Дублиране на файлови дескриптори

Със следния синтаксис от командния интерпретатор `2>&1` се задава пренасочване на стандартния изход за грешки (файлов дескриптор с номер 2) към същото място, към което сочи стандартният изход (файлов дескриптор 1). След това и стандартният изход и стандартният изход за грешки могат да се насочат към файла `results.log`:

```
./myscript > results.log 2>&1
```

Командният интерпретатор извършва пренасочването на стандартния изход за грешки, чрез дублиране на дескриптор 2, така че той да сочи към същия запис на отворен файл в ТОФ, към който сочи дескриптор 1 (по същия начин както дескриптори 1 и 20 на процес А сочат към един и същ запис в ТОФ в схемата с таблиците за отворените файлове в ядрото, на фигурата по-горе). Този ефект може да се постигне със системните извиквания `dup()` и `dup2()`.

За постигане на този резултат не е достатъчно да се отвори файлът results.log два пъти: веднъж за дескриптор 1 и веднъж за дескриптор 2. Една от причините за това е, че двата файлови дескриптора няма да споделят указателя за отместване на файла. Освен това файлът може да не е дисков файл. Анализирайте например следната команда:

```
./myscript 2>&1 | less
```

Извикването `dup()` получава като параметър отворен файлов дескриптор `oldfd` и връща като резултат нов дескриптор, който сочи към същия запис в ТОФ, към който сочи `oldfd`. Новият дескриптор е първия неизползван дескриптор с най-нисък номер в ТОФП. При грешка, извикването връща -1.

```
#include <unistd.h>
int dup(int oldfd);
```

Нека е извършено следното извикване:

```
newfd = dup(1);
```

При нормална ситуация, в която командният интерпретатор е отворил файловите дескриптори 0, 1 и 2 и не се използват други дескриптори в процеса, `dup()` ще създаде дубликат на дескриптор 1, използвайки дескриптор с номер 3 в ТОФП.

При необходимост от дублиране на дескриптор 1 в дескриптор 2, може да се използва следната техника:

```
close(2);          /* освобождава дескриптор с номер 2 */
newfd = dup(1);    /* дублира дескриптор 1 в дескриптор 2 */
```

Този код е ефективен само ако дескриптор 0 е отворен. За да се гарантира че се прави дублиране в необходимият файлов дескриптор, може да се използва `dup2()`.

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

Системното извикване `dup2()` дублира файловия дескриптор зададен чрез `oldfd`, използвайки номера на дескриптора зададен в `newfd` и при успех връща като резултат `newfd`, а при грешка -1. Ако файловият дескриптор зададен в `newfd` вече е отворен, `dup2()` първо го затваря.

***Забележка**: Ако по време на затварянето на newfd възникне грешка, тя се игнорира. Затова по-безопасна практика е явно да се изпълни close() за newfd ако е отворен преди извикване на dup2().*

Така дублирането на дескриптор 1 в дескриптор 2, може да се извърши по следния начин:

```
dup2(1, 2);
```

Ако `oldfd` не е валиден файлов дескриптор, то `dup2()` връща -1 и записва в `errno` грешка `EBADF`, като при това `newfd` не се затваря. Ако `oldfd` е валиден файлов дескриптор и той съвпада с `newfd`, то `dup2()` не затваря `newfd`, а връща `newfd` като резултат.

Друг начин за дублиране на файлови дескриптори е чрез извикването `fcntl()` с операция `F_DUPFD`:

```
newfd = fcntl(oldfd, F_DUPFD, startfd);
```

Това извикване дублира `oldfd` в най-малкия неизползван номер на файлов дескриптор, по-голям от или равен на `startfd`.

В повечето случаи, извикванията `dup()` и `dup2()`могат да се представят чрез комбинация от `close()` и `fcntl()`, въпреки че кодът с първите извиквания е по-кратък.

***Забележка**: Трябва да се има предвид, че някои от грешките на dup2() и fcntl(), които се записват в errno се различават, както е описано в помощните страници.*

Тъй като сочат към един общ запис в ТОФ, дублираните файлови дескриптори споделят файловото отместване и статус флагове. Новият файлов дескриптор обаче има негово собствено множество от флагове на файловия дескриптор в ТОФП и неговият `close-on-exec` флаг (`FD_CLOEXEC`) винаги е свален. Интерфейсът, който се описва

по-долу явно разрешава контрол върху новия файлов дескрипторен close-on-exec флаг.

Системното извикване dup3() извършва същата операция, както dup2(), но има допълнителен трети аргумент flags, който е битова маска, която модифицира поведението на системното извикване.

```
#define _GNU_SOURCE
#include <unistd.h>
int dup3(int oldfd, int newfd, int flags);
```

dup3() текущо поддържа един флаг, O_CLOEXEC, който указва на ядрото да вдигне флага close-on-exec (FD_CLOEXEC) за новия файлов дескриптор. Този флаг е същия, както O_CLOEXEC флагът при извикванена open() и се описва по-долу.

Извикването dup3() е характерно за Linux и е въведено в по-късните му версии. От по-късните му версии се поддържа и допълнителна fcntl() операция за дублиране на файлови дескриптори: F_DUPFD_CLOEXEC. Тази операция изпълнява същата работа, както F_DUPFD, но допълнително задава флага close-on-exec (FD_CLOEXEC) за новия файлов дескриптор. F_DUPFD_CLOEXEC е специфицирано в по-новите UNIX стандарти.

Флаг close-on-exec (FD_CLOEXEC)

Понякога се налага да се затворят определени файлови дескриптори преди изпълнение на exec(). Това е особено важно ако с exec се изпълнява програма писана от някой друг или която няма нужда от дескрипторите за отворените файлове от текущата програма. В такива случаи с цел безопасност е добре да се затворят ненужните файлови дескриптори преди зареждане на новата програма. Това може да се направи с извикване на close() за всички такива файлови дескриптори, но това не винаги е възможно или има определени недостатъци. Такива проблеми могат да настъпят например ако файлът е отворен от библиотечна функция и главната програма няма контрол върху затварянето му. Поради това принципно библиотечните функции винаги трябва да задават флага close-on-exec, използвайки техниката по-долу, за всички отворени от тях файлове. Друг проблем е, че ако exec() извикването пропадне поради някаква причина, то отворените файлове вероятно трябва да останат отворени, при това със запазване на текущата стойност на отместването във файла и статус флаговете.

Ако флагът close-on-exec е зададен, то файловият дескриптор автоматично се затваря по време на успешното изпълнение на exec(), но остава отворен ако exec() пропадне. Флагът close-on-exec за файлов дескриптор може да се достъпи със системното извикване fcntl(). Операцията F_GETFD връща копие на флаговете на файловия дескриптор:

```
int flags;
flags = fcntl(fd, F_GETFD);
```

След получаване на тези флагове, флагът `FD_CLOEXEC` може да се модифицира и да се използва втори `fcntl()`, с операцията `F_SETFD` за обновяване на флаговете:

```
flags |= FD_CLOEXEC;
fcntl(fd, F_SETFD, flags);
```

***Забележка**:*
FD_CLOEXEC текущо е само един бит, използван във флаговете на файловия дескриптор. Този бит съответства на стойност 1. В по-старите програми, може да се види задаване на флага close-on-exec чрез извикването fcntl(fd, F_SETFD, 1), поради това че няма други битове, които могат да бъдат повлияни от тази операция. Теоретично това не винаги може да е така, тъй като в бъдеще, някои UNIX системи могат да имплементират допълнителни флагови битове, така че трябва да се използв атехниката показана по-горе.

Много имплементации на UNIX, включително Linux, дават възможност за модифициране на флага close-on-exec и с използване на нестандартизираното извикване ioctl().

При използване на `dup()`, `dup2()` и `fcntl()` за създаване на дубликат на файлов дескриптор, флагът `close-on-exec` е свален за дублирания дескриптор.
Програмният фрагмент по-долу демонстрира работата с този флаг. В зависимост от наличието на аргумент от командния ред, който може да е някакъв произволен низ, тази програма първо задава вдигане на флага за стандартния изход и след това изпълнява програмата `ls`.

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

int main(int argc, char *argv[]) {
      int flags;
      if (argc > 1) {
            flags = fcntl(STDOUT_FILENO, F_GETFD);
            if (flags == -1)  {
                  perror("fcntl - F_GETFD");
                  exit(EXIT_FAILURE);
            }
            flags |= FD_CLOEXEC;
            if (fcntl(STDOUT_FILENO, F_SETFD, flags) == -1) {
                  perror("fcntl - F_SETFD");
                  exit(EXIT_FAILURE);
            }
      }
      execlp("ls", "ls", "-l", argv[1], NULL);
      perror("execlp");
      exit(EXIT_FAILURE);
```

```
}
```

Тук е показано изпълнението на програмата:

```
$ ./closeonexec
      -rwxr-xr-x 1 mtk users 28098 Jun 15 13:59 closeonexec
$ ./closeonexec n
      ls: write error: Bad file descriptor
```

Файлови дескриптори и exec()

По подразбиране всички файлови дескриптори отворени от програма изпълняваща `exec()` остават отворени и са налични за използване от новата програма. Обикновено това дава определени удобства, тъй като извикващата програма може да е отворила определени специфични файлове и техните дескриптори автоматично стават достъпни за новата програма, без да има нужда от отваряне на файловете в новата програма и без да се знаят техните имена.

Командният интерпретатор използва този факт при пренасочването на входа или изхода за изпълняваните от него програми. Например, ако се въведеа следната команда:

```
ls /tmp > dir.txt
```

интерпретаторът изпълнява следните стъпки за изпълнение на тази команда:

1. Изпълнява се `fork()` за създаване на процес наследник, който да изпълни командата.
2. Процесът наследник отваря файла `dir.txt` за запис, използвайки файлов дескриптор с номер 1 (стандартния изход). Това може да се извърши по един от следните начини:
   a. Процесът наследник затваря дескриптор с номер 1 (`STDOUT_FILENO`) и след това отваря файла `dir.txt`. Тъй като `open()` използва първия свободен файлов дескриптор с най-малък номер, а стандартният вход (дескриптор с номер 0) остава отворен, файлът ще бъде отворен с дескриптор 1.
   b. Шелът отваря `dir.txt` за запис. Ако този дескриптор не е с номер 1, съответстващ на стандартния изход, процесът изпълнява `dup2()`, за да направи стандартния изход дубликат на дескриптора за `dir.txt`, след което затваря дескриптора за `dir.txt`, тъй като той вече не е необходим. Тази техника е по-безопасна от предишната, тъй като не разчита на това, че най-малкият номер на дескриптор е отворен. Това може да се изпълни например по следния начин:

```
fd = open("dir.txt", 0755);
if (fd != STDOUT_FILENO) {
      dup2(fd, STDOUT_FILENO);
      close(fd);
}
```

Процесът наследник изпълнява програмата `ls`. Програмата `ls` записва изхода си към стандартния изход, който съвпада с файла `dir.txt`.

***Забележка**:*
Т. нар. вградени команди, се изпълняват в процеса на интерпретатора, без създаване на нов процес. Такива команди трябва да се третират по по-различен начин по отношение на пренасочването. Такива команди се наричат вградени и те се изпълняват без създаване на нов процес поради една от следните две причини:
1) Ефективност. Някой често използвани команди в командния интерпретатор, като pwd, echo и test са сравнително прости, така че те могат по-ефективно да се имплементират в процеса на самия интерпретатор, без създаване на нов процес.
2) За получаване на определени странични ефекти в процеса на командния интерпретатор. Такива команди трябва да имат определени странични ефекти върху самия процес на интерпретатора, т.е. те трябва да променят информацията съхранена в него или да модифицират определени негови атрибути. Например командата cd трбва да промени работната директория на процеса на командния интерпретатор и не може да се изпълни от отделен процес. Други примери за команди, които са вградени поради техния страничен ефект са exec, exit, read, set, source, ulimit, umask, wait и командите за управление на задачите: jobs, fg и bg.