LINUX
PROGRAMOWANIE SYSTEMOWE
[3/3]

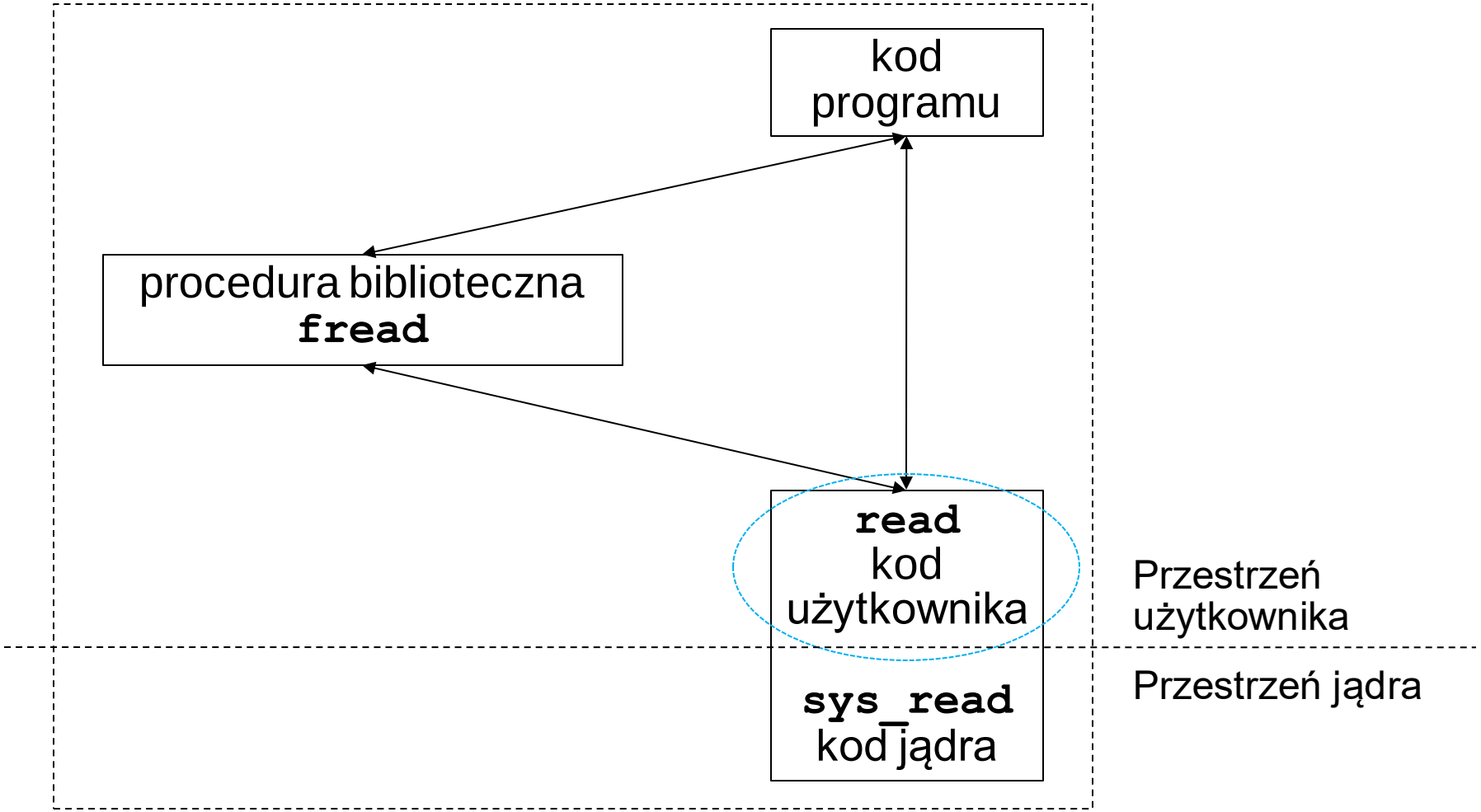
Plan wykładu

- Pliki
  - Podstawowe operacje na plikach
  - Pliki w środowisku wielu użytkowników
  - Pliki z wieloma nazwami
  - Informacje o plikach
  - Rozszerzona kontrola dostępu
- Katalogi
  - Implementacja katalogu
  - Podstawowe operacje na katalogach
- Potoki
  - Nienazwane
  - Nazwane
- Odwzorowanie plików w pamięci

Pliki

- **Pliki**
  - Podstawowe operacje na plikach
  - Pliki w środowisku wielu użytkowników
  - Pliki z wieloma nazwami
  - Informacje o plikach
  - Rozszerzona kontrola dostępu
- Katalogi
- Potoki
- Odwzorowanie plików w pamięci

Procedury biblioteczne a funkcje systemowe



Plan wykładu

- **Pliki**
  - **Podstawowe operacje na plikach**
  - Pliki w środowisku wielu użytkowników
  - Pliki z wieloma nazwami
  - Informacje o plikach
  - Rozszerzona kontrola dostępu
- Katalogi
- Potoki
- Odwzorowanie plików w pamięci

Funkcje **open**, **create** i **close**

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

int open( const char *pathname, int flags, [ mode_t mode]);
int creat( const char *pathname, mode_t mode);
int close( int filedes);
```

- Funkcja **open** przekształca ścieżkę na **deskryptor pliku** (liczbę całkowitą używaną w późniejszych operacjach we/wy, takich jak **read**, **write**, itd.).
- Zwrócony deskryptor pliku będzie najmniejszym **do tej pory** nie otwartym deskryptorem pliku dla tego procesu.
- Funkcja ta tworzy nowy otwarty plik, nie współdzielony z żadnym innym procesem.
- Przesunięcie pliku jest ustawiane na jego początek.

---

*wywołanie **creat** jest równoważne wywołaniu **open** z flagami
O_WRONLY|O_CREAT|O_TRUNC

W niektórych systemach funkcja **create** nie jest nawet wywołaniem systemowym, a jedynie funkcją biblioteczną odwołującą się pośrednio do wywołania **sys_open**.

`open` - parametr `flags` 1/2

- **`O_RDONLY`** Plik otwierany tylko do czytania
- **`O_WRONLY`** Plik otwierany tylko do pisania
- **`O_RDWR`** Plik otwierany tylko do czytania i pisania
- **`O_CREAT`** Jeśli plik nie istnieje, to będzie utworzony. Właściciel (ID użytkownika) tego pliku jest ustawiany na efektywny ID użytkownika procesu. Grupa właściciela (ID grupy) jest ustawiana albo na efektywny ID grupy procesu albo na ID grupy katalogu nadrzędnego
- **`O_EXCL`** Gdy zostanie użyte w połączeniu z **`O_CREAT`**, to jeśli plik już istnieje, **`open`** się nie powiedzie. W tym kontekście dowiązanie symboliczne jest istniejącym plikiem, niezależnie od tego, na co wskazuje. **`O_EXCL`** nie działa poprawnie na systemach plików NFS
- **`O_TRUNC`** Jeśli plik już istnieje, jest zwykłym plikiem i tryb otwarcia pozwala na zapis (tzn. jest to **`O_RDWR`** lub **`O_WRONLY`**), to plik ten zostanie obcięty do zerowej długości. Jeśli plik to FIFO lub urządzenie terminalowe, to znacznik **`O_TRUNC`** jest ignorowany. W pozostałych przypadkach efekt użycia znacznika **`O_TRUNC`** jest nieokreślony.
- **`O_APPEND`** Plik jest otwierany w trybie dopisywania. Przed każdą operacją **`write`**, wskaźnik pliku jest ustawiany na koniec pliku, jak z **`lseek`**. **`O_APPEND`** może prowadzić do zepsucia plików na systemach plików NFS, gdy więcej niż jeden proces naraz dopisuje dane do pliku. Jest to związane z faktem, że NFS nie wspiera dopisywania do pliku, więc jądro klienta musi to zasymulować, co nie może zostać wykonane **bez sytuacji wyścigu**.

`open` - parametr `flags` 2/2

- **`O_CLOEXEC`** Plik jest otwierany z ustawionym **znacznikiem zamykania** (close-on-exec). Plik otwarty w ten sposób zostanie automatycznie zamknięty, jeżeli proces wywoła z powodzeniem jedną z funkcji z rodziny **`exec`**.
- **`O_LARGEFILE`** Umożliwia pracę z plikami większymi niż 2GB.
- **`O_NOATIME+`** Czas ostatniego dostępu nie jest aktualizowany przy odczycie (w niektórych przypadkach może być przydatne ograniczenie ilości modyfikacji zawartości i-węzła - efektywność).
- **`O_NOFOLLOW`** Jeżeli otwierany plik jest dowiązaniem symbolicznym to jego otwarcie się nie powiedzie (normalnie otworzylibyśmy wskazywany obiekt docelowy).
- **`O_SYNC`** Plik jest otwierany dla zsynchronizowanych operacji wejścia i wyjścia. Ma większe znaczenie dla operacji wyjścia (zapisu na dysk) – operacja zapisu nie zakończy się dopóki dane nie znajdą się fizycznie na dysku (standardowo powrót z funkcji następuje wcześniej, dane są buforowane przez jądro). Flaga może mieć znaczący wpływ na obniżenie efektywności zapisów – nawet o dwa rzędy wielkości.
- **`O_NONBLOCK`** Plik jest otwierany w trybie nieblokującym, tzn. ani funkcja **`open`** ani inne operacje wejścia i wyjścia związane z otwartym deskryptorem nie będą blokowały (usypiały) procesu. Flaga ma znaczenie jedynie przy otwieraniu potoków nazwanych (kolejek FIFO).

`open` - parametr `mode`

- **`S_IRWXU`** 00700 użytkownik (właściciel pliku) ma prawa odczytu, zapisu i uruchamiania.
- **`S_IRUSR`** 00400 użytkownik ma prawa odczytu.
- **`S_IWUSR`** 00200 użytkownik ma prawa zapisu.
- **`S_IXUSR`** 00100 użytkownik ma prawa uruchamiania.
- **`S_IRWXG`** 00070 grupa ma prawa odczytu, zapisu i uruchamiania.
- **`S_IRGRP`** 00040 grupa ma prawa odczytu.
- **`S_IWGRP`** 00020 grupa ma prawa zapisu.
- **`S_IXGRP`** 00010 grupa ma prawa uruchamiania.
- **`S_IRWXO`** 00007 inni mają prawa odczytu, zapisu i uruchamiania.
- **`S_IROTH`** 00004 inni mają prawa odczytu.
- **`S_IWOTH`** 00002 inni mają prawa zapisu.
- **`S_IXOTH`** 00001 inni mają prawa uruchamiania.

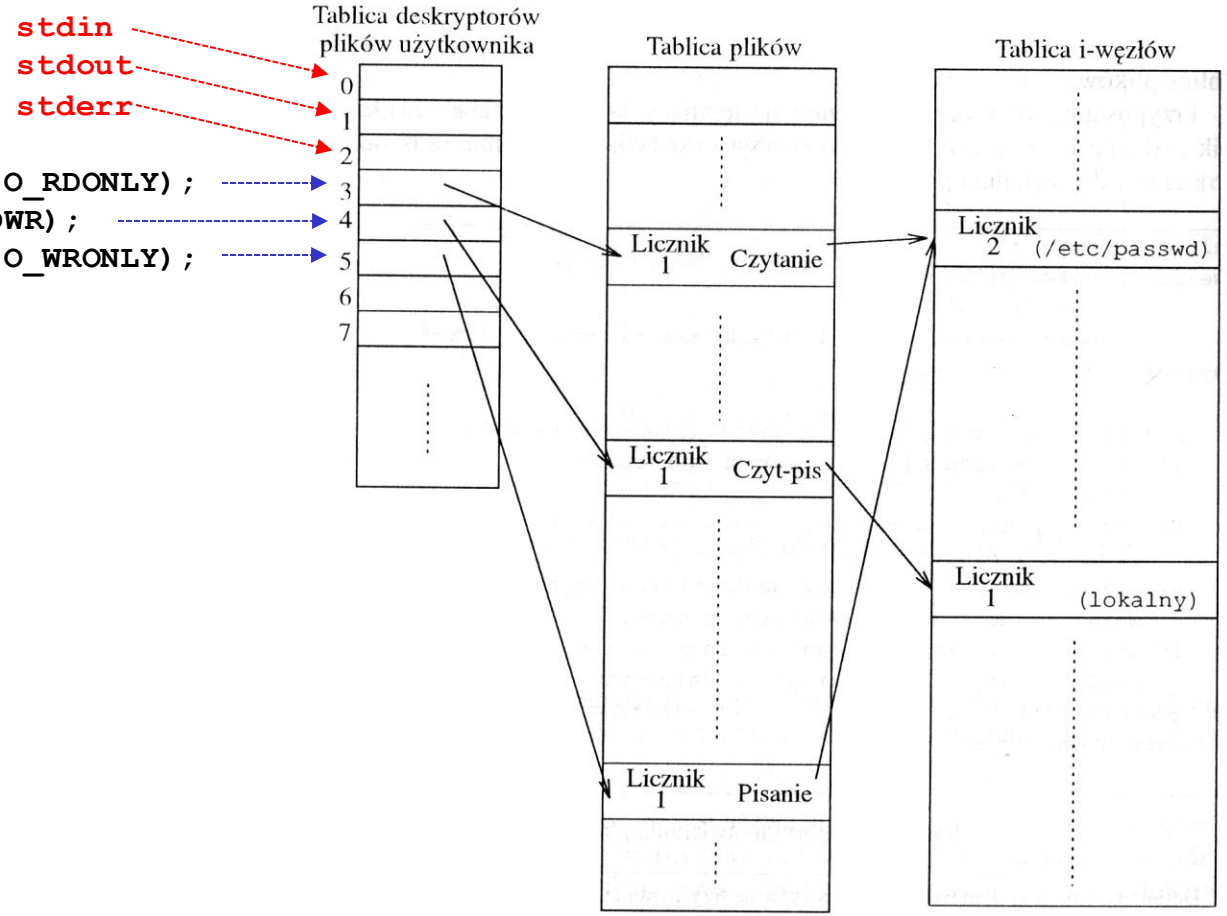
Otwieranie i zamykanie pliku - przykład

test-open.c

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

int main ()
{
  int p1, p2, p3, p4;
  p1 = open( "test_1", O_RDONLY);
  p2 = open( "test_2", O_CREAT | O_TRUNC | O_RDWR, 0640 );
  p3 = open( "test_3", O_CREAT | O_EXCL | O_WRONLY, 0664 );
  p4 = creat( "test_4", 0770 );
  ...
  close( p1);
  close( p2);
  close( p3);
  close( p4);
  return 0;
}
```

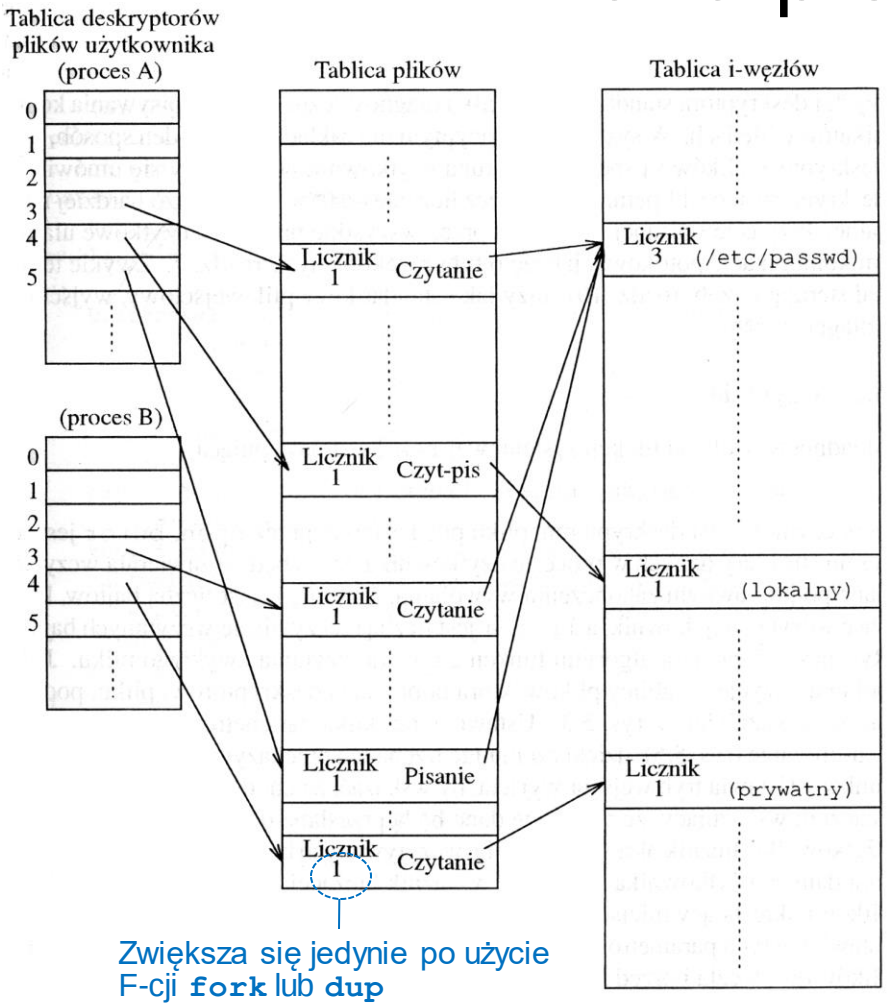
Struktury danych po wywołaniu `open`

```
dp1 = open("/etc/passwd", O_RDONLY);
dp2 = open("lokalny", O_RDWR);
dp3 = open("/etc/passwd", O_WRONLY);
```



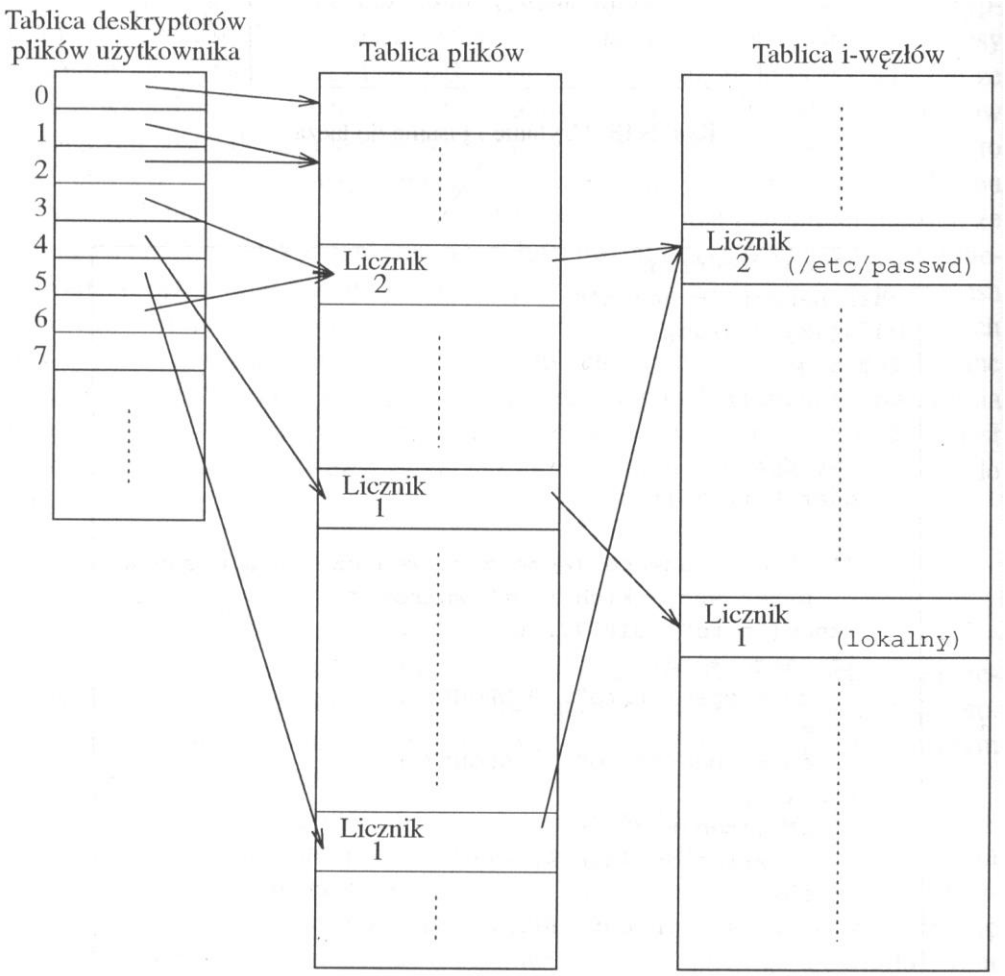
„Budowa systemu operacyjnego UNIX" J.Bach

Struktury danych po otwarciu plików przez dwa procesy




```
Proces A:
dp1 = open("/etc/passwd", O_RDONLY);
dp2 = open("lokalny", O_RDWR);
dp3 = open("/etc/passwd", O_WRONLY);

Proces B:
dp1 = open("/etc/passwd", O_RDONLY);
dp2 = open("prywatny", O_RDONLY);
```

Struktury danych po wywołaniu **dup**




```
nowydp = dup( dp);
```

Funkcje **read**, **write**, **lseek**

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

ssize_t read(  int filedes, void *buffer, size_t n );        // odczyt z pliku
ssize_t write( int filedes, void *buffer, size_t n );        // zapis do pliku
```

deskryptor pliku — bufor danych — ile bajtów

```
off_t lseek( int filedes, off_t offset, int start_flag );    // przesunięcie wskaźnika
```

deskryptor pliku — o ile bajtów — od jakiego miejsca

```
SEEK_SET
SEEK_CUR
SEEK_END
```

wskaźnik pliku = wskaźnik odczytu = wskaźnik zapisu

```
ssize_t pread(  int filedes, void *buffer, size_t n, off_t offset );
ssize_t pwrite( int filedes, void *buffer, size_t n, off_t offset );
```

Właściwości `read`, `write`, `lseek` dla plików zwykłych

- Po pomyślnym zakończeniu **`read`** zwracana jest liczba odczytanych bajtów (zero oznacza koniec pliku), oraz o tę wartość przesuwana jest pozycja w pliku. Nie jest błędem, jeśli liczba ta jest mniejsza niż liczba żądanych bajtów.
- Po pomyślnym zakończeniu **`write`** zwracana jest liczba zapisanych bajtów (zero oznacza nie zapisanie niczego), oraz o tę wartość przesuwana jest pozycja w pliku.
- Po pomyślnym zakończeniu **`lseek`** zwraca ustawione przesunięcie, liczone w bajtach od początku pliku (jak sprawdzić wielkość pliku?).
- **Dla plików specjalnych wyniki są nieprzenośne!**
- Przy dużych plikach najlepsza wydajność gdy odczyt i zapis wykonywane są paczkami będącymi wielokrotnością bloków dyskowych w aktualnym systemie plików

Odczyt i przesunięcie w pliku - przykład

test-read.c

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

char buff[ 4];
unsigned int a, retVal;

int main () {
  int p1 = open( "test_1", O_RDONLY);
  if( p1 != -1) {
    retVal = read( p1, buff, sizeof( buff));
    retVal = lseek( p1, 0, SEEK_SET);
    retVal = read( p1, &a, sizeof( a));
    retVal = lseek( p1, -sizeof( a), SEEK_CUR);
    close( p1);
  }
  return 0;
}
```

Opóźniony zapis i synchronizacja danych

- Standardowo funkcje zapisu wracają **nie czekając**, aż dane i metadane nie zostaną fizycznie zapisane na dysku.
- **Jądro buforuje** zapisywane dane i w optymalnym dla siebie momencie dokonuje zapisu (synchronizacji) „zabrudzonych" i-węzłów i bloków danych.
- Co z odczytem niezsynchronizowanych jeszcze danych? (czytanie z bufora).
- Co w przypadku błędu zapisu na dysk? (brak możliwości zwrotu informacji do procesu – dla niego zapis się zakończył).

- Możliwość jawnego synchronizowania danych i metadanych wskazanego pliku (co najmniej dwa zapisy dyskowe – czemu?):

```
int fsync( int fd); /* synchronizacja danych (bloki
danych) i metadanych (atrybuty i-węzła) - funkcja czeka
na ich zakończenie */

int fdatasync( int fd); /* jak fsync, ale nie są
aktualizowane wszystkie atrybuty (tylko te niezbędne do
odczytu zawartości pliku) */
```

Kasowanie pliku **unlink**, **remove**

- **int unlink(const char *pathname);**
- **int remove(const char *pathname);**

- **unlink** **usuwa nazwę** z systemu plików (zmniejsza licznik dowiązań sztywnych do i-węzła).
- Jeżeli nazwa była **ostatnim dowiązaniem** do pliku to jest on fizycznie usuwany z systemu plików.
- Jeżeli nazwa była ostatnim dowiązaniem do pliku ale istnieje proces, który w danej chwili używa ten plik, to jest on tylko **zaznaczany do usunięcia** (usunięcie opóźnione do momentu zwolnienia ostatniego deskryptora).
- Jeżeli nazwa dotyczy **dowiązania symbolicznego** to usuwane jest to dowiązanie.
- **remove** usuwa nazwę z systemu plików. Odwołuje się bezpośrednio do **unlink** dla plików lub **rmdir** dla katalogów.

Kontrola otwartego pliku **fcntl** 1/4

- **int fcntl(int *fd*, int *cmd*);**
- **int fcntl(int *fd*, int *cmd*, long *arg*);**
- **int fcntl(int *fd*, int *cmd*, struct flock **lock*);**

- **fcntl** dokonuje jednej z wielu różnych operacji na deskryptorze *fd*. Wykonywana operacja zdeterminowana jest przez *cmd*.

- **F_GETFD** Odczytanie znacznika “zamknięcia przy uruchomieniu” (close-onexec). Jeśli bit **FD_CLOEXEC** jest równy 0, to plik pozostanie otwarty po wykonaniu **exec**, w przeciwnym przypadku zostanie zamknięty.
- **F_SETFD** Nadanie znacznikowi “zamknięcia przy uruchomieniu” (close-onexec) wartości określonej przez bit **FD_CLOEXEC** zmiennej *arg*.

Kontrola otwartego pliku `fcntl` 2/4

- **`F_NOTIFY`** Zapewnia powiadamianie o **modyfikacji katalogu**, do którego odnosi się *fd* lub o **modyfikacji któregokolwiek z plików w tym katalogu**. Zdarzenia, powiadamianie o których ma nastąpić, są określone w *arg*, będącym maską bitową utworzoną jako suma logiczna (OR) zera lub więcej spośród następujących bitów:

`DN_ACCESS`	Dostęp do pliku (read, pread, readv)
`DN_MODIFY`	Modyfikacja pliku (write, pwrite, writev, truncate, ftruncate)
`DN_CREATE`	Utworzenie pliku (open, creat, mknod, mkdir, link, symlink, rename)
`DN_DELETE`	Usunięcie pliku (unlink, rename do innego katalogu, rmdir)
`DN_RENAME`	Zmiana nazwy w obrębie katalogu (rename)
`DN_ATTRIB`	Zmiana atrybutów pliku (chown, chmod, utime[s])

- Powiadomienia są domyślnie **jednorazowe**, więc aplikacja musi się ponownie zarejestrować, aby otrzymać dalsze powiadomienia. Alternatywnie, jeśli w *arg* włączono **`DN_MULTISHOT`**, to powiadomienia będą dokonywane aż do ich jawnego usunięcia. Aby wyłączyć powiadamianie o jakichkolwiek zdarzeniach, należy w wywołaniu **`F_NOTIFY`** podać *arg* równe 0.
- Powiadamianie odbywa się poprzez dostarczenie sygnału. Domyślnym sygnałem jest **`SIGIO`**, ale można go zamienić za pomocą polecenia **F_SETSIG** w **fcntl**().

Kontrola otwartego pliku `fcntl` 3/4

- **`F_SETLEASE`** i **`F_GETLEASE`** ustanowienia i pobrania aktualnego ustawienia dzierżawy na pliku określonym przez *fd* dla procesu wywołującego funkcję. Dzierżawa pliku zapewnia mechanizm, w którym proces utrzymujący dzierżawę ("dzierżawca") jest zawiadamiany (poprzez dostarczenie sygnału) o tym, że inny proces ("współzawodnik") próbuje wykonać **`open`** lub **`truncate`** na tym pliku. **Proces może utrzymywać na pliku dzierżawę tylko jednego typu.**
- **`F_SETLEASE`** Ustawia lub usuwa dzierżawę pliku w zależności od tego, która z następujących wartości zostanie podana jako argument *arg* typu integer :
  - **`F_RDLCK`** Wzięcie dzierżawy odczytu. Spowoduje to zawiadamianie o otwarciu pliku do zapisu lub jego obcięciu przez inny proces.
  - **`F_WRLCK`** Wzięcie dzierżawy zapisu. Spowoduje to zawiadamianie o otwarciu pliku (do odczytu lub do zapisu) lub obcięciu go przez inny proces. Dzierżawa zapisu może zostać nałożona na plik tylko wtedy, gdy plik ten nie jest aktualnie otwarty przez żaden inny proces.
  - **`F_UNLCK`** Zdjęcie własnej dzierżawy z pliku.
- **`F_GETLEASE`** Wskazuje rodzaj dzierżawy utrzymywanej przez aktualny proces na pliku określonym przez deskryptor *fd*, zwracając **`F_RDLCK`**, **`F_WRLCK`** albo **`F_UNLCK`**, w zależności od tego, czy (odpowiednio) aktualny proces utrzymuje dzierżawę odczytu, zapisu, czy nie utrzymuje żadnej dzierżawy na danym pliku.
- Domyślnym sygnałem stosowanym do zawiadamiania dzierżawcy jest **`SIGIO`**, lecz można go zmienić za pomocą polecenia **`F_SETSIG`** w **`fcntl()`.**

Kontrola otwartego pliku **fcntl** 4/4

- **F_GETLK**, **F_SETLK** i **F_SETLKW** służą odpowiednio do sprawdzania, zakładania i zakładania z czekaniem **blokad rekordów** (znanych również jako blokady segmentów lub obszarów pliku). Trzeci argument, **lock**, jest wskaźnikiem do struktury zawierającej co najmniej następujące pola (kolejność nie jest określona).

```
struct flock {
  ...
  short l_type; /* Rodzaj blokady: F_RDLCK, F_WRLCK, F_UNLCK */
  short l_whence; /* Sposób interpretacji l_start: SEEK_SET, SEEK_CUR, SEEK_END */
  off_t l_start; /* Początek (offset) blokady */
  off_t l_len; /* Liczba blokowanych bajtów */
  pid_t l_pid; /* PID procesu uniemożliwiającego blokadę (tylko F_GETLK) */
  ...
};
```

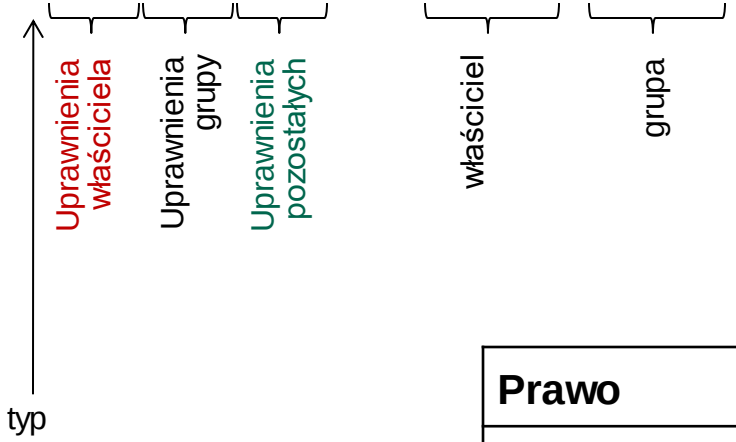
- Blokady są usuwane w wyniku jawnego **F_UNLCK**, jak też są one automatycznie zwalniane gdy proces kończy działanie lub **zamyka *dowolny* deskryptor** odnoszący się do pliku, na którym blokady są utrzymywane.

Plan wykładu

- **Pliki**
  - Podstawowe operacje na plikach
  - **Pliki w środowisku wielu użytkowników**
  - Pliki z wieloma nazwami
  - Informacje o plikach
  - Rozszerzona kontrola dostępu
- Katalogi
- Potoki
- Odwzorowanie plików w pamięci

Uprawnienia i właściciele

```
-rw-r--r-- 1 root root 31624 kwi 7 2006 agpgart.ko
drwxr-xr-x 3 root root 4096 lut 11 2005 app-portage
drwxr-xr-x 2 root root 4096 gru 11 2005 arsene
-rw-r--r-- 1 root root 0 maj 26 20:39 CHG
-rw-r--r-- 1 root root 0 maj 26 20:39 CHGCAR
```

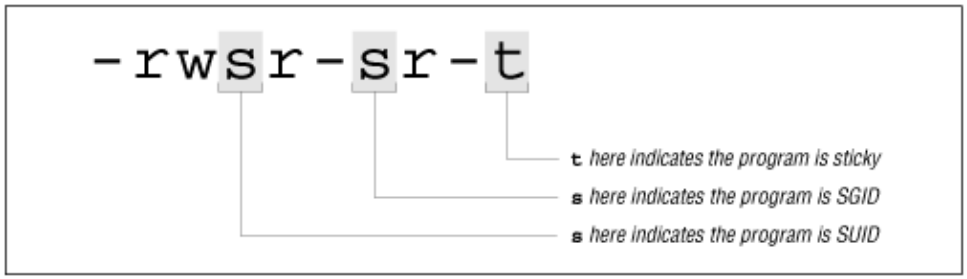


Prawo	Plik	Katalog
r	**czytania zawartości**	przeszukania zawartości
w	**zmiany zawartości**	zmiany zawartości
x	**wykonywanie**	przejścia do tego katalogu

Metody zapisu uprawnień

Prawa dostępu	**Zapis numeryczny**	**Zapis znakowy**
Tylko do czytania	**4 (100)**	**r--**
Tylko do pisania	**2 (010)**	**-w-**
Tylko do wykonywania	**1 (001)**	**--x**
Do czytania i pisania	**6 (110)**	**rw-**
Do czytania i wykonywania	**5 (101)**	**r-x**
Czytania, pisania i wykonywania	**7 (111)**	**rwx**

Dodatkowe uprawnienia dla plików wykonywalnych



- **`---s------`**
- SUID - proces uruchamiający dany program przejmie identyfikator właściciela programu jako swój obowiązujący efektywny identyfikator użytkownika.

- **`------s---`**
- SGID - proces uruchamiający dany program przejmie identyfikator grupy właściciela programu jako swój obowiązujący efektywny identyfikator grupy.

- **`---------t`**
- *sitcky bit* (bit lepkości) ustawiony w pliku oznacza, że program po zakończeniu procesu nie jest zwalniany z pamięci (przestarzałe, w systemie Linux ignorowane).
- W katalogu z ustawionym *sticky bit* pliki mogą być usuwane lub przemianowywane tylko przez właściciela pliku, właściciela owego katalogu lub administratora (root).

Maska tworzenia pliku **umask**

- `mode_t umask( mode_t maska);`
- F-cja modyfikuje aktualną maskę **UMASK** procesu według schematu: ***mask*** `& 0777`.
- F-cja zwraca wartość maski sprzed zmiany.
- Maska jest wykorzystywana m.in. przez **open** do nadawania nowoutworzonym plikom początkowych praw dostępu (prawa z maski są wyłączane z praw podanych w **open**).

```
UMASK = 022          open( "test", O_CREAT, 0666);
```

```
test        rw- r-- r--        (0644)
```

Dostępność pliku **access**

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

- **access** sprawdza, czy proces może odczytywać, zapisywać i sprawdzać istnienie pliku (lub innego obiektu systemu plików) o nazwie *pathname*.
- Jeśli *pathname* jest dowiązaniem symbolicznym, sprawdzane są prawa do pliku wskazywanego przez to dowiązanie.
- *mode* jest maską składającą się z jednego lub więcej znaczników spośród **R_OK**, **W_OK**, **X_OK** i **F_OK**.
- **R_OK**, **W_OK** i **X_OK** sprawdzają, czy plik istnieje i ma odpowiednio prawa do odczytu, zapisu i uruchamiania. **F_OK** sprawdza tylko czy plik istnieje.

- Gdy wszystko pójdzie dobrze (wszystkie żądane prawa są zapewnione), zwracane jest **zero**. W wypadku błędu (przynajmniej jeden bit z żądanych w *mode* uprawnień nie jest ustawiony lub wystąpiły inne błędy), zwracane jest -1

- Sprawdzenie jest dokonywane **z rzeczywistymi uid i gid** procesu, a nie efektywnymi, jak to się zwykle robi przy wykonywaniu rzeczywistych operacji.

Zmiana uprawnień **chmod**

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

- Zmienione zostają prawa dostępu do pliku określonego przez *path*
- Prawa są podawane jako *or* następujących wartości:
  - **`S_ISUID`** **04000** ustawia ID użytkowinka przy uruchomieniu
  - **`S_ISGID`** **02000** ustawia ID grupy przy uruchomieniu
  - **`S_ISVTX`** **01000** bit lepkości
  - **`S_IRUSR`** (**`S_IREAD`**) **00400** odczyt przez właściciela
  - **`S_IWUSR`** (**`S_IWRITE`**) **00200** zapis przez właściciela
  - **`S_IXUSR`** (**`S_IEXEC`**) **00100** uruchomianie/przeszukiwanie przez właściciela
  - **`S_IRGRP`** **00040** odczyt przez grupę
  - **`S_IWGRP`** **00020** zapis przez grupę
  - **`S_IXGRP`** **00010** uruchomianie/przeszukiwanie przez grupę
  - **`S_IROTH`** **00004** odczyt przez pozostałych
  - **`S_IWOTH`** **00002** zapis przez pozostałych
  - **`S_IXOTH`** **00001** uruchomianie/przeszukiwanie przez pozostałych

- Efektywny UID procesu wywołującego powyższą funkcję musi być zerem, lub odpowiadać właścicielowi pliku.

Zmiana właściciela **chown**

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

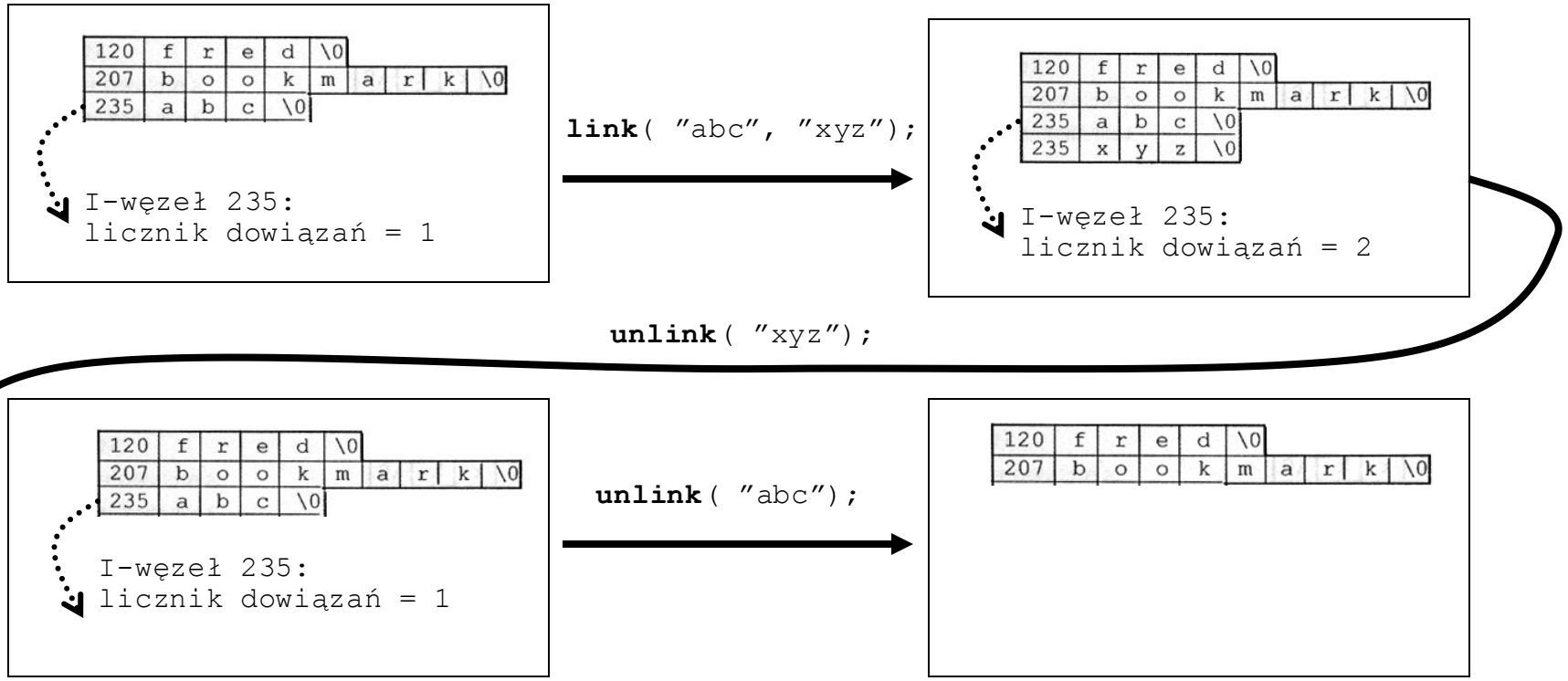
- Zmieniony zostaje właściciel pliku określonego przez *path*.
- **Tylko superużytkownik może zmieniać właściciela pliku!**
- Właściciel pliku może zmieniać tylko grupę pliku na dowolną grupę, której jest członkiem.
- **Superużytkownik może zmieniać grupę bez ograniczeń.**
- Jeśli argument ***owner*** lub ***group*** jest ustawiony jako **-1**, to identyfikator ten nie będzie zmieniany.

Pliki z wieloma nazwami

- **Pliki**
  - Podstawowe operacje na plikach
  - Pliki w środowisku wielu użytkowników
  - **Pliki z wieloma nazwami**
  - Informacje o plikach
  - Rozszerzona kontrola dostępu
- Katalogi
- Potoki
- Odwzorowanie plików w pamięci

Łącza twarde i symboliczne

- Łącze twarde - f-cje *link* i *unlink*



- Łącze symboliczne - f-cje *symlink* i *readlink*

Dowiązanie sztywne `link`

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

- `link` tworzy nowe dowiązanie (nazywane też dowiązaniem twardym lub sztywnym) do istniejącego pliku.
- Jeśli plik *newpath* już istnieje, to **nie będzie** nadpisany.
- Nowa nazwa może być używana dokładnie tak samo jak stara w dowolnych operacjach - obie nazwy odnoszą się do tego samego pliku i w związku z tym mają te same prawa i właścicielstwo. Nie można też powiedzieć, która nazwa jest '**oryginalna**'.
- Twarde dowiązania, tworzone z pomocą `link`, nie mogą wykraczać poza **jeden system plików** ani odnosić się do obiektu **katalogu**. W takich sytuacjach można użyć funkcji `symlink`.

Dowiązanie symboliczne **symlink**

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

- **`symlink`** tworzy dowiązanie symboliczne o nazwie *newpath* które zawiera łańcuch znakowy *oldpath*.
- Dowiązania symboliczne są interpretowane w czasie działania, tak jakby zawartość dowiązania była podstawiana do ścieżki, przeglądanej by znaleźć plik lub katalog.
- Dowiązania symboliczne może zawierać fragmenty ścieżki względnej.
- Dowiązanie symboliczne może wskazywać na plik istniejący, lub nie istniejący (wiszące dowiązanie).
- Prawa dostępu dla dowiązania symbolicznego są **nieistotne** - jego właścicielstwo jest ignorowane podczas podążania za nim, lecz sprawdzane podczas usuwania lub przemianowywania, gdy dowiązanie jest w katalogu z ustawionym bitem ‘sticky’.
- Jeśli ścieżka *newpath* istnieje to **nie będzie** nadpisana.

Usuwanie dowiązania `unlink`

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

- `unlink` usuwa nazwę z systemu plików (zmniejsza licznik dowiązań sztywnych do i-węzła).
- Jeżeli nazwa była ostatnim dowiązaniem do pliku to jest on fizycznie usuwany z systemu plików.
- Jeżeli nazwa była ostatnim dowiązaniem do pliku ale istnieje proces, który w danej chwili używa ten plik, to jest on tylko zaznaczany do usunięcia (usunięcie opóźnione do momentu zwolnienia ostatniego deskryptora).
- Jeżeli nazwa dotyczy dowiązania symbolicznego to usuwane jest to dowiązanie.
- Jeżeli nazwa odnosi się do wiązania symbolicznego to jest ono usuwane.

Odczyt dowiązania **readlink**

- **size_t readlink(const char *pathname, char *buf, size_t bufsize);**

- **readlink** umieszcza w zmiennej **buf** zawartość dowiązania symbolicznego wskazanego argumentem **pathname**
- Zawartość dowiązania (czyli łańcuch znaków zawierający ścieżkę wskazującą na plik docelowy) obcinana jest przed umieszczeniem w buforze do podanego rozmiaru **bufsize**
- Funkcja nie dodaje na końcu łańcucha zera kończącego string

Informacje o plikach

- **Pliki**
  - Podstawowe operacje na plikach
  - Pliki w środowisku wielu użytkowników
  - Pliki z wieloma nazwami
  - **Informacje o plikach**
  - Rozszerzona kontrola dostępu
- Katalogi
- Potoki
- Odwzorowanie plików w pamięci

Informacje o plikach **stat**, **fstat** (1/3)

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

- Funkcje zwracają informacje o podanym pliku. Do uzyskania tej informacji nie są wymagane prawa dostępu do samego pliku, lecz w przypadku **stat** konieczne są prawa wykonywania (przeszukiwania) do wszystkich katalogów na prowadzącej do pliku ścieżce *path*.

Informacje o plikach **stat**, **fstat** (2/3)

```
struct stat {
    dev_t st_dev;       /* ID urządzenia zawierającego plik */
    ino_t st_ino;       /* numer i-węzła (inode) */
    mode_t st_mode;     /* ochrona */
    nlink_t st_nlink;   /* liczba dowiązań stałych */
    uid_t st_uid;       /* ID użytkownika właściciela */
    gid_t st_gid;       /* ID grupy właściciela */
    dev_t st_rdev; /* ID urządzenia (jeśli plik specjalny) */
    off_t st_size;      /* całkowity rozmiar w bajtach */
    blksize_t st_blksize; /* wielkość bloku dla I/O systemu
      plików */
    blkcnt_t st_blocks; /* liczba zaalokowanych bloków */
    time_t st_atime;    /* czas ostatniego dostępu */
    time_t st_mtime;    /* czas ostatniej modyfikacji */
    time_t st_ctime;    /* czas ostatniej zmiany */
};
```

Informacje o plikach **`stat`**, **`fstat`** (3/3)

- Zdefiniowane są następujące makra sprawdzające typ pliku przy użyciu pola *st_mode*:
- **`S_ISREG(m)`** czy plik jest regularny?
- **`S_ISDIR(m)`** katalog?
- **`S_ISCHR(m)`** urządzenie znakowe?
- **`S_ISBLK(m)`** urządzenie blokowe?
- **`S_ISFIFO(m)`** kolejka FIFO (potok nazwany)?
- **`S_ISLNK(m)`** dowiązanie symboliczne? (Nie w POSIX.1-1996).
- **`S_ISSOCK(m)`** gniazdo? (Nie w POSIX.1-1996).

Plan wykładu

- **Pliki**
  - Podstawowe operacje na plikach
  - Pliki w środowisku wielu użytkowników
  - Pliki z wieloma nazwami
  - Informacje o plikach
  - **Rozszerzona kontrola dostępu**
- Katalogi
- Potoki
- Odwzorowanie plików w pamięci

ACL (1/4)

Podstawowym mechanizmem rozszerzającym kontrolę dostępu do plików jest ACL (ang. Access Control List). Poniżej przedstawiono na przykładzie dystrybucji Ubuntu proces instalacji i konfiguracji tego mechanizmu (w podstawowej wersji POSIX).

1. Zainstalować pakiety:

```
$ apt-get install acl
$ apt-get install acl-dev
$ apt-get install eiciel
```

2. Edytować plik **/etc/fstab** dodając opcję montowania **acl** dla wybranego systemu plików, np.:

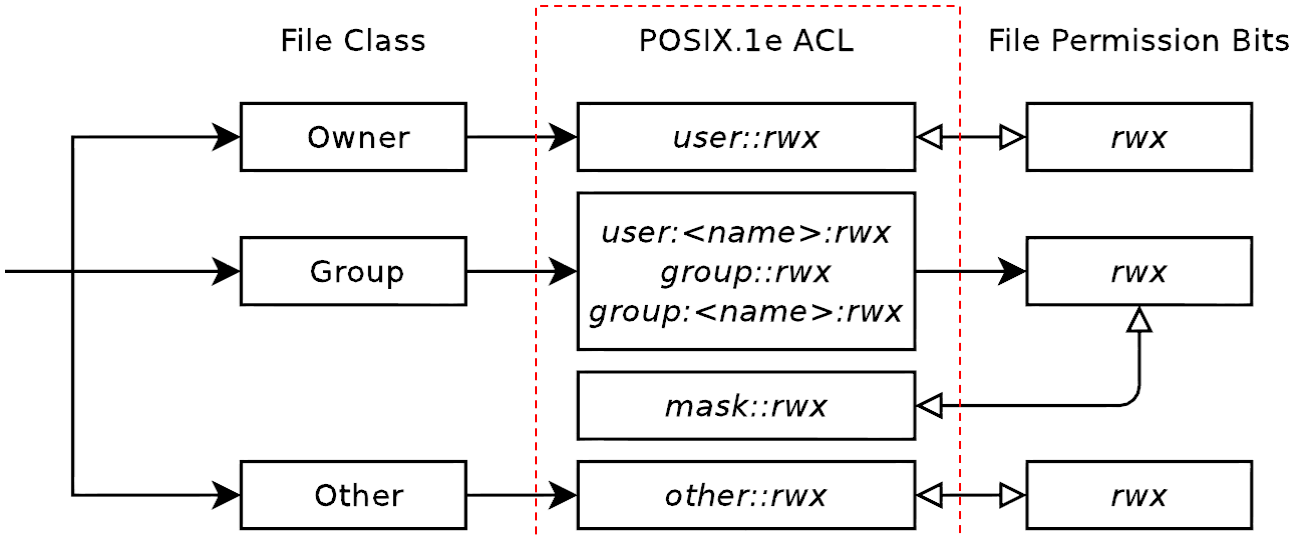
```
…
# device name   mount point     fs-type      options                dump-freq pass-num
LABEL=/         /               ext3         defaults               1 1
…
```

```
…
# device name   mount point     fs-type      options                dump-freq pass-num
LABEL=/         /               ext3         acl,defaults           1 1
…
```

3. Ponownie zamontować system plików:

```
$ mount –o remount /
```

ACL (2/4)



Typ	Forma tekstowa
owner	user::rwx
nemed user	user:name:rwx
owning group	group::rwx
named group	group:name:rwx
mask	mask::rwx
other	other::rwx

ACL (3/4)

1. Przykład efektu działania maski

typ	tekst	prawa dostępu
named user	user:jane:r-x	r-x
mask	mask::rw-	rw-
	affective permissions:	r--

2. Przykład wykorzystania poleceń **`getfacl`** i **`setfacl`**

```
$ mkdir mydir
$ getfacl mydir
# file: mydir/
# owner: root
# group: root
user::rwx
group::r-x
other::r-x
$ setfacl -m user:wmackow:rwx mydir
$ getfacl mydir
# file: mydir/
# owner: root
# group: root
user::rwx
user:wmackow:rwx
group::r-x
other::r-x
```

ACL (4/4)

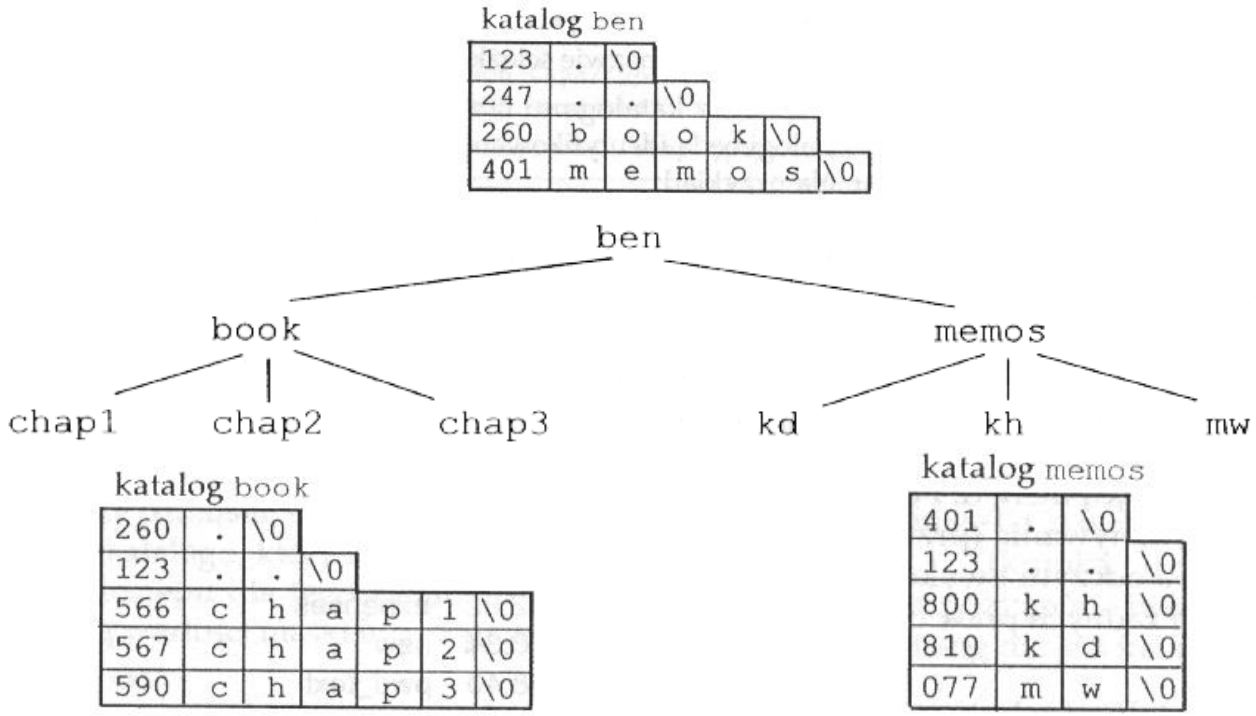
testACL.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/acl.h>
#include <acl/libacl.h>

int main( int ac, char **av)
{
    if( ac > 1)
    {
     acl_t aclSet;
     char *aclString;
     int retVal;

     aclSet = acl_get_file( av[ 1], ACL_TYPE_ACCESS);
     if( aclSet)
     {
          aclString = acl_to_any_text( aclSet, NULL, '\n', TEXT_ABBREVIATE);
          printf( "%s\n", aclString);
          acl_free( aclString);
          acl_free( aclSet);
     }
    }
    return 0;
}
```

Implementacja katalogu

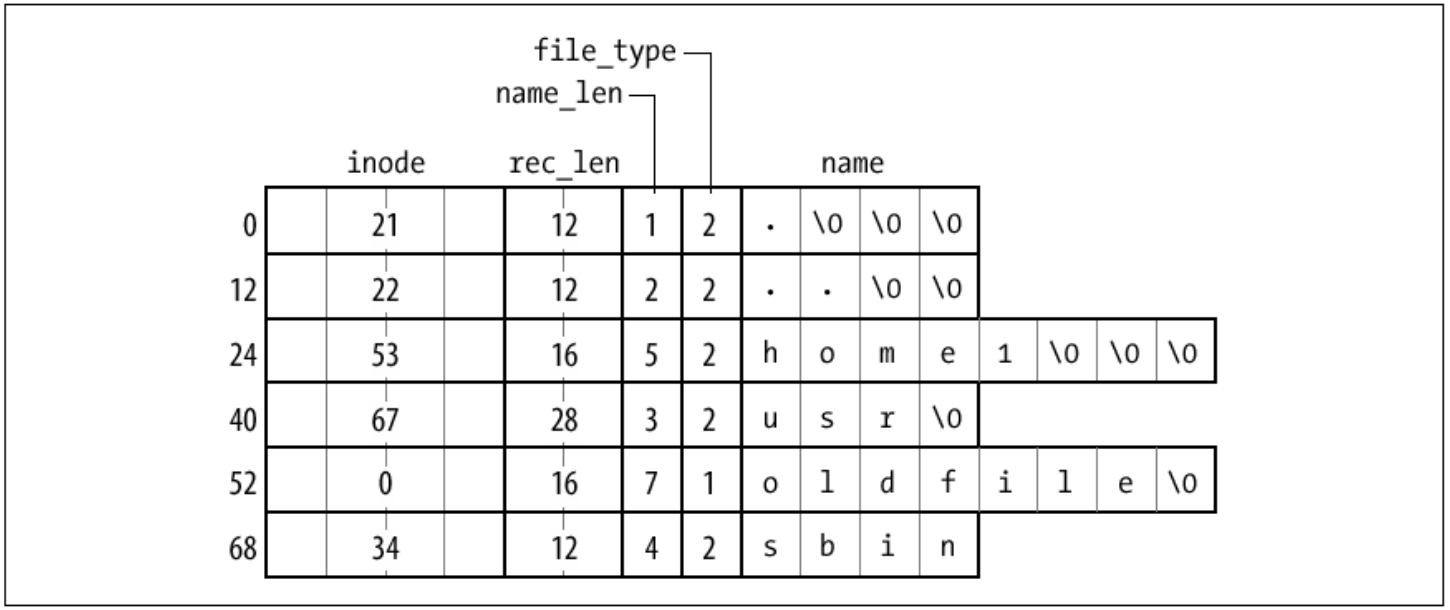
- Pliki
- **Katalogi**
  - **Implementacja katalogu**
  - Podstawowe operacje na katalogach
- Potoki
- Odwzorowanie plików w pamięci

Implementacja katalogu (system indeksowy)



Prawo	Plik	Katalog
r	czytania zawartości	**przeszukania zawartości**
w	zmiany zawartości	**zmiany zawartości**
x	wykonywanie	**przejścia do tego katalogu**

Przykład katalogu w systemie plików ext2



Pole **rec_len** wskazuje na następną poprawną pozycję w katalogu: stanowi offest, który należy dodać do adresu początkowego pozycji katalogu, aby otrzymać adres początkowy następnej prawidłowej pozycji.

Pozycja ***oldfile*** została skasowana (**inode** równa się 0), stąd **rec_len** wpisu *usr* jest ustawione na 12+16 (długość pozycji *usr* i *oldfile*)

Podstawowe operacje na katalogach

- Pliki
- **Katalogi**
  - Implementacja katalogu
  - **Podstawowe operacje na katalogach**
- Potoki
- Odwzorowanie plików w pamięci

Podstawowe funkcje

- Większość funkcji systemowych do operowania na plikach może być również używana do manipulowania katalogami (chociaż nie jest to wskazane) - **open, read, lseek, fstat, close**
- Nie można za pomocą tych f-cji ingerować w zawartość katalogów - nie można więc używać **creat, write** i **open** z parametrami O_WRONLY, O_RDWR lub O_CREAT

- Funkcje dedykowane do obsługi katalogów:

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

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

DIR *opendir( const char *dirname);
int closedir( DIR *dirptr);

struct dirent *readdir( DIR *dirptr);
void rewinddir( DIR *dirptr);
```

Tworzenie i usuwanie katalogów

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

- **mkdir** próbuje utworzyć katalog *pathname*. Argument *mode* prawa dostępu uwzględniana jest maska *umask* procesu

- **rmdir** usuwa pusty katalog.

Otwieranie i zamykanie katalogów

- `DIR *opendir( const char *name);`
- `int closedir( DIR *dir);`

- `opendir` otwiera strumień katalogu wskazanego nazwą *name* i zwraca wskaźnik do tego strumienia. Strumień jest ustawiany na pierwszym wpisie w katalogu.
- w przypadku błędu funkcja zwraca NULL

- `closedir` zamyka strumień katalogu.

Czytanie katalogów

- **`struct dirent *readdir(DIR *dir);`**
- **`readdir`** zwraca wskaźnik do struktury typu **`dirent`** opisujący wpis w katalogu *dir* i przesuwa wskaźnik pozycji w katalogu na kolejny wpis. Funkcja zwraca NULL po dojściu do końca katalogu.

```
struct dirent {
    ino_t d_ino; /* inode number */
    off_t d_off; /* offset to the next dirent */
    unsigned short d_reclen; /* length of this record */
    unsigned char d_type; /* type of file */
    char d_name[256]; /* filename */
 };
```

- **`void rewinddir( DIR *dir);`**
- **`rewinddir`** resetuje wskaźnik pozycji w strumieniu katalogu *dir* (ustawia wskaźnik na pierwszym wpisie).

Bieżący katalog roboczy

- `int chdir( const char *path);`
- Funkcja zmienia katalog bieżący na katalog podany w *path*.

- `char *getcwd(char *buf, size_t size);`
- Funkcja kopiuje nazwę bezwzględnej ścieżki dostępu dla bieżącego katalogu roboczego do tablicy wskazywanej przez *buf*, która to tablica ma długość *size*. Jeśli nazwa bieżącej bezwzględnej ścieżki dostępu wymaga bufora dłuższego niż *size* elementów, to zwracane jest **NULL**.

- `int chroot( const char *path);`
- Funkcja zmienia katalog **root** dla aktualnego procesu. Aktualne ustawienie jest dziedziczone przez procesy potomne. Jedynie proces uprzywilejowany może wykonać tę funkcję (uid 0 i ustawiona własność **CAP_SYS_CHROOT**)

Skanowanie katalogu 1/3

```
int scandir(const char *dir,
            struct dirent ***namelist,
            int(*filter)(const struct dirent *),
            int(*compar)(const struct dirent **,
                         const struct dirent **));
```

- Funkcja **scandir** skanuje katalog d*ir*, wywołując funkcję *filter*() dla każdego wpisu w katalogu. Wpisy, dla których funkcja ***filter*()** zwrócic wartość różną od zera są sortowane przy użyciu funkcji porównującej ***compar***() i umieszczane w tablicy *namelist* (alokowanej automatycznie przez funkcję scandir).
- Jeżeli *filter* jest ustawiony na NULL to pobierane są wszystkie wpisy.

Skanowanie katalogu 2/3

- `int alphasort(const void *a, const void *b);`
- `int versionsort(const void *a, const void b);`

- Funkcje `alphasort` and `versionsort` mogą być użyte jako gotowe funkce porównujące w *compar*().
- Własna funkcja porównująca musi zwracać liczbę całkowitą, która jest mniejsza, równa, lub większa od zera. Oznacza to wtedy, odpowiednio, że pierwszy argument jest mniejszy, równy, lub większy od drugiego.

Skanowanie katalogu 3/3

dir-scan.c

```
/* wyświetl zawartość aktualnego katalogu w odwróconej kolejności */
#include <dirent.h>

main()
{
    struct dirent **namelist;
    int n;
    n = scandir(".", &namelist, 0, alphasort);
    if (n < 0)
        perror("scandir");
    else {
        while(n--) {
            printf("%s\n", namelist[n]->d_name);
            free(namelist[n]);
        }
        free(namelist);
    }
}
```

Przechodzenie drzewa katalogów **ftw** 1/2

```
int ftw( const char *dirpath,
         int (*fn) ( const char *fpath,
                     const struct stat *sb,
                     int typeflag),
         int nopenfd);
```

- funkcja przechodzi przez drzewo katalogu ***dirpath***, wywołując funkcję ***fn*()** dla każdego znalezionego wpisu.
- argument ***nopenfd*** określa maksymalną liczbę jednocześnie otwartych katalogów.

- funkcja użytkownika ***fn()*** otrzymuje dla każdego wpisu inne wartości argumentów:
  - ***fpath*** – ścieżka względna do aktualnego wpisu (względem katalogu ***dirpath***)
  - ***sb*** – informacje o wpisie (w formacie identycznym jak zwracanym przez funkcję **stat**)
  - ***typeflag*** **–** typ wpisu (**FTW_F** – plik, **FTW_D** – katalog, **FTW_DNR** - katalog bez możliwości odczytu, **FTW_SL** - łącze symboliczne, FTW_NS - dla tego obiektu nie można wypełnić **stat** (nie jest łączem symbolicznym))

Przechodzenie drzewa katalogów **ftw** 2/2

ftw-test.c

```
#include <ftw.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static int display_info(const char *fpath, const struct stat *sb, int tflag, struct FTW *ftwbuf)
{
      printf("%-3s %2d %7lld   %-40s %d %s\n", tflag == FTW_D) ?   "d"   :
            (tflag == FTW_DNR) ? "dnr" : (tflag == FTW_DP) ?  "dp"  : (tflag == FTW_F) ?   "f" :
            (tflag == FTW_DP) ?  "dp"  : (tflag == FTW_SL) ?  "sl" : (tflag == FTW_SLN) ? "sln" : "?",
            ftwbuf->level, (long long) sb->st_size, fpath, ftwbuf->base, fpath + ftwbuf->base);
      return 0;
}

int main(int argc, char *argv[])
{
      int flags = 0;
      if (argc > 2 && strchr(argv[2], 'd') != NULL)
            flags |= FTW_DEPTH;
      if (argc > 2 && strchr(argv[2], 'p') != NULL)
      flags |= FTW_PHYS;
      nftw((argc < 2) ? "." : argv[1], display_info, 20, flags);
      exit(EXIT_SUCCESS);
}
```

Dodatkowy argument f-cji **fn()** dla rozszerzonej wersji **nftw**

Rozszerzona wersja f-cji **ftw**

Potoki

- Pliki
- Katalogi
- **Potoki**
  - Nienazwane
  - Nazwane
- Odwzorowanie plików w pamięci

Potoki

- Potoki **nienazwane** odwołują się bezpośrednio do i-węzłów na dysku, nie mają nazw w systemie plików – stąd mogą służyć jedynie do komunikacji między **bezpośrednio spokrewnionymi procesami !**

- Potoki nazwane (kolejki FIFO) są plikami specjalnymi, posiadającymi nazwy w systemie plików.

Potoki nienazwane

- Pliki
- Katalogi
- **Potoki**
  - **Nienazwane**
  - Nazwane
- Odwzorowanie plików w pamięci

Funkcja **pipe**

- **int pipe(int *filedes*[2]);**

- funkcja tworzy parę sprzężonych deskryptorów pliku, wskazujących na i-węzeł potoku i umieszcza je w tablicy wskazywanej przez ***filedes***.
- ***filedes[0]*** jest dla odczytu, a ***filedes[1]*** dla zapisu.
- Potok traktuje dane jak kolejkę **FIFO** (*ang. First In, First Out)* i ma ograniczoną pojemność, tj. w potoku może znajdować się tylko pewna ilość bajtów.

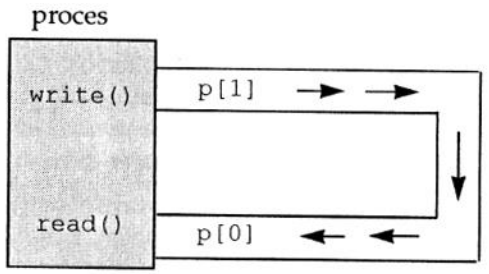
- Odczyt z potoku (funkcja **read**) powoduje usunięcie odczytanych danych.
- Jeżeli w potoku jest mniej danych niż chcemy odczytać, to odczytane i usunięte zostaną dane dostępne i nastąpi powrót z funkcji **read** (która zwróci ilość rzeczywiście odczytanych bajtów).
- Próba odczytu z pustego potoku powoduje zawieszenie wykonania funkcji **read** do momentu pojawienia się w potoku jakichkolwiek danych.

- Jeżeli zapisujemy do potoku, w którym jest dostateczna ilość miejsca, to dane są umieszczane w potoku, a funkcji **write** zwraca ile bajtów zostało zapisanych.
- Jeżeli wykonujemy zapis, który przepełnia potok, wykonanie procesu zostaje zawieszone do momentu gdy inny proces nie zrobi miejsca, odczytując dane z potoku.

Użycie `pipe` 1/3

pipe-1.c

```
...
char inbuf[ 6];
int p[2];
int retVal;
if( pipe( p) != -1)
{
    write( p[ 1], "abcd\0efgh\0", 10);

    retVal = read( p[ 0], inbuf, 5);
    printf("%d %s", retVal, inbuf);
    retVal = read( p[ 0], inbuf, 5);
    printf("%d %s", retVal, inbuf);
}
...
```

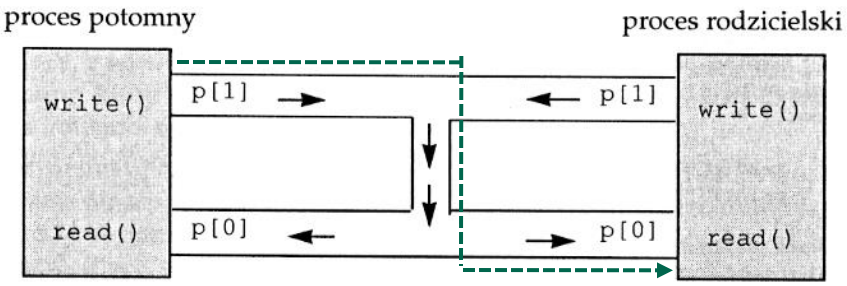



```
5 abcd
5 efgh
```

Użycie `pipe` 2/3

pipe-2.c

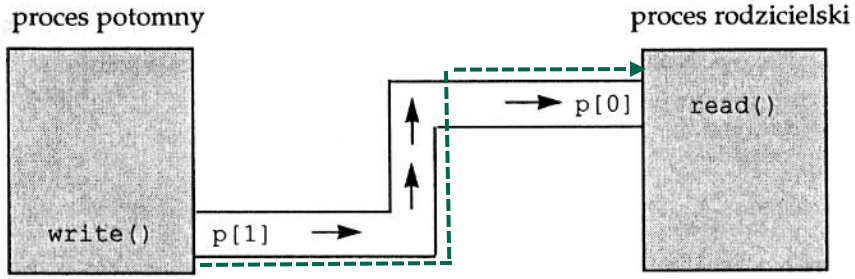
```
...
char inbuf[ 6];
int p[2];
int retVal;
if( pipe( p) != -1)
{
    if( fork() == 0)
        write( p[ 1], "abcd\0efgh\0", 10);
    else
    {
        retVal = read( p[ 0], inbuf, 5);
        printf("%d %s", retVal, inbuf);
        wait( NULL);
    }
    exit( 0);
}
...
```




```
5 abcd
```

Użycie `pipe` 3/3

pipe-3.c




```
...
if( pipe( p) != -1)
{
    if( fork() == 0)
    {
        close( p[0]);
        write( p[ 1], "abcd\0efgh\0", 10);
    }
    else
    {
        close( p[1]);
        retVal = read( p[ 0], inbuf, 5);
        printf("%d %s", retVal, inbuf);
        wait( NULL);
    }
    exit( 0);
}
...
```

```
5 abcd
```

Zamykanie potoków

- **Zamknięcie deskryptora pliku do zapisu**
  - jeżeli istnieją inne procesy, które mają potok otwarty **do zapisu**, to nic się nie dzieje.
  - jeżeli nie ma więcej procesów z potokiem otwartym do zapisu, a potok jest pusty, każdy proces próbujący odczytać dane z potoku wraca bez danych. Procesy, które w uśpieniu czekały na odczyt, zostaną obudzone, a ich funkcje **`read`** zwrócą zero.

- **Zamknięcie deskryptora pliku do odczytu**
  - jeżeli istnieją inne procesy, mające potok otwarty **do odczytu**, to nic się nie dzieje.
  - jeśli takie procesy nie istnieją, to do wszystkich procesów czekających na zapis do potoku będzie przez jądro wysłany sygnał **SIGPIPE**. Jeśli sygnał nie zostanie przechwycony, proces zakończy się. Jeśli sygnał zostanie przechwycony, po jego obsłudze, write zwróci -1, a zmienna ***errno*** będzie zawierać **EPIPE**.

Nieblokujące odczyty i zapisy

- Przy użyciu funkcji **`fcntl`** można zmienić zachowanie funkcji odczytu i zapisu w przypadku pustego i przepełnionego potoku:

```
fcntl(filedes, F_SETFL, O_NONBLOCK);
```

- Jeżeli ***filedes*** jest dekryptorem potoku do zapisu, to następne wywołania funkcji **`write`** nie będą blokowane, nawet jeśli potok jest pełny. W zamian zwracają -1 i ustawiają ***errno*** na **EAGAIN**.

- Jeżeli ***filedes*** jest deskryptorem pustego potoku do odczytu, to następne wywołania funkcji **`read`** nie będą blokowane, zwrócą wartości jak w przypadku **`write`**.

Potoki nazwane

- Pliki
- Katalogi
- **Potoki**
  - Nienazwane
  - **Nazwane**
- Odwzorowanie plików w pamięci

Funkcja `mkfifo`

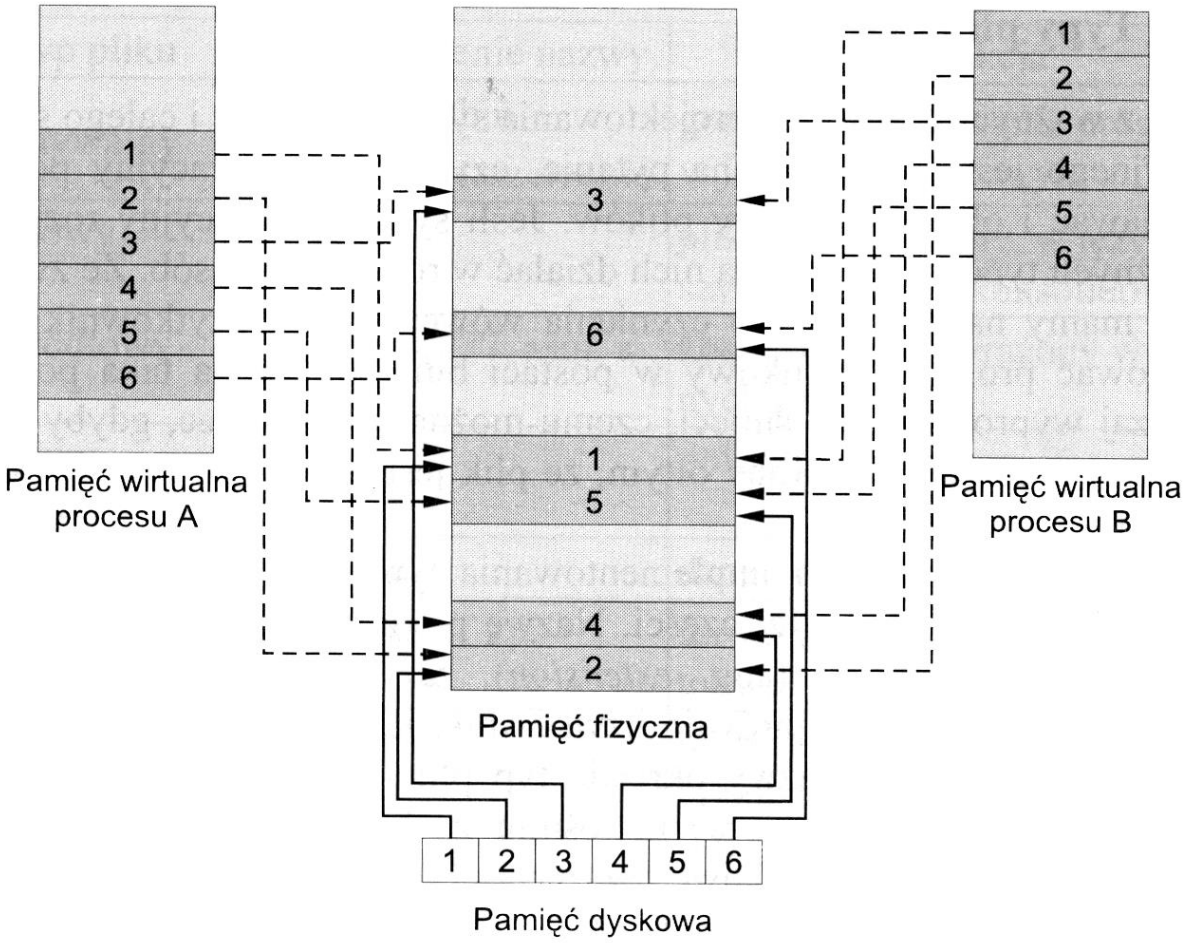
- **`int mkfifo( const char *pathname, mode_t mode);`**

- Funkcja tworzy specjalny plik FIFO o nazwie *pathname*.
- Argument ***mode*** określa prawa dostępu dla nowo tworzonego potoku (jest on modyfikowany podobnie jak w przypadku funkcji **`creat`** w oparciu o wartość **umask** procesu: ***mode*** **& ~umask**.
- Po utworzeniu kolejka FIFO może być obsługiwana za pomocą funkcji typowych dla plików (**`open`**, **`read`**, **`write`**, **`close`**)
- Domyślnie kolejka otwierana jest w trybie blokującym:
  - zawieszenie funkcji **`open`** przy otwarciu do **odczytu**, jeżeli inny proces nie otworzył potoku do **zapisu**;
  - zawieszenie funkcji **`open`** przy otwarciu do **zapisu**, jeżeli inny proces nie otworzył potoku do **odczytu**;
  - zawieszenie funkcji **`read`** przy odczycie z pustego potoku;
  - zawieszenie funkcji **`write`** przy zapisie do przepełnionego potoku;
- Użycie flagi **O_NONBLOCK** przy otwarciu kolejki powoduje, ze będzie ona działał w trybie nieblokującym:
  - próba odczytu z pustego potoku, którego nie otworzył do zapisu żaden proces zakończy się zwróceniem przez **read** wartości **0**,
  - Próba odczytu z pustego potoku, do którego w trybie zapisu podłączony jest inny proces, spowoduje, że **read** zwróci **-1** oraz błąd **EAGAIN**.

Odwzorowanie plików w pamięci

- Pliki
- Katalogi
- Potoki
  - Nienazwane
  - Nazwane
- **Odwzorowanie plików w pamięci**

Odwzorowanie plików w pamięci



UNIX: f-cje *mmap* i *munmap* 1/2

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

void * mmap(void *address, size_t length, int protection,
            int flags, int filedes, off_t offset);

int munmap(void *address, size_t length);
```

PROT_READ	Pamięć może być odczytywana.
PROT_WRITE	Pamięć może być zapisywana.
PROT_EXEC	Pamięć może być wykonywana.
PROT_NONE	Pamięć nie jest dostępna.

MAP_SHARED	Sprawia, że dowolne zmiany regionu są widziane przez inne procesy odwzorowujące plik, a modyfikacje są zapisywane do faktycznego pliku.
MAP_PRIVATE	Modyfikacje regionu nie są widziane przez inne procesy i nie są zapisywane do faktycznego pliku.

UNIX: f-cje *mmap* i *munmap* 2/2

```
/* odwzoruj w pamięci plik wejściowy i wyjściowy */
if( (source = mmap(0, filesize, PROT_READ, MAP_SHARED, input,
                   0)) == (void *)-1)
{
   fprintf(stderr, "Error mapping first file\n");
   exit(1);
}
if( (target = mmap(0, filesize, PROT_WRITE, MAP_SHARED, output,
                   0)) == (void *)-1)
{
   fprintf(stderr, "Error mapping second file\n");
   exit(2);
}

/* kopiuj */
memcpy(target, source, filesize);

/* zakończ odwzorowanie obu plików */
munmap(source, filesize);
munmap(target, filesize);
```