

# Операционни системи

## UNIX – подобни операционни системи

Въпросник на Георги Георгиев – Скелета  
(за теоритични контролни и изпит по ОС, ФМИ)

2020-06-29

### ☐ СЪДЪРЖАНИЕ

1. Race Condition	2
2. Spinlock	3
3. Semaphore	5
4. Semaphore Implementation	6
5. Deadlock	7
6. Redirection/Pipeline	8
7. Философите и спагетите	11
8. Читателите и писателите	13
9. Състояния на процесите	14
10. Task Scheduler	15
11. Синхронни и асинхронни операции	17
12. Сигнали	18
13. Файлова система. Пространството на имената (namespace)	20
14. IPC и многонишкови процеси	22
15. Възможности за реализация на многонишкови процеси	24
16. Socket	25
17. Производителите и консуматорите	26
18. Памет и управление на паметта	27
19. Реализация на комуникационна тръба (pipe) чрез семафори	28
20. Основни комуникационни канали в Linux	29




## 1. Race Condition

Планирате да отидете да гледате филм с гаджето на кино от 21:00 часа. Проверявате за свободни билети в 19:00 часа, виждате че има такива и си отдъхвате. Решавате да си ги закупите на място 5 минути преди прожекцията. Оказва се, че кинозалата е пълна, а Вие сте на път да останете без гадже. Проблемът е, че между проверката и закупуването, други са изкупили билетите.

Race Condition на български може да се преведе като борба или състезание за ресурси и то възниква когато два или повече процеса/нишки имат достъп до споделени ресурси и се опитват да ги променят едновременно. Алгоритъмът за даване на процесорно време за изпълнение на процес може да скача непредвидимо и по всяко време. Ние не знаем реда, в който процесите ще се опитат да получат достъп до споделените данни (този ред може да се счита за хаотичен) и следователно резултатът от промяната в данните зависи от алгоритъма за планиране на процесите. По този начин два или повече процеса се „състезават“ за достъп до данните. Този достъп до данни, често е свързан и с някаква промяна и това може да доведе до нарушаването им, като в някой момент структурата от данни на общата памет е била нарушена временно от методите ѝ (тоест инвариантът на структурата е бил временно нарушен). Но именно ако през това време друг метод използва структурата, то ресурса ще бъде използван докато се е актуализирал – моментите, в които се актуализира се наричат критични секции. Проблемът често възниква когато един процес или нишка изпълнява операцията „провери и след това действай“ (например „провери“ ако стойността е  $x$  и след това „действай“, за да направиш нещо, което зависи от това, че стойността е била такава), а друг процес направи нещо със стойността на  $x$  между „провери“ и „действай“ на първоначално споменатия процес:

```
if (x == 5){ // „провери“
    y = x * 5 // „действай“
} /* но ако някой друг процес или нишка е променила стойността на x между
    проверката и действието, то y няма да е това което очакваме, тоест 25 */
```



(примера по-горе е валиден и за еднопроцесорни системи)

Въпросът е, че  $y$  може и да е 25, но може и да бъде всичко (като например парче от стринг), в зависимост от това дали друг процес е бъркал в критичен момент и в критична секция на кода. Няма как да знаем дали това се е случило. За да предотвратим появата на борба за ресурси, обикновено поставяме заключване около споделените данни, за да гарантираме, че само един процес може да има достъп до тях за определено време. Тоест нещо подобно:

```
lock(x) // заключи x
if (x == 5){
    y = x * 5 /* сега всеки друг процес или нишка няма да може да пипа x,
                докато заключването не се отключи и следователно ще знаем,
                че y = 25 */
}
unlock(x) // отключи x
```

Без изрично/експлицитно синхронизиране, което да синхронизира/координира достъпа до споделени данни, нишка може да промени променливи, които друга нишка е започнала вече да използва и това може да доведе до непредвидими резултати.

## 2. Spinlock

Повишаването на честотата на процесорите вече е станало невъзможно, поради чисто физически причини. За да се развиват системите, те стават все по-сложни с все повече ядра и повече от един процесор и тогава имаме реален паралелизъм. Паралелно работещите процесори правят метода с прекъсването неработещ. Може да прекъснем процесите от едно ядро, но процесите от другото ядро може да правят каквото си поискат и в това число и да бъркат в критични секции. Трябва да пазим някакъв допълнителен бит в споделената структура от данни, за да индикираме чрез него за това дали тя се ползва в момента от някой друг процес или нишка. Този бит се нарича *lock*, а метода, който ползваме се нарича *spinlock*.

За разсъжденията по-долу се базираме на допускането, че хардуера може да блокира прекъсвания. Този факт беше достатъчен за еднопроцесорна система. Допускаме още, че хардуера може да има специални атомарни инструкции, които да променят бит и да го прочитат след това като една единна непрекъсната инструкция.

```
spin_lock : R = test.and.set(lock)
            if (R == 1) goto spin_lock
            critical_section {
                P1
                :
                Pk
            }
spin_unlock : lock = 0
```

В момента, в който процес е в критична секция и друг процес изяви желание да я ползва, то програмата ще го връща циклично докато работещия в критичната секция процес не излезе от нея и не промени *lock bit*-а (макроса) на 0, което ще позволи на втория процес да влезе и да заключи след себе си.

В многопроцесорна система – това би трябвало да осигури защита, обаче възникват допълнителни проблеми: ами ако друга програма подбудена рекурсивно от критичната секция се извика? – *lock*-а вече ще е 1-ца и ще влезе в безкраен цикъл без да може да излезе, защото излизането и писането на 0 се случва само в края на критичната секция. Да допуснем, че по някакъв начин може да забраним на програмистите рекурсивното извикване в критичната секция. Но същия проблем може да се случи и при прекъсването. Нека обаче се опитаме да забраним прекъсванията:

```
spin_lock : disable_interrupt
            R = test.and.set(lock)
            if (R == 1) goto spin_lock
            critical_section {
                P1
                :
                Pk
            }
spin_unlock : lock = 0
            enable_interrupt
```

Нека дпуснем, че два процеса *p* и *q* искат да влязат в критичната секция и *p* е влязъл, а *q* цикли. Обаче *p* е забранил прекъсванията и не само *p*, ами и всички които чакат също са забранили прекъсванията, което е недопустимо, тъй като това е все едно да забраним на периферията да общува с всички тези циклещи процеси, което прави системата неефективна и слаба. От друга страна, другите процесори хем въртят цикли и вършат работа, хем тази работа е безполезна. Затова ще направим още едно последно подобрение и ще спрем до тук:

```

spin_lock : disable_interrupt
            R = test.and.set(lock)
            if (R == 0) goto critical_section
            enable_interrupt
            goto spin_lock

critical_section {
    p1
    ⋮
    pk
    lock = 0
}

enable_interrupt

```

Сега вече, ако критичната секция е заета, другите процеси ще циклят, но ще имат възможността да свършат и някоя друга полезна работа през това време.

### 3. Semaphore

Тъй като *spinlock*-а не реализира свойствата на хардуера, които ги имаме – като прекъсвания (а и самия той е непрекъсваем), както и за да може да реализираме пъргави системи за управление, е необходимо да може да осигурим възможност на критичните секции да са прекъсваеми и да има смяна на контекста. Това става с похвати на по-високо ниво, като един от тях е семафорът, който е въведен от Едгар Дейкстра през 60-те години на 20-ти век (приблизително през 1965 г.).

Семафорът е абстрактен механизъм за синхронизация, който предпазва ресурс от прекомерно използване, за да не настъпи катастрофа. Най-грубо казано, той представлява ключалка с брояч, който казва колко процеси/нишки могат да се намират едновременно в критичната секция, която пази ключалката. Структурата на данните му е следната:

Два атрибута:

- $cnt : int \rightarrow$  брояч (показва колко процеса могат да преминат бариерата, за да ползват споделения ресурс). Ако  $cnt < 0$ , то  $|cnt|$  е броя на приспани процеси;
- $L : list \rightarrow$  контейнер/списък, който може да го разглеждаме като множество от приспани процеси.

Силен семафор е този, който използва обикновена опашка за множеството  $L$ . Слаб семафор е този, при който има някакъв критерии, който се ползва за приоритет при обслужването на процес. Слабостта идва от факта, че такъв критерии може да причини така нареченото „гладуване“ на процеси/нишки с най-нисък приоритет, което се счита за проблем със синхронизацията. Не искаме да имаме процеси, които да чакат твърде много.

три метода:

- $init(int\ cnt = 0) \rightarrow$  инициализатор/конструктор на семафор, който приема аргумент по подразбиране – цяло число, с което се инициализира брояча; (виж т. 4)
- $wait() \rightarrow$  с този метод възлагаме работа на друг процес или искаме да ползваме общ ресурс. Опитва се да достъпи общ ресурс, ако получи достъп – декрементира брояча с 1, ако не получи – влиза в списъка с приспани процеси;
- $signal() \rightarrow$  освобождава общия ресурс (аналог на хардуерно прекъсване).

Хубавата операционна система предоставя готови комуникационни канали и възможност за ползване на семафори като допълнителен механизъм. Реалното място, където масово се използват семафорите е в ядрото на операционната система.

#### 4. Semaphore Implementation

```
struct Semaphore {  
  
    attribute int cnt = 0  
    attribute list L =  $\emptyset$   
    attribute bit lock = 0  
  
    constructor init(int _cnt) {  
        cnt = _cnt  
    }  
  
    procedure wait() {  
        spin_lock(lock)  
        cnt = cnt - 1  
        if (cnt < 0) {  
            L.put(self)  
            spin_unlock(lock)  
            block()  
        } else {  
            spin_unlock(lock)  
        }  
    }  
  
    procedure singal() {  
        spin_lock(lock)  
        cnt = cnt + 1  
        if (cnt ≤ 0) {  
            pid = L.get()  
            spin_unlock(lock)  
            wakeup(pid)  
        } else {  
            spin_unlock(lock)  
        }  
    }  
}
```

Имплементцията на семафор е демонстрация на факта, че механизмите за синхронизация от високо ниво, които са по абстрактни и по-далеч от хардуера (тоест ги разглеждаме само като софтуерни обекти), за да бъдат реализирани – трябва да се използват механизми от ниско ниво като *spinlock*.

## 5. Deadlock

*Deadlock* е събитие, което се получава когато процес чака друго събитие, което никога няма да се случи. Типично, това се получава когато нишка/процес чака мутекс или друг семафор, който никога няма да се освободи от предходния си ползвач, както и при ситуации когато имаме два процеса и две заключвания:

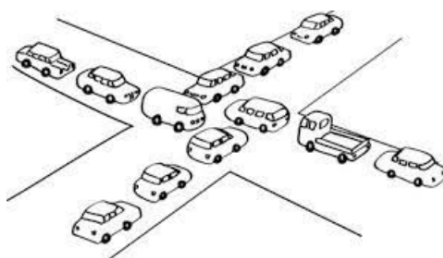
<i>Thread 1 :</i>	<i>Thread 2 :</i>
$Lock_1 \rightarrow lock()$	$Lock_2 \rightarrow lock()$
$WaitForLock_2()$	$WaitForLock_1()$

Също така, *deadlock* може да се появи и при цикличност, както в проблема с философите и спагетите (виж т. 7).

В учебника на *Таненбаум* се описват 4 правила, по които може да установим, че има *deadlock*. Необходимо е и 4-те условия да са изпълнени, за да може да кажем, че ще възникне *deadlock*.

1. Условие за взаимно изключване. Поне един процес да използва споделен ресурс монополно, тоест да има ексклузивни права над ресурса (да няма възможност друг процес да наруши работата му).
2. Условие за задържане и изчакване. Може да се случат поредици, при които процес взима даден ресурс и чака да вземе следващия.
3. Това ползване на споделен ресурс в 1. да не може да бъде прекъсвано от външен фактор.
4. Усowie за кръгово (циклично) изчакване (както при синхронизационния проблем с философите и спагетите (виж т. 7))

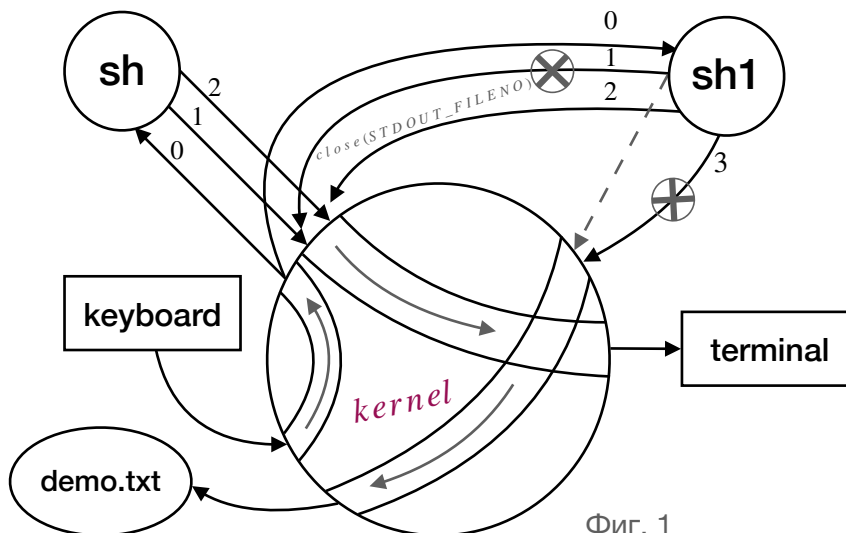
Аналогичен пример за *deadlock*, на този с философите е примера със задръстването на нерегулирано натоварено кръстовище:



В случая кръстовището е ресурс, а превозните средства за нишките.

## 6. Redirection pipeline

Разглеждаме ядрото с неговите канали и shell, който изпълнява нашите команди. Този shell (sh) си има 3 файлови дескриптора (0 – stdin (стандартен вход), 1 – stdout (стандартен изход) и 2 – stderr (стандартна грешка)).



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

int main(void) {
    int pid, fd;
    pid = fork();
    if (pid > 0) wait(NULL);
    char* args[] = { "ps", "-a", (char*)0 };
    if (pid == 0) {
        fd = open("demo.txt", O_WRONLY|O_CREAT, 00640);
        close(STDOUT_FILENO);
        dup(fd);
        close(fd);
        execvp(args[0], args);
    }
    if (pid < 0) perror("fork");
}
```

След извикването на системната команда `fork( )`, детето на процеса наследява същите входно-изходни канали. Входовете на двата процеса четат битове от клавиатурата, а стандартния изход и стандартната грешка изтичат към терминала. Но ние искаме стандартния изход да не отива към терминала, а да го записваме във файл с име `demo.txt`. Необходимо е да променим начина на работа на детинския процес и да пренасочим неговите файлови дескриптори.

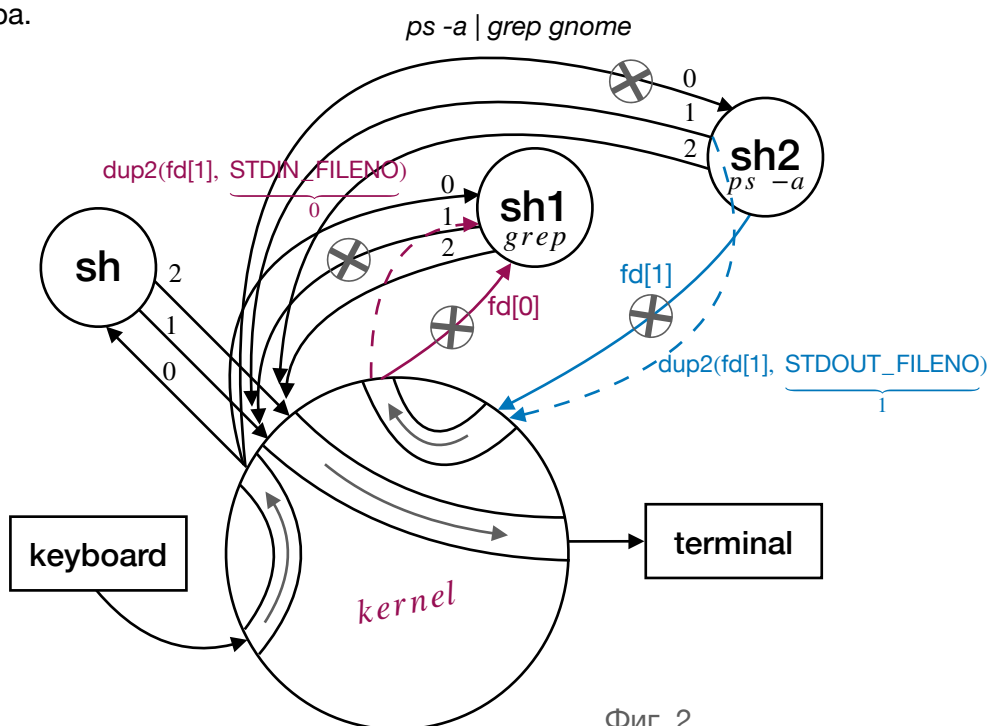
Системното извикване `open(...)` създава нов комуникационен канал в ядрото, който свързва процеса с файла `demo.txt`. Създава се файлов дескриптор в детето, където е извикано `open(...)` (в случая `fd 3`), който сочи към единия край на комуникационния канал, а другия край сочи към файла. Единствено детинския процес знае за този комуникационен канал и може да го адресира.

Искаме детето да изпълни командата `"ps -a"`, но тя очаква да има 3 отворени файлови дескриптора, които да са 0, 1, 2, а пък на този етап нашия процес има 4 отворени – 0, 1, 2, 3. За да работи командата както искаме и както очакваме, трябва да нагласим правилно файловете дескриптори.



Първо трябва да затворим стандартния изход "close(STDOUT\_FILENO)". Сега sh1 има точно 3 fd (файлови дескриптора), но не са правилно насочени. Копираме файловия дескриптор 3, който "open(...)" е създал в този, който е с най-нисък свободен номер. В нашия случай това ще е 1 (stdout, който преди малко затворихме). Това става чрез командата "dup(fd)". Сега файловете дескриптори на sh1 са наредени точно както искаме. Остана само fd 3, който сочи към същото място към което сочи и fd 1 и за да не създава проблеми, просто го затваряме.

Другата важна дейност е направата на две или повече команди да работят в конвейер: "cmd1 | cmd2". В съвременните операционни системи, двете команди се пускат да работят паралелно и ги свързваме с комуникационна тръба, в която първата команда изпраща битовите излизаци на изхода ѝ на единия край на тръбата, а втората команда ги чете от другия край на същата тази тръба. Това е по-ефективно, тъй като работата с файлове е много бавна. Достъпа до тях, когато са на други устройства (флашки, твърди дискове) е около 1000 пъти по-бавен отколкото достъпа до RAM паметта, а и този файл може да не ни трябва, а просто да искаме композиция от команди и техния резултат. На фиг. 2 е показано как се организира комуникационен канал, а кода след нея показва как се реализира.



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

int main(void){
    int pid, pid1, fd[2];
    char* args1[] = { "ps", "-a", (char*)0 };
    char* args2[] = { "grep", "gnome", (char*)0 };
    pid=fork();
    if (pid > 0) wait(NULL); // Waits the child termination
    if (pid == 0) { /* Child shell */
        pipe(fd); /* Create pipe */
        pid1 = fork();
        if (pid1 == 0) { /* process for cmd1 */
            close(fd[0]); /* Close unused read end */
            dup2(fd[1], STDOUT_FILENO);
            close(fd[1]); /* Close duplicated write end */
            execvp(args1[0], args1);
        }
    }
}
```

```

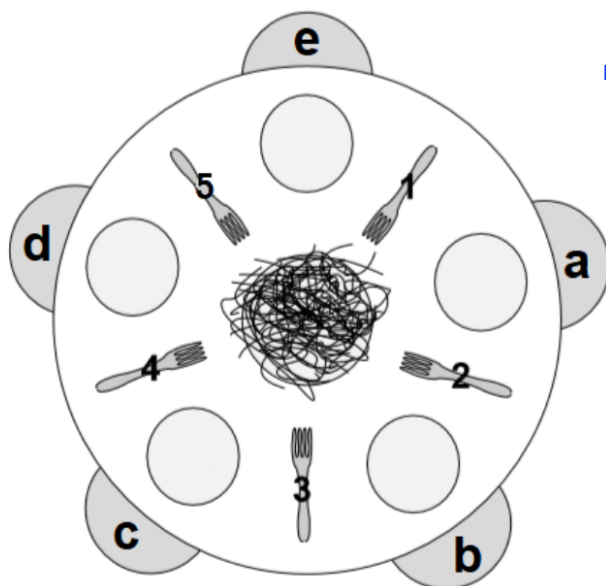
        if (pid1 > 0) {      /* process for cmd2 */
            close(fd[1]); /* Close unused write end */
            dup2(fd[0], STDIN_FILENO);
            close(fd[0]); /* Close duplicated read end */
            execvp(args2[0], args2);
        }
    }
    if (pid<0) perror("fork");
}

```

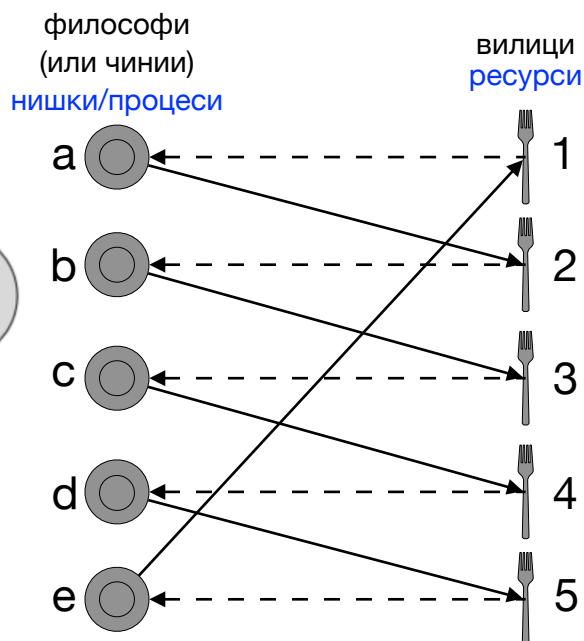
sh1 и sh2 се изпълняват паралелно в детински процеси на sh (тоест и трите процеса са пуснати едновременно). Последователността на завършване на командите не ни интересува, тъй като тръбата очаква да постъпят битове от единия процес, за да ги предаде на другия. По този начин оставяме на ядрото само да се грижи за това и така комуникационните тръби подреждат кой процес да работи и кой да си почива в зависимост от състоянието (синхронизацията ще се получи по естествен път).

## 7. Философите и спагетите

Разглеждаме кръгла маса, на която са седнали 5 философа и се опитват да се нахранят. По средата на масата има купа със спагети, а пред всеки философ има по една чиния и между всеки две чинии има вилица. За да си сложи от спагетите в чинията и да се нахрани, на даден философ са му необходими да използва две вилици едновременно.



Фиг. 3



Фиг. 4

Правим допускането, че философите знаят как да мислят и как да се хранят. Нашата работа е да напишем версия на разпределение на ресурсите (вилците), в която да са изпълнени следните синхронизационни 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) философа 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 (в нашия случай лявата – фиг. 3), тогава ще настъпи *deadlock*, защото за да се нахрани са му нужни две вилици, които да използва едновременно и всеки философ ще чака и ще гладува вечно.

Може да моделираме лесно тази ситуация с двуделен граф показан на фиг. 4. Процесите ще са философите, а ресурсите са вилците. С насочените ребра в графа ще обозначаваме необходимостта от ресурс на процес, за да може да си свърши работата.

Нека разгледаме горепосочената ситуация, в която всеки философ си е взел лявата вилица и се опитва да вземе дясната. Ще насочим ребрата на графа, за да обозначим това, като посоката ще е от процес към ресурс, а пък обратната (пунктирна) посока ще означава, че процеса се опитва да вземе този ресурс, но той е зает. При такова ориентиране на ребрата се получава цикъл и това води до *deadlock*.

Естествено няма единен подход за справяне с този казус. Нещо повече, всеки един от подходите изисква да знае нещо допълнително за системата, за да се справи с казуса. Нека например номерираме множеството от процеси и множеството от ресурси, така че да знаем на кой процес – кои ресурси са необходими (тоест тук използваме система за номериране, ако изобщо можем, или поросто наредба в линеен ред). 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. Твърдението е, че ако ресурсите се вземат в реда на нарастване на номерата – няма да възникне *deadlock*.

Тогава алгоритъма няма да е: „всеки да взима първо лявата вилица“, а ще е „всеки да взима първо вилицата с по-малък номер“. Така *deadlock* няма да настъпи (естествено има малка вероятност за гладуване на някои философи, но *deadlock* няма да има. За да се гарантира липсата на гладуване е необходим по-задълбочен анализ).

## 8. Читателите и писателите

Проблемът в задачата с читателите и писателите се отнася до всички ситуации, в които от общи ресурси (структури от данни, бази от данни или файлови системи) се чете и се модифицира от паралелно изпълняващи се процеси. Докато в структурата от данни се пише или се модифицира е необходимо да се забрани на другите нишки/процеси да я четат, за да се предотврати прекъсването на писането, както и прочитането на нещо неконсистентно или невалидно.

Както в задачата за производителите и консуматорите (виж т. 17), така и тук решението ще е асиметрично. Читателите и писателите ще изпълняват различен код преди да навлязат в критичната секция. Синхронизационните ограничения са следните:

1. Произволен брой читатели могат да бъдат допуснати до критичната секция едновременно;
2. Писателите трябва да имат ексклузивни права до критичната секция. С други думи, писател не може да навлезе в критичната секция, ако някоя друга нишка (писател или читател) е там, и докато писател е там, никой друг не може да влиза.

Двата типа процеси, които имаме тук, сформират така наречената категоризация на взаимното изключване.

Идеята 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. Тази техника се нарича *lightswitch* (първият, който влиза в стаята – светва лампата, за да сигнализира на останалите).

```
semaphore roomEmpty.init(1),
```

```
int cnt = 0 (cnt  $\equiv$  readers)
```

```
semaphore mutex.init(1) // с този мутекс ще защитаваме брояча, за да не може да бъде  
нападат паралелно от няколко читателя. Чрез него избягваме race condition.
```

READER

```
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()
```

WRITER

```
roomEmpty.wait()
```

WRITE

```
roomEmpty.signal()
```

До тук решението е коректно, НО има недостатък. Възможно е писателите да гладуват, заради наличието на много на брой читатели или бавни такива. Необходим е допълнителен трик. Ще сложим бариера, която ще спира читатели да навлизат, при условие, че има поне един писател, който чака да влезе в критичната секция. За целта ще добавим още един семафор:

```
semaphore barrier.init(1)
```

READER

```
barrier.wait()
```

```
barrier.signal()
```

```
...
```

WRITER

```
barrier.wait()
```

```
roomEmpty.wait()
```

WRITE

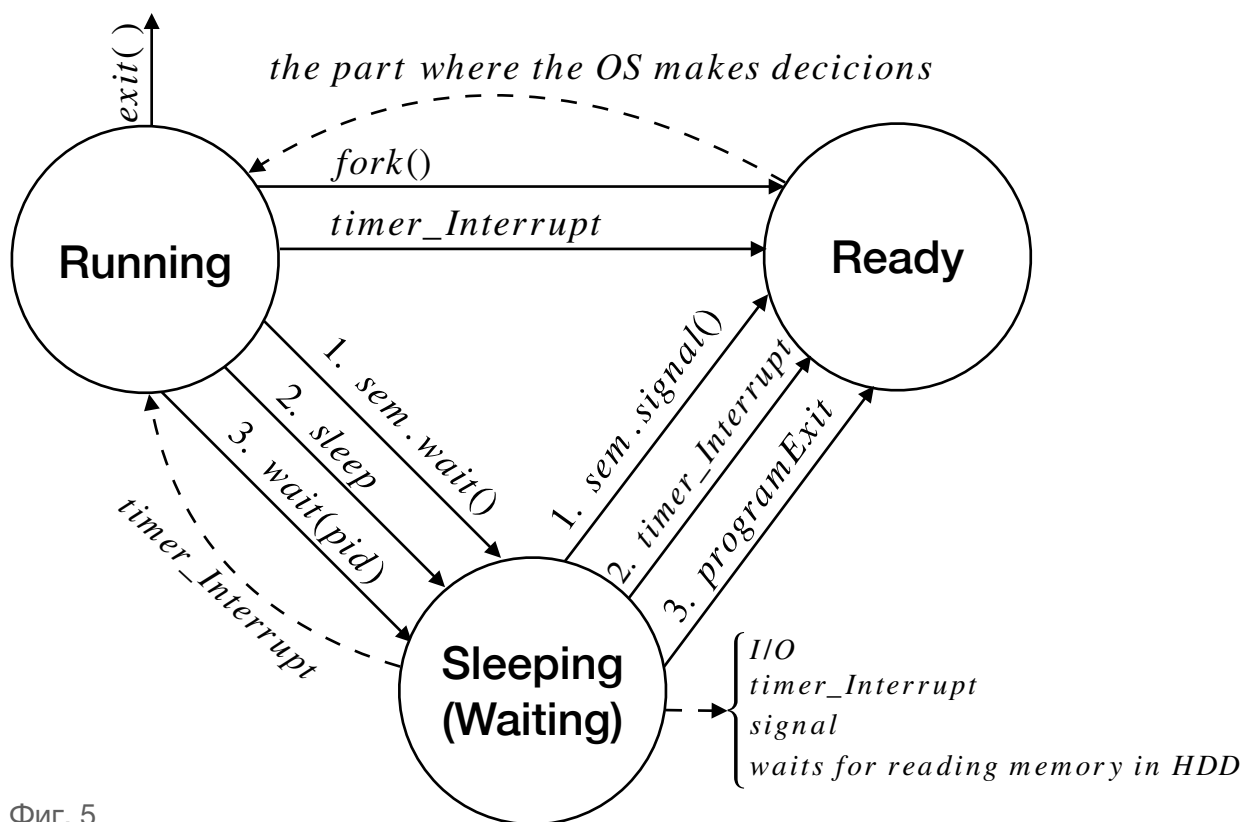
```
barrier.signal()
```

```
...
```

## 9. Състояния на процесите

- 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 потребителя 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 до 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)

Разликите в състоянията на процесите въздействат само на процесорите. Спящият процес не използва компютърно време (изчислителен ресурс), а пък работещият процес се бори за компютърно време.



Фиг. 5

Избора кой от очакващите процесор процеси трябва да му се разреши да работи (най-горната пунктирна стрелка) е най-тънката част от управлението на работата на процесите. Това се решава от специален алгоритъм в ядрото, който в някои системи се нарича диспетчер а в други task scheduler. Това е частта, в която ОС взима решенията.

## 10. Task Scheduler

Task Scheduler или диспетчера на задачи е алгоритъм, който се намира вътре в ядрото и решава кой от готовите процеси чакащи процесор ще започне да работи. Този диспетчер трябва да отговори на редица най-разнообразни изисквания, които зависят както от функционалните особености на типовете процеси, така и на изисквания за коректна работа на системата.

От гледна точка на функционалността на процесите те могат да се разделят на:

- ◆ I/O – основно или често извършващи входно изходни операции (interactive, foreground). За такъв процес е характерно, че ще прави малко изчисления, например чака някакъв текст и го изпраща някъде. Но той често ще бъде приспиван и често ще прекарва повече време в другите структури между Running и Ready. Затова ще искаме бързо да ги вкараме в активно състояние, за да има ниска латентност (low latency);
- ◆ CPU – такива които смятат повече (background). За тях важното е да им се предостави много изчислително време, но не е толкова важно кога;
- ◆ Real-time – при тях има изисквания, които са доста по-строги. Това са приложения, които ще комуникират с околния свят – т.е. управляват някакви важни машини, устройства или инструменти и е критично важно да няма забавяне, спрямо предварително изграден времеви план. Те имат т.нар. *deadline* – предварително дефинирани срокове, за които да се получи управлението на системата.

Най-простия алгоритъм, който работи за системи с времеделение и създава илюзията, че работят паралелно процесите, се казва *Round Robin*. Той нарежда чакащите да получат управление процеси в обикновена опашка. В този алгоритъм няма никакви приоритети. Всички типове процеси разгледани по-горе се разглеждат като едно еднообразно множество, което няма структура и специфика.

Предимства: при него гладуване няма да възникне, тъй като опашката е обикновена. Никой процес няма да седи безкрайно дълго в тази опашка. След  $n$  кванта време, гарантирано ще получи управлението и ще се достави процесорно време.

Недостатъци: ако се натрупат много интерактивни процеси, те много бързо ще напускат процесора и техният квант време ще остане неизползван, обаче когато дойде входно-изходното събитие те няма да могат да реагират пълноценно, защото преди тях в опашката ще има много процеси. Те няма да имат специален приоритет. За real-time процесите пък изобщо няма да бъдат удовлетворени изискванията след като за I/O процесите не са.

Какви параметри може да променяме в този *Round Robin* алгоритъм, ако искаме да даваме някакви предимства? Този квант време, който даваме може да го направим много малък, за да има много честа смяна на процесите и да идва по-бързо кванта време за процес (което ще е добре за тези които имат изискване за много малко процесорно време). Тогава обаче ще навредим на CPU процесите, а и на цялата система, защото при смяна на процесите имаме смяна на контекста, която е загуба на време (запомняне на регистри, сменяне на параметри, прекъсвания и т.н.).

В съвременните операционни системи, тази структура, в която пазим готовите процеси не е обикновена опашка, а е много по-сложна структура, която е изградена от няколко структури. Всеки един от типовете процеси ще складираме в различна структура и алгоритъма за избор да решава във всеки момент от коя структура да избере процес. Също така, процесите които имат голяма нужда да започнат работа, вместо от готови да отиват в действащи, може директно да се вкарват в действащи, без да отиват в готови (т.е. да се усложни събуждането на процес, ако е много важен).

Естествено самото наличие на приоритети създава рискове за гладуване. Трик за справянето с този проблем е наличието на т.нар. плаващи приоритети. Въвеждат се двойка приоритети – h (hard) и f (floating) приоритети (твърд и плаващ) <h, f>. Така всеки път, когато процес иска процесорно време и не му се дава, диспетчера ще вдига плаващия му приоритет. При запитване за приоритет ще се гледа само максималния приоритет от двата приоритета.

Друга техника е да променяме квантът време. Той ще играе особена роля за Real-time процесите. Там ще има специална структура, в която си пази deadline-а за тези процеси. Друга еволюция на тези алгоритми за управление на времето се дължи на това до колко са автоматизирани и до колко са зависими от потребителите (администраторите).

*В Linux, Scheduler-а сам разпознава типа на процеса, по това каква част от такта използва. Той може да го определи като следи статистика за работата на процеса.*



## 11. Синхронни и асинхронни входно-изходни операции

При синхронна операция, процесът може да се блокира, да се приспи и операциите се извършват в тяхната цялост. Ако операционната система може да окомплектова операцията – тя я окомплектова и тогава връща резултат, иначе връща код за грешка по някаква друга причина.

При асинхронна операция, процесът не се приспива, но това се заплаща с цената на върнат на части резултат от операцията и това се индикира със специален код за грешка, които показва, че процеса трябва да се довършва. Потребителя сам се грижи да прави последващи извиквания, за да довършва операцията. Използването на такъв тип 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.

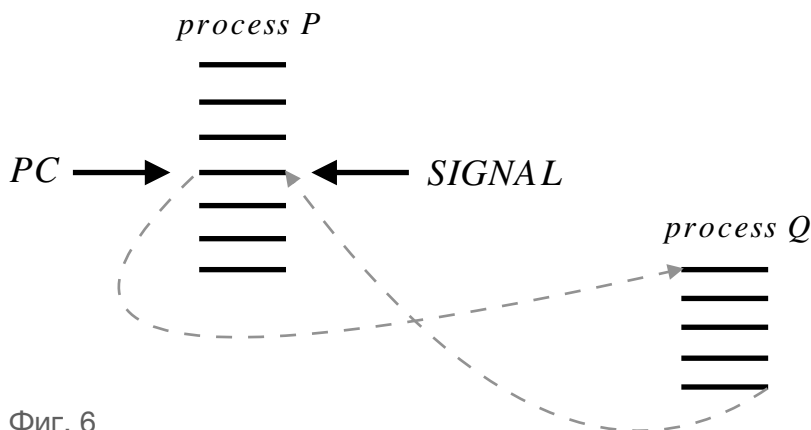
За примери може да дадем програми, които искат пъргаво да работят с няколко комуникационни канала без да бъдат приспани от някой канал и да не могат да обслужват другия. Т.е. да трябва паралелно да се реагира на информацията постъпваща или пътуваща по няколко канала. Класически пример са графинчните приложения (използващи информация от мишка, клавиатура, touch screen и т.н.) или сървъри, които работят с много клиенти.

Например, 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. Browser-ът проверява чрез асинхронни опити за четене – по кой от каналите постъпва информация и реагира адекватно. Също така на Browser-ът може да му се наложи да извлече някаква информация от даден URL и в същото време да даде възможност на потребителя да извършва други операции, докато чака да се зареди информацията асинхронно.

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

## 12. Сигнали

Сигналят от една страна е абстракция на хардуерните прекъсвания, а от друга – служи за предаване на прости съобщения между процесите.



Сигналите са прекъсвания на софтуера, изпратени до програма или процес, за да покажат, че е настъпило важно събитие. В контекста на *Windows*, те се появяват на по-късен етап и се разглеждат като *event driven programs* (събития, които могат да се обработват от процес). Но в *UNIX*, тези абстракции присъстват от почти самото начало и са добре описани в документацията.

Сигналите се предават под формата на номер. Когато даден процес получи сигнал, то все едно че е прекъснат, но не от хардуера, а от операционната система. След това абстрактно прекъсване – друг процес или програма започва обработката на сигнала (фиг. 6), като тази програма не е от общия код. В *Linux* нормалните сигнали са 32.

За да може да изпращаме сигнали на всички работещи процеси, трябва да имаме *root* (административни) права. Може да слагаме побитови маски, за да подтиснем временно получаването на сигнали.

Има няколко начина, по които даден процес може да получи сигнал:

- ◆ Сигнал изпратен от друг процес (обикновено това е процес, който има роднинска връзка с получаващия сигнала процес);
- ◆ Сигнал изпратен от повреда на комуникационен канал;

Такива сигнали са например *SIGHUP*(1). Този сигнал възниква при спиране на връзката между контролиращ терминал или прекратяване на контролиращ процес на даден процес.

*SIGPIPE*(13) също е подобен тип сигнал. Той се изпраща, когато например работещият процес е свързан с друг процес с комуникационна тръба, но тръбата се разруши, защото другият процес е затворил файловия дескриптор и тръбата няма друг край.

- ◆ Сигнал, който се изпраща, когато има наличието на грешка в кода или невъзможност да се изпълни инструкцията.

*SIGILL*(4) – невалидна инструкция (процесора не може да изпълни инструкцията която е в програмата);

*SIGFPE*(8) – Floating-point exception – невъзможна числова операция (например деление на 0 или събиране на числа, но сумарното число прелива заделената за него памет);

Други сигнали:

SIGTERM(15) е сигнал за извършване на действия по спиране на процеса (сигнал тип „събери си багажа“). Този сигнал дава възможност на процеса да си изтрие/„почисти“ заделените/използваните от него ресурси и чак тогава да спре. Тази процедура по събиране на багажа може и да е от тип възстановяване на инварианта на структурата от данни, която процеса може да е разбутал по време на работата си, докато получава сигнала (това възстановяване изобщо не е тривиално, дори и за прости структури и се счита за висш пилотаж в програмирането);

SIGSTOP(17,19,23) – не прекратява работата на процеса, а временно го спира (все едно го приспива);

SIGCONT(19,18,25) – събужда процес, който е приспан (когато процеса е стопиран със SIGSTOP, със SIGCONT може да му кажем да продължи изпълнението си);

SIGKILL(9) – прекратява незабавно даден процес, без да му дава възможност да извърши действия за почистване след него;

SIGSEGV(11) – когато процеса се опитва да ползва памет, която не му е разрешена, например страница, която не е дефинирана във виртуалната му памет.

Интересна част за разглеждане е когато се праща сигнал на процес, който се намира в структурите за изчакване (*sleeping* – виж т. 9). Тогава записваме сигнала в тази структура, която описва процеса и когато събуждаме процеса, тогава му подаваме сигнала. (*man 7 signal*)

### 13. Файлова система. Пространство на имената (namespace)

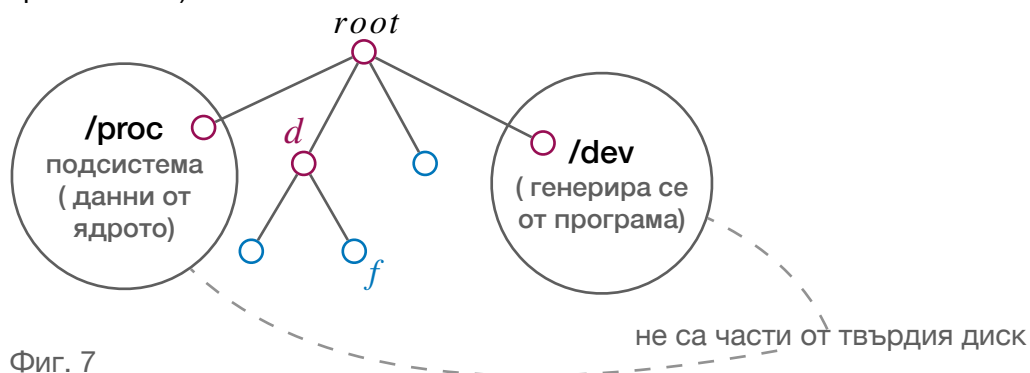
В съвременната операционна система има 2 слоя на абстракция:

1. Пространство на имената – всички дълготрайни обекти, които съществуват в системата. Това е абстрактната файлова система;
2. Реализацията – как са представени тези обекти и как са съпоставени на хардуера (всеки информационен обект, как и къде ще бъде в хардуера).  
В UNIX, абстрактната файлова система е *едно* кореново дърво. В Windows е *гора* от коренови дървета.

"–" – обикновени файлове; "*d*" – папка или директория; "*l*" – символен линк (друго име на файл – съществуващ или не); "*c*" – символно устройство (character device);

"*b*" – масив от байтове, който не е RAM паметта на компютъра, нито процесора и неговите регистри, а е някакво външно устройство, което съхранява някакъв масив от байтове и може да се ползват или записват байтове където решим (може да се индексира);

"*p*" – именувана тръба (FIFO); "*s*" – крайна точка за комуникация (socket). Процес сървър, който предоставя връзки за съответни услуги (служи, за да може клиента да намира сървъра и да се свързва с него).



Реализация на файлова система:

Твърд диск – това е диск разделен на много „пътечки“:

- ➔ Над повърхнината има електрическо устройство, което засича дали бита 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;
- ➔ Файлът 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.

Алгоритъм на асансьора:

- Разместват се така заявките за четене и писане, че главата да изминава минимално разстояние;
- Има две приоритетни опашки: едната се състои от файлове, които се намират по посока на движението на главата, а другата от файлове, които се намират в обратната посока;
- Ако е празна главата на опашката по посока на главата на устройството, то се сменя посоката.

Двете стратегии – кеширане и алгоритъмът на асансьора, пораждават проблем за надеждността (внезапно спиране на тока). За да се справим с този проблем съществуват така наречените журнали (лог файлове). Записваме промените във файл, който се нарича журнал, в него се запазва пълната информация за операциите над файлове и когато се позапълни/препълни журнала, спираме и отразяваме операциите от журнала над файловете, ако спре тока, журналът ще пази последните промени, а файловете ще са си консистентни. Четат се първо старите файлове и се проверява дали в журнала са променени, ако спре тока, 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 (лог файла) 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.

Две фази:

1. Файловите операции в истинския ред по време се записват в журнала;
2. Когато се натрупат се подават заявките на алгоритъма на асансьора.

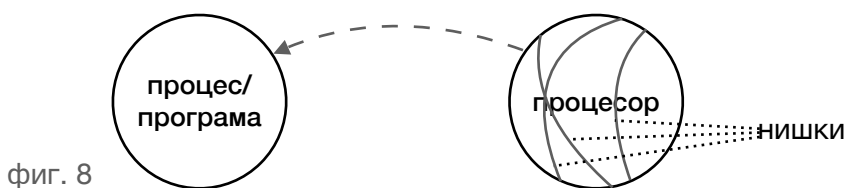
## 14. IPC и многонишкови процеси

За да може да се изграждат и по-специфични решения, чрез които потребителската програма да може да изгражда собствени средства за споделяне на информация с други процеси, в UNIX има един допълнителен механизъм, който е от по ниско ниво и е по-близък до тези механизми за синхронизация като *race condition*, *semaphores*, *spinlock*, *deadlock*, *starvation* и други условни понятия. Този механизъм е стандарт за UNIX и за POSIX.

Този инструмент се казва IPC и се представя като набор от системни извиквания, които дават възможност да се изграждат споделени ресурси между процесите и средства за синхронизация на достъпа до тези ресурси. Тези извиквания са три вида процедури:

1. Споделени памети – с помощта на тези системни извиквания може да се дефинират парчета памет (блок RAM), които да може да се ползват от няколко процеса. Извикванията, които обслужват тая дейност започват с *shm\** (*shared memory*). Някой процес изгражда/инициализира такъв блок споделена памет и друг процес я използва. (Споделените ресурси си имат пространство на имената – на създадената споделена памет се дава име и се създават и права за достъп до нея);
2. Семафори – след като имаме споделени памети трябва да имаме и механизми за справяне с *race condition*, за това имаме и тези семафори. Те също са споделени и групата им за извиквания са с начало *sem\**. Тази група семафори са малко по-сложни от разглежданите до сега класически семафори в общата теория. Те могат да увеличават и намаляват брояча на семафора не с 1-ца ами с някаква по-голяма константа;
3. Механизъм за предаване на съобщения – този механизъм може да се използва и за синхронизация и за обмен на информация между процесите. Командите в този механизъм започват с *msg\**. Това са нещо като пощенски кутии, които може да си дефинираме и един процес да ги създава и да се ползват от други процеси, включително и от създателя си. Това което се изпраща е масив от байтове. Този механизъм пак може да се разглежда като метод за синхронизация. Чрез механизма за изпращане на съобщения може да се реализират и семафори както и обратното.

Друг механизъм, който го има в съвременните операционни системи са нишките (*threads*). До сега говорехме за процес като програма, която е стартирана, изпълнява се и има една нишка на изпълнение (т.е. кода на програмата се изпълнява от виртуална машина – все едно е един компютър и няма паралелизъм).



За многонишков процес ще предпологаеме, че сме разширили използваните изчислителни ресурси, така че да може няколко процеса (все едно) да изпълняват програмата, която сме пуснали. Няколко паралелно работещи нишки работят – т.е. паралелно работещи изчисления да вървят по програмата, която описва процеса.

Ако се опитаме да я направим тази по-сложна конструкция чрез няколко паралелни процесора (било то виртуални), които изпълняват някоя нишка, дължни ли са нишките всичко да споделят или всяка нишка ще има нещо собствено, което да е уникално за нея и да е локален ресурс за нея? Като минимум всяка нишка трябва да има собствен *call stack* (локални данни на нишката). Освен това тези нишки трябва да имат локални памети за състоянието ѝ (дали е приспана). Другите неща може да бъдат споделени – файлови дескриптори, комуникационни канали, обкръжение и т.н. . Многонишковите процеси се дефинират от съображение за ефективност и ползване на споделени ресурси. Причината

за употребата на такива процеси в UNIX средите е схващането, че по-бързо ще работят някои услуги, ако се реализират като многонишкови процеси. Идеята е, че превключването на управление между нишки може да се направи много по-бързо от превключването между процеси. При смяната на контекста на нишките няма да сменяме таблицата на виртуалната памет, която процеса използва. Ще трябва да се сменим само стека и да се запомнят регистрите при смяна на работа на нишките. Няма да се извършва смяна на цялостния контекст. Така ще се спести време от разчистването на кеша, виртуалната памет ще остане същата и т.н.

## 15. Възможности за реализация на многонишковы процеси

Едната възможност е самото ядро да не знае, че съществуват нишки в потребителския процес и да се реализират с помощта на някаква структура, която да управлява различните нишки. Т.е. процеса да може да си дефинира стекове и подобни области и когато една нишка е активна – тя да може да предаде управлението. Например, някаква библиотека за управление на нишките. Така няма да се губи време при системни извиквания.

Недостатък тук е, че паралелизма трябва да се поддържа от самия потребителски процес. Т.е. една от активните нишки (нишка диспетчер) да се грижи сама да предава управлението на други нишки, за да се създаде поне илюзията, че всички нишки работят едновременно/паралелно.

Другият недостатък е, че така ще се загуби ползата от това да може няколко нишки да смятат даден процес, след като една ще работи, а другите ще чакат. Идеята ще бъде пренебрегната и ще се изгуби смисъла.

Постепенно се прибягва към решение, в което ядрото поддържа всички нишки. Т.е. всяка нишка да бъде дефинирана като абстракция в ядрото (*Cernel Level Threads*).

Като недостатъци на нишките са затрудняването на програмирането от гледна точка на програмиста. Ако нашата програма например има много нишки, ние ще трябва да се грижим да не настъпва *race condition* в общите за нишките ресурси (например паметта). Необходимо ще е да се използват различни механизми за синхронизация (конкурентното програмиране е по-сложно).

В операционната система, в която ползваме само еднонишковы процеси и в която комуникацията между тези процеси става през стандартно добре дефинирани комуникационни канали, които операционната система поддържа – цялото конкурентно програмиране е вкарано в ядрото. Логиката на програмата, която пише програмиста е естествената логика, която се изпълнява от една машина (фон Нойманова последователност). Но когато самият програмист трябва да управлява паралелизъм, той трябва да синхронизира конкурентната работа на нишките.

Другият проблем в UNIX е как механизма на нишките да се синхронизира с другите абстракции в комуникационната система (сигнали, стандартни комуникационни канали). Например, следния казус: когато един процес изпраща сигнал на друг, коя нишка ще получи сигнала? (Ако се приеме от някоя произволна нишка, то програмата ще има грижа да разпространи тази информация, която е постъпила, и до останалите.)

За работа с много нишки се използва по дълбокото от `fork( )` системно извикване `clone(...)`. Всъщност, `fork( )` се реализира чрез `clone(...)`. То е извикване, което може да създава и нишка и процес.

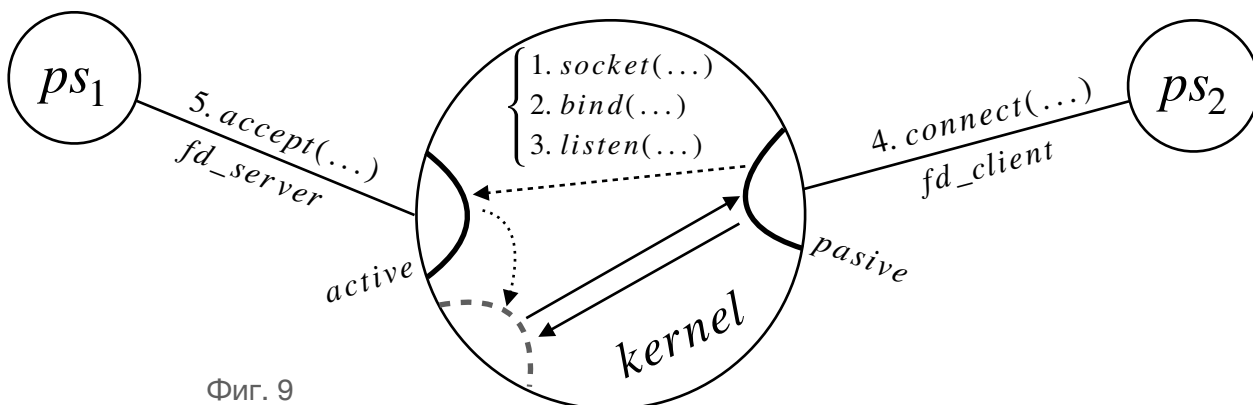
Разликата между нишка и процес в Linux е, че процеса има `pid` (*process identification descriptor*), а пък нишката си има и `pid` и уникален номер на нишката. `clone(...)` създава номер на нишка, като той има номер на опции (няколко допълнителни параметри, които казват дали ще има същия `pid`). Ако при `clone(...)` кажем нищо да не бъде споделено между новата нишка и старата, то все едно сме казали `fork( )`. Простата/чистата нишка е когато `clone(...)` се извика с максимално повече споделени данни между новата и старата нишка (споделен `pid`, памет, данни от обкръжението, стари адресни пространства и т.н.).



## 16. Socket

До сега разгледахме два метода за разговор между процеси, които са абстрактни комуникационни канали в UNIX. Това са тръбата, която е връзка между два роднински процеса, като единият предава информация в тръбата, а другият я приема. Другият абстрактен комуникационен канал е връзката между процес и файл. Когато програмата отвори файл, се изгражда такава връзка и процеса и файла си комуникират (може да си представим файла като някакъв много прост абстрактен процес, който само съхранява информация).

Има и трети вид абстрактен канал – връзка/connection (socket). За разлика от тръбата тя може да свърже процеси, които нямат роднинска връзка. Това може да са процеси от различни изчислителни системи. За да изградим връзка между два непознати процеса е необходимо да направим така, че единият процес да разбере за другия и да може да го адресира и посочи. Т.е. трябва да има някакво пространство на имената, за да може да го посочи по име.



Фиг. 9

( $ps_1$  и  $ps_2$  са два процеса, които нямат роднинска връзка и искат да се свържат и да си комуникират)

Метода за запознаване на два непознати процеса е именно този обект `socket`, който съществува и се изгражда от ядрото на системата. Той се нарича още – *крайна точка за комуникация*. `socket`-а сам по себе си е само начало на установяване на връзка (но все още не е установена такава). Следващата стъпка в този процес на установяване на връзка е на единия `socket` да му се присвои име и той да стане видим в пространството на имената. Даването на име на `socket`-а се извършва чрез системното извикване `bind(...)`. Има три адресни пространства – `AF_UNIX` (локално пространство от имена), `AF_INET`, `AF_INET6`. Последните две са видими не само в локалната система на изчисления, а в целия интернет. Чрез такива сокети може да се правят връзки между два компютъра в интернет.

След като на `socket`-а е присвоено видимо име, той трябва да се активира и да започне да работи като телефонна секретарка, която разполага с телефонна централа и може да прави връзки. Това става чрез системното извикване `listen(...)`. Тогава ядрото започва да поддържа структури в този `socket`, които подслушват дали някои други програми не търсят именувания `socket`. От страна на клиента, той също трябва да си създаде `socket` (*пасивен*), но не е нужно да го именува. Той трябва да изпълни системното извикване `connect(...)`, което изпраща заявка до именувания `socket` под формата на запитване дали може да установи връзка, а пък приемането на това запитване става от страна на именувания `socket`, чрез системното извикване `accept(...)`. Пасивният `socket` получава тази заявка и вече може да установи връзката, тъй като той вече е разбрал за другия процес. Слушащият (активен) `socket` изгражда нов комуникационен обект и този комуникационен обект вече е връзката между двата процеса. Конекцията е еднопосочна тръба, затова, за да има пълна комуникация е необходимо да има две тръби. След изграждането на този нов комуникационен обект, слушащият `socket` продължава да работи и може да слуша за нови връзки. Този процес на слушане от страна на `socket`-а играе ролята на сървър. Процесът изградил `socket`-а е процес, който позволява на други процеси да се свързват с него и да провеждат някакви комуникации и той да ги обслужва.

## 17. Производителите и консуматорите

В многонишковите програми, които разгледахме в т. 14., често има разделение на труда между нишките. В един и същ процес – едни нишки може да са производители, а други – консуматори. Производителите създават елемент от някакъв вид и го добавят в някаква структура от данни, а консуматорите го вадят от тази структура от данни и работят с него.

Програмите управлявани от събития са добър пример. Под събитие ще разбираме нещо което се случва в даден момент (натискане на бутон или движение на мишка, блок от данни пристига от диска, пакет пристига от мрежата, чакаща операция приключва и т.н.) и изисква програмата да реагира по някакъв начин.

Когато събитие се появи, производителска нишка създава обект на събитието и го добавя в буфера на събитията. Конкурентно с това, консуматорски нишки вимат от този буфер събития и ги обработват. В този случай, консуматорите се наричат мениджъри на събития (*event handlers*).

Появяват се няколко синхронизационни ограничения, които ние трябва да наложим, за да работи тази система коректно:

- ◆ Когато обект на събитие се добавя или премахва от буфера, той (буфера) е в неконсистентно състояние. Следователно, нишките трябва да имат ексклузивни права до буфера;
- ◆ Ако консуматорска нишка пристигне в буфера, когато той е празен, тя се блокира, докато производителска нишка не добави нов обект.

Допускаме, че производителите и консуматорите изпълняват циклично следните парчета код:

$$\text{producers} \begin{cases} \text{event} = \text{waitForEvent}() \\ \text{buffer.add(event)} \end{cases} \quad \text{consumers} \begin{cases} \text{event} = \text{buffer.get}() \\ \text{event.process}() \end{cases}$$

Както споменахме, достъпа на нишките до буфера трябва да е ексклузивен, но от друга страна *waitForEvent()* и *event.process()* може да се изпълняват едновременно (не си пречат).

```
semaphore mutex.init(1)
semaphore items.init(0)
local event
```

```
PRODUCER
PRODUCE_ITEM
event = waitForEvent()
mutex.wait()
    buffer.add(event)
mutex.signal()
items.signal()
```

```
CONSUMER
items.wait()
mutex.wait()
event = buffer.extract()
    mutex.signal()
event.process()
CONSUME_ITEM
```

*mutex*-а осигурява ексклузивната работа на нишките в буфера. Когато *items* е с положителен брояч, той индикира броя на обектите в буфера, а когато е с отрицателен – броя на консуматорските нишки, които чакат в опашката на блокираните. *event* е локална променлива.

## 18. Памет и управление на паметта

Когато говорим за процесори и време, се интересуваме само от текущите процеси, които чакат и са готови да смятат и има какво да смятат. Спящите процеси не използват процесор, но когато говорим за памет – всички процеси използват памет, дори и спящите (те също са работещи програми, които се изпълняват). Физическите свойства на паметта са: енергозависимост (RAM са енергозависими, а HDD и flash паметите са енерго независими), време за достъп и многократно използване. Локалните данни, с които работи процеса, трябва да са си лично негови и да са предпазени (друг процес да не може да ги достъпва).

Процесора има два режима:

1. Real mode – програмата има достъп до цялата памет (до всички ресурси);
2. Protected mode – програмата няма достъп до цялата памет, а само до тази, която и е предоставена, както и до някои инструкции.

Техники за управление:

- ◆ Проста техника (сегментация): за паметта, която е предоставена на всеки процес имаме два регистъра: начало (f) и дължина (s) и проверяваме адреса на всеки байт, който използва този процес, дали е в този интервал, т.е.  $f \leq \text{address} < f + s$ ;
- ◆ IBM 360 техника (Paging): имаме  $n$  страници и всяка страница има атрибути, в които е записано как и кои процеси могат да използват тази страница.

В съвременната операционна система аналог на Paging, но с доста свойства и хардуер, който да го поддържа, е известната с името „виртуална памет“ (VM). Виртуалната памет е абстрактно понятие. Всеки процес си има VM. Няма съответствие между VM и RAM. На някои страници от VM се съпоставя страница от RAM, на други може нищо да не е съпоставено, а на трети може да е съпоставена страница от HDD (hard disk drive).

На всяка наша страница с данни се съпоставя страница от RAM и я използваме монополно. Може да има страници от RAM, които се използват от няколко процеса, но тези страници трябва да са само за четене. Затова трябва пак да има информация за всяка страница как ще се ползва, т.е. трябва да има тагове. Към всяка страница от VM има тагове, които казват къде е записана тя в RAM или HDD и казват за какво ще се ползва тази страница. За всяка страница от VM трябва да знаем къде се намира тя реално и тага ѝ. Тази информация се съдържа в таблица, която се нарича пряка адресна таблица, което означава, че за всеки процес имаме масив, който описва всяка негова страница.

Има и друг вариант да се поддържа обща структура, която да показва тези данни за страниците от VM на всеки процес и тогава ключът в тази таблица ще е: <номер на страница, номер на процес>, тази таблица се нарича обратна таблица.

Memory management unit – в съвременния процесор го има и се грижи за адресните таблици, които са масиви някъде в паметта и има регистър, който сочи към тях.

Видове кеш (три):

1. Кеш за инструкции – малко на брой дълги интервали, само се чете;
2. Кеш за данни – четене и писане на данни;
3. Кеш за таблиците за управление на виртуална памет.

TLB – Translation Lookahead Buffer:

- Кешове за няколко отделни таблици за управление на виртуална памет – съдържа 32-битови думи;
- Няма голям размер;
- Има две нива.

При процесите делим паметта между тях и даваме на всеки толкова, колкото му е необходима, но те са анонимни, те са парчета информация. При файлове, парчетата от памет трябва да са именувани и централния въпрос е за имената. Когато файла пази информация трябва да е в устойчивата памет. Когато променяме информация във файла, той трябва да е свързан с процес.

Паметта може да я представим като множество от файлове – дълготрайно живеещи в информационната среда – информационни обекти (виж т. 13). Тъй като файловете ще съществуват дълго (те са споделяни между различни потребители и процеси) са важни имената им.

## 19. Реализация на комуникационна тръба (pipe) чрез семафори

Предполагаме, че тръбата може да съхранява до  $n$  байта, подредени в обикновена опашка. Тръбата се използва от няколко паралелно работещи „изпращачи/получатели“ на байтове. Процесите изпращачи – слагат байтове в края на опашката, а получателите – четат байтове от началото на опашката.

За да реализираме комуникационната тръба ще използваме следните инструменти:

- ◆ 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-бързия процес (приспиването е бавна операция, тъй като включва в себе си смяна на контекста). В началото тази опашка ще е празна;
- ◆  $free\_bytes$  – семафор, който ще индикира за свободните байтове в опашката  $Q$ ;
- ◆  $ready\_bytes$  – семафор, който ще индикира колко байта са готови за четене (до колко е запълнена опашката  $Q$ );
- ◆  $mutex\_read$  – мутекс, който ще защитава критичната секция за четене;
- ◆  $mutex\_write$  – мутекс, който ще защитава критичната секция за писане.

Инициализация:

```
free_bytes.init(n)  
ready_bytes.init(0)  
mutex_read.init(1)  
mutex_write.init(1)
```

### READ

*byte b*

$P_1$

$P_2$

:

*ready\_bytes.wait()*

*mutex\_read.wait()*

$b = Q.get()$

*mutex\_read.signal()*

*free\_bytes.signal()*

:

### WRITE

*byte b*

$q_1$

$q_2$

:

*free\_bytes.wait()*

*mutex\_write.wait()*

$Q.put(b)$

*mutex\_write.signal()*

*ready\_bytes.signal()*

:

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 приспи. Ако не, той ще прочете байт и ще го запише в променливата  $b$ . 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  $b$  в опашката и ще сигнализира че още един байт е готов за четене, което се осъществява чрез подаване на сигнал на семафора  $ready\_bytes$ .

Кодовете по-горе, показващи как може да се реализира комуникационна тръба чрез семафори са валидни само за синхронни входно-изходни операции, тъй като процесите се приспиват, когато има недостиг на байтове и по този начин се изчаква за да може да се окомплектоват (виж т. 11).

## 20. Основни комуникационни канали в Linux

1. Неименувана тръба (pipe) се създава чрез системното извикване `pipe(fd[2])`. То връща два файлови дескриптора на мястото на елементите на подадения масив `int fd[2]`. `fd[0]` съхранява файловия дескриптор за четене, а `fd[1]` съхранява файловия дескриптор за писане. Тъй като тази тръба не е именувана, тя е видима само за процеса, който я е създал, както и от наследниците му (децата му, децата на децата му и т.н.). Тази тръба може да се използва само в системата, тоест не може да се използва в интернет. Тя служи за свързка между процес и друг процес, който има роднинска връзка с първоначалния процес. Тъй като тръбата е неименувана, тя не използва пространството на имената.
2. Именуванa тръба (FIFO) – този вид тръба се създава чрез библиотечното извикване `mkfifo(...)`. За разлика от неименуваната, този вид тръба е видима за всички процеси в системата и може да бъде ползвана от тях. Тя се използва за осъществяване на комуникация между два процеса, които не са задължително в роднинска връзка. Този вид тръба се обвързва с име, откъдето следва, че тя използва пространството на имената.
3. Писане във файл и четене от файл. Чрез системното извикване `open(filepath, open_flags, ...)` се създава файлов дескриптор, който сочи към единия край на комуникационния канал, който се изгражда в ядрото на операционната система и е за писане, а на другия край сочи към файла, който може да се разглежда като много прост абстрактен процес, служещ само за съхраняване на данни. Чрез системните команди `read(int fd, void *buf, size_t cnt)` и `write(int fd, const void *buf, size_t cnt)` може да се чете и пише от и във файла. Този вид комуникационен канал използва пространството на имената (виж т. 6).
4. Конекция (socket) (виж т. 16). Това е именуван обект, който играе ролята на крайна точка за комуникация между отдалечени обекти и който се използва за адрес при изграждането на връзка с друг процес. Тук има два вида процеси – единият се нарича сървър и работи като такъв (ще обслужва други процеси), а другият – клиент. И двата вида процеси ще трябва да изпълнят различни поредици от извиквания, за да изградят връзка помежду си.

Сървърът трябва да изпълни следната процедура:

```
server_fd = socket(domain, type, protocol); // създава конекцията
bind(server_fd, &my_addr, addrlen); // именува конекцията
listen(server_fd, backlog); /* слуша за заявки от други процеси за изграждане на
                               връзка (backlog се нарича опашката, в която се
                               съхраняват заявките) */
client_fd = accept(server_fd, &peer_addr, addrlen); /* приема заявката за изграждане на
                                                       връзка, като връща файловия
                                                       дескриптор на клиента */
```

Клиента трябва да изпълни следните две извиквания:

```
fd = socket(domain, type, protocol); // създава конекция
connect(fd, &server_addr, addrlen); /* подава заявка за изграждане на връзка със
                                       сървъра */
```

(за правилния ред на системните извиквания виж фиг. 9 от т. 16)

Предимствата на сокетите пред тръбите е, че процесите, с които може да се изгради връзка, могат да не се намират в една и съща система. Сокета на сървъра използва пространството на имената, тъй като трябва да му бъде зададено име, с което да бъде видимо за останалите процеси. Сокета на клиента обаче не използва пространството на имената, тъй като не се обвързва с име.

## Източници:

- [1] д-р Георги Георгиев – Скелета, Лекции от ФМИ, <http://skelet.ludost.net/>
- [2] Allen B. Downey, The Little Book of Semaphores
- [3] Andrew S. Tanenbaum, Herbert Bos, Modern Operating Systems