Файлов указател

За всеки отворен файл, ядрото записва *отместване на файла*, наричано още *отместване за запис-четене* или *указател*. Това е мястото във файла, от което започва следващото четене или запис. *Отместването* във файла се изразява като поредната позиция, изразена в байтове, относителна по отношение на началото на файла. Първият байт на файла е с отместване 0.

При отваряне на файла, *отместването* сочи в в началото на файла и автоматично се променя при всяко четене с `read()` или запис с `write()`, след което то сочи към следващия байт на файла, след последния прочетен или записан байт. Така извиквания на `read()` и `write()` се осъществяват последователно през файла.

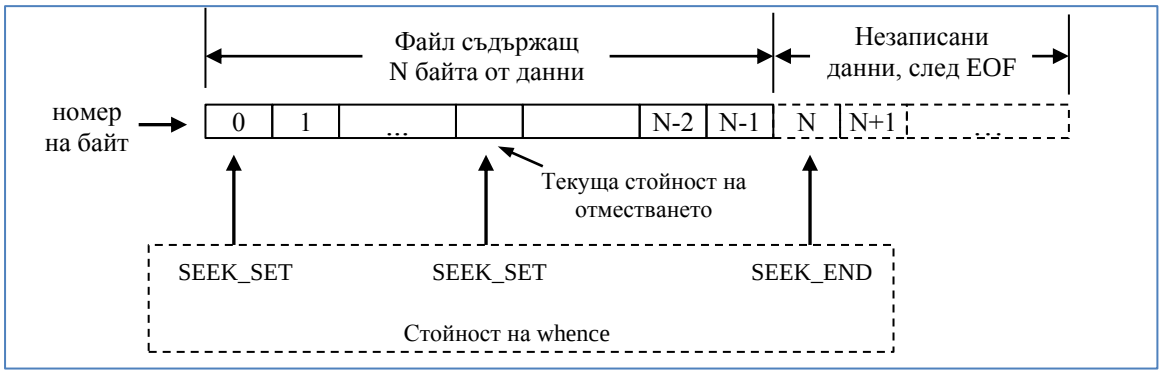
Системното извикване `lseek()` премества отместването на отворен файл, зададен с файлов дескриптор, в зависимост от стойностите на `offset` и `whence`.

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

Извикването `lseek` връща новата стойност на отместването при успех или -1 при грешка.

Аргументът `offset` задава стойност в байтове. Типът `off_t` е предефиниран целочислен, със знак. Аргументът `whence` задава базовата точка от която се интерпретира `offset` и може да бъде една от следните възможности:

- `SEEK_SET` – отместването на файла се задава по отношение на началото на файла.
- `SEEK_CUR` – отместването на файла се задава по отношение на текущата стойност на отместването на файла.
- `SEEK_END` – отместването на файла се задава по отношение на байта следващ последния байт на файла.



Интерпретиране на аргумента whence при lseek()

Ако whence е `SEEK_CUR` или `SEEK_END`, `offset` може да се зададе с отрицателна или положителна стойност, а ако е зададен `SEEK_SET`, то `offset` трябва да е с неотрицателна стойност.

Примери:

```
curr = lseek(fd, 0, SEEK_CUR); /* връща текущата позиция на отместването без промяна */
lseek(fd, 0, SEEK_SET);            /* начало на файла */
lseek(fd, 0, SEEK_END);            /* Байтът след края на файла */
lseek(fd, -1, SEEK_END);           /* Последният байт на файла */
lseek(fd, -10, SEEK_CUR);          /* Десет байта преди текущата позиция */
lseek(fd, 10000, SEEK_END);    /* 10001 байта след последния файл на файла */
```

Извикването на `lseek()` единствено променя информацията в ядрото за отместването на файла асоцииран със зададения файлов дескриптор. То не води до физически достъп до външното устройство, на което се съхранява файлът.

Извикването `lseek()` не може да се прилага към всички типове файлове. Например, не е възможно то да се прилага към именован канал, неименован канал или сокет, като в тези случай се получава грешка, а `errno` приема стойност `ESPIPE`.

***Забележка**:*
Буквата l в lseek() произлиза от това, че и аргументът offset, и върнатата стойност се представят с тип long. По-ранните имплементации на UNIX предоставят системното извикване seek(), при което тези стойности се представят с тип int.

Файлови дупки

Може да се изследва въпросът, какво се случва при преместване на отместването след края на файла и след това се изпълни входно-изходна операция. Извикването `read()` ще върне 0, показвайки, че е достигнат края на файла. Операцията запис обаче успява и тя дава възможност да се запишат значещи байтове в произволна точка след края на файла.

Мястото между предишния край на файла и новозаписаните байтове се нарича *файлова дупка*. От програмна гледна точка, байтовете в дупката съществуват и четенето от нея връща буфер от байтове със тойност `null`.

Файловите дупки не заемат място на диска. Файловите системи не алокират дискови блокове за дупките, докато не се запишат някакви данни в тях. Основното предимство на файловите дупки е, че се консумира по-малко пространство отколкото би било необходимо при действително се записване на байтове `null`.

Повечето `UNIX` файлови системи поддържат възможност за съществуване на файлови дупки, но някои други файлови системи (напр. `VFAT` на `Microsoft`) не ги поддържат. На файлова система, която не поддържа дупки, явно се записват байтове `null` във файла.

Тъй като размерът на файла обикновено се изчислява чрез разликата между най-старшия и най-младшия байт, то наличието на дупки означава, че номиналният размер на файла може да е по-голям от размера на дисковото пространство, което той заема, като в някои случаи тази разлика може да бъде значителна. Записът на байтове във вътрешността на файлова дупка намаля размера на свободното дисково пространство,

тъй като ядрото алокира блокове за запълване на дупката, въпреки че размерът на файла не се променя. Макар че рядко възникват подобни ситуации, те все пак трябва да се имат предвид.

Операции за контрол на файла `fcntl()`

Системното извикване `fcntl()` може да се използва за изпълнение на множество различни операции към отворен файл, зададен с файлов дескриптор.

```
#include <fcntl.h>
int fcntl(int fd, int cmd, ...);
```

Върнатата стойност, при успех зависи от изпълняваната операция, а при грешка е -1. Аргументът `cmd` може да задава различни операции. Третият аргумент може да бъде от различен тип, който зависи от изпълняваната операция или може да не съществува. Ядрото използва стойността на аргумента `cmd` за да определи типа данни, които да очаква като последен аргумент.

Флагове за статус на отворен файл

Една от възможните операции, които могат да се изпълняват с `fcntl()` е получаване или модифициране на режима за достъп и флаговете за статус на отворен файл. Това са стойностите, които се задават с втория аргумент на извикването `open()`. За получаване на тези настройки, за `cmd` се задава `F_GETFL`:

Пример:

```
int flags, accessMode;
flags = fcntl(fd, F_GETFL); /* третият аргумент в случая не е необходим */
```

След това по следния начин може да се провери дали файлът е отворен за синхронизиран запис:

```
if (flags & O_SYNC) {
    printf("Synchronized write allowed \n");
}
```

Проверката за правата на достъп на файл е малко по-сложна, тъй като могат да се използват комбинации с константите `O_RDONLY`, `O_WRONLY` и `O_RDWR`, съответстващи на статус флаговете на отворен файл и проверката не може да се осъществи само към един бит. Поради тази причина за тази проверка се маскира стойността на `flags` с константата `O_ACCMODE` и след това се проверява за равенство с една от константите:

```
accessMode = flags & O_ACCMODE;
if (accessMode == O_WRONLY || accessMode == O_RDWR) {
    printf("file is writable\n");
```

```
}
```

За модифициране на статус флаговете на отворен файл, `fcntl()` може да се използва с командата `F_SETFL`. Флаговете които могат да се модифицират са `O_APPEND`, `O_NONBLOCK`, `O_NOATIME`, `O_ASYNC` и `O_DIRECT`. При опит за модифициране на други флагове, операцията се игнорира. Само някои `UNIX` имплементации дават възможност `fcntl()` да модифицира и други флагове, като `O_SYNC`.

Модифицирането на статус флаговете на отворен файл с `fcntl()` най-често се налага в следните случаи:

- Ако файлът не е отворен от текущо изпълняваната програма, така че тя няма достъп до флаговете в извикването `open()`. Например файлът може да е един от трите стандартни дескриптори, отворени преди стартиране на програмата, съответстващи на стандартния вход, изход и изход за грешки.
- В случаи при които файловият дескриптор е получен от системно извикване различно от `open()`. Примери за такива извиквания са `pipe()`, което създава неименован канал и връща два файлови дескриптора, сочещи към всеки от краищата на канала, или `socket()`, което създава сокет и връща файлов дескриптор сочещ сокета.

За модифициране на статус флаговете на отворен файл, първо се използва `fcntl()`, за да се извлекат стойностите на необходимите флагове, след това се модифицират необходимите битове и накрая се прави следващо извикване на `fcntl()` за обновяване на стойностите на флаговете в ядрото. Например, за разрешаване на флага `O_APPEND`, може да се запише следният фрагмент:

```
int flags = fcntl(fd, F_GETFL);
flags |= O_APPEND;

fcntl(fd, F_SETFL, flags);
```

Взаимовръзка между файлови дескриптори и отворени файлове

На пръв поглед връзката между файлов дескриптор и отворен файл изглежда едно към едно. Това обаче не е така. Възможно е да съществуват множество различни дескриптори сочещи към един и същ отворен файл. Тези файлови дескриптори могат да са отворени в един и същ процес или в различни процеси.

За изясняване на взаимовръзката между дескриптори и отворени файлове е необходимо да се разгледат следните три даннови таблици поддържани в ядрото:

- Таблица на отворените файлове за процес (ТОФП).

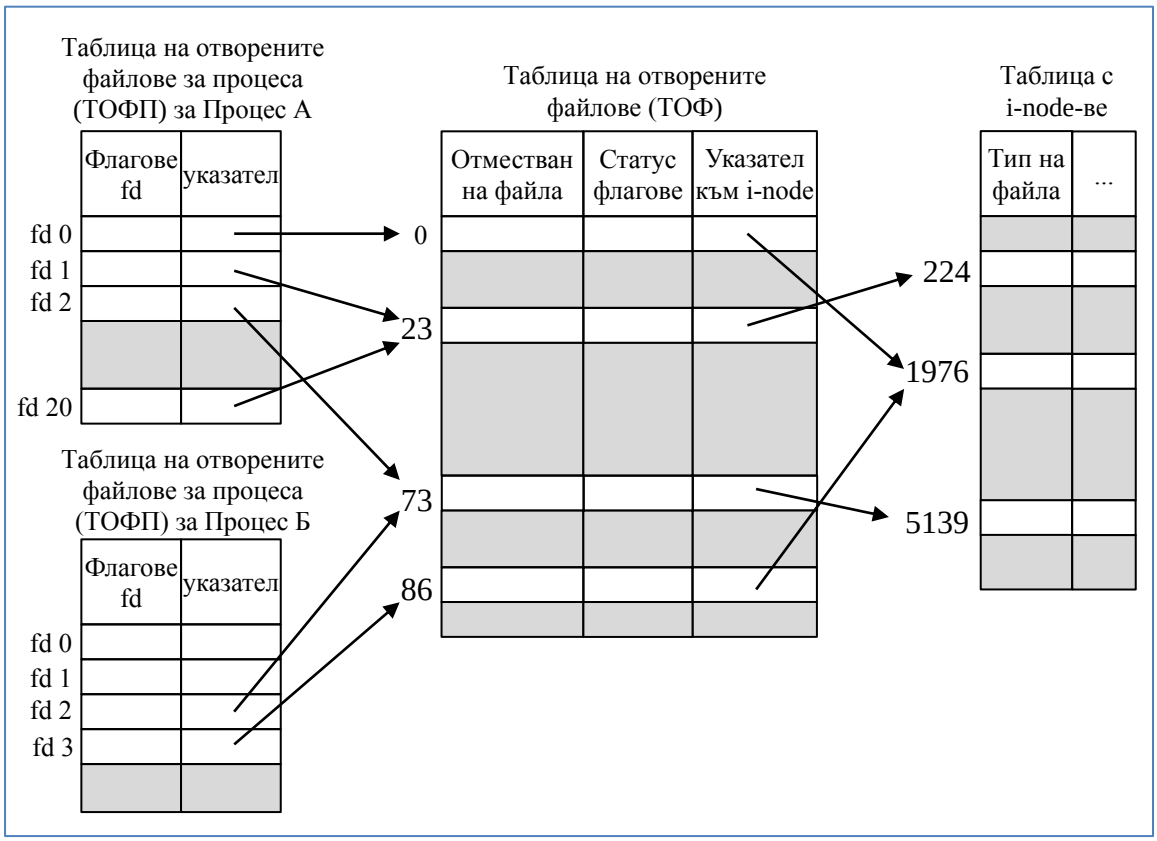
  За всеки процес ядрото поддържа таблица с дескриптори на отворените файлове. Всеки запис в тази таблица съдържа информация за един файлов дескриптор, включително:

- Множество от флагове контролиращи операцията с файловия дескриптор (има само един такъв флаг, флагът `close-and-exec`, който ще бъде разгледан по-късно;
- Референция към запис в ТОФ.

- Системна таблица на отворените файлове (ТОФ).
  Ядрото поддържа системна таблица на всички отворени файлове. Записите на тази таблица се наричат понякога манипулатори на отворените файлове (open file handlers). Един запис в тази таблица съдържа следната информация свързана с отворения файл:
  - Текущото отместване на файла, което се променя неявно при изпълнение на `read()` и `write()` или явно с `lseek()`;
  - Статус флаговете зададени при отваряне на файла, например с втория аргумент на `open()`;
  - Режимът на достъп до файла: само за четене, само за запис или за четене и запис;
  - Указател към запис в таблицата с i-node-ве;
  - Друга информация.
- Таблица с "i-node"-ве на файловата система.
  Всяка файлова система има таблица от i-node-ве за всички файлове намиращи се във файловата система. Структурата i-node и файловите системи като цяло ще бъдат разглеждани по-късно. Засега може да се каже, че всеки i-node за файл съдържа следната информация:
  - Тип на файла (напр. обикновен файл, сокет, именован канал и т.н.)
  - Права на достъп до файла;
  - Свойства на файла, например размер, времена на извършване на различни типове операции и т.н.;
  - Друга информация.

***Забележка**:*
Трябва да се има предвид, че има разлика между представянето на i-node в паметта и върху диска. Представянето му върху диска записва атрибути на файла, като тип на файла, права и т.н. При достъп до файла, се създава копие от i-node в паметта и в него се записва допълнителна информация, като брои на записите в ТОФ сочещи към този i-node, както идентификаторите на устройството от което е копиран i-node.

Следната фигура показва връзката между трите таблици, за два процеса.



Връзка между ТОФ, ТОФП и таблицата с i-node-ве

В процес А, дескриптори 1 и 20 сочат към един и същ запис в ТОФ. Такава ситуация може да възникне в резултат на извикване `dup()`, `dup2()` или `fcntl()`. Дескриптор 2 на процес А и дескриптор 2 на процес Б сочат към един и същ запис в ТОФ. Този сценарии може да се получи след извикване на `fork()` в някой от двата процеса, при което процес А става родител на процес Б или обратно.

Дескриптор 0 на процес А и дескриптор 3 на процес Б сочат към различни записи в ТОФ, но тези записи сочат към един и същ i-node, този с номер 1976, т.е. към същия файл. Това се получава когато всеки от двата процеса извиква `open()` към един и същ файл. Подобна ситуация се получава и в рамките на един процес, когато той отваря един и същ файл два пъти.

Тези ситуации могат да се обобщят по следния начин.

- Два различни файлови дескриптора, т.е. записа в ТОФП, които сочат към един и същ запис в ТОФ имат една и съща стойност за отместване на файла. Така ако отместването на файла се променя чрез един от дескрипторите, в резултат на `read()`, `write()` или `lseek()`, тази промяна е видима през другия файлов

дескриптор. Това е сила и когато двата файлови дескриптора са в един процес, и когато са в различни процеси.

- По същия начин стоят нещата и при връщане или промяна на статус флаговете на отворен файл, например `O_APPEND`, `O_NONBLOCK`, `O_ASYNC` и т.н., чрез изполване на `fcntl()` операциите `F_GETFL` и `F_SETFL`.
- Флаговете на файловите дескриптори обаче, т.е. на записите в ТОФП (например флагът `close-on-exec`) са специфични и частни за процеса и файловия дескриптор. Модифицирането на тези флагове не променя другите файлови дескриптори в същия процес или в другите процеси.