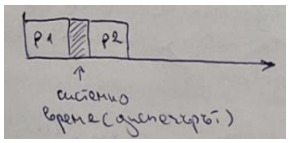
Тема 1

1. Опишете как ОС разделят ресурсите на изчислителната система, дайте примери за основните типове разделяне:
   Разделяне на пространството (памети).
   - Основна функционалност на ОС е да разпределя паметта. Тя трябва да може динамично да разпределя наличната физ. Памет между процесите, за да организира ефективно изчисл. процеси.
   - При разделяне на пространството от значение са не само **активните** процеси, но и **приспаните**. Ресурсът се разделя на части и всяка програма или потребител получава част от него. Проблемите, които ОС трябва да решава са следните: Да следи **свободните** и **заети** части, да осигури **защита** на частите, **справедливо** да разделя ресурса
   - Има **два вида** разделяне на паметта - статично и динамично;
     a. при **статичното** трябва да се знае каква е максималната големина на всеки файл.
     b. При **динамичното**, файлът се разделя на т.нар. **фрагменти**, като тези фрагменти се разхвърлят по паметта (процесът се нарича фрагментация); по този начин не се пази максимална големина на файла, и **нарастването** му няма да доведе до проблеми.
   - Разделяне на времето (процесори, други у-ва).
   - Отговаря за разпр на времето – **диспечер/Task scheduler**. предоставянето на процесорно време да бъде съобразено с **типа** на процесите. Подход за справяне с тази ситуация е употребата на **приоритети** на процесите – на всеки процес присвояваме приоритет 1000 и при всяко изпълнение на процеса, от този приоритет се **изважда** времето, за което е работил, и след него процесорно време се дава на процеса с **най-висок приоритет**. Процесите могат да бъдат и разпределени на база асоциация с даден процесор, на който вече са работили. В по-сложни съвременни системи се използват допълнителни виртуални процесори, като един хардуер се разпределя върху няколко резервни виртуални машини, които при натоварване се включват в употреба.
     - Процеси:
       - foreground/IO процеси – висок приеритет. често приспивани и бързо си изпълняват задачата
       - background/CPU процеси – нисък приоритет. много процесорно време, но няма такова значение кога ще им бъде предоставено
       - real-time процеси - имат определен срок (deadline), в който трябва да им се предостави процесорно време. Използват се комп с достатъчно процесори



2. Опишете с по едно-две изречения работата на следните системни извиквания в стандарта POSIX: pipe() dup2() fork() exec() wait() waitpid()

Тема 2

1. Опишете разликата между времеделене и многозадачност.
   Какви ресурси разделя еднозадачна, еднопотребителска ОС?

- **Времеделене** – метод за **организиране на работата** на ОС, при който **множество потребители или процеси споделят един и същ процесор**, като всеки получава малък интервал от време за изпълнение. **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. Това бързо превключване създава **илюзията**, че процесите/потребителите работят **едновременно**. Така една машина може да обслужва много потребители, въпреки че реално изпълнява само **една задача в даден момент.** Това е основата на **многопотребителските системи**, като например сървърите с терминален достъп. **Времеделене** = техника за споделяне на процесора между **много потребители** (**многопотребителски фокус**).

- **Многозадачната** - **Многозадачността** означава, че операционната система може да обработва **няколко задачи (програми, процеси)** симулира "паралелно" според инструкции на task scheduler. Това отново се постига чрез **разделяне на времето** на процесора между задачите. При разпределена многозадачност 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.
- **Еднозадачната еднопотребителска** ОС е операционна система, която позволява на един потребител да изпълнява само една задача наведнъж. Функции като отпечатване на документ, изтегляне на изображения и т.н., могат да се изпълняват само по една. Тази операционна система заема по-малко място в паметта. Еднозначната еднопотребителската заделя всички ресурси за този един потребител и тази една задача, няма разделение на RAM паметта, нито на процесорното време. (За разлика при многопотребителската многозадачна система, където повече от един потребител едновременно може да изпълнява задачи и работят няколко процеса едновременно, но са необходими наличието на изолация между отделните потребители и дефиниране на правата).

2. Опишете с по едно-две изречения работата на следните системни извик-
   вания в стандарта POSIX: open() close() read() write() lseek()

   read - приема три аргумента – файлов дескриптор, буфер и брой байтове. Опитва се да прочете съответния брой байтове от файловия дескриптор и да ги запише в буфера.

   lseek() – Приема три аргумента – файлов дескриптор, отместване и отправна точка. Премества „курсор“-а във файловия дескриптор, на позиция <отместване> според отправната точка. Има три вида отправна точка – **SEEK_SET** – от началото, **SEEK_END** – от края, **SEEK_CUR** – от текущото положение

Тема 3

1. Дайте кратко определение за: многозадачна ОС, многопотребителска ОС, времеделене. Опишете разликата между многопотребителска и многозадачна работа. Какви качества на ОС характеризират тези две понятия?
    - Тема 2 – многозадачност, времеделене
    - **Многопотребителските** ОС - разширяват концепцията за **многозадачност**, като **различава потребителите** по отношение **ползването** на **процеси и ресурси**, като например дисково пространство. Те **планира 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**.

- **Разликата** между многопотребителска и многозадачна операционна система е, че при многозадачната ОС имаме един потребител, който едновременно изпълнява няколко задачи. При многопотребителската система имаме множество от потребители, всеки от които изпълнява някакви задачи, като се предоставя възможност на всеки от потребителите да работи, като по този начин изглежда, че за всеки потребител има отделно ядро, което всъщност не е така.

2. Опишете с по едно-две изречения работата на следните системни извик-
вания в стандарта POSIX: open() close() lseek() pipe() dup2()

Тема 4

Опишете ситуацията съревнование за ресурси (race condition), дайте
пример.
Опишете накратко инструментите за избягване на race condition.

- **Race condition** представлява, когато няколко процеса искат да използват един общ ресурс и редът на изпълнение влияе върху резултата. Пример: имаме някъде глобална променлива А=0 и имаме процес Р, който иска да инкрементира А с 1 и имаме процес Б, който в същото време иска да увеличи стойността и с 3. Във 2 различни разигравания на ситуацията може да има различни отговори, защото не знаем кой процес първи ще промени стойността на А. Може да се получи процес Р да започне да инкрементира А, но в същия момент Б да прочете А като 0 и да започне да я увеличава с 3, така процес Р ще направи А=1, но веднага след това Б ще направи А=3, а ние сме искали крайният резултата да е 4.

(а) дефинирайте критична секция, атомарна обработка на ресурса.

- **Критична секция**: Всеки процес, който работи със споделен ресурс има поне едно място в кода си, в което той използва този споделен ресурс, който **не трябва да се използва от друг процес по същото време**.
- Атомарна обработка на ресурса:
  Атомарно = **непрекъснато, цяло, неразделимо**.
  Това означава, че **операцията се изпълнява от началото до края, без да бъде прекъсната** от други процеси.
  Операциите със семафори са атомарни, защитени – в даден момент един процес може да изпълнява само една от двете команди, т.е. или wait, или signal.

(б) инструменти от ниско ниво, специфични хардуерни средства.

- Най-простият **инструмент** за избягване на race condition е **spinlock**. Това е инструмент на най-ниско ниво, който на практика ни дава възможност да "**заключим**" достъпа до даден ресурс, когато го използваме ние, и да го "**отключим**" в последствие, когато вече не го използваме. Това става с наличието на един **допълнителен бит**, който ни указва дали ресурса е свободен за използване, или не е.

(в) инструменти от високо ниво, които блокират и събуждат процес.

- Семафорът е **абстрактен механизъм** от високо ниво за **синхронизация** на процеси, използващи общи ресурси. Основава се на принципа за **приспиване** и **събуждане** на процес. Ако един процес **няма работа**, която да извърши в даден момент, и очаква външно събитие, то той бива приспан до настъпване на **чаканото събитие**. Когато това събитие настъпи, приспаният процес се събужда и продължава от там

където е стигнал. Биват 2 вида семафори – **слаб**(имплементиран с опашка) и **силен**(имплементиран не на принципа FIFO)

Каква е спецификата на файловете в следните директории в Linux:
/etc /dev /var /boot /usr/bin /home /usr/lib /var/log

/etc - Съдържа **конфигурационни файлове** на системата и програмите.

Пример: `passwd`, `fstab`, `hosts`.

/dev - Съдържа **файлове на устройства** – представят хардуерните устройства като файлове.

Пример: `sda` (диск), `tty` (терминал), `null`.

/var - Съдържа **променящи се данни** – файлове, които се обновяват по време на работа. Пример: поща, опашки за печат, временни файлове.

/boot - Съдържа файлове, нужни за **стартиране на системата** (bootloader, ядро).

/user/bin - Съдържа **изпълними файлове (програми)**, достъпни за всички потребители.

/home - Съдържа **домашните директории на потребителите** – лични файлове, настройки.

/user/lib - Съдържа **библиотеки**, нужни за програмите в `/usr/bin` и други.

/var/log - Съдържа **лог файлове** – записи за събития в системата.

Тема 5

1. Хардуерни инструменти за защита (lock) на ресурс:
   (a) **enable/disable interrupt**
   - **Инстукрцията блокира временно хардуерните прекъсвания**. Когато се влезе в **режим на работата в ядрото**, кода извиква процедура от ядрото да го изпълни и може обработката на прекъсването да се сложи в опашката
     Disable intr - Тази инструкция **временно изключва прекъсванията**, за да може процесорът да **завърши критична операция**, без да бъде прекъснат.
     Enable - След като критичната операция завърши, прекъсванията се **разрешават отново**, и процесорът може да реагира на нови сигнали.

   (b) test and set
   - Тази инструкция се използва, за да се **пише** в паметта и да се **върне старата** стойност като атомарна операция.

   (c) atomic swap
     Atomic swap (или атомарна размяна) е **инструкция на процесора**, която **разменя стойностите между регистър и памет**, **без възможност друг процес да се намеси по средата**.

   Опишете инструмента **spinlock**, неговите предимства и недостатъци.
   - Тема 4 – б
   - При него имаме **допълнителен бит** – lock, който пази стойност 0 ако ресурсът е свободен, и стойност 1, ако е зает. Тъй като този бит е споделен ресурс, за обработката му ще използваме **test-and-set**.
   - Тук обаче е възможно да възникне проблем, ако процесорът прекъсне изпълнението на процеса, затова е важно да се забранят **прекъсванията**.
   - Важно е да се отбележи, че **не** можем да имаме **рекурсия** в **критичната** секция, защото по този начин извикания процес ще чака lock-а да се освободи, а текущия процес ще чака рекурсивно извикания да приключи своето изпълнение - което никога няма да се случи, т.е. ще настъпи **deadlock**.

2. Каква е спецификата на файловете в следните директории в Linux:

/etc /dev /var /proc /bin /home /usr/doc

/proc - Съдържа **виртуална файлова система**, отразяваща **информация за процеси и ядрото**.

/user/doc - Съдържаше **документация за инсталираните програми**.

Тема 6

1. Опишете понятията приспиване и събуждане на процес (block/wakeup). Семафор – дефиниция и реализация. Опишете разликата между слаб и силен семафор.

```
struct Semaphore {

    int cnt = 0
    list L = Ø
    bit lock = 0

    init(int _cnt) {
        cnt = _cnt
    }

    wait() {
        spin_lock(lock)
        cnt = cnt − 1
        if cnt < 0 {
            L.put(self)
            spin_unlock(lock)
            block()
        } else spin_unlock(lock)
    }

    singal() {
        spin_lock(lock)
        cnt = cnt + 1
        if cnt ≤ 0 {
            pid = L.get()
            spin_unlock(lock)
            wakeup(pid)
        } else spin_unlock(lock)
    }
}
```

- Процес може да бъде **приспан** по 2 причини - **Изчакване на събитие**(определено събитие да настъпи, независимо дали това събитие е някакъв **момент** във времето, някакви **входно-изходно операции**, или **конкретно събитие**) или конкурентност и **споделени ресурси**(процесът може да бъде приспан, докато изчаква достъп до споделен ресурс, като например обща памет, файл или устройство). Когато настъпи конкретното събитие, някакъв **друг процес/сигнал** го **събужда** (wakeup) или ако е приспан заради конкурентносто - семафор или спинлог, той го събужда. Обикновено сам процес може да промени състоянието си от running в sleeping, но обатното не е възможно. Block() се използва, когато процес сам приспива себе си, докато при wakeup(pid) – процес поръчва друг процес да бъде събуден.
    - Тема 4 – в + Семафорът е споделен обект, процесите, които го използват, го виждат. Операциите със семафори са атомарни операции – sem.wait(), sem.signal(). Със семафора асоцииираме структура L, към която са асоциирани приспаните процеси. Ако L е имплементирана с опашка на принципа FIFO , то това е силен семафор, в противен случай е слаб. При слабият семафор нямаме гаранция за реда на събуждане и процес може да умре чакайки.

2. Опишете накратко различните видове специални файлове в Linux: външни устройства, именувани в /dev, псевдофайлове в /proc, линкове – твърди и символни, команда ln сокети
    - Външни устройства в /**dev** – в linux всичко е файлове, включително и устройствата. Т.е. /dev е специална директория, където се създават **специално файлове, които представляват хардуерни устройства.** Character device – достъпват се данните байт по байт(колавиатура). Block device – данните се достъпват на блокове(хард диск)
    - **Псевдофайлове** в /**proc** (виртуална файлова система, предоставяща инфо за процеси в ядрото и инфо за хардуера. Тя **не съществува физически** на диска(затова псевдофайлове) – генерира се **в реално време от ядрото**.). Всеки процес има собствена поддиректория `/proc/<PID>/`, където `<PID>` е неговото ID.
    - *линкове* – твърди и символни, команда ln
        - Symbolic link – псевдофайл, който представлява алтернативно **име** за друг файл или директория. Те могат да сочат към всички видове файлове (нормални файлове, директории, специални файлове, други символни линкове и т.н.). Ако бъде преименуван, преместен или изтрит оригиналният файл, линкът става невалиден (счупен; broken). Те могат да имат различни права на достъп от файловете, към които сочат.
        - Hardlink – За разлика от symbolic links, които сочат към името на друг файл, твърдите линкове се насочват към **inode**-а на **съществуващ** файл. По този начин различни имена на файлове се свързват към един и същ inode. Не могат да сочат към директории. Преименуването, преместването или изтриването на „оригиналния“ файл няма ефект върху тях. Промяната на правата на достъп на файл променя правата на достъп на всички негови твърди линкове.

- **ln** – създава линкове към файлове; ако не подадем на командата никакви флагове, тя създава hardlink на подадения файл; ако подадем на командата -s, тя създава symbolic link на подадения файл

- ***сокет*** - специален файл, използван за комуникация между процесите (инструмент за създаване на комуникационен канал от тип **конекция**), който позволява комуникация между два процеса без роднинска връзка. За разлика от именуваните тръби, които позволяват само еднопосочен поток от данни, сокетите са напълно съвместими с дуплекс(двупосочна комуникация). При изпълнение на команда ls -l, сокетите са отбелязани със символ ‚s' като първа буква на символната репрезентация на правата за достъп.

Тема 7

s.init(1);
P
s.wait()
p1
p2
p3
s.signal()

1. **Взаимно изключване** – допускане само на един процес до общ ресурс. Опишете решение със семафори.
   - **Mutex** – или семафора декларирана с коунтър 1(s.init(1)). Използва се за защита на критични секции. Той позволява само един процес или копие да влезе в дадена критична секция и да работи там сам. След приключване на критичната секция той освобождава мутекса, той става пак 1 и някое друго копие или процес може да влезе да ползва критичната секция.
2. Качества и свойства на конкретните файловите системи, реализирани върху block devices. Ефективна реализация, отлагане на записа, алгоритъм на асансьора.
   - Йерархична структура, журналиране, управление на права и защита на данните, буфериране, кеширане за ефективност, индексна организация(иноди)
   - В UNIX е прието в директориите да се съдържат прости обекти, които се състоят от <име, inode>. **Inode** идва от **index node – номер на блок върху диска**, по който може да разберем всички атрибути на файла. Под име се приема абсолютен път към наследника. Тази особеност на UNIX позволява един и същи файл да има няколко имена (hardlinks). По този начин, един файл ще бъде изтрит, чак когато се изтрият всички негови твърди връзки. При всяко отваряне на файл се създава нова твърда връзка, което дава възможност на програмата да продължи своята работа, дори ако друг процес промени съдържанието на файла.
   - **Inode** е **указател към таблица**, която се състои от 2 части: атрибути и указатели (адресна таблица); например, ако в адресната таблица **има 13 указателя**, първите 10 са директни адреси към първите 10 сектора на файла; 11ти,12ти указател са прости таблици, т.е. сектори, които съдържат адреси на сектори; 13 указател е таблица с 3 слоя (сочи към таблица, изградена от таблици, които съдържат сектори)
     11-ти указател – единичен индиректен:
     Сочи към блок, който съдържа адреси на други блокове.

     12-ти указател – двойно индиректен:
     Сочи към блок → който сочи към блокове → които съдържат адреси на блокове с данни.

     13-ти указател – тройно индиректен:
     Сочи към блок → който сочи към блокове → които сочат към блокове → с адреси на реални блокове с данни.

   - Самият **твърд диск** се състои от **няколко области**: **група inodes** (които от своя страна са разделени на отделни области), по-голяма **област със сектори**, които съдържат

метаданни и две служебни зони, в които да се запише информация за това кои сектори са свободни и кои inodes са свободни.

- Придвижването на главата на това устройство от една пътечка на друга е механично и става бавно; Всяка пътечка е разделена на няколко сектора от по 512 или 1024 байта; Софтуера знае колко е голям сектора и колко е времето за преминаване от един до друг сектор; Файла се представя като много сектори; Файлът се разполага там където може, процесът на разхвърляне се нарича фрагментация. (ТОВА Е ИНФОРМАЦИЯ ЗА ЕФЕКТИВНА РЕАЛИЗАЦИЯ – НЕ Е СИГУРНА)
- Записваме промените във файл, който се нарича **журнал**. В журнала се запазва пълна информация за операциите над файлове и при запълване, спираме и отразяваме операциите от журнала над файловете. Ако спре токът, журналът ще пази последните промени, а файловете ще са консистентни (т.е. няма да настъпят повреди). Четат се първо старите файлове и се проверява дали в журнала са променени. Ако спре токът, докато пишем операцията/транзакцията, тя няма да се е изтрила преди да е завършила и затова ще я пазим все още в журнала. Във файловите системи транзакцията е елементарна файлова операция (read, write, open). Журналът (log файл) се намира или в друг диск, или в друг дисков дял, за да може двете паралелно да работят, но в персоналните устройства журналът е файл в самата файлова система или в същия дял.

- В операционните системи **алгоритъмът на асансьора** се използва за управление на споделените ресурси и оптимизиране на достъпа до тях. Обикновено се прилага в контекста на управлението на дискови операции, където множество процеси или нишки изпращат заявки за четене или запис на данни на диск. Имаме **2 приоритетни опашки** като едната е за нагоре, другата за надолу. В случай на пътник, който е натиснал копчето за асансьора в дадена посока, асансьора тръгва от 0вия етаж примерно нагоре като взима само тези пътници, които са за нагоре, докато през това време приоритетната опашка за надолу се пълни. Като стигне на последния етаж се изпълнява пр. опашка за надолу като с най-голям приоритет са тези хора, които са най-близо да асансьора. Идеята му е, че близки сектори на диска ще се променят бързо.

  Това **намалява времето за търсене (seek time)** – т.е. времето, за което главата на диска стига до търсения сектор.

  Затова се казва, че „**близки сектори ще се променят бързо**“ – защото се обработват в текущата посока на движение, без излишно прескачане.

s1.init(1) s2.init(0)	
WRITER:	READER:
s1.wait() .. WRITE .. s2.signal()	s2.wait() .. READ .. s1.signal()

Тема 8

1. Комуникационна тръба (pipe), която съхранява един пакет информация – реализация чрез редуване на изпращача/получателя. pipe с буфер – тръба, съхраняваща n пакета информация. Използване на семафорите като броячи на свободни ресурси.

- Идеята на решението е, че тъй като може да се записва само по един пакет, трябва да се редуват изпълненията на 2-та типа процеси. Така процесът, който чете от тръбата трябва да изчака нещо да бъде записано вътре и се приспива, докато това не се случи. Първият процес записва в тръбата един пакет информация. Всеки следващ процес, който пише в тръбата ще бъде приспан, докато процес от другия тип не прочете записаното в тръбата. След като го прочете той ще сигнализира на процеса писател, като един от приспаните ще се събуди и ще запише нещо.

- Тръбата се ползва от няколко паралелно работещи изпращачи/получатели на байтове. Процесите изпращачи слагат байтове в края на опашката, получателите четат байтове от началото на опашката.
- Необходими структури от данни:
    - **Опашка** (или масив) Q с **n елемента** – тъй като е възможно да имаме бърз и бавен процес (например: единият чете бързо, а другият пише бавно), е нужно опашката да бъде **по-дълга**, за да не се приспива по-бързият процес (приспиването е бавна операция, тъй като включва в себе си смяна на контекста). В началото тази опашка е празна;
    - **free_bytes** – семафор, който ще индикира за свободните байтове в опашката;
    - **ready_bytes** – семафор, който ще индикира колко байта са готови за четене (до колко е запълнена опашката);
    - **mutex_read** – мутекс, който ще защитава критичната секция за четене;
    - **mutex_write** – мутекс, който ще защитава критичната секция за писане.
- Процесът, който чете, първоначално ще проверява дали има готови за четене байтове. Ако няма, той ще се приспи и ще чака да постъпят такива. Ако има, той ще провери дали някое друго копие на P не чете в момента. Ако да, той отново се приспива. Ако не, той ще прочете байт и ще го запише в променливата. След това ще сигнализира, че в опашката има още един свободен байт чрез подаване на сигнал на семафора free_bytes.
- Процесът, който пише, първоначално ще провери дали има свободни байтове в опашката. Ако няма, ще се приспи и ще чака да се освободят байтове. Ако има, ще трябва да провери дали някое друго копие на пишещ процес не пише в момента в опашката. Ако да, то отново процеса ще се приспи. Ако не, ще сложи байта си в опашката и ще сигнализира, че още един байт е готов за четене, което се осъществява чрез подаване на сигнал на семафора ready_bytes.

Инициализация:

```
buf Item[n]
free_bytes.init(n)
ready_bytes.init(0)
mutex_read.init(1)
mutex_write.init(1)
int last = 0;
first = 0;
```

pop:	**push(item b):**
ready_bytes.wait()	free_bytes.wait()
mutex_read.wait()	mutex_write.wait()
Item t = buf[last];	buf[first] = b
last = (last + 1) % n	first = (first + 1) % n
mutex_read.signal()	mutex_write.signal()
free_bytes.signal()	ready_bytes.signal()
return t;	

2. Права и роли в UNIX, команда chmod.

- роли – u/g/o user/group/others - Всеки файл и директория имат 3 вида групи по отношение на правата: Потребител (User/Owner), Група (Group) и Други (others).

Потребителят е собственикът на файла. По подразбиране, при създаване на файл, лицето, създало файла, става негов собственик.

Групата съдържа множество потребители. Всички потребители, принадлежащи към групата, на която принадлежи файлът, имат едни и същи права за достъп.

Други са всички останали потребители - нито са собственици на файла, нито принадлежат към потребителската група, притежаваща файла.

- Права за достъп – r/w/x read/write/execute

Всеки файл и директория имат 3 вида права: Четене (Read), Писане (Write) и Изпълнение (Execute).

Четене: Дава правото да **отваряме** и да **четем** съдържание на даден файл. При директориите, дава възможност да **видим нейното съдържание** (например с командата ls).

Писане: Дава право да променяме съдържанието на файл. При директориите, дава право да добавяме, премахваме и преименуваме файлове, съхранявани в директорията (ако съответните файлове също го разрешат).

Изпълнение: Ако разрешението за изпълнение не е зададено, не можем да стартираме изпълним файл. При директориите ситуацията е малко по-особена. Ако сравним с правата за четене, там можем само да видим съдържанието на дадената директория (например с команда ls), но **ако нямаме права за изпълнение за дадената директория, не можем да я достъпваме.**

- chmod – правата за различните роли и групи се сменят с командата chmod. Тя приема като аргументи символна или числена репрезентация на новите права, които искаме да зададем за конкретния файл, както и самия файл, който искаме да променим.

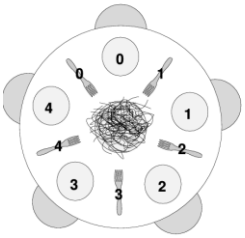
Тема 9

1. Взаимно блокиране (deadlock). Гладуване (livelock, resource starvation) Пример: задача за философите и макароните

s.init(0)	z.init(0)
p1	K1
s.wait()	z.wait()
z.signal()	s.signal()
p2	k2

- Deadlock е събитие, което се настъпва, когато процес чака друго събитие, което никога няма да се случи. Обикновено това се получава, когато нишка/процес чака мютекс или друг семафор, който никога не се освобождава от предишния си ползвател, както и при ситуации, когато имаме два процеса и две заключвания.
- Примерно имаме рандеву p1 > к2 и к1 > p2, обаче след преминаването на p1 или к1, процеса започва да чака другия да уведоми, че е приключил своята работа. Обаче това никога няма да стане, защото пък 2рия процес изчаква първия и се получава deadlock.

- ***Livelock*** се случва, когато два или повече процеса непрекъснато повтарят едно и също взаимодействие в отговор на промените в другите процеси, без да извършват полезна работа. Тези процеси не са в състояние на изчакване и те се изпълняват едновременно. Това е различно от deadlock, тъй като при deadlock всички процеси са в състояние на изчакване, а тук те са „живи“ (все едно вършат дадена работа), но всъщност не вършат нищо полезно.
- ***Starvation*** - настъпва, когато имаме процеси, които чакат своето време за изпълнение, но дадени процеси винаги се изместват да са по-назадв опашката и така техният ред може да не дойде никога. Друг вариант за гладуване е „алчните“ (greedy) нишки използват споделените ресурси прекалено дълго и по този начин ги правят недостъпни за останалите за дълги периоди от време. Пример може да се даде, като охрана на бар и хора чакащи да влязат в този бар. Освобождава се място в бара, но охраната няма да пусне първия наредил се на опашката,а ще пусне току-що наредилия се негов приятел, който е най-отзад на опашката. И Ако така продължат да идват приятели на охраната, ще е се осъществи гладуване спрямо първите наредили се хора. Обикновено гладуването е причинено от прилагането на твърде опростен алгоритъм за съставяне на разписания, по които процесите се изпълняват – **не се взима под внимание типът** на различните процеси.




```
philosopher():
    infinite loop:
        think()
        get_forks()
        eat()
        put_forks()
```

- Ако всички философи вземат десните си вилици едновременно, получаваме deadlock

Решението със `sem(table, N-1)`

```
init():
    sem_init(table, N-1)
    for i in 0 .. N-1:
        sem_init(forks[i], 1)

get_forks(i):
    sem_wait(table)
    sem_wait(forks[right(i)])
    sem_wait(forks[left(i)])
```

Позволяваме максимум **N - 1 = 4 философа** да се опитат да вземат вилици едновременно.

- Ако само 4 философа влязат в `get_forks()`, а 5-ият чака, **винаги ще има поне една вилица, която не е взета**. **Максимум 4 ще се опитват едновременно да вземат вилици**, за да избегнем deadlock. Реално, **само 2 философа могат да ядат едновременно**, но **решението позволява да се редуват без блокиране**.
- Премахваме един от философите от масата
- Това не е проблем, защото така или иначе няма как всички да ядат едновременно. (4 философа, 5 вилици, поне един яде. Нямаstarvation (краен брой стъпки до храната)

```
init():
    for i in 0 .. N-1:
        sem_init(forks[i], 1)

get_forks(i):
    if i mod 2 == 0:
        sem_wait(forks[right(i)])
        sem_wait(forks[left(i)])
    else:
        sem_wait(forks[left(i)])
        sem_wait(forks[right(i)])

put_forks(i):
    sem_signal(forks[right(i)])
    sem_signal(forks[left(i)])
```

Решение 2:

Част от философите взимат лявата вилица, а други взимат дясната

Решение 3: (на deadlock, но не напълно на starvation)

```
init():
    for i in 0 .. N-1:
        sem_init(forks[i], 1)

get_forks(i):
    f1 := min(left(i), right(i))
    f2 := max(left(i), right(i))
    sem_wait(forks[f1])
    sem_wait(forks[f2])

put_forks(i):
    sem_signal(forks[right(i)])
    sem_signal(forks[left(i)])
```

1. За решаване на задачата за **философите**, е нужно поставянето на ограничения: Когато един философ е взел вилица, никой друг не може да я ползва; Да не е възможно да се появи deadlock; Да е възможно повече от един (в нашия случай – повече от двама) философи да се хранят по едно и също време; Да не е възможно философ да гладува дълго време, докато чака за вилица.

Ако всеки философ се опита да вземе първо лявата или дясната вилица, ще н настъпи deadlock, защото, за да се нахрани са му нужни две вилици, които да и използва едновременно и всеки философ ще чака и ще гладува вечно. Решение на този проблем може да се направи и с двуделен граф(където процесите ще са философите, а ресурсите ще са вилиците). Нека номерираме множеството от процеси и множеството от ресурси, така че да знаем на кой процес кои ресурси са необходими. След като знаем на кой процес какво множество от ресурси е необходимо, за да си свърши работата, то той ще може да си го сортира и да ги взема по реда на номерата. Тогава алгоритъмът няма да е: „всеки да взима лявата вилица“, а ще е „**всеки да взима вилицата с по-малък номер**“(напр философ 4 и философ 0 искат вилица 0 първо и ще остане влициа 4 свободна в най лошия случай). Така deadlock няма да настъпи (естествено има малка вероятност за гладуване на някои философи, но deadlock няма да има). **не може да се образува кръг**, защото номерата на ресурсите ще бъдат **възходящи**, а не **циклични**. тази стратегия **не гарантира справедливост**

2. Единна йерархична файлова система в UNIX.
   Файлове и директории, команди – cd, mkdir, rmdir, cp, mv, rm
   - В UNIX абстрактната файлова система е кореново дърво. Коренът представя началото на файловата система. Върховете представляват директории, а листата - файлове, като всеки връх (директория) също представлява кореново дърво. Има някои специални директории, които присъстват във всяка файлова система (напр. /dev, /etc, /tmp и т.н.) и служат за правилната **организация** и **работа** на системата. Към всеки файл, освен **името** и **съдържанието**, се асоциират и други атрибути - **тип** на файла ("-" – обикновен файл, "d" – директория, "c" – символно устройство/character special device (напр. клавиатура), "b" – блоково устройство/block special device (външно устройство, което съхранява масив от байтове), "p" – именувана тръба (FIFO), "s" – socket/конектор (крайна точка за комуникация)), **права за достъп**, потребител-**собственик**, потребителска **група**, **размер**, **дата на промяна**, **дата на създаване** и др.
   - В Linux тези атрибути се пазят **отделно от самия файл**, което от една страна прави достъпа до тях по-бавен, но от друга страна дава **възможност** за съществуването на т.нар. **hardlinks**, т.е. за създаване на друго име на файла (файлът ще бъде същия,

понеже ще сочи към същите атрибути, но името му може да бъде различно). По този начин, файлът ще бъде изтрит от паметта чак когато всички hardlinks към него се изтрият.

Тема 10

1. Процеси в многозадачната система. Превключване, управлявано от синхронизация. Превключване в система с времеделене – timer interrupt.
    - **Процес** е програма по време на изпълнение.(**Процесът** е изпълнение на една програма(**файл с инструкции**, записани на диск). ОС създава **процеса**) Той има различни състояния: R, A, S. Процесите могат да бъдат **спящи** (те чакат входно-изходни операции или момент от времето), **активни** (в момента активен процес, но който чакат CPU) и **работещи**(такива, които ползват CPU в момента).
    - Преход: Работещ процес да премине в спящо състояние. Този процес се нарича **блокиране**. Това настъпва, когато процесът чака входно/изходна операция (wait – предизвиква се от синхронизиращия механизъм - семафора, който от своя страна приспива процеса, само ако ресурсът е зает) или ако чака момент от времето (sleep – предизвикан от синхронизираща операция, която задължително го приспива).
    - Спящият процес може да премине в активен (да се събуди). Събуждането на процеса се предизвиква от завършването на входно-изходна операция на друг процес, който чрез signal() ще подаде сигнал, че е освободен ресурс.
    - Работещ (Running) процес може да промени състоянието си на блокиран (да му бъде отнето процесорното време), ако твърде много време прекара в процесора. В такъв случай времето му изтича и той бива прекъснат от таймер, за да освободи CPU ресурс. Това е така заради **времеделенето** – всеки процес има максимално количество време, което може да работи. **Обслужването на спирането** се извършва от **алгоритъм в ядрото**, а самото **спиране** – от **часовника**.
    - Активният (Ready) процес, преминава в работещ (Running), когато му се предостави процесорно време. Извършва се смяна на работещия процес, поради изтичане на време на даден процес. Ядрото тогава решава, кой чакащ процес да заработи (зависи от алгоритъма на ядрото и от неговия Task Scheduler). Когато спящият процес е приспан заради очакване на момент от времето, то той може да стане активен, като този процес се инициира от timer (прекъсване на часовника).
2. Опишете функционалността на следните команди в Linux: ls, who, find, ps, top(Показва динамично (в реално време) използването на системни ресурси от процесите.)

Тема 11

1. Възможни състояния на процес. Механизми и структури за приспиване/събуждане. Диаграма на състоянията и преходите между тях.
    - **Running** – процеси, които активно се нуждаят и използват ядро и процесор, който изчислява. Това са процеси, които от гледна точка на потребителя имат работа за вършене. Те може да се изчакват и редуват, ако не стигат процесорите, но като цяло имат нужда от изчислителна мощ. От гледна точка на потебителя те са една група, но от гледна точка на ядрото те са:
        - Running – изчисляват се в момента;
        - **Ready** – в момента няма процесор за тях, но при следващия такт те ще получат управление.
    - **Sleeping** – спящите процеси от гледна точка на потребителя са работещи програми, които са стигнали в състояние, при което нямат нужда да изчисляват нещо докато не настъпят интересни за тях събития в системата. Те са в комуникация с друг процес или

устройство и очакват да им се подадат данни, но очакваните процеси нямат готовност да им подадат (например поради запушен комуникационен канал или поради бавна работа на другата страна и т.н.). От гледна точка на реализацията може да чакат:

- I/O – очаква извършване на входно изходни операции;
- Time – очаква да настъпи времеви момент;
- Signal - очаква сигнал от друг процес за промяна на състоянието му (на другия процес);
- Процеса бива приспан, защото страницата, с която иска да работи не е на реалната памет, а е някъде на твърдия диск.

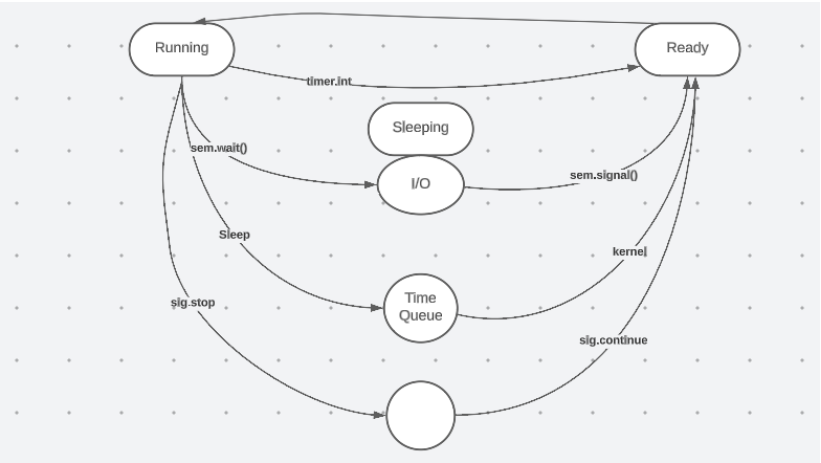
- **Stopped** – спрян процес (не представлява интерес нито за потребителите, нито за операционната система)
- **Zombie** – процес, при който е започнало спирането но не е завършило (пускането и спирането на процес са бавни и многостъпкови събития)

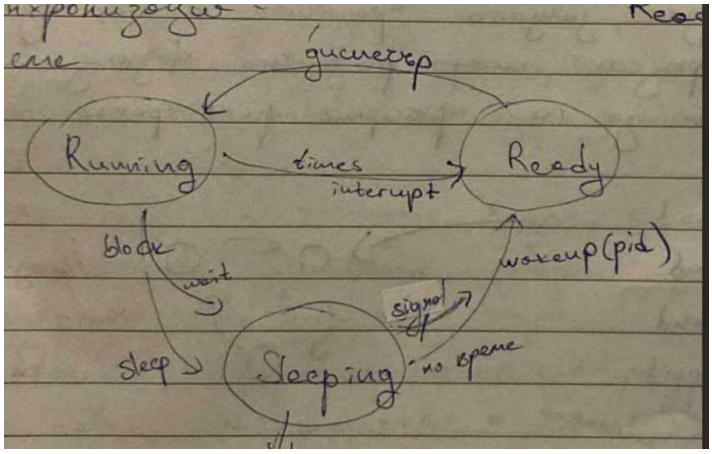
- Механизъм от ниско ниво е spinlock- който се използва за защита на критични секции от паралелни достъпи от множество процеси или нишки. Когато процесът или нишката се опитват да достъпят критичната секция и забележат, че спин-локът е зает, те продължават да "въртят" (spin) в цикъл (без да спят), докато спин-локът не бъде освободен.

  Като при достъпване на критичната секция ппц ще е хубаво е да бъде приспан, за да не губи процесорно време докато чака. критичната секция е дълга или локът ще е зает дълго. Spinlock = бърз, но неефективен при дълго чакане.

- Механизъм от по-високо ниво Семафор - 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

- // схема на диаграмата на състоянията





-

2. Опишете функционалността на следните команди в Linux:
   vi, tar, gcc

vi - Текстов редактор в терминала (един от най-старите и мощни); normal, insert, visual, realpce, command line

gcc - **Компилатор за C и C++ програми.** Превежда `.c` или `.cpp` файлове в изпълними програми.

Тема 12

1. Процес и неговата локална памет – методи за изолация и защита. Йерархия на паметите – кеш, RAM, swap. Виртуална памет на процеса – функционално разделяне (програма, данни, стек, heap, споделени библиотеки).
   - ОС трябва да може динамично да разпределя наличната физическа памет между процесите и да се съобрази с тяхната активност и до колко те използват паметта, за да организира ефективно изчислителния процес. Всеки процес има собствено адресно пространство – тоест локална памет, изолирана от другите процеси.
   - Методи:
   - Виртуална памет Всеки процес вижда „собствена" (виртуална) памет – изолирана от другите
   - MMU (Memory Management Unit) Хардуер, който превежда виртуални адреси към физически; защитава достъпа
   - Права на достъп Страници в паметта могат да са само за четене, изпълнение или запис
   - Kernel/User Mode Процесите работят в потребителски режим, а достъп до чувствителни ресурси е само чрез системни повиквания (syscalls) към ядрото
   - Механизми, които ползва са:
     - a. Сегментация – Програмата **не се съхранява като един блок**, а се разделя на **логически части**, наречени **сегменти**. Процесорът използва регистър, за да намери физическия адрес на сегм, той се съхранява в непрекъснат блок.
     - b. Управление на базов и граничен регистър – базов – сухр нач адрес на областта от паметта. Логическите адреси се интерпретират като отместване спрямо стойността в базовия рег.
     - c. организация чрез стр – дели паметта на части и позволява на програми да се сухраняват на произв места, без да се нуждаят от последователни блокове памет. Паметта се разделя на **малки еднакви блокове**. в хардуера има таблица, която казва за всяка страница къде се намира в реалната памет и какви права има. **всеки процес да има своя виртуална памет.**

   - **Кеш**: Кешът е малка, бърза и изключително близка до процесора памет, която се използва за временно съхранение на данни, които се използват най-често от процесора. Кешът помага за намаляване на времето за достъп до данните, което подобрява общата производителност на системата.
   - **RAM** (**памет с пряк достъп**): RAM е основната памет на компютъра и се използва за временно съхранение на данни и инструкции, които процесорът трябва да достъпва бързо. Тя предоставя бърз достъп до данните и е необходима за изпълнение на програмите и операционната система. Всяка активна програма и данни се зареждат в RAM, като по този начин процесорът може да ги достъпва по-бързо, отколкото да ги чете от по-бавния твърд диск.
   - **Swap**: Swap е **техника за управление на паметта**, която се използва от операционната система, когато активната RAM памет е напълно използвана. Когато това се случи, операционната система прехвърля част от неактивните данни и процеси от RAM на специално създаденото пространство на твърдия диск, наречено "**swap space**" или "**файл за размяна**". Това освобождава RAM за по-активните процеси и помага за поддържането на стабилна работа на системата.
   - **Виртуалната памет на процеса е концепция**, която позволява на операционната система да разделя физическата памет (RAM) между различните процеси, които работят в системата. Функционалното разделяне на виртуалната памет се отнася до начина, по който процесът е организиран в паметта.
   - **Програма**: съхраняват **инструкциите** на програмата. Това включва изпълнимия код и константи.

- **Данни**: съхраняват променливите и данните(глобални и статични променливи), използвани от програмата. Те се инициализират по време на стартиране на програмата и се запазват в секцията за данни на виртуалната памет.
- **Стек**: Стекът е областта от виртуалната памет, която се използва за съхранение на локални променливи, параметри на функции и връщане на адреси от функции. Стекът расте и се намалява динамично, по време на изпълнение на програмата. Грижи се за рекурсивни извиквания и за това да няма stack overflow от тях
- **Heap**: Heap е областта от виртуалната памет, където се създават и унищожават динамично заделени обекти като динамични масиви и обекти. Приложението има контрол върху управлението на паметта в тази област. Това значи, че **самата програма решава кога да задели и освободи памет в heap-а**, **не операционната система автоматично**.
- **Споделени библиотеки**: Това са общо използвани библиотеки, които могат да бъдат споделени между различни процеси. Те се зареждат във виртуалната памет на процеса и всеки процес може да ги използва без да ги зарежда отново. те **се виждат** във виртуалното адресно пространство на процеса. Когато процесът се стартира, споделените библиотеки **се "вмъкват"** (mapped) в неговото виртуално пространство. Не са като `data segment`, `stack` или `heap`. Кодът в библиотеките **не се копира наново**, а е **споделен.** Вместо това, **всички процеси споделят една и съща копие** в паметта (обикновено само за четене)

2. Опишете функционалността на следните команди в shell: echo, read, test, if, for, while

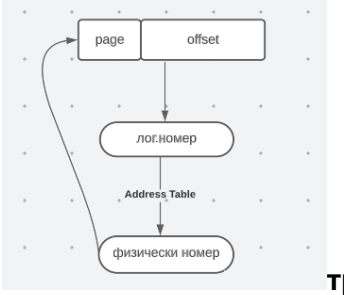
Read - Използва се за **въвеждане от клавиатурата (стандартен вход)** и записва стойността в променлива

Test - Проверява **логическо условие** (числово сравнение, съществуване на файл и т.н.). Често се използва с `[` и `]` (които всъщност са символи за `test`)

- If - Условна конструкция – **изпълнява команда/код само ако условието е вярно**.

Тема 13

1. Таблици за съответствието виртуална/реална памет. Ефективна обработка на адресацията – MMU, TLB.

- Някои от страниците на виртуалната памет съответстват на реални страници от RAM паметта. Други могат само да се четат. Тези страници се предоставят при стартиране на програмата. Даден **адрес** изглежда по следния начин: **Page number - offset**. Page num - логически номер на страница, който се използва за влизане в адресната таблица(Виртуална страница №X → Физическа страница №Y). Offset – при реалния адрес, откъде почва инфото
- Страници на Ram, Страници изместени на диска (swap), Страници от споделени библиотеки или файлове, незаредени/неинициализирани
- 

трансформацията обаче губи време, поради което се използва **допълнително устройство в хардуера** – **MMU**.

- Хардуерът, който отговаря за транслацията между логически и физически номер на страницата и въобще за управлението на адресацията, се нарича **MMU** (**Memory Management Unit**). Инструментът, който използваме за оптимизиране на работата на MMU, се нарича **TLB (Translation Lookaside Buffer**).

- **MMU** (Memory management unit) – е хардуер, през който всички референции на паметта преминават, като главно извършват транслацията от виртуален адрес във физически адрес, използвайки таблицата със страници и от таблицата със сегменти. Модерните MMU разделят виртуалното адресно 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**) – **кеш**, който хардуера за управление на паметта използва, за да подобри скоростта при транслация на виртуални адреси. Съдържа наскоро вече пресметнати страници като при поискване, затова ги връща бързо. 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 във физически адреси, а записите от таблицата със сегменти – за преобразуване на виртуалните адреси в сегментни адреси.

Сегментна таблица?

Сегментацията разделя програмата на логически части:

- код (инструкции)
- данни
- стек
- heap

Тези части се наричат сегменти и всеки от тях има:

- Base (начален адрес в паметта)
- Limit (размер на сегмента)

2. Какво е Таблица на страниците (Page Table)?

След като сме получили **логически адрес**, сега го превеждаме в **физически адрес** чрез **paging** (странициране).

Паметта е разделена на:

Страници (Pages) – във виртуалната памет

Кадри (Frames) – във физическата памет

Таблицата на страниците казва:
"Страница X от виртуалната памет се намира във Frame Y от физическата RAM."

2. Файлови дескриптори, номера на стандартните fd, пренасочване филтри – cat, grep, cut, sort, wc, tr

Тема 14

1. Избройте видове събития, причиняващи повреда на данните във файловите системи. Опишете накратко стандарта RAID5. Какво е журнална файлова система?

- Има множество причини поради, които може да се появи повреда на файловата система като: загуба на данни поради потребителска грешка(rm); Хардуерни грешки - Изгаряне на хард диск или остаряване, Повреда на дискови сектори, Грешки при четене или запис на данни; Софтуерни грешки - Грешки в драйверите или операционната система, Грешки в самата файлова система, зле написан софтуер; спиране на ток; вирус; различни кодировки;
- **RAID** – Redundant Array of Independent/Inexpensive disks - **стандарт** който осигурява **защита на данните, та**ка че ако един диск изгори, **информацията да не се загуби**. Комбинира множество физически драйвъри в 1 логическа част. Това може да повиши производителността, увеличи повторението на данни(излишък – ако нещо фейлне)
  RAID 0 – няма предпазване от повреда, бърз е тъй като данните се разделят и обработват от няколко диска едновременно. *Тъй като разделянето на дискове разпределя съдържанието на всеки файл между всички дискове, повредата на който и да е диск прави целия RAID 0 том недостъпен. Обикновено всички данни се губят и файловете не могат да бъдат възстановени без резервно копие.*
  RAID 1 – всички дискове имат еднакви данни, записва се на всички едновременно. В случай на проблем просто се ползва друг диск. *Данните се записват идентично на два или повече диска, като по този начин се създава „огледален набор“ от дискове. По този начин всяка заявка за четене може да бъде обслужена от всеки диск в набора. Пропускателната способност за запис винаги е по-бавна, защото всеки диск трябва да се актуализира, а най-бавният диск ограничава производителността на запис.*
  RAID 2 – използва **Хамингово кодиране** на данни. Използва специален диск за паритет, който съхранява информация, необходима за възстановяване на данните в случай на повреда на диск. всеки бит от байт се записва на отделен диск. Бърз **четене**, но **бавен запис** заради пресмятания
  RAID 4 – Имаме **паритетен диск**, който е математическа сумата от блоковете данни на другите дискове и **при загуба на даден диск**, от паритетния могат математически да се извадят останалите и така ще получим данните на изгорелия. Разделя (striping) данните на **блокове**, които се записват на няколко диска. При запис на нови данни: Системата изчислява новия **паритет,** Записва новите данни + новия паритет. Недостатък - Всички операции по запис трябва да обновят **същия паритет диск.** Това го прави "тясно място" (bottleneck), особено при много паралелни записи
  RAID level 5 – е конфигурация на дискова система, която **комбинира производителност, капацитет и надеждност**. Тук **дисковете са разделени на блокове** и от всички първи блокове на всички дискове имам един **паритетен блок, който пази данните на осталите за съответния блок**. Разпределение на данните по блокове (striping). Изисква минимум 3 диска. Разпределена четност – вместо да запазва цялата информация за четност на 1 диск, я разпределя между всички. Всеки блок данни се записва на 1 диск, инфо за възстановяване се разпределя. Допустима е загуба на 1 диск. Бавно записване заради изчисляване на четност, но реално по бързо от дргите варианти с паритет, бързо четене(данните може да се четат паралелно)
  Raid 6 – същото но с по 2 паритета. Поне 4 диска
- В **журналната файлова** система всички промени, които трябва да бъдат направени върху файловата система, се записват първо в журнала. След това, когато системата е готова, тези промени се прилагат върху съответните файлове и структури на файловата система. Това осигурява цялост на файловата система и подобрява времето за възстановяване след смущение.

2. Свързване и допускане до UNIX система – login. Конзола – стандартен вход, стандартен изход, стандартна грешка. Команден интерпретатор – shell. Изпълнение на команди, параметри на команди

- Съществуват 3 връзки когато се работи в shell: stdin(стандартен вход, по дефолт 0), stdout(стандартен вход, по дефолт 1), stderr(стандартна грешка 2). При изпълняването на команди или скриптове те могат да бъдат пренасочени(pipeline). Чрез тях се осъществява комуникация/връзка между потребителя и shell-а. При създаване на процес дете, то наследява стандартните потоци(дескриптори) на родителя.
- За да се достъпи системата и нейните данни и функционалности, трябва да се изпълни процеса по идентификация(login). След успешно преминаване на login(правилно въведени username и парола) може да се използва shell и чрез него също да се изиска някакъв процес по идентификация (примерно при свържем с отдалечен достъп - ssh).
- След като сме се идентифицирали и стартираме shell, се отваря черен прозорец(prompt) като показва от какъв потребител работи: обикновен потребител или root. Различни реализации на shell - bash, zsh, tcsh, sh. Чрез този prompt можем с команди да указваме какво да се изпълни или свърши, като той може да изведе пълна информация за нашата система.
- **Командите** в shell представляват някаква дума или съкращение, като в зависимост от командата тя може да има нужда от различни флагове. Флагът представлява след изписването на командата да се остави празно място и да не напише тире и дадена буква/дума и след нея в зависимост дъден параметър. Една команда може да има множество флагове. Командите са мощен инструмент, защото чрез тях може да се терминираме, създаваме, паузираме работещи процеси, да менажираме паметта на системата, да следим времетраене и хронология на дадени събития по системата. Също съществуват и филтриращи команди, които принтират информация, търсят, броят, редактират символи/байтове. Всяка команда има стандартни потоци, които могат да се пренасочват.

Тема 15

1. Опишете разликата между синхронни и асинхронни входно-изходни операции. Дайте примери за програми, при които се налага използването на 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 коректно комплектоване на операцията – четене/запис на всички предоставени/поръчани данни във/от входно-изходния канал, или цялостно изпълнение на друг вид операция (примерно, изграждане на TCP връзка).
- При **асинхронна** входно-изходна операция системното извикване не приспива (не блокира) потребителския процес, поръчал операцията. Същевременно, при невъзможност да се окомплектова операцията, ядрото връща управлението на процеса със специфичен код на грешка и друга информация, която служи за определяне на степента на завършеност на 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 буфер за изходни данни или друга ситуация, водеща до блокиране.
- **Примери** за асинхронни входно-изходни операции: Когато ползваме **WEB-browser**, той трябва да реагира на входни данни от клавиатура и мишка, както и на данните, постъпващи от интернет, т.е. на поне 3 входни канала. Браузерът проверява чрез асинхронни опити за четене по кой от каналите постъпва информация и реагира адекватно. **Сървър в интернет** може да обслужва много на брой клиентски програми, като поддържа отворени TCP 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.

2. Опишете с по едно-две изречения работата на следните системни извиквания в стандарта POSIX: socket(), bind(), connect(), listen(), accept()

Тема 16

barrier.init(1);
mutex.init(1);
roomEmpty.init(1);
cnt=0	

Инструкции на READER:	Инструкции на WRITER:

barrier.wait()
barrier.signal()
mutex.wait()
cnt=cnt+1
if (cnt==1) roomEmpty.wait()
mutex.signal()
READ
mutex.wait()
cnt=cnt-1
if (cnt==0) roomEmpty.signal()
mutex.signal()
	barrier.wait()
	roomEmpty.wait()
	WRITE
	roomEmpty().signal()
	barrier.signal()

1. Опишете понятието „пространство на имената“. Как изглежда това пространство в ОС Linux?

- **Пространството на имената** – използва се за идентифициране и препращане към обекти от различен вид; гарантира, че всички дадени групи обекти имат уникални имена, така че да могат лесно да бъдат идентифицирани. механизъм за изолиране на ресурси между процеси.
- В Linux операционната система има различни типове пространства на имената, като пространство на имената на **процесите**, **файловата** система и **мрежовите интерфейси**, междупроцесна комуникация. Тези пространства на имената предоставят изолация и контрол върху съответните ресурси, позволявайки на процесите да работят в изолирани и независими среди. Те са **структури на ядрото**, които изолират различни ресурси между групи от процеси. Всеки вид namespace изолира различен тип ресурс. един набор от ресурси, докато друг набор от процеси вижда различен набор от ресурси. Процесите могат да създават допълнителни именни пространства и също така могат да се присъединяват към различни именни пространства.

- //структура, обекти и техните атрибути във VFS за ОС Linux.
- **VFS (Virtual File System)** е абстрактен слой в ядрото на Linux, който позволява на операционната система да работи с различни видове файлови системи

2. **`super_block`** – представя монтирана файлова система; тип на ФС, размер на блоковете, брой иноди и блокове, указател към кореновия инод.
3. **`Inod`** - описва съдържанието на файл или директория, размер, време на създ...
4. Dentry – свързва име с inode
5. File – описва отворен файл
6.
7.
8. Една от класическите задачи за синхронизация се нарича „Задача за читателите и писателите“ (readers-writers problem). Опишете условието на задачата и решение, използващо семафори.

- Идеята на решението на задачата е Идеята за решаване на задачана е следната: Когато първият читател се опита да влезе в стаята (под стая разбираме критична секция), той трябва да провери дали е празна, след което всеки следващ читател е необходимо да проверява само дали в стаята има поне един читател. Възможно е обаче писателите да гладуват, заради наличието на много на брой читатели или бавни такива. Затова използваме бариера, която ще спира читатели да навлизат, при условие, че има поне един писател, който чака да влезе в критичната секция.

Тема 17

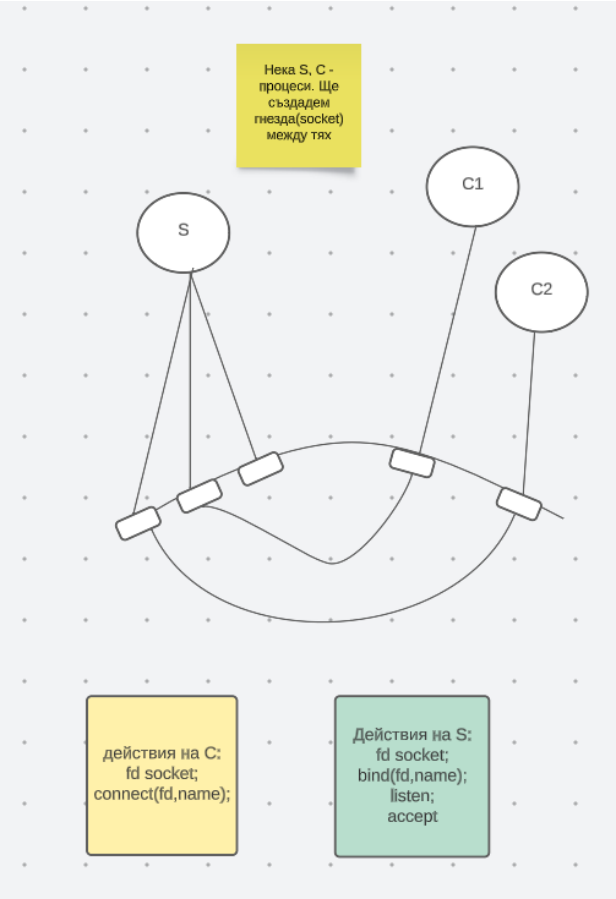
1. Опишете какви атрибути имат файловете в съвременна файлова система, реализирана върху блочно устройство (block device). Опишете накратко реализацията и целта на следните инструменти:

- **Атрибутите** в съвременната файлова система: Име на файла (File Name); Собственик(owner); Група(Group); Размер на файла (File Size); Атрибути за достъп (Access Permissions): Тези атрибути контролират правата за достъп до файла. Те определят кой може да прочете, запише или изпълни файловете; Дата и час на създаване (Creation Date and Time): Това е дата и часът, в които файлът е създаден във файловата система; Дата и час на последна промяна (Last Modified Date and Time); Дата и час на последен достъп (Last Access Date and Time); Индекс (File Index): Това е уникален идентификатор, присвоен на файла във файловата система. Индексът може да се използва за бързо намиране и управление на файловете;

  (а) **отлагане на записа**, алгоритъм на асансьора.
  - Записваме промените във файл, който се нарича **журнал**. В журнала се запазва пълна информация за операциите над файлове и при запълване, спираме и отразяваме операциите от журнала над файловете. Ако спре токът, журналът ще пази последните промени, а файловете ще са консистентни (т.е. няма да настъпят повреди). Четат се първо старите файлове и се проверява дали в журнала са променени. Ако спре токът, докато пишем операцията/транзакцията, тя няма да се е изтрила преди да е завършила и затова ще я пазим все още в журнала. Във файловите системи транзакцията е елементарна файлова операция (read, write, open). Журналът (log файл) се намира или в друг диск, или в друг дисков дял, за да може двете паралелно да работят, но в персоналните устройства журналът е файл в самата файлова система или в същия дял.
  - Тема 7 - 2

  (б) поддържане на **буфери** (**кеширане**) на файловата система.
  - За да се ускори работата на системата се пазят специални кешове, в който се пазят най-често използваните редове от адресната таблица. Има 3 вида кешове:
  - Кеш за инструкции - изпълнимия код се разполага в специален кеш, който поддържа сравнително дълги парчета от RAM паметта и сравнително малко на брой;
  - Кеш за работа с данни - къси фрагменти ще се използват много често (напр. броячи, локални променливи); четене и писане на данни
  - Кеш за упавление на виртуална памет – използва се т.нар TLB - кешира съответствия между виртуални и физически адреси, за да ускори адресното преобразуване.
    - Входовете съдържат виртуален адрес → физически адрес и други флагове (напр. права за достъп).
    - Няма голям размер
    - Използва се от хардуера за бързо преобразуване на адреси при работа с виртуална памет, като значително намалява необходимостта от достъп до таблиците на страниците в RAM.
  - Буфери при писане **(Write-back caching):** Операционната система може да задържа промените по файловете в кеша и да ги записва на физическия диск с известно закъснение. Това намалява броя на операциите с диска и подобрява ефективността.
  -
  - Файловете, за разлика от оперативната памет, са устойчиви обекти – те съществуват независимо от работата на процесите и се идентифицират по имена в йерархичната структура на файловата система.
  -

2. Опишете как се изгражда комуникационен канал (connection) между процес-сървер и процес-клиент със следните системни извиквания в стандарта POSIX: socket(), bind(), connect(), listen(), accept()



- S – server, Ci – client “I”. Сървърът предлага на останалите процеси да изградят връзки с него, като изпълнява предваритерните действия – socket, bind, listen;
- **Socket** – връща файлов дескриптор, който представлява гнездото за комуникация
  int sockfd = socket(domain, type, protocol);
- **Bind** – дава име на сокета в среда, в която и двата процеса имат видимост. Сървърът свързва своя сокет с конкретен адрес (IP + порт), чрез извикването `bind()`. Това придава "име" на сокета в комуникационното пространство, което позволява другите процеси (клиенти) да го намерят.
  int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
- **Listen** – сървърът извиква `listen()`, за да обяви сокета като "слушащ". Това го прави готов да приема входящи връзки. Той поставя сокета в пасивно състояние.
  int listen(int sockfd, int backlog); `backlog` – броят на чакащите заявки
- **Accept** – приема заявка за свързване, създава гнездо, връща нов файлов дескриптор
  int new_sockfd = accept(int sockfd);
- **Connect** – Това е активното действие по установяване на връзката – клиентът „пуска жица“ от своя сокет към сървърния адрес.
  int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

Тема 18

1. Опишете накратко **основните комуникационни канали** в ОС Linux. Кои канали използват пространството на имената и кои не го правят?

- **Неименуваната тръба** (pipe) се създава чрез системното извикване pipe(int fd[2]). То запазва в подадения масив два файлови дескриптора – fd[0] съхранява файловия дескриптор за четене, а fd[1] – този за писане. Тази тръба е видима само за процеса, който я е създал, и за неговите наследници, тъй като не е именувана – *не използва пространството на имената*.

- **Именуваната тръба** (FIFO) се създава чрез извикването mkfifo(…). За разлика от неименуваната тръба, този вид тръба *използва пространството на имената* и е видима за всички процеси в системата и може да бъде ползвана от тях – може да бъде използвана от процеси, които нямат роднинска връзка. Може да се отвори с open, но не знаем колко процеса си комуникират.
- **Връзката процес-файл**. Чрез системното извикване open(…) се създава файлов дескриптор, сочещ към края на комуникационния канал, който се изгражда в ядрото и отговаря за писане, а другият край на канала сочи към файла. Чрез извикванията read(..) и write(…) може да се чете и пише от/в този файл. Този вид канал също използва *пространството на имената*.
- **Конекцията** (изградена с механизма socket) е **именуван обект**, който се използва за адрес при изграждането на връзка с друг процес. Тук можем да разделим процесите на два вида – сървър (този, който обслужва останалите) и клиент. При конекцията не е необходимо процесите да са на една и съща система. Сокетът на сървъра *използва пространството на имената*, тъй като клиентите трябва да го виждат по някакъв начин, но клиентите не се обвързват с име – *не ползват пространството на имената*.

2. Опишете какви изисквания удовлетворява съвременна файлова система, реализирана върху блочно устройство (block device). Опишете предназначението на журнала на файловата система.
   - Една файлова система в наши дни трябва да има изисквания като:
     - **Цялостност на данните** - предотвратява загуба на информация при срив, прекъсване, грешки(журнал);
     - **Ефективност** - при записване, четене и търсене на данни, оптимизиран достъп чрез кеш;
     - **Управление на пространството** - да управлява дисковото пространство, включително разпределение на свободно пространство, управление на алокираното пространство и преодоляване на фрагментацията на данните;
     - **Сигурност** - контрол на достъпа до файловете и криптиране на данните;
     - **Поддръжка на множество операционни системи** - различни операционни системи, преносимост, съвместимост;
     - **Възстановяване след грешка** без загуба на важна информация;
     - **Скалируемост** - Тя трябва да може да управлява големи дискови пространства и да поддържа бърз достъп до информацията;

   - Твърд диск – това е диск разделен на много „пътечки“
   - Над повърхнината има устройство, което засича дали битът е 1 или 0;
   - Придвижването на главата на това устройство от една пътечка на друга е механично и става бавно;
   - Всяка пътечка е разделена на няколко сектора от по 512 или 1024 байта;
   - Главата не се допира до диска, а лети много малко над повърхността;
   - Софтуера знае колко е голям сектора и колко е времето за преминаване от един до друг сектор;
   - Файла се представя като много сектори;
   - Не са последователно разпределени байтовете на файловете, защото така не могат лесно да нарастват;
   - Файлът се разполага там където може, процесът на разхвърляне се нарича фрагментация.

   - Тема 17 – а

Опишете с по едно-две изречения работата на следните системни извиквания в стандарта POSIX:

pipe() – Създава неименувана тръба. Приема масив от две цели числа int fd[2], в който се съхраняват двата нови файлови дескриптора, отнасящи се до краищата на тръбата. fd[0] съхранява файловия дескриптор за четене, а fd[1] – за писане.

dup2(oldfd, newfd) – Създава копие на файлов дескриптор, подаден като първи аргумент(old) и го записва на **номера** на файлов дескриптор, подаден като втори аргумент(new). Ако вторият файлов дескриптор(new) е отворен, той първо бива затворен преди да бъде използван повторно. прави така, че newfd **да стане пълно копие на** oldfd, като:

- затваря newfd, ако вече е бил отворен;
- използва точно стойността newfd;
- и двата дескриптора сочат към един и същ **ресурс** споделят **позиция** и **флагове**.

fork() – Създава нов процес, копирайки извикващия процес. Не приема аргументи и връща идентификатор на процеса – новият процес (детето) има стойност 0, а извикващия процес (бащата) има стойност по-голяма от 0. При грешка, връща стойност -1.

exec() – Системните извиквания от това семейство заменят изпълнимия файл на процеса, който се изпълнява в момента, с друг изпълним файл, подаден като първи аргумент. Следващите подадени аргументи могат да се интерпретират като масив от аргументи, които желаната команда да приеме за своето изпълнение.

wait() – Спира изпълнението на извикващия процес, докато някой от процесите-дете не прекрати своето изпълнение. При подаден аргумент, съхранява подробна информация за детето, променило своето състояние.

waitpid() – Спира изпълнението на извикващия процес, докато процесът-дете с идентификатор, равен на подадения аргумент, не промени състоянието си.

open() – Отваря (или създава) файл с име, подадено като аргумент. Връща като резултат стойността на файловия дескриптор, асоцииран с този файл, а при грешка връща -1.

close() – Затваря файлов дескриптор, асоцииран с даден файл.

read() – Приема три аргумента – файлов дескриптор, буфер и брой байтове. Опитва се да прочете съответния брой байтове от файловия дескриптор и да ги запише в буфера.

write() – Приема три аргумента – файлов дескриптор, буфер и брой байтове. Опитва се да прочете съответния брой байтове от буфера и да ги запише във файловия дескриптор.

lseek() – Приема три аргумента – файлов дескриптор, отместване и отправна точка. Премества „курсор“-а във файловия дескриптор, на позиция <отместване> според отправната точка. Има три вида отправна точка – SEEK_SET – от началото, SEEK_END – от края, SEEK_CUR – от текущото положение.

socket() – Използва се за създаване на комуникационен канал от тип „конекция“. Създава сокет (крайна точка за комуникация) и връща файловия дескриптор, асоцииран с него.

bind() – Присвоява адрес на сокет, създаден със socket(). Това извикване се използва за сървърите в конекцията, изградена със socket(), а при клиентите не е необходимо. Тази операция се нарича “именуване на сокет”.

connect() – Свързва сокета, посочен от файловия дескриптор, подаден като аргумент, към адреса на сокета, подаден като аргумент.

listen() – Означава сокета, подаден като аргумент, като готов да приема нови входящи заявки за свързване.

accept() – Блокиращо повикване, което чака входящи връзки. След като заявката за връзка бъде обработена, нов файлов дескриптор се връща. Този нов сокет е свързан с клиента (установена е конекцията), а другият сокет остава в състояние listen в очакване на допълнителни връзки. Реално е цикъл.

Каква е спецификата на файловете в следните директории в Linux:

/etc – съдържа **конфигурационните** файлове на с-мата и програмите; информация за потребителите, пароли и т.н.; писане само от root

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

/var - **системни** файлове; за **споделени данни**

/boot – информация за ОС преди самото й стартиране; неща, свързани със **зареждането на операционната система**

/usr/bin – съдържа повечето **изпълними файлове**, които **не** са необходими за стартиране или поправяне на системата. предназначени за **обща употреба от всички потребители**

/home – **home директориите** на потребителите в системата; файловете на конкретните потребители

/usr/lib – **динамични библ заредени по време на изпълнение. Използват се от командите в /usr/bin**

/var/log – директорията, съдържаща всички log (журнални) файлове

/proc – информация за операционната система в **реално** време; съдържа **псевдо файлове** със **статистическа информация от ядрото**

/bin – съдържа **изпълними файлове**; **важни програми**, които се изпълняват при **стартирането на системата**, **общодостъпни** програми

/usr/doc – съдържа всичката необходима **документация**.

Опишете функционалността на следните команди в Linux:

cd – променя текущата директория

mkdir – създава директория

rmdir – премахва директория

cp – копира файл/файлове в посочена дестинация

mv – премества/преименува файлове в зададена дестинация

rm – премахва файлове

ls – показва съдържанието на директория

who – показва активните потребители в момента на извикване

find – търси файлове/директории по зададени параметри

ps – показва подробна извадка за процесите в ОС в момента на извикване

top – подобна на командата ps, показва подробна информация за процесите в ОС в реално време

vi – отваря на конзолата текстов редактор

tar – създава компресирани или некомпресирани архивни файлове или ги поддържа и модифицира

gcc – компилира програми, написани на C и C++ код

echo – показва на дисплея ред от текст/низ, подаден като аргумент/на стандартния вход

read – прочита ред текст, подаден на стандартния вход

test – проверява типове на файлове и сравнява стойности (числови или низове)

if – (if <условие>; then <списък команди> fi); изпълнява команди въз основа на подадени условия; ако подаденото <условие> е изпълнено, се изпълнява <списък команди>; тялото на конструкцията приключва с думата fi; в случай че първоначалното условие не е изпълнено, позволява проверка на допълнителни условия с ключовата дума elif – от else if; ако никое от подадените условия не е вярно, в else могат да се запишат команди, които да бъдат изпълнени.

for – (for <елемент> in <множество>; do <команди> done); изпълнява итеративно (циклично) набор от <команди> за всеки <елемент>, присъстващ в подадено <множество>

while – (while <условие>; do <команди> done); изпълнява итеративно (циклично) набор от <команди> докато дадено <условие> е изпълнено

chmod – променя правата на подадените файлове според подадените стойности (могат да бъдат представени както числово, така и символно)

cat – прочита съдържанието на файл и го извежда на стандартния изход

grep – търси редове, отговарящи на подаден образец (pattern), и извежда само съвпадащите редове на стандартния изход

cut – извежда само избрани секции/части от редовете на подаден файл/текст; разделя редовете на отделни колони спрямо подаден символ за разделител (по подразбиране е спейс/табулация)

sort – сортира редовете на подаден файл/текст и извежда сортираното съдържание на стандартния изход

wc – извежда броя на новите редове, думите и байтовете за подаден файл

tr – заменя символи в даден файл според подадено множество със съответните символи на същата позиция във второто множество или изтрива всички срещания на символите от първото множество