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 воперационната 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 едновременото 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 пораждане 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 става 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. Програмата `getty` чака вход от съответния терминал и когото 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.

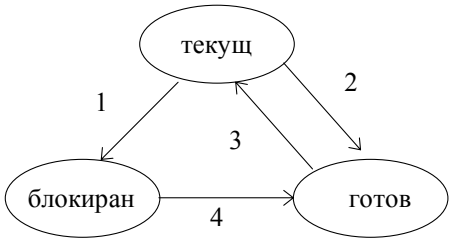
1.2. СЪСТОЯНИЕ НА ПРОЦЕС

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. 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 подробно по-нататък.

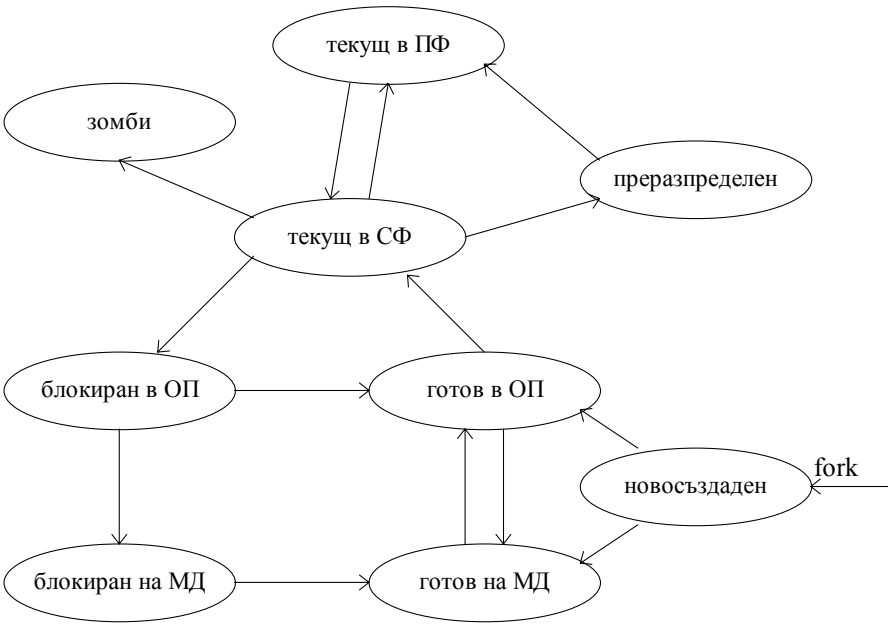


Фиг. 1. Модел на процесите с три състояния и диаграма на преходите

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 изпълни `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 "преразпределен". 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 предизвиква прекъсване и някой друг процес, който в момента е в състояние "текущ в потребителска фаза", ще влезе в състояние "текущ в системна фаза", а модулът 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 състоние "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 "преразпределен" 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 "преразпределен", 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 процеса 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 съдържа данни, локални за функциите 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.

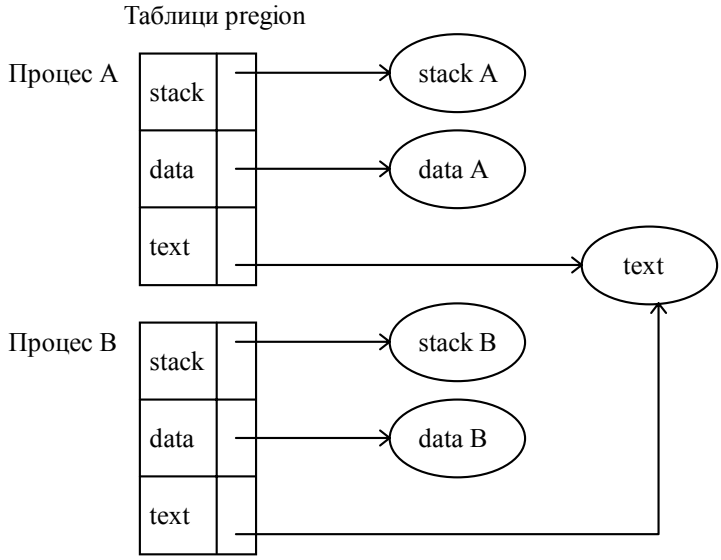
Информацията за всички активни региони се съхранява в ***Таблица на регионите***. Всеки запис в тази таблица описва един регион и съдържа:

- 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 irije;kd 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, като 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, 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 (U area) - съдържа данни за процеса, които са необходими и достъпни за ядрото само когато той е 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 за изпратените на процеса, но още необработени сигнали
- полета, съхраняващи времена - използваното от процеса време на ЦП в системна и потребителска фази, таймер, зареден с примитива `alarm` и други
- код на завършване на процеса

Следващите полета са в **потребителската област**:

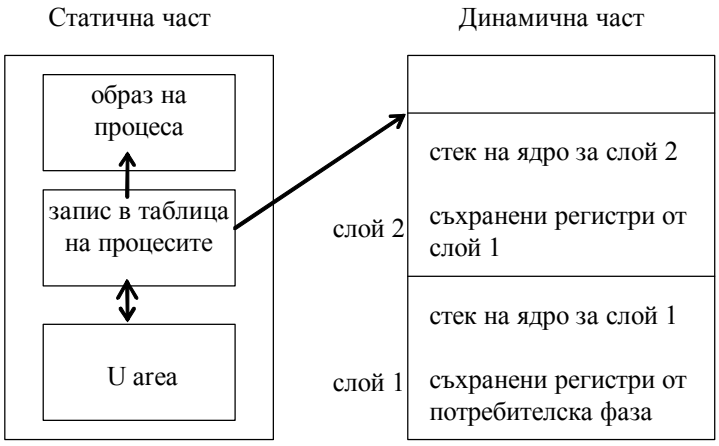
- указател към записа от таблицата на процесите
- файлови дескриптори (таблица на файловите дескриптори)
- текущ каталог - указател към запис от таблицата на индексните описатели
- сменен коренен каталог
- управляващ терминал – указател към tty структура на терминал или `NULL`
- маска, използвана при създаване на файлове и заредена с примитива `umask`
- 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, определящи реакцията на процеса при получаване на различните сигнали
- 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. 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, съставлявщи 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 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 C 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 C 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:

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 библиотечните функции ще наричаме 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

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. Един процес “влиза” във функцията `fork`, а два процеса “излизат от” `fork` с различни връщани значения. Да разгледаме по-подробно това, което става и 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 (файловите дескриптори, текущ каталог, управляващ терминал, маска при създаване на файл, потребителските идентификатори - euid, ruid, egid и rgid, реакциите на сигнали).
3. Създава образ на новия процес - копие на образа на процеса-баща (региона за код 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 - в системна фаза на `fork` и ще върне 0.

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

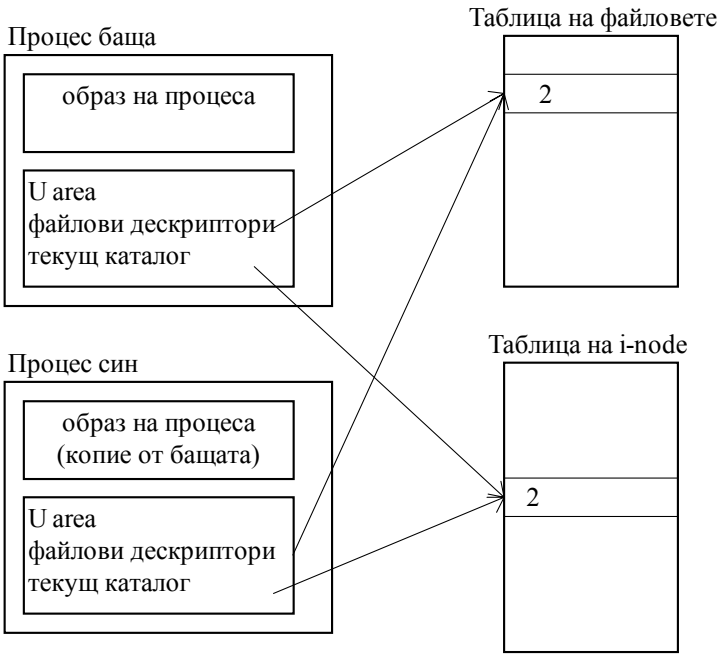
- разделят достъпа до файловете, които бащата е отворил преди да изпълни `fork`. Това позволява пренасочване на входа и изхода и свързване на процеси с програмни канали.

- имат един и същ текущ каталог, управляващ терминал и група процеси и сесия;

- имат еднакви права - реален и ефективен потребителски идентификатори, което гарантира неизменност на привилегиите на потребител при работа в системата;

- 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.

Фиг.5 илюстрира общите елементи в контекста на двата процеса.



Фиг. 5. Процеси син и баща

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 процеса `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 четири 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. 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.

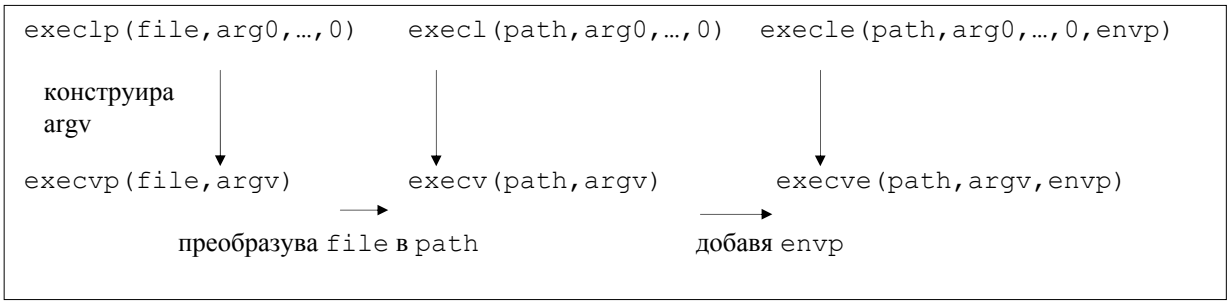
4. Създава нов образ на процеса, използвайки изпълнимия код във файла и копира аргументите на `exec` в новия потребителски стек.

5. Изменя значенията 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 файловите дескриптори, текущия каталог, управляващия терминал, групата на процесите, сесията, маската при създаване на файлове. 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.

Връзката между шесте функции на примитива `exec` е показана на следната схема:



В трите функции на горния ред, всеки аргумент на `main` е зададен като отделен аргумент на `exec` функцията. В трите функции на долния ред има един аргумент `argv`, който е масив от указатели към аргументите за `main`.

В двете функции в ляво `file` е собствено име на файл, което се преобразува в пълно име чрез променливата `PATH`.

Двете функции в дясно имат аргумент `envp`, който е масив от указатели към символни низове, съдържащи променливите в обкръжението на процеса. В четирите функции в ляво няма аргумент за обкръжението на процеса и се използва значението на глобалната променливата `environ`.

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 файловите 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 и пренасочване на стандартния изход на процеса-син към файл `outfile`.

```
main()
{
int pid, fd, status;
```

```
pid = fork();
if (pid == 0) {       /* in child process */
      if ( ( fd=creat("outfile", 0644) ) == -1) {
            perror("Child cannot create outfile. ");
            exit(1); }
      close(1);
      dup(fd);
      close(fd);
      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);}
}
```

Потребителски идентификатори на процес

С всеки процес са свързани два потребителски идентификатора: реален - ruid и ефективен - euid. Реалният е идентификатора на потребителя, който е създал процеса, а по ефективния се определят правата на процеса при работа с файлове и при изпращане на сигнали. Обикновено ефективният е еднакъв с реалния, освен ако не е променен. В същност се пази още един ефективен идентификатор (в таблицата на процесите), наричан съхранен uid - suid, който се използва, за да може да се възстанови временно изменен euid. Промяна на ефективния идентификатор може да стане по два начина.

Когато се изпълнява системен примитив `exec` и програмата е set-UID се променя ефективния потребителски идентификатор. Една програма се нарича set-UID, когато в кода на защита на файла с изпълнимия код битът "изменение на UID при изпълнение" е 1 (т.е. 04000). Тогава при `exec` се сменя euid и suid на процеса със собственика на файла с изпълнимия код. Това означава, че по време на изпълнение на новата програма процесът ще има правата на собственика на файла с програмата. Този начин се използва от някои команди за контролирано и временно повишаване на правата на потребителите. Например, set-UID програма е `passwd`, чрез която всеки потребител може да смени паролата си, т.е. да пише във файла с паролите. Файлът на командата `passwd` е собственост на `root`, следователно процесът, в който се изпълнява тя има правата на `root`.

Другият начин за промяна на потребителските идентификатори е чрез системен примитив.

`int setuid(uid_t uid);`

Ако текущият euid на процеса е 0 (на `root`), то се променя euid, ruid и suid с параметъра *`uid`*. Ако текущият euid на процеса не е 0 (на `root`), то се променя euid с параметъра *`uid`*, само ако параметърът *`uid`* е равен на ruid или на suid. При успех връща 0.

`uid_t getuid(void);`

`uid_t geteuid(void);`

Системният примитив `getuid` връща реалния потребителски идентификатор на процеса, а `geteuid` - ефективния потребителски идентификатор. Винаги завършват успешно.

Групи процеси, сесия и управляващ терминал

Всеки процес принадлежи на група от процеси, която включва един или повече процеса. Всяка група има лидер на групата (group leader), който е процесът, създал групата. Групата съществува докато съществува поне един от процесите в нея, независимо дали лидерът е завършил или не. Последният процес от групата може или да завърши или да премине към

друга група. Групата се идентифицира чрез идентификатор на група процеси (process group ID или PGID), който в същност е pid на процеса-лидер на групата. Следователно, всеки процес има и идентификатор на група процеси, ще го наричаме групов идентификатор за по-кратко. Процес-син наследява груповия идентификатор от процеса-баща, а при `exec` груповият идентификатор не се променя.

Следват системните примитиви свързани с понятието група процеси.

```
pid_t getpgrp(void);        /* POSIX */
pid_t getpgid(pid_t pid);   /* SVr4 */
```

Системният примитив `getpgrp` връща груповия идентификатор на процеса, който го изпълнява, т.е. всеки процес може да научи към коя група принадлежи. Системният примитив `getpgid` връща груповия идентификатор на процес с идентификатор *pid*. Ако *pid* е 0, то се връща групата на текущия процес. Процесът, изпълняващ системния примитив трябва да принадлежи на сесията, към която принадлежи и процеса *pid*. При грешка `getpgid` връща -1, а `getpgrp` винаги завършва успешно.

Процес може да смени групата, към която принадлежи като създаде своя група или се присъедини към друга съществуваща група, чрез системните примитиви:

```
int setpgrp(void);                      /* BSD4.2 */
int setpgid(pid_t pid, pid_t pgid);     /* POSIX */
```

Системният примитив `setpgrp` създава нова група и процесът, който го изпълнява става неин лидер, т.е. групов идентификатор на процеса става неговия `pid`.
При `setpgid` процес с идентификатор *pid* преминава към група *pgid*. Ако *pid* е 0, то се използва идентификатора на текущия процес, а ако *pgid* е 0, то се използва идентификатора *pid*. Чрез този примитив процес може да смени групата за себе си или процес-баща да смени групата за свой процес-син. Примитивът `setpgid(0,0)` има същото действие както `setpgrp()`. При успех `setpgrp` и `setpgid` връщат 0.

Понятието група процеси се използва от механизма на сигналите.

Понятието сесия е въведено в UNIX системите с цел логическо обединение на процесите, създадени в резултат на login и последващата работа на един потребител. Сесията включва една или повече групи процеси. Всяка сесия има лидер на сесия, който е процесът, създал сесията. Аналогично на групите, сесията се идентифицира чрез идентификатор на сесия (session ID или SID), който в същност е pid на процеса-лидер на сесията. Следователно, всеки процес притежава и идентификатор на сесията, който наследява от процеса-баща, а при `exec` идентификатора на сесия не се променя.

Следват системните примитиви свързани с понятието сесия.

```
pid_t getsid(pid_t pid);
pid_t setsid(void);
```

Системният примитив `getsid` връща идентификатора на сесия за процес с идентификатор *pid*. Ако *pid* е 0, то се използва идентификатора на текущия процес. Процес с правата на `root` може да изпълни примитива за всеки друг процес, но за процес с обикновени права *pid* трябва да е идентификатор на процес от същата сесия.

Нова сесия се създава с `setsid` ако процесът, изпълняващ примитива, преди това не е лидер на група. При успех процесът, изпълняващ примитива става лидер на новата сесия, лидер на първата група в тази сесия и няма управляващ терминал. При успех примитивът връща идентификатора на новата сесия, а при грешка връща -1.

Понятията група и сесия са тясно свързани с понятието терминал. Това позволява на ядрото да контролира стандартния вход и изход на процесите, а също и да им изпраща сигнали за събития, свързани с терминала. Всеки процес има поле за управляващ терминал в U area. Какво съдържа това поле? Всеки терминал има свързана с него tty структура. Полето за управляващ терминал в U area на процеса е указател към tty структурата на управляващия му терминал. Ако това поле има значение `NULL`, то процесът няма управляващ терминал.

Как се свързва управляващ терминал с процес? Обикновено ние не трябва да се тревожим за управляващия терминал, тъй като го получаваме при login. Всеки процес наследява управляващия терминал от процеса-баща при `fork`, а при `exec` управляващият терминал не се променя. Процесът `getty` има управлявляващ терминал, който се наследява от login-shell процеса, който става лидер на сесия. След това всички процеси, пораждани от login-shell процеса при изпълнение на команди наследяват от него управляващия терминал и идентификатора на сесия, т.е. принадлежат на една сесия и са свързани с един управляващ терминал.

Но как `getty` процесът е получил управляващ терминал? Процес лидер на сесия установява връзка с управляващ терминал. Начинът, по който се установява връзка с управляващ терминал е системно зависим. Например, в Unix System V и LINUX ядрото свързва управляващ терминал с процес, когато той изпълнява примитива `open` на специален файл за терминал (напр., `fd=open("/dev/tty1", …);`) и ако са изпълнени условията:

- Терминалът в момента не е управляващ терминал на сесия.
- Процесът е лидер на сесия. (в 4.3BSD е различно)

Следващият въпрос е как се прекъсва връзката на процес с управляващия му терминал? Това пак е различно в различните UNIX и LINUX системи. В LINUX това става с примитива `setsid`, когато се изпълнява от процес който не е лидер на група. Тогава той става лидер на нова група, на нова сесия и губи управляващия си терминал.

Когато сесията има управляващ терминал, групите в нея са: една привилегирована (foreground group) и всички други - фонови групи (background group). В tty структура на управляващия терминал има поле, което съдържа идентификатор на група процеси (terminal group ID или TPGID) - привилегированата в момента. Това поле определя групата процеси, на които се изпращат сигнали, свързани с терминала: `SIGINT`, `SIGQUIT`, `SIGHUP`, `SIGTSTP`, `SIGCONT` (първите три са общи за всички UNIX и LINUX системи, последните два са от 4.3BSD). Така, когато въведем <Ctrl>+<C> на терминала, ядрото изпраща сигнал `SIGINT` на всички процеси от привилегированата група. Освен това, входа от терминала също се изпраща към процесите от привилегированата група.

При `exit` на процес-лидер на сесия с управляващ терминал, се прекъсва връзката между сесията и терминала, т.е. сесията вече няма управляващ терминал. Също така се изпраща сигнал `SIGHUP` на процесите от привилегированата група.

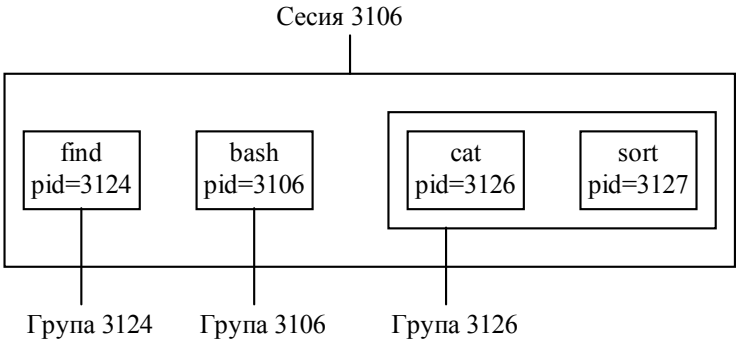
Управление на заданията

Управление на заданията (Job control) е възможност добавена в BSD към 1980г. и приета в POSIX. Някои командни интерпретатори, като Korn shell, C shell и Bash реализират понятието **задание** (наричат ги job control shell) като използват групи процеси. Други командни интерпретатори, като B shell не поддържат задания. Заданието е един или повече процеса, които са в една група. Всички процеси пораждани от процеса login shell са организирани като задания, т.е. групирани в групи, като една от тях е привилегирована (foreground group), а всички останали са фонови групи (background group). Реализацията на задания изисква подръжка и от ядрото - от терминалния драйвер и механизма на сигналите.

Да разгледаме изпълнението на следващите команди от поддържащ заданията Bash:

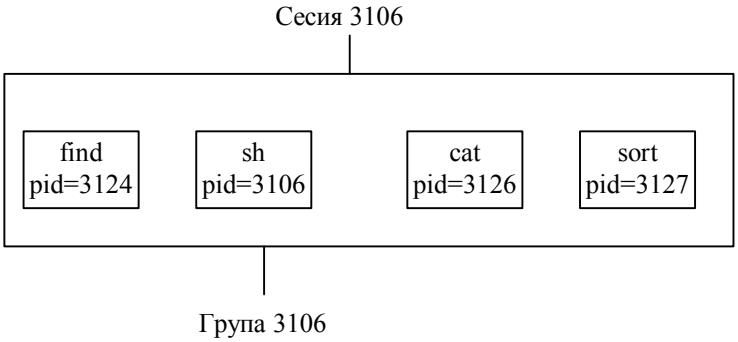
```
$ find /home -name core -print  > outfind 2>&1 &
$ cat | sort
```

Всичките четири процеса bash, find, cat и sort принадлежат на една сесия с лидер процеса bash и имат един и същ управляващ терминал. Процесите cat и sort принадлежат на една група с лидер cat, която в момента е привилегирована, процесът find е лидер на друга група и bash е лидер на третата група.



Фиг. 6а. Сесия и групи процеси

При неподдържащ заданията B shell, всичките четири процеса sh, find, cat и sort принадлежат на една група с лидер процеса sh и на една сесия със същия лидер и имат един и същ управляващ терминал.



Фиг. 6б. Сесия и групи процеси в B shell

Следната програма `job_test.c`, ще ни покаже с какъв shell работим.

```
main()
{
  printf("pid=%d, pgrp=%d\n", getpid(), getpgrp());
}
```

Ако изпълним тази програма в неподдържащ заданията B shell, ще получим резултати, подобни на следните. Предполагаме, че login shell процесът е с pid 3106. В последния случай, липсващия pid 3131 е на процеса subshell. И в трите случая пораждaните процеси са в една група с лидер процеса login shell.

`job_test`	`pid=3128, pgrp=3106`
`job_test & job_test &`	`pid=3129, pgrp=3106` `pid=3130, pgrp=3106`
`(job_test & job_test &)`	`pid=3132, pgrp=3106` `pid=3133, pgrp=3106`

Ако изпълним тази програма в поддържащ заданията Bash, ще получим резултати, подобни на следните. Всеки път програмата се изпълнява в собствена група. В последния случай, резултатите може да са различни, в зависимост от това дали процес subshell (pid 3131) поддържа задания. В случая двата процеса са в една група с лидер процеса subshell.

`job_test`	`pid=3128, pgrp=3128`
`job_test & job_test &`	`pid=3129, pgrp=3129` `pid=3130, pgrp=3130`
`(job_test & job_test &)`	`pid=3132, pgrp=3131` `pid=3133, pgrp=3131`

Освен това, когато поддържащ заданията shell стартира ново задание в привилегирован режим, сменя полето TPGID в tty структурата на управляващия терминал да съответства на това задание. Докато неподдържащ заданията shell никога не сменя значението на полето TPGID, т.е. това поле винаги съдържа pid на login shell процеса.

4. СИГНАЛИ

Сигналите информират процес за настъпване на асинхронни събития вън от процеса или на особени събития в самия процес. Сигналите могат да се разглеждат като най-примитивния механизъм за междупроцесни комуникации, но също така те много напомнят механизма на прекъсвания. Сигналите се появяват още в най-ранните версии на UNIX, но в реализацията им има някои недостатъци. В следващите версии на BSD и UNIX System V са внесени изменения, но моделите в двете версии са несъвместими, затова всички версии на UNIX и LINUX поддържат и първоначалната семантиката на "ненадеждните" сигнали. По нататък ние ще разглеждаме именно тези сигнали.

Типове сигнали

Всеки сигнал има уникален номер и символно име, които определят събитието, за което информира сигнала. В ранните версии има 15 типа сигнала, а в новите броят им е около 30. Типовете сигнали могат да се класифицират в зависимост от събитието, свързано със сигнала:

- Сигнали, свързани с управляващия терминал:

Когато потребителят натисне клавиша <Del> или <Ctrl>+<C> на процеса (процесите от привилегированата групата) се изпраща сигнал `SIGINT`. Сигнал `SIGQUIT` се изпраща при клавиши <Ctrl>+<\>. Сигнал `SIGHUP` се изпраща при прекъсване на връзката с управляващия терминал.

- Сигнали свързани с апаратни особени ситуации

Сигналите в тази категория са свързани със събития, откривани от апаратурата и сигнализирани чрез прекъсване. Ядрото реагира на това като изпраща сигнал на процеса, който е текущ в момента. Например, някои от типовете и събитията са:

`SIGFPE` Изпраща се при деление на 0 или препълване при операции с плаваща точка.
`SIGBUSS` Изпраща се при обръщение към адрес, за който отсъства физическа страница.
`SIGILL` Изпраща се при опит за изпълнение на недопустима инструкция.
`SIGSEGV` Изпраща се при обръщение към недопустим адрес или към адрес, за който процесът няма права.

- Сигнали свързани с програмни ситуации

Сигналите в тази категория са свързани с най-различни събития, синхронни или асинхронни с процеса, на който са изпратени, които имат чисто програмен характер и не се сигнализират от апаратурата. Някои от типовете сигнали са:

`SIGCHLD` Изпраща се на процес-баща когато някой негов процес-син завърши.
`SIGALRM` Изпраща се когато изтече времето, заредено от процеса, чрез системния примитив `alarm`.
`SIGPIPE` Изпраща се при опит на процес да пише в програмен канал, който вече не е отворен за четене.

Един процес може да изпрати на друг процес сигнал чрез системен примитив `kill`. Някои от типовете сигнали, които могат да бъдат изпратени само чрез `kill` са:

`SIGKILL` Предизвиква безусловно завършване на процеса.
`SIGTERM` Предупреждение за завършване на процеса.
`SIGUSR1` Потребителски сигнал, използван от потребителските процеси като средство за междупроцесни комуникации.
`SIGUSR2` Още един потребителски сигнал

Изпращане и обработка на сигнали

Сигнал може да бъде изпратен на процес или от ядрото или от друг процес чрез системния примитив `kill`. Ядрото помни изпратените, но още необработени от процеса сигнали в записа от таблицата на процесите. Полето е масив от битове, в който всеки бит отговаря на тип сигнал. При изпращане на сигнал, съответния бит се вдига. С това работата по изпращане е завършена.

Обработката на изпратените сигнали се извършва в контекста на процеса, получил сигнала, когато процесът се връща от системна в потребителска фаза. Следователно, сигналите нямат ефект върху процес, работещ в системна фаза докато тя не завърши. Има три възможни начина да бъде обработен един сигнал, ще ги наричаме реакции на сигнал:

- Процесът завършва (това е реакцията по премълчаване за повечето типове сигнали).
- Сигналът се игнорира.
- Процесът изпълнява определена потребителска функция, след което продължава изпълнението си от мястото където е бил прекъснат.

За всеки тип сигнал реакцията при получаването му се помни в U area, където полето е масив от адресите на обработчиците на сигналите, един за всеки тип сигнал.

Процес може да определи реакцията си при получаване на сигнал от определен тип, ако иска тя да е различна от тази по премълчаване, чрез системния примитив `signal`.

```
#include <signal.h>
void (*signal(int sig, void (*sighandler)())());
```

Аргументът *sig* задава номера на сигнала, а *sighandler* определя каква да е реакцията при на получаване на сигнал. Значението на втория аргумент може да е `SIG_IGN` - игнориране на сигнал, `SIG_DFL` - реакция по премълчаване, т.е. завършване на процеса или име на потребителската функция. Реакцията се запомня в съответното поле от U area и с това работата на примитива приключва. При успех връща адреса на предишната реакция, който може да бъде запомнен и по-късно възстановен. При грешка връща -1.

Процес-син наследява реакциите на сигнали от процеса-баща. След `exec` за всички сигнали, за които реакцията е била променена с потребителска функция, се връща реакцията по премълчаване. Това е естествено поведение, тъй като при `exec` се сменя образа на процеса.

Когато по-късно пристигне сигнал от съответния тип, той ще бъде обработен според запомнената в U area реакция. Ако това е била потребителска функция, то преди обработката се връща реакцията по премълчаване. Следователно, ако процес иска да обработва повтарящи се сигнали от един тип чрез потребителска функция, трябва отново да изпълнява `signal` след всеки получен сигнал. Това решение изглежда вярно, защото в повечето случаи работи правилно, но не е. Нов сигнал може да пристигне преди процесът да успее да изпълни `signal`, тъй като когато процес работи в потребителска фаза, ядрото може да направи превключване на контекста преди процесът да е стигнал до `signal`. Затова се препоръчва `signal` да е в началото на функцията, обработваща сигнала.

Сега вече се виждат недостатъците в семантиката на ненадеждните сигнали са:

- Може да се получи състезание при повтарящи се сигнали от един тип.
- Може да има загуба на сигнали, тъй като няма памет, в която да се помнят няколко изпратени сигнала от един тип.
- Процес не може да блокира и след това разблокира, получаването на сигнали за известно време, като ядрото да помни изпратените през това време сигнали (подобно на апаратните прекъсвания).
- Процес не може да провери реакцията си за определен тип сигнал без да я променя.

Изпращане на сигнал от един процес към друг(и) се извършва с примитива `kill`.

```
int kill(pid_t pid, int sig);
```

Аргументът *sig* задава номера на типа сигнал, който се изпраща. Аргументът *pid* определя процесите, на които се изпраща сигнала. Възможните значения на *pid* са:

- число по-голямо от 0 - Сигнал се изпраща на процеса с идентификатор *pid*.
- 0 - Сигнал се изпраща на всички процеси от групата на процеса, изпращащ сигнала.
- число по-малко от -1 - Сигнал се изпраща на всички процеси от групата с идентификатор |*pid*|.
- -1 - Сигнал се изпраща на всички процеси в таблицата на процесите, започвайки от процеса с най-голям идентификатор, без процеса `init` (с pid 1) и системните процеси.

Във всичките случаи е необходимо процесът, изпращащ сигнала да има права:

- ефективният потребителски идентификатор (euid) на процеса, изпращащ сигнала, да е 0 (`root`).

- ruid или euid на процеса, изпращащ сигнала, да е еднакъв с ruid или suid на процеса, на когото се изпраща сигнала.

В противен случай сигнал не се изпраща и примитивът връща -1. При успех връща 0. Ако *sig* е 0, сигнал не се изпраща, но се прави проверка за грешка.

`int pause(void);`

Системният примитив `pause` блокира процеса, който го изпълнява до получаването на първия сигнал, за който реакцията не е игнориране. Ако реакцията за първия пристигнал сигнал е `SIG_DFL`, то процесът завършва, т.е. връщане от `pause` няма. Ако за сигнала е предвидена потребителска функция и от тази функция има връщане, то след изпълнение на функцията процесът продължава след `pause`. В този случай има връщане от `pause` и функцията връща -1.

`unsigned int alarm(unsigned int sec);`

Системният примитив `alarm` планира изпращането на сигнал `SIGALRM` на процеса, изпълняващ примитива, след *sec* секунди, т.е. зарежда в таймера на процеса значението на аргумента *sec*. Когато изтече този интервал от време ядрото ще изпрати сигнал `SIGALRM` на процеса. Ако преди това е било планирано друго изпращане на сигнал `SIGALRM`, което не се е състояло, то се анулира и примитивът връща броя секунди, оставащи до него. Иначе връща 0. Чрез аргумент *sec* 0 може да се анулира зареден преди това таймер без да се планира нов сигнал.

Пример. Програмата `faccess` илюстрира механизма на сигналите. На всяка минута проверява дали през последната минута е осъществяван достъп до файл със зададено в аргумента име и ако е така извежда съобщение. Процесът може да бъде прекратен с клавишите <Ctrl>+<C> или с командата `$kill <pid>`.

```
#include <signal.h>
#include <sys/stat.h>
#include <stdio.h>

void wakeup()
{
      signal(SIGALRM, wakeup);
}

void quit()
{
      printf("Termination of %d\n", getpid()); exit(0);
}

main(int argc, char *argv[])
{
struct stat statbuf;
time_t atime;
if (argc != 2) {
   fprintf(stderr, "usage: faccess filename\n");
   exit(1); }
if ( stat(argv[1], &statbuf) == -1 ) {
   fprintf(stderr, "faccess: %s: No such file\n", argv[1]);
   exit(1); }
atime = statbuf.st_atime;
signal(SIGINT, quit);
signal(SIGTERM, quit);
signal(SIGALRM, wakeup);
for(;;) {
   if ( stat(argv[1], &statbuf) == -1 ) {
      fprintf(stderr, "faccess: %s: No such file\n", argv[1]);
      exit(1); }
   if ( atime != statbuf.st_atime ) {
```

```
      printf("file %s: accessed\n", argv[1]);
      atime = statbuf.st_atime; }
      alarm(60);
      pause();
   }
}
```

Пример. Програмата илюстрира механизма на сигналите и групи процеси.

```
#include <signal.h>

void wakeup()

{

}

main ()
{
int i;
   printf("Parent %d, group %d\n", getpid(), getpgrp());
   for (i=0; i<2; i++) {
      if ( fork() == 0 ) {   /* in child processes  */
         if (i & 01 ) setpgrp();
         printf("Child %d, group %d\n", getpid(), getpgrp());
         pause();
      }
   }
   signal(SIGINT, SIG_IGN); /*without signal,kill also parent*/
   signal(SIGALRM, wakeup);
   alarm(5);
   pause();
   kill(0, SIGINT);
   printf("Parent after kill\n");
}
```

Процесът-баща създава два процеса-синове. Единият от тях изпълнява `setpgrp` и става лидер на нова група процеси, а другият остава в групата на процеса-баща. След това и двата сина изпълняват `pause` и се блокират до получаване на първи сигнал. Процесът-баща изчаква 5 секунди е изпраща сигнал `SIGINT` на всички процеси от групата си. Този сигнал ще убие единия от синовете му, който е в неговата група, но не и другия син, който е в друга група. Също така няма да убие и самия процес-баща, тъй като той игнорира сигнала, изпълнил е `signal`. След това бащата извежда съобщението с `printf` и завършва. Другият процес-син може да бъде убит с командата `kill` (той е фонова група).

5. ПРОГРАМНИ КАНАЛИ

Програмният канал е механизъм за комуникация между процеси. Той осигурява еднопосочно предаване на неформатиран поток от данни (поток от байтове) между процеси и синхронизация на работата им. Реализират се два типа канали в различните версии на UNIX и LINUX системите:

- **неименован** (**pipe**) - за комуникация между родствени процеси
- **именован** (**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.
- Използват се само с директни блокове, т.е. каналът има доста ограничен капацитет 10 или 12 блока.

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. По-нататък ще

разглеждаме по-простия тип неименовани програмни канали, които се реализират още от най ранните версии на UNIX и за по-кратко ще го наричаме програмен канал.

Програмен канал се създава чрез примитива `pipe`.

`int pipe(int fd[2]);`

Създава се нов файл от тип програмен канал, което включва разпределяне и инициализиране на свободен индексен описател, както и при обикновените файлове, но за разлика от тях, каналът няма външно име и следователно не е част от йерархията на файловата система. След това каналът се отваря два пъти - един път за четене и един път за писане. Примитивът връща файлов дескриптор за четене в *`fd[0]`* и файлов дескриптор за писане в *`fd[1]`*.

Писане и четене в програмен канал се извършва с примитивите `write` и `read`, но достъпа до данните е с дисциплина FIFO, т.е. всяко писане е добавяне в края на файла и данните се четат от канала в реда, в който са записани. За програмен канал не е разрешен примитива `lseek`. Броят на процесите-четящи и процесите-пишещи в канала може да е различен и да е по-голям от 1, но тогава синхронизацията, осигурявана от механизма не е достатъчна.

Писане в програмен канал

1. Ако в канала има достатъчно място, то данните се записват в края на файла, увеличава се размера на файла със записания брой байта и се събуждат всички процеси, чакащи да четат от канала.

2. Ако в канала няма достатъчно място за всичките данни и броя байта, които се пишат при това извикване е по-малък от капацитета на канала, то ядрото блокира процеса. Когато бъде събуден от друг процес, изпълняващ `read`, той продължава както в случай 1.

3. Ако в канала няма достатъчно място за всичките данни, но броя байта, които се пишат при това извикване е по-голям от капацитета на канала, то в канала се записват толкова байта, колкото е възможно и ядрото блокира процеса. Когато бъде събуден, той продължава да пише. В този случай операцията писане не е атомарна и е възможно състезание когато броят на процесите-писатели е по-голям от 1.

Четене от програмен канал

1. Ако в канала има някакви данни, то започва четене от началото на файла докато се удовлетвори искането на процеса или докато има данни в канала. Намалява размера на файла с прочетения брой байта и събужда всички процеси, чакащи да пишат в канала.

2. Ако каналът е празен, ядрото блокира процеса. Когато бъде събуден от друг процес, изпълняващ `write`, той продължава както в случай 1.

Затваряне на програмен канал

Файловите дескриптори, върнати от `pipe`, за четене и писане в канал се освобождават с `close` както и при работа с обикновени файлове, но има някои допълнения към алгоритъма на `close` при канали, чрез които се реализира синхронизацията на комуникиращите процеси и унищожаването на канала.

Ако при `close` се освободи последния файлов дескриптор за писане в канала, се събуждат всички процеси, чакащи да четат от канала, като `read` връща 0 (това означава EOF). Ако при `close` се освободи последния файлов дескриптор за четене от канала, се събуждат всички процеси, чакащи да пишат в канала като им се изпраща сигнал `SIGPIPE`. Когато се освободи и последния файлов дескриптор за работа с канала, той се унищожава.

Пример. Програмният канал в един процес (Фиг. 7а) не е от голяма полза, но програмата показва как се създава и използва програмен канал в един процес.

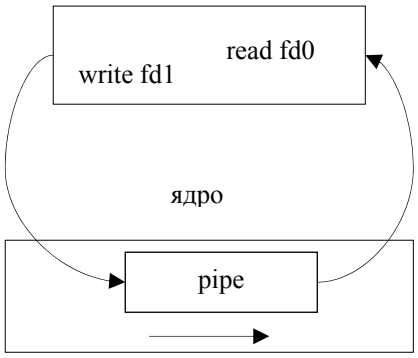
```
main()
{
  int pd[2], n;
  char buff[256];
  if (pipe(pd) < 0) {
```

```
      perror("pipe error. ");
      exit(1); }
  printf("read fd=%d, write fd=%d\n", pd[0], pd[1]);
  if ( write(pd[1], "Hello World\n", 12) != 12 ) {
      perror("write error. ");
      exit(1); }
  if ( ( n = read(pd[0], buff, sizeof(buff)) ) <= 0 )
      perror("read error. ");
      exit(1); }
  write(1, buff, n);
  exit(0);
}
```

Изходът от тази програма е:

```
Hello Word
read fd=3, write fd=4
```

Забележете, че изходът от `write` е преди този от `printf` въпреки, че в програмата са в обратен ред. Причината е, че изходът от `printf` се буферира и не се извежда на стандартния изход докато не се напълни буфера или процесът не завърши. Изходът при системен примитив `write` не се буферира.

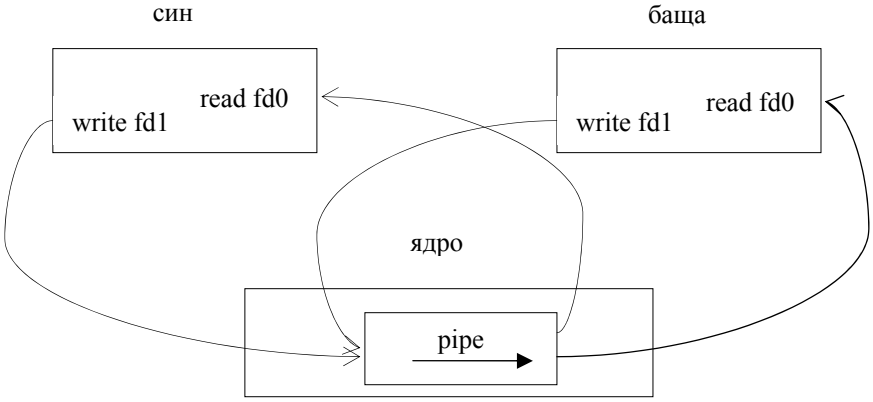


Фиг. 7а. Програмен канал в един процес

Обикновено програмен канал се използва за комуникация между два процеса. Последователността от стъпки е следната:

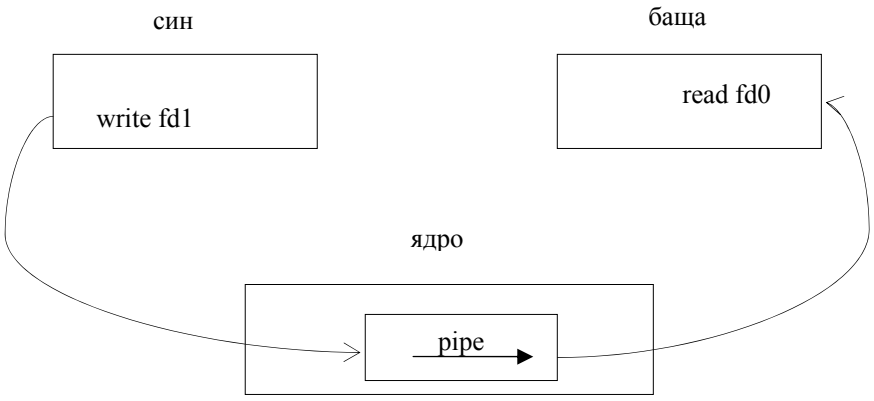
- процесът създава програмен канал - изпълнява `pipe`
- процесът създава нов процес - изпълнява `fork`

Резултатът от това е показан на следващата Фиг.7б, има два процеса - баща и син, като синът е наследил от баща си двата файлови дескриптора за програмния канал.



Фиг. 7б. Програмен канал между два процеса веднага след fork

Ако след това бащата затвори файловия дескриптор за писане в програмния канал, а синът затвори този за четене (може и обратното), ще се получи еднопосочен канал за предаване на данни между два процеса, показан на Фиг. 7в.



Фиг. 7в. Еднопосочна комуникация между два процеса с програмен канал

Ако е необходима двупосочна комуникация между два процеса, то трябва да се създадат два програмни канала, по един за всяко направление.

Пример. Програмата (`pipe1.c`) илюстрира механизма на програмните канали, като реализира конвейер на две програми, а по-точно "`ls | wc -l`".

```
#define READ 0
#define WRITE 1

main ()
{
int pd[2], status;
if ( fork() == 0 ) {        /* in child */
      pipe(pd);
      if ( fork() == 0) {   /* in grandchild  */
            close(1);
            dup(pd[WRITE]);
            close(pd[READ]);
            close(pd[WRITE]);
            execl("/bin/ls", "ls", 0);
            perror("Cannot exec ls. ");
            exit(1);
      } else {              /* in child  */
            close(0);
            dup(pd[READ]);
            close(pd[READ]);
            close(pd[WRITE]);
            execl("/usr/bin/wc", "wc", "-l", 0);
            perror("Cannot exec wc. ");
            exit(1);
      }
}
wait(&status);              /* in parent  */
printf("Parent after end of pipe: status=%d\n", status);
}
```

Процесът, в който работи за програмата `pipe1`, създава син, в който пуска програмата `wc`, който от своя страна също създава син за програмата `ls`. След това процесът `pipe1` чака завършването на своя процес-син (`wc`).

Пример. Друг вариант за реализация на конвейера "`ls | wc -l`" (`pipe2.c`).
Двата процеса, в които работят програмите `ls` и `wc`, са синове на процеса за програмата `pipe2`. Процесът `pipe2` чака завършването на първия си процес-син (`wc`).

```
#define READ 0
#define WRITE 1

main ()
```

```
{
int pd[2], status, pid;
pipe(pd);
pid = fork();
if ( pid == 0 ) {     /* in first child */
      close(1);
      dup(pd[WRITE]);
      close(pd[READ]);
      close(pd[WRITE]);
      execl("/bin/ls", "ls", 0);
      perror("Cannot exec ls. ");
      exit(1);
}
/* in parent  */
if ( pid == -1 ) {
      perror("Cannot fork first child. ");
      exit(1);
}
pid = fork();
if ( pid == 0 ) {     /* in second child */
      close(0);
      dup(pd[READ]);
      close(pd[READ]);
      close(pd[WRITE]);
      execl("/usr/bin/wc", "wc", "-l", 0);
      perror("Cannot exec wc. ");
      exit(1);
}
/* in parent  */
close(pd[READ]);
close(pd[WRITE]);
if ( pid == -1 ) {
      perror("Cannot fork second child. ");
      exit(1);
}
waitpid(pid, &status, 0);
printf("Parent after end of pipe: status=%d\n", status);
}
```

6. МЕЖДУПРОЦЕСНИ КОМУНИКАЦИИ

Сигналите и програмните канали в UNIX и LINUX системите са два от механизмите за комуникации между процеси, реализирани от най-ранните версии. Сега ще разгледаме проблемите при междупроцесни комуникации (Interprocess Communication) и някои други механизми за решаването им.

Най-простият начин, по който два или повече процеса могат да взаимодействат е да се конкурират за достъп до общ ресурс. Например, два процеса А и В 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 А	**Процес В**
1.Чете `counter` в локална променлива `pa`.	
	2.Чете `counter` в локална променлива `pb`.
	3. `pb = pb + 2`
	4. Записва `pb` в `counter`.
5. `pa = pa + 1`	
6. Записва `pa` 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 В 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 петте философа (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.

6.1. ВЗАИМНО ИЗКЛЮЧВАНЕ

Проблемът за избягване на състезанието е бил формулиран от Е. Дейкстра чрез термина ***критичен участък*** (***critical section***). Част от кода на процеса реализира вътрешни изчисления, които не могат да доведат да състезание, в друга част от кода си процесът осъществява достъп да обща памет или върши неща, които могат да доведат до състезание. Тази част от програмата ще наричаме критичен участък и ще казваме, че процес е в критичния си участък, ако е започнал и не е завършил изпълнението му, независимо от състоянието си. За избягване на състезанието и коректното взаимодействие на конкуриращите се процеси трябва да са 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 пръв предложи решение на тази задача, а по-късно бе предложено още едно решение от Питерсон. 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.

```
shared int turn=0;

P0()                                P1()
{                                   {
while (TRUE) {                      while (TRUE)
      while (turn != 0);                  while (turn != 1);
      critical_section0();                critical_section1();
      turn = 1;                           turn = 0;
      noncritical_section0();             noncritical_section1();
      }                                   }
}                                   }
```

Този алгоритъм използва една обща променлива `turn`, чието значение е номер на процес, който е на ред да влезе в критичния си участък. Двата процеса 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;

P0()                                P1()
{                                   {
while (TRUE) {                      while (TRUE) {
      wants0 = TRUE;                      wants1 = TRUE;
      while (wants1)                      while (wants0)
        if (turn == 1) {                    if (turn == 0) {
            wants0 = FALSE;                     wants1 = FALSE;
            while (turn == 1);                  while (turn == 0);
            wants0 = TRUE; }                wants1 = TRUE; }
      critical_section0();                critical_section1();
      turn = 1;                           turn = 0;
      wants0 = FALSE;                     wants1 = FALSE;
      noncritical_section0();             noncritical_section1();
      }                                   }
}                                   }
```

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;
}
```

```
P0()                                P1()
{                                   {
while (TRUE) {                      while (TRUE) {
      enter_region(0);              enter_region(1);
      critical_section0();                critical_section1();
      leave_region(0);              leave_region(1);
      noncritical_section0();             noncritical_section1();
      }                                   }
}                                   }
```

Основният недостатък и на двата алгоритма 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, но е по-сложен.

6.2. 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. 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;
}
```

Съществото за двете операции е, че са неделими (атомарни), т.е. никой процес, изпълняващ V не може да бъде блокиран и докато един процес изпълнява операция над семафор `s` друг процес не може да започне операция над `s` докато не завърши първата. 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 0 и 1.

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

P0()                        . . .       Pn()
{                                       {
while (TRUE) {                          while (TRUE) {
      . . .                                   . . .
      P(mutex);                               P(mutex);
      critical_section0();                    critical_section1();
      V(mutex);                               V(mutex);
      . . .                                   . . .
      }                                       }
}                                       }
```

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;

P1()                                    P2()
{                                       {
   . . .                                   . . .
   S1;                                     P(s);
   V(s);                                   S2;
   . . .                                   . . .
}                                       }
```

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. Преди разгледахме решаването на този проблем чрез програмен канал, който се използва и от командните интерпретатори в UNIX и LINUX системите. Сега 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. Производителят записва в буфера всеки произведен от него елемент, а потребителят чете от буфера елементите, за да ги обработи. Двата процеса работят едновременно и 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 буфера. 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 производителя ще се блокира когато няма място в буфера. Семафорът `full` ще брои запълнените места в буфера и по него потребителя ще се блокира когато в буфера няма данни.

```
#define N 100
#define TRUE 1

shared buffer buf;
semaphore mutex=1, empty=N, full=0;

producer()                          consumer()
{                                   {
int item;                           int item;
while (TRUE) {                      while (TRUE) {
      produce_item(&item);                P(full);
      P(empty);                           P(mutex);
      P(mutex);                           remove_item(&item);
      enter_item(&item);                  V(mutex);
      V(mutex);                           V(empty);
      V(full);                            consume_item(item);
      }                                   }
}                                   }
```

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 петте философа

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
#define TRUE 1

philosopher(int i)
{
while(TRUE) {
   think();
   take_fork(i); /* взима лявата вилица,като чака докато тя
                    стане достъпна*/
   take_fork((i+1)%N); /*взима дясната вилица, като също чака*/
   eat();
   put_fork(i);         /* връща лявата вилица */
   put_fork((i+1)%N);   /* връща дясната вилица */
   }
}
```

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
```

```
#define RIGHT (i +1) % N
#define THINKING 0
#define HUNGRY 1
#define EATING 2

shared int state[N] ]={0,0,0,0,0};
semaphore mutex=1;
semaphore s[N]={0,0,0,0,0};

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.

6.3. 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.

6.3.1 АДРЕСИРАНЕ НА СЪОБЩЕНИЯТА

Ще разгледаме първите четири от поставените въпроси, отговора 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.

Процес P	**Процес Q**
`send(Q, message);`	`receive(P, message);`

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.

Косвена комуникация

Въвежда се нов обект, наричан пощенска кутия (mailbox), порт (port) или опашка на съобщенията (message queue). Съощенията 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	**Процес Q**
`send(A, message);`	`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.

Освен примитивите `send` и `receive` трябва да има и примитив за създаване на пощенска кутия. Освен това възниква и въпросът за собствеността на пощенската кутия и за правата на процесите да изпращат или 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 й. Друга възможност е процесите да могат да ползват обща пощенска кутия чрез механизма за създаване на процеси и когато последния процес, ползващ определена пощенска кутия, завърши тя да се унищожава автоматично от системата.

6.3.2. 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 чака 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 чакат в комуникационната връзка. 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 да прилагат протокол на потвърждаване.

Процес P (изпращач)	**Процес Q (получател)**
`send(Q, message);`	`receive(P, message);`
`receive(Q, message);`	`send(P, "acknowledgement");`

За комуникация между процеси в UNIX, LINUX и MINIX системите може да се използва програмен канал. Този механизъм 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.

Освен програмните канали в тези системи е реализиран и механизъм на съобщенията. В MINIX механизмът 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. 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]; }
```

Обект, в който се изпращат и от който се получават съобщения, се нарича опашка на съобщенията (message queue). Всяка опашка на съобщенията се създава явно чрез системен примитив. 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. 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 да се получават съобщения от опашката, а правото `w` - да се изпращат съобщения в опашката. Една опашка на съобщенията съществува докато не се унищожави явно чрез системен примитив и само собственикът има право за това.

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);}
}
}
```

6.4. НИШКИ

Нишките (***threads***) са сравнително нова абстракция в операционните системи. Понякога ги наричат ***lightweight processes*** или ***подпроцеси*** 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 UNIX процес, така както го разглеждахме до сега. Той включва програмен код, различни ресурси, като променливи, файлови дескриптори, текущ каталог, управляващ терминал и др. и досега един набор от машинни регистри, т.е. един регистър PC. Това означава, че всеки процес има една последователност на управление (flow of control) или накратко е последователен процес.

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 се нарича още ***multithreaded process***. Всяка последователност на управление в рамките на процес се нарича нишка, т.е.тя е част от процес, която има собствен набор от регистри и стек. Всички други ресурси – променливи, файлови дескриптори, текущ каталог и др. принадлежат на процеса.

Поддържането на нишки дава възможност да се пишат конкурентни приложения, които могат да работят по-ефективно като в еднопроцесорна така и в многопроцесорна среда.

По-нататък ще разгледаме по-подробно нишките по стандарта POSIX 1003.1c, известни още като pthreads, и реализирани в съвременните версии на UNIX и LINUX системите. Функциите за работа с нишки осигуряват:

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

6.4.1. 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 й 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 й, 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 параметрите на планиране.

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 кода 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);
}
```

6.4.2. 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 създала.

`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 й 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 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);
}
```

7. 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 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.

7.1. НЕОБХОДИМИ УСЛОВИЯ ЗА ДЕДЛОК

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

1. Взаимно изключване (Mutual exclusion)

В системата има поне един ресурс, който трябва да се използва монополно, т.е. достъпен е или е предоставен точно на 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 $\{ p_1, p_2, \ldots, p_k\}$, 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.

7.2. ГРАФ НА РАЗПРЕДЕЛЕНИЕ НА РЕСУРСИТЕ

Състоянието дедлок в системата може формално да бъде описано чрез ориентиран граф $G = \{V, E\}$. Множеството на върховете V е обединение на две непресичащи се подмножества:

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

$\{ r_1, r_2, \ldots, r_m\}$ - 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)

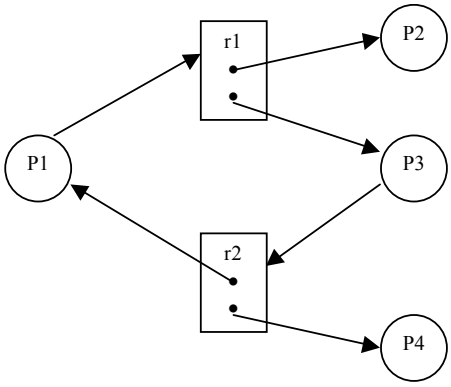
(r_j, p_i) – означава, че един екземпляр от ресурс r_j е предоставен на процес p_i (assignment edge).

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) си изключва от графа. Така графът представя разпределението на ресурсите и чакащите заявки във всеки един момент.

Може ли по графа 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 това е равносилно на четвъртото от условията на Кофман.

Следващият пример 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.



7.3. ПРЕДОТВРАТЯВАНЕ 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 можем да осигурим неизпълнението на поне едно от тези условия, тогава дедлок няма да настъпи. Най-известните методи за предотвратяване са стратегиите на Хавендер.

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, 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 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 и след това 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 $R = \{ r_1, r_2, \ldots, r_m \}$ 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 $F(r_i) \geq F(r_j)$.

7.4. 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 удоволетворяването й 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 (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 $<p_1, p_2, \ldots, p_n>$, за която е изпълнено следното: нуждите на всеки процес p_i могат да се удоволетворят от свободните в момента ресурси и ресурсите държани от процесите p_j, където $j<i$. Следователно, съществува някаква последователност от други състояния, в която системата може да удоволетвори 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 бройки. Това е пример за надеждно състояние, защото съществува последователността $<p_2, p_1, p_3>$, 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 удоволетворява само ако новото състояние е надеждно. В противен случай процесът трябва да чака.

8. ПЛАНИРАНЕ НА ПРОЦЕСИ

8.1. НИВА НА ПЛАНИРАНЕ

8.2. ДИСЦИПЛИНИ ЗА ПЛАНИРАНЕ