Funkcje wejścia/wyjścia niskiego poziomu

Witold Paluszyński
witold.paluszynski@pwr.edu.pl
http://sequoia.iiar.pwr.edu.pl/~witold/

Niniejszy dokument zawiera materiały do wykładu na temat funkcji wejścia/wyjścia niskiego poziomu w systemie Unix. Jest on udostępniony pod warunkiem wykorzystania wyłącznie do własnych, prywatnych potrzeb i może być kopiowany wyłącznie w całości, razem z niniejszą stroną tytułową.

Przykład: kopiowanie plików (5)

```
#include <stdio.h>        /* wersja 5: operacje IO */
#include <fcntl.h>        /* niskiego poziomu      */

main(int argc, char *argv[])
{
  char buf[1024];
  int fd1, fd2, fdin;

  fd1 = open(argv[1], O_RDONLY);
  fd2 = open(argv[2], O_WRONLY|O_CREAT, 0644);
  while ((fdin = read(fd1, buf, sizeof buf)) > 0)
    write(fd2, buf, fdin);
  exit(0);
}
```

- Funkcje `open`, `read`, `write`, zwane wbudowanym systemem wejścia/wyjścia, lub systemem I/O niskiego poziomu, są głównym mechanizmem operacji na plikach i urządzeniach w Unix'ie.
- Odwołują się one do otwartych plików przez numery deskryptorów.

Deskryptory plików

- Deskryptory są małymi liczbami przyporządkowanymi kolejno otwieranym plikom. Program ma gwarancję uzyskania najniższego wolnego deskryptora. Deskryptory 0,1,2 reprezentują automatycznie otwierane pliki stdin, stdout, i stderr.
- Numery deskryptorów otwartych plików są lokalne dla danego procesu. Jeśli inny proces otworzy ten sam plik to otrzyma własny (choć być może ten sam) numer deskryptora, z którym związany jest kursor pliku wskazujący pozycję wykonywanych operacji zapisu/odczytu.
- Operacje odczytu pliku otwartego jednocześnie przez dwa procesy mogą poprawnie przebiegać niezależnie od siebie.
- Jeśli dwa procesy jednocześnie otworzą ten sam plik do zapisu (lub zapisu i odczytu), to ich operacje zapisu będą się na siebie nakładały, jedne dane nadpiszą drugie, i otrzymamy w pliku tzw. sieczkę.
- Jednak dwa procesy mogą współdzielić jeden deskryptor otwartego pliku, i wtedy ich operacje będą się ze sobą przeplatać. Jeśli nie będziemy tego przeplatania kontrolować to również możemy otrzymać sieczkę, aczkolwiek w tym przypadku dane nie zostaną nadpisane.
- W każdym przypadku do synchronizowania operacji wejścia/wyjścia na plikach możemy użyć blokad.

Funkcje niskiego poziomu a biblioteka `stdio`

Funkcje I/O niskiego poziomu są innym, bardziej surowym, sposobem wykonywania operacji wejścia/wyjścia na plikach niż biblioteka `stdio`.

- Operacje I/O niskiego poziomu nie mają dostępu do struktur danych biblioteki `stdio` i w odniesieniu do danego otwartego pliku nie można ich mieszać z operacjami na poziomie tej biblioteki.
- Jednak w czasie pracy z biblioteką `stdio` funkcje niskiego poziomu też pracują (na niższym poziomie), i można korzystać z niektórych z nich, jeśli się wie co się robi.
- Na przykład, numer deskryptora pliku można uzyskać z file pointera biblioteki `stdio` (`fileno(fp)`), ale nie na odwrót (dlaczego?).

Szczegóły działania funkcji `read, write`

- Przy czytaniu funkcją `read` liczba znaków przeczytanych może nie być równa liczbie znaków żądanych ponieważ w pliku może nie być tylu znaków.
  - W szczególności przy czytaniu z terminala funkcja `read` normalnie czyta tylko do końca linijki.
  - Wartość 0 oznacza brak danych do czytania z pliku (koniec pliku), co jednak nie oznacza, że przy kolejnej próbie czytania jakieś dane się nie pojawią (inny proces może zapisać coś do pliku), zatem nie jest to błąd.
  - Wartość ujemna oznacza błąd.
- Przy pisaniu funkcją `write` wartość zwrócona jest normalnie równa liczbie znaków żądanych do zapisu; jeśli nie to wystąpił jakiś błąd.
- Drugi parametr funkcji `open` może przyjmować trzy wartości zadane następującymi stałymi: `O_RDONLY` (tylko czytanie), `O_WRONLY` (tylko pisanie), lub `O_RDWR` (czytanie i pisanie). Poza tym można dodać do tej wartości modyfikatory (bitowo), które zmieniają sposób działania operacji na pliku.
- Ostatni parametr funkcji `open` zadaje 9 bitów praw dostępu dla tworzonego pliku (np. ósemkowo 0755).

Dygresja: prawa dostępu do plików

Nadawanie i sprawdzanie praw dostępu:

```
% chmod u=rw,g=r,o=x moj_program
% ls -l moj_program
-rw-r----x   1 witold grupa ...    moj_program
```

⇒ Zauważmy, że użytkownik i członkowie jego grupy nadal mają prawo wykonywania programu.

Kodowanie praw dostępu liczbą oktalną (`read=4`, `write=2`, `execute=1`):

```
% chmod 751 moj_program
% ls -l moj_program
-rwxr-x--x   1 witold grupa ...    moj_program
% umask 17
% touch nowy_plik
% ls -l nowy_plik
-rw-rw----   1 witold grupa ...    nowy_plik
```

⇒ Przy tworzeniu zwykłych plików bit prawa wykonywania jest automatycznie kasowany.

W dodatku do powyższych 9 bitów praw dostępu pliki mają jeszcze 3 inne bity, tzw. suid, sgid, i *sticky bit* (patrz `man -s 2 chmod`):

```
% chmod u+s moj_program; ls -l moj_program
-rwsr-x--x   1 witold grupa ...    moj_program
% chmod g+s moj_program; ls -l moj_program
-rwsr-s--x   1 witold grupa ...    moj_program
% chmod u+t moj_program; ls -l moj_program
-rwsr-s--t   1 witold grupa ...    moj_program
```

⇒ Ostatnią operację zwykle może wykonać tylko użytkownik o numerze 0.

Te dodatkowe bity również można zakodować ósemkowo, na najstarszej pozycji:

```
% chmod 7751 moj_program; ls -l moj_program
-rwsr-s--t   1 witold grupa ...    moj_program
% chmod 7640 moj_program; ls -l moj_program
-rwSr-l--T   1 witold grupa ...    moj_program
```

⇒ Znaczenie tych bitów jest zależne od tego czy plik ma jednocześnie ustawiony bit x na konkretnej pozycji.

Przykład: cierpliwe czytanie

Próba dalszego czytania po osiągnięciu końca pliku (może należałoby je nazwać uporczywym?):

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

main(int argc, char *argv[])
{
  char buf[1024];
  int fd, fdin;

  fd = open(argv[1], O_RDONLY);
  for (;;) {
    while ((fdin = read(fd, buf, sizeof buf)) > 0)
      write(1, buf, fdin);
    sleep(1);
  }
}
```

Można byłoby zadać sobie pytanie o inne operacje możliwe do wykonania na plikach, wykraczające poza zwykłe sekwencyjne czytanie i pisanie, np.:

- Gdybyśmy chcieli pominąć istniejącą już część pliku i — jakby niecierpliwie — przejść na koniec aby tam czekać na dalszy ciąg pliku, to czy jest to możliwe?
- Cierpliwe czytanie zakłada, że inny proces (inne procesy) mogą dopisywać dane do naszego pliku w trybie `O_APPEND`. Jednak gdyby taki inny proces chciał rozpocząć pisanie w trybie `O_RDWR` na czytanym przez nas cierpliwie pliku, to nasz program nie zauważy tego, podobnie jak ten inny proces; jak temu zaradzić?
- Gdyby z kolei inny proces nadpisał czytany przez nas cierpliwie plik, skracając go, to czy jest sposób by się o tym dowiedzieć?

Operacja seek — przesuwanie kursora pliku

```
lseek(fd, 0L, SEEK_END);          /* ustaw na koncu pliku */
lseek(fd, 0L, SEEK_SET);          /* przewin do poczatku  */
poz=lseek(fd, 0L, SEEK_CUR);      /* tylko podaj pozycje  */
```

Przesuwanie kursora pliku funkcją `lseek` odbywa się bez czytania danych, transferów z dysku, itp.

Wywołanie `lseek(fd,0,SEEK_CUR)` pozwala obliczyć aktualną bezwzględną pozycję kursora w pliku. Równoważnie można to osiągnąć funkcją `tell(fd)`.

Poza zwykłym przesuwaniem kursora pliku w ramach jego rozmiaru można ustawić pozycję w pliku poza jego końcem. Nie można z takiej pozycji nic czytać, można tam jednak pisać. Powoduje to zwiększenie rozmiaru pliku.

W tym przypadku tworzy się plik „rzadki" (ang. *sparse*). Taki plik zajmuje na dysku tylko tyle miejsca ile potrzeba na dane rzeczywiście zapisane.
W przypadku próby odczytu danych z niezapisanych obszarów „dziur" dostajemy zera. Taki odczyt może jednak spowodować przydzielenie bloków dyskowych dla dziur (zależy to od implementacji systemu plików).

Skracanie plików

W niektórych przypadkach chcemy uzyskać skrócenie istniejącego pliku otwartego do zapisu, np. w trybie `O_RDWR`. Umożliwia to funkcja `ftruncate`:

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

int truncate(const char *path, off_t length);
int ftruncate(int fildes, off_t length);
```

Blokowanie plików i rekordów

Funkcja `lockf` pozwala zakładać na plikach otwartych do zapisu (`O_WRONLY` lub `O_RDWR`) blokady wykluczające jednoczesny dostęp. Blokady zakłada się na sekcję pliku, definiowaną jako określona liczba bajtów poza aktualną pozycję kursora.

```
lockf(fd, F_LOCK, 0);       /* blok.od biez.do konca pliku */
spr=lockf(fd, F_TEST, 512); /* sprawdz. istnienia blokady  */
lockf(fd, F_TLOCK, 64);     /* jednocz. sprawdz. i blokada */
lockf(fd, F_ULOCK, 1000);   /* zdjecie blokady             */
```

Próba założenia blokady na sekcję, której jakikolwiek fragment jest już zablokowany przez inny proces, powoduje oczekiwanie (`F_LOCK`), lub powrót z błędem (`F_TLOCK`).

Blokady mają charakter addytywny, tzn. dwie blokady założone na zachodzące na siebie fragmenty pliku dają ten sam efekt, co jedna blokada założona na obszar łączny. Podobnie, blokada może być zdejmowana po kawałku, a kilka blokad może być zdjętych jednym wywołaniem obejmującym większy zakres.

Blokowanie rekordów do zapisu i odczytu

Blokady zakładane funkcją `lockf` pozwalają na poprawną współpracę wielu procesów operujących na pliku. Jednak powodują one szeregowanie operacji w różnych procesach, które normalnie wykonywane byłyby równolegle. Może to powodować blokowanie (oczekiwanie), i wydłużenie czasu działania.

Jeśli wiele procesów usiłuje wpisać informacje do tej samej sekcji pliku, to trudno jest wykonać to inaczej niż sekwencyjnie, a więc blokady i oczekiwanie są nieuniknione. Jednak może się zdarzyć, że wiele procesów chce tylko przeczytać fragment pliku. Blokada jest niezbędna, aby zapewnić, że inny proces nie rozpocznie pisania w tym samym czasie, powodując potencjalnie błędne odczyty. Jeśli operacji odczytu jest znacznie więcej niż zapisu, to można całość znacznie usprawnić stosując dwa rodzaje blokad: blokady zapisu, które są blokadami pełnej wyłączności, nie pozwalające na jednoczesne istnienie żadnej innej blokady, oraz blokady czytania, których jednocześnie może istnieć wiele na jednej sekcji pliku, i które jedynie uniemożliwiają równoczesne założenie blokady zapisu.

Funkcja `lockf` jest uproszczonym interfejsem blokowania, tworzącym jedynie blokady wyłączności (zapisu). Blokady odczytu można zakładać funkcją `fcntl`, która jednocześnie daje najbardziej uniwersalny interfejs blokowania rekordów.

Zakładanie blokad funkcją `fcntl`

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

read_lock(int fd)   /* a shared lock on an entire file */
    { fcntl(fd, F_SETLKW, file_lock(F_RDLCK, SEEK_SET)); }

write_lock(int fd)  /* an exclusive lock on an entire file */
    { fcntl(fd, F_SETLKW, file_lock(F_WRLCK, SEEK_SET)); }

append_lock(int fd) /* a lock on the _end_ of a file */
    { fcntl(fd, F_SETLKW, file_lock(F_WRLCK, SEEK_END)); }

struct flock* file_lock(short type, short whence)
{
    static struct flock ret;

    ret.l_type = type;
    ret.l_start = 0;
    ret.l_whence = whence;
    ret.l_len = 0;
    ret.l_pid = getpid();
    return &ret;
}
```

Blokady dobrowolne i wymuszane

Blokady zakładane na pliki, lub ich części, normalnie mają charakter dobrowolny (ang. *advisory locks*). Oznacza to, że blokady nie uniemożliwiają innym procesom dostępu do zablokowanej części pliku (np. zapisu). Procesy będą poprawnie współpracować z pomocą mechanizmu blokad jeśli będą sprawdzać istnienie blokad i przestrzegać ich semantyki, tzn. czekać na zwolnienie blokady lub zasygnalizują użytkownikowi niemożność wykonania operacji.

Wiele współczesnych systemów wspierają dodatkowo blokady wymuszane (ang. *mandatory locks*). W takich przypadkach próba dostępu przez inny proces do zablokowanego pliku nie udaje się. Domyślnie blokady są dobrowolne, i aby je wymusić trzeba nadać plikowi specjalne prawa dostępu.

```
shasta-798> touch dane
shasta-803> chmod 2640 dane
shasta-804> ls -l dane
-rw-r-l---   1 witold   gurus          0 Dec 29 15:53 dane
```

Blokady wymuszane nie są lepsze niż dobrowolne. Dla dobrze napisanego systemu oba rodzaje blokad są równoważne, ale źle napisanej aplikacji wymuszane blokady zwykle nie uratują.

Własności blokad

Blokady są własnością danego procesu. Tylko ten sam proces może zdjąć założoną przez siebie blokadę. Może również założyć blokadę zachodzącą na inną blokadę założoną wcześniej. Dotyczy to wszystkich wątków danego procesu.

Blokady są usuwane w momencie zamknięcia pliku. Blokady, których proces nie usunął, są usuwane automatycznie w chwili poprawnego zakończenia procesu. System usuwa również blokady pozostawione przez niepoprawnie zakończone procesy, aczkolwiek może to być wykonane z opóźnieniem.

Blokady są cechą plików, a nie deskryptorów plików. Jeśli proces utworzy inny deskryptor odwołujący się do tego samego pliku, to uzyska te same blokady. Zamknięcie dowolnego z deskryptorów danego pliku powoduje usunięcie blokad.

System Unix sprawdza w chwili zakładania blokady, czy nie spowoduje ona „zakleszczenia się" (ang. *deadlock*) z innym procesem, który mógłby być zawieszony w oczekiwaniu na blokadę założoną wcześniej przez dany proces. (To sprawdzanie nie zapobiega jednak wszystkim możliwościom „zakleszczenia").

Funkcja `flock`

Poza funkcjami `lockf` i `fcntl` tworzenie i sprawdzanie blokad jest jeszcze możliwe za pomocą funkcji `flock`. Umożliwia ona tworzenie blokad zapisu (nazywanych przez siebie blokadami wyłączności `LOCK_EX`), oraz odczytu (zwanych blokadami współdzielonymi `LOCK_SH`). Jednak funkcja `flock` nie operuje na sekcjach plików, tylko blokuje całe pliki.

Funkcja `flock` pochodzi z rodziny BSD systemu Unixa. Jednak nie została ona uwzględniona w standardzie POSIX definiującym interfejs funkcji systemowych Unixa, zatem nie należy jej normalnie stosować w przenośnych aplikacjach. Co prawda, funkcja istnieje w większości współczesnych Unixów dla podtrzymania wstecznej kompatybilności, ale np. w obecnej edycji systemu Solaris (10) funkcja ta nie jest obsługiwana, wbrew twierdzeniem podręcznika `man flock`.

Blokady zakładane funkcją `flock` mogą być zgodne z blokadami zakładanymi funkcjami `lockf/fcntl`, albo mogą być różnymi rodzajami blokad, w zależności od systemu. Najbezpieczniej jest operować na blokadach wyłącznie funkcją `fcntl`, ponieważ tylko ta funkcja jest objęta standardem POSIX.