1.**Файлов дескриптор: абстракция за информационен канал между процес и други обекти в ОС** UNIX

Всяка ОС(т.е. ядро) осигурява определен набор от операции за изпълняваните програми. Тези операции се наричат системни извиквания(system calls).

Файлове

Повечето ОС реализират дървовидна организация на файловете чрез специален тип файл, наречен директория. Всичко започва от „/”. Всеки файл има собствено име, абсолютно пълно име(absolute path name)-единственият път в дървото от корена до файла, относително име-списък от имената на директориите в пътя, тръгвайки от текущата директория. Всеки потребител има home directory.

Програми и процеси

Програма е файл, съдържащ изпълним код и съхраняван на диска като обикновен файл. След като програма бъде заредена в паметта и започне нейното изпълнение под управлението на ядрото, започва животът на процеса. Процес е програма в хода на нейното действие, т.е. инстанция на работеща програма. В някои ОС се използва и термина task. Всеки процес има уникален идентификатор, който се нарича process ID(pid). Идентификаторът на процес е неотрицателно число. ОС осигурява четири основни примитива за работа с процеси – fork, exec,exit,wait

Потребителски идентификатори

Всеки потребител, регистриран в системата има потребителско име и уникален числов идентификатор, наричан потребителски идентификатор(User ID). Този uid е неотрицателно цяло число и се използва в структурите на ОС при определяне на правата на потребителите. Потребителят с uid 0 е администратора или привилегирования потребител, известен още като root. Всеки потребител принадлежи на определена потребителска група и се идентифицира чрез group ID . С всеки процес се свързва uid и git на потребителя.

Системни примитиви и библиотечни функции

Системните примитиви са програмният интерфейс на ядрото на ОС. За всички системни примитиви има поне една функция в стандартната библиотека на C . Потребителската програма съдържа обръщение към тази функция. Това, което прави функцията е да предаде управлението на ядрото, като най-често използваната техника е с програмно прекъсване. Действителната работа по системните примитиви се върши от програмен код в ядрото. Те са входове в ядрото и затова ги наричат системни извиквания

Обработка на грешки

При успех функцията на системния примитив обикновено връща цяло неотрицателно число, когато се случи грешка връща -1. Механизмът за докладване на грешки е чрез глобалната променлива errno, чрез която ядрото връща код на грешката.

Примитивни системни типове данни

Стандартът POSIX въвежда примитивни системни типове данни, те са производни типове данни, чиито имена завършват на _t. Обикновено са дефинирани в заглавния файл <sys.types.h>.

Идентификатор на отворен файл се нарича файлов дескриптор. Файловият дескриптор е цяло неотрицателно число. Когато се изпълни системен примитив open за отваряне на файл, ядрото връща в процеса файлов дескриптор. При всички следващи операции четене, писане и др.

процесът се идентифицира с този файлов дескриптор. Файловият дескриптор се освобождава при изпълнение на close. Дескрипторът има локално значение- връзката между дескриптор(число) и отворен файл важи само за текущия процес. Всеки процес разполага с N файлови дескриптора. Всеки процес стандартно има 3 отворени дескриптора:

- 0 (стандартен вход)
- 1 (стандартен изход)
- 2 (стандартен изход за грешки)

С всяко отваряне на файл се свързва и указател към текущата позиция (file offset, file pointer), което определя позицията на файла, от която ще бъде четено или записвано. От гледна точка на ядрото файлът представлява масив от байтове . Отместването(offset) указва броя байтове от началото на файла до текущата позиция. Всеки файлов дескриптор има собствена позиция, която се променя от извършваните върху него процедури.

С всяко отваряне на файл(с всеки файлове дескриптор) се свързва и режим на отваряне на файла, който определя начинът на достът до файла от съответният процес, напр. само четене, писане и т.н. При всеки следващ опит за достъп до файла се проверява дали той не противоречи на режима на отваряне, и ако е така достъпът се отказва

Системни таблици

Три основни структури данни се използват в ядрото при реализацията на системните примитиви на файловата система. Наричат ги системни таблици и са разположени в пространството на ядрото. Това са:

1. Таблица на индексните описатели

   За всеки отворен файл в таблицата на индексните описатели се пази точно един запис, съдържащ копие на индексният описател (i-node) от диска и някои други полета, като номер на устройството , номер на i-node

2. Таблица на отворените файлове

   При всяко отваряне на файл се отделя един запис в тази таблица, който съдържа:

   Текущата позиция във файла

   Режима на отваряне на файла

   Указател към съответния запис от таблицата на i-node

   Брояч- брой указатели, насочени към записа

3. Таблици на файловите дескриптори

   Всеки процес има собствена таблица на файловите дескриптори. Броят на елементите в нея определя максималния брой едновременно отворени файлове за процес. Записите в тези таблици съдържат само указател към запис от

   таблицата на отворените файлове и флагове на файловия дескриптор. Файловият

   дескриптор всъщност представлява индекс на използвания при отварянето запис от

   таблицата на файловите дескриптори.

3.**Основни системни извиквания за ползване на файлов дескриптор**: read, write, close

Една от основните задачи на ОС е да съхранява файлове. Файловата система е множество от файлове с техните имена.

read

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

Системният примитив read чете n последователни байта от файла идентифициран с файлов дескриптор, като четенето започва от текущата позиция. Прочетените байтове се записват в buffer, указателят на текущата позиция се увеличава с действителният брой прочетени байтове, тоест сочи байта след последният прочетен файл. Функцията връща действителния брой прочетени байта.

write

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

Пише n байта от buffer във файла, данните ще бъдат пратени в комуникационния канал, връща реално прочетените байтове. Write може да пише и в позиция след края на файла, при което се извършва увеличаване на размера.

close

Примитивът close затваря отворен файл, тоест прекратя връзката между процеса и файла, като освобождава файловия дескриптор, който може да бъде използван при други извиквания

```
#include <unistd.h>

int close(int fd);
```

Когато процес завършва(изпълнява exit), ядрото автоматично затваря всички отворени файлове. Затова понякога в програмите не се извиква явно close за затваряне на файловете.

4. **Работа с обикновени файлове**: open(), lseek();

ОС предлага прости инструменти за работа с файлове. Съвременният UNIX предлага абстракцията файл да бъде разглеждана като масив от байтове.

Open

```
#include <fcntl.h>

int open (const char *filename, int oflag [,mode_t mode]);
```

- отваря файла име filename в режим oflag, а ако не съществува го създава с режим mode
- връща файлов дескриптор, с който работим , а при неуспех -1.

Oflag указва режимът на отваряне и действията, които да се извършат:

- O_RDONLY- само за четене

- O_WRONLY- само за писане
- O_RDRW – за четене и писане
- O_CREAT – създава файла
- O_EXCL- заедно с O_CREAT, ако файлът не съществува се създава, иначе се връща грешка
- O_TRUNC- старото съдържание на файла се изтрива след отваряне
- O_APPEND- старото съдържание се запазва, а новото се вписва в края на файла

Open създава връзка между програмата(процеса) и файла, връща номер на комуникационния канал(файлове дескриптор). Комуникационните канали са средства за комуникация между процеси. Open създава връзка между процеса и указаният файл, която се идентифицира чрез файлов дескриптор и включва режима на отваряне на файла и текущата позиция във файла. Чрез open може да се създаде файл, ако не съществува и след това да се отвори.

lseek

След open текущата позиция във файла сочи началото му, т.е 0. Примивът lseek позволява да се премести указателя на текущата позиция на произволна позиция във файла.

```
#include <fcntl.h>
off_t lseek(int fd, off_t offset,int whence);
```

- fd -номер на файлов дескриптор
- offstet – самото отместване
- whence -указател за интерпретация на отместването

Ø SEEK_SET – премества указателя от началото до offset байт (the file offset is set to offset bytes)

lseek(fd,0, SEEK_SET) – започваме от 0

lseek(fd,20, SEEK_SET) – започваме от 20-та позиция нататък

Ø SEEK_CUR – премества указателя от сегашната му позиция до offset байта(the file offset is set to current location + offset bytes)

Ø SEEK_END -премества указателя от края на файла (отзад-напред) до offset байта.

При изпълнението на примитива lseek не се изисква достъп до диска. Изменя се единствено полето за текуща позиция в съответния запис от таблицата на отворените файлове. Lseek е специфично само за обикновени файлове и блочни устройства, тоест само за масиви от байтове. Когато използваме системни извиквания трябва да знаем какъв обект имаме отсреща.

Неделимост на операциите

Под неделима или атомарна операция се разбира операция, която се състои от няколко стъпки, но изпълнението им се осигурява от принципа „всичко или нищо“, тоест или всички

стъпки се изпълняват, или нито една не се изпълнява. Ядрото гарантира неделимост за всеки отделен системен примитив.

5.**Анонимни и именувани тръби** : pipe(), mkfifo()

Неименувана(анонимна тръба) се създава чрез системното извикване pipe(fd[2]). То връща два файлови дескриптора на мястото на елементите на подадения масив int fd[2]. fd[0] съхранява файловия дескриптор за четене, а fd[1] за писане. Тази тръба не е именувана, тя е видима само за процеса, който я е създал , както и от наследниците му. Тази тръба може да се използва само в системата, тоест не може да се използва в интернет. Тя служи за връзка между процес и друг процес, който има роднинска връзка с първоначалния процес. Тази тръба е неименувана, тя не се използва в пространството на имената. Тръбата е канал за обмен на информация между процеси. На логическо ниво обикновената тръба е едностранна връзка, целият механизъм е реализиран в ядрото, не е видимо за програмата. Тръбите осигуряват еднопосочен канал за комуникация между процесите. Тръбата има край за четене и край за писане. Данни, записани в края за писане могат да бъдат прочетени от края за четене. Тръба се създава с pipe(2) и връща 2 файлови дескриптора, единият се отнася за края за четене, а другият за края за писане. Тръбите се използват за създаване на комуникационен канал между свързани процеси.

```
int pipe2(int pipefd[2], int flags);
```

Масивът pipefd[2] се използва, за да върне 2-та файлови дескриптора, отнасящи се за крайовете на тръбата, ако flags е 0, тогава pipe2() и pipe() са еднакви.

Преимущества на pipe

- Синхронизира работата на 2 процеса
- Не прави временен файл, тоест не се ползва файлово пространство
- Синхронизация се налага, когато един процес работи по-бързо от друг
- Осигурява наредба във времето
- Не използва съществени хардуерни ресурси.
- Направена е по такъв начин, че да няма Race Condition

Особености на съвременната тръба

Като пишем байтове и тяхната бройка не надвишава определена константа, тогава операцията се извършва атомарно

- Ако размера е под 64К, целият масив ще се запише
- Ако размера е > 64К, потокът ще се нареже на части.

mkfifo

```
#include <sys/types.h>
  #include <sys/stat.h>

  int
  mkfifo(const char *path, mode_t mode);
```

Именуваната тръба(FIFO) се създава чрез библиотечното извикване mkfifo(). Тази тръба е видима за всички процеси в системата и може да бъде ползвана от тях. Тя се използва за осъществяване на комуникация между 2 процеса, които не са задължително в роднинска връзка. Този вид тръба се свързва с име, откъдето следва че я се използва в пространството на имената.

6.**Атрибути на файлове и тяхната промяна**: stat, chmod, chown, unmask, utime

stat

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

       int stat(const char *pathname, struct stat *statbuf);
```

Получава статуса на файла. Вторият аргумент statbuf е указател на структура stat, в която примитивът записва информацията за файла. Stat получава информация за файла, сочен от pathname

Типове файлове

Обикновен файл е файл, съдържащ данни в някакъв формат

Директория- файл, съдържащ записи за други файлове, всеки от който съдържа собствено име за файл и номера на i-node му. Чрез този тип се реализира йерархична организация на файловата система

Символен специален файл(character special device file)- всеки файл от този тип съответства на символно устройство, напр. терминал, печатащо устройство и др.

Блоков специален файл(block special device file) – всеки файл от този тип съответства на блоково устройство , например диск

Символна връзка- файл, който сочи към друг файл

Програмен канал(pipe || FIFO file)-тип файл, реализиращ механизъм за междупроцесни комуникации

Код на защита на файл и права на достъп

С всеки файл е свързан потребител- собственик на файла и потребителска група, към която принадлежи собственика. Тези два атрибута са съответно в елементите st_uid и st_gid.

chmod

```
int chmod(const char *pathname, mode_t mode);
```

Променя правата на файл. Примитивът chmod изменя кода на защита на файла filename според указаното в аргумента mode. Процесът, който изпълнява chmod или fchmod, трябва да принадлежи на собственика на файла или на привилегирования потребител.

Chown

```
int chown(const char *pathname, uid_t owner, gid_t
group);
```

Смяна собствеността на файла. Всеки файл има атрибути собственик на файла и потребителска група, към която принадлежи собственика (елементите st_uid и st_gid в структурата stat). Когато се създава файл с creat, open, mkdir или mknod, той трябва да получи значения за

тези два атрибута (за тях няма аргументи в примитивите). Собственик на файла става ефективния потребителски идентификатор на процеса (euid). По отношение на групата на файла има различни реализации:

- Групов идентификатор на файла става ефективния групов идентификатор на процеса (egid).
- Групов идентификатор на файла става груповия идентификатор на родителския му каталог. Новосъздаденият файл наследява групата от родителския си каталог.

Изменят собственика и/или групата на файла според указаното в аргументите owner и group. Ако значението на аргумента owner или group е -1, то не се изменя съответно собственика или групата.

Umask

```
mode_t umask(mode_t mask);
```

По отношение на компютрите **маска** е специална стойност, която действа като филтър за данни. Нарича се „маска“, защото разкрива някои части от цифровата информация и прикрива или променя други. В Unix-подобни операционни системи като Linux, BSD и macOS X, umask е маска на осмични стойности, която задава разрешенията за нови файлове, създадени в системата.

utime

Променя времето на последния достъп до файла и последната модификация

```
#include <sys/types.h>
#include <utime.h>

int utime(const char *filename, const struct utimbuf
*times);
```

Всеки файл има три атрибута за време, които се съхраняват в i-node и се намират

в елементите st_atime, st_mtime и st_ctime на структурата stat. Всяко от тези

полета съдържа дата и време на определено събитие от живота на файла:

- st_atime - на последния достъп до файла
- st_mtime - на последното изменение на данните на файла
- st_ctime - на последното изменение на i-node на файла

Системният примитив utime позволява да се изменя значението на две от времената: на последен достъп и на последно изменение на данните на файла (st_atime и st_mtime).

Структурата, използвана във функцията, е определена в заглавния файл <utime.h>:

```
struct utimbuf {
time_t actime; /* access time */
time_t modtime; /* modification time */
};
```

Значението на двете времена в структурата е цяло число – брой секунди от началото на Unix епохата, която започва в 00:00:00 на 01.01.1970. Първият аргумент filename определя името на файла, за който ще се променят времената. Действието на примитива зависи от значението на втория аргумент.

1. Ако аргументът times е указател NULL, то и двете времена се изменят с времето от системния часовник. За успешното изпълнение е необходимо euid на процеса да е равен на собственика на файла или процесът да има право за писане във файла.
2. Ако аргументът times е различен от указател NULL, то времената се изменят със значенията на двата елемента от структурата, сочена от него. В този случай euid на процеса трябва да е равен на собственика на файла или да е 0 (root).

7.**Манипулиране на файлов дескриптор**: fcntl(),dup()

dup()

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Системното извикване dup() създава копие на oldfd, като използва възможно най-малкото неизползвано число за новия файлов дескриптор, който е дубликат на oldfd(копира се на 1-вото свободно място в таблицата на файловите дескриптори). След успешно изпълнение, новият и старият файлов дескриптор стават взаимозаменяеми. Следователно 2-та файлови дескриптора имат един и същи отворен файл, общ указател на текуща позиция и еднакъв режим на отваряне на файла. Например, ако отместването на файла е модифицирано чрез lseek() от единия файлов дескриптор, то отместването също се сменя и за другия. Примитивът dup се използва за пренасочване на стандартен вход , изход или изход за грешки и при организиране на конвейер от програми.

fcntl()

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* arg */ );
```

fcntl манипулира файловия декриптор, изпълнява една от операциите върху отворения файлов дескриптор(cmd). Могат да се четат или изменят различни свойства на отворен файлов дескриптор. Разглеждаме основните приложение на fcntl:

четене/ изменение на флагове на файлов дескриптор

четене/изменение на режим на отваряне

копиране на файлов дескриптор

Дублициране на файлов дескриптор

F_DUPFD(int) – дубликираме файловия дескриптор като използваме най-малкото възможно число. При успех връщаме новия файлов дескриптор

8.**Връзки в пространството на имената**: link(), symlink(), unlink()

Разглеждаме системни примитиви, чрез които се изгражда и манипулира системата от директории, създават се и се унищожават връзки към файл и се изгражда единна йерархична структура на файловата система. Всеки файл има индексен описател или i-node , който е с фиксирана дължина. Индексните описатели на всички файлове в един том са разположени в специална индексна област на диска и се адресират чрез номер на i-node(пореден номер на i-node в индексната област, започвайки от 1). В индексния описател се съхраняват всички атрибути на файла: тип, код на защита, собственик, група, размер в байтове и в блокове, дати на последен достъп до файла. Съхраняват се и определен брой адреси на дискови блокове, разпределени за файла, тези блокове съдържат данни или адресна информация за файла.

Предназначението на връзките е да се позволи достъп към един файл чрез различни имена, евентуално разположени в различни директории във файловата система. Реализират се 2 вида връзки към файл – твърда връзка(hard link) и символна връзка(soft link || symbolic link).

link()

```
int link(const char *oldpath, const char *newpath);
```

Създава ново име на файл, тоест създава твърда връзка към файла. Ако съществува файл с име newpath това е грешка. Това, което прави link е включи нов запис в родителската директория с име newpath. файла. Освен това в индексния описател на файла се увеличава с 1 броя на твърдите връзки за файла. Двете стъпки – добавянето на записа и изменението на i-node, се

изпълняват като неделима операция. Когато имаме твърда връзка , промените който се правят върху единия файл стават и в другия(тоест имаме 2 имена за един файл), и при символните връзки е така.

symlink()

Символната връзка е тип файл, който представлява символен указател към друг файл. При символните връзки, когато имаме един файл и създадем символна връзка към него , ако изтрием или преименуваме оригиналния файл и след това се опитаме чрез cat да видим съдържанието на символната връзка, то то е невалидно, докато при твърдите връзки, ако изтрием оригиналният файл , после ако се опитаме чрез cat да видим съдържанието на Hard link-а то ще се изпринтира.

```
int symlink(const char *target, const char *linkpath);
```

Създаваме нов файл от тип символна връзка с име Linkpath , който сочи към target. При създаване на символна връзка дори не се проверява дали target съществува. Когато отваряме файл с open, ако аргументът на файла е символна връзка, то се следва символната връзка и се отваря файлът, към който тя сочи. Ако соченият файл не съществува се връща грешка.

unlink()

Изтрива връзка към файл(изтрива име във файловата система.

```
int unlink(const char *pathname);
```

Освен това в индексния описател на файла се намалява с 1 броя на твърдите връзки за файла. Ако броят стане 0, т.е. това е било последното име за файла, то той се унищожава освобождават се блоковете с данни, индексния описател и косвените блокове). Ако по време на изпълнението на unlink файлът е отворен, то действителното му унищожаване се извършва когато последният процес, в който файлът е бил отворен, го затвори (при изпълнение на close).

Изтриваме име от файловата система, ако това име е било последният линк към файла и никои процеси нямат файла отворен, тогава файлът бива изтрит, и паметта, която е използвал е свободна за нова употреба. Ако името е било последният линк към файл, но за някой процеси файлът все още е отворен, файлът ще съществува, докато последният файлове дескриптор, който сочи към него не се затвори. Ако името е за socket/ fifo или друго устройство, името за него е изтрито, но процесите, които имат обекта отворен може да продължат да го използват.

9.**Работа с каталози**:mkdir(),rmdir(), chdir(), opendir(), readdir(),closedir(),rewinddir()

mkdir()

```
#include <sys/stat.h>
    #include <sys/types.h>
```

```
int mkdir(const char *pathname, mode_t mode);
```

mkdir създава нов правен каталог(директрория) с име pathname и код на защита mode. Празна директория означава, че тя съдържа само „.“ И „..“. Процесът трябва да има права x за всички директории по пътя в пълното име и право w за родителския каталог. Грешка ще има, ако съществува файл със същото име.

rmdir()

```
#include <unistd.h>

int rmdir(const char *pathname);
```

Изтрива директория, която трябва да е празна, тоест да има само „.“ И „..“. Процесът трябва да има права x за всички каталози на пътя в пълното име и право w за родителския.

chdir()

```
#include <unistd.h>

int chdir(const char *path);
```

Сменя текущата директория с тази, която е специфицирана в path. Указаният чрез аргумента каталог dirname става новия текущ каталог на процеса. Текущият каталог представлява активен файл, т.е. при изпълнение на chdir индексният описател на новия каталог се зарежда в таблицата на индексните описатели, а i-node на стария текущ каталог се освобождава. Процесът трябва да има права x за всички каталози в пълното име dirname. При успех функцията връща 0, а при грешка -1 и текущият каталог не се променя.

Opendir()

```
DIR *opendir(const char *name);
```

Структурата DIR се свързва с отворен за четене каталог и използва останалите функции. Структурата dirent е определена в заглавния файл <dirent.h> и е системно зависима, но съдържа поне следните два елемента:

```
struct dirent {
ino_t d_ino; /* inode number */
char d_name[NAME_MAX+1]; /* file name (null-terminated) */
};
```

Елементът d_name съдържа собственото име на файла, като в края на низа е добавен символа ’\0’. Значението на NAME_MAX зависи от типа на файловата система и може да бъде определено чрез функцията pathconf (виж Програма 1.3). Елементът d_ino съдържа номера на i-node на файла.

opendir()

```
#include <sys/types.h>
#include <dirent.h>
```

```
DIR *opendir(const char *name);
```

отваря директория за четене с име name и връща указател към структурата dir, при грешка връщаме NULL.

readdir()

```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

Чете поредния следващ запис от каталога (при първо извикване чете първия запис), като наредбата за записите не е по името на файла, а е системно зависима. Прочетеният запис се предава чрез структурата dirent, указател кум която се връща като значение функцията при успех. При достигане на край на каталог или грешка функцията връща NULL. Ако след opendir многократно извикваме readdir докато върне NULL, ще прочетем последователно записите в каталога.

rewinddir()

```
void rewinddir(DIR *dirp);
```

Позиционира в началото на каталога. Това позволява след отваряне на каталог да се изпълнят няколко последователни преминавания през него.

closedir()

```
int closedir(DIR *dirp);
```

Функцията затваря каталога и затваря (underlying) файлов дескриптор, асоцииран с dirp.

10.**Монтиране и синхронизация**: mount, umount, sync, fsync

Монтиране и демонтиране на файлова система и специални файлове

Системните примитиви mount и umount позволяват за потребителя винаги да се изгражда единна йерархична файлова система независимо от броя на носителите(твърди дискове или техни дялове). На всеки носител е изградена йерархична файлова система чрез командата mkfs, една от които съдържа програмата за начално зареждане и ядрото на ОС и се нарича коренова файлова система. Чрез примитива mount некоренова файлова система на определен носител може да бъде присъединена(монтирана) към коренова файлова система.

```
int mount(const char *source, const char *target,
                const char *filesystemtype, unsigned long
mountflags,
                const void *data);
```

Mount() прикрепя файлова система, специфицирана от source(често е път или име, отнасящо се до устройство, но също може да бъде и име или път на директория или файл) към локация(директория или файл) специфицирана от target.

```
#include <sys/mount.h>

int umount(const char *target);
```

Разрушаване на връзката между коренната и коя да е друга монтирана файлова система се извършва чрез този системен примитив

Буфериране на входно-изходните операции

Ядрото изпълнява входно-изходните операции за блоковите специални устройства като използва буфериране(определен брой буфери, намиращи се в пространството на ядрото и наричани буферен кеш). Когато се изпълнява примитив read първо се чете от дисковия блок в системен буфер, и след това исканите байтове се копират в областта на процеса. Аналогично при изпълнение на примитив write, даните първо се копират в съответния буфер, той се отбелязва за запис и примитивът завършва, но действителното записване на буфера на диска се отлага за по-късен момент. Това е така наречената стратегия на отложен запис (delayed write). Обикновено ядрото записва отбелязаните блокове когато трябва да освободи буфер за друга входно-изходна операция. Ако е вдигнат флаг O_SYNC при open, то write връща управлението след физическото записване на данните и управляващата информация на диска, т.е. писането на диска е синхронно.

Обикновено в повечето случаи за файловете се прилага стратегия на отложен запис. Ако искаме да синхронизираме данните на диска със съдържанието на буферния кеш, то можем да ползваме sync и fsync.

```
#include <unistd.h>

void sync(void);
```

Примитивът sync планира запис на диска за всички отбелязани буфери в буферния кеш, но може да не чака действителното приключване на записа. (В някои версии на Linux се чака приключването на записа.) Обикновено sync се извиква периодично на всеки 30 секунди от специален процес-демон. Това гарантира регулярното синхронизиране на диска с буферния кеш. Има и команда sync, която всъщност извиква примитива.

```
int fsync(int fd);
```

Примитивът fsync приема аргумент fd, който е файлов дескриптор на отворен файл и синхронизира данните и атрибутите само за него. Примитивът fdatasync синхронизира само данните на файла, без атрибутите му. И двата примитива изчакват да завърши дисковата операция и тогава връщат управлението. (В някои случаи, в зависимост от типа на дисковото устройство, данните може и да не са стигнали до дисковата памет при връщане от примитива.)

11.**Управление на процеси**: fork(), exit(),atexit(), wait(), exec(), getpid(), getppid()

Инстанция на работеща програма наричаме процес. В понятието процес освен програмата се включва и информация за процеса, която ще наричаме атрибути на процес. Основни атрибути на процес, които се съхраняват в системните структури Таблица на процесите и потребителска област:

- Идентификатор на процеса(pid)
- Идентификатор на процеса -баща (ppid or parent pid)

- 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 за създаване на файлове заредена с примитива umask
- Реакция на процеса при получване на различни сигнали
- Време за изпълнение на процеса в системна и потребителска фаза

fork()

Единственият начин за създаване на нов процес от ядрото е когато съществуващ процес извика fork. Новосъзданеният процес се нарича дете, а оригиналния процес се нарича родител.

```
#include <unistd.h>

    pid_t fork(void);
```

Родителският и детинският процес се изпълняват на отделни места(memory space). По време на fork двете места имат едно и също съдържание. Детинският процес е точен дубликат на родителския освен:

- Детето има свой pid
- Ppid на детето е същият като pid на родителя

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.

```
#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. 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.

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 сървъри.

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.

exit()

```
#include <stdlib.h>

        void exit(int status);
```

Осигурява чисто завършване на процеса

Atexit()

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

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.

wait()

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

Родителският процес може да разбере как е завършило детето му чрез wait(). Ако детето не е завършило, то родителският процес бива блокиран докато синът не извърши exit, тоест изчаква неговото завършване. Функциите на системния примитив връщат пид-а на завършилият детински процес, а чрез аргумента статус получаваме информация за начина мъ на завършване, тоест кода на завършване. Родителският процес е отговорен за своите деца. Когато процес завърши се очаква неговият баща да се осведоми за завършването му с wait. При изпълнение на wait от родителския процес се изчиства сина-зомби от системата, тоест 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 родителския процес 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 зомбите-сираци.

exec()

Когато с форк се създава нов процес, той е копие на баща си, тоест продължава да изпълнява същата програма. Чрез примитива exec всеки процес може да извика за изпълнение друга програма по всяко време от своя живот, която веднага след създаването си така и по-късно, дори няколко пъти в живота си. Семейството от функции на exec замества текущият процес с друга програма

getpid(), getppid()

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

getpid() връща пида на процеса.

getppid() връща пида на родителския процес.

14.**Изпращане и обработка на сигнали**: signal(), kill(), pause(), alarm()

Сигналът е имитация на хардуерно прекъсване. В ОС работим с абстракции (процеси, канали и т.н.). Абстракциите са изолирани от хардуера. Програмата ползва чисти абстракции. Interrupt handler – обработчик на прекъсванията. 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. Сигналите осигуряват начин за справяне с асинхронни събития- напр. потребител, който

използва терминал и пише the interrupt key, за да спре програмата. Най – напред всеки сигнал си има име и уникален номер. Тези имена започват с първите 3 букви SIG.

Сигнали, свързани с управляващия терминал:

- SIGINT Изпраща се когато потребителят натисне клавиша <Del> или <Ctrl>+<C>.
- SIGQUIT Изпраща се когато потребителят натисне клавишите <Ctrl>+<\>.
- SIGHUP Изпраща се при прекъсване на връзката с управляващия терминал.

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 системния 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.

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 от сигнала.

Реакцията за всички типове сигнали се помни в потребителската област на процеса, 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.

Signal()

```
#include <signal.h>

        typedef void (*sighandler_t)(int);

        sighandler_t signal(int signum, sighandler_t handler);
```

Аргументът signum задава номера на сигнала, а sighandler определя каква да е реакцията при на получаване на сигнал. Значението на втория аргумент може да е едно от следните:

- SIG_IGN - игнориране на сигнал;

- SIG_DFL - реакция по премълчаване, която за повечето типове сигнали е завършване на процеса;

- име на потребителската функция.

Kill()

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

Системното извикване изпраща сигнал към някой процеси или процесна група.

Ако:

- pid> 0 , тогава сигналът sig се изпраща до процеса с id, определено от pid
- pid==0, тогава сигналът се изпраща на всички процеси от групата на процеса, изпращащ сигнала
- pid==-1, тогава сигналът се изпраща на всички процеси, които имат права да изпращат сигнали, освен на процеса 1(init)
- pid<-1, тогава сигналът се изпраща на всеки процес от групата на процеса, чието id е pid.

Процесът, който изпраща сигналът трябва да има права, т.е. трябва да е изпълнено едно от условията:

- Ефективният потребителски идентификатор на процеса(euid), изпращащ сигнала, да е 0
- 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 аргументът sig е 0, сигнал не се изпраща, но се прави проверка за грешка.

Pause()

```
int pause(void);
```

Системният примитив кара викащият процес(или нишката) да заспи, докато не се достави сигнал, който или приключва процеса, или кара извикването на други функции, които хващат сигнала. Функцията връща резултат само когато сме хванали сигнал и функциите, който хващат сигнали са върнали. В този случая, примитивът връща -1.

Alarm()

```
unsigned int alarm(unsigned int seconds);
```

Системният примитив планира изпращането на сигнал SIGALARM на процеса, изпълняващ примитива. В seconds се задават брой секунди, тоест при изпълняване на аларм в таймера на процеса се зарежда значението на аргумента секунди. Когато изтече този интервал от време, ядрото ще изпрати сигнал SIGALARM на процеса
Комуникация между процеси

IPC е съкращение от Interprocess Communication или комуникация между процеси. Когато два или повече конкурентни процеса взаимодействат помежду си, то между тях трябва да има съглашение за това и операционната система трябва да осигури някакъв механизъм за предаване на данни и синхронизацията им.

С развитието на на операциоммитр системи ЮНИКС в тях са включени различни методи и механизми за междупроцесна комуникация:

- програмни канали (pipes)
- именовани програмни канали (named pipes или FIFO файлове)
- съобщения (message queues)
- обща памет (shared memory)
- семафори (semaphores)

Когато два или повече процеса използват някакъв механизъм за обмен на информация, то IPC обекта трябва да има име или идентификатор. Така един от процесите ще създаде IPC обекта, а останалите ще получат достъп към този конкретен обект. Множеството от имена за определен вид IPC, наричаме пространство на имената.

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.

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 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.

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 броят байта, които се пишат при това извикване, е по-малък или равен на 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 броят 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 се размера на файла с прочетения брой байта и се събуждат всички процеси, чакащи да пишат в канала.

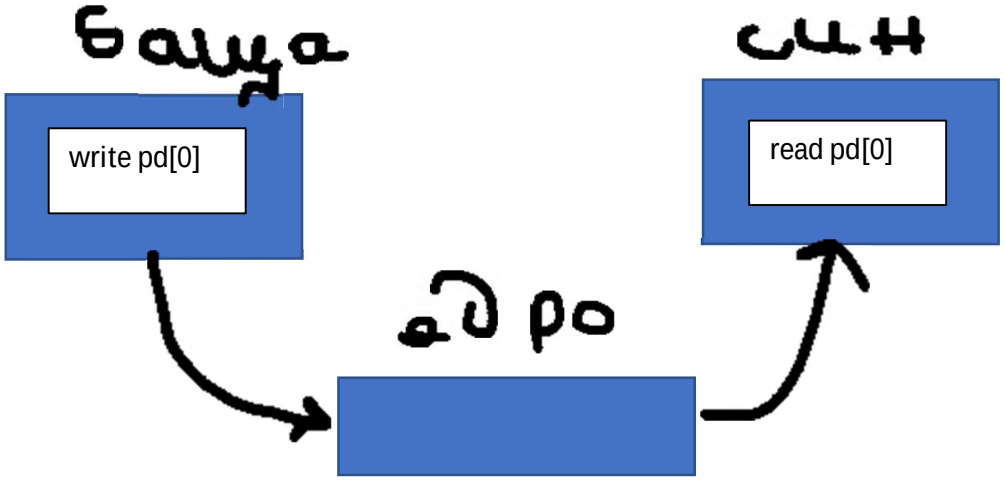
2. Ако каналът е празен, ядрото блокира процеса. Когато бъде събуден от друг

процес, изпълняващ write, той продължава както в случай 1. 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.

Затваряне на програмен канал

Файловите дескриптори, върнати от pipe, за четене и писане в канал се освобождават с примитива 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.

1. Ако при close се освободи последният файлов дескриптор за писане в канала (във всички процеси), то се събуждат всички процеси, чакащи да четат от канала, като read връща 0 (това означава край на файл).



След като бащата е затворил файловия си дескриптор за четене, а синът този за писа, получаваме еднопосочен канал за предаване на данни между 2 процеса- от бащата към сина.

```
main(void)
{
int pd[2], n;
pid_t pid;
char buff[MAXLINE];
if (pipe(pd) < 0)
err_sys_exit("pipe error");
if ((pid = fork()) < 0)
err_sys_exit("fork error");
else if (pid > 0) { /* parent */
close(pd[0]);
write(pd[1], "Hello World\n", 12);
} else { /* child */
close(pd[1]);
n = read(pd[0], buff, MAXLINE);
write(1, buff, n);
}
exit(0);
}
```

Процесът, в който работи програмата, създава програмен канал след което създава и процес-син. Бащата пише в програмния канал, а синът чете от програмния канал и извежда прочетеното на стандартния изход.

16.**Споделена памет** shm_overview: shm_open, ftruncate,mmap

Shared memory application interface – позволява на процесите да предават информация като споделят регион от памет.

```
int shm_open(const char *name, int oflag, mode_t mode);
```

Създава / отваря съществуващ shared memory object. Аналогично е на open. Фукнцията връща файлов дескриптор, които се използва при ftruncate и mmap. A shared memory object is in effect a handle which can be used by unrelated processes to mmap the same region of shared memory.

```
int ftruncate(int fd, off_t length);
```

-truncates a file to a specified length. Ако файлът е бил по-голям от от дължината, някои от данните се губят, ако е бил по-малък обаче размерът на файла се увеличава

```
void *mmap(void *addr, size_t length, int prot, int flags,
```

```
int fd, off_t offset);
```
-създава ново свързване (маппинг) във виртуалното адресно пространство на извикващия процес.

17.**Семафори** sem_overview: sem_init, sem_post, sem_wait, sem_destroy

Семафорите позволяват на процесите и нишките да синхронизират техните действия. Семафорът е цяло число, чиято стойност никога не е разрешено да падне под 0. Изпълняват се 2 операции върху семафорите:

- Увеличаване на стойността на семафора с 1 (sem_post)
- намаляване на стойността на семафора с 1(sem_wait)

Ако стойността на семафорът е 0, тогава операция от типа sem_wait ще се блокира, докато стойността не стане по-голяма от 0.
POSIX семафорите са 2 вида: именувани семафори и неименувани семафори
Именувани семафори
Един именуван семафор се идентифицира от име, което е стринг. 2 процеса могат да работят върху един и същ семафор, като подадат същото име на sem_open.
Sem_open функцията създава нов именуван семафор или отваря съществуващ вече такъв. След като семафорът е бил отворен може да се работи с него с sem_post и sem_wait. Когато процес е приключил с работата си със семафора, той може да използва sem_close, за да го затвори. След като всички процеси са приключили работата си със семафора, той може да се премахне от системата с sem_unlink.
Неименувани семафори
Неименуваният семафор няма име. Вместо това семафорът се поставя на място от паметта, което е споделено между нишки(thread-shared semaphore). Този семафор се поставя на такова място в паметта, което е споделено между нишки на процес, напр. глобална променлива. A process-shared semaphore е поставен в такъв споделен регион от памет, който е създаден чрез shmge или споделен обект(shared memory object).
Преди да бъде използван, неименуваният семафор трябва да бъде инициализира с sem_init. Работи се върху него с операциите sem_post и sem_wait. Когато повече нямаме нужда от семафор и преди паметта, в която той се намира, да е освободена, семафорът трябва да бъде унищожен с sem_destroy.

```
sem_t *sem_open(const char *name, int oflag);
```
При успех тази ф-я връща адресът на новия семафор,този адрес се ползва от други операции, свързани със семафора, при грешка връщаме SEM_FAILED.

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```
Инициализира неименуван семафор на адреса, сочен от sem. Value специфицира началната стойност на семафора. Pshared индикира дали семафорът е споделен между нишки на даден процес, или е споделен между процеси. Ако Pshared е 0 тогава процесът е споделен между нишки на даден процес и трябва да е локализиран на някой адрес, който е видим за всички нишки (напр. глобална променлива или променлива, заделена в хийпа.) Ако Pshared е различно от 0, тогава семафорът е споделен между процеси и трябва да бъде локализиран в регион на shared memory. (shm_open, mmap). Всеки процес, който може да достъпва споделена памет, може да работи върху семафора, използвайки sem_post, sem_wait. Инициализиране на вече инициализиран семафор може да резултира в неопределено поведение.

```
int sem_post(sem_t *sem);
```
Увеличава семафорът, сочен от sem. Ако стойността на семафорът стане > 0, тогава друг процес или блокирана нишка в sem_wait ще бъде събудена.

```
int sem_wait(sem_t *sem);
```
Намалява семафорът, сочен от sem. Ако стойността на семафорът е > 0, тогава намаляването се изпълнява и функцията връща резултат, иначе ако семафорът има стойност 0, тогава извикването се блокира, докато не стане възможно за него да извърши намаляването или докато не получим сигнал, за да прекъснем извикването.
```
int sem_destroy(sem_t *sem);
```

Разрушава неименуван семафор на адреса, сочен от sem. Само семафор, който е бил инициализира с sem_init трябва да бъде разрушен с sem_destroy. Разрушаване на семафор, който има други блокирани процеси или нишки довежда до недефинирано поведение.

18.**Съобщения** mq_overview: mq_open, mq_send, mq_receive

Общ поглед върху message queues(опашка на съобщенията). Message queues позволяват на процесите са обменят информация под формата на съобщения. Message queues се създават и отварят с mq_open. Тази функция връща message queue дескриптор, който се използва за рефериране на последващи извиквания в опашката. Всяка опашка има име.2 процеса могат са работят върху една и съща опашка, като им се подава същото име на mq_open. Съобщенията се предат към и от опашката чрез mq_send и mq_receive, и когато опашката вече не ни трябва може да я изтрием чрез mq_unlink. След форк детето наследява копие на message queue descriptors на родителя, и тези дескриптори сочат към същата опашка.

```
mqd_t mq_open(const char *name, int oflag);
int mq_send(mqd_t mqdes, const char *msg_ptr,
                    size_t msg_len, unsigned int msg_prio);
```

Ако опашката е вече пълна mq_send се блокира докато не се освободи достатъчно място. При успех връща 0.

```
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
                           size_t msg_len, unsigned int
*msg_prio);
```
-Премахва най-старото съощение с най-висок приоритет от опашката. Ако опашката е празна, тогава по дифолт, mq_receive блокира докато не се появи съобщение или докато извикването не е прекъснато от обработчика на сигнали. Връща броя байтове на полученото съобщение.