ТЕМА 2. ПРОЦЕСИ В ОПЕРАЦИОННАТА СИСТЕМА UNIX

Понятие за процес в UNIX. Контекст. Идентификация на процеса. Състояние на процеса. Кратка диаграма на състоянията. Йерархия на процеса. Системни извиквания getpid(), getppid(). Създаване на процес в UNIX. Системно извикване fork(). Прекратяване на процес. Функция exit(). Параметри на функцията main() в езика C. Променливи на средата и аргументи на командния ред. Изменение на потребителския контекст на процеса. Функции за системни извиквания exec().

1. Процеси в UNIX и Linux

Под процес се разбира програма по време на изпълнение, за която има отделени ресурси на компютърната система.

Съвременните компютърни системи са способни да изпълняват няколко програми едновременно. Този паралелизъм е наречен (не много удачно) мултипрограмиране (multiprogramming). Целата на мултипрограмирането е по-ефективното използване на ресурсите на компютърната система. При мултипрограмирането в оперативната памет са заредени няколко програми, а централния процесор (ЦП) във всеки един момент изпълнява само една от тях. Тъй като, ЦП превключва много бързо от изпълнение на една програма към друга у потребителя се създава впечатление за едновременното изпълнение на няколко програми. Този псевдо паралелизъм се осъществява като се съхранява специфична информация за всяка от работещите програми, така че след определен период да се възстанови изпълнението на прекъсната програма. За да се изолира потребителя от детайлите по поддържането на, този псевдо паралелизъм е необходима концепция, която да опрости работата на потребителя с операционната система. Именно такава е концепцията на процесите, която обозначава дейността по изпълнение на програма(и) за определен потребител. В различни операционни системи се използват различни понятия: задание (job), задача (task), процес (process). Терминът процес за първи път е бил използван в операционната система MULTICS на MIT през 60-те години и преобладава в съвременните операционни системи. Най-краткото определение за процес е програма в хода на

нейното изпълнение. Следователно, има връзка между понятията процес и програма, но те не са идентични. За разлика от програмата, която е нещо статично - файл записан на диска и съдържащ изпълним код, процесът е дейност. В понятието процес освен инструкциите на 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 една такава концепция, цялото програмно осигуряване, работещо на компютъра е организирано 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 и Linux са типични многопроцесни системи.

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

2. Йерархия на процесите

Всички процеси в ОС UNIX, освен един, който се създава при стартиране на ОС, се пораждат единствено от други процеси. В качеството на прародител. В ОС Linux е процеса *kernel* с идентификатор *0*.

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-родител, а новият е процес-дете. След *fork* процесът-родител продължава изпълнението си паралелно с новия процес-дете. Следователно, след това процесът-родител може да създаде с *fork* и други процеси-деца, т.е. един процес може едновременно да има няколко процеса-деца. Процесът-дете също може да изпълни *fork* и да създаде свой процес-дете. Следователно, всички процеси в UNIX са свързани в отношение процес-родител – процес-дете в йерархична схема наречена родословно дърво на процесите.

Всеки процес получава уникален идентификационен номер – *PID* (process identificator). Чрез него процеса се идентифицира в операционната система. Освен, че е уникален за процеса този номер е и неизменен през целия му живот. При създаването на нов процес операционната система се опитва да му присвои свободен номер, който е по-голям от този на процеса създаден преди него. Ако липсва такъв номер (например, при достигане на максималния възможен номер за присвояване на процеси), то ОС избира най-малкия номер от всички свободни номера. В ОС Linux идентификационните номера на процесите започват от *0*, който се присвоява на процеса *kernel* при стартиране на ОС. В последствие този номер не може да бъде присвоен на никой друг процес. Най-голямата стойност на номер на процес в Linux за 32-разредни процесори на Intel е $2^{31} - 1$.

При стартиране на UNIX или Linux се създават няколко важни процеса. Първият от тях не може да бъде създаден нормално, тъй като преди него няма друг процес. Програмата за начално зареждане, която е в *boot* блока на твърдия диск, 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 процеса, които се наричат системни процеси (kernel processes), и самия той става системен процес (това важи за някои версии).

Процесът *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 когато той постъпи приема, че потребител иска вход в системата. Тогава програмата *getty* в процеса се сменя с

програмата *login*, която извършва идентификация на потребителя и при успех се сменя с програмата *shell* за съответния потребител (или в някои версии, поражда нов процес за програмата *shell*). 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 процеса *init* приключи с инициализацията на процесите според описанието в */etc/inittab*, той изпълнява безкраен цикъл, в който чака завършване на своя процес-дете и изпълнява довършителни дейности, като съхранване на целостта на родословното дърво на процесите, тъй като единствено процеса *init* може да стане родидел на процес-сирак (процес, чийто процес-родител е завършил своята работа).

Времето на съществуване на процеса *init* се определя от времето на функциониране на ОС.

3. Състояния на процесите

В многопроцесните операционни системи едновремено съществуват много процеси, а ЦП е един и във всеки един момент може да изпълнява само един от тези процеси. Този процес се намира в състояние изпълнение (running). Останалите процеси са в някакво друго състояние.

Процесите в UNIX могат да се намират в едно от състояния представени на фигура 2.1.

1. Изпълнение в потребителски режим (user running). Процесът се изпълнява в потребителски режим, като ЦП изпълнява команди от потребителската програма свързана с процеса.

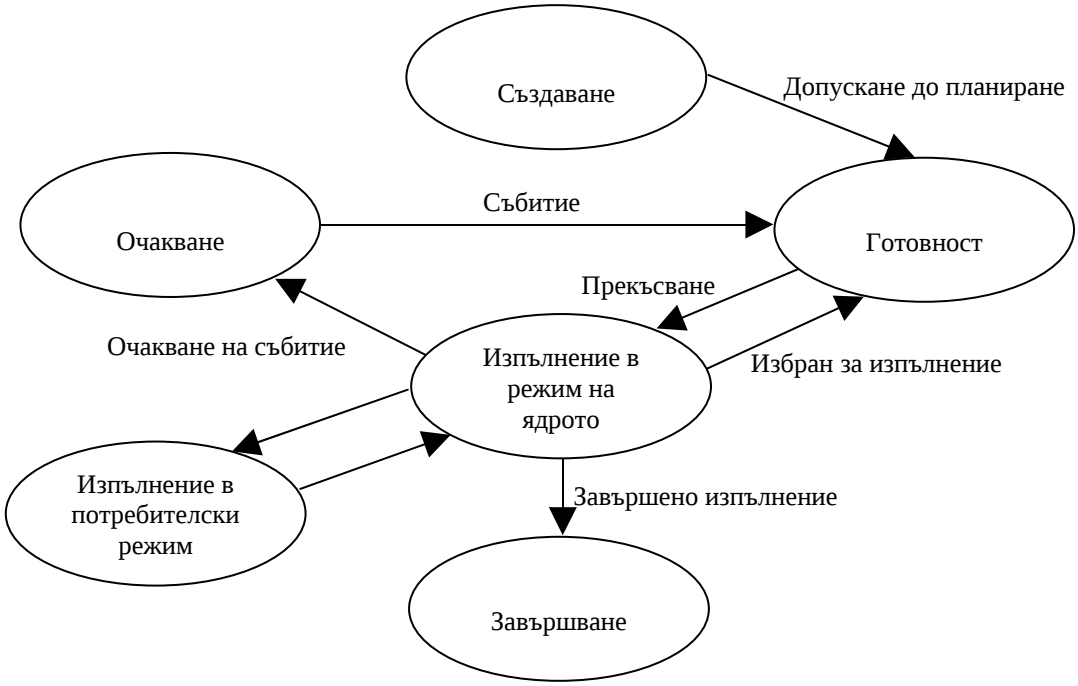
2. Изпълнение в режим на ядрото (kernel running). Процесът се изпълнява в режим ядро, като ЦП изпълнява команди от ядрото, т.е. от името на процеса работят модули на ядрото, които извършват системна работа.

3. Готовност (ready in memory). Процесът е готов за изпълнение и се намира в паметта.

4. Очакване (blocked in memory). Процесът чака настъпването на някакво събитие и се намира в паметта.

5. Създаване (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.

6. Завършване (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.



Фиг. 2.1. Диаграма на състоянията на процесите в UNIX

Както се вижда от фиг. 2.1 състоянието на изпълнение е разделено на две: изпълнение в режим на ядрото и изпълнение в потребителски режим. Когато процеса е в състояние на изпълнение в потребителски режим, се изпълняват потребителки инструкции. При състоянието на изпълнение в режим на ядрото се изпълняват инструкции на ядрото на ОС в контекста на текущия процес (например, при обработка на системно извикване или прекъсване). Процесът не може директно да премине от режим на изпълнение в потребителски режим в състоянията очакване, готовност или

завършване на изпълнението. Такива преходи са възможни само през междинното състояние "Изпълнение в режим на ядрото". Също така е забранен прекия преход от състояние "Готовност" в състояние "Изпълнение в потребителски режим".

Всеки процес влиза в системата в състояние "Създаване", когато процес-баща изпълни *fork*. След това преминава в състояние "Готовност". След време планировчикът ще го избере за текущ и той ще влезе в състояние "Изпълнение в режим на ядрото", където ще довърши своята част от *fork*. След това може да премине в състояние "Изпълнение в потребителски режим", 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 "Изпълнение в режим на ядрото". Когато завърши обработката на прекъсването, ако това е била причината за вход в състоянието, ядрото може да реши да върне процеса обратно в състояние "Изпълнение в потребителски режим". Ако причината за прехода от потребителски към системен режим 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 "Очакване ". Когато входно-изходната операция завърши, устройството ще предизвиква прекъсване и някой друг процес, който в момента е в състояние "Изпълнение в потребителски режим", ще влезе в състояние "Изпълнение в режим на ядрото", а модулът обработващ прекъсването ще смени състоянието на първия процес от "Очакване" в "Готовност ".

Когато процес завършва той изпълнява системния примитив *exit* и състоянието му се сменя от "Изпълнение в потребителски режим" в "изпълнение в режим на ядрото". Когато системното изпълнеие на *exit* завърши състоянието се сменя в "Завършване". В това състояние процесът остава докато неговият процес-родител не се погрижи, чрез използване на специален системен примитив *wait*, след което процесът напълно изчезва от системата.

4. Контекст на процес

Съхраняваната в оперативнта памет информация за един процес се

нарича контекст на процеса. Контекста на процес съдържа програмния код и данните, използвани от процеса в потребителски режим на изпълнение (потребителски контекст) и в режим на изпълнение на ядрото (контекст на ядрото).

Потребителския контекст се създава от изпълним (програмен) файл и се разполага в адресното пространство на процеса. Състои се от:

• Изпълним код (text) - съдържа машинните команди, изпълнявани от процеса в потребителски режим. Зарежда се от изпълнимия файл.

• Данни (data) - съдържа глобалните данни, с които процеса работи в потребителски режим. Инициализираните данни (константи и променливи на които се присвояват начални стойности на етапа на компилацията) се зареждат от изпълнимия файл. За неинициализираните данни (статични променливи, на които не са присвоени начални стойности на етапа на компилацията) в изпълнимия файл се съхранява размера им и при създаване на потребителския контекст на процеса се отделя съответния обем памет. Съществуват и данни, разположени в динамично заделена памет (например, с помощта на стандартните библиотечни функции на C *malloc()*, *calloc()*, *realloc()*).

• Потребителски стек (stack) - създава се автоматично с размер определян от ядрото. Чрез него се реализира обръщението към потребителски функции. Той представлява стек от слоеве, като има по един слой за всяка извикана функция, която още не е изпълнила *return*. Всеки слой съдържа данни, локални за функцията и достатъчно данни за връщане от функция.

Изпълнимия код и инициализираните данни съставляват съдържанието на програмния файл, който се изпълнява в контекста на процеса. Тази част от контекста е статична и не се променя по време на работата на процеса. Потребителския стек, неинициализираните данни и данните в динамично заделената памет се променя при работа на процеса в потребителски режим (user mode).

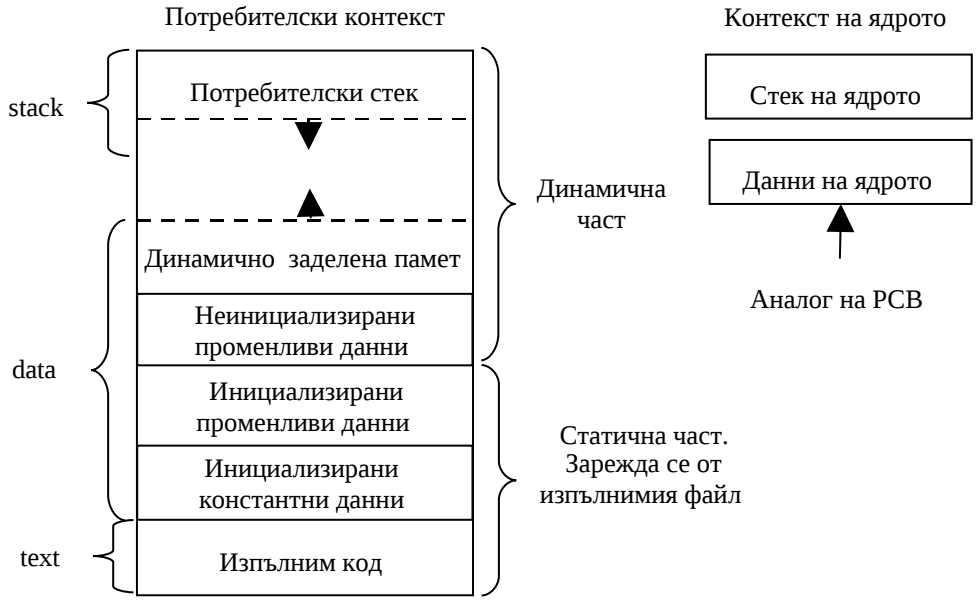
Когато процес работи в режим на изпълнение на ядрото е необходим стек на ядрото. Всеки процес трябва да има свой собствен стек на ядрото, който да съответства на неговото обръщение към ядрото. Освен това,

когато процес преминава от потребителски режим в режим изпълнеие на ядрото 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 на ядрото обединява системнния контекст (стек на ядрото и данни на ядрото) и регистровия контекст (област за съхранение на регистрите на ЦП).

• Стек на ядрото - използва се при работа на процеса в режим на ядрото (kernel mode) и има същата структура като потребителския стек, но в него се записват данни при обръщение към функции от ядрото.

• Данни на ядрото – съхраняват се в структура, която се явява аналог на блока за управление на процеса — *PCB* (Process Control Block). Данните на ядрото включват: идентификатор на потребителя — *UID* (User Identificator), идентификатор на групата на потребителя — *GID* (Group Identificator), идентификатор на процеса — *PID* (Process Identificator), идентификатор на родителския процес — *PPID* (Parent Process Identificator).

Фиг. 2.2 илюстрира компонентите, съставлявщи контекста на процес.



Фиг. 2.2. Контекст на процеса в UNIX

5. Системни извиквания getppid() и getpid()

Данните за един процес, които се намират в контекста на ядрото не могат да бъдат четени непосредствено от процеса. За да получи информацията от тези данни процесът трябва да направи съответните системни извиквания. Стойността на идентификатора на текущия процес може да бъде получена с помощта на системното извикване *getpid()*, а стойността на идентификатора на родителския процес – с помощта на системното извикване *getppid()*. Прототипите на тези системни извиквания и съответните типове данни са описани във файловете *<sys/types.h>* и *<unistd.h>*. Системните извиквания нямат параметри. Те връщат идентификатора на текущия процес и на родителския процес.

ИМЕ

```
getpid и getppid
```

ПРОТОТИП

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

ОПИСАНИЕ

```
Системното извикване getpid връща като резултат идентификатора на текущия
процес.
Системното извикване getppid връща като резултат идентификатора на процеса-
родител на текущия процес.
Типът данни pid_t се явява синоним на един от целочислените типове данни в
езика C.
```

Задача 1: Напишете самостоятелно програма, отпечатваща стойностите на *PID* и *PPID* за текущия процес, като използвате системните извиквания getpid() и getppid(). Стартирайте я няколко пъти поред. Разгледайте как се изменя идентификатора на текущия процес. Обяснете наблюдаваните промени.

6. Създаване на процеси в UNIX

Както бе изяснено в т. 2 в ОС UNIX нов процес може да бъде породен по един единствен начин – с помощта на системното извикване *fork()*. При това новосъздадения процес е копие на процеса родител. В породения процес се изменят следните параметри:

• идентификатор на процеса – *PID*;

• идентификатор на родителския процес – *PPID*.

Допълнително може да се измени поведението на породения процес

по отношение на някой сигнали. (Сигналите в ОС UNIX са разгледани в следващото упражнение.)

ИМЕ

fork

ПРОТОТИП

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

ОПИСАНИЕ

Системното извикване fork служи за създаване на нов процес в ОС UNIX. Процесът, който инициира системното извикване fork, е прието да се нарича процес-родител (parent process). Новопородените процес е прието да се нарича процес-дете (child process). Процесът-дете се явява почти пълно копие на процеса-родител. Процесът-дете започва изпълнението на потребителската програма на процеса-баща от оператора след fork. В породения процес за разлика от родителския се променят стойностите на следните параметри:
- идентификатор на процеса;
- идентификатор на родителския процес;
- времето оставащо до получаване на сигнала SIGALRM;
- сигналите, които се очаква да бъдат издадени към родителския процес няма да бъдат насочени към породения процес.

При това системно извикване един процес “влиза” във функцията fork, а два процеса “излизат” от fork и връщат различни стойности. Ако създаването на новия процес е извършено успешно, то в породения процес системното извикване ще върне стойност 0, а в родителския процес – положителна стойност, равна на идентификатора на процеса-дете. Ако не е създаден нов процес, то системното извикване ще върне отрицателна стойност в инициращия процес.
Системното извикване fork се явява единствения способ да се породи нов процес след инициализацията на ОС UNIX.

В процеса на изпълнение на системното извикване *fork()* се създава копие на родителския процес и връщането от системното извикване може да се случи както в родителския, така и в породения процес. Ако системното извикване завърши успешно, то връща два резултата – в процеса родител се връща *PID* на процеса-дете, а в процеса-дете – *0*. След завършване на системното извикване и двата процеса продължават изпълнението на заложения в програмата потребителски код.

За илюстрация на казаното по-горе е дадена следната програма:

```
/* Програма 2_1.c – пример създаване на нов процес
   с еднакво действие на процесите дете и родител */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
    pid_t pid, ppid;
    int a = 0;
    (void)fork();
    /* При успешно създаване на новия процес, на това място
       псевдопаралелно започват да работят два процеса:
       съществуващия и новия */
    /* Преди изпълнението на следващият израз стойността
       на променливата "a" и в двата процеса е 0 */
    a = a+1;
    /* Откриване на идентификаторите на текущия
```

```
        и родителския процеси (във всеки от процесите !!!) */
    pid = getpid();
    ppid = getppid();
    /* Отпечатва стойностите PID, PPID и изчислената стойност
       на променливата a (във всеки от процесите !!!) */
    printf("My pid = %d, my ppid = %d, result = %d\n", (int)pid, (int)ppid,
a);
    return 0;
}
```

Програма 2_1.c – пример за създаване на нов процес с еднакво действие на процесите дете и родител

Задача 2: Въведете програмата *2_1.c*, компилирайте и я изпълнете. Анализирайте полученият резултат.

След завършването на системното извикване *fork()*, в двата процеса (родител и дете) се връщат различни стойности. При успешно създаване на новия процес в процеса-родител се връща положителна стойност, равна на идентификатора на процеса-дете. В процеса-дете винаги се връща стойност *0*. Ако създаването на нов процес пропадне, то системното извикване връща в инициращия процес стойност -1. В този смисъл, обобщената схема на различно действие на процеса-дете и процеса-родител изглежда така:

```
pid = fork();
if(pid == -1){
    ...
    /* грешка */
    ...
} else if (pid == 0){
    ...
    /* дете */
    ...
} else {
    ...
    /* родител */
    ...
}
```

Задача 3: Изменете предходната програма с *fork()* (програма *2_1.c*) така, че родителя и детето да извършват различни действия (в случая не е важно какви точно действия ще бъдат извършени).

7. Завършване на процес

Съществуват два способа за коректно завършване на процес:

1. Процесът завършва при достигане до края на функцията *main()* или при изпълнение на оператора *return* във функцията *main()*;

2. Процесът завършва при изпълнение на функцията *exit()* от стандартната библиотека с функции в C. При изпълнение на тази функция се изпразват всички частично запълнени В/И буфери и се затварят

съответните потоци. След това се инициира системно извикване за прекратяване работата на процеса и преминаването му в състояние "Завършване". След приключване на функцията *exit()* не следва връщане в процеса, тъй като, той е завършил. Функцията не връща никакъв резултат.

Параметъра на функцията *exit()* се предава на ядрото на операционната система, след което може да бъде прочетен от процеса-родител на завършващия процес. По подобен начин, но неявно при достигане на края на функцията *main()* се извиква същата функция със стойност на параметъра *0*.

ФУНКЦИЯ

```
exit
```

ПРОТОТИП

```
#include <stdlib.h>
void exit(int status);
```

ОПИСАНИЕ

```
Функцията exit служи за нормално завършване на процес. При изпълнение на тази
функция се изпразват всички частично запълнени В/И буфери и се затварят
съответстващите им  потоци (файлове, pipe, FIFO, сокети), след което се
инициира системно извикване, прекратяващо работата на процеса и преминаването
му в състояние завършен.
Връщането от тази функция в текущия процес не произтича и функцията не връща
никакъв резултат.
Стойността на параметъра status се предава на ядрото на операционната
система. След това може да бъде получена от процеса породил завършващия. При
това се използват само младшите 8 бита на параметъра. За кода на завършване
са допустими стойностите от 0 до 255. Код на завършване 0 означава завършване
на процеса без грешки.
```

Ако един процес завърши своето изпълнение по-рано от родителя си и родителския процес не е указал явно, че не желае да получава информация за състоянието на завършване на породения процес (това е подробно разгледано в следващото упражнение), то този процес остава в състояние "Завършване" до завършване на процеса-родител или до момента когато родителя получи тази информация. Процес в ОС UNIX, който се намират в състояние "Завършване", е прието да се нарича процес-зомби (*zombie*, *defunct*).

8. Променливи на обкръжението и аргументи на командния ред

Процес е програма в хода на нейното изпълнение. Програмите на C започват изпълнението си от функцията *main()*, чийто прототип е:

```
int main(int argc, char *argv[], char *envp[]);
```

Тя има три параметъра, които може да получи от операционната система. Първите два, позволяват да се разбере пълното съдържание на командния ред при стартиране на програмата от командния ред на терминала. Всеки команден ред се разглежда като набор от думи разделени с интервали. Параметърът *argc* е броят на думите (аргументите) в командния ред. Параметърът *argv* е масив от указатели към отделните думи. Така, например, ако програмата е стартирана с командата

```
a.out 12 abcd
```

параметъра *argc* ще бъде равен на *3*, *argv[0]* ще указва първата дума (името на програмата) "*a.out*", *argv[1]* — думата "*12*", *argv[2]* — думата "*abcd*". Тъй като името на програмата винаги присъства на първо място в командния ред, то *argc* винаги ще бъде по-голямо от *0*, а *argv[0]* винаги ще бъде указател към името на стартираната програма.

Анализирайки съдържанието на командния ред в програмата, може да се предвиди различно поведение на програмата, респ. нейните процеси в зависимост от думите, следващи името й. По този начин без да се изменя кода в програмата, тя може да се застави да работи по различен начин, в според съдържанието на командния ред, с който бива стартирана. Например, компилатора *gcc*, извикан с командата *gcc 1.c* генерира изпълним файл с име *a.out*, а при извикване с командата *gcc 1.c –o 1.exe* – файл с име *1.exe*.

На една програма, освен аргументи, може да се предава и списък от променливи на обкръжението (environment variables). Списъкът от променливи се предава на програмата като масив от указатели на символни низове, посредством третият параметър *envp* на функцията *main()*. Всеки указател сочи към символен низ, който има вида: *променлива=стойност*.

Началните стойности на променливите на обкръжението за един процес се задават в специални конфигурационни файлове. Те са различни за всеки потребител и се установяват при входа на потребителя в системата. В последствие променливите на обкръжението могат да бъдат променяни с помощта на специални команди или програмни функции. Променливите на обкръжението се използват за промяна на дългосрочното поведение на процесите. Например, променливата

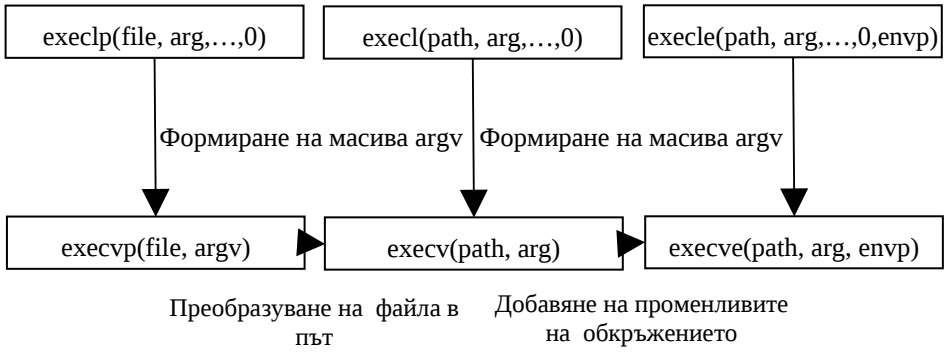
TERM=vt100 може да укаже на процесите, за чийто изход служи дисплея, че работят с терминал от вида *vt100*. Изменяйки стойността на променливата на *TERM=console*, на тези процеси се съобщава, че са длъжни да изменят своето поведение и да използват за изход системната конзола.

За да се определи размера на масива с променливи на обкръжението се използва факта, че последния елемент на този масив е указателя *NULL*.

Задача 4: Напишете самостоятелно програма, отпечатваща стойностите на аргументите на командния ред и променливите на обкръжението за процеса.

9. Промяна на потребителския контекст на процеса

Промяната на потребителския контекст на процеса се извършва с помощта на системното извикване *exec()*. То заменя потребителския контекст на текущия процес със съдържанието на някой изпълним файл и установява началните стойности на регистрите на процесора (в това число на програмния брояч). Това извикване изисква да се зададе името на изпълнимия файл, аргументите на командния ред и променливите на обкръжението. За осъществяване на извикването програмиста може да се използва една от следните шест функции: *execlp()*, *execvp()*, *execl()*, *execv()*, *execle()* и *execve()*. Те се различават една от друга по начина на предаване на аргументите и използване на променливите от обкръжението на процеса.



Фиг. 2.3. Връзки между функциите на системното извикване exec()

В трите функции от първия ред, всеки аргумент на main е зададен като отделен аргумент на функцията *exec()*. В трите функции от долния

ред има един аргумент *argv*, който е масив от указатели към аргументите за *main()*.

В двете функции в ляво *file* може да е собствено име на файл, което се преобразува в пълно име чрез променливата *path*. В останалите функции *path* трябва да е пълно име на файл.

Двете функции в дясно имат аргумент *envp*, който е масив от указатели към символни низове, съдържащи променливите от обкръжението на процеса.

При успех, когато процесът се върне от *exec()* в потребителски режим на изпълнение, 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 елементи на контекста му. При грешка по време на *exec()* става връщане в стария потребителски контекст, така че функцията връща *-1* при грешка, а при успех не връща нищо защото няма връщане в стария потребителски контекст.

ИМЕ

```
execlp execvp execl execv execle execve
```

ПРОТОТИПИ

```
#include <unistd.h>
int execlp(const char *file, const char *arg0,
... const char *argN,(char *)NULL)
int execvp(const char *file, char *argv[])
int execl(const char *path, const char *arg0,
... const char *argN,(char *)NULL)
int execv(const char *path, char *argv[])
int execle(const char *path, const char *arg0,
... const char *argN,(char *)NULL, char * envp[])
int execve(const char *path, char *argv[], char *envp[])
```

ОПИСАНИЕ

```
За зареждане на нова програма в системния контекст на текущия процес се
използва семейство от взаимосвързани функции, отличаващи се една от друга по
формата на представяне на параметрите.
Аргументът file се явява указател към името на файла, който трябва да бъде
зареден. Аргументът path – това е указател към пълния път на файла, който
трябва да бъде зареден.
Аргументите arg0, ..., argN представляват указатели към аргументите на
командния ред. Необходимо е да се отбележи, че аргумента arg0 трябва да
указва името на зареждания файл. Аргументът argv представлява масив от
указатели към аргументите на командния ред. Началният елемент на масива
трябва да указва името на зарежданата програма, а последния елемент на масива
трябва да съдържа указателя NULL.
Аргументът envp се явява масив от указатели към параметрите на обкръжаващата
среда, зададени във вида "променлива=стойност". Последният елемент на този
масив трябва да съдържа указателя NULL.
Извикването на функциите не изменя системния контекст на текущия процес.
Зарежданата програма унаследява от зареждащия я процес следните атрибути:
```

- идентификатор на процеса;
- идентификатор на родителския процес;
- идентификатор на групата на процеса;
- идентификатор на сесията;
- време, оставащо до възникване на сигнала SIGALRM;
- текущата работна директория;
- маска на създаваните файлове;
- идентификатор на потребителя;
- идентификатор на групата на потребителя;
- явно игнориране на сигнали;
- таблица на отворените файлове (ако във файловия дескриптор не е установен флага "затвори файла при изпълнение на exec()").

В случай на успешно изпълнение не следва връщане от функцията в програмата, осъществила извикването, а управлението се предава на заредената програма. В случай на неуспешно изпълнение – в програмата, инициирала извикването се връща отрицателна стойност.

Доколкото системния контекст на процеса при извикване на *exec()* остава практически непроменен, болшинството атрибути на процеса (*PID*, *UID*, *GID*, *PPID* и др.), достъпни до потребителя, чрез системните извиквания, след изпълнението на нови програми също не се изменят.

Важно е да се разбере разликата между системните извиквания *fork()* и *exec()*. Системното извикване *fork()* създава нов процес, който е копие на процеса-родител (потребителския контекст съвпада с потребителския контекст на процеса-родител) и продължава да изпълнява същата програма. Системното извикване *exec()* изменя потребителския контекст на текущия процес, а не създава нов процес. Чрез него всеки процес може да извика за изпълнение друга програма по всяко време от своя живот.

За илюстрация на използването на системното извикване *exec()* е дадена следната програма:

```
/* Програма 2_2.c, променяща потребителския контекст на процеса
  (зарежда за изпълнение друга програма) */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char *argv[], char *envp[]){
/* Зарежда се командата cat с аргумент от командния ред 2_2.c
   без изменение на параметрите на средата, т.е. изпълнява се
   командата "cat 2_2.c", която показва съдържанието на посочения
   файл на дисплея. Първия параметър на функцията execle е името
   на стартираната програма. В случая е указано нейното пълно
   име с пътя от кореновата директория –/bin/cat.
   Първата дума в командния ред трябва да съвпада с името на
   заредената програма. Втората дума в командния ред е името на
   файла, съдържанието на който ще бъде разпечатано. */
(void) execle("/bin/cat", "/bin/cat",
              "03_2.c", 0, envp);
/* Следващата инструкция се изпълнява само при
   възникване на грешка */
printf("Error on program start\n");
exit(-1);
return 0;     /* Никога не се изпълнява,
                 необходимо е само, за да
```

```
                    не се издават предупреждения
                    от компилатора - warnings */
}
```

Програма 2_2.c, променяща потребителския контекст на процеса

Задача 5: Въведете програмата и я запишете във файл с име *2_2.c* в текущата директория. Компилирайте и изпълнете програмата. Анализирайте резултата.

Задача 6: Променете програмата, която създадохте в задача 3, така че породения процес да стартира изпълнението на нова програма (по ваш избор).