

# ПРОЦЕСИ

В операционните системи понятието процес е централно, всичко останало, включително и абстракцията файл, се гради върху него. Съвременните компютри са способни да изпълняват няколко операции едновременно. Докато централният процесор (ЦП) изпълнява команди, дисковото устройство може да чете и терминалът или принтерът да извеждат данни. Съществуването на този реален паралелизъм довежда до революционна за операционните системи идея, която в началото бе наречена не много удачно **мултипрограмиране (multiprogramming)**. В оперативната памет са заредени няколко програми и макар, че ЦП във всеки един момент изпълнява само една команда, той може да се превключва от изпълнение на една програма към друга и да го прави много бързо. Първоначално мултипрограмирането се въвежда с цел по-ефективното използване на ресурсите на компютъра. Но също така у потребителя се създава впечатление за едновременното изпълнение на няколко програми, което е в същност една илюзия, поддържана от операционната система. Истината е, че ЦП изпълнява няколко последователни дейности, като за тази цел трябва да се съхранява информация за тези дейности, за да е възможно да се възстановява изпълнението на прекъсната дейност. За да се изолира потребителя от детайлите по поддържането на този псевдо паралелизъм, е необходим модел, който да опрости работата на човека в операционната система.

## 1. МОДЕЛ НА ПРОЦЕСИТЕ

В такъв един модел се въвежда понятие за обозначаване на дейността по изпълнение на програма(и) за определен потребител. В различни операционни системи са използвани понятията **задание (job)**, **задача (task)**, **процес (process)**. Терминът процес за първи път е бил използван в операционната система MULTICS на MIT през 60-те години и преобладава в съвременните операционни системи. Най-краткото определение за **процес е програма в хода на нейното изпълнение**. Следователно, има връзка между понятията процес и програма, но те не са идентични. За разлика от програмата, която е нещо статично - файл записан на диска и съдържащ изпълним код, процесът е дейност. В понятието процес освен програмата се включват и текущите стойности на регистъра PC (programm counter), на другите регистри, на програмните променливи, състоянието на отворените файлове и други. В операционна система, която реализира такъв един модел, всичкия софтуер работещ на компютъра е организиран в процеси, т.е. ЦП винаги изпълнява процес и нищо друго. Когато операционната система поддържа едновременното съществуване на няколко процеса се казва, че е многопроцесна, в противен случай е еднопроцесна. Операционните системи UNIX, MINIX и LINUX са типични многопроцесни системи.

### 1.1. ЙЕРАРХИЯ НА ПРОЦЕСИТЕ

Операционна система, която реализира абстракцията процес, трябва да предоставя възможност за създаване на процеси и за унищожаване на процеси, а също така и начин за идентифициране (именуване) на процес.

В UNIX, LINUX и MINIX процес се създава със системния примитив `fork`. В тези системи най-точното определение за **процес е обект, който се създава от `fork`**. Когато един процес изпълни `fork` ядрото създава нов процес, който е почти точно негово копие. Първият процес се нарича процес-баща, а новият е процес-син. След `fork` процесът-баща продължава изпълнението си паралелно с новия процес-син. Следователно, след това процесът-баща може да създаде с `fork` и други процеси-синове, т.е. един процес може едновременно да има няколко процеса-синове. Процесът-син също може да изпълни `fork` и да създаде свой процес-син. Следователно, ако разглеждаме връзките на пораждаване на процеси, т.е. връзката баща-син, то всички едновременно съществуващи процеси са свързани в йерархия. Всеки процес се идентифицира чрез уникален номер, наричан ***pid (process identifier)***, който освен, че е уникален за процеса е и неизменен през целия му живот.

Нека разгледаме какво се случва при стартиране на UNIX или LINUX и някои важни процеси, които се създават. В същност първият процес не може да бъде създаден нормално, тъй като преди него няма друг процес. Програмата за начално зареждане, която е в `boot` блока,

зарежда ядрото в паметта и предава управлението на модула `start`, който инициализира структурите в ядрото (системните таблици) и ръчно създава процес с `pid 0`. От тук нататък ядрото продължава да работи като процес с `pid 0` и създава нормално с `fork` процес с `pid 1`, в който пуска за изпълнение програмата `init`. След това процес 0 създава няколко други процеси, които се наричат системни процеси (`kernel processes`), и самия той става системен процес (това важи за някои версии).

Процес `init` е първият нормално създаден процес и затова ядрото го счита за корен на дървото на процесите. Той се грижи за инициализация на процесите. Когато процес `init` заработи, чете файл `/etc/inittab` и създава процеси според съдържанието му. Например, за всеки терминал в системата `init` създава процес, в който пуска за изпълнение специална програма - `getty` в повечето версии на UNIX (`mingetty` в LINUX). Програмата `getty` чака вход от съответния терминал и когато той постъпи приема, че потребител иска вход в системата. Тогава програмата `getty` в процеса се сменя с програмата `login`, която извършва идентификация на потребителя и при успех се сменя с програмата `shell` за съответния потребител (или поражда нов процес за програмата `shell` в някои версии). Следователно, в живота на един процес могат последователно да се сменят различни програми, които той изпълнява, а също така няколко едновременно съществуващи процеса могат да изпълняват една и съща програма, но те са различни обекти за ядрото.

След като процесът `init` приключи с инициализацията на процесите според описанието в `/etc/inittab`, той изпълнява безкраен цикъл, в който чака завършване на свой процес-син и изпълнява довършителни дейности.

## 1.2. СЪСТОЯНИЕ НА ПРОЦЕС

В многопроцесните операционни системи едновременно съществуват много процеси, а ЦП е един и във всеки един момент може да изпълнява само един от тези процеси. За този процес казваме, че се намира в състояние текущ (`running`). Останалите процеси са в някакво друго състояние. Най-опростеният модел на процесите включва три вида състояния:

### 1. текущ ( **running** )

ЦП изпълнява команди на процеса.

### 2. готов ( **ready** )

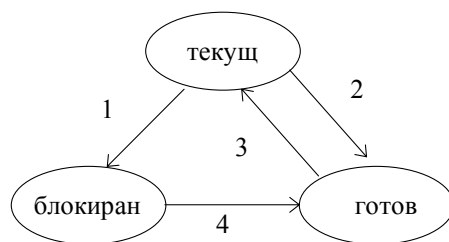
Процесът може да продължи изпълнението си, ако му се предостави ЦП, който в момента се използва от друг процес.

### 3. блокиран ( **blocked** )

Процесът чака настъпването на някакво събитие, много често завършването на входно-изходна операция.

Логически състоянията 1 и 2 са подобни, в смисъл, че и в двата случая процесът логически е работоспособен, но при състояние 2 няма свободен ЦП. При състояние 3 е различно, процесът логически не може да работи дори ако ЦП няма какво друго да прави. На Фиг.1 е изобразена диаграмата на състоянията и възможните преходи от едно състояние в друго, които са показани с дъги.

Между тези три състояния са възможни четири прехода. Преход 1 се случва когато процес открие, че не може да продължи изпълнението си, например докато не завърши входно-изходната операция, която той е поискал, изпълнявайки системния примитив `read`. Преход 4 се извършва когато настъпи събитието, което процесът чака. Преходи 2 и 3 се управляват от модул на операционната система, наричан *планировчик* (*scheduler*), без процесът да ги желае и дори да знае за тях. Когато планировчикът реши, че процес достатъчно дълго е използвал ЦП, той насилствено му го отнема, за да го предостави на друг готов процес, т.е. извършва за него прехода 2. Когато ЦП се освободи поради блокиране, завършване или сваляне на текущия процес, планировчикът избира един от готовите процеси за текущ, т.е. извършва за него прехода 3. Планирането, т.е. решаването кой процес да работи, кога и колко време, е важна функция в многопроцесните операционни системи, критична за производителността им. Планиране, при което се реализира прехода 2 се нарича планиране с преразпределение на ЦП (*preemptive scheduling*). Планирането ще бъде разгледано по-нататък.



Фиг. 1. Модел на процесите с три състояния и диаграма на преходите

Описаният модел дава начална представа за понятието състояние на процес, но е прекалено опростен, например в него не е ясно как работи ядрото. Моделът на процесите в UNIX включва девет състояния:

**1. текущ в потребителска фаза ( user running )**

Процесът се изпълнява в потребителски режим, като ЦП изпълнява команди от потребителската програма свързана с процеса.

**2. текущ в системна фаза ( kernel running )**

Процесът се изпълнява в режим ядро, като ЦП изпълнява команди от ядрото, т.е. от името на процеса работят модули на ядрото, които вършат системна работа.

**3. готов в паметта ( ready in memory )**

Процесът е готов за изпълнение и се намира в паметта.

**4. блокиран в паметта ( blocked in memory )**

Процесът чака настъпването на някакво събитие и се намира в паметта.

**5. готов на диска ( ready, swapped )**

Процесът е готов за изпълнение, но планировчикът трябва да го зареди в паметта преди да може да бъде избран за текущ.

**6. блокиран на диска ( blocked, swapped )**

Процесът е блокиран и планировчикът го е изхвърлил от паметта в специална област на диска - свопинг област, която представлява разширение на паметта, за да освободи място за други процеси.

**7. преразпределен ( preempted )**

Процесът е бил на път да се върне в състояние 1, след като е завършила системната фаза - състояние 2, но планировчикът му е отнел ЦП насилствено, за да го предостави на друг процес (преразпределил е ЦП).

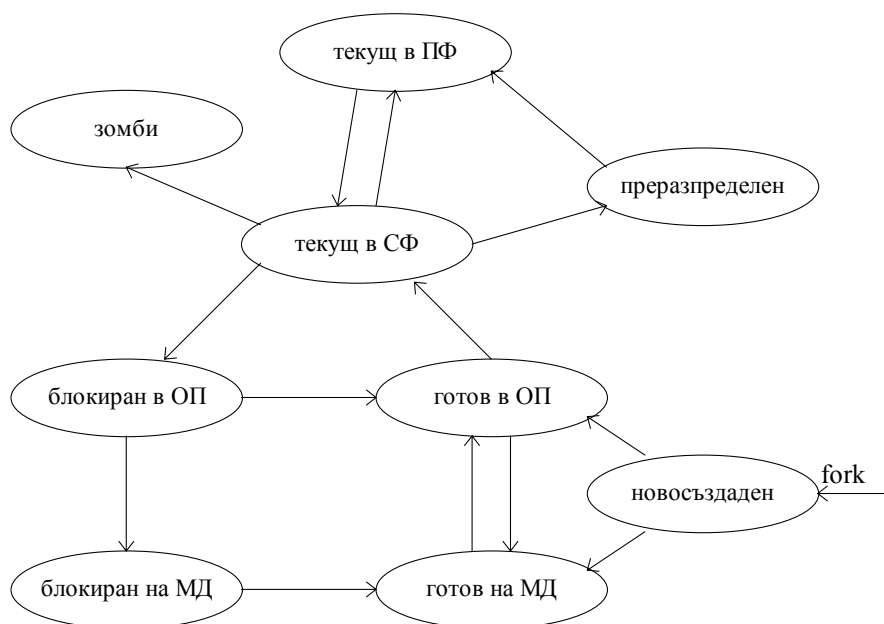
**8. новосъздаден ( created )**

Това е началното състояние, в което процес влиза в системата. Той е почти създаден, но още не е напълно работоспособен. Това е преходно състояние при създаване на всеки процес.

**9. зомби ( zombie )**

Процесът е изпълнил системния примитив `exit` и вече не съществува, но от него все още има някакви следи в системата. Съхранява се информация за него, която може да бъде предадена на процеса-баща. Това е крайното състояние на всеки процес преди да изчезне напълно от системата.

На Фиг. 2 са изобразени възможните преходи от едно състояние в друго.



Фиг. 2. Диаграма на състоянията и преходите в UNIX

Сега да разгледаме събитията, които предизвикват смените на състоянията. Всеки процес влиза в системата в състояние "новосъздаден", когато процес-баща изпълнява `fork`. След това преминава в състояние "готов в паметта" или "готов на диска" в зависимост от ядрото и наличните в момента свободни ресурси. Нека приемем, че процесът е преминал в състояние "готов в паметта". След време планировчикът ще го избере за текущ и той ще влезе в състояние "текущ в системна фаза", където ще довърши своята част от `fork`. След това той може да премине в състояние "текущ в потребителска фаза", където ще започне изпълнението на потребителската си програма. След време ще настъпи някакво прекъсване или в потребителската програма ще има системен примитив. И двете събития ще предизвикат преход в състояние "текущ в системна фаза". Когато завърши обработката на прекъсването, ако това е била причината за вход в състоянието, ядрото може да реши да върне процеса обратно в състояние "текущ в потребителска фаза", или да го свали от ЦП, т.е. да извърши преход в състояние "предазпределен". Ако причината за прехода от потребителска към системна фаза е била извикване на системен примитив, то за някои примитиви, например `read` или `write`, се налага процесът да изчака, т.е. да премине в състояние "блокиран в паметта". Когато входно-изходната операция завърши, устройството ще предизвика прекъсване и някой друг процес, който в момента е в състояние "текущ в потребителска фаза", ще влезе в състояние "текущ в системна фаза", а модулет обработващ прекъсването ще смени състоянието на първия процес от "блокиран в паметта" в "готов в паметта".

В многопроцесна система обикновено е невъзможно всички процеси да се съхраняват едновременно в паметта. Затова част от процесите временно се съхраняват в специална област на диска. Тази техника се нарича свопинг (`swapping`), а дисковата област - свопинг област. Специален планировчик (`swapper`) решава кой от процесите в състояние "блокиран в паметта" или "готов в паметта" да бъде изхвърлен временно от паметта и кой от процесите в свопинг областта да бъде върнат обратно в паметта, т.е. управлява преходите на състояния 3 в 5, 5 в 3 и 4 в 6.

Състоянията "готов в паметта" и "предазпределен" са подобни в смисъл, че процесът е готов за изпълнение когато му се предостави ЦП, но са отделени в модела, за да се подчертае факта, че процес работещ в системна фаза не може да бъде свален докато не я завърши. Следователно свопинг на процес може да се извършва и от състояние "предазпределен", т.е. има преходи от състояние 7 в 5 и обратно, които не са изобразени на Фиг. 2.

Когато процес завършва той изпълнява системния примитив `exit` и състоянието му се сменя от "текущ в потребителска фаза" в "текущ в системна фаза". Когато системната фаза на `exit` завърши състоянието се сменя в "зомби". В това състояние процесът остава докато

неговият процес-баща не се погрижи чрез специален системен примитив `wait`, след което процесът напълно изчезва от системата.

## 2. КОНТЕКСТ НА ПРОЦЕС

За да се реализира този модел и възможността операционната система да управлява преходите, трябва да се съхранява информация за процесите. Сега ще разгледаме структурите в паметта, които представят един процес или от какво се състои един процес, имайки предвид основно UNIX и LINUX, въпреки, че принципите са общи за операционните системи. В UNIX системите всичко, което реализира един процес, се нарича контекст на процес. Компонентите на контекста могат да се разделят на три части:

**Потребителска част** - определя работата на процеса в потребителска фаза и включва образа на процеса.

**Машинна (регистрова) част** - съдържанието на машинните регистри - PC, съдържащ адреса на следващата машинна команда; PSW, определящ режима на ЦП по отношение на процеса, признак за резултата от последната команда и др.; други регистри, съдържащи данни на процеса.

**Системна част** - структури в пространството на ядрото, описващи процеса, някои от които ще разгледаме по-нататък.

### 2.1. ОБРАЗ НА ПРОЦЕС

Образът на процеса съдържа програмния код и данните, използвани от процеса в потребителска фаза. Той се създава като се използва файл с изпълним код. Обикновено образът на процеса се състои от логическите единици:

- **код (text)** - съдържа машинните команди, изпълнявани от процеса в потребителска фаза. Зарежда се от изпълнимия файл.
- **данни (data)** - съдържа глобалните данни, с които процесът работи в потребителска фаза. Инициализираните данни се зареждат от изпълнимия файл. За неинициализираните данни в изпълнимия файл се съхранява размера им и при създаване на образа на процеса се отделя съответния обем памет.
- **стек (stack)** - създава се автоматично с размер определян от ядрото. Чрез него се реализира обръщението към потребителски функции. Той представлява стек от слоеве, като има по един слой за всяка извикана функция, която още не е изпълнила `return`. Всеки слой съдържа данни, локални за функцията и достатъчно данни за връщане от функция. Това е стекът, използван при работа на процеса в потребителска фаза. При работа в системна фаза се използва друг стек - стек на ядрото, който има същата структура, но в него се записват данни при обръщение към функции от ядрото.

В UNIX системите тези логически единици, на които се дели образа на процеса, се наричат области или региони (*regions*). Всеки регион е последователна област от виртуалното адресно пространство на процеса и се разглежда като независим обект за защита или съвместно използване. Например, регион за код обикновено е *read-only*. Това позволява няколко процеса, които изпълняват една и съща програма, да могат да разделят достъпа до един и същи регион за код, т.е. да използват едно копие. Има и други типове региони, които могат да са общи за няколко процеса, например **обща памет (shared memory)**. Следователно, на базата на регионите, няколко процеса могат да делят части на своите образи, но общите региони се считат за част от образа на всеки от тези процеси. Понятието регион е логическо и не зависи от реализираното управление на паметта, което ще разгледаме по-нататък.

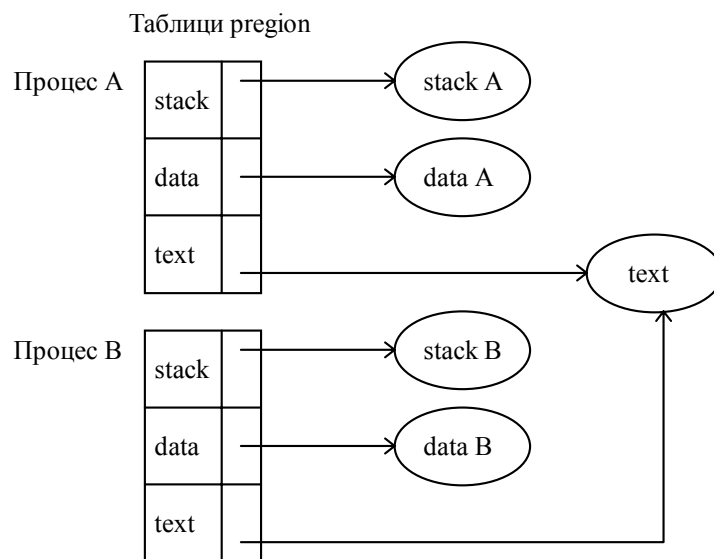
Информацията за всички активни региони се съхранява в **Таблица на регионите**. Всеки запис в тази таблица описва един регион и съдържа:

- тип на региона - код, данни, стек и др.;
- размер на региона;
- адрес на региона в паметта;
- състояние на региона - заключен, зарежда се в паметта и др.;
- брой процеси, използващи региона.

Тъй като един регион може да е общ за няколко процеса, то всеки процес има собствена системна структура, описваща неговите региони - **Таблица pregion (Per process region table)**. Всеки запис в тази таблица описва един регион на процеса и съдържа:

- тип на разрешения на процеса достъп до региона - read-only, read-write, read-execute;
- виртуален адрес на региона в процеса;
- указател към запис от Таблицата на регионите.

Таблицата pregion и съответните записи от Таблицата на регионите са част от системното ниво на контекста на процес, както и други системни таблици, необходими за изобразяване на виртуалното адресно пространство във физическо, в зависимост от реализираното управление на паметта. На Фиг.3 са изобразени два процеса, които разделят достъпа до общ регион за код, т.е. изпълняват една програма.



Фиг. 3. Процеси и региони

## 2.2. ТАБЛИЦА НА ПРОЦЕСИТЕ

Тази структура представлява масив от записи в пространството на ядрото, като всеки запис описва един процес. Записът съдържа информация за процеса, всичко което ядрото трябва да знае независимо от състоянието му. Освен наименованието таблица на процесите, в литературата се срещат и други, като **дескриптор на процес, блок за управление на процес**. Независимо от терминологичните различия структура с такова предназначение присъства във всяка операционна система. Така че, друго определение за процес, което можем да приемем, е: **процес е обект, който е описан в запис от таблицата на процесите**.

Точната структура на запис от таблицата на процесите е различна в различните операционни системи, дори за UNIX и LINUX системите няма абсолютна еднаквост. Информацията, описваща един процес в тези системи, обикновено е разделена в две системни структури:

**Таблица на процесите** - съдържа данни за процеса, които са нужни и достъпни за ядрото независимо от състоянието на процеса.

**Потребителска област (User area или U area)** - съдържа данни за процеса, които са необходими и достъпни за ядрото само когато той е в състояние текущ.

Следват някои от полетата, които се включват в **таблицата на процесите**:

- идентификатор на процеса (pid)
- идентификатор на процеса-баща (ppid - parent pid)
- идентификатор на група процеси (pid на процеса-лидер на групата)
- идентификатор на сесия (pid на процеса-лидер на сесията)
- състояние на процеса
- събитие, настъпването на което процеса чака в състояние блокиран
- полета, осигуряващи достъп до образа на процеса и U area (адрес на таблица pregion)

- полета, определящи приоритета на процеса при планиране
- полета, съхраняващи времена - използваното от процеса време на ЦП в системна и потребителска фази и други
- код на завършване на процеса

Следващите полета са в **потребителската област**:

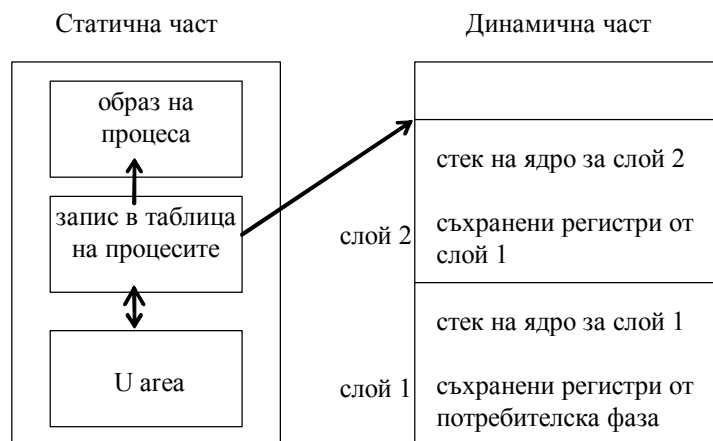
- указател към записа от таблицата на процесите
- файлови дескриптори
- текущ каталог на процеса
- управляващ терминал на процеса или NULL ако няма такъв
- реален потребителски идентификатор (ruid) - определя потребителя, създал процеса
- ефективен потребителски идентификатор (euid) - определя правата на процеса
- реален идентификатор на потребителска група (rgid)
- ефективен идентификатор на потребителска група (egid)
- полета за параметри на текущия системен примитив и върнати от него стойности

### 2.3. СТЕК НА ЯДРОТО И ДИНАМИЧНА ЧАСТ ОТ КОНТЕКСТА НА ПРОЦЕС

Когато процес работи в системна фаза е необходим стек на ядрото. Всеки процес трябва да има свой собствен стек на ядрото, който да съответства на неговото обръщение към ядрото. Освен това, когато процес преминава от потребителска в системна фаза е необходимо преди това да се съхрани съдържанието на машинните регистри, за да може по-късно процесът да се върне към прекъснатата потребителска фаза. Следователно за всеки процес контекстът включва и стек на ядрото и област за съхранение на регистрите.

В същност нещата са по-сложни, тъй като в диаграмата на преходите (Фиг.2) не е показан един преход от състояние 2 (текущ в системна фаза) в състояние 2 и обратно. Докато процес работи в системна фаза, обработвайки системен примитив или прекъсване, може да настъпи прекъсване с по-висок приоритет. Тогава процесът прекъсва първата системна фаза и започва нова системна фаза, което означава, че трябва да се съхранят регистрите от старата системна фаза и да се започне нов стек на ядрото за новата системна фаза. Така когато приключи обработката на новата системна фаза ще е възможно процесът да се върне към прекъснатата системна фаза.

Затова контекстът включва динамична част, която представлява стек от слоеве. Слой се записва при прекъсване или системно извикване и включва съхранените регистри от работата на процеса преди прекъсването и стек на ядрото, използван при обработка на новото прекъсване. Слой се изключва при връщане след обработка на прекъсването или на системния примитив. Когато процес работи в потребителска фаза, динамичната част от контекста е празна. Процес, работещ в системна фаза, се изпълнява в контекста на последния записан слой. Броят на нивата на прекъсване е хардуерно зависим и това ограничава максималния брой слоеве в динамичната част. Фиг.4 илюстрира компонентите, съставляващи контекста на процес.



Фиг. 4. Компоненти на контекста на процес

Регистровият контекст на процес трябва да се съхранява и когато ЦП се отнема от текущия процес (процесът преминава в състояние преразпределен) и да се възстановява когато планировчикът отново избере процеса за текущ. Смяната на контекста на текущия процес с контекста на друг процес се нарича *превключване на контекста (context switch)*. Има три случая, в които се прави превключване на контекст:

- Текущият процес се блокира.
- Текущият процес е изпълнил системния примитив `exit` и минава в състояние зомби.
- Текущият процес е завършил системната си фаза, след обработка на системен примитив или на прекъсване и ще трябва да се върне в потребителска фаза, но има готов процес с по-висок приоритет и планировчикът решава да свали текущия процес от ЦП.

И в трите случая текущият процес, който е в състояние "текущ в системна фаза", излиза от това състояние (всички системни операции са завършени и структурите в ядрото са в коректно състояние), тъй като не може или не трябва да продължи изпълнението си. Тогава ядрото трябва да избере друг готов процес (в състояние "готов в паметта" или "преразпределен") за текущ. Стъпките, които ядрото изпълнява са следните:

1. Решава дали да прави превключване на контекста.
2. Съхранява контекста на "стария" процес, т.е. записва слой в динамичната част на неговия контекст (push в динамичната част).
3. Избира "най-подходящия" процес за текущ, използвайки алгоритъма за планиране.
4. Възстановява контекста на избрания процес, използвайки най-горния слой в динамичната част на неговия контекст (pop от динамичната част). От тук нататък системата продължава да работи в контекста на "новия" процес.

#### 2.4. СИСТЕМНИ ПРИМИТИВИ - ИНТЕРФЕЙС И ИЗПЪЛНЕНИЕ

В програма на С извикването на системен примитив изглежда като обръщение към функция, но всеки системен примитив е вход в ядрото. Как се реализира това? За всеки системен примитив стандартната библиотека на С включва функция, в някои случаи има няколко функции за един системен примитив. Тези функции се свързват с потребителската програма и се изпълняват в потребителска фаза. Това, което извършват тези функции, е по принцип едно и също за всеки системен примитив, а именно:

1. Зарежда номера системния примитив в регистър.
2. Изпълнява команда, предизвикваща програмно прекъсване (trap).
3. След връщане от trap проверява за грешка регистъра PSW и преобразува връщаните стойности към формата, използван от функциите на системните примитиви (кода на грешката в глобалната променлива `errno` и връщаната от функцията стойност в регистър 0).

Следователно фактическата обработка на системните примитиви се върши от модули в ядрото, а кода на библиотечните функции служи като обвивка, която осигурява по-удобен потребителски интерфейс. За простота на езика обаче, ние ще наричаме библиотечните функции системни примитиви.

При изпълнение на trap в библиотечната функция заработва системата за прекъсване. Ядрото обработва всички прекъсвания по следния сценарий:

1. Съхранява текущия регистров контекст, като записва слой в динамичната част (слой 1).
2. Определя източника на прекъсване и от вектора на прекъсване извлича адреса на съответния обработчик на прекъсването.
3. Извиква обработчика на прекъсването.
4. Възстановява съхранения в стъпка 1 слой на контекста, т.е. става връщане от прекъсване.

При програмно прекъсване управлението в стъпка 3 на горния сценарий се предава на модул от ядрото, който получава като вход номера на системния примитив. Следва алгоритъма на този модул.



### Алгоритъм на `syscall` (номер на примитива).

1. Намира записа в таблицата на системните примитиви, който съответства на получения номер, т.е. определя адреса на модула в ядрото за системния примитив и необходимия брой параметри.
2. Копира параметрите от потребителския стек в област на ядрото - U area.
3. Извиква модула в ядрото за системния примитив.

Връщаните от системния примитив стойности се записват в областта за съхранените регистри от потребителска фаза (слой 1 от динамичната част на контекста). При грешка в регистър PSW се вдига бит, а в регистър 0 - код на грешката, иначе при нормално завършване в регистри 0 и 1 - връщаните от системния примитив стойности.

## 3. СИСТЕМНИ ПРИМИТИВИ ЗА ПРОЦЕСИ

Ще разгледаме системните примитиви по стандарта POSIX, които реализират основните операции за процеси, а именно създаване на процес, завършване на процес и др.

### Създаване на процес

`pid_t fork(void);`

Единственият начин да се създаде процес в UNIX и LINUX системите е чрез `fork`. Процесът, който изпълнява `fork` се нарича процес-баща, а новосъздаденият е процес-син. При връщане от `fork` двата процеса имат еднакви образи, с изключение на връщаната стойност: в процеса-баща `fork` връща `pid` на процеса-син при успех или `-1` при грешка, а в сина връща `0`. Сложността при този примитив се състои в това, че един процес "влиза" във функцията `fork`, а два процеса "излизат от" `fork` с различни връщани значения. Да разгледаме по-подробно това, което става когато процес-баща изпълнява `fork` и така ще видим какво е общото между процесите баща и син.

### Алгоритъм на `fork`

1. Определя уникален идентификатор за новия процес и създава запис в таблицата на процесите, като инициализира полетата в него (група и сесия от процеса-баща, състояние "новосъздаден" и т.н.).
2. Създава U area, където повечето от полетата се копират от процеса-баща (файловете дескриптори, текущ каталог, управляващ терминал, потребителските идентификатори - `euclid`, `ruid`, `egid` и `rgid`).
3. Създава образ на новия процес - копие на образа на процеса-баща (региона за код обикновено е общ за двата процеса).
4. Създава динамичната част от контекста на новия процес: Слой 1 е копие на слой 1 от контекста на бащата. Създава слой 2, в който записва съхранените регистри от слой 1, като регистър PC е изменен така, че синът да започне изпълнението си в `fork` от стъпка 7.
5. Изменя състоянието на процеса-син в "готов в паметта".
6. В процеса-баща връща `pid` на новосъздадения процес-син.
7. В процеса-син, връща `0`, т.е. когато по-късно планировчикът избере новосъздадения процес за текущ той ще заработи според най-горното ниво на динамичната част от контекста си (слой 2) - в системна фаза на `fork` и ще върне `0`.

И така процесите син и баща имат следното общо помежду си:

- Двата процеса изпълняват една и съща програма. Дори процесът-син, който преди това не е работил, започва изпълнението на потребителската програма от оператора след `fork`. Но връщаните стойности в двата процеса са различни, което позволява да се определи кой процес е бащата и кой сина и да се раздели тяхната функционалност макар, че изпълняват една и съща програма.
- Процесът-син наследява от бащата файловете дескриптори, т.е. двата процеса ще разделят достъпа до файловете, които бащата е отворил преди да изпълни `fork`. Това позволява пренасочване на входа и изхода и свързване на процеси с програмни канали.

- Двата процеса имат един и същ текущ каталог, управляващ терминал, група процеси и сесия.
- Двата процеса имат еднакви права - реален и ефективен потребителски идентификатори, което гарантира неизменност на привилегиите на потребител при работа в системата.

Има два начина за използване на `fork`. Процес иска да създаде свое копие, което да изпълнява една операция, докато другото копие изпълнява друга, напр. при процеси сървери. Процес иска да извика за изпълнение друга програма, тогава първо създава свое копие, след което в едното копие (обикновено в процеса-син) се извиква другата програма, напр. при командния интерпретатор.

### Завършване на процес

**`void exit(int status);`**

Системата не поставя ограничение за времето на съществуване на процес. Има процеси, например процес `init` (с `pid 1`), които съществуват вечно в смисъл от `boot` до `shutdown`. Когато процес завършва той изпълнява `exit`. Процесът може явно да го извика или неявно в края на програмата, но може и ядрото вътрешно да извика `exit` без знанието на процеса, например когато процеса получи сигнал, за който не е предвидил друга реакция. Значението на аргумента е кода на завършване, който се предава на процеса-баща, когато той се заинтересува. Този системен примитив не връща нищо, защото няма връщане от него, винаги завършва успешно и след него процесът почти не съществува, т.е. той става зомби.

### Алгоритъм на `exit`

1. Изпълнява `close` за всички отворени файлове и освобождава текущия каталог.
2. Освобождава паметта, заемана от образа на процеса и `U area`.
3. Сменя състоянието на процеса в зомби. Записва кода на завършване в таблицата на процесите. Ако процес завършва по сигнал, код на завършване е номера на сигнала.
4. Урежда изключването на процеса от йерархията на процесите. Ако процесът има синове, то техен баща става процесът `init` и ако някой от тези синове е зомби изпраща сигнал "death of child" на `init`. Изпраща същия сигнал и на процеса-баща на завършващия процес.

### Изчакване завършването на процес-син

**`pid_t wait(int *status);`**

Процес-баща може да синхронизира работата си със завършването на свой процес-син, т.е. да изчака неговото завършване ако още не завършил и да разбере как е завършил, чрез `wait`. Функцията на системния примитив връща `pid` на завършилия син, а чрез аргумента `status` кода му на завършване.

### Алгоритъм на `wait`

1. Ако процесът няма синове, връща -1.
2. Ако процесът има син в състояние зомби, т.е. синът вече е изпълнил `exit`, освобождава записа му от таблицата на процесите, като взема кода на завършване и неговия `pid` и ги връща.
3. Ако процесът има синове, но никой от тях не е зомби, той се блокира, като чака сигнал "death of child". Когато получи такъв сигнал, което означава, че някой негов син току що е станал зомби, продължава както в точка 2.

Сега след като разгледахме системните примитиви `fork`, `exit` и `wait`, става по-ясно значението на състоянието зомби. След `fork` двата процеса - баща и син съществуват едновременно и се изпълняват асинхронно. Това означава, че бащата може да изпълни `wait` както преди така и след `exit` на сина. Освен това не бива да се задължава процес-баща да изпълнява `wait`, той може да завърши веднага след като е създал син. Но тогава в системата ще останат вечни зомбита, които заемат записи в таблицата на процесите. Затова когато процес

завършва неговите синове се осиновяват от процеса `init`, който се грижи за изчистване на системата от зомбита-сираци.

### Изпълнение на програма

Когато с `fork` се създава нов процес, той наследява образа си от бащата, т.е. продължава да изпълнява същата програма. Но чрез `exec` всеки процес може да смени образа си с друга програма по всяко време от своя живот, както веднага след създаването си така и по-късно, дори няколко пъти в своя живот. За този системен примитив има няколко функции в стандартната библиотека, които се различават по начина на предаване на аргументите и използване на променливите от обкръжението на процеса. Следва синтаксиса на част от тях.

```
int execl(const char *name, const char *arg0
          [, const char * arg1]..., 0);

int execlp(const char *name, const char *arg0
          [, const char * arg1]..., 0);

int execv(const char *name, const char *argv[ ]);

int execvp(const char *name, const char *argv[ ]);
```

Първият аргумент `name` указва към името на файл, съдържащ изпълним код, от който ще се създаде новия образ. При `execvp` и `execlp` `name` може да е собствено име на файл и тогава файлът се търси в каталозите от променливата `PATH`. В останалите случаи `name` трябва да е пълно име на файл. Останалите аргументи в `execl` и `execlp` са указатели към параметрите, които ще се предадат на функцията `main` когато новият образ започне изпълнението си. Във функциите `execv` и `execvp` има един аргумент `argv`, който е масив от указатели на аргументите за функцията `main`, който също трябва да завършва с елемент `NULL`.

### Алгоритъм на `exec`

1. Намира файла, чието име е в аргумента `name` и проверява дали процесът има право за изпълнение на този файл.
2. Проверява дали файлът съдържа изпълним код.
3. Освобождава паметта, заемана от стария образ на процеса.
4. Създава нов образ на процеса, използвайки изпълнимия код във файла и копира аргументите на `exec` в новия потребителски стек.
5. Изменя значения на някои регистри в областта за съхранени регистри в слой 1 от динамичната част на контекста, например на `PC`, указател на стек. Така, когато процесът се върне в потребителска фаза ще заработи от началото на функцията `main` на новия образ.
6. Ако програмата е `set-UID` прави съответните промени на потребителските идентификатори на процеса.

При успех, когато процесът се върне от `exec` в потребителска фаза, той изпълнява кода на новата програма, започвайки от началото, но това си остава същия процес. Не е променен идентификатора му, позицията му в йерархията на процесите дори голяма част от потребителската област, като файловете дескриптори, текущия каталог, управляващия терминал, групата на процесите, сесията. При грешка по време на `exec` става връщане в стария образ, така че функцията връща `-1` при грешка, а при успех не връща нищо защото няма връщане в стария образ.

Разделянето на създаването на процес и извикването на програма за изпълнение в два системни примитива играе важна роля. Това дава възможност програмата на процес-баща да определя поне в началото функционалността на свой процес-син и например, да се реализира пренасочване на входа и изхода, и свързването на роднински процеси чрез програмни канали (заслуга за това има и наследяването на файловете дескриптори).

### Информация за процес

```
pid_t getpid(void);
pid_t getppid(void);
```

Системният примитив `getpid` връща идентификатора на процеса, който го изпълнява, а `getppid` връща идентификатора на неговия процес-баща. И двата примитива винаги завършват успешно.

**Пример.** Програмата илюстрира разгледаните системни примитиви.

```
main()
{
    int pid, status;
    pid = fork();
    if (pid == 0) {          /* in child process */
        execl("/bin/ls", "ls", "-l", 0);
        perror("Cannot exec ls. ");
        exit(1); }
    else                    /* in parent process */
        if (pid < 0)
            perror("Cannot fork. ");
        else {
            wait(&status);
            printf("Parent after death of child: status=%d.\n",
                status);}
}
```

Процесът, в който се изпълнява програмата създава нов процес. В процеса-син се изпълнява командата `ls`. Процесът баща изчаква завършването на сина и извежда съобщение за кода му на завършване с функцията `printf`. Съобщенията за грешки се извеждат с функцията `perror`. Тя извежда на стандартния изход за грешки, като освен текста в аргумента си извежда и системно съобщение за грешката в изпълнението преди нея системен примитив.

#### 4. МЕЖДУПРОЦЕСНИ КОМУНИКАЦИИ

В този раздел ще разгледаме проблемите при междупроцесни комуникации (Interprocess Communication или IPC) и някои механизми за решаването им. Проблемите при комуникации между процеси имат два аспекта. Първият е предаването на информация между процесите. Другият е свързан със съгласуване на действието на процесите, които работят асинхронно, така че да се гарантира правилното им взаимодействие.

Най-простият начин, по който два или повече процеса могат да взаимодействат е да се конкурират за достъп до общ ресурс. Например, два процеса P и Q четат и пишат в обща променлива брояч counter, като всеки увеличава променливата. Нека значението на counter е 7 и достъпа на двата процеса до нея се извърши в следния ред:

##### Процес P

1. Чете counter в локална променлива pa.

5.  $pa = pa + 1$

6. Записва pa в counter.

##### Процес Q

2. Чете counter в локална променлива pb.

3.  $pb = pb + 2$

4. Записва pb в counter.

При тази последователност на изпълнение на двата процеса резултатът в counter ще е неправилен - 8, а не 10, тъй като изменението на процеса Q ще се загуби. Такава ситуация, при която два или повече процеса четат и пишат в обща памет и крайният резултат зависи от реда, в който работят процесите се нарича **състезание (race condition)**. Как да се избегне състезанието? Решението на този проблем се нарича **взаимно изключване (mutual exclusion)**, т.е. по такъв начин да се организира работата на двата (или повече) процеса, че когато един от тях осъществява достъп до общия ресурс (изпълнява трите стъпки в примера) за другия (другите) да се изключи възможността да прави същото.

Друг по-сложен начин на взаимодействие на два или повече процеса е когато те извършват обща работа. Съществуват няколко класически модела на взаимодействие на процеси, извършващи обща работа, например:

- Производител-Потребител (The Producer-Consumer Problem)
- Читатели-Писатели (The Readers and Writers Problem)
- Задачата за обядващите философи (The Dining Philosophers Problem)

Това, което е необходимо за коректното взаимодействие на процесите (освен евентуално взаимно изключване), се нарича **синхронизация (synchronization)**. Например в задачата Производител-Потребител, синхронизацията изисква всяка произведена от Производителя данна да се предаде на Потребителя и обработи точно веднаж, като нищо не се загуби.

##### 4.1. ВЗАИМНО ИЗКЛЮЧВАНЕ

Проблемът за избягване на състезанието е бил формулиран от Е. Дейкстра (E.Dijkstra) чрез термина **критичен участък (critical section)**. Част от кода на процеса реализира вътрешни изчисления, които не могат да доведат да състезание. В друга част от кода си процесът осъществява достъп до обща памет или върши неща, които могат да доведат до състезание. 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 изпълнени 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.

Ще разгледаме решения на задачата за взаимното изключване чрез използване само на обща памет. Холандският математик Т. Декер (T.Dekker) пръв предложи решение на тази задача, а по-късно бе предложено още едно решение от Питерсон (G.Peterson). Ще разгледаме двата алгоритъма в най-простите им варианти за два процеса. Но преди това ще започнем с едно не съвсем коректно решение.

#### Алгоритъм с редуване на процесите

```
#define TRUE 1
shared int turn=0;
```

<pre>P0() { while (TRUE) {     while (turn != 0);     critical_section0();     turn = 1;     noncritical_section0(); } }</pre>	<pre>P1() { while (TRUE) {     while (turn != 1);     critical_section1();     turn = 0;     noncritical_section1(); } }</pre>
--	--

Този алгоритъм използва една обща променлива `turn`, чието значение е номер на процес, който е на ред да влезе в критичния си участък. В този алгоритъм и останалите в раздела общата памет ще записваме като деклариране на променлива с думата `shared`. Двата процеса строго се редуват. Недостатък в това решение е нарушение на изискване 3. Ако един от процесите е по-бавен това ще пречи на другия процес да влиза по-често в критичния си участък, или ако един от процесите завърши другият повече няма въобще да може да влиза в критичния си участък.

#### Алгоритъм на Декер

```
#define FALSE 0
#define TRUE 1
shared int wants0=FALSE, wants1=FALSE;
shared int turn=0;
```

<pre>P0() { while (TRUE) {     wants0 = TRUE;     while (wants1)         if (turn == 1) {             wants0 = FALSE;             while (turn == 1);             wants0 = TRUE;         }     critical_section0();     turn = 1;     wants0 = FALSE;     noncritical_section0(); } }</pre>	<pre>P1() { while (TRUE) {     wants1 = TRUE;     while (wants0)         if (turn == 0) {             wants1 = FALSE;             while (turn == 0);             wants1 = TRUE;         }     critical_section1();     turn = 0;     wants1 = FALSE;     noncritical_section1(); } }</pre>
--	--

Този алгоритъм използва три общи променливи за осигуряване на взаимно изключване. Променливите `wants0` и `wants1` са флагове за всеки от процесите. Флаг `FALSE` означава, че съответният процес не е в критичния си участък и не иска вход. Когато процес иска вход в критичния си участък, той вдига флага си (`TRUE`). И този алгоритъм използва променлива `turn`, чието значение е номер на процес, който е на ред да влезе в критичния си участък, но двата процеса не се редуват строго. Променливата се използва когато и двата процеса желаят вход в критичните си участъци, за да разреши конфликта.

### Алгоритъм на Питерсон

```
#define FALSE 0
#define TRUE 1
shared int turn=0;
shared int interested[2] = {FALSE, FALSE};

enter_region(int process)
{
    int other;
    other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while (turn == process && interested[other] == TRUE);
}

leave_region(int process)
{
    interested[process] = FALSE;
}
```

<pre>P0() {     while (TRUE) {         enter_region(0);         critical_section0();         leave_region(0);         noncritical_section0();     } }</pre>	<pre>P1() {     while (TRUE) {         enter_region(1);         critical_section1();         leave_region(1);         noncritical_section1();     } }</pre>
---	---

Алгоритъма на Питерсон също използва три общи променливи за осигуряване на взаимно изключване на два процеса. Променливите `interested[2]` също са флагове на процесите. И тук се използва променлива `turn`, чието значение е номер на процес, който е на ред. Начинът, по който променливата `turn` се изменя и проверява, тук е по-различен.

Основният недостатък и на двата алгоритъма е, че за осигуряване на взаимното изключване се използва активно чакане (*busy waiting*). Всеки от процесите, който желае да влезе в критичния си участък изпълнява цикъл, в който непрекъснато проверява дали това е възможно, докато стане възможно. Много по-естествено и ефективно би било, когато процес не може да влезе критичния си участък да бъде блокиран и когато влизането стане възможно операционната система да го събуди. Освен това и двата алгоритъма реализират взаимно изключване на два процеса. Алгоритъм за взаимно изключване на  $n$  процеса, известен като *Bakery algorithm*, е бил предложен от Лампорт (L.Lamport), но е по-сложен.

#### 4.2. СЕМАФОРИ

През 1965г. Дейкстра предложи нов механизъм за междупроцесни комуникации и го нарече **семафори** (*semaphores*). С всеки семафор се свързва цяла променлива - брояч и списък на чакащи в състояние блокиран процеси. Значението на брояча е неотрицателно число. За семафорите Дейкстра определи три операции - инициализация, операция *P* и *V*. При инициализацията се създава нов обект семафор и се зарежда начално значение в брояча му, което е неотрицателно цяло число. Тази операция ще записваме като деклариране и инициализация на променлива, например:

```
semaphore s = 1;
```

Операцията *P* проверява и намалява значението на брояча на семафора, ако това е възможно, в противен случай блокира процеса и го добавя в списъка на чакащи процеси, свързан със съответния семафор. Събитието, което блокирания процес чака, е увеличение на брояча на семафора. Това събитие ще настъпи когато друг процес изпълни операцията *V* над същия

семафор. Операцията V събужда един блокиран по семафора процес, ако има такива, а в противен случай увеличава брояча на семафора т.е. запомня едно събуждане. За операциите P и V се използва и обозначението down и up. Алгоритмите на двете операции са следните:

```
P(s)
{
    if ( s > 0 )    s = s - 1;
    else    блокира процеса, изпълняващ P по семафора s;
}

V(s)
{
    if (има блокирани по семафора s процеси) събужда един процес;
    else    s = s + 1;
}
```

Същественото за двете операции е, че са неделими (атомарни), т.е. никой процес, изпълняващ V(s) не може да бъде блокиран и докато един процес изпълнява операция над семафор s друг процес не може да започне операция над s докато първата не завърши. Тази неделимост на операциите може да бъде осигурена при реализацията им в ядрото като системни примитиви. Модулите, реализиращи операциите, са част от ядрото на операционната система и там за осигуряване на взаимно изключване може да се използва забрана на прекъсванията или други апаратни средства за взаимно изключване.

### Взаимно изключване на n процеса чрез семафор

За осигуряване на взаимното изключване на произволен брой процеси е необходим един семафор, инициализиран с 1. Освен това всеки от процесите трябва да загради критичния си участък с операциите P и V над този семафор. Такъв семафор се нарича двоичен, тъй като значенията, които заема брояча му са само две - 0 и 1.

```
#define TRUE 1
semaphore mutex=1;
```

<pre>P0() { while (TRUE) {     P(mutex);     critical_section0();     V(mutex);     noncritical_section1(); } }</pre>	<pre>P1() { while (TRUE) {     P(mutex);     critical_section1();     V(mutex);     noncritical_section1(); } }</pre>	...
---	---	-----

### Синхронизация чрез семафори

Съществуват няколко класически задачи за междупроцесни комуникации и качествата на всеки нов механизъм се демонстрират чрез решаването на тези задачи. Сега ще разгледаме решаването на някои от тези задачи чрез механизмите обща памет и семафори, но преди това един по-прост пример. Имаме два процеса P1 и P2, които работят асинхронно и искаме операторите S1 в процеса P1 да се изпълнят преди S2 в P2. Решението е следното:



```
semaphore s=0;
```

<pre>P0 () {     S1;     V(s); }</pre>	<pre>P1 () {     P(s);     S2; }</pre>
--	--

### Производител - Потребител

Задачата Производител-Потребител е модел на един начин за взаимодействие на два процеса, познат ни от командните езици в UNIX и LINUX. Когато командният интерпретатор изпълнява конвейер, например:

```
who | wc -l
```

той решава задачата Производител-Потребител. Има два процеса, които взаимодействат, като извършват обща работа. Процесът-производител (who) произвежда данни, които се предават на процеса-потребител (wc), който ги използва. Командните интерпретатори в UNIX и LINUX решават тази задача чрез програмен канал.

Тук ще използваме обща памет за предаване на данните от производителя към потребителя. Предполагаме, че двата процеса използват общ буфер, който може да поеме N елемента данни (нека за простота елементите са цели числа). Организацията на буфера няма отношение към проблема за синхронизация, затова няма да я конкретизираме. Производителът записва в буфера всеки произведен от него елемент, а потребителят чете от буфера елементите, за да ги обработи. Двата процеса работят едновременно, с различни и неизвестни относителни скорости. Следователно, задачата за синхронизация се състои в това да не се позволи на производителя да пише в пълен буфер, да не се позволи на потребителя да чете от празен буфер и всеки произведен елемент да бъде обработен точно един път. Освен това трябва да се осигури и взаимно изключване при достъп до общия буфер. Решението на Дейкстра използва три семафора: mutex за взаимното изключване, empty и full за синхронизацията. Броячът на empty съдържа броя на свободните места в буфера и по него производителът ще се блокира когато няма място в буфера. Семафорът full ще брой запълнените места в буфера и по него потребителят ще се блокира когато в буфера няма данни.

```
#define N 100
```

```
#define TRUE 1
```

```
shared buffer buf; /* общ буфер с капацитет N елемента */
semaphore mutex = 1; /* за взаимното изключване */
semaphore empty = N; /* брой свободните елементи в буфера */
semaphore full = 0; /* брой запълнените елементи в буфера */
```

<pre>producer() {     int item;     while (TRUE) {         produce_item(&amp;item);         P(empty);         P(mutex);         insert_item(&amp;item);         V(mutex);         V(full);     } }</pre>	<pre>consumer() {     int item;     while (TRUE) {         P(full);         P(mutex);         remove_item(&amp;item);         V(mutex);         V(empty);         consume_item(item);     } }</pre>
--	---

### Читатели - Писатели

Тази задача е модел на достъп до база данни от много конкурентни процеси, които се делят на два вида. Единият вид са процеси-читатели, които само четат данните в базата, а другият вид са процеси-писатели, които изменят по някакъв начин данните в базата. Искаме във всеки момент достъп до базата данни да могат да осъществяват или много процеси-

читатели или един процес-писател. Тази задачата за синхронизация е решена от Куртоа, Хейманс и Парнас чрез една обща променлива брояч и два двоични семафора.

```
#define TRUE 1
shared int rcount=0; /* брой процеси-читатели, които четат */
semaphore mutex=1; /* управлява достъпа до rcount */
semaphore db=1; /* управлява достъпа до базата данни */

reader()
{
while(TRUE) {
    P(mutex);
    rcount = rcount + 1;
    if (rcount == 1) P(db);
    V(mutex);
    read_data_base();
    P(mutex);
    rcount = rcount - 1;
    if (rcount == 0) V(db);
    V(mutex);
    use_data_read();
}
}

writer()
{
while (TRUE) {
    collect_data();
    P(db);
    write_data_base();
    V(db);
}
}
```

Ако никой от процесите не осъществява достъп до базата данни, то двата семафора са 1 и `rcount` е 0, тогава първият процес, който се появи ще изпълни `P(db)` и ще затвори този семафор (броячът му ще стане 0). Ако това е процес-читател, то `rcount` ще стане 1 и следващите читатели няма да проверяват семафора `db`, а само ще увеличават `rcount` и ще започват да четат базата данни. Когато читател завърши четенето той намалява `rcount`, а последният читател изпълнява `V(db)` и ще отвори семафора `db` (ще събуди един писател, блокиран по `db`, или броячът на `db` ще стане 1).

Ако първият процес, получил достъп до базата данни е писател, то първият читател ще се блокира по семафора `db`, следващите читатели ще се блокират по `mutex`, а следващите писатели ще се блокират по `db`. Когато писателят завърши писането той ще изпълни `V(db)` и с това ще събуди един процес, чакащ за достъп до базата данни. Ако това е първият чакащ читател, той ще изпълни `V(mutex)` и ще събуди следващия чакащ читател, който ще събуди следващия и т.н. докато всички читатели бъдат събудени.

Недостатък на това решение е, че то дава преимущество на читателите, което в една реална база данни не е добра стратегия.

### Задачата за обядващите философи

И тази задача е поставена и решена за първи път от Дейкстра чрез семафори и обща памет. Задачата се състои в следното. Пет философа седят около кръгла маса, като пред всеки от тях има чиния със спагети, а между всеки две чинии има само по една вилица. Животът на всеки философ представлява цикъл, в който той размишлява, в резултат на което огладнява и се опитва да вземе двете вилици около своята чиния. Ако успее известно време се храни, а след това връща вилиците. Задачата е да се напише програма, която да прави това, което се очаква от философа.

Едно очевидно, но грешно решение е следното.

```
#define N 5          /* броя на философите */
#define TRUE 1

philosopher(int i)   /* i е номер на философа, от 0 до 4 */
{
while(TRUE) {
    think();
    take_fork(i);     /* взима лявата вилица, като чака докато тя
                       стане достъпна */
    take_fork((i+1)%N); /*взема дясната вилица, като също чака*/
    eat();
    put_fork(i);      /* връща лявата вилица */
    put_fork((i+1)%N); /* връща дясната вилица */
}
}
```

Всеки философ изпълнява функцията `philosopher`. Грешката в това решение е, че може да доведе до дедлок. Ако всичките пет философа вземат едновременно левите си вилици, то никой няма да може да вземе дясна вилица и всички ще останат вечно блокирани.

Теоретически правилно и несложно решение, може лесно да се получи от горното като петте оператора след `think()` се направят критичен участък. Но от практическа гледна точка това решение не е добро, защото с наличните пет вилици във всеки момент биха могли да се хранят едновременно два философа, а при това решение във всеки момент се храни най-много един философ, а останалите гладни чакат.

Решението на Дейкстра използва общ масив `state[N]`, като всеки елемент описва състоянието на съответния философ. Възможните състояния са: мисли, гладен е (опитва се да вземе двете вилици) и храни се. Достъпа до общия масив се регулира от двоичен семафор `mutex`. Освен него се използва масив от семафори `s[N]`, по един за всеки философ, по който той се блокира когато трябва да чака освобождаването на вилиците.

```
#define N 5
#define TRUE 1
#define LEFT (i-1)%N /* номер на левия съсед на философ i */
#define RIGHT (i+1)%N /* номер на десния съсед на философ i */
#define THINKING 0 /* философът мисли */
#define HUNGRY 1 /* философът се опитва да вземе вилици */
#define EATING 2 /* философът се храни */

shared int state[N] = {0,0,0,0,0};
semaphore mutex=1; /* за взаимно изключване на state[N] */
semaphore s[N]={0,0,0,0,0}; /* семафори, по един на философ */

philosopher(int i)
{
while(TRUE) {
    think();
    take_forks(i);
    eat();
    put_forks(i);
}
}

take_forks(int i)
{
    P(mutex);
    state[i] = HUNGRY;
    test(i);
    V(mutex);
    P(s[i]);
}
```

```

put_forks(int i)
{
    P(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    V(mutex);
}

test(int i)
{
    if (state[i]==HUNGRY&&state[LEFT]!=EATING&&state[RIGHT]!=EATING) {
        state[i] = EATING;
        V(s[i]);
    }
}

```

При използването на обща памет и семафори за междупроцесни комуникации отговорността за правилното взаимодействие на процесите е на програмиста. Поради това този метод не е безопасен. Грешки в програмите могат да доведат до дедлок, състезание и други форми на непредсказуемо и невъзпроизводимо поведение. Съществува друг по-безопасен метод за комуникация, при който процесите обменят съобщения.

### 4.3. СЪОБЩЕНИЯ

За да комуникират два процеса чрез механизъм на съобщения между тях трябва да се установи комуникационна връзка (communication link). След това процесите обменят съобщения чрез два примитива:

```

send(destination, message)

receive(source, message)

```

Чрез `send` процес изпраща съобщение `message` към процес `destination`. Процес, който иска да получи съобщение, трябва да изпълни `receive`, при което съобщението се записва в `message`. Следователно, за да се осъществи предаване на едно съобщение между два процеса, е необходимо единият процес да изпълни `send`, а другият `receive`. При реализацията на механизъм на съобщенията е необходимо да се дадат отговори на следните въпроси, определящи логическите свойства на комуникационната връзка.

1. Как се установява комуникационна връзка между процесите?
2. Може ли една комуникационна връзка да свързва повече от два процеса?
3. Колко комуникационни връзки може да има между два процеса?
4. Еднопосочна или двупосочна е комуникационната връзка?
5. Какво е капацитет на комуникационната връзка?
6. Има ли някакви изисквания за размер и/или структура на съобщенията предавани по определена комуникационна връзка?

В операционните системи има различни реализации на механизъм на съобщения.

#### 4.3.1 АДРЕСИРАНЕ НА СЪОБЩЕНИЯТА

Ще разгледаме първите четири от поставените въпроси, отговорът на които зависи от това, как се адресира получателя и изпращача в примитивите `send` и `receive`.

##### Директна комуникация

Всеки процес именува явно процеса-получател или процеса-изпращач, т.е. `destination` и `source` са идентификатори/имена на процеси. За да се предаде съобщение от процес `P` към процес `Q`, трябва да се изпълнят примитивите.

<b>Процес P</b>	<b>Процес Q</b>
<code>send(Q, message);</code>	<code>receive(P, message);</code>

При този начин на адресация отговорите на първите четири въпроса са следните.

1. Комуникационната връзка се установява автоматично между двойката процеси, но всеки от тях трябва да знае идентификатора на другия.
2. Комуникационната връзка свързва точно два процеса.
3. Между всяка двойка процеси може да има само една комуникационна връзка.
4. Комуникационната връзка е двупосочна.

При този начин съществува симетрия при адресацията, всеки от процесите трябва да укаже идентификатора на другия процес. Може да се реализира асиметричен вариант на директна комуникация. В примитива `receive` се указва идентификатор на процес, както по-горе или може да е от вида:

```
receive (ANY, message);
```

Това означава, че процесът иска да получи съобщение от всеки, който му изпрати такова. Функцията ще върне идентификатора на процеса, от който е полученото съобщение.

### Косвена комуникация

Въвежда се нов обект, наричан пощенска кутия (`mailbox`), опашка на съобщенията (`message queue`) или порт (`port`). Съобщенията се изпращат в или се получават от пощенска кутия. Следователно, `destination` и `source` са идентификатори на пощенска кутия. За да се предаде съобщение от процес `P` към процес `Q`, трябва да се използва обща пощенска кутия, например с име `A` и да се изпълнят примитивите.

#### Процес P

```
send(A, message);
```

#### Процес Q

```
receive(A, message);
```

При този метод на адресация отговорите на въпросите са следните:

1. Комуникационната връзка се установява между процесите когато те използват обща пощенска кутия.
2. Комуникационната връзка може да свързва и повече от два процеса.
3. Между два процеса може да съществува повече от една комуникационна връзка.
4. Комуникационната връзка може да е еднопосочна или двупосочна.

В този случай освен примитивите `send` и `receive` трябва да има и примитив за създаване на пощенска кутия. Също така възниква и въпросът за собствеността на пощенската кутия и за правата на процесите да изпращат съобщения в или да получават съобщения от определена пощенска кутия. Друг въпрос е как се унищожава пощенска кутия.

Една възможна реализация е собственик на пощенската кутия да става процесът, който я създаде. Всеки друг процес, който знае името на пощенската кутия и на който собственикът е дал права, може да я използва. Унищожаването на пощенска кутия може да се реализира чрез системен примитив, извикван явно от собственика ѝ. Друга възможност е процесите да могат да ползват обща пощенска кутия чрез механизма за създаване на процеси и когато последният процес, ползващ определена пощенска кутия, завърши тя да се унищожава автоматично от системата.

### 4.3.2. БУФЕРИРАНЕ НА СЪОБЩЕНИЯТА

Ще разгледаме и последните два от поставените въпроси, отнасящи се до логическите свойства на комуникационната връзка. Какъв е капацитета на комуникационната връзка? Той определя броя на временно съхраняваните в комуникационната връзка изпратени, но още неполучени съобщения.

#### Съобщения без буфериране

Комуникационната връзка има капацитет нула, т.е. не може да има временно чакащи съобщения. Това означава, че ако първо се изпълни `send`, то изпращачът ще трябва да изчака докато получателят изпълни `receive`. Процесите изпращач и получател синхронизират работата си в момента на предаване на съобщението, затова този метод се нарича още "рандеву".

## Съобщения с автоматично буфериране

Комуникационната връзка има ограничен капацитет. Определен брой (обем) съобщения могат временно да се съхраняват в комуникационната връзка, докато получателят изпълни `receive`. Когато се изпълнява `send`, ако комуникационната връзка не е пълна, съобщението се съхранява в нея и изпращачът продължава работата си. Ако комуникационната връзка е пълна, изпращачът ще трябва да чака, докато се освободи място в нея. И в двата случая изпращачът не може да знае дали съобщението му е получено след като `send` завърши. Ако това е важно за комуникацията, то процесите трябва явно да прилагат протокол за потвърждаване. Следният фрагмент илюстрира предаването на едно съобщение от процес P към процес Q с потвърждаване.

### Процес P (изпращач)

```
send(Q, message);  
receive(Q, message);
```

### Процес Q (получател)

```
receive(P, message);  
send(P, "acknowledgement");
```

В MINIX механизъм на съобщенията е реализиран с директна адресация, без буфериране и съобщенията са с фиксирана дължина.

В IPC пакета на UNIX System V, който се поддържа и от други UNIX и LINUX системи, са включени съобщения. Там се използва косвена адресация и автоматично буфериране.

Обект, в който се изпращат и от който се получават съобщения, се нарича опашка на съобщенията (`message queue`). Всяка опашка на съобщенията има външно име, което е цяло положително число и се нарича ключ. Това позволява една опашка на съобщенията да се използва за комуникация между неродствени процеси. Опашка на съобщенията се създава явно чрез системен примитив `msgget`. Чрез същия системен примитив процес може да получи достъп до създадена вече опашка. Процесът, създал една опашка на съобщенията, става неин собственик и има право да определя правата на другите процеси за достъп до нея. При определяне правата за достъп до опашка на съобщенията се използва същия метод както при файловете, а именно код на защита. Правото `r` означава правото да се получават съобщения от опашката, а правото `w` - да се изпращат съобщения в опашката. Една опашка на съобщенията съществува докато не се унищожи явно чрез системен примитив `msgctl` и само собственикът има право за това. Чрез този системен примитив се извършват и други операции по управление на опашка, например промяна на правата на достъп. Изпращането и получаването на съобщение се извършва с примитивите `msgsnd` и `msgrcv`.

По отношение на структурата на съобщенията има следните изисквания. Всяко съобщение се състои от тип-цяло положително число и текст-масив от байтове с променлива дължина. Структурата на съобщение е следната:

```
struct msgbuf {  
    long mtype;  
    char mtext[N]; }
```

Чрез типа на съобщението може да се реализират няколко потока за предаване на съобщения в рамките на една опашка на съобщенията, включително и двупосочна комуникация. За това съществена роля има и примитива `msgrcv`, който позволява да се поиска първото съобщение от определен тип.

В UNIX, LINUX и MINIX системите се реализират два традиционни механизма за комуникации между процеси - сигнали и програмни канали. Програмният канал осигурява еднопосочно предаване на неформатиран поток от данни (поток от байтове) между процеси и синхронизация на работата им. Реализират се два типа програмни канали:

- **неименован програмен канал (`unnamed pipe` или `pipe`)** - за комуникация между родствени процеси
- **именован програмен канал (`named pipe` или **FIFO файл**)** - за комуникация между независими процеси.

Като механизъм за комуникация те са еднакви. Реализират се като тип файл, който се различава от обикновените файлове и има следните особености:

- За четене и писане в него се използват системните примитиви `read` и `write`, но дисциплината е FIFO.

- Каналът има доста ограничен капацитет.

Двата типа програмни канала се различават по начина, по който се създават и унищожават и по начина, по който процес първоначално осъществява достъп към канала. Програмният канал може да се разглежда като механизъм на съобщения с косвена адресация и автоматично буфериране. Разликата е, че в програмния канал няма граници между съобщенията, т.е. процес P може да запише едно съобщение от 1000 байта, а процес Q да го прочете като 10 съобщения от по 100 байта или обратното.

**Пример.** Програмата илюстрира механизма на съобщенията в UNIX и LINUX, като реализира задачата Производител - Потребител.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <signal.h>
#define KEY 24
struct msgb{
    long mtype;
    char mtext[256]; };
int mid;

void cleanup() {
    if ( msgctl(mid, IPC_RMID, 0) == -1 ) {
        perror( "Msgctl failed. " );
        exit(1); }
    printf("MQ destroyed\n");
    exit(0);
}

main() {
    struct msgb msg;
    int i, j, buf;
    if ( ( mid=msgget(KEY, IPC_CREAT|0666) ) == -1 ) {
        perror("Msgget failed. ");
        exit(1); }
    signal(SIGINT, cleanup);
    if ( fork() ) { /* parent - consumer */
        for ( i=0; ; i++ ) {
            msgrcv(mid, &msg, 256,1,0);
            buf = *((int*)msg.mtext);
            buf *= 2;
            printf("Consumer: %d\n", buf); }
    } else { /* child - producer */
        signal(SIGINT, SIG_DFL);
        for (i=0; ; i++) {
            for ( j=0; j<100000; j++ );
            msg.mtype = 1;
            *(int*)msg.mtext = i;
            msgsnd(mid, &msg, sizeof(int), 0); }
    }
}
```

При използване на съобщения и двата аспекта на комуникацията - предаването на данните и синхронизацията се решават от механизма. Затова съобщенията са по-безопасни. Но двата разгледани метода за междупроцесни комуникации:

- чрез обща памет и семафори
- чрез съобщения

не са взаимно изключващи се, т.е. могат да са реализирани и използват в една операционна система. IPC пакета на UNIX System V включва и трите механизма.

## 5. НИШКИ

Нишките (*threads*) са сравнително нова абстракция в операционните системи. Понякога ги наричат *lightweight processes* или *нодпроцеси* и макар, че това наименование представлява известно опростяване, то е добра начална точка. Нишките не са процеси, но те имат нещо общо с процесите, представляват част от процес. Да си припомним какво е процес, така както го разглеждахме до сега. Той включва програмен код, данни, различни други ресурси, като файлови дескриптори, текущ каталог, управляващ терминал и др. и досега един набор от машинни регистри, т.е. един регистър РС. Това означава, че всеки процес има една последователност на изпълнение на команди или накратко е последователен процес.

Разгледаният модел на процесите се базира на две независими концепции:

- групиране на ресурсите - процесът има различни ресурси;
- изпълнение на програма - процесът е последователност на изпълнение на команди (thread of execution или накратко само thread).

Тези два аспекта на процеса могат да бъдат разделени и така се появява понятието нишка. Процесът се използва за групиране на ресурсите, а нишките са обектите, изпълнявани от ЦП. Всяка последователност на изпълнение в рамките на процес се нарича нишка, т.е. тя е част от процес, която има собствен набор от регистри и стек.

Идеята на въвеждането на понятието нишка е процесът да може да има няколко последователности на управление, т.е. да е конкурентен, а не последователен процес. Такъв процес, който включва няколко нишки се нарича *multithreaded process*. Различните нишки в един процес не са така независими, както различните процеси. Следва списък на елементите на нишка и тези общи за всички нишки в един процес.

Елементи на нишка	Елементи на процес
РС и други регистри стек състояние	адресно пространство глобални променливи отворени файлове текущ каталог процеси синове

Всяка нишка има свой собствен стек, който съдържа по един слой за всяка извикана в нишката функция, която още не е завършила. Една нишка може да се намира в едно от няколко състояния: текуща, готова, блокирана, завършила (зомби). Преходите от едно състояние в друго са както в разгледания модел на процесите.

Поддържането на нишки дава възможност да се програмират конкурентни приложения, които могат да работят по-ефективно като в еднопроцесорна така и в многопроцесорна среда. Една причина за появата на нишки са приложения, които изпълняват няколко дейности. Проектът на програмата ще е по-ясен ако приложението се декомпозира на няколко последователни нишки, които работят в едно адресно пространство. Като пример за такова приложение, където нишките са полезни, да разгледаме текстообработваща програма (Word processor). Обикновено тя визуализира документа на екрана форматиран така както ще изглежда на печат. Освен това периодично и автоматично тя записва редактирания файл на диска. Нека проектираме нашия Word с три нишки. Една нишка ще взаимодейства с потребителя (интерактивна нишка) и ще прима неговите команди за изтриване, добавяне на текст и т.н. Друга нишка ще преформатира документа при необходимост, т.е. по команда от интерактивната нишка. Третата нишка периодически ще записва документа на диска. Такъв модел на програма Word е по-ясен и по-прост за реализация. Освен това в този случай модел с три процеса, комуникиращи с някакъв механизъм, не е подходящ.

Друг аргумент в полза на нишките е производителността. Операциите с нишки - създаване и унищожаване, ще са по-бързи отколкото при процеси, защото нишката няма собствени ресурси. Също така по-ефективно може да е изпълнението на един многонишков процес. Когато някоя от нишките на процеса се блокира, чакайки например входно-изходна операция, друга нишка на процеса може да продължи изпълнението. Така ще се увеличи общата скорост на работа на процеса.

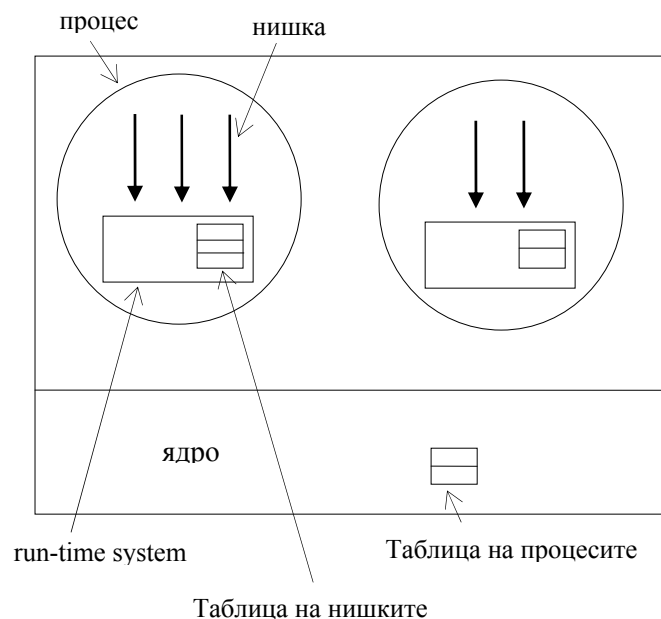


## 5.1. РЕАЛИЗАЦИЯ НА НИШКИ

Пакетът, реализиращ абстракцията нишка, трябва да осигури набор от операции, подобни на тези при процеси: създаване на нова нишка, завършване на нишка, изчакване на завършването на друга нишка (аналози на `fork`, `exit`, `wait`) и др. Има два основни начина за реализация на нишки:

- в потребителското пространство
- в ядрото.

Първият начин означава, че ядрото не знае нищо за нишки и управлява последователни процеси. Общата схема при тази реализация е показана на Фиг.5.

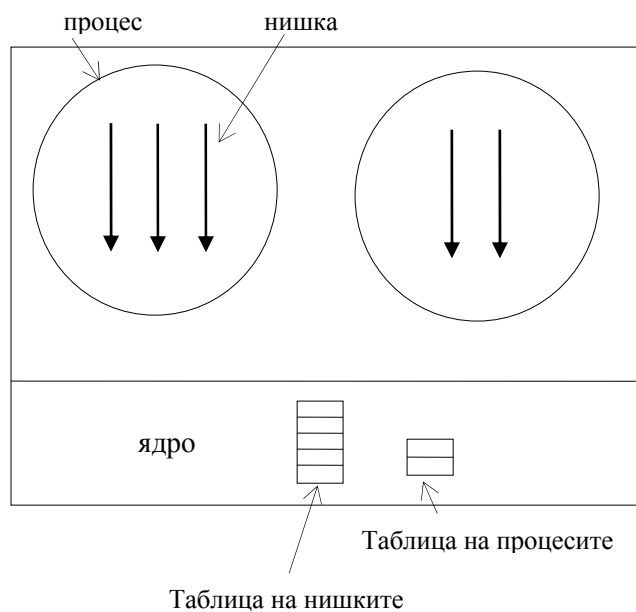


Фиг. 5. Нишки в потребителското пространство

Целият пакет функции, реализиращи нишките (`run-time system`), работи в потребителското пространство. Всеки процес има собствена Таблица на нишките, която съдържа информация за нишките на процеса, като състояние, съхранени регистри и др. Тази таблица също се намира в потребителското пространство и се управлява от `run-time` системата. Например, когато една нишка трябва да изчака завършването на друга нишка на процеса, тя извиква съответната функция от `run-time` системата. Там се променя състоянието на нишката в блокирана, съхраняват се регистрите на нишката, избира се друга нишка на процеса, зареждат се съхранените регистри на новата нишка и така се прави превключване на нишки (`thread switching`). Има едно съществено различие при управлението на нишки и процеси. Превключването на нишки не изисква прекъсване и вход в ядрото. Всички функции, реализиращи описаните действия, са в потребителското пространство. Това е едно от предимствата, което този начин за реализация на нишки, дава, а именно по-бързо превключване между нишки в един процес. Друго предимство е, че този метод може да се използва в съществуваща операционна система без изменение в ядрото, а с добавяне на нова библиотека функции.

Но съществуват и проблеми. Един от тях е свързан със системните примитиви, които блокират процеса, като `read`, `write` и др. Ако при изпълнението на такъв примитив се наложи, се блокира целият процес, а не само нишката, извикала примитива. Такова поведение противоречи на основната идея за използване на нишки в приложения, а именно когато една нишка в процеса се блокира друга негова нишка да продължи работа. Друг проблем се проявява при планиране на нишки. Когато една нишка започне работа тя ще работи докато доброволно не освободи ЦП. Това е така, защото планировчикът на нишките не може да работи докато управлението не се предаде в `run-time` системата. Следователно, планирането на нишките в един процес е без преразпределение.

Вторият начин за реализация на нишки е в пространството на ядрото, т.е. ядрото знае за съществуването им (Фиг.6). В този случай в потребителското пространство няма `run-time` система и Таблица на нишките. В ядрото има една глобална Таблица на нишките, съхраняваща информация за всички нишки в системата. Освен това, както и преди, в ядрото има Таблица на процесите, описваща процесите. Всички операции с нишки са реализирани като системни примитиви, т.е. в ядрото. Това означава, че когато нишка трябва да се блокира, ядрото може да избере друга нишка от същия или от друг процес. Следователно, системните примитиви като `read` не създават проблеми. Недостатък на този метод е, че операциите с нишки стават по-бавни (включват прекъсване, съхранение на контекста и възстановяването му).



Фиг.6. Нишки в пространство на ядрото

По-нататък ще разгледаме по-подробно нишките по стандарта POSIX 1003.1c, известни още като `pthread`s, които могат да се използват в съвременните версии на UNIX и LINUX системите. В някои версии, като UNIX System V, Solaris, LINUX и др., нишките се реализират в ядрото. В други UNIX системи, като BSD нишките не се поддържат от ядрото и реализацията им е по първия начин в една от библиотеките. Функциите за работа с нишки, които ще разгледаме, осигуряват:

- Основни операции с нишки, като създаване, завършване и др.
- Механизми за синхронизация на работата на конкурентните нишки в един процес.

## 5.2. ОСНОВНИ ОПЕРАЦИИ С НИШКИ

Първата - главна нишка във всеки процес се създава автоматично при създаване на процес. Друга нишка може да се създаде, когато една нишка изпълни функцията:

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,  
void *(start_routine)(void *), void *arg);
```

Всяка нишка има идентификатор, който ѝ се присвоява при създаването и е от тип `pthread_t`. При успех се връща идентификатора на новата нишка чрез аргумента `thread`.

Аргументът `attr` определя някои характеристики на създаваната нишка или както ги наричат атрибути. Ако е указано `NULL`, то атрибутите имат значения по премълчаване. Един от атрибутите определя типа на нишката: `joinable` или `detached`. Тип `joinable` означава, че друга нишка в процеса може да се синхронизира с момента на завършването ѝ, т.е. след завършване на нишката тя не изчезва веднага (подобно на състояние зомби при процеси). Тип `detached` означава, че при завършване на нишката веднага се освобождават всички ресурси, заемани от нея, т.е. тя изчезва в момента на завършване. По премълчаване нишката се създава `joinable`. Има и други атрибути, които определят дисциплината и параметри на планиране на нишки.

Когато нишката бъде създадена тя започва да изпълнява функцията *start\_routine*, на която се предава аргумент *arg*. За разлика от процесите, където бащата и синът продължават изпълнението си след *fork* от една и съща точка, тук не е така, защото нишките на един процес имат общи ресурси. При успех функцията връща 0, а при грешка връща различен от 0 код на грешка.

Така създадената нишка завършва когато:

- Изпълни *return* от *start\_routine*;
- Извика явно *pthread\_exit*;
- Друга нишка я прекрати чрез *pthread\_cancel*.

**`void pthread_exit(void *retval);`**

Аргументът *retval* е код на завършване, който нишката изработва. Той е предназначен за всяка друга нишка на процеса, която изпълни *pthread\_join*. Но ако нишката е от тип *detached*, то след *pthread\_exit* от нея не остава никаква следа, следователно и кода не се съхранява. Няма връщане от тази функция.

**`int pthread_join(pthread_t thread, void **value_ptr);`**

Чрез тази функция една нишка може да синхронизира изпълнението си със завършването на друга нишка (аналог на *wait* при процеси, но тук няма изискването текущата нишка да е баща на чаканата). Аргументът *thread* е идентификатор на нишката, чието завършване се чака от текущата нишка. Текущата нишка се блокира докато не завърши нишка *thread*. Най-много една нишка може да изчака завършването на коя да е друга нишка, т.е. ако няколко нишки изпълнят *pthread\_join* за една и съща нишка, вторият *pthread\_join* ще върне грешка. При успех функцията връща 0 и чрез аргумента *value\_ptr* се предава кода на завършване на нишката *thread*. При грешка връща код на грешка различен от 0.

**Пример.** Програмата “Hello World” чрез нишки. Програмата извежда “Hello World” или “World Hello”, защото всяка от двете думи се извежда от отделна нишка.

```
#include <pthread.h>
main()
{
    int ret, status;
    pthread_t th1, th2;
    void *print_mess();
    char *mess1 = "Hello ";
    char *mess2 = "World\n";

    ret = pthread_create(&th1, NULL, print_mess, (void *)mess1);
    if ( ret != 0) { perror("Error in pthread_create 1 ");
                    exit(1); }
    ret = pthread_create(&th2, NULL, print_mess, (void *)mess2);
    if ( ret != 0) { perror("Error in pthread_create 2 ");
                    exit(1); }
    ret = pthread_join(th1, (void *)&status);
    printf("Thread 1 returned status %d\n", status);
    ret = pthread_join(th2, &status);
    printf("Thread 2 returned status %d\n", status);
}

void* print_mess(void *str)
{
    char *msg;
    msg = (char *)str;
    printf("%s", msg);
    pthread_exit((void *)0);
}
```

### 5.3. МЕХАНИЗЪМ `mutex`

`Mutex` е механизъм за блокиране и деблокиране на нишки, чрез който може да се реализира взаимно изключване на нишки при достъп до общи променливи. Един обект `mutex` има две състояния: `unlocked` – свободен и не принадлежи на никоя нишка и `locked` – заключен и принадлежи на нишката, която го е заключила. Обект `mutex`, ако е в състояние `locked`, може да принадлежи само на една нишка. Нов обект `mutex` се създава чрез функцията:

```
int pthread_mutex_init(pthread_mutex_t * mutex,  
pthread_mutexattr_t *mutexattr);
```

Идентификатор на новосъздадения обект `mutex` е променлива от тип `pthread_mutex_t` и се връща чрез аргумента `mutex`. Първоначално `mutex` е в състояние `unlocked`. Вторият аргумент задава атрибутите на създавания `mutex`. В LINUX се реализира атрибут тип на `mutex`, който може да е: `fast`, `recursive` или `error checking`. Типът влияе на семантиката на последващите операции над обекта `mutex`. По премълчаване, ако аргументът е `NULL`, се създава `mutex` от тип `fast`. При успех функцията връща 0, а при грешка връща различен от 0 код на грешка.

`Mutex` е общ ресурс, т.е. ресурс на процеса, а не на нишката, която го създава. Това означава, че не се унищожава или освобождава при завършване на нишката, която го е създавала.

```
int pthread_mutex_lock(pthread_mutex_t * mutex);
```

С тази функция текущата нишка се опитва да заключи обекта `mutex`. Възможните случаи при изпълнението са:

Ако `mutex` е свободен, то състоянието му се сменя в заключен от текущата нишка и тя продължава изпълнението си.

Ако `mutex` е заключен от някоя друга нишка, то текущата нишка бива блокирана, докато се смени състоянието `mutex` в `unlocked` от нишката, която го притежава в момента.

Ако `mutex` е заключен от текущата нишка, действието зависи от типа на `mutex`: при тип `fast` текущата нишка се блокира (това може да доведе до дедлок), при тип `recursive` се увеличава брояч (броят се многократните заключвания на един обект `mutex` от една нишка) и функцията завършва, при тип `error checking` това се счита за грешка. При успех функцията връща 0, а при грешка връща различен от 0 код на грешка.

```
int pthread_mutex_unlock(pthread_mutex_t * mutex);
```

С тази функция текущата нишка се опитва да освободи обекта `mutex`. Възможните случаи при изпълнението са:

Ако текущата нишка е настоящият притежател на `mutex`, то се сменя състоянието му в `unlocked`. Ако има нишки, чакащи този `mutex` (блокирани в `pthread_mutex_lock`), то една от тях се събужда и ѝ се дава възможност да се опита отново да получи `mutex`. Но ако `mutex` е от тип `recursive` се намалява броячът и когато той стане 0, тогава се прави отключване и събуждане.

Ако `mutex` е в състояние `unlocked` или е `locked` но от друга нишка, то действието зависи от типа на `mutex`: при тип `error checking` това е грешка, при тип `fast` и `recursive` не прави проверки.

При успех функцията връща 0, а при грешка връща различен от 0 код на грешка.

```
int pthread_mutex_destroy(pthread_mutex_t * mutex);
```

Унищожава `mutex`, който трябва да е в състояние отключен иначе се счита за грешка. При успех функцията връща 0, а при грешка връща различен от 0 код на грешка.

**Пример.** Програмата илюстрира взаимно изключване чрез `mutex` при нишки.

```
#include <pthread.h>  
pthread_mutex_t mut;  
int sum;
```

```

main()
{
int ret, status, cnt1, cnt2;
pthread_t th1, th2;
void *race_func();
cnt1 = 1;
cnt2 = 2;
sum = 0;
    ret = pthread_mutex_init(&mut, NULL);
    if ( ret != 0 ) { perror("Error in pthread_mutex_init ");
                    exit(1); }
    ret = pthread_create(&th1, NULL, race_func, (void *)&cnt1);
    if ( ret != 0 ) { perror("Error in pthread_create 1 ");
                    exit(1); }
    ret = pthread_create(&th2, NULL, race_func, (void *)&cnt2);
    if ( ret != 0 ) { perror("Error in pthread_create 2 ");
                    exit(1); }
    ret = pthread_join(th1, (void *)&status);
    printf("Thread 1 returned status %d\n", status);

    ret = pthread_join(th2, &status);
    printf("Thread 2 returned status %d\n", status);
    printf("\nSUM = %d\n", sum);
}

void* race_func(void *ptr)
{
int cnt, i,j;
    cnt = *(int *)ptr;
    for (i=1, i <= 300, i++) {
        pthread_mutex_lock(&mut);
        sum = sum + cnt;
        printf("%d ", sum);
        pthread_mutex_unlock(&mut);
        for (j=0;j<1000000;j++); /* for some delay */
    }
    pthread_exit((void *)0);
}

```

## 6. ДЕДЛОК

Казваме, че множество от два или повече процеса са в дедлок (deadlock), ако всички те са в състояние блокиран и всеки чака настъпването на събитие, което може да бъде предизвикано само от друг процес в множеството. Тъй като всички процеси са блокирани, никой от тях не може да работи и да предизвика събитие, което да събуди някой друг процес от множеството. Следователно, ако системата не се намеси, всички процеси ще останат вечно блокирани. Събитието, което процесите чакат най-често е предоставяне на ресурс на системата.

Кои са участниците в проблема дедлок? Системата включва различни типове ресурси и състезаващи се за тях процеси. Всеки тип ресурс може да има няколко идентични екземпляра. Ресурси могат да са:

- апаратни устройства – печатащо устройство, лентово устройство и др.
- данни – запис в системна структура, файл или част от файл и др.

Ресурсите са обектите, които процесите искат да използват монополно. Последователността от действия при използване на ресурс от процес е:

1. заявка за ресурс (request)
2. използване на ресурс (use)
3. освобождаване на ресурса (release)

Ако ресурсът не е свободен при стъпка 1, то процесът бива блокиран и се събужда, когато друг процес изпълни стъпка 3. Начинът по който се изпълняват тези стъпки зависи от типа на ресурса, например може да е системен примитив или част от системен примитив.

### 6.1. НЕОБХОДИМИ УСЛОВИЯ ЗА ДЕДЛОК

Дедлок може да възникне ако в системата са изпълнени следните условия, формулирани от Кофман (E.Coffman) като необходими условия за дедлок.

#### 1. Взаимно изключване (Mutual exclusion)

В системата има поне един ресурс, който трябва да се използва монополно, т.е. или е свободен или е предоставен на точно един процес.

#### 2. Очакване на допълнителен ресурс (Hold and Wait)

Процесите могат да получават ресурси на части, като съществува поне един процес, който задържа получени ресурси и чака предоставяне на допълнителен ресурс.

#### 3. Непреразпределение (No preemption)

Ресурс, предоставен на процес, не може насилствено да му бъде отнет. Процесите доброволно освобождават ресурсите, след като приключат работата си с тях.

#### 4. Кръгово чакане (Circular wait)

Трябва да съществува множество от блокирани процеси  $\{p_1, p_2, \dots, p_k\}$ , такива че  $p_1$  чака ресурс държан от  $p_2$ ,  $p_2$  чака ресурс държан от  $p_3$  и т.н.  $p_k$  чака ресурс държан от  $p_1$ .

Последното условие предполага изпълнението на другите три, така че условията не са напълно независими, но е полезно да се разглеждат отделно.

### 6.2. ГРАФ НА РАЗПРЕДЕЛЕНИЕ НА РЕСУРСИТЕ

Холт (R.Holt) е предложил математически модел на дедлока. Състоянието дедлок в системата може формално да бъде описано чрез двуделен ориентиран граф  $G = \{V, E\}$ . Множеството на върховете  $V$  е обединение на две непресичащи се подмножества:

$\{p_1, p_2, \dots, p_n\}$  - множество на процесите в системата

$\{r_1, r_2, \dots, r_m\}$  - множество на типовете ресурси в системата.

Множеството на ребрата  $E$  включва два вида ребра:

$(p_i, r_j)$  – означава, че процес  $p_i$  иска ресурс  $r_j$  (request edge)

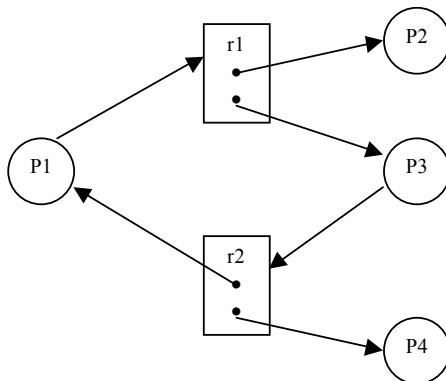
$(r_j, p_i)$  – означава, че един екземпляр от ресурс  $r_j$  е предоставен на процес  $p_i$  (assignment edge).

Когато процес  $p_i$  поиска ресурс  $r_j$ , в графа се добавя ребро  $(p_i, r_j)$ . Когато тази заявка бъде изпълнена, това ребро се трансформира в ребро  $(r_j, p_i)$ . Когато процесът освободи ресурса, реброто  $(r_j, p_i)$  се изключва от графа. Така графът представя разпределението на ресурсите и чакащите заявки във всеки един момент.

Може ли по графа на разпределение на ресурсите да се открие наличието на дедлок? Ако в графа няма цикъл, то в системата няма дедлок. Ако графът съдържа цикъл, то може да

има дедлок. Ако всеки тип ресурс, участващ в цикъла има един екземпляр, тогава има дедлок, в който участват процесите, включени в цикъла. Наличието на цикъл в графа е необходимо, а в горния случай и достатъчно условие за дедлок. В същност наличието на цикъл в графа е равносилно на четвъртото от условията на Кофман.

Следващият пример, показан на Фиг. 7, илюстрира горното твърдение, т.е. в графа има цикъл, включващ процесите  $p_1$  и  $p_3$  и ресурсите  $r_1$  и  $r_2$ , но няма дедлок.



Фиг. 7. Граф на разпределение на ресурсите

### 6.3. ПРЕДОТВРЯВАНЕ НА ДЕДЛОК

Методите за предотвратяване налагат ограничения на процесите при получаване на ресурси, така че дедлок става принципно невъзможен. Четирите необходими условия са ключ към някои възможни решения. Ако при разпределението на ресурсите в системата можем да осигурим неизпълнението на поне едно от тези условия, тогава дедлок няма да настъпи. Най-известните методи за предотвратяване са стратегиите на Хавендер (J.Havender).

#### 1. Взаимно изключване

Ако никой ресурс никога не се предоставя за монополно използване, то дедлок няма да настъпи. За съжаление обаче има ресурси, които не могат да не се използват монополно, например запис в таблицата на процесите, печатащо устройство, лентово устройство. Затова това условие не може да бъде нарушено за всички ресурси.

#### 2. Очакване на допълнителен ресурс

За да се наруши това условие трябва да се гарантира, че когато един процес иска ресурс той не задържа други ресурси.

Един възможен протокол на разпределение е следният. Всеки процес трябва да иска всичките необходими му ресурси наведнаж и преди започване на работа, и те да му бъдат предоставени преди началото на изпълнение.

Друг алтернативен протокол позволява на процес да иска ресурси и след започване на изпълнението, само когато не задържа никакви други ресурси. Процес може да иска ресурси, да ги използва, но преди да поиска допълнителни ресурси трябва да е освободил всички дадени му преди това.

Макар, че има разлика между двете стратегии, общият им недостатък е неефективното използване на ресурсите, тъй като процесите ще трябва да ги задържат по-дълго време отколкото са им необходими.

#### 3. Непреразпределение

Ако процес, който е получил и държи някакви ресурси, поиска допълнителни и системата не може да му ги предостави веднага, то процесът бива блокиран и всички дадени му до момента ресурси му се отнемат. Те се добавят към списъка на ресурсите, които процесът е поискал допълнително и които чака. Процесът ще бъде събуден и ще продължи изпълнението си, когато системата може да му даде всичките ресурси – новите (поисканите) и старите (отнетите му). Проблем при този метод е, че той може да се прилага, само по отношение на ресурси, чието състояние лесно може да се съхрани при насилствено отнемане от процес и след това да се възстанови (например, регистрите на ЦП, оперативна памет). Но как да се приложи към печатащо устройство или лентово устройство.

#### 4. Кръгово чакане

Въвежда се наредба на всички типове ресурси, като с всеки тип ресурс се свързва цяло число, което е поредния му номер. Нека  $R = \{ r_1, r_2, \dots, r_m \}$  е множеството на типовете ресурси. Следователно, определяме функция  $F: R \rightarrow N$ .

Един възможен протокол е следния. Всеки процес може да иска ресурси само по нарастване на номера на типа. Ако първоначално е получил ресурси от тип  $r_i$ , то след това може да иска само ресурси от тип  $r_j$ , където  $F(r_j) > F(r_i)$ . Ако от определен тип са му необходими няколко екземпляра, трябва да ги поиска наведнаж.

Друг възможен протокол е следния. Когато процес иска ресурс от тип  $r_j$ , той трябва да е освободил всички ресурси от тип  $r_i$ , за които  $F(r_i) \geq F(r_j)$ .

#### 6.4. ЗАОБИКАЛЯНЕ НА ДЕДЛОК

Дедлок може да се избегне и без да се поставят такива строги правила на процесите, а чрез внимателно анализиране на всяка заявка с цел да се прецени дали удовлетворяването ѝ е безопасно. Когато опасността от бъдещ дедлок се увеличи, ресурсът не се дава на процеса, за да се избегне дедлока. За тази цел системата трябва да разполага с информация, за това как процесите ще искат ресурси по време на своето изпълнение.

Най-популярният метод за заобикаляне на дедлок е известен като алгоритъм на банкера и е предложен от Дейкстра за 1 тип ресурс и от Хаберман (Habermann) за  $m$  типа ресурса. Всеки процес трябва да даде предварителна информация за максималния брой екземпляри от всеки тип ресурс, които може да поиска. Състоянието на разпределение на ресурсите се описва чрез:

- Брой свободни ресурси
- Максимален брой ресурси за всеки процес
- Брой ресурси дадени на всеки процес
- Брой ресурси, които процесът може още да поиска

Казваме, че системата се намира в *надеждно състояние* (*safe state*) на разпределение на ресурсите, ако съществува поне една последователна наредба на процесите  $\langle p_1, p_2, \dots, p_n \rangle$ , за която е изпълнено следното: нуждите на всеки процес  $p_i$  могат да се удовлетворят от свободните в момента ресурси и ресурсите държани от процесите  $p_j$ , където  $j < i$ . Следователно, от текущото разпределение на ресурсите съществува някаква последователност от други състояния, в която системата може да удовлетвори максималните потребности на всеки процес и той след време да завърши. Когато за текущото разпределение на ресурсите не съществува нито една такава последователност се казва, че състоянието е ненадеждно.

Нека системата разполага с 12 екземпляра от един тип ресурс и текущото разпределение е следното:

Процеси	Максимален брой	Получен брой	Оставащ брой
p1	10	5	5
p2	4	2	2
p3	9	2	7

Следователно в момента системата разполага с 3 свободни екземпляра. Това е пример за надеждно състояние, защото съществува последователността  $\langle p_2, p_1, p_3 \rangle$ , в която се гарантира завършването и на трите процеса.

Ако предположим, че процес  $p_3$  поиска една допълнителна бройка и системата му я даде, то новото състояние ще е ненадеждно, тъй като се гарантира завършването само на процес  $p_2$ .

Процеси	Максимален брой	Получен брой	Оставащ брой
p1	10	5	5
p2	4	2	2
p3	9	3	6

Алгоритъмът на банкера анализира всяка заявка за ресурс и я удовлетворява само ако новото състояние е надеждно. В противен случай процесът трябва да чака, т.е. блокира се и ще бъде събуден когато системата е в състояние да удовлетвори заявката му. Следва описание на Алгоритъма на банкера.



Структурите данни, представящи разпределението на ресурсите при  $n$  процеса и  $m$  типа ресурса, са следните масиви:

- Available[m] - брой свободни ресурси за всеки тип
- Max[n,m] - максимален брой ресурси за всеки процес. Max[i,j] е броят ресурси от тип  $r_j$ , който процес  $p_i$  може да поиска.
- Allocation[n,m] - разпределението в момента ресурси. Allocation[i,j] е броят ресурси от тип  $r_j$ , който процес  $p_i$  е получил и държи в момента.
- Need[n,m] - оставащите ресурси за всеки процес, т.е.  $Need[i,j] = Max[i,j] - Allocation[i,j]$

При описанието на алгоритъма ще използваме следните обозначения. Нека  $X[n]$  и  $Y[n]$  са едномерни масиви. Ще казваме, че  $X \leq Y$ , когато  $X[i] \leq Y[i]$  за  $i=1,2,\dots,n$ . Ще казваме, че  $X < Y$ , когато  $X \leq Y$  и  $X \neq Y$ . Всеки ред на матрицата Allocation ще разглеждаме като вектор, който описва разпределението на процеса  $p_i$  ресурси и ще означаваме  $Allocation_i$ . Аналогично, с  $Need_i$  ще означаваме нуждите на процеса  $p_i$ .

#### Алгоритъм на Банкера

Нека масивът Request[m] е една заявка на процес  $p_i$  за ресурси от различни типове.

1. if  $Need_i < Request$  then { “Грешка”; return; }
2. if Available < Request then { “ресурсите не са свободни”; блокира  $p_i$ ; return; }
3. Available = Available - Request;  
Allocation<sub>i</sub> = Allocation<sub>i</sub> + Request;  
Need<sub>i</sub> = Need<sub>i</sub> - Request;
4. if safe\_state() then return; else { възстановява се старото състояние; блокира  $p_i$ ; return; }

#### Алгоритъм на safe\_state

Нека Work[m] и Finish[n] са работни масиви, а flag е променлива.

1. Work = Available; flag = true; for ( $i = 1$ ;  $i \leq n$ ;  $i++$ ) Finish[i] = false;
2. while (flag == true) {  
    flag = false;  
    for ( $i = 1$ ;  $i \leq n$ ;  $i++$ ) {  
        if ( Finish[i] == false and  $Need_i \leq Work$  ) {  
            Work = Work + Allocation<sub>i</sub>;  
            Finish[i] = true;  
            flag = true; }  
    }  
}
3. for ( $i = 1$ ;  $i \leq n$ ;  $i++$ )  
    if ( Finish[i] == true ) then continue; else return false;  
return true;

Макар, че алгоритъмът на банкера е доста популярен (има го във всеки учебник по операционни системи), използването му в практиката е трудно. Броят на процесите не е фиксиран, а се изменя динамично, тъй като потребителите непрекъснато създават нови процеси. Възможна е и динамична промяна на броя на ресурсите от определен тип, например поради повреда на някое печатащо устройство. И накрая, сложността на алгоритъма при  $n$  процеса и  $m$  типа ресурса е от порядъка на  $m \cdot n^2$ . Това означава допълнително време, тъй като алгоритъмът трябва да се изпълнява при всяка заявка на процес за ресурс.

## 7. ПЛАНИРАНЕ НА ПРОЦЕСИ

В разгледания модел на многопроцесна операционна система в системата обикновено има много процеси в състояние готов и значително по-малко на брой ЦП. Тази част от ядрото, която избира най-подходящия процес за текущ и решава колко дълго ще работи се нарича **планировчик (scheduler)**. При наличие на много заявки за използване на определен ресурс, вземането на решение коя от тях да бъде удовлетворена се нарича **планиране**. Планирането е една от важните функции на операционните системи, тъй като обикновено заявките за използване на определен ресурс са повече от възможностите му. Тук ще разглеждаме планирането на ЦП, като го наричаме съкратено планиране.

### 7.1. НИВА НА ПЛАНИРАНЕ

В операционните системи често се извършва планиране на няколко нива.

- **Планиране на високо ниво** (планиране на заданията)

Това ниво присъства в пакетните ОС, където обикновено постъпват повече задания отколкото могат да бъдат изпълнени. При постъпване на ново задание то първо се поставя в опашка, съхранявана на диска. Планировчикът на това ниво решава кое от чакащите задания да бъде заредено в системата, т.е. да се създадат задачи (процеси) за него. Целта на този планировчик е получаването на добра смес от задания и увеличаване на пропускателната способност на системата.

- **Планиране на ниско ниво**

На това ниво се определя кой от готовите процеси да бъде избран за текущ и колко време да използва ЦП. Това е най-важното ниво, защото този планировчик работи най-често, (много пъти в секундата), винаги го има в ОС и е критичен за производителността на системата. Когато в ОС се използва думата планировчик, обикновено се разбира този.

- **Планиране на междинно ниво** (планиране на свопинга)

В някои ОС при управление на паметта се реализира метод, наречен свопинг. Планировчикът на това ниво управлява изхвърлянето на процеси на диска в свопинг областта и обратното им връщане в паметта.

Планиране на три нива се реализира в някои пакетните ОС. В интерактивните ОС обикновено планирането е на две нива - без високото ниво. Всеки нов процес се приема в системата, като стабилността се контролира от физически ограничения (брой терминали, таблица на процесите и др.) и от човешката природа.

### 7.2. ЦЕЛИ НА ПЛАНИРАНЕТО

За да се разработи добър алгоритъм за планиране е необходимо да се определят целите, които трябва да постига планировчикът. Ще разгледаме някои от основните цели.

- **Справедливост**

Времето на ЦП да се разпределя справедливо между процесите. Планировчикът да не дискриминира едни процеси като ги отлага безкрайно дълго.

- **Баланс**

Да се осигури пълно натоварване на всички ресурси на системата, като се създаде добра смес от процеси. Например в паметта да има както ограничени от ЦП така и ограничени от В/И процеси.

- **Ефективност**

Голяма част от времето на ЦП да се използва за изпълнение на потребителски процеси, а не за служебни цели, като превключване на контекста или просто ЦП да престоява, защото няма готови процеси.

- **Време за отговор (response time)**

Това е времето за обслужване на една потребителска заявка, т.е. времето от въвеждане на данни или команда от потребителя до получаване на поредния отговор. Тази цел е важна при интерактивни процеси (процеси, изпълнявани в интерактивен режим). За потребителя е важно системата бързо да реагира след вход от клавиатура или мишка на поредната му заявка.

- **Време за чакане (waiting time)**

Това е общото време, през което процесът чака обслужване в състояние готов. От него зависи времето за изпълнение на процеса, т.е. от създаване до получаване на окончателния резултат. Тази цел е важна при пакетни процеси (процеси, изпълнявани във пакетен или фонов режим). Там не е важно как процесът работи във времето, а да се намали общото време, през което той е в системата.

- **Пропусквателна способност**

Това е още една цел важна при пакетните системи. Пропусквателна способност е брой процеси или задания, преминали през системата за единица време. Целта естествено е тя да се увеличи.

- **Предсказуемост**

Процесите да се изпълняват приблизително за едно и също време независимо от натоварването на системата. Алгоритъмът за планиране да гарантира завършването на всеки процес, т.е. да не се случва безкрайно отлагане.

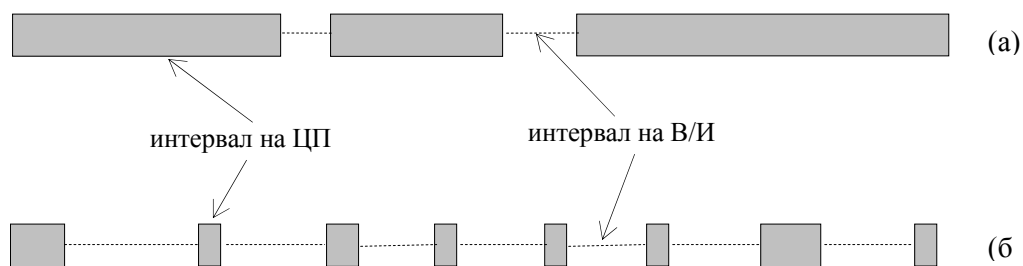
От изброените цели се вижда, че има такива, които са общи, но има и цели, които зависят от типа на процеса. Освен това някои от целите са противоречиви. Това прави планирането една доста сложна задача. Затова има много алгоритми за планиране, реализиращи различни дисциплини.

### 7.3. ДИСЦИПЛИНИ ЗА ПЛАНИРАНЕ

От гледна точка на планирането работата на всеки процес представлява последователност от редуващи се интервали от два типа:

- интервал, в който процесът изпълнява команди (интервал на ЦП или CPU burst)
- интервал, в който процесът чака завършването на входно-изходна операция (интервал на В/И или I/O burst).

Броят и дължината на тези интервали са индивидуални за всеки процес и предварително неизвестни на планировчика. Има процеси, които по-голяма част от времето изпълняват някакви изчисления, т.е. те имат малко на брой но дълги интервали на ЦП (Фиг.8а). Такива процеси се наричат CPU bound или ограничени от възможностите на ЦП. Други процеси често извикват входно-изходна операция и по-голяма част от времето чакат завършването ѝ, т.е. те са с много на брой и къси интервали на ЦП (Фиг.8б). Такива процеси се наричат I/O bound или ограничени от възможностите на В/И устройства.



Фиг. 8. Процес ограничен от ЦП (а); процес ограничен от В/И (б)

Трудността при планиране идва от това, че всеки процес е уникален и непредсказуем за планировчика. Когато планировчикът избере процес за текущ, той не знае колко дълъг е поредния интервал на ЦП, т.е. колко време ще мине преди процесът да се блокира и доброволно да освободи ЦП. Затова важен въпрос при планирането е кога работи планировчикът? Има ситуации, в които той очевидно трябва да се намеси.

- Когато текущият процес завърши като изпълни примитива `exit` и стане зомби.
- Когато текущият процес извика примитив, изискващ блокиране като `read`, `write`, `wait` или друг.

И в двата случая текущият процес доброволно освобождава ЦП защото повече не е в състояние да работи. Ако планировчикът се намесва само в тези два случая, се казва че реализира планиране **без преразпределение (nonpreemptive scheduling)**.

Възможно е планировчикът да се намесва още в един случай. Апаратният таймер изработва прекъсване периферично, например 50 или 60 пъти в секундата. При всяко прекъсване

от таймера работи обработчика на прекъсването в ядрото. След като обработката завърши и преди процесът да се върне в потребителска фаза, може да се намеси планировчика и да избере друг процес за текущ. Планировчикът може да се намесва и след обработката на други апаратни прекъсвания. Например, ако прекъсването е от входно-изходно устройство, което е завършило работа, то ще бъде събуден процесът извикал съответната операция и той ще мине в състояние готов. Планировчикът може да избере този процес за текущ. Планиране, при което ЦП може да се отнема насилствено от текущия процес, се нарича планиране с **преразпределение (preemptive scheduling)**.

Ще разгледаме няколко популярни дисциплини за планиране, които могат да бъдат използвани на различни нива.

#### **Дисциплина FCFS (First-Come-First-Served)**

Процесите се обслужват в реда на появяването им. Има една опашка на готовите процеси, в която новите процеси се добавят в края. Същото се прави и за процесите, които са били блокирани, след като бъдат деблокирани. Когато работи планировчикът избира първия процес в опашката и го изключва от нея. Всеки избран процес работи докато се блокира или завърши, т.е. това е дисциплина без преразпределение и без приоритети на процесите.

Основното преимущество е, че това е лесна за разбиране и реализация дисциплина. Освен това е справедлива, в смисъл че предоставя на всички процеси еднакви услуги, т.е. еднакво средно време за чакане. Недостатъците са повече: Не е приложима в интерактивни ОС, защото няма преразпределение. Кратките процеси чакат толкова колкото и дългите, което някой ще каже, че не е справедливо.

#### **Дисциплина SJF (Shortest-Job-First или най-краткото задание първо)**

Това е също дисциплина без преразпределение, но за разлика от FCFS е приоритетна дисциплина. За планировчика процесите не са еднакви, а имат приоритети, които са обратно пропорционални на времето за изпълнение. Точното време за изпълнение е неизвестно предварително, затова се използва очаквано време, задавано от потребителя. Това е един от недостатъците, тъй като потребителят сам определя приоритета на процесите си. Освен това е неприложима за интерактивни процеси, тъй като се иска оценка за необходимото време за изпълнение.

Предимство на дисциплината е, че кратките процеси се обслужват с предимство, което ще намали тяхното време за чакане, а така ще се намали и средното време за чакане в системата. Нека четири процеса A, B, C и D, които имат времена за изпълнение 8, 4, 4 и 4 съответно, се появят в този ред в опашката на готовите. Да предположим, че те не се блокират, т.е. след като всеки от тях получи ЦП работи съответното време и завършва. При дисциплина FCFS те ще бъдат обслужени в същия ред, следователно времето за чакане на A е 0, на B - 8, на C - 12 и на D - 16, а средното време за чакане е 9. При дисциплина SJF, процесите ще се подредят в реда B, C, D, A. Времената им за чакане ще са 0, 4, 8, 12 и средното време за чакане ще е 6.

#### **Дисциплина SRT (Shortest-Remaining-Time или най-малко оставащо време)**

Идеята на тази дисциплина е винаги да работи процесът, на който му остава най-малко до завършване. Това е приоритетна дисциплина, като приоритетът на процес е обратно пропорционален на оставащото му време за изпълнение, което се изчислява по формулата:

оставащо време = очаквано време - получено време

Прилага се с преразпределение, т.е. когато се появи нов готов процес, ако неговото оставащо време е по-малко от това на текущия в момента, се прави превключване на контекста. При тази дисциплина кратките процеси печелят още повече и се постига минимално средно време за чакане. Но въпреки, че е с преразпределение, и SRT е неприложима за интерактивни процеси, защото се иска оценка на очакваното време.

Общ недостатък на последните две дисциплини е задържането на дългите процеси, което може да доведе до безкрайното им отлагане.

### **Циклична дисциплина или дисциплина RR (Round-Robin)**

Една от най-използваните в интерактивните ОС е цикличната дисциплина, наричана още дисциплина RR. Всички готови процеси са подредени в опашка по реда на появяването им (нов или деблокиран процес се добавя в края). Планировчикът избира първия процес от опашката и му определя интервал време, наричан квант (quantum), през който той може да използва ЦП. Ако при изтичане на кванта процесът не е завършил или не се е блокирал, то му се отнема ЦП и се поставя в края на опашката на готовите процеси. Следователно това е дисциплина с преразпределение. Предимствата на дисциплината RR са много:

- Дава предимство на кратките процеси без да дискриминира дългите.
- Осигурява приемливо време за отговор в интерактивните ОС за процесите ограничени от В/И.
- Проста за реализация е.
- Справедлива е.

Основен параметър на тази дисциплина е размерът на кванта. Голям или малък да е кванта, с фиксиран или променлив размер да е за различните процеси. От отговорите на тези въпроси зависи ефективността на RR. Ако дисциплината използва голям квант, то ще нарастне времето за чакане на процес между два последователни кванта, което е недостатък при процесите ограничени от В/И. При достатъчно голям квант, дисциплината става FCFS за повечето процеси. Малък квант означава често превключване на контекста, което намалява ефективността на системата. Например, нека превключването на контекста изисква 5 msec. Ако квантът е 20 msec, то 20% от времето на ЦП ще се използва за служебни цели. Ако квантът е 500 msec, то по-малко от 1% от времето ще е за превключване на контекста (предполагаме, че процесите използват целия си квант). От друга страна, ако предположим, че 10 потребителя почти едновременно натиснат клавиш ENTER, то 10 процеса ще се наредят в опашката на готовите процеси. При квант 500 msec, последният процес ще чака около 5 sec.

### **Дисциплини с няколко опашки**

В дисциплината RR всички процеси са равноправни. Но често в ОС това не е вярно. Има процеси, които трябва да се изпълняват с по-висок приоритет, например системни процеси, процеси демони или процесите ограничени от В/И да са по-приоритетни от процесите ограничени от ЦП. Идеята на тази група дисциплини е процесите да се класифицират в няколко класи и за всеки клас да се поддържа опашка на готовите процеси. Всяка опашка има свой приоритет и може да се обслужва с различна дисциплина. Когато ЦП се освободи планировчикът избира процес от непразната опашка с най-висок приоритет и го обслужва според дисциплината на опашката му. Процес от опашка с по-нисък приоритет се избира само, ако опашките с по-висок приоритет са празни.

Важен въпрос е как процесите се разпределят по опашките. Съществуват два основни принципа за това:

Връзката между процес и опашка да е статична, което означава, че определен процес винаги попада в една и съща опашка когато е готов. Такава дисциплина е използвана в MINIX.

Друга възможност е тази връзка да е динамична, т.е. един процес през своя живот може да попада в различни опашки в зависимост от поведението си. Такава дисциплина се нарича **опашки с обратна връзка**, защото дисциплината се адаптира към поведението на процеса като повишава или намалява приоритета му. Пример за ОС, реализираща такава дисциплина е UNIX.

## Планиране на процесите в UNIX

Алгоритъмът за планиране на процесите в UNIX използва няколко опашки с обратна връзка. Всяка опашка има свързан с нея приоритет. Приоритетите са цели числа, като по-малко значение означава по-висок приоритет. Дапазонът на приоритетите се дели на два непресячащи се класа: потребителски приоритети и системни приоритети, които са по-високи от първите. Фиг. 9. илюстрира използваните опашки за планиране в UNIX.

Системен приоритет се дава на процес, изпълняван в системна фаза, който се е блокирал. Значението на системния приоритет зависи от събитието, което процесът чака. Когато такъв процес се деблокира и мине в състояние готов, ще бъде добавен в опашката на съответния системен приоритет.

Потребителски приоритет има процес, изпълняван в потребителска фаза и такъв, който е бил свален от планировчика в преразпределен.

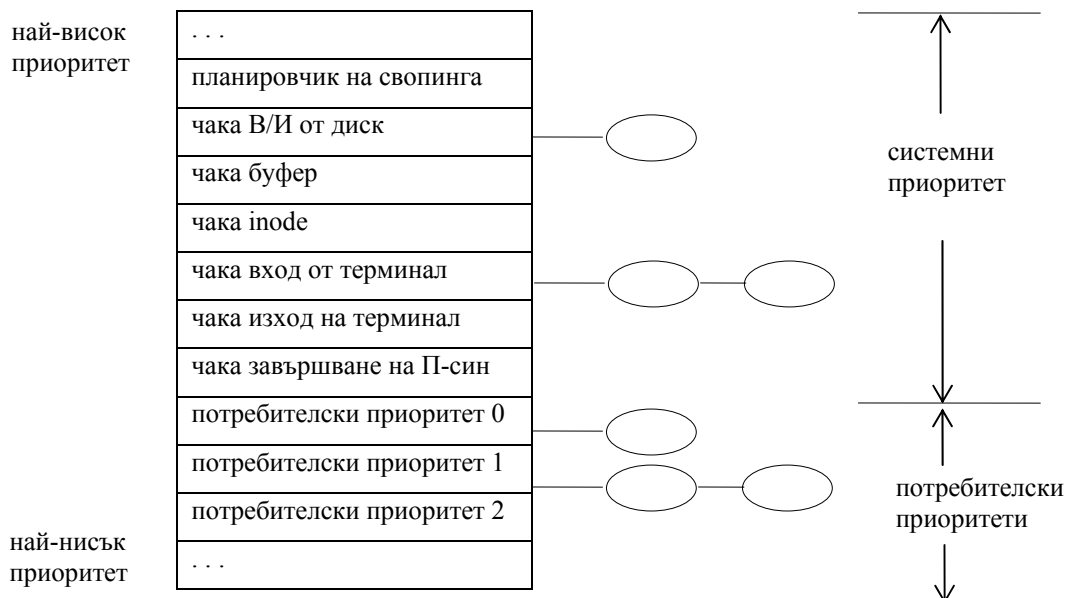
Кога и как планировчикът преизчислява приоритетите на процесите?

- Когато процес минава в състояние блокиран, му се дава съответен системен приоритет.
- Когато процес се връща от системна в потребителска фаза му се дава потребителски приоритет, защото може да е бил блокиран и приоритетът му е системен.
- При обработка на прекъсване от апаратния таймер на всяка секунда се преизчисляват всички потребителски приоритети по формулата:

$$priority = CPU/2 + base + nice$$

В тази формула *CPU* и *nice* са полета в таблицата на процесите. Полето *CPU* отчита използването на ЦП от процеса напоследък. При всяко прекъсване от таймера неговото значение се увеличава с 1 за текущия процес. Но наказанието за използването на ЦП не е вечно, периодично полето *CPU* се намалява (дели се на 2 в горната формула или по друг начин). Компонентата *base* е прагов приоритет между системни и потребителски приоритети, който не позволява процес в потребителска фаза да получи системен приоритет. Полето *nice* се зарежда чрез едноименен системен примитив, изпълнен от процеса. Чрез него процес може да понижи или повиши приоритета си, но само процес на администратора може да зададе значение, с което да повиши приоритета.

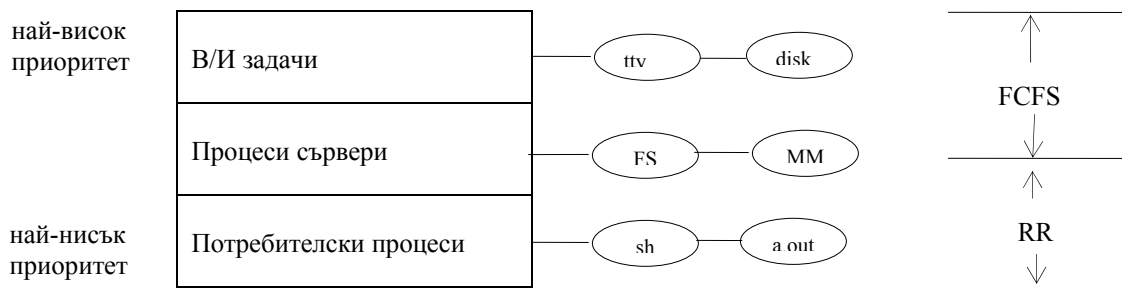
Основната идея на алгоритъма за планиране е процесите бързо да преминават през системна фаза.



Фиг. 9. Планиране на процесите в UNIX

## Планиране на процесите в MINIX

Алгоритъмът за планиране на процесите в MINIX използва три опашки. Всяка опашка има свързан с нея приоритет. Връзка между процесите и опашките е статична и е показана на Фиг. 10.



Фиг. 10. Планиране на процесите в MINIX

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