**12. Управление на процеси и междупроцесни комуникации.   
Основни системни примитиви за управление на процеси. Създаване на процес.   
Изпълнение на програма. Завършване на процес. Синхронизация със завършването на   
процеса-син. Права на процеси – потребителски идентификатори на процес. Групи   
процеси и сесия. Механизми за междупроцесни комуникации. Сигнали. Програмни   
канали. IPC пакет на UNIX System V: Обща памет. Семафори. Съобщения.**

**Някои неща може би са разгледани по-подробно, отколкото ще иска**

Всяка операционна система (в тесен смисъл на това понятие, т.е. ядрото)  
осигурява определен набор от операции за изпълняваните програми. Тези операции  
обикновено се наричат системни извиквания, system calls, системни примитиви,  
системни функции. Тук ще разгледаме операциите, реализирани в различните версии  
на Unix и Linux системите, които са включени в стандартите POSIX. Ще започнем с  
кратък преглед на основните понятия, които по-нататък ще бъдат по-подробно  
разгледани от гледната точка на програмиста.

Програма е файл, съдържащ изпълним код и съхраняван на диска като  
обикновен файл. След като програмата бъде заредена в паметта и започне нейното  
изпълнение под управлението на ядрото, започва животът на процеса. Процес е  
програма в хода на нейното изпълнение. В някои ОС се използва и термина задача  
(task).  
Всеки процес има уникален идентификатор, който се нарича process ID (pid)  
Идентификаторът на процес е неотрицателно цяло число и ОС осигурява неговата  
неизменност през целия живот на процеса.  
ОС осигурява четири основни системни примитива за управление на процесите  
- fork, exec, exit и wait. Нов процес се създава само чрез fork. Процесът, извикващ  
fork, се нарича процес-баща, а новосъздаденият е процес-син. Чрез exec процес може  
да извика за изпълнение нова програма. Процес завършва своя живот когато изпълни  
exit. Чрез wait процес-баща може да изчака завършването на свой процес-син.

**Системен примитив fork**

Единственият начин за създаване на нов процес от ядрото е когато съществуващ  
процес извика примитива fork. Новосъздаденият процес се нарича процес-син, а  
процесът, извикал fork, е процес-баща.

#include <sys/types.h>  
#include <unistd.h>  
pid\_t fork(void);  
Връща 0 в процеса-син, pid на сина в процеса-баща, -1 при грешка

Сложността при този примитив се състои в това, че един процес “влиза” във функцията  
fork, а два процеса “излизат” от fork с различни връщани значения: в процеса-баща  
функцията връща pid на процеса-син при успех или -1 при грешка, а в сина връща 0.  
Новият процес представлява почти точно копие на процеса-баща. Той изпълнява  
програмата на бащата и дори от мястото, до което е стигнал той, т.е. процесът-син  
започва изпълнението на потребителската програма от оператора след fork.

/\* Example of fork \*/

#include "ourhdr.h"

main(void)

{

pid\_t pid;

if ((pid = fork()) < 0)

err\_sys\_exit("fork error");

else if (pid == 0) /\* in child \*/

printf("PID %d: Child started, parent is %d.\n",

getpid(), /\* Child PID \*/

getppid()); /\* Parent PID \*/

else { /\* in parent \*/

printf("PID %d: Started child PID %d.\n",

getpid(), /\* Current parent PID \*/

pid); /\* Child's PID \*/

sleep(3); /\* Wait 3 seconds \*/

}

exit(0);

}

/\*

След fork и двата процеса работят асинхронно и не можем да разчитаме кой ще работи  
първи и кой втори. Затова процесът-баща извиква библиотечната функция sleep (която  
блокира процеса за 3 секунди), за да даде възможност на сина да го види и завърши.  
Друг вариант е и двата процеса да извикат sleep. Това също ни осигурява тяхното  
съществуване достатъчно дълго, така че да могат да се видят един друг. При  
изпълнението на програмата получихме следния изход.

$ a.out  
PID 24713: Child started, parent is 24712.  
PID 24712: Started child PID 24713.

Друго, което процес-син наследява от бащата, са отворените файлове. Процесът-  
син получава копие на файловите дескриптори, които бащата е отворил преди да  
изпълни fork.

И така процес-син наследява от бащата следните атрибути и елементи от

контекста си:

- идентификатор на група процеси

- идентификатор на сесия

- реален потребителски идентификатор

- ефективен потребителски идентификатор

- реален идентификатор на потребителска група

- ефективен идентификатор на потребителска група

- файлови дескриптори на отворените файлове

- текущ каталог

- управляващ терминал

- маска, използвана при създаване на файлове

- реакция на процеса при получаване на различни сигнали

- променливи от обкръжението

Това, по което се отличават процесите син и баща, е:

- значението, върнато от функцията fork в двата процеса е различно

- идентификатор на процеса

- идентификатор на процеса-баща

- времената за изпълнение на процеса-син в системна и потребителска фаза са 0

Има два основни начина за използване на fork.

1. Процес иска да създаде свое копие, което да изпълнява една операция, докато

другото копие изпълнява друга. Този модел се използва при процеси сървери.

2. 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. Този начин се използва от командните интерпретатори при

изпълнение на външни команди.

**Функции на системния примитив exec**

Когато с fork се създава нов процес, той е копие на баща си, т.е. продължава да

изпълнява същата програма. Чрез примитива exec всеки процес може да извика за

изпълнение друга програма по всяко време от своя живот, както веднага след

създаването си така и по-късно, дори няколко пъти в живота си. 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.

При успех, когато процесът се върне от exec в потребителска фаза, той

изпълнява кода на новата програма, започвайки от функцията main, но това си остава

същия процес. Не е променен идентификатора му, позицията му в йерархията на

процесите дори голяма част от елeменти на контекста му. 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.

**5.3. Завършване на процес - exit и \_exit**

Има няколко начина за завършване на процес:

1. Нормално завършване:

- при извикване на exit или \_exit

- при изпълнение на return от функцията main, което е еквивалентно на

извикването на exit

2. Аварийно завършване:

- при получаване на сигнал, за който реакцията на процеса е завършване на

процеса

- при извикване на функцията abort (В този случай на процеса се изпраща

сигнала SIGABRT.)

Обикновено \_exit се реализира като системен примитив и предизвиква незабавен вход  
в ядрото, а exit е библиотечна функция. Тя осигурява „чисто” завършване на процеса.  
Грижи се да запише във файл буферите, използвани от функциите на стандартната В/И  
библиотека (напр., printf, fwrite), т.е. изпълнява fclose за всички незатворени  
потоци. Освен това извиква всички функции, регистрирани преди това с функцията  
atexit, наричани обработчици при завършване или exit handlers. След това извиква  
примитива \_exit. Функциите exit и atexit са включени в ANSI C стандарта.

#include <stdlib.h>

int atexit(void (\*function)(void));

Връща 0 при успех, -1 при грешка

Аргументът function е адрес на функция, която ще бъде изпълнена при exit. Ако са

регистрирани няколко функции, то те се изпълняват в ред обратен на реда на

регистрацията им.

Независимо от начина, по който процесът завършва, накрая управлението се

предава на ядрото. Там се освобождават почти всички елементи от контекста на

процеса и от него остава само записа в таблицата на процесите. Функциите exit и

\_exit не връщат нищо, защото няма връщане от тях, винаги завършват успешно и след

тях процесът почти не съществува, т.е. той става зомби. Значението на аргумента е кода

на завършване на процеса, който се съхранява в таблицата на процесите. При аварийно

завършване на процес, ядрото генерира код на завършване, който съобщава причината

за аварийното завършване. Кодът на завършване е предназначен за процеса-баща и му

се предава, когато той се заинтересува като извика примитив wait.

**5. 4. Системни примитиви wait**

Процес-баща може да разбере как е завършил негов процес-син, чрез примитив

wait. Ако синът още не е завършил, то процесът-баща бива блокиран, докато синът не

изпълни exit, т.е. изчаква неговото завършване.

#include <sys/types.h>

#include <sys/wait.h>

pid\_t wait(int \*status);

pid\_t waitpid(pid\_t pid, int \*status, int options);

Връща pid на процеса при успех, -1 при грешка.

Функциите на системния примитив връщат pid на завършилия син, а чрез аргумента

status информация за начина, по който е завършил, т.е. кода на завършване.

Различията между двете функции са следните:

1. Функцията wait чака първия син, който завърши. Функцията waitpid има

аргумент pid, чрез който може да се чака завършването на определен син.

Интерпретацията на аргумента pid е следната:

pid == -1 Чака първия син, който завърши.

pid > 0 Чака син с идентификатор pid.

pid == 0 Чака първия син от същата група процеси.

pid < -1 Чака първия син от група процеси с идентификатор |pid|.

2. Функцията wait винаги блокира процеса, ако синът не е завършил. Чрез

аргумента options на waitpid може да се предотврати блокирането на

процеса. При значение WNOHANG, процесът-баща не се блокира ако синът не е

завършил, а функцията връща 0.

От значението, върнато в аргумента status на wait или waitpid, процес-баща

може да разбере:

- Как е завършил сина - нормално или аварийно?

- Какъв е кода на завършване, предаден в exit, или номера на сигнала, който е

предизвикал аварийното му завършване?

- Създаден ли е файл с име core (някои сигнали предизвикват създаването му)?

В заглавния файл <sys/wait.h> са определени няколко макроса, чрез които бащата

може да определи как е завършил сина му.

Процес-баща е отговорен за своите процеси-синове. Когато процес завърши се очаква неговият баща да се осведоми за завършването му с wait. При изпълнение на wait от процес-баща се изчиства сина-зомби от системата, т.е. освобождава се записа му от таблицата на процесите. Това означава, че всеки завършил процес остава в системата в състояние зомби докато баща му не изпълни wait. Но не бива да се задължава процес-баща да изпълнява wait, например той може да завърши веднага след като е създал син. Затова когато процес завършва неговите синове се осиновяват от процеса init, който се грижи за изчистване на системата от зомбита-сираци (така в системата няма да останат вечни зомбита).

Но какво точно означава израза „трябва да имаме права„? Когато процес извика системен примитив за отваряне, създаване, изтриване на файл и т.н. ядрото проверява правата му относно файла. При тази проверка се използват атрибутите на файла – st\_uid, st\_gid, st\_mode и атрибутите на процеса. С всеки процес са свързани четири

потребителски идентификатори:

- реален потребителски идентификатор (ruid)

- ефективен потребителски идентификатор (euid)

- реален идентификатор на потребителска група (rgid)

- ефективен идентификатор на потребителска група (egid)

Реалните идентификатори определят потребителя, който е стартирал процеса (в чиято  
сесия се изпълнява процеса). По ефективните идентификатори се определят правата на

процеса. И така проверката, която ядрото изпълнява следва следните четири стъпки.

1. Ако euid на процеса е 0 (root), то достъпът се разрешава.

2. Ако euid на процеса е еднакъв с този на собственика на файла (st\_uid), то:

- ако в кода на защита на файла битовете за съответния тип достъп за собственика са вдигнати, достъпът се разрешава

- иначе не се разрешава.

3. Ако egid на процеса е еднакъв с този на групата на файла (st\_gid), то:  
- ако в кода на защита на файла битовете за съответния тип достъп за групата са вдигнати, достъпът се разрешава

- иначе не се разрешава.

4. Ако в кода на защита на файла битовете за съответния тип достъп за другите са  
вдигнати, достъпът се разрешава, иначе не се разрешава.

**5.7. Групи процеси и сесия**

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.

**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.

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 от сигнала.

**7.1. Програмни канали**

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

• неименован програмен канал (pipe) - за комуникация между родствени процеси  
• именован програмен канал (named pipe или FIFO файл) - за комуникация  
между независими процеси. Като механизъм за комуникация те са еднакви. Реализират се като тип файл, който се различава от обикновените файлове и има следните особености:

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

Програмен канал се създава чрез примитива pipe. 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 достъпът 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. Това означава, че има някои особености в алгоритъма на примитивите write и read, когато първият аргумент е файлов дескриптор на програмен канал.

**7.3. Съобщения**

Един от методите за комуникация между процеси е чрез съобщения (Message passing). Процесите взаимодействат като си предават съобщения - един процес изпраща едно съобщение, а друг го получава. Съществуват различни логически модели на съобщения. Съобщенията в IPC пакета на UNIX System V са с: - Косвена адресация Съобщения се предават в и получават от опашка на съобщенията (Message queue ще я съкращаваме на MQ). - Автоматично буфериране Съществува системен буфер с ограничен капацитет, в който временно могат да се съхраняват изпратени и още неполучени съобщения.

**7.4. Обща памет**

Общата памет е най-бързият и прост метод за комуникция между процеси. Процесите взаимодействат като осъществяват достъп до една и съща област в паметта. Методът е бърз защото не изисква системни примитиви за предаване на данни, т.е. няма вход в ядрото и копиране на данни между потребителския процес и ядрото. Четенето и писането в общата памет е толкова бързо, колкото и достъпа до всяка една променлива на процеса.

Проблемът при използването на обща памет за комуникация между процеси (демонстриран в Пример 7.11) е, че ядрото не синхронизира достъпа на процесите до общата памет, което води до състезания или други проблеми. Затова при метода за комуникация с обща памет трябва да се използва допълнителен механизъм за синхронизация. Такъв механизъм може да са семафорите.

**7.5. Семафори**

Семафорите са предложени от Дейкстра като механизам за осигуряване на взаимно изключване и синхронизация на процеси, използващи общи ресурси. Семафорите от IPC пакета на UNIX System V са по-сложни от тези на Дейкстра и имат следните особености:

- Семафорите се създават на масиви от един или повече семафора. Всеки елемент на масива има собствен брояч, който може да приема цели неотрицателни значения.

- Всеки масив семафори (понякога ще го наричаме за простота семафор) има ключ и идентификатор и се представя в ядрото чрез структура semid\_ds, определена в заглавния файл <sys/sem.h>.

- Една операция може да се изпълни върху няколко елемента на масива семафори, т.е. да включва проверка и изменение на няколко брояча в масива семафори, като ядрото гарантира атомарността на цялата операция.

- Съществуват ограничения при всяка конкретна реализацията на този механизъм