ОБЩА ТЕОРИЯ

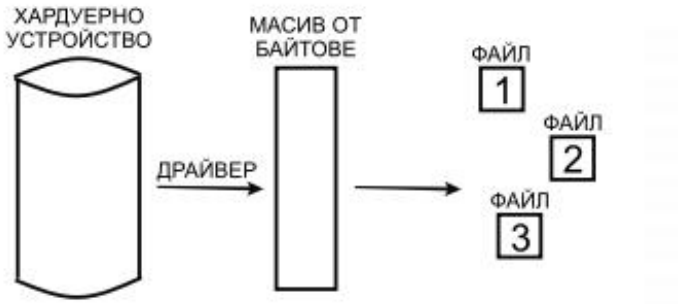
НЕФОРМАЛНИ ОПИСАНИЯ НА ОПЕРАЦИОННАТА СИСТЕМА:

Няма точно определение за операционна система. ОС е практическо използване на изчислителната система. Идеята на ОС е да предостави на програмистите и потребителите си (общо наричани „агенти“) прости и ясни абстракции и инструкции. Има 2 вида агенти: обикновени потребители – не се интересуват от имплементацията и работата на ОС; програмисти – за тях е важно ОС да предостави инструменти за разработка на софтуер. Други компютри, свързани в същата мрежа, също се считат за агенти. ОС трябва да осигурява диалог между всички хардуерни компоненти на системата.

ОСНОВНИ АБСТРАКЦИИ:

ОС е съвкупност от абстрактни структури:

- Драйвери = замяна на хардуера със софтуер; свежда физическата структура на хардуера до логическа (масив от байтове); ОС трябва да осигурява диалог между всички хардуерни компоненти на системата;
- Файлове = информацията в масива се представя като функционално обособени единици (файлове) – всеки файл се свързва с име/адрес, за да може да се осигури комуникация/адресация, и съдържание;



- Процес = абстракция на обработката на информация; работеща програма; процесът е програма, която е предназначена да работи на един процесор; след превеждане до машинен език, процесът става резултат от изпълнението на този машинен код от компютъра; Една ОС наричаме многозадачна, ако паралелно работят много процеси. Това дава възможност големите и сложни процеси да бъдат разбити на малки такива, които комуникират помежду си.
- Комуникационни канали = ОС може да се представи абстрактно чрез процеси, които комуникират помежду си. Тази комуникация се извършва чрез комуникационните канали; те са средства за комуникация между процесите и другите обекти;
- Сигурност = сигурност за изпълнение на отделните компоненти
    - Изолация = всеки процес работи самостоятелно, в собствено пространство;
    - Права на достъп (за многопотребителска ОС) = защита на информацията от неправомерно ползване – кой и как може да ползва информацията;

Частта от ОС, която реализира основните абстракции, можем да наречем ОС в тесен смисъл – ядрото (kernel), което се намира в RAM и поддържа тези абстракции. ОС в широк смисъл включва програми, които позволяват на потребителя да управлява компютъра, наричани потребителския интерфейс – GUI, Shell, API (application programming interface);

РАЗПРЕДЕЛЕНО ПОЛЗВАНЕ НА РЕСУРСИ:

Процесите се разпределят в RAM паметта, т.е. паметта е разделена (**разделяне на пространството**); Ако искаме N брой процеса да работят едновременно, но има <N брой процесора, тогава разделяме и процесорите(**разделяне на време**) – това се извършва в RAM паметта; Разликата между многозадачната ОС и времеделението е, че времеделението симулира многозадачност, докато многозадачната ОС изпълнява няколко задачи паралелно (пример: ос с 2 ядра: 2 процеса – многозадачност; 100 процеса – времеделене). **Разделяне на ресурса** (**мултиплексиране**) – технология на разделяне на средствата за предаване на данни

между група използващи ги обекти. В резултат от мултиплексирането в един физически канал се създава група логически канали.

ВИДОВЕ ОС ПО НАЧИНА НА РАЗДЕЛЯНЕ И ИЗОЛАЦИЯ НА РЕСУРСИТЕ:

Важно за една ОС е да осигури **сигурност** на отделните компоненти. Това понятие има два аспекта – **изолация** и **права на достъп**. Когато говорим за изолация, имаме предвид възможността на всеки процес да работи самостоятелно, в собствено пространство в паметта. Ако имаме няколко работещи процеса, не трябва на единия да му е разрешено писането в паметта на другия. Когато процесите са на машинен код обаче няма как да ограничим да работят само на едно място в паметта. За целта се използват хардуерни инструменти за защита на паметта (**protected mode**). Всеки процес, който върши някаква работа в потребителски режим, той работи и в защитен режим – ползва само част от паметта и не може да извършва операции с периферните устройства, а вместо това използва ОС като посредник.
В една изчислителна система може всички данни да са на един човек (например собствените ни компютри), но може една ИС да се ползва и от много хора (ел. поща) и трябва системата да се грижи да пази информацията на всеки потребител от другите. Такъв тип ОС се нарича **многопотребителска**. Терминът „**права на достъп**“ се въвежда с точно с тази цел – да ограничим достъпа до файловете за различните потребители. Има 3 вида права: за четене, за писане и за изпълнение
Една ОС наричаме **многозадачна**, ако паралелно работят много процеси. Това дава възможност големите и сложни процеси да бъдат разбити на малки такива, които комуникират помежду си.
Процесите се разпределят в RAM паметта, т.е. паметта е разделена (**разделяне на пространството**); Ако искаме N брой процеса да работят едновременно, но има <N брой процесора, тогава разделяме и процесорите(**разделяне на време**) – това се извършва в RAM паметта; Разликата между многозадачната ОС и времеделението е, че времеделението симулира многозадачност, докато многозадачната ОС изпълнява няколко задачи паралелно (пример: ос с 2 ядра: 2 процеса – многозадачност; 100 процеса – времеделене). Еднозадачната ОС за разлика от многозадачната, изпълнява само една задача в даден момент, а еднопотребителската – не различава потребителите.

КОМУНИКАЦИЯ И ОБЩО ПОЛЗВАНЕ НА РЕСУРС:

За разлика от хардуера, ОС изпълнява процесите последователно. За да е коректен един изчислителен процес, трябва да се поддържа синхронизация между паралелно работещите процеси (и хардуер). За да завърши правилно един изчислителен процес, трябва да се поддържа **синхронизация** между отделните процеси; Когато 2 или повече хардуерни устройства работят успоредно върху даден процес, те си сигнализират чрез **прекъсвания** (**timer interrupt**); Събитията, които управляват превключванията между различните процеси са входно-изходни; По време на изпълнение на код на даден процес, той може да бъде прекъснат в произволен непредсказуем момент;
Най-простия начин за взаимодействие на два процеса е, когато те се конкурират за достъп до общ ресурс – например обща памет, файл и т.н.; Пример:

1. П1 чете counter в своя локална променлива с1;
2. П2 чете counter в своя локална променлива с2;
3. П2 увеличава с2 с 1;
4. П2 записва стойността на с2 в counter;
5. П1 увеличава с1 с 1;
6. П1 записва стойността на с1 в counter.

И двата процеса увеличават стойността на counter с 1, но накрая counter е увеличен само с 1, а не с 2, т.к. изменението на П2 се губи; По този начин получаването на верен резултат зависи от реда, в който процесите се изпълняват(**синхронизацията**). Този проблем се нарича **race condition** (**състезание**);
Проблемът за избягване на състезанието е бил формулиран от Е. Дейкстра (E.Dijkstra) чрез термина критичен участък ***(critical section).*** Част от кода на процеса реализира вътрешни изчисления, които не могат да доведат да състезание. В друга част от кода си процесът осъществява достъп да обща памет или върши неща, които могат да доведат до състезание. Тази част от програмата ще наричаме критичен участък и ще казваме, че процес е в критичния си участък, ако е започнал и не е завършил изпълнението му, независимо от състоянието си. За избягване на състезанието и коректното взаимодействие на конкуриращите се процеси трябва да са изпълн ни следните условия: 1.Във всеки един момент най-много един процес може да се

намира в критичния си участък (взаимно изключване); 2.Никой процес да не остава в критичния си участък безкрайно дълго; 3. Никой процес, намиращ се вън от критичния си участък, да не пречи на друг процес да влезе в своя критичен участък; 4.Решението не бива да се основава на предположения за относителните скорости на процесите.
Инварианти на структурата = състояние на структурата бива нарушено временно, което може да доведе до загуба на данни; тези секции код, които временно нарушават структурата, се наричат критични секции. Решението е по време на изпълнение на критичните секции да бъде забранено действието на други процеси върху същата инварианта (**mutual exclusion – взаимно изключване**); Реализацията на взаимното изключване най-общо означава изчакване;
Пояснение: **Взаимно изключване** е механизъм, забраняващ два процеса едновременно да изпълняват код, който променя данни в споделена структура (ресурс); прилагането му е достатъчно условие да няма race condition, но не е необходимо условие, т.к. може да използваме едновременно различни инварианти в случай, че промените по тях са независими. Използват се няколко метода:
- В еднопроцесорните системи решението на race condition е командата disable/enable interrupt, която временно забранява прекъсванията. Така, ако преди критичната секция напишем disable interrupt, а след нея enable interrupt, процесът няма да може да бъде прекъснат.
- За многопроцесорна система се ползва допълнителен бит, който пази информация дали данните се ползват от някой процес/критична секция (1) или са свободни (0); този бит се нарича **lock**, а методът – **spinlock**; Инструкцията **test and set** (или **atomic swap**) трябва да е единна инструкция, която може да бъде прекъсвана (**атомарна инструкция**); по време на критичната секция не трябва да има рекурсия или извикване на друга програма, т.к. тя ще изпълни spin-lock, което вече е 1 и ще влезе в безкраен цикъл; аналогично трябва да се забранят и прекъсванията; Недостатък на spinlock е, че не пести процесора.
Следователно, кодът има следния вид:

```
- spin-lock:
disable_interrupt
R = test_and_set (lock)
if R = 0
   go to critical-section
else
   enable_interrupt
   go to spin-lock

- critical-section:
… критична секция …
spin_unclock: lock = 0
                    enable_interrupt
```

СИНХРОНИЗАЦИЯ ОТ ВИСОКО НИВО – СЕМАФОР:

Семафорът е механизъм за синхронизация от по-високо ниво, който може да бъде прекъсван, за разлика от spinlock. Семафорите се предоставят от ядрото на ОС.
block = обръщение/процедура на процес към ядрото (без аргументи), за прекратяване работата на процесора; за да заработи отново процесора, той трябва да бъде събуден (wake up – команда с аргумент номера на процеса, който събуждаме)
Идеята на семафора е да защити информационен ресурс, който има структура:

- **cnt** = брояч, който показва колко процеса могат да влязат в ресурса и да го ползват
- **L** = списък, в който когато cnt = -1, всички процеси на входа биват блокирани и записвани (изчакващи процеси); след като cnt = -1, |cnt| е броят на приспаните процеси (блокирани); ако cnt ++ , един от приспаните процеси се пуска да влезе;

Силен семафор е такъв, в който L се представя като обикновена опашка (т.е. преди всеки процес в опашката има краен брой други процеси). Слаб е такъв, в който списъкът не е опашка и може да има прежеждания

(например приоритетна опашка).

Семафорът (s) може да се разглежда като структура от данни, която има брояча cnt и списъка L за член-данни и 3 метода:

- За инициализация: s.init(cnt != 0) – смисълът на началната стойност е колко процеса могат да ползват едновременно ресурса.
- Заявка за вход към ресурса: s.wait
- Освобождаване на ресурса: s.signal()

Реализации на методите:

init(0):	wait():	signal():

cnt = cnt0;
lock = 0;
L = празно множ.;
	Spin.lock(lock);
	cnt = cnt – 1;
	if cnt < 0
	  L.put(self);
	  block();
	else
	  spin.unlock(lock);
		Spin.lock(lock)
		cnt = cnt + 1;
		if nct <= 0
		  pid = L.get();
		  wakeup(pid);
		spin.unclock(lock)

Пояснение: Семафорът и неговите данни са споделени данни. Той е обект, който няколко процеса използват, т.е. данните му се намират в памет, която е видима за всички процеси, които го ползват, и трябва да бъдат защитени от race condition. Тоест самият семафор представлява също критична секция. Тя обаче не може да се защити с друг семафор, защото и за него ще възникне същия проблем, затова единственото решение е да защитим семафора чрез spinlock => освен брояча и опашката за член-данни, въвеждаме и 1 байт lock.

Когато ресурсът е пълен и даден процес p1 заяви, че иска да го ползва чрез wait, тогава на p1 ще му бъде отнето управлението и той ще бъде приспан (не заема процесорно време). Когато някой процес p2, който вече е имал достъп до ресурса и го освободи чрез методът signal, тогава signal ще събуди вече приспания процес p1. Какво се случва с брояча cnt? В началото init му задава някаква стойност. Когато wait пропуска процеси към ресурса, броячът намалява с 1. В даден момент той ще стане 0 и всеки следващ процес, отправил заявление към ресурса ще бъде приспиван, като едновременно с това и броячът ще намалява (ще стане отрицателен). Когато се освободи място, броячът се увеличава с 1 и signal събужда някой приспан процес. А това кой точно се определя от вида на списъка L.
Важно е да се спомене, че процесите нямат информация за текущата стойност на брояча.
Гладуване (**starvation**) се получава, когато в L някои процеси са с по-висок приоритет от други по някакъв признак; така тези процеси с най-нисък приоритет могат никога да не се изпълнят и да се считат за убити!!!

ПРОСТИ ЗАДАЧИ ЗА СИНХРОНИЗАЦИЯ ЧРЕЗ СЕМАФОРИ:

Взаимно изключване на N процесора чрез семафори: **Семафорът mutex** има начална стойност на брояча 1. Всеки процес загражда критичния си участък с wait и signal. Ако един процес влезе в крит. си участък, останалите процеси се блокират в wait операцията преди своя критичен участък, т.к. броячът на Mutex вече е 0. Когато излезе от критичния си участък, процесът ще освободи някой блокиран процес, който ще влезе в своя критичен участък. Ако няма блокирани процеси, стойността на брояча на mutex ще стане 1. Следователно броячът на mutex може да приема само две стойности 0 и 1. Такъв семафор се нарича **двоичен**.
Свойства: Ако s.cnt = 1, следователно никой процес не изпълнява критичната си секция; ако s.cnt = 0, следователно точно един процес изпълнява критичната си секция.

Задачата „среща във времето“ (**randezvous**):
Имаме два процеса P и Q и съответните им инструкции p1, p2 и q1, q2. Искаме p1 да се изпълни преди q2 (p1<q2) и q1 да се изпълни преди p2 (q1<p2). За целта използваме два семафора s1 и s2:

semaphore s1, s2;
s1.init(0); s2.init(0);

P	Q
p1	q1
s1.signal()	s2.signal()
s2.wait()	s1.wait()
p2	q2

Ако P първи стигне до 2-рия си ред, ще подаде сигнал и броячът на s1 ще се увеличи с 1. Когато стигне до ред 3 ще изчака и броячът на s2 да стане 1, т.е. докато Q не изпълни s2.signal(); Аналогично, ако B започне първи;
Ако например разменим редовете 2 и 3 на Q също ще се изпълнят процесите, но тогава ще има две заспивания (т.е. ще е по-бавно общото изпълнение) в случай, че Q се изпълни пръв.
Ако разменим редовете 2 и 3 при P и при Q ще настъпи deadlock, т.к. процесите взаимно ще се изчакват, т.е. ще приспим и двата процеса.
Въпреки това, дори и ако разменим двата реда 2 и 3 при двата процеса, можем да избегнем deadlock, ако в началото инициализираме единият от семафорите с 1, т.е. нека s2.init(1); В този случай процесът Q винаги ще се изпълни първи, но тогава двата процеса няма да се срещнат във времето. Тогава това се нарича „**редуване във времето**“.

ОЩЕ ЗАДАЧИ ЗА СИНХРОНИЗАЦИЯ ЧРЕЗ СЕМАФОРИ:

Програмните канали (**pipes**) са механизъм за междупроцесна комуникация, при която процесите разменят данни. По-точно чрез тях се осъществява еднопосочно предаване на неформатиран поток от данни (байтове) между процеси, както и синхронизация на работата на тези процеси.

Има 2 вида програмни канали:

- **Неименован** (pipe)
- **Именован** (named pipe или FIFO файл)

Съществена разлика между тях е че именования програмен канал може да се използва от всеки процес, който знае неговото име и има право на достъп до него, а неименования може да се използва само от родствени процеси, т.к. той е достъпен само чрез файлов дескриптор.

Pipe: Използва се за създаване и отваряне на програмен канал. Прототип: int pipe (int fd[2]). При изпълнението му се създава нов файл от тип програмен канал, който се отваря два пъти – един път за четене и един път за писане. В аргумента fd[0] се връща файлов дескриптор за четене, а в fd[1] – за писане. Pipe връща 0 при успех, и -1 в противен случай. За да се реализира някаква комуникация, процесът изпълнил pipe трябва да създаде процес-син, който ще наследи неговите файлови дескриптори. След това тва процеса мога да комуникират.

Чрез семафори можем да реализираме и комуникационна тръба(pipe). Решението е предложено от Дийкстра чрез 2 семафора: имаме буфер-тръба с n байта и искаме паралелно няколко процеса да доставят байтове до други процеси. Ще има масив от байтове data[n] и два указателя l – за началото на масива, и h – за края на масива. Тогава единият семафор се инициализира с n: free.init(n), а другия с 0: used.init(0). Семафорът free има смисъл на празни места в буфера, а семафорът free – броят заети места в буфера.

ПРОБЛЕМИ ПРИ НЕКОРЕКТНА СИНХРОНИЗАЦИЯ:

Deadlock = когато два или повече процеса са едновременно приспани и всеки процес чака настъпване на събитие, което може да бъде предизвикано само от някой от блокираните(заспалите) процеси;
Кофман формулира 4 условия, нужни за настъпване на deadlock:
- взаимно изключване – всеки ресурс може да се ползва само от един процес
- в системата има поне един процес, който е приспан, задържа получени ресурси и чака предоставяне на допълнителен ресурс
- употребата на ресурси не трябва да се прекъсва
- цикличност (p1 чака ресурс, който се държи от p2; p2 чака ресурс, който се държи от p3 и т.н. …)

Възможност за възникване на т.нар. **гладуване**, може да настъпи когато семафорът не е силен, т.е. списъкът му не е обикновена опашка (FIFO). В този случай, ако например процесите в списъка се допускат на базата на приоритети, тогава някой процес може никога да не достъпи до ресурса, ако постоянно влизат нови процеси в списъка, които са с по-висок приоритет от него.

Задача за читатели и писатели – и двете групи – читатели и писатели, ползват общ ресурс – някаква стая например; когато няма писатели в стаята, може да има произволен брой читатели в нея; т.к. само читателите могат да правят промени по ресурса, то се допуска само този, който прави промените, т.е. само 1 читател;

За ресурс има 3 варианта: в него няма никой; в него има само един писател; в него има 1 или повече читатели;
Ако се допускаше само по 1 читател – чрез mutex семафор: semaphore m; m.init(1);

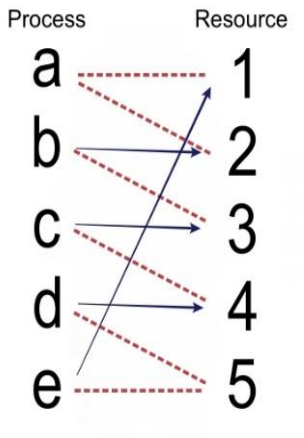
reader	writer
sem.wait()	sem.wait()
…чете…	…пише…
sem.signal()	sem.signal()

За повече от 1 читател: Идеята е първият читател, който влиза гледа дали стаята е свободна – ако е свободна влиза(и следователно броячът става 0). Следващите читатели вече не гледат семафора, а само дали има читател вътре – ако има, влизат. Обратното, когато излизат читателите, те не променят стойността на брояча. Само този, който излезе последен, променя стойността му на 1, за да сигнализира, че стаята е свободна. Тази техника се нарича light switch, т.к. има аналог със светването на лампа в някаква стая – първият човек, влязъл в някаква стая, светка лампата, а последния я гаси).

Решение: semaphore re.init(1), m.init(1), t.init(1);
int f = 0; //брой читатели в стаята
// re = room empty, m = mutex (защитава f, т.к. f може да се променя само от 1 читател в даден момент), t = за
// да не настъпи гладуване при писателите(т.е. да влизат само читатели и ресурса да не може да бъде
// освободен)

Readers:	Writers:
t.wait();	t.wait();
t.signal();	re.wait();
m.wait();	… писане/критична секция …
f=f+1;	t.signal();
if (f==1)	re.signal();
re.wait();	
m.signal();	
… четене/критична секция …	
m.wait();	
f=f-1;	
if (f==0)	
re.signal();	
m.signal();	

Задача Dining philosophers (философите и макаронитe) – 5 философа обядват на една кръгла маса, на която има 5 вилици и 5 чинии с макарониииии. Всеки философ иска да яде с 2 вилици. всеки философ (процес) трябва да вземе тази вилица (ресурс) отляво или отдясно, която е с по-малък пореден номер.
Един вариант е, ако всеки вземе първо вилицата от дясната си страна. Така обаче лявата вилица ще е взета и ще настъпи deadlock. Друг вариант е, когато някой хване дясната вилица и види, че лява е заета – да пусне дясната. Това няма да доведе до deadlock, но има вероятност от гладуване, т.к. ако някой философ вземе двете вилици, другите двама от двете му страни ще останат блокирани за неопределено време. Налице са 4-те условия на Кофман за получаване на deadlock.

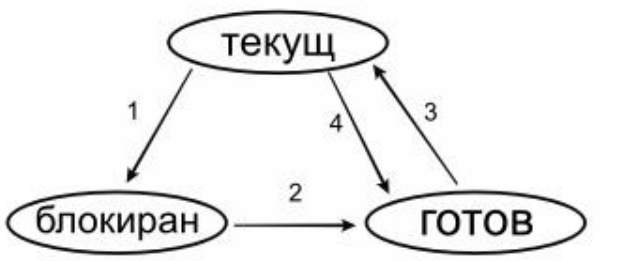


Решението е да се преномерират ресурсите – т.е. ако се взимат в нарастване на номерата няма да настъпи deadlock.

Но *a* и *e* искат да вземат 1. Допускаме, че *e* взима 1. Тогава *a* няма да има достъп до нито един от двата ресурса. Следователно *a* чака за 1 и 2, *b* чака за 3, *c* чака за 4, *d* и *e* чакат за 5. Следователно или *d* или *e* ще вземе 5, следователно или *d* или *e* ще се изпълни, т.е. няма да настъпи deadlock. (въпреки че може да настъпи гладуване)

ПРОЦЕСИ И ТЯХНОТО УПРАВЛЕНИЕ и СЪСТОЯНИЕ НА ПРОЦЕСА И ПРЕХОДИ МЕЖДУ ТЯХ:

В ОС има много процеси, но в даден момент централния процесор изпълнява точно един от тях. Всеки процес се намира в едно от трите състояния: текущ(running) = изпълнява се в момента от централния процесор; готов(ready) = логически може да се изпълнява, но централният процесор е зает с друг процес; блокиран(blocked/sleeping) = чака настъпване на някакво събитие, най-често завършване на вх.-изх. операция; zombie(крайното състояние на всеки процес) = процесът е завършил, но все още не е изхвърлен от системата – неговият баща може все още да получи инфо за него. Във всеки един момент точно един процес е текущ.



Създаването на процес го вкарва в опашката Ready. Преход 1 настъпва, когато текущия процес не може да продължи работата си по някаква причина. Когато причината (проблемът) е разрешен, изпълнява се преход 2 за процеса. Преходите 3 и 4 се осъществяват от **планировчика** (**scheduler**). Например той може да смени състоянията на два процеса между 3 и 4, ако текущият процес е работил твърде дълго време. Тоест, когато изтече кванта време, който е предоставен на процеса, той ще бъде изхвърлен насила в Ready от прекъсването на таймера. А кой процес след това ще влезе в Running зависи изцяло от ОС – когато CPU се освободи, той се заема от ядрото. Това се нар. **системно време.** Решението кой процес да премине към Running се взима от т.нар **диспечър**.
Убиването на процес може да стане само, ако процесът е в Running режим.

Под **превключване на процес** (**context switch**) се разбира смяна на контекста на текущия процес с контекста на друг процес. Това става при една от следните 3 ситуации:

- текущият процес преминава в състояние Блокиран
- текущият процес преминава в състояние Зомби
- текущият процес преминава в състояние Преразпределен – планировчикът/диспечерът е решил да го свали.

Класове процеси:

- I/O-bound interactive foreground – прави малко изчисления; предаване на команди; няма прекъсвания;
- SPU-bound background – малко вх.-изх. дейности извършват; нужда от време за изчисление;
- Real-time deadline – по-строги изисквания; приложения, комуникиращи с околния свят; нужда от процесорно време; срок за изпълнение на процес (например таймер);

Класовете 1 и 2 се основните класове процеси, които ОС използват.
Система за времеделение = създава илюзия за паралелно работене на процесите.
Най-простия алгоритъм е **Round-robin scheduling**:

- Нарежда процесите в обикновена опашка (в състояние ready); Няма приоритети;
- Пред определен интервал от време се разменят процеси между ready и running;

Плюсове: не може да настъпи гладуване. Минуси: интерактивните процеси от клас 1 при натрупване няма да могат да се изпълнят (бързо), а real-time процесите ще изостават или избързват, т.к. тяхното време за

УПРАВЛЕНИЕ НА ПАМЕТТА:

Съвременните КАРХ са симетрични, т.е. няколко процесора или ядра работят с една обща памет, но се очаква в бъдеще да станат асиметрични (архитектури от клас О), където всеки процесор ще има различни времена на достъп до различни памети.

Важен момент при обработката на информация от един процес е, че то той трябва да я съхранява. Следователно, освен процесор, се ползва и памет.
Когато говорим за процесор, спящите процеси не го ползват, но когато става въпрос за памет – всички процеси я използват, защото независимо дали даден процес спи, той трябва да пази данните си.
Следователно наличната памет трябва да е разпределена измежду процесите в системата.

Физически аспекти на реалната памет:

- енергозависимост – важно за файловете при изключване на компютъра е те да се запазят; по-бързите памети са енергозависими, за разлика от по-бавните;
- многократно ползване – краен/безкраен брой записи; краен брой имат флашките например, а безкраен – RAM паметта.

Как се представя паметта на работещи процеси? Най-важното изискване е локалните данни , с които работи процесът, да бъдат защитени. Т.е. те да са собственост единствено на процеси и други процеси да нямат достъп до тях. За тази цел самите процеси трябва да работят в защитено пространство. Съвременните процесори имат няколко режима на работа: **real mode** (режимът, в който работи самото ядро; програмата на ядрото има достъп до всички ресурси на системата), **protected mode** (режимът на работа на потребителската програма; няма достъп до всички ресурси, а само до този, който й е предоставен). Паметта, която ползва потребителската програма е означена с два регистъра – един за началото на сегмента(f) и друг за дължината му(s). Всяко четене и писане в паметта се проверява дали адресът попада в позволения ни сегмент:
f <= address < f + s и ако тази проверка върне false, значи програмата нарушава областта на памет, която й е заделена (**segmentation fault**). Когато работим в real mode, тази проверка не се прави.

Разделянето на паметта на страници (**paging**) е по-известно като виртуална памет. Идеята на страниците е точно да обособи отделни сегменти от паметта и да им осигури защита. Ако реалната памет е някакъв физически блок, виртуалната памет е абстрактното понятие, която за процеса изглежда като единен масив, приличащ на RAM-а. Но няма съответствие между адресите. Виртуалната и реалната памет са разделени са разделени на равни по размер страници и на всяка страница от виртуалната памет е съпоставена страница от реалната. Има вариант на някои страници да не е съпоставено нищо. Може да има и страници, които се ползват едновременно от няколко процеса, стига тези данни да са за четене, но не трябва няколко процеса на пишат на едно място, т.е. трябва да има и тагове за това как ще се ползва страницата. Може да има и страници извън RAM-а в swap-а, ако RAM-а не стига. Техниката е въведена първо при IBM 360. За всяка страница се поддържа битова маска, която показва какви са свойствата на тази страница. На всяка страница се съпоставя адресът й и таг - той съдържа информация дали нашия процес може да ползва в паметта, дали е негова собственост.
(((Всяка страница съдържа offset и p-битове. Адресацията към страницата става като процесорът подаде на RAM offset-а, а чрез битовете в p ще намери тага на страницата – той съдържа информация дали нашия процес може да ползва в паметта, дали е негова собственост. Таблицата с всички тагове в малка и образува нещо като микропроцесор, който се намира в CPU.)))

Предимството на този метод е, че можем да направим някои части на паметта, достъпни за всички програми и ще имаме достъп само за четене.

Недостатъци: Ако използваме абсолютни адреси при стартиране на програмата, тя трябва да знае точно в кой интервал от паметта работи или да знае точните физически адреси, които ще използва. Друг проблем при сегментацията е, ако програмата ни иска да работи с по-голям обем динамично заделена памет. Тогава може да настъпи конфликт.

ВИРТУАЛНА ПАМЕТ – РЕАЛИЗАЦИЯ:

В CPU съществува таблица, която описва всички страници – къде се намира и какви свойства има. На всяка страница се съпоставя ключ <i, pid> = <№ страница, № процес>; Чрез този таг се намира адресът на страницата и нейния таг в реалната памет. Друг вариант за ключа е <Zi, tag> = <адресът на страницата, таг>, където i е i-тата страница. Вторият метод се нарича пряка адресна таблица и за всеки процес имаме по един такъв масив.

Работата с тези таблици се осъществява чрез хардуер, наречен MMU. Процесорът поддържа таблица, която на всяка страница от паметта съпоставя реална физическа страница и съответния й таг. Тези страници са големи и не се разполагат в самия CPU, а в RAM. Има специален регистър, който сочи къде е тази таблица.

Как се извършва адресацията: Изчислява се адрес на клетката в паметта – този адрес е съставен от две части: адрес на страницата (virtual page address = vpa) и отместване (offset). За да работи правилно програмата, тя трябва да знае къде се намира тази информация в реалната памет – следователно трябва да се чете таблицата. Това четене се извършва от хардуера. Ако при четенето всичко е наред, хардуерът ще замени този адрес с друг, която се нарича real page address (rpa), но offset-а ще си остане същия. Следва да се извърши четенето/писането.

Следователно потребителската програма ще работи 2 пъти по-бавно, ако е в защитен режим. Решението на този проблем е таблиците да се кешират. Друг проблем е, че тези таблици използват 15% от RAM.

Има 3 вида Cache в зависимост от дейността, която извършва:

- кеш за инструкции
- кеш за данни
- кеш за таблиците за управление на виртуалната памет

По номера на таблицата ще разберем къде в адресната таблица да идем и от там къде е реалния адрес. Ще кешираме части от адресната таблица. Името на тези кешове е TLB. Дължината на една такава таблица е 32 бита. В тази таблица на i-тата позиция имаме номер на страница и съдържание. Единият начин да ги намираме е като получим номера на страница i е да го сравним със стойностите в тази таблицата и ако съвпада, да извлечем стойността. Когато данните са в кеш паметта имаме hit, а когато не са – miss, и тогава трябва да се вземе решение коя да се изхвърли и да се зареди в кеш следващата данна. Този метод обаче е бавен. Най-бързо е паралелно да се търси (асоциативен кеш). Сравнява се дали i=i0, i=i1, … и ако някъде е вярно, подава се сигнал. TLB най-често се реализират именно с асоциативен кеш.

СИНХРОНЕН И АСИНХРОНЕН ВХОД/ИЗХОД:

При синхронната входно-изходна операция процес може да бъде блокиран/приспан при системно извикване. Същевременно, при нормално завършване, потребителския 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.

Докато при асинхронната – процесът може и да не се блокира/приспива. В този случай, при невъзможност да се комплектова 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/процеси, без да бъде блокиран.

Пример: към WEB-browser постъпват поне 3 входни канала – от мишка, клавиатура и данни от интернет. Браузърът чете асинхронно от тези канали; За да обслужва паралелно сървър в Интернет клиентите си, той трябва да ползва асинхронни операции, за да види по кои връзки тече информация и кои са пасивни.

ТЕМИ, СПИЦИФИЧНИ ЗА LINUX/UNIX

ТЕКСТОВА КОНЗОЛА – SHELL

Основни принципи в LINUX/UNIX:

- многопотребителска система – потребителят трябва да има акаунт; потребителят трябва да се log-не; пълно разделение между потребителските файлове и конфигурирането на системата
- малки компоненти – всяка компонента изпълнява една проста задача; верига от компоненти за по-сложни задачи

Login prompt:

- log in/log out
- може да бъде текстов (shell) или графичен (GUI); затова shell се нарича още команден интерпретатор (command-line interface) – най-популярният команден интерпретатор е bash;

Процесите по подразбиране са свързани с 3 комуникационни канала(файлове) -stdin, sdtout, stderr. Stdin е свързан с клавиатурата, а другите два с монитора. Под пренасочване разбираме свързване на трите стандартни файла с други файлове. За всеки файл, отворен от някой процес, ОС създава файлов дескриптор – цяло, неотрицателно число. За стандартните файлове, дескрипторите са 0,1 и2. А пренасочването им става съответно с „<команда> < <файл>“, „<команда> > <файл>“ (или >>), „<команда> 2> <файл>“ (или 2>>). Конвейерът е конструкция, която също ползва пренасочване на входа и изхода (cmd1 | cmd2 |…). При него ст.изход на cmd1 се насочва към ст.вход на cmd2; за cmd2 аналогично към cmd3;

Основни недостатъци в UNIX са, че няма диалог с потребителя и има много опции.

Файловата система в LINUX представлява кореново дърво с два вида върхове:

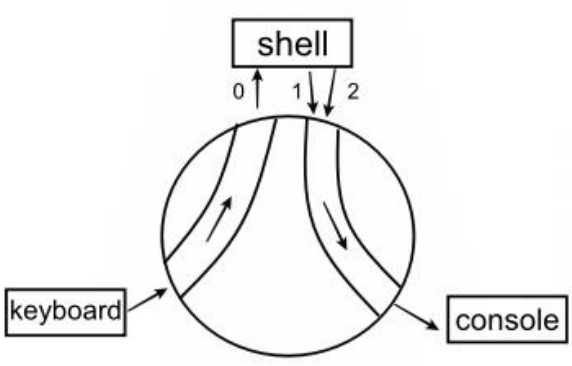
- Файлове (винаги листа в дървото)
- Директории (техни наследници могат да са както файлове, така и други директории)

Главната директория (коренът) е “/”; Въпреки това към дървото можем да добавяме цели дървета, като техния корен отива на мястото на някое листо в базовото дърво и така се симулира друг диск или инсталиране на някакво периферно устройство;

SHELL – КОНВЕЙЕРИ, ПРЕНАСОЧВАНЕ, ФИЛТРИ

Знаем, че структурата на процесите в LINUX представлява кореново дърво, където коренът – първият процес init се пуска от ядрото. Всеки следващ новосъздаден процес наследява няколко неща от своя родител: обкръжение env (списък от променливи и стойности); списък от аргументи args; 3 файлови дескриптора по подразбиране – stdin, stdout, stderr, които представляват цифра, сочеща към реалните канали, предоставени от ядрото, за пренос на данни.

Stdin е свързан с клавиатурата, а другите два с монитора. Под пренасочване разбираме свързване на трите стандартни файла с други файлове. За всеки файл, отворен от някой процес, ОС създава файлов дескриптор – цяло, неотрицателно число. За стандартните файлове, дескрипторите са 0,1 и2.



Пренасочването им става съответно с „<команда> < <файл>“, „<команда> > <файл>“ (или >>), „<команда> 2> <файл>“ (или 2>>). Конвейерът е конструкция, която също ползва пренасочване на входа и изхода (cmd1 | cmd2 |…). При него ст.изход на cmd1 се насочва към ст.вход на cmd2; за cmd2 аналогично към cmd3; В линукс тази връзка се

реализира чрез програмен канал (**pipe**) за еднопосочна комуникация между процесите с цел разменяне на данни. Когато каналът е двупосочен, се нарича **конекция**. За да се осъществи конекция между два процеса, които нямат роднинска връзка, се използва друг комуникационен обект, който се нарича **socket**. Той е генератор на конекции.

Филтри:
cat [f1…fN] = изкарва съдържанието на файловете; ако някой не е readable – пропуска го; само cat – адресът на файла се въвежда от конзолата;
grep <рег.израз> [f1…fN] = търси рег.израз в съдържанието на файла и извежда редовете, в които се среща; **-h** не показва името на файла, -**n** на кой ред се намира, **-v** редовете, в които изразът не се среща, -w търси само думи, а не и подстрингове;
cut **-c/d/f** n f1 = изрязва първите n символа/разделител/поле от всеки ред на подадения вход (файл) Пр: cut -c 2,3,4 f1 = взима 2,3 и 4-тия символ на всеки ред (може и „2-4“);
sort f1 = сортира f1, но не запазва сортирането във файла;
wc [-c/w/l] [f1…fN] = брой символи/думи/редове във файла; без опции изкарва име на файл, бр. редове, думи и символи * wc -c f1 = размер на файла f1 в байтове (За всеки символ по един байт);

SHELL – ПРОГРАМИ (СКРИПТОВЕ)

Shell е интерпретатор, т.е. когато му дадем да изпълни програма, той я разглежда като обикновен текст и започва символ по символ да го анализира и да изпълнява инструкциите, които текстът указва.

Тази програма(текст-указания) може да бъде прочетена по 3 начина – от файл, от стандартния вход и чрез компилация на изпълним файл.

Командата **echo** <args> → извежда аргументите на екрана
Пример: name = Ivan
echo name → name
echo $name → Ivan #чрез $ взимаме оценката на аргумента

Командата **read** [име на променливи] → чете се един ред от стандартния вход и се разделя на думи (разделителят е интервал) като всяка дума се съпоставя на съответната променлива
Пример:
read a b c → въвеждаме a b c
read a b c → въвеждаме a b → a b се променят, c остава празен
read a b c → въвеждаме 1 2 3 4 → $a=1, $b=2, $c=3 4

Изход на команда като стойност на променлива: чрез **`cmd`** или **$(cmd)**
Пример:
path = $(pwd)
size = `wc -c file1`
num = $(ps aux | grep $(whoami) | wc -l)

Командата **test** <израз> → командата не връща нищо, но има код на изход – 0 за истина(успешен тест), 1 за лъжа(неуспешен тест); взимаме кода чрез $.

Сравнение на цели числа:
gt → >
lt → <
ge → >=
le → <=
eq → =
ne → !=

Сравнение на низове:
-z → проверка за празен низ
-n → проверка за непразен низ
= → равни низове
!= → различни низове

Проверки за файлове:
-f → файлът съществува
-s → файлът не съществува
-d → файлът е директория
-z → файлът може да се чете
-w → във файла може да се записва
-x → файлът може да се изпълнява

Логически оператор:
! → отрицание
-a → И
-o → ИЛИ

Оператор if:
if <условие/команда>
then <команда2>
[elif <команда 3> then <команда 4>]
…
[else <командаN>]
fi

Пример:
```
read num1 num2
if [$num1 -eq $num2]
then echo “Numbers are equal”
elif [$num1 -lt $num2]
fi
```

Оператор while И for:
while <условие/команда>
do
команда1
команда2
…
done

for <променлива> ***in <***списък>
do
команда1
команда2
…
done

Пример:
```
read user
until who | grep $user
do
echo “Waiting for user $user to login”
sleep 10
done
```

Break = прекъсва цикъла
continue = приключва текущото изпълнение на тялото на цикъла и започва да го изпълнява отново със следващата стойност на променливата

Изпълнение на командна процедура: 1) $ <има не командната процедура> 2) $ bash <име на командната процедура> 3) bash path_to/<има не командната процедура> 4) bash -x path_to/<има не командната процедура> (дава ни допълнително инфо кои команди се изпълняват) 5) добиваме права за изпълнение първо: chmod a+x <има не командната процедура> → директно я извикваме: <има не командната процедура>, ако се намираме в същата директория 6) ./ <има не командната процедура> , ако се намираме в същата директория

Позиционни параметри: Имената на позиционните параметри се изписват по следния начин: $0, $1, $2, …, ${10}, … , където $0 се използва за името на файла с командната процедура, а $i - за i-тия аргумент на процедурата, зададен след името на файла (при номер на аргумента по-голям от 10, номерът се загражда в {} скоби).
Тоест:
echo $0 → #comproc_name;
echo $1 → #param1; …

set arg1 arg2 … argN = даване на стойности на позиционните параметри; ако отново използваме set, тогава променливите ще запазят новите стойности
shift n → преместване стойностите на позиционните параметри с n позиции наляво

Запазени стойности на позиционните параметри:
$* = дава символен низ с всички параметри (без нулевия) от командния ред
$@ = дава масив с параметрите от командния ред (без нулевия) като: # “$1” “$2” …
$# = дава броя на параметрите от командния ред
$? = код на завършване на последния завършил процес
$$ = PID на текущия процес
$! = PID на последния завършил процес във фонов режим

Vi, TAR, g++

Текстовите редактори са интерактивни, т.е. те едновременно четат подадената информация от входа (файла) и комуникират с потребителя; Стандартният конзолен редактор в UNIX е VI. Той работи само с клавиатуарата
vi [нов файл/съществуващ файл] → стартиране; при без аргументи, указването на името на файла става при записване
Режим промяна: винаги при стартиране на vi; Може да се изпълняват команди на екрана, чрез които да копираме, променяме, заместваме, изтриваме и т.н. + движение на курсора
Въвеждащ режим: Влизане в режима чрез командата **a + _** , Излизане чрез **[ESC]**, стъпка назад чрез **u**;
Операции: **a** = въвеждане след курсора, **i** = въвеждане на мястото на курсора, **A** = въвеждане в края на реда, **I** = въвеждане в началото на реда, **o** = добавя нов празен ред след текущия, **O** = добавя нов празен ред преди текущия;
Движение на курсора: **k** = премества курсора нагоре, **j** = надолу, **h** = наляво, **l** = надясно, ^ = към първия символ на текущия ред, **$** = към последния символ на текущия ред….;
Търсене в текст: **fc** = намира символа c; **/низ** = търсене на низ надолу от текущия ред; **?низ** = нагоре от текущия ред; **n** = търсене на следващия резултат в същата посока; **N** = търсене на следващ резултат в обратната посока;
Запис и изход от vi: **:w** = записва промените във файла; ако сме отворили vi без указване на име, го указваме сега; **:q** = излиза от vi; **:q!** = излиза от vi без да записва промените; **:wq** = записване и изход; **:e file** = започва редактирането на нов файл; **:e! fail** = започва редактирането на нов файл без да записва промените в стария; **:r file** = прочита файла file и добавя съдържанието му след курсора; **:!cmd** = изпълнява команда от средата (например :!pwd, за да видим къде сме във файловата система)
Забележка: можем да записваме само в текущата дир.; ако искаме в някоя друга :w <абс. име на файл>

ПРОГРАМИ ЗА РАЗГЛЕЖДАНЕ И ТЪРСЕНЕ

ls [-опции] [аргументи] = връща списък на файловете и директориите в текущата директория; опции: **-l** в подробен формат, **-a** показва и скрити файлове, **-t** сортира файлове по дата на създаване, **-r** в обратен ред, **-S** сортирани по размер, **-d** само директории (всеки елемент има 10-байтово поле за достъп, като първият байт е –(ако е файл) или d(ако е директория), останалите са 3x3 потребител, група, всички и приемат стойности r/w/x/-); операциите могат да се комбинират: например ls -tlr изкарва файловете сортирани по дата в обратен ред в подробен формат

who = списък на хора в сървъра
who -u = показва дали даден потребител е активен (чрез някаква точка)
whoami = връща текущия потребител (може и who am i)

find <къде> <какво> = <къде>: **/** (цялата файлова система), **.** (текуща директория), конкретен адрес; <какво>: -name f1, user x81808, size n, group student, exec cmd (exec cat {}\)

ps = показва всички процеси на текущия потребител; опции: **-a** процеси на всички потребители, **-u** [име] собственик на процеса, **-x** показва и процесите не свързани за даден терминал
ps aux = инфо за всички процеси на всички потребители (т.е. процесите в системата)
kill [опции] [pid] = приемр за опция: -9 безпрекусловно убийство
& = за създаване на процес на фонов режим

ФАЙЛОВА СИСТЕМА

При процесите делим паметта между тях и стига да има достатъчно, даваме на всеки колкото му трябва. Те си работят в своите си области и не е нужно да ги именуваме, докато при файловете, блоковете памет, които им съответстват ще съществуват дълго във времето и ще бъдат ползвани от различни агенти. Другата разлика е, че, когато един файл е в своето пасивно състояние, цялата информация в него трябва да е в памет, която е устойчива и е енерго независима. Стремим се към вечно съхранение. Когато променяме информацията във файл, той трябва да е свързан с процес, който го променя и в този случай може да се окаже, че той е в RAM паметта.

Файловете са първата сложна абстракция в правенето на ОС. В съвременната ОС има два слоя на абстракция: пространството на имената(видимо за потребителя) и реализацията(как ще бъдат разположени обектите в хардуера; конкретната реализация на всеки информационен обект).

В UNIX абстрактната файлова система представлява кореново дърво с два вида върхове

- Файлове (винаги листа в дървото)
- Директории (техни наследници могат да са както файлове, така и други директории)

Главната директория (коренът) е “/”; Въпреки това към дървото можем да добавяме цели дървета, като техния корен отива на мястото на някое листо в базовото дърво и така се симулира друг диск или инсталиране на някакво периферно устройство;

Видовете на файловете в текущата директория можем да намерим чрез операцията ls -l; Първото нещо, което ни дава е видът на файла и веднага след него правата на достъп. Видът се определя чрез „-„, ако е файл, и „d“ – ако е директория. Можем да срещнем и виртуални в/и устройства, означени с „c“ за **символно устройство/character device** (устройство, с което можем да четем/пишем байтове последователно, като поток - например принтер) или „b“ за **block device** (външни масиви от байтове). Може и „p“ за именувана тръба, „s“ – крайна точка за комуникация.

Името, което идентифицира обекта е пълното(абсолютно) име. Изписвайки само име на файл, ще имаме предвид файла с това име само в текущата директория. За разлика от името на файла, абсолютното име е уникално и единствено.

Името, което съдържа само . е указател към текущата директория; .. към родителската;

Всеки файл има характеристики: пълно име; съдържание (масив от байтове); права на достъп – read, write, execute за 3 вида групи: самия потребител, някаква група от агенти, всички останали; дата на създаване; размер на файла; колко link-а има към файла (т.е. файлове със същото име)

Формално файловете представляват масив от байтове; всеки байт варира от 0 до 255, а символите са букви

от азбуката, с която работим; съответствието между букви и байтове се нарича кодова таблица и пример за кодова таблица е ASCII – стандарт за буквите на латиница от о до 127.

mkdir dir1 [dir2…dirN] = създаване на директория
rmdir dir1 [dir2…dirN] = триене на директория
rm [-i,r] dir1 [dir2…dirN] = -i пита за всеки файл дали да бъде изтрит; -r = трие всичко от dir1 надолу по дървото
cp f1 f2 = копира файла f1 във файл с име f2; ако f2 съществува – замества го, ако не – създава го
cp f1 f2 … fN dir = копира f1 … fN в dir
mv f1 f2 = прехвърляне (cut)
mv f1 … fN dir
mv dir1 dir2

cat [f1…fN] = изкарва съдържанието на файловете; ако някой не е readable – пропуска го; само cat – адресът на файла се въвежда от конзолата

МНОГОПОТРЕБИТЕЛСКА РАБОТА:

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.

Командата **chmod** има следния синтаксис: *chmod* [*ugoa][+-=][rwx] файл1 файл2 … файлN*. Тя се използва за промяна на кода на защита на указаните файлове. Самата промяна става чрез задаване на опции (без да се пишат тирета):
- опция u указва промяна на правата на собственика на файла, опция g указва промяна на правата на групата на собственика, опция o указва промяна на правата на всички останали потребители, опция a указва промяна на правата на всички потребители;
- + означава добавяне на права, - означава отнемане на права, = означава установяване на нови права;
- r е право за четене, w е право за писане, x е право за изпълнение.

Възможно е кодът на защита да бъде зададен чрез осмично число с три цифри – първите 3 бита са r, w, x правата за собственика, следващите 3 бита са r, w, x правата за групата на собственика, следващите 3 бита са r, w, x правата за останалите потребители. Всъщност в кода на защита има още 3 бита, но засега няма да ги разглеждаме.
Пример: *chmod u+x proc* добавя право за изпълнение на файла proc за собственика на този файл.

ФИЗИЧЕСКИ ФАЙЛОВИ СИСТЕМИ:

Блочни и символни устройства:

Блочни специални файлове или **блочни устройства** предлагат буфериран достъп до хардуерни устройства и предлагат някаква абстракция 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 ядрото в истинското устройство или в какъв ред 2 отделни записа ще пристигнат във физическото устройство.

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

Блочното устройство комуникира 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 дялове 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 свързва 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 разкачена 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.

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.

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

ФАЙЛОВА СИСТЕМА:

/bin – 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

API, POSIX – РАБОТА С ФАЙЛОВЕ

Отваряне/създаване на файл:

#include <fcntl.h>

int open (const char *filename, int oflag [, mode_t mode]); = връща файлов дескриптор или -1 при грешка

Oflag: O_RDONLY, O_WRONLY, O_RDWR, O_CREATE (не връща нищо, ако файлът съществува; ако не съществува, трябва да се въведат и права за достъп), O_EXCL (използва се с O_CREAT; връща грешка, ако файлът съществува; ако не съществува, трябва да се въведат и права за достъп), O_TRUNC (старото съдържание на файла се изтрива), O_APPEND (новото съдържание се добавя в края на старото).

Ако има повече от един флаг, те се разделят чрез “|”, а правата за достъп се отделят cys “, “

Четене от файл:
#include <unistd.h>
ssize_t read (int fd, void *buf, size_t nbytes); = връща броя на реално прочетените байтове, иначе -1; при край на файл – 0 (fd е номер на файлов дескриптор, buf е указател към областта на програмата, където се записват данните, nbytes указва броят на байтове за четене)
Писане във файл:
ssize_t write (int fd, void *buf, size_t nbytes); = връща броя на реално записаните байтове, иначе -1;
Позициониране във файл:
off_t lseek (int fd, off_t offset, int whence);
(offset е самото изместване, whence указва как се интерпретира отместването: SEEK_SET премества указателя от началото до offset-байта; SEEK_CUR премества указателя от сегашната му позиция до offset-байта; SEEK_END премества указателя от края на файла (отзад-напред) до offset_байта (отрицателно число))
Затваряне на файл:
int close (int fd);

API, POSIX – РАБОТА С ПРОЦЕСИ И ТРЪБИ

#include <sys/types.h>
#include <unistd.h>
pid_t fork(void); Връща -1 при грешка, 0 в процеса-дете и x>0 в процеса родител (x е pid на процеса-дете)
Получаване на pid на процес-родител и процес-дете:
pid_t getpid (void) – връща pid на текущия процес.
pid_t getppid (void) – връща pid на процеса-родител на съответния процес.
Завършване на процес:
void exit (int return_code); указва на процеса-родител с какъв код е завършил процесът.
Системен примитив wait:
#include <sys/types.h>
#include <wait.h>
pid_t wait (int * status); процесът-родител изчаква първото свое процес-дете да приключи; връща pid на детето; status дава кода на завършване
pid_t waitpid (pid_t pid, int * status, int options); указваме на родителя кой специфичен процес-дете да изчака (pid указва кой процес-дете на изчака; чрез options може да се предотврати блокирането на родителя - чрез WHOHANG процесът-родител не се блокира, ако синът не е завършили функцията връща 0)
Смяна на образ на процес exec():
използва се за промяна на програмата, която процес изпълнява. При успех не връща нищо, иначе връща -1.
#include <unistd.h>
int execv (const char *path, char *const argv[]); - приема пътя на изпълнимия файл
int execvp (const char *file, char *const argv[]); - приема името на изпълнимия файл и го търси в директориите, които са зададени в PATH
int execl (const char *path, const char *arg, …);
int execlp (const char *file, const char *arg, …);

Пример: искаме да изпълним команда ls, с опция -l
execl("/bin/ls", "ls", "-l", 0);
**Особености: 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.

Pipe: Използва се за създаване и отваряне на програмен канал. Прототип: int pipe (int fd[2]). При изпълнението му се създава нов файл от тип програмен канал, който се отваря два пъти – един път за четене и един път за писане. В аргумента fd[0] се връща файлов дескриптор за четене, а в fd[1] – за писане. Pipe връща 0 при успех, и -1 в противен случай. (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). За да се реализира някаква комуникация, процесът изпълнил pipe трябва да създаде процес-син, който ще наследи неговите файлови дескриптори. След това двата процеса мога да комуникират.

dup2(oldfd, newfd)- makes newfd be the copy of oldfd, closing newfd first if necessary
dup() - получаваме копие на първия свободен файлов дескриптор

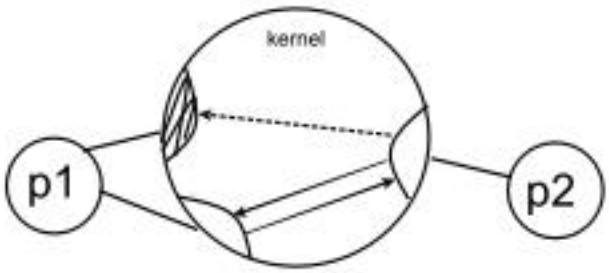
API, POSIX – СОКЕТИ

Сокетите също са връзка между два процеса, но за разлика от тръбата, могат да се свързват процеси без роднинска връзка, нар. конекция. За да изградим такава връзка между процесите, те трябва да знаят един за друг. Това става чрез пространството на имената.

Нека p1 и p2 не са роднини, но искат да се свържат. За да се свържат единият трябва да научи за съществуването на другия и за възможността да се свържат. Методът за научаване е **socket** – комуникационен обект, който съществува в ядрото на ОС и се нарича **крайна точка на комуникация,** т.е. двата процеса изграждат сокет (още не е конекция), то е начало на установяване на връзка.

Системното извикване socket() създава такива крайни точки. След това трябва на единия сокет да му се присвои име, да стане видим в пространството на имената. Създаването на име на сокет става със **bind().** Активирането на такъв именуван сокет става със системно извикване **listen()**. От другата страна няма нужда процесът да е именуван. Той ще изпрати **connect()** – изпраща заявка към именувания сокет за установяване на връзка. Сега вече може да се установи връзка. Именувания сокет създава нов комуникационен обкет (тръба) е реално връзката е между него и неименувания сокет. Има две тръби: едната е от клиента към сървъра, а другата от сървъра към клиента. Конекцията е двупосочна тръба.

Процесът p1 при установяване на връзката създава конекцията като нов комуникационен обект, а слушащият сокет продължава да работи. Той може да слуша и за нови връзки. Понеже така всички процеси могат да се свържат с него, той играе ролята на сървър, който изпълнява командата **accept()**, която приема заявки и създава файловия дескриптор от сървърния процес.



Server можем да определим по следния начин: процес, който е създал име (сокет) в множество, видимо за други процеси (пространството от имената) и така дава възможност на други процеси (клиенти) да изградят връзка с към него.

Но какво да прави сървърът: да гледа за нови конекции или да обслужва клиента? По подразбиране всички канали като ги създадем работят в синхронен режим, т.е. ако сървърът започне да търси нови конекции, но те не се появят, тогава сървърът не може да води разговор с клиента защото ще бъде приспан (от командата accept). Ако пък започне да разговаря с клиента и чака от него заявка, а той не я изпълнява, тогава пак ще заспи (от съответната команда, която чака).

Най-простият начин за решаване на този проблем в многозадачна среда е да създаваме копия на сървъра чрез fork и едното копие да обслужва вече свързаните клиенти, а другото да приема нови заявки.