;

- Има парчета за споделени библиотеки, които са достъпни едновременно за няколко процеса и се предоставят на процеса;

o Динамична част(read-write памет)

- Стекове-намира към края на адресното пространство; Нужен на процеса е за управление на програмата, в него съществува реализацията на структурите за управление на програмата предоставени от езика за програмиране(функции, променливи, подпрограми). Стекът е проста структура, заема обем от паметта, който може да нараства и да намалява.

- Heap-памет, която се използва от процеса, ако е необходимо да се задели допълнително обеми памет, които не са в локалните променливи(използва се при нужда и тя варира от процес до процес, блок с неопределена дължина)

- Блок за статична памет-променливи, които при стартиране на процеса се заделя винаги тази памет; място за данни-обем на масиви, константи и др.

Структурата на виртуалната памет може да се разгледа по следния начин:

**Данни** - Глобални променливи, константи, статични променливи от програмата

**Kernel** - Докато това е част от адресното пространство, то не може да се адресира директно от някой процес. Трябва да се адресира чрез системни извиквания.

**Стек** - Използва се от програмата за променливи и запазване. Расте и и си смалява размера в зависимост от това какви практики са извикани и какво са техните изисквания за размера на стека (8MB в размер).

**Heap** - Използва се за някои видове работни хранилища.

**Споделени библиотеки** - Споделените библиотеки са позиционно независими и така те могат да бъдат споделени от всички програми, които искат да ги използват.

**13. Виртуална памет – реализацията**

Таблици за съответствието на виртуална/реална памет

Хардуерът, който отговаря за транслацията между логически и физически номер на страницата и въобще за управлението на адресацията, се нарича **MMU** (Memory Management Unit). Инструмента, който използваме за оптимизиране на работата на MMU, се нарича **TLB** (Translation Lookaside Buffer). В зависимост от схемата за транслация, TLB може да се реализира по различни начини За конкретния случай.

Ефективна обработка на адресацията - MMU, TLB

**MMU**(Memory management unit) - е хардуер, през който всички референции на паметта преминават, като главно извървшват транслацията от виртуален адрес в физически адрес.

Модерните MMU попринцип разделят виртуалното адресово пространство в страници, всяка от които има размер,който е степен на 2,обикновено няколко килобайта,но може и да са по-големи. Най-долните битове на адреса остават непроменени. Горните битове на адреса са числата на виртуалната страница.

**TLB** (Translation lookaside buffer) е кеш който хардуера за управление на паметта използва, за да подобри скоростта при транслация на виртуални адреси.TLB има фиксиран брой слотове,съдържащи записи от таблицата със страници и от таблицата със сегменти;

Записите от таблицата със страници се използват за преобразуване на виртуалните адреси в физически адреси, докато записите от таблица със сегменти преобразува виртуалните адреси в сегментни адреси.

**14. Синхронен и асинхронен вход-изход**

При синхронна входно-изходна операция системното извикване може да доведе до приспиване (блокиране) на потребителския процес, поръчал операцията. Същевременно, при нормално завършване, потребителският процес разчита на конкретно комплектоване на операцията – четене/запис на всички предоставени/поръчани данни във/от входно-изходния канал, или цялостно изпълнение на друг вид операциа (примерно изграждане на TCP връзка).

При асинхронна входно-изходна операция системното извеждане не приспива (не блокира) потребителия процес, поръчал операцията. Същевременно, при невъзможност да се комплектова операцията, ядрото въща управлението на процеса със специфичен код на грешка и друга информация, която служи за определяне на степента на завършеност на операцията.. Потребителския процес трябва да анализира ситуацията и при нужда да направи ново системно повикване по-късно, с цел да довърши операцията.

Използването на асинхронни операции позволява на един процес да извършва паралелна комуникация по няколко канала с различни устройства или процеси, без да бъде блокиран в случай на липса на входни данни, препълване на буфера за изходни данни или друга ситуация, водеща до блокиране.

Типични примери:

1. Когато ползваме WEB-browser, той трябва да регулира на входни данни от клавиатурата и мишката, както и на данни, постъпващи от интернет, т.е. на поне входни канала. Браузърът проверява чрез асинхронни опити за четене по кой от каналите може да обслужва информация и реагира адекватно

2. Сървър в интернет може да обслужва много на брой клиентски програми, като поддържа отворени ТСР връзки към всяка от тях. За да обслужва паралелно клиентите, сървърът трябва да ползва асинхронни операции, за да следи по кои връзки протича информация и кои са пасивни.

Когато програмата ползва асинхронни операции и никой от входно-изходните канали не е готов за обмен на данни, тя има нужда от специален механизъм за предоставяне на изчислителния ресурс на останалите процеси. Обикновено в такива случаи програматасе приспива сама за кратък период от време (в UNIX това става с извикването на sleep(), usleep() или nanosleep().

**Теми, специфични за Linux/UNIX**

**15. Текстова консола, shell**

Свързане и допускане до UNIX система- login

The login program is used to establish a new session with the system. It is normally invoked automatically by responding to the "login:" prompt on the user's terminal. login may be special to the shell and may not be invoked as a sub-process. When called from a shell, login should be executed as exec login which will cause the user to exit from the current shell (and thus will prevent the new logged in user to return to the session of the caller). Attempting to execute login from any shell but the login shell will produce an error message.

Конзола - стандартен вход, стандартен изход, стандартна грешка

Стандартните потоци са вход и изход комуникационните канали между програма и нейното обкръжение, когато започне да се изпълнява. Трите I/O връзки се наричат стандартен вход(stdin), стандартен изход(stdout) и стандартна грешка(stderr). Когато команда е изпълнена през интерактивен shell, потоците са свързани към текстовия терминал, на който работи shell-a, но може да се промени чрез пренасочване(pipeline). По общо казано, child process ще наследи стандартните потоци на родителя си.

Shell - команден интерпретатор

Когато включим конзолата (след като сме се идентифицирали), стартира програма shell. Тя има много различни реализации (bash, zsh, tcsh, sh). Редът преди курсора се нарича prompt и ни казва в какъв режим работим (обикновен потребител, root)

Изпълнение на команди, параметри на команди

Най-общо има два вида команди - команди-филтри и команди, които показват състоянието на системата.

Основни команди:

**df** - (disc free) файловите системи

**ps** - показва текущите процеси

**ps -ef** - показва всички процеси

**wc -l** - брой редове

**cut** - вади дадени колони от изхода

**uname** - unix name

**16. Shell – конвейри, пренасочване, филтри**

Програмта при стартиране има три отворени файла – stdiн(0) stdout(1) stderr(2). При стартиране на shell се чете от клавиатурата. В самия език всеки такъв отворен файл се начира файлов дескриптор, представен чрез число от 0 до ? .

**|** - конвейр, данните като изход от 1 се пренасочват като вход към 2

**<** - пренасочване на входните параметри от стандартния вход stdin;

**>** - пренасочване на изходните параметри stdout; манипулатори, като можем да записваме във файл

**2>** - пренасочване на изхода от грешки и можем да ги записваме във файл

**>>** - добавя; може да се манипулира и да добавяме във файл

**&** - за паралелна работа на р1& със shell (background process); докато се изпълнява този процес можем да стартираме и други от shell

**;** - разделител, използва се последователно за приключване на р1, като не се стартира р2 (р1; р2)

филтри – командни, които обработват данните без да променят първоначалните

**cat –** принтира на конзолата съдържанието на подадения файл като аргумент

**wc** – извежда информация за броя на редовете (-l), байтовете (-c), думите (-w) и символите (-m)

**grep –** реже по редове

**сut –** реже по колони

**sort –** сортира числово (-n) и лексикално (-d)

**tr \_ \_ -** заменя значи, може да трие конкретен символ или да сплескта повторенията

**17. Shell – програми (скриптове)**

**echo -** (-n - no newline, -е - указва на команда echo да разбира от специални символи като \t и \n)

**read -** четене на потребителски вход

**test** - всички условия в if ()

-числово сравняване(-eq-equal, -ne-not equal, -gt-greater than, -ge-greater than or equal, -lt-less than, -le-less than or equal)

сравняване на низове(=, !=, >, >=, <, <=)

-логически сравнения(-o-or, a-and, &&, ||)

-f-дали даден файл съществува и е обикновен, -d-дали съществува и е директория, -е-дали файл съществува, без значение какъв е, -z,-n - дали даден файл/променлива е с празно/непразно съдържание, -r,-w,-x - дали файл има права за четене/писане/изпълнение

**if -**  if [ ... ] then ... elif [ ... ] then ... else ... fi

**for** - for ... in ... do ... done

**while -** while [ ... ] do ... done

**18. VI, tar, g++**

Vi – има два режима на работа – insertion mode/ command mode

**tar** - tape archive; команда за правене на архиви

Най-ранната функционалност на tar е просто да обедини няколко файла в един, по-голям файл, без да ги компресира по какъвто и да било начин. Това става като направи каталог, в който пише кой файл как се казва и колко е голям, а след него следват самите файлове.

tar [опции] [списък от имена на файлове]

tar cf archive.tar file1 file2 - създава архив с име archive.tar, който съдържа file1 и file2

Има два вида компресиране на данни - със и без загуба на информация.

z - като част от опциите указва на tar да компресира данните, не просто да ги обедини в по-голям файл

c - create, създаване на архив

t - показва съдържанието на архива

x - extract, разархивира

**19. Програми за разглеждане и търсене**

**ls** (list directory contents) имената на файловете и директориите в текущата директория (-l-допълнителна информация, -a-показва и скритите файлове, започващи с ".", -dl-показва само директорията без съдържанието й)

**who** - текущо влезлите потребители в системата

**find** - търси файлове в йерархията на директориите

**ps** -( process state) - текущите активни процеси(-ef-списък от всички процеси)

ps -aux - абсолютно всички процеси

**top** - показва изглед на стартираните процеси в реално време. Може да сортира процесите по CPU usage, memory usage и runtime

-p - показва само процесите с дадено process id

**20. Файлова система**

В UNIX файловата система е йерархична, дървовидна. Съставена е от многожество от директории. Корена на йерархията е “/” (home).

Важно е да знаем как тази структура е разположена в/у хардуера. В UNIX може да има няколко диска в системата (partitions) и всяко парче такав диск е линеен диск от байтове (последователност от блокове). Тази абстрактна представа е осигурена от по-долния слой на входно-изходната подсистема на ядрото (драйверите)

**Приета е следната конвенция:** всяко парче или всеки такъв логически диск представлява поддърво на файловата система.

**Файл** – основната единица, чрез която операционните системи осигуряват съхраняването на постоянни обекти данни. Всеки файл има име, което е низ от символи и се състои от две части – първата е името а втората е разширенеито на файла, което характеризира каква е информацията в него.

Типове:

* обикновен файл (regular file) - файл служещ за съхраняване на информация върху външен носител
* специален файл (character special device file, block special device file) - един или повече файлове служещи като абстракция за външно устройство
* каталог (справочник, directory) - файл с определен формат съдържащ информация за част или всички файлове в системата, позволяващ да се осъществи връзката межу името на файла и данните му ( каталогът може и да не е файл)
* символна връзка (symbolic link) - файл, чийто данни представляват името на друг файл и служат като указател към него.
* механизми за комуникация – програмен канал (pipe), FIFO файл - услуга предоставяна от ОС позволяваща паралелно работещи процеси да си комуникират

Файловете имат атрибути – размер, дата и време на създаване, достъпране и променяне, собственик, права за достъп, пароли, флагове.

**Директория (каталог)** – структура в ОС, чрез която можем да разделим отделните файлове, да ги обособим по някакви критерии.; Всяка директория може да съдържа информация както за файлове така и за други директории. По този начин се реализира дървоводна структура, която позволява доброто структуриране при разположението на файловете. Корен на дървото е главната директория (root).

Вследствие от въвеждането на дървовидната структура всеки файл притежава освен обикновеното си име (относително име) и “пълно име”(пътя до файла, абсолютно име)

**21. Многопотребителска работа**

Многопотребителските ОС разширяват концепцията за многозадачност, като различават потребителите по отношение ползването на процеси и ресурси, като например дисково пространство. Те планират ефикасното използване на ресурсите на системата и могат да съдържат специализиран софтуер за изчисление на процесорното време от много потребители, както и да отчитат използваната памет, ползване на принтер и други използвани ресурси.

Права и роли в UNIX

Всеки потребител има user ID (uid). За всеки файл в системата е дефинирано кой е собственикът (кой user). Има и групи от user-и. За всеки файл е казано: собственикът, членовете на неговата група и останалите хора какво могат да правят с него.

ls -l - първите символи указват правата за достъп до него; 1 - вид на файла; 2, 3 и 4 - права на собственика; 5, 6 и 7 - права на групата; 8, 9 и 10 - права на всички останали; Тройките представляват 3 отделни бита. Първият може да е - или r, вторият - да е - или w, третият - да е - или x. Отговарят съответно за четене, писане и изпълняване на файл. Правата за достъп могат да се променят от собственика на файла или от супер потребителят. Това става чрез chmod.

Права - u/g/o user/group/other

Роли r/w/x read/write/execute

chmod - (change file mode bits) - добавяне/премахване на права

**22. Физически файлови системи**

Блочни и символни устройства

Блочни специални файлове или **блочни устройства** предлагат буфериран достъп до хардуерни устройства и предлагат някаква абстрацкия от тяхната специфика. За разлика от символните устройства,блочните устройства винаги ще позволят на програмиста да чете или пише блок от всякакъв размер и подравняване. Отрицателното на това е,че тъй като блочните устройства са буферирани, програмистът не знае колко време ще отнеме записаните данни да се прехвърлят от кернела в истинското устройство или в какъв ред 2 отделни записа ще пристигнат във физическото устройство.

??

**Символно устройство** е такова устойство с което диска комуникира чрез изпращане и получаване на информация байт по байт. Пример са звуковите карти, USB портовете , parallel ports

Блочното устройство комуника като изпраща цял блок от данни. Пример са твърдите дискове, USB камерите,

**Монтиране** – закачането на физическите дискове като поддървета (пример е когато включим флашка)

**etc/fstab** - системен конфигурационен файл. Този файл изкарва списък с всички налични дискови партитишиони и други типове на файлове системи,които не трябва задължително да са дисково-базирани, и показва как трябва да се инициализират или интегрират в по-голямата файлова-системна структура.

**etc/mtab** - системен конфигурационен файл. Този файл изкарва списък с всички текущо монтирани файлови системи заедно с техните инициализиращи опции. Mtab прилича на fstab, разликата между двете е,че /dev/fstab изкарва списък с кои налични файлови системи трябва да се монтират по време на буутването,докато първото показва кои са монтирани в момента.

**mount** - инструктира операционната система,че файловата система е готова за използване и я асоциацира с отделна точка в общата йерархия на файловата система(mount point) и задава опции свързани с нейния достъп.

**Umount** - инструктира операционната система,че файловата система трябва да бъде деасоциацирана от точката и за маунтване, правейки я вече недостъпна и може да бъде изтрита от компютъра. Важно е първо да се umount-не устройство преди да се изтрие.

**23. Физически файлови системи – реализация**

Ефективна реализация, отлагане на записа , алгоритъм на асансьора

Има специални програми (check and repair), които се стартират когато е ясно, че има някакви нарушения в структурата на файловата система, и които проверяват съответствието между метаданните и данните на целия диск. Когато системата има много процеси и едновременно се работи с много файлове, може да се окаже, че тези програми не могат да оправят нещата.

Всички съвременни файлови системи са журнални (Journal File System). Концепцията за журналност е въведена първо при базите данни - за транзакциите, които са одобрени и които трябва да се запишат в дисковите пространства, за да бъдат съхранени дълготрайно. За да не се загубят при авария, освен нормалния файл, в който са записани таблиците на базите данни, се съхранява и друг файл, който се нарича журнал (log file). Той е един за цялата база и всяка транзакция се записва първо в журнала. Извърши ли се дадена транзакция изцяло, тя се изтрива от журнала. Тази концепция се използва и при файловите системи - някакъв специален файл (или част от диска) се обявява за журнал и отложените операции се записват в правилния ред в журнала. Докато присъстват там се опитваме да ги запишем в реалните места на диска. Ако успеем, цялата поредица от действия, които са свързани с реалната промяна по даден файл, се изтриват от журнала. В различните журнали се описват различни типове данни.

Алгоритъм на асансьора

Алгоритъмът, който се използва най-често, се нарича алгоритъм на асансьора и предполага, че близки сектори на диска ще се променят бързо. При него имаме указател, който сочи къде се намира главата на твърдия диск в момента, помни се и посоката, в която се движи тя. Заявките се обработват по посока на движение на главата подобно на движението на асансьор - слиза надолу докато изпълни всички заявки, след това обръща посоката и се движи нагоре докато изпълни всичко. Недостатъкът на този алгоритъм е, че може да доведе до голямо забавяне на заявките. Друг алгоритъм е алгоритъмът на най-близкия сектор, който оправя недостатъка на алгоритъма на асансьора, но при него може да се получи starvation.

**24. Специални файлове**

**/dev** -съдържа драйверите на системата; не съдържа файлове, в нея се описват устройствата, които са част от изчислителната система; псевдо файлове, които задават какъв хардуер може да бъде обслужван от нашата система и какъв точно се обслужва

**mknod** - системно извикания mknod() създава файлово-системен възел(файл,специален файл или именуван pipe) именуван pathname, с атрибути специфицирани от mode и dev.

Линкове - твърди и символни, команда ln

**Символният линк** е прякора на всеки файл, който съдържа референция към друг файл или директория в формата на абсолютен или относителен път.

Твърдият линк е запис на директорията, който асоциацира име с файл във файловата система. Всички директории-базирани файлове системи трябва да имат поне 1 твърд линк давайки оригиналното име за всеки файл.

ln [OPTION]... TARGET [LINK\_NAME] - създава линк към специфицирания TARGET с допълнителната настройка LINK\_NAME. Ако LINK\_NAME е пропуснат, линкът се създава със същото базово име като TARGET-а в текущата директория.

**socket** - двупосочен комуникационен канал; socket listen и socket connect; тези връзки могат да се създават своевременно и процесите от двете страни не е необходимо да са наследници на общ родител

**25. Файлова система**

**/bin** – бинарi тук се намират важни програмите, които са необходими за стартирането на ОС; главния файл; програми, които са общодостъпни за всички потребители (ps, ls...), не са системни програмите тук

**/boot** – неща, необходими за зареждането на ОС преди тя да влезе в нормалнот осъстояние; примерно ако имаме няколко ОС на нашата машина, нещата от boot се зареждат за да може да се стартира самата ОС (пример е отварянето на екрана за избиране на ОС), програмите тук управляват процеса за зареждане на ОС

**/dev** – не се съдържат файлове; тук се описват самите устройства, които са част изчислителната система; тук се намират псевдо файлове, които имат уникална структура и те задават какъв хардуер могат да обслужват и какаъв реално обслужват; тук има логически изображения, които предоставят драйверите на устройствата

**/proc** – подобна na dev; съдържа статистическа информация за ядрото – кои процеси са активни, връзките м/у отделните процеси, връзка с интернет; псевдофайлове

**/etc** – специална директория, която съдържа файлове, които управляват конфигурацията на конкретната изчислителна система (uname → debian, ubuntu,; каква конкретна е ОС, какви са и параметрите, какви са и характеристиките и т.н.) ; информация за потребителите как се логва, какви са им правата, коя е хоме директорията; структурата на файловете тук е важна за системните администратори; тук е записано и кои дискове се монтират при първо началното зареждане, като можем и ние да добавяме такива дискове за монтиране;

- връзка с /boot – в нея се задават нещата преди стартирането, докато в /etc са нещата след самото зареждане на ядрото, как ще се конструира файловата сситема и т.н.

**/home** – данните за всеки потребител си има своя директория

**/lib** – само по време на стартиране на системата; служебна директория, ако се повреди някоя файлова система, при възстановяването на файловете се салгат тук

**/media** – точка за монтиране на устройства, които не са постоянни – USB, CD, DVD,

**/root** – директорията на суперюсер; ако нещо се повреди при стартирането трябва да се създаде достъп до администратора

**/sbin** – при зареждането на системни файлове, само root може да ги използва

**/usr** – за програмите и други неща свързани с тях, които са достъпни за пълноценната работа на системата; обикновено /usr и /home са на различни физически дискове, за да не стават колизии.; /usr/bin - всички програми, които са вече изпълними процеси

**/var** – съдържа споделени данни, примерно ако използваме база данни тя ще се разположи тук; не е достъпна за обикновените потребители

**26. API, POSIX – работа с файлове**

**оpen()**

**close()**

**read()**

**write()**

**lseek ()**- lseek(fd1,int offset,SEEK\_SET); SEEK\_SET премества указателя fd1 с оffset; SEEK\_END

The offset is set to the size of the file plus offset bytes.

SEEK\_CUR

The offset is set to its current location plus offset bytes.

**scanf()** - int scanf(const char \*format, ...);

The scanf() function reads input from the standard input stream stdin,

**fgets()** -char \*fgets(char \*s, int size, FILE \*stream);

ako imame da chetem string ove;

reads in at most one less than size characters from stream and

stores them into the buffer pointed to by s. Reading stops after an

EOF or a newline. If a newline is read, it is stored into the buffer.

A terminating null byte ('\0') is stored after the last character in

the buffer.

**27. API, POSIX – работа с процеси и тръби**

**pipe(int pipefd[2])**

creates a pipe, a unidirectional data channel that can be used

for interprocess communication. The array pipefd is used to return two

file descriptors referring to the ends of the pipe. pipefd[0] refers

to the read end of the pipe. pipefd[1] refers to the write end of the

pipe. Data written to the write end of the pipe is buffered by the

kernel until it is read from the read end of the pipe.

**dup2(oldfd, newfd)**- makes newfd be the copy of oldfd, closing newfd first if necessary

**dup()** - poluchavame kopie na pyrviq svoboden failove deskriptor

**fork() -** Създава нов процес(дете)

Ново- създаденият процес е почти идентичен на бащиния (този, който го е създал). Защо почти? ами различават се например по PID (process ID, всеки процес има уникален PID)

Тъй като са почти идентични процесите, трябва някак да ги различаваме.

Различаваме ги по това какво връща fork().

fork() връща 0 при детето и > 0 при бащата.

**exec()**

int execl(const char \*path, const char \*arg, ..., 0);

- първият аргумент е пълно име на бинарен файл (команда), който искаме да изпълним

- вторият аргумент е името на самата команда

- Следват N на брой аргументи - опции към командата, която искаме да изпълним

- Последният аргумент е 0, указва край на въвеждане на аргументи.

Пример:

искамe да изпълним команда ls, с опция -l

execl("/bin/ls", "ls", "-l", 0);

**\*\***Особености (които трябва да знаем):

При изпълнение на системния примитив execl се сменя образа на процеса, който я е изпълнил. т.е. изпълни ли се execl -> сменя се текущата програма с програмата ls и връщане назад, няма.

Oще примери - най-долу директно в задача.

**\*\***Особености (които не е нужно да знаем):

- Първият аргумент на функцията е бинарния файл, който искаме да изпълним.

- Вторият аргумент е какво ще бъде името на стартирания процес, което може да видим с команда ps.

**wait()**

wait(status);

Тъй като процесите се изпълняват на конкурентен принцип, бащата може да завърши изпълнението си преди детето. Но ако искаме изхода от детето да го ползваме в бащата, то трябва да укажем на бащата да изчака детето, да завърши.

Това става чрез системен примитив wait(). Той следи за промяна на статуса на детето:

Например настъпване на терминиране на процеса или на пауза или на събуждане от пауза.

waitpid()

**28. API, POSIX – сокети**

**socket()**

**bin()**

**connect()**

**listen ()**

**accept()**