11. Zarządzanie plikami

Wstęp

Trwałe przechowywanie i udostępnianie danych należy do podstawowych i najbardziej widocznych zadań systemu operacyjnego. Realizując to zadanie, większość systemów posługuje się pojęciem pliku jako logicznej jednostki przechowywania informacji.

Wykład 11 opisuje sposób zarządzania plikami realizowany przez jądro systemu Linux. Przedstawiamy warstwową strukturę systemu plików i omawiamy przeznaczenie poszczególnych warstw. Opisujemy sposób reprezentacji różnych typów plików oraz organizację struktury katalogowej. Omawiamy również interfejs funkcji systemowych umożliwiający operacje na plikach i katalogach.

11.1. Struktura systemu plików

System plików tworzy mechanizm bezpośredniego przechowywania i dostępu do informacji w systemie operacyjnym. Odwzorowuje logiczną koncepcję pliku na fizyczne urządzenia pamięci masowej. System operacyjny powinien realizować następujące zadania:

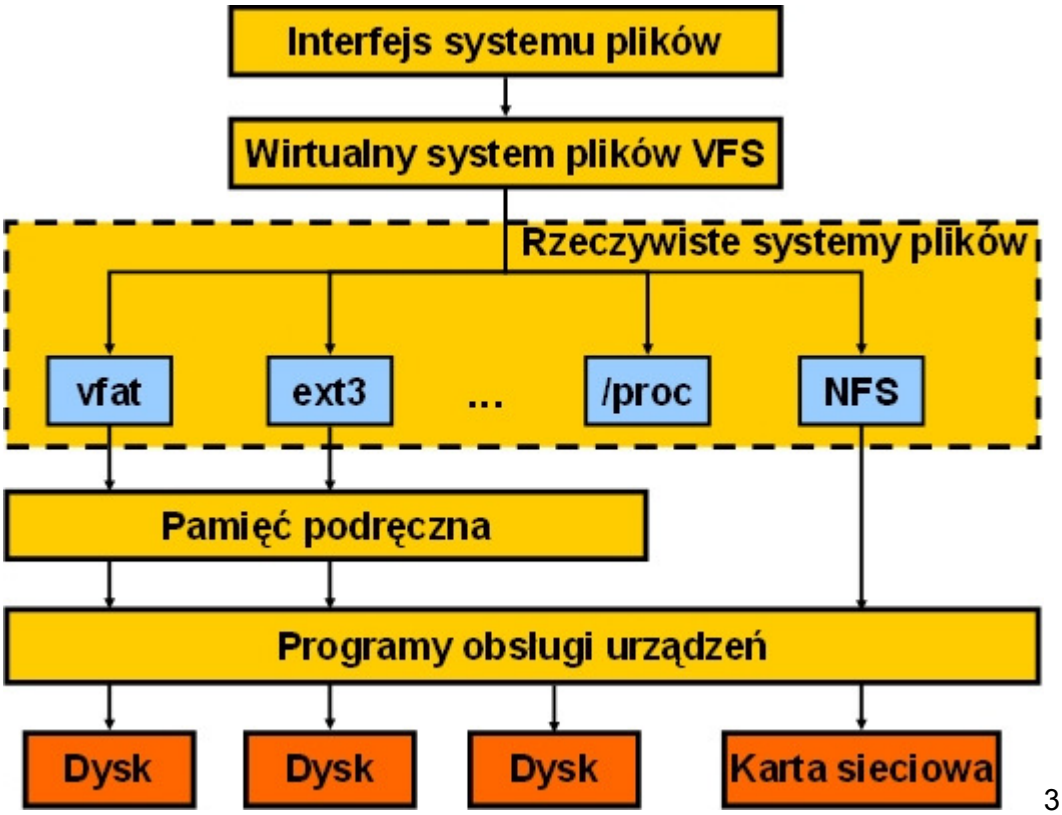
- tworzenie i usuwanie plików i katalogów,
- udostępnianie podstawowych operacji na plikach i katalogach,
- ochrona plików,
- odwzorowanie plików w pamięci,
- przechowywanie plików na urządzeniach pamięci masowej.

Większość współczesnych systemów operacyjnych może obsługiwać kilka różnych systemów plików. Posługują się w tym celu koncepcją wirtualnego systemu plików VFS (ang. Virtual File System), który pełni rolę jednolitego interfejsu pomiędzy systemami rzeczywistymi a programami. Realizacja koncepcji VFS zostanie tu omówiona na przykładzie systemu Linux.

W systemie Linux, podobnie jak w systemie Unix, system plików oferuje wspólny interfejs do plików, urządzeń wejścia-wyjścia, łączy komunikacyjnych i gniazdek komunikacji sieciowej.

Struktura warstwowa systemu plików

System plików Linuxa ma strukturę warstwową przedstawioną na Rys. 11.1.



3

Rys.11.1 Struktura systemu plików w systemie Linux

Urządzenia i programy obsługi (sterowniki)

Pliki przechowywane są zazwyczaj na urządzeniach blokowych o dostępie swobodnym, takich jak dyski magnetyczne lub optyczne. Każdy dysk może być podzielony na kilka partycji logicznych,

reprezentowanych w systemie jako odrębne urządzenia logiczne. Programy obsługi urządzeń stanowią część składową jądra systemu, która musi być zaimplementowana w specyficzny sposób dla każdej platformy sprzętowej. Każde urządzenie logiczne ma swoją reprezentację w postaci pliku specjalnego w systemie plików Linuksa.

Przykład:
`/dev/hda1` - reprezentuje pierwszą partycję na pierwszym dysku IDE w systemie,

`/dev/sda1` - reprezentuje pierwszą partycję na pierwszym dysku SATA w systemie,

`/dev/sdb2` - reprezentuje drugą partycję na drugim dysku SATA.

Każda partycja może zawierać odrębny system plików.

Podręczna pamięć buforowa

Podręczna pamięć buforowa (ang. *buffer cache*) pośredniczy w dostępie rzeczywistych systemów plików do blokowych urządzeń pamięci masowej. Bloki danych odczytane z urządzeń umieszczane są w buforach pamięci podręcznej i przechowywane przez pewien czas. Daje to możliwość ponownego ich użycia bez konieczności powtarzania operacji odczytu z urządzenia. Również wszystkie modyfikacje wprowadzane są najpierw w pamięci podręcznej i dopiero w później zapisywane w pamięci masowej w dogodnym dla systemu czasie. Dostęp do pamięci podręcznej realizowany jest znacznie szybciej i w ten sposób system zwiększa szybkość dostępu do danych przechowywanych przez fizyczne urządzenia pamięci.

Podręczna pamięć buforowa jest dzielona pomiędzy wszystkie urządzenia blokowe. W każdej chwili mogą się w niej znajdować bloki pochodzące z różnych urządzeń dyskowych. W celu poprawnej identyfikacji bloków, każdy bufor zaopatrzony jest w nagłówek przechowujący informacje o numerze urządzenia i numerze bloku.

Rzeczywiste systemy plików

System Linux może obsługiwać wiele popularnych systemów plików. Stanowi to jego niewątpliwą zaletę, gdyż umożliwia użytkownikom łatwe posługiwanie się plikami z różnych systemów bez konieczności dokonywania kłopotliwych konwersji danych.

Konkretny, działający system Linux obsługuje tylko te typy systemu plików, które zostały zarejestrowane w pliku `/etc/filesystems`. Jego zawartość może wyglądać następująco:

```
$ cat /etc/filesystems
ext3
ext2
nodev proc
nodev devpts
iso9660
vfat
hfs
```

Tablica 8.1 Wybrane rzeczywiste systemy plików obsługiwane w systemie Linux

Typ systemu plików	Opis
minix	Prosty system plików systemu operacyjnego Minix, zaadoptowany jako pierwszy system plików dla systemu Linux.
ext	Rozwinięcie systemu minix.
ext2	Podstawowy system plików systemu Linux.
ext3	System plików **ext2** z kronikowaniem, czyli rejestrowaniem transakcji systemu plików.
ext4	Nowy system plików z kronikowaniem.
ReiserFS	System plików z kronikowaniem.
msdos	System plików systemu MSDOS.
umsdos	Rozszerzona wersja systemu MSDOS.
vfat	System plików systemu Windows 95, NT.
ntfs	System plików systemu Windows NT, 2000.
hpfs	System plików systemu OS/2.
ufs	System plików systemu Solaris.
iso9660	System plików nośników CD-ROM zgodny z normą ISO 9660.
nfs	Sieciowy system plików firmy Sun.
smb	Sieciowy system plików oparty na protokole SMB.
proc	Pseudo-system plików stanowiący interfejs do struktur **task_struct** procesów w pamięci jądra.

Wirtualny system plików

System Linux musi zarządzać jednocześnie wieloma typami rzeczywistych systemów plików. System zawiera w tym celu dodatkową warstwę jednolitego interfejsu jądra, która separuje rzeczywiste systemy od reszty systemu operacyjnego. Interfejs ten nosi nazwę wirtualnego systemu plików VFS (ang. *Virtual File System*).

Zadaniem wirtualnego systemu plików jest połączenie poszczególnych systemów, zrealizowanych na odrębnych urządzeniach logicznych, w jeden wspólny system plików. System VFS organizuje wspólną strukturą katalogową i zapewnia wspólny interfejs funkcji systemowych, niezależny od fizycznej implementacji plików w systemach składowych. W tym celu VFS wykorzystuje specjalne struktury danych, które zapewniają dodatkowy poziom jednolitej reprezentacji poszczególnych plików i całych systemów plików.

Pojedynczy plik reprezentowany jest przez i-węzeł VFS, czyli wirtualny węzeł, opisujący jego atrybuty oraz wskazujący urządzenie logiczne i zestaw procedur do operacji na rzeczywistym pliku. Jądro tworzy taką reprezentację przy pierwszym dostępie do pliku.

Rzeczywisty system plików reprezentuje specjalny superblok VFS, który zawiera m.in.:

- nazwę urządzenia logicznego, zawierającego system pików,
- typ systemu plików,
- wskazanie na zestaw procedur operujących na bloku identyfikacyjnym rzeczywistego systemu,
- informacje specyficzne dla rzeczywistego systemu plików.

Dołączanie nowego systemu plików do wspólnej struktury katalogowej określane jest jako montowanie systemu plików. Operacja wymaga podania punktu montowania, którym powinien być pusty katalog. W przypadku podania niepustego katalogu, jego zawartość zostanie przykryta przez nowy fragment struktury katalogowej.

Polecenia systemowe

Tworzenie i nadzorowanie systemu plików należy do najważniejszych zadań administratora systemu. Wykorzystuje przy tym bogaty zestaw narzędzi programowych oferowanych przez system Linux. Do najważniejszych programów należą:

fdisk - tworzenie i manipulacja tablicą partycji dysku

mkfs - tworzenie systemu plików

fsck - sprawdzanie i naprawianie systemu plików

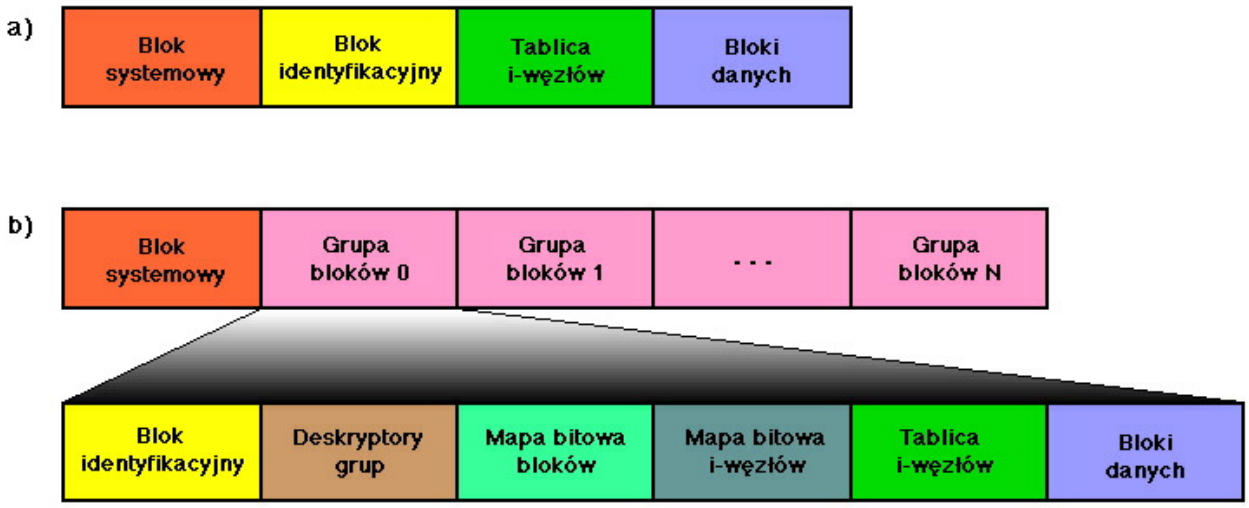
mount - montowanie systemu plików

umount - odmontowanie systemu plików

df - podaje zajętość przestrzeni dyskowej dla wszystkich zamontowanych systemów plików

11.2. System plików EXT3

System EXT3 jest jednym z podstawowych systemów plików Linuksa. Na rysunku 11.2 przedstawiono jego strukturę fizyczną w porównaniu ze strukturą klasycznego systemu plików systemów uniksowych.



Rys. 11.2 Struktura fizyczna systemu plików: a) klasyczna, b) systemu EXT3

System plików zrealizowany na urządzeniu blokowym składa się z ciągu bloków dyskowych o rozmiarze ustalonym w granicach od 512 B do 8 kB, będącym wielokrotnością 512 bajtów. Rozmiar bloku musi być ustalony podczas tworzenia systemu plików.

Niektóre spośród tych bloków mają szczególne przeznaczenie:

blok systemowy lub inicjujący (ang. *boot block*) - może zawierać podstawowy program ładujący jądro systemu operacyjnego do pamięci operacyjnej w czasie startowania systemu,

blok identyfikacyjny (ang. *superblock*) - zawiera informacje w pełni opisujące aktualny stan systemu plików.

Pozostałe bloki przechowują struktury i-węzłów reprezentujących pliki oraz dane plików. System EXT3 wyróżnia grupy bloków. Każda grupa zawiera:

- kopię bloku identyfikacyjnego systemu,
- bloki zawierające informacje o pozostałych grupach (deskryptory grup),
- bloki przechowujące mapy bitowe przydziału i-węzłów i bloków w grupie ,
- bloki zawierające tablicę i-węzłów plików ,
- bloki danych.

Dzięki zwielokrotnieniu informacji o całym systemie plików i o grupach bloków, zwiększa się bezpieczeństwo przechowywania danych. Ponadto, podział bloków na grupy ułatwia przechowywanie bloków danych pliku w pobliżu jego i-węzła oraz przechowywanie i-węzłów plików blisko i-węzła katalogu, do którego są dowiązane. Taka organizacja przyspiesza dostęp do danych.

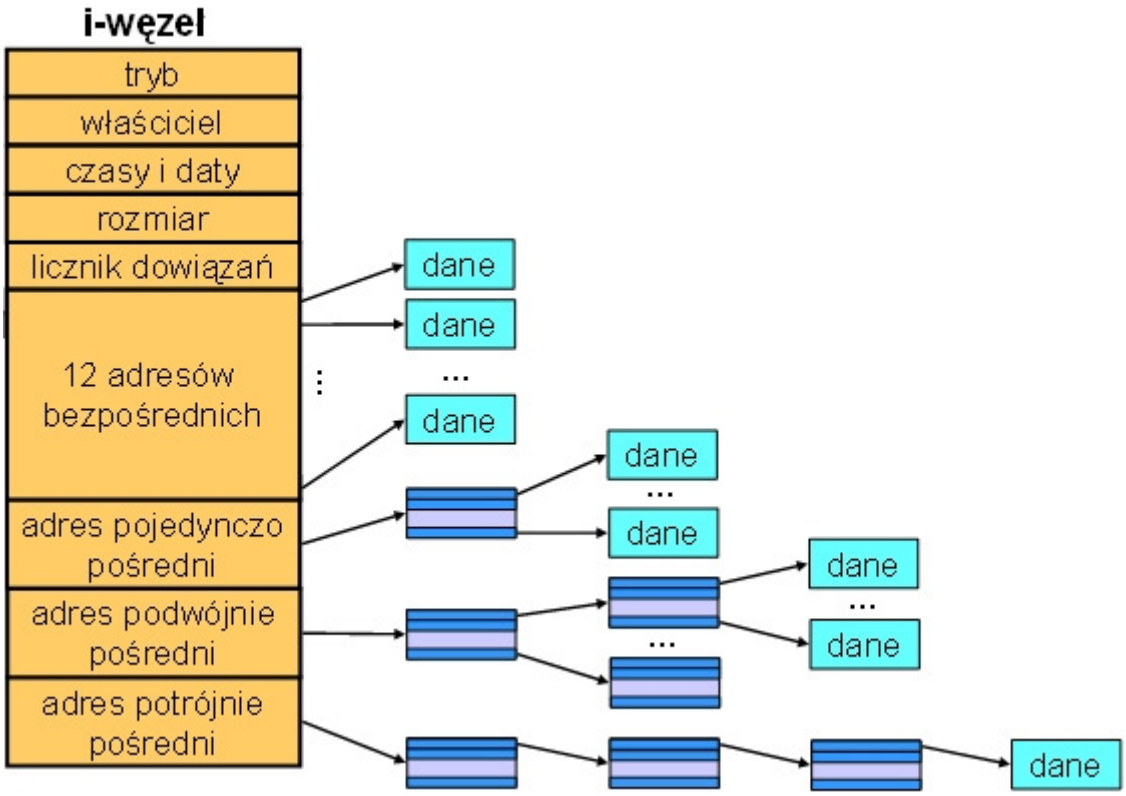
Obecnie większość systemów uniksowych stosuje systemu plików UFS, który ma strukturę zbliżoną do systemu EXT3.

Reprezentacja pliku w systemie EXT3

Każdy plik w systemie EXT3 reprezentowany jest przez strukturę nazywaną **i-węzłem** pliku (`struct inode`). W strukturze tej zapisane są wszystkie atrybuty pliku m. in.:

- tryb pliku,
- rozmiar pliku w bajtach,
- identyfikatory właściciela UID,
- identyfikator grupy GID,
- daty i czasy:
  - utworzenia pliku,
  - ostatniej modyfikacji zawartości,
  - ostatniej zmiany atrybutów,
- liczba dowiązań,
- adresy bloków danych:
  - 12 adresów bezpośrednich, wskazujących bloki danych,
  - 1 adres pośredni, wskazujący blok indeksowy zawierający właściwe adresy bloków danych,
  - 1 adres podwójnie pośredni, zawierający dwa poziomy wskazań pośrednich bloków indeksowych z adresami,
  - 1 adres potrójnie pośredni, zawierający trzy poziomy wskazań pośrednich bloków indeksowych z adresami.

Taki sposób reprezentacji pliku ilustruje rysunek 11.3.



Rys. 11.3 Reprezentacja pliku w systemie EXT3

Dzięki zastosowaniu pośredniego adresowania bloków danych, możliwe jest przechowywanie bardzo dużych plików przy zachowaniu stosunkowo niewielkich rozmiarów i-węzła, który zawiera

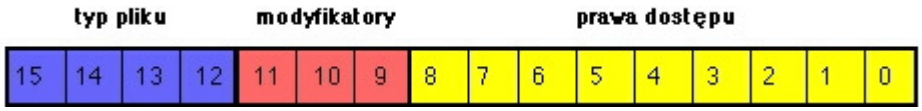
tylko15 adresów. Małe pliki mieszczą się w blokach adresowanych bezpośrednio. Większe pliki korzystają również z adresów pośrednich. Ceną za taką elastyczność jest wydłużanie czasu dostępu do danych wraz ze wzrostem rozmiaru pliku. Dostęp do początkowych danych pliku, mieszczących się w blokach adresowanych bezpośrednio, wymaga tylko jednej operacji dyskowej w celu odczytania zawartości bloku. Dane umieszczone w dalszych blokach adresowane są w sposób wielostopniowy. Z tego względu wymagają jednej, dwóch lub nawet trzech operacji dostępu do dysku w celu pobrania adresów i jednej operacji odczytania bloku danych. We wszystkich przypadkach dochodzi jeszcze konieczność odczytania z dysku i-węzła pliku (jeśli nie ma go w pamięci podręcznej).

Tryb pliku

Tryb pliku jest liczbą 16-bitową opisującą trzy atrybuty pliku:

- typ pliku (4 bity),
- modyfikatory praw dostępu (3 bity),
- prawa dostępu (9 bitów).

Przeznaczenie poszczególnych bitów pokazuje rysunek 11.4.



Rys.11.4 Tryb pliku

Pierwsze 4 bity kodują typ pliku zgodnie z opisem w tablicy 11.2.

Tablica 11.2 Kodowanie typu pliku

Stała symboliczna	Wartość w kodzie ósemkowym	Znaczenie
S_IFMT	0170000	maska bitowa typu pliku
S_IFSOCK	0140000	gniazdo
S_IFLNK	0120000	dowiązanie symboliczne
S_IFREG	0100000	plik zwykły
S_IFBLK	0060000	plik specjalny blokowy
S_IFDIR	0040000	katalog
S_IFCHR	0020000	plik specjalny znakowy
S_IFIFO	0010000	kolejka FIFO

Wymienione stałe mogą być używane do rozpoznawania typu pliku na podstawie wartości trybu. Stała S_IFMT definiuje maskę wycinającą bity określające typ pliku z 16-bitowej liczby trybu. Wynik porównuje się następnie z jedną z pozostałych stałych w celu rozpoznania typu. Na przykład poniższe wyrażenie sprawdza, czy plik jest katalogiem:

```
(tryb & S_IFMT) == S_IFDIR
```

Kolejne 3 bity trybu opisują modyfikatory praw dostępu do pliku. Znaczenie poszczególnych bitów opisano w tablicy 11.3.

Tablica 11.3 Modyfikatory praw dostępu

Nazwa bitu	Stała	Wartość	Znaczenie bitu
bit ustanowienia użytkownika (ang. *setuid bit*)	S_ISUID	0004000	Bit ma znaczenie tylko dla plików wykonywalnych. Powoduje, że proces wykonuje program z identyfikatorem obowiązującym użytkownika ustawionym na identyfikator własciciela pliku.
bit ustanowienia grupy (ang. *setgid bit*)	S_ISGID	0002000	Bit ma znaczenie tylko dla plików wykonywalnych. Powoduje, że proces wykonuje program z identyfikatorem obowiązującym grupy ustawionym na identyfikator grupy pliku.
bit lepkości (ang. *sticky bit)*	S_ISVTX	0001000	Dla pliku zwykłego: obecnie bit jest ignorowany. Dawniej wymuszał pozostawienie kodu programu w pamięci po zakończeniu jego wykonywania. Dla katalogu: pliki mogą być usuwane tylko przez właściciela lub administratora niezależnie od posiadania prawa pisania w katalogu.

Ostatnie 9 bitów trybu opisuje prawa dostępu do pliku dla właściciela, grupy i pozostałych użytkowników. Znaczenie tych praw zostało opisane w wykładzie 3. Istnieje również zestaw stałych symbolicznych dla różnych ustawień praw dostępu, jednak powszechnie stosuje się wartości w kodzie ósemkowym.

Katalogi i dowiązania

Katalog przechowuje listę pozycji wiążących nazwy plików z numerami ich i-węzłów. Każda pozycja w katalogu, nazywana dowiązaniem lub dowiązaniem twardym, reprezentowana jest przez strukturę `struci dirent` opisaną poniżej:

```
struct dirent {
        long d_ino;                     - numer i-węzła
        off_t d_off;                    - przesunięcie, wskazujące na nastepną pozycję w
                                          katalogu (początek nastepnej struktury dirent)
        unsigned short d_reclen;        - długość nazwy pliku
        char d_name [NAME_MAX+1];       - nazwa pliku
}
```

Plik może być dowiązany do wielu katalogów pod różnymi nazwami i wszystkie dowiązania są równorzędne. Utworzenie każdego nowego dowiązania pliku do katalogu powoduje zwiększenie wartości licznika dowiązań przechowywanego w i-węźle pliku. Usunięcie dowiązania powoduje zmniejszenie wartości tego licznika i nie ma wpływu na pozostałe dowiązania do pliku. Po usunięciu ostatniego dowiązania i wyzerowaniu licznika, jądro systemu zwalnia i-węzeł i bloki danych pliku.

Nowe dowiązanie pliku nie może przekroczyć granicy systemu plików. Oznacza to, że katalog zawierający dowiązanie i wskazywany przez nie plik muszą znajdować się na tej samej partycji logicznej.

Dowiązanie symboliczne jest plikiem specjalnego typu, który wskazuje położenie innego pliku w strukturze katalogowej. Reprezentowane jest przez i-węzeł i ewentualnie jeden blok danych zawierający nazwę ścieżkową wskazywanego pliku. Jeśli nazwa jest krótka, to przechowywana jest bezpośrednio w i-węźle, zamiast adresów bloków danych. Jako plik, dowiązanie symboliczne musi być też dowiązane do jakiegoś katalogu. Stworzenie dowiązania symbolicznego oznacza więc konieczność utworzenia nowego pliku i nowego dowiązania twardego w katalogu.

Dowiązanie symboliczne może wskazywać dowolny plik, przechowywany na dowolnej partycji logicznej. W ramach jednej struktury katalogowej. Dowiązanie symboliczne może przekraczać granicę systemu plików w ramach jednej struktury katalogowej. Może zatem wskazywać dowolny plik, przechowywany na dowolnej partycji logicznej, ponieważ przechowuje tylko jego nazwę ścieżkową w strukturze katalogowej.

Pliki specjalne

Wszystkie urządzenia są w systemie Linux reprezentowane przez pliki specjalne. System rozróżnia dwa typy urządzeń:

urządzenia blokowe o dostępie swobodnym - reprezentowane przez pliki specjalne blokowe,

urządzenia znakowe o dostępie sekwencyjnym - reprezentowane przez pliki specjalne znakowe.

Reprezentacja plików specjalnych różni się trochę od reprezentacji innych plików. Plik specjalny nie korzysta z bloków danych, a w jego i-węźle zamiast adresów przechowywane są dwa numery programu obsługi urządzenia: główny i drugorzędny.

Do tworzenia plików specjalnych służy funkcja systemowa **mknod**().

```
int mknod(const char *pathname, mode_t mode, dev_t dev);
```

gdzie:

pathname - nazwa scieżkowa tworzonego pliku,

mode - tryb pliku, definiujący typ i prawa dostępu do pliku,

dev - numery urządzenia, główny i drugorzędny.

11.3. Reprezentacja otwartych plików

System Linux przechowuje następujące informacje o każdym otwartym pliku:

- położenie na dysku,
- licznik operacji otwarcia przez wiele procesów,
- wskaźnik bieżącej pozycji w pliku, czyli wskaźnik pozycji ostatniej operacji zapisu/odczytu, oddzielnie dla każdego procesu, który otworzył plik,
- tryb dostępu do pliku, oddzielnie dla każdego procesu.

Jądro tworzy w tym celu strukturę `struct file`.

W każdym procesie tworzona jest tablica deskryptorów otwartych plików, w której zapisywane są wskaźniki do odpowiednich struktur `file`. Trzy najniższe numery deskryptorów są zarezerwowane dla plików związanych ze standardowymi strumieniami danych:

0 - standardowy strumień wejściowy (stdin),

1 - standardowy strumień wyjściowy (stdout),

2 - standardowy strumień wyjściowy błędów, czyli diagnostyczny (stderr).

Tablica deskryptorów może być alokowana statycznie lub dynamicznie. W przypadku alokacji statycznej występuje ograniczenie liczby plików otwartych przez proces (zwykle do 64). Z tego względu, we współczesnych wersjach systemu Linux stosuje się dynamiczną alokację tablicy z możliwością późniejszego rozszerzania.

11.4. Operacje na plikach

Podstawowe operacje na plikach zwykłych

Utworzenie nowego pliku zwykłego lub otwarcie istniejącego umożliwiają dwie funkcje systemowe:

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

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

gdzie:

pathname - nazwa ścieżkowa pliku,

flags - flagi,

mode - prawa dostępu do pliku.

Funkcja **open**() otwiera plik o podanej nazwie, przydziela mu najmniejszy z dostępnych w procesie deskryptorów i zwraca jego wartość. Jeżeli podany plik nie istnieje, to funkcja zwraca błąd albo tworzy ten plik, gdy ustawione są odpowiednie flagi.

Argument **flags** musi zawierać znacznik określający typ dostępu do pliku:

O_RDONLY - otwarty tylko do czytania,

O_WRONLY - otwarty tylko do pisania,

O_RDWR - otwarty do czytania i pisania.

Dodatkowo można dołączyć w postaci sumy bitowej inne flagi modyfikujące sposób działania funkcji oraz sposób korzystania z otwartego pliku. W tablicy 8.4 opisano przeznaczenie kilku najczęściej używanych flag.

Tablica 8.4 Flagi modyfikujące sposób dostępu do pliku

Flaga	Znaczenie
O_CREAT	Funkcja tworzy podany plik lub otwiera go, jeśli plik już istnieje.
O_EXCL	Powinna być używana wyłącznie z O_CREAT. Funkcja tworzy podany plik lub zwraca błąd, jeśli plik już istnieje.
O_TRUNC	Jeżeli plik już istnieje, to funkcja kasuje jego zawartość i ustawia rozmiar 0.
O_APPEND	Funkcja otwiera plik w trybie dopisywania. Wskaźnik bieżącej pozycji ustawiany jest na koniec pliku przed każda operacją zapisu.
O_NONBLOCK O_NDELAY	Funkcja otwiera plik w trybie operacji bez blokowania. Taki tryb można ustawić jedynie dla kolejek FIFO i gniazd. Operacje na zwykłych plikach są zawsze blokujące.
O_SYNC	Funkcja otwiera plik w trybie synchronicznych operacji zapisu. Każda operacja zapisu do pliku powoduje zablokowanie procesu do momentu fizycznego zapisania danych na dysku. W zwykłym trybie zapisane dane podlegają buforowaniu w pamięci i są zapisywane na dysku w dogodnym dla systemu momencie.

Wywołanie funkcji **creat**() jest równoważne wywołaniu funkcji **open**() z flagami: O_CREAT | O_WRONLY | O_TRUNC.

Do zamykania otwartych plików służy funkcja **close**().

```
int close(int fd);
```

Funkcja zamyka podany deskryptor pliku. Jeśli jest to ostatnia kopia deskryptora wskazującego otwarty plik, to struktura **file** tego pliku jest usuwana z pamięci. Przed zamknięciem funkcja zapisuje na dysku wszystkie zbuforowane dane. Jeśli ta operacja nie powiedzie się, to funkcja zwróci błąd.

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

gdzie:

fd - deskryptor otwartego pliku,

buf - wskaźnik do bufora, w którym zostaną zapisane dane odczytane z pliku,

count - liczba bajtów danych do odczytania.

Funkcja wczytuje **count** bajtów z pliku o deskryptorze **fd** zaczynając od bieżącej pozycji wskaźnika w pliku i umieszcza odczytane dane w buforze **buf.** Funkcja zwraca liczbę wczytanych bajtów, która może być mniejsza od wartości **count**. Domyślnie proces jest blokowany do momentu zakończenia operacji. Każda operacja odczytu powoduje przesunięcie wskaźnika bieżącej pozycji o liczbę odczytanych bajtów.

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

gdzie:

fd - deskryptor otwartego pliku,

buf - wskaźnik do bufora, z którego zostaną pobrane dane przeznaczone do zapisania w pliku,

count - liczba bajtów danych do zapisania.

Funkcja zapisuje **count** bajtów z bufora **buf** w pliku o deskryptorze **fd** zaczynając od bieżącej pozycji wskaźnika w pliku**.** Funkcja zwraca liczbę zapisanych bajtów, która może być mniejsza od wartości **count**. Domyślnie proces jest blokowany do momentu zakończenia operacji. Każda operacja zapisu powoduje przesunięcie wskaźnika bieżącej pozycji o liczbę zapisanych bajtów. Dzięki temu kolejne dane zostaną zapisane za poprzednimi, najczęściej na końcu pliku.

Istnieje możliwość ustawienia wskaźnika na dowolną pozycję w pliku przy pomocy funkcji **lseek**().

```
off_t lseek(int fd, off_t offset, int whence);
```

gdzie:

fd - deskryptor otwartego pliku,

offset - przesunięcie wskaźnika,

whence - miejsce w pliku, względem którego następuje przesunięcie.

Funkcja przesuwa wskaźnik bieżącej pozycji w pliku o **offset** bajtów względem miejsca określonego przez argument **whence**:

SEEK_SET - przesunięcie względem początku pliku,

SEEK_CUR - przesunięcie względem bieżącej pozycji w pliku,

SEEK_END - przesunięcie względem końca pliku.

Dodatnia wartość offset powoduje przesunięcie w kierunku końca pliku. Wartość ujemna może wystąpić tylko dla SEEK_CUR i SEEK_END i powoduje przesunięcie w kierunku początku pliku.

Wskaźnik może być nawet przesunięty poza aktualny koniec pliku.

Funkcja **truncate**() umożliwia skrócenie pliku do podanej długości poprzez ustawienie nowej wartości atrybutu rozmiar w i-węźle. Dane położone poza nowym rozmiarem są tracone.

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

gdzie:

path - nazwa pliku,

length - długość pliku w bajtach.

Przykład
Przedstawiamy kod programu, który odwraca kolejność znaków w pliku podanym jako argument wywołania (zapisuje ten sam plik od końca). Program wykorzystuje funkcje lseek() do przesuwania wskaźnika oraz funkcje read() i write() do zamiany znaków.

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

#define MAXNAME 100
#define PERM 0666

main(int argc, char *argv[])
{
        int dfile, dtmp;
        off_t ptr;
        char tmpfile[MAXNAME]="\0", c;

        if (argc != 2)
        {
                printf("Skladnia: %s plik\n", argv[0]);
                exit(1);
        }
        if ((dfile = open(argv[1], O_RDONLY)) == -1)
        {
                printf("Blad otwarcia pliku: %s\n", argv[1]);
                exit(2);
        }
        sprintf(tmpfile, "tmp%d", getpid());
        if ((dtmp = creat(tmpfile, PERM)) == -1)
        {
                printf("Blad utworzenia pliku: %s\n", tmpfile);
                exit(2);
        }
        if ((ptr = lseek(dfile, -1L, 2)) == -1)
        {
                exit(3);
```

```
        }
        while (read(dfile, &c, 1) == 1)
        {
                write(dtmp, &c, 1);
                if (ptr == 0)
                        break;
                if ((ptr = lseek(dfile, -2L, 1)) == -1)
                {
                        exit(3);
                }
        }
        if (close(dfile) == -1)
        {
                printf("Blad zamkniecia pliku: %s\n", argv[1]);
                exit(4);
        }
        if (close(dtmp) == -1)
        {
                printf("Blad zamkniecia pliku: %s\n", tmpfile);
                exit(4);
        }
        unlink(argv[1]);
        link(tmpfile, argv[1]);
        unlink(tmpfile);
}
```

Pobieranie i modyfikacja atrybutów pliku

Atrybuty pliku zapisane w i-węźle można pobrać jedną z trzech funkcji systemowych:

```
int stat(const char *file_name, struct stat *buf);

int lstat(const char *file_name, struct stat *buf);

int fstat(int filedes, struct stat *buf);
```

gdzie:

file_name - nazwa ścieżkowa pliku,

filedes - deskryptor pliku,

buf - wskaźnik do struktury **stat**, w której zostaną zapisane atrybuty pliku.

Funkcje odczytują zawartość i-węzła wskazanego pliku i zapisują w buforze. Plik może być wskazany przez nazwę w funkcjach **stat**() i **lstat**() lub przez numer deskryptora otwartego pliku w funkcji **fstat**(). Proces nie musi posiadać żadnych uprawnień do pliku, ale musi posiadać prawo przeglądania wszystkich katalogów podanych w nazwie ścieżkowej pliku.

Funkcja **lstat**() pozwala odczytać atrybuty dowiązań symbolicznych, podczas gdy **stat**() podąża za dowiązaniem i operuje na pliku wskazywanym.

Funkcje zapisują informacje o pliku w strukturze **stat** zdefiniowanej następująco:

```
struct stat {

        dev_t   st_dev;             - nazwa urządzenia, na którym plik jest zapisany
```

```
        ino_t   st_ino;             - numer i-węzła
        mode_t   st_mode;           - tryb pliku
        nlink_t   st_nlink;         - liczba dowiązań
        uid_t   st_uid;             - identyfikator właściciela UID
        gid_t   st_gid;             - identyfikator grupy GID
        dev_t   st_rdev;            - typ urządzenia dla plików specjalnych
        off_t   st_size;            - rozmiar pliku w bajtach
        unsigned long st_blksize;   - zalecany rozmiar bloku dla operacji wejścia/wyjścia
        unsigned long st_blocks;    - liczba zajmowanych bloków dyskowych
        time_t   st_atime;          - czas ostatniego dostępu do pliku
        time_t   st_mtime;          - czas ostatniej modyfikacji zawartości
        time_t   st_ctime;          - czas ostatniej zmiany atrybutów
}
```

Wartości atrybutów pliku ulegają zmianie w wyniku działania różnych funkcji systemowych, np. dopisanie danych do pliku powoduje zmianę rozmiaru i daty modyfikacji. Niektóre atrybuty można zmieniać wprost posługując się specjalnymi funkcjami systemowymi. Dotyczy to w szczególności identyfikatorów użytkowników i praw dostępu do pliku.

Zmianę identyfikatorów umożliwia jedna z poniższych funkcji:

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

int lchown(const char *path, uid_t owner, gid_t group);

int fchown(int fd, uid_t owner, gid_t group);
```

gdzie:

path - nazwa ścieżkowa pliku,

fd - deskryptor otwartego pliku,

owner - identyfikator właściciela,

group - identyfikator grupy.

Wyłącznie administrator systemu (użytkownik **root**) może zmienić dowolnie obydwa identyfikatory. Właściciel pliku może jedynie ustawić numer jednej z grup, do których sam należy. Różnice w działaniu poszczególnych funkcji są analogiczne jak w przypadku funkcji **stat**().

Proces chcący określić swoje uprawnienia do pliku musi zinterpretować wartości identyfikatorów i trybu pliku, odczytane funkcją **stat**(). Jest to operacja dość kłopotliwa i czasochłonna dla twórcy programu. System udostępnia więc funkcję **access**() dającą możliwość prostego sprawdzania uprawnień do pliku.

```
int access(const char *pathname, int mode);
```

gdzie:

pathname - nazwa ścieżkowa pliku,

mode - maska uprawnień, które mają być sprawdzone.

Argument **mode** może przyjąć jedną lub kilka z następujących wartości:

F_OK - sprawdza, czy plik istnieje,

R_OK - sprawdza, czy plik istnieje oraz czy proces posiada prawo do czytania z pliku,

W_OK - sprawdza, czy plik istnieje oraz czy proces posiada prawo do pisania do pliku,

X_OK - sprawdza, czy plik istnieje oraz czy proces posiada prawo do wykonywania pliku.

Funkcja zwraca 0, jeśli wyspecyfikowane uprawnienia są nadane.

Prawa dostępu oraz ich modyfikatory można zmieniać funkcją systemową **chmod**().

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

int fchmod(int fildes, mode_t mode);
```

gdzie:

pathname - nazwa ścieżkowa pliku,

fildes - deskryptor otwartego pliku,

mode - tryb dostępu do pliku.

Tryb podaje się w kodzie ósemkowym lub w postaci symbolicznej, wykorzystując zestaw zdefiniowanych stałych symbolicznych. Pełny zestaw stałych można odnaleźć w dokumentacji elektronicznej. Zmian może dokonać proces z identyfikatorem obowiązującym właściciela pliku lub administratora.

Pliki tworzone przez proces mają ustawiany tryb dostępu na podstawie atrybutu **umask** procesu. Atrybut definiuje maskę praw dostępu wskazującą, które bity praw dostępu mają być wyłączone. Proces może zmienić swoją maskę wywołując funkcję systemową:

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

gdzie:

mask - maska praw dostępu w kodzie ósemkowym.

Funkcja zwraca starą wartość maski i ustawia nową.

11.5. Operacje na deskryptorach

Większość funkcji systemowych operujących na plikach odnosi się do ich fizycznej reprezentacji w postaci i-węzła i bloków danych. Funkcje te mogą odczytywać lub modyfikować zawartość pliku albo jego atrybuty zapisane w i-węźle. Wszystkie zmiany stają się widoczne dla innych procesów korzystających z pliku.

Niektóre funkcje dają procesom możliwość modyfikacji tylko tych własności, które nie są na trwałe związane z plikiem, a jedynie określają sposób dostępu procesu do otwartego pliku. Funkcje te operują na strukturach **file** otwartych plików wskazywanych przez deskryptory. Powszechnie przyjęło się określać w skrócie, że funkcje wykonują operacje na deskryptorach plików.

Funkcja sterująca **fcntl**() umożliwia wykonywanie różnorodnych operacji na deskryptorach, które mogą dotyczyć między innymi:

- odczytania lub ustawienia flag (znaczników), określających np. trybu dostępu do otwartego pliku,
- powielania deskryptorów,
- blokowania rekordów w pliku,
- asynchronicznych operacji wejścia - wyjścia sterowanych sygnałem SIGIO.

Omówione tu zostaną tylko dwa pierwsze z wymienionych zagadnień. Pozostałe zagadnienia wykraczają poza tematykę tego podręcznika a ich omówienie można znaleźć w [3].

```
int fcntl(int fd, int cmd);

int fcntl(int fd, int cmd, long arg);
```

gdzie:

fd - deskryptor otwartego pliku,

cmd - operacja na deskryptorze,

arg - argument wymagany przez niektóre operacje.

Argument **cmd** może przyjąć jedną z wielu wartości zdefiniowanych w postaci stałych, takich jak:

F_GETFL - odczytuje wartości wszystkich flag otwartego pliku,

F_SETFL - ustawia wartości flag na podstawie argumentu **arg**,

F_DUPFD - powiela deskryptor otwartego pliku.

Flagi określające tryb dostępu ustalane są podczas otwierania pliku funkcją **open**(). Specyfikuje się wtedy, czy plik ma być otwarty do czytania, pisania lub dopisywania. Domyślnie ustawiany jest też tryb operacji blokujących proces do momentu zakończenia operacji. Tylko niektóre flagi mogą być ustawiane funkcją **fcntl**() już po otwarciu pliku:

O_APPEND - określająca tryb dopisywania do pliku,

O_NONBLOCK - określająca tryb operacji bez blokowania,

O_ASYNC - określająca tryb operacji asynchronicznych.

Proces ma możliwość uzyskania nowego deskryptora do otwartego pliku poprzez powielenie aktualnego deskryptora. Taką możliwość zapewnia funkcja **fcntl**() oraz dwie inne funkcje systemowe **dup**() i **dup2**(). Każda z funkcji zwraca nowy deskryptor wskazujący ten sam otwarty plik, co stary deskryptor. W tablicy **fd**[] otwartych plików procesu, funkcja tworzy kopię wskaźnika do struktury **file** pliku określonego przez stary deskryptor. Nowy deskryptor jest indeksem nowego wskaźnika w tablicy. Obydwa deskryptory mogą być odtąd używane zamiennie w dostępie do pliku.

Funkcja **fcntl**() powinna być wywołana z poleceniem F_DUPFD i argumentem określającym minimalny numer nowego deskryptora np.:

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

Proces może również wykorzystać jedną z pozostał funkcji do powielania deskryptorów:

```
int dup(int oldfd);

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

gdzie:

oldfd - stary deskryptor związany z otwartym plikiem,

newfd - nowy deskryptor pliku.

Funkcja **dup**() przydziela najmniejszy deskryptor aktualnie dostępny w procesie i zwraca jego wartość. Nie daje więc możliwosci wyboru konkretnego numeru nowego deskryptora pliku. Często jednak proces chce uzyskać konkretny deskryptor, na przykład deskryptor 1 w celu przekierowania do pliku standardowego strumienia wyjściowego (stdout). Powinien w tym celu zamknąć plik związany z tym deskryptorem, a następnie użyć funkcji systemowej **dup2**(), która pozwala wskazać nowy deskryptor pliku **newfd**. Jeśli wskazany deskryptor jest już używany, to zostanie najpierw zamknięty.

11.6. Operacje na dowiązaniach

Operacje na dowiązaniach (dowiązaniach twardych) są w istocie operacjami na zawartości katalogu. Sprowadzają się do dodania lub usunięcia pozycji w katalogu albo na zmianie nazwy istniejącej pozycji. Z tego względu nie są wymagane żadne uprawnienia do plików wskazywanych przez dowiązania, a jedynie do katalogów, których dane zmieniamy.

Funkcja systemowa **link**() umożliwia tworzenie nowych dowiązań do istniejącego pliku.

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

gdzie:

oldpath - nazwa ścieżkowa istniejącego dowiązania do pliku,

newpath - nazwa ścieżkowa nowego dowiązania do pliku.

Nowa nazwa **newpath** musi dotyczyć tego samego systemu plików (tej samej partycji logicznej). Jeśli istnieje już dowiązanie o podanej nazwie, to nie zostanie zmienione. W wyniku pomyślnego zakończenia funkcji plik będzie widoczny w strukturze katalogowej pod dwiema (lub więcej) nazwami.

Funkcja systemowa **rename**() zmienia nazwę w istniejącym dowiązaniu albo usuwa dowiązanie w jednym katalogu i tworzy w innym nowe dowiązanie do tego samego pliku.

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

Funkcja zmienia tylko jedno dowiązanie do pliku, wyspecyfikowane w wywołaniu. Pozostałe dowiązania, jeśli istnieją, nie ulegają zmianie.

Każde dowiązanie można usunąć funkcją systemową **unlink**().

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

Operacja powoduje zmniejszenie wartości licznika dowiązań w i-węźle pliku i nie ma wpływu na pozostałe dowiązania do pliku. Po usunięciu ostatniego dowiązania i wyzerowaniu licznika plik również zostaje usunięty przez jądro systemu.

Funkcja **symlink**() umożliwia stworzenie dowiązania symbolicznego do istniejącego pliku.

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

gdzie:

oldpath - nazwa ścieżkowa istniejącego dowiązania do pliku,

newpath - nazwa ścieżkowa nowego dowiązania symbolicznego do pliku.

W wyniku działania funkcji powstaje nowy plik typu dowiązanie symboliczne wskazujący nazwę ścieżkową innego pliku dokładnie w takiej postaci, w jakiej została podana w argumencie **oldpath**. Powstaje również zwykłe dowiązanie do nowego pliku w katalogu wynikającym z nazwy ścieżkowej **newpath** (w katalogu bieżącym, jeśli **newpath** zawiera tylko nazwę pliku). Nowa nazwa może dotyczyć innego systemu plików (innej partycji), ponieważ dowiązanie symboliczne wskazuje na nazwę a nie na i-węzeł.

11.7. Zwielokrotnianie wejścia i wyjścia

Procesy często korzystają z wielu źródeł danych i w związku z tym są zmuszone obsługiwać wiele deskryptorów otwartych plików. Źródłem danych mogą być pliki urządzeń, np. plik terminala, łącza komunikacyjne lub gniazda. Jeżeli nowe dane nadchodzą w nieustalonej kolejności, to pojawia się problem właściwej obsługi wszystkich otwartych plików. Istnieją dwie podstawowe metody rozwiązania tego problemu:

- aktywne czekanie, polegające na sekwencyjnym sprawdzaniu kolejno wszystkich otwartych plików,
- wstrzymanie działania procesu w oczekiwaniu na pojawienie się nowych danych w którymkolwiek z otwartych plików.

Metoda aktywnego czekania wymaga użycia operacji nieblokujących, aby wykluczyć możliwość wstrzymania procesu przy próbie odczytania nowych danych z pliku. Istotną wadą tej metody jest duże obciążenie systemu przez stale działający proces, który wciąż usiłuje odczytywać dane.

Zastosowanie drugiej metody umożliwia funkcja **select**(). Wstrzymuje ona wykonywanie procesu w oczekiwaniu na zmianę statusu podanych deskryptorów otwartych plików.

```
int select(int n, fd_set *readfds, fd_set *writefds, fd_set
*exceptfds, struct timeval *timeout);
```

gdzie:

n - liczba deskryptorów do sprawdzenia, obliczana jako najwyższy numer użytego deskryptora plus 1,

readfds - zbiór deskryptorów sprawdzanych pod kątem możliwości odczytania danych,

writefds - zbiór deskryptorów sprawdzanych pod kątem możliwości zapisania danych,

exceptfds - zbiór deskryptorów sprawdzanych pod kątem nieobsłużonych sytuacji wyjątkowych,

timeout - struktura opisująca maksymalny czas oczekiwania funkcji **select**() na zajście jakiegoś zdarzenia.

Deskryptory pogrupowane są w trzy zbiory typu **fd_set** w zależności od rodzaju zdarzenia, na które proces chce oczekiwać.

Operacje na zbiorze deskryptorów ułatwia zestaw następujących makrodefinicji:

FD_ZERO(fd_set *set); - zeruje zbiór deskryptorów,

FD_SET(int fd, fd_set *set); - ustawia podany deskryptor w zbiorze,

FD_CLR(int fd, fd_set *set); - usuwa podany deskryptor ze zbioru,

FD_ISSET(int fd, fd_set *set); - sprawdza, czy podany deskryptor jest ustawiony.

11.8. Operacje na katalogach

Zmiana katalogu głównego i katalogu bieżącego procesu

W każdym procesie zdefiniowane są dwa atrybuty określające położenie w strukturze katalogowej systemu:

- katalog główny procesu,
- katalog bieżący procesu.

W systemie plików istnieje jeden katalog główny /. Proces może jednak wybrać inny katalog, do którego będzie się odnosił przez nazwę / i traktował jako katalog główny systemu. Wszystkie odwołania procesu poprzez ścieżkę bezwzględną będą interpretowane względem nowego katalogu głównego, przez co proces uzyska dostęp ograniczony tylko do określonego poddrzewa struktury katalogowej.

Zdefiniowanie nowego katalogu głównego w procesie umożliwia funkcja systemowa:

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

Funkcja ustala nowy katalog główny, ale nie zmienia katalogu bieżącego procesu. Może więc dojść do sytuacji, gdy katalog bieżący procesu nie leży wewnątrz jego katalogu głównego. Proces powinien wtedy zmienić bieżący katalog roboczy przy pomocy funkcji systemowej **chdir**().

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

Nazwę bieżącego katalogu można uzyskać funkcją biblioteczną:

```
char *getcwd(char *buf, size_t size);
```

gdzie:

buf - wskaźnik do bufora przeznaczonego na nazwę katalogu,

size - rozmiar bufora.

Funkcja zapisuje nazwę katalogu zakończoną znakiem '\0' do bufora i zwraca wskaźnik do tego bufora. Jeżeli długość nazwy przekracza rozmiar **size - 1**, to funkcja zwraca błąd. Dopuszczalne jest użycie argumentu NULL zamiast wskaźnika do bufora, aby zezwolić funkcji na dynamiczne zarezerwowanie bufora odpowiednich rozmiarów.

Przykład
Poniższy program to nieco bardziej rozbudowany interpreter poleceń **msh**. Powłoka **msh** analizuje wprowadzone polecenia i wykonuje je w pierwszym planie lub w tle. Ponadto, powłoka msh realizuje dwa polecenia wbudowane: **cd** i **exit**. Program pokazuje praktyczne zastosowanie funkcji systemowych **fork(), execvp(), waitpid(), chdir(), getenv()**.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
  char bufor[80];
  char* arg[10];
  int bg;
  while (1)
  {
    printf("msh $ ");
    fgets(bufor, 80, stdin);
```

```
      bg = AnalizujPolecenie(bufor, arg);
    if (arg[0] == NULL)
      continue;
    else if (strcmp(arg[0], "exit")==0)
      exit(0);
    else if (strcmp(arg[0], "cd")==0)
      Cd(arg);
    else
      Wykonaj(arg, bg);
  }
}
int AnalizujPolecenie(char *bufor, char *arg[])
{
  int counter=0;
  int i=0, j=0;
  while (bufor[counter] != '\n')
  {
    while (bufor[counter] == ' ' || bufor[counter] == '\t')
      counter++;
    if (bufor[counter] != '\n')
    {
      arg[i++] = &bufor[counter];
      while (!isspace(bufor[counter]))
        counter++;
      if (bufor[counter] != '\n')
        bufor[counter++] = '\0';
    }
  }
  bufor[counter] = '\0';
  arg[i]=NULL;
  if (i>0)
  while (arg[i-1][j] != '\0')
  {
    if (arg[i-1][j] == '&')
    {
      if (j == 0)
        arg[i-1] = NULL;
      else
        arg[i-1][j] = '\0';
      return 1;
    }
    j++;
 }
 return 0;
}
int Wykonaj(char **arg, int bg)
{
  pid_t pid;
  int status;
  if ((pid=fork()) == 0)
  {
    execvp(arg[0],arg);
    perror("Blad exec");
    return 1;
 }
 else if (pid > 0)
 {
    if (bg == 0)
```

```
      waitpid(pid, &status, 0);
    return 0;
 }
 else
 {
    perror("Blad fork");
    return 1;
 }
}
int Cd(char **arg)
{
 char *katalog;
  if (arg[1] == NULL)
    katalog = getenv("HOME");
 else
    katalog = arg[1];
 if (chdir(katalog) == -1)
 {
    perror("Blad chdir");
    return 1;
 }
}
```

Tworzenie i usuwanie katalogów

Funkcja **mkdir**() umożliwia utworzenie katalogu z podanymi prawami dostępu zmodyfikowanymi przez maskę procesu.

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

gdzie:

pathname - nazwa ścieżkowa katalogu,

mode - prawa dostępu do katalogu.

Do usuwania pustych katalogów służy funkcja **rmdir**().

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

Argumentem funkcji nie mogą być pozycje **.** i **..** ani niepusty katalog.

Przeglądanie zawartości katalogów

Katalogi są reprezentowane tak samo, jak zwykłe pliki. Inna powinna być tylko interpretacja zawartości ich bloków danych. W zasadzie można więc używać funkcji systemowych **open**(), **read**() i **close**() do odczytania zawartości katalogu i właściwie zinterpretować odczytane dane. Zaleca się jednak korzystanie z zestawu funkcji zdefiniowanych w bibliotece języka C.

Funkcja **opendir**() otwiera katalog wyłącznie do czytania i zwraca wskaźnik do struktury DIR, reprezentującej otwarty katalog. Nie ma możliwości otwarcia katalogu do pisania.

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

Zwrócony wskaźnik identyfikuje otwarty katalog w wywołaniach pozostałych funkcji bibliotecznych.

Zawartość katalogu odczytywana jest sekwencyjnie pozycja po pozycji przy pomocy funkcji **readdir**().

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

Funkcja odczytuje jedną pozycję z katalogu i zapisuje w strukturze **dirent** opisanej powyżej. Jednocześnie ustawia wskaźnik na następną pozycję w katalogu, która zostanie odczytana w następnym wywołaniu funkcji.

Aktualne położenie wskaźnika można uzyskać funkcją **telldir**() a zmianę tego położenia umożliwia funkcja **seekdir**().

```
off_t telldir(DIR *dir);

void seekdir(DIR *dir, off_t offset);
```

Argument **offset** określa przesunięcie względem bieżącej pozycji wskaźnika i powinno stanowić wielokrotność rozmiaru struktury **dirent**.

Funkcja **rewinddir**() przestawia wskaźnik na początek katalogu umożliwiając ponowne odczytanie pierwszej pozycji.

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

Po zakończeniu czytania proces wywołuje funkcję **closedir**() w celu zamknięcia katalogu.

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

Przykład

Prezentowany poniżej program to prosta implementacja polecenia **ls**. Wypisuje nazwy plików z katalogu podanego jako argument wywołania lub z katalogu bieżącego. Program ilustruje zastosowanie funkcji do operacji na katalogach.

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

int main(int argc, char *argv[])
{
  char *katalog;
  DIR *dir;
  struct dirent *pozycja;
  switch(argc)
  {
    case 1: katalog = "."; break;
    case 2: katalog = argv[1]; break;
    default : printf("Poprawna skladnia:\t%s katalog\n", argv[0]);
    exit(1);
  }
  if ((dir = opendir(katalog)) == NULL)
  {
    perror("Blad opendir");
    return 1;
  }
  errno = 0;
  while ((pozycja = readdir(dir)) != NULL)
    printf("%s\n", pozycja->d_name);
  if (errno)
  {
    perror("Blad readdir");
    return 1;
```

```
  }
  if (closedir(dir) == -1)
  {
    perror("Blad closedir");
    return 1;
  }
  return 0;
}
```

11.9. Odwzorowanie plików w pamięci

System Linux umożliwia odwzorowanie zawartości pliku w pamięci wirtualnej procesu. Zawartość pliku nie jest od razu kopiowana do pamięci operacyjnej, a jedynie dołączana jako nowy obszar do pamięci wirtualnej procesu. Plik może być wtedy traktowany jak ciągły obszar w pamięci procesu, dostępny poprzez bezpośrednie operacje pobrania i podstawienia wartości. Wszystkie modyfikacje dokonane w pamięci są później zapisywane w pliku na dysku. Unika się w ten sposób stosowania funkcji systemowych do operacji na plikach i pośredniego etapu kopiowania danych do podręcznej pamięci buforowej.

Mechanizm odwzorowania pamięci znajduje kilka zastosowań:

- odwzorowanie plików z kodem wykonywalnym w pamięci procesów, które ten kod wykonują (realizacja funkcji **exec**()),
- dynamiczne ładowanie modułów programu i bibliotek dynamicznych,
- odwzorowanie plików w pamięci procesu w celu ułatwienia i przyspieszenia operacji na ich zawartości,
- współdzielenie przez wiele procesów obszarów pamięci, których zawartość przechowywana jest w odwzorowanych plikach niezależnie od tworzenia i kończenia procesów.

Funkcja systemowa **mmap**() tworzy odwzorowanie fragmentu lub całej zawartości pliku na obszar w pamięci wirtualnej procesu.

```
void mmap(void *start, size_t length, int prot, int flags, int fd,
off_t offset);
```

gdzie:

start - początkowy adres odwzorowanego obszaru w pamięci,

length - rozmiar odwzorowanego obszaru w bajtach,

prot - ochrona obszaru pamięci,

flags - flagi,

fd - deskryptor otwartego pliku,

offset - przesunięcie w pliku, określające początek obszaru, który ma być odwzorowany w pamięci.

Początek fragmentu pliku, który ma być odwzorowany, wskazywany jest przez argument **offset**. Jego długość określa argument **length**. Funkcja odwzorowuje ten fragment pliku w pamięci, począwszy od adresu **start**. Podany adres musi być **wyrównany do strony**, czyli wskazywać na początek strony w pamięci. Rozmiar strony jest uzależniony od platformy sprzętowej i może być odczytany funkcją **getpagesize**(). Najczęściej jednak podaje się wartość 0 (lub stałą symboliczną NULL) pozwalając, aby jądro systemu wybrało adres początkowy obszaru w pamięci. Adres ten jest zwracany jako wartość funkcji **mmap**().

Argument **prot** pozwala określić sposób ochrony odwzorowanego obszaru pamięci. Może przyjmować następujące wartości lub ich sumę bitową:

PROT_EXEC - strony obszaru mogą być wykonywane,

PROT_READ - strony obszaru mogą być odczytywane,

PROT_WRITE - strony obszaru mogą być zapisywane,

PROT_NONE - strony obszaru nie są dostępne.

Tryb dostępu do obszaru pamięci nie może być sprzeczny z trybem dostępu do otwartego pliku.

Znaczniki ustawione argumentem **flags** określają parametry odwzorowania pamięci. Najważniejsze z nich, opisane w standardzie POSIX, zamieszczono poniżej:

MAP_FIXED - odwzorowanie powinno nastąpić od wskazanego adresu **start** lub zakończyć się błędem,

MAP_PRIVATE - tworzy prywatne odwzorowanie pamięci dla procesu a wprowadzone w pamięci zmiany nie są zapisywane do pliku i nie są widoczne dla innych procesów,

MAP_SHARED - tworzy odworowanie współdzielone z innymi procesami, wszystkie wprowadzone w pamięci zmiany są zapisywane do pliku i stają się widoczne dla innych procesów.

Każde odworowanie wymaga podania jednej z flag: MAP_PRIVATE lub MAP_SHARED. Użycie konkretnej flagi ma znaczenie tylko dla obszarów z prawem do pisania.

Adres początkowy obszaru w pamięci i przesunięcie w pliku powinny być wielokrotnością rozmiaru strony. Funkcja **getpagesize**() zwraca rozmiar strony pamięci w bajtach.

```
size_t getpagesize(void);
```

Usunięcie odwzorowania pamięci realizuje funkcja **munmap**() poprzez odłączenie odpowiedniego obszaru od wirtualnej przestrzeni adresowej procesu.

```
int munmap(void *start, size_t lenght);
```

gdzie:

start - początkowy adres odwzorowanego obszaru w pamięci,

lenght - rozmiar odwzorowanego obszaru w bajtach.

Funkcja **msync**() umożliwia zsynchronizowanie zawartości obszaru pamięci z obrazem pliku na dysku. Bez wywołania tej funkcji nie ma pewności, że zmiany dokonane w pamięci zostaną zapisywane na dysku przed usunięciem odwzorowania.

```
int msync(const void *start, size_t lenght, int flags);
```

gdzie:

start - początkowy adres odwzorowanego obszaru w pamięci,

lenght - rozmiar odwzorowanego obszaru w bajtach,

flags - flagi.

Przykład

Prezentujemy program ilustrujący wykorzystanie funkcji mmap do odwzorowania pliku w pamięci procesu. Zadaniem programu jest modyfikowanie początkowych bajtów pliku.

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

```
#include <errno.h>
#include <fcntl.h>
int main(int argc, char *argv[])
{
  char *plik;
  int fd;
  struct stat iwezel;
  void *adres;
  switch(argc)
  {
    case 2: plik = argv[1];
    break;
    default : printf("Poprawna skladnia:\t%s plik\n", argv[0]);
    exit(1);
  }
  if ((fd = open(argv[1], O_RDWR)) == -1)
  {
    perror("Blad open");
    return 1;
  }
  printf("HALO\n");
  if (fstat(fd, &iwezel))
  {
    perror("Blad fstat");
    return 1;
  }
  printf("Rozmiar pliku: %d\n",iwezel.st_size);
  if ((adres = mmap(NULL, iwezel.st_size, PROT_READ|PROT_WRITE,
MAP_SHARED, fd, 0)) == (void *)-1)
  {
    perror("Blad mmap");
    return 1;
  }
  close(fd);
  write(1, adres, 1);
  write(1, adres+iwezel.st_size-2, 1);
  write(1, "\n", 1);
  memcpy(adres, "XXX", 3);
  munmap(adres, iwezel.st_size);
  return 0;
}
```

Bibliografia podstawowa

[1] Silberschatz A., Galvin P.B.: Podstawy systemów operacyjnych, WNT 2000 (rozdział 22)

[2] Glass G., Ables K.: Linux dla programistów i użytkowników, Wydawnictwo Helion 2007 (rozdziały: 12.3, 13.2, 13.5)

[3] Johnson M.K., Troan E.W.: Oprogramowanie użytkowe w systemie Linux, WNT 2000 (rozdziały: 10, 11, 12)

[4] Rochkind M.J.: Programowanie w systemie UNIX dla zaawansowanych, WNT 2007 (rozdziały: 2, 3)

Słownik

Termin	Objaśnienie
blok identyfikacyjny (ang. superblock)	blok dyskowy zawierający informacje w pełni opisujące aktualny stan systemu plików, przechowywanego na bieżącej partycji dyskowej
blok systemowy lub inicjujący (ang. boot block)	blok dyskowy, który może zawierać podstawowy program ładujący jądro systemu operacyjnego do pamięci operacyjnej w czasie startowania systemu
i-węzeł	struktura danych reprezentująca plik i przechowująca wszystkie jego atrybuty
podręczna pamięć buforowa	pamięć pośrednicząca w dostępie rzeczywistych systemów plików do blokowych urządzeń pamięci masowej
wirtualny system plików	jednolity interfejs pomiędzy rzeczywistymi systemami plików a programami

Zadania do wykładu 11

Zadanie 1

Napisać program implementujący uproszczoną wersję polecenia **cp**. Program nie przyjmuje żadnych opcji. Wywołanie programu ma postać:

```
cp plik_źródłowy plik_docelowy
```

Zadanie 2

Napisać program, który implementuje polecenie **ls**. Załóż, że wywołanie ma następującą postać:

```
ls [-opcje] [plik|katalog ...]
```

Program powinien akceptować dowolną kombinację opcji: -a, -l, -i (również sklejenia np. -al). Format wyświetlanych informacji może być dowolny.