

Obsługa plików

- Dwa poziomy obsługi plików:
 - standardowa biblioteka we-we
 - niskopoziomowa biblioteka we-wy
- Standardowa biblioteka we-we
 - wchodzi w skład języka C, określona jest w standardzie ANSI C, dostępna jest na różnych platformach, umożliwia wprowadzanie lub wyprowadzanie danych do/z programu niezależne od systemu operacyjnego
 - otwarty plik reprezentowany jest za pomocą wskaźnika do struktury typu FILE. Z otwartym plikiem jest związany bufor nazywany *strumieniem* (ang. *stream*).
 - wejście-wyjście jest buforowane
 - we-wy podlega konwersji i formatowaniu
- Niskopoziomowa biblioteka we-we
 - jest budowana dla konkretnego systemu operacyjnego
 - otwarty plik jest reprezentowany za pomocą liczby całkowitej nazywanej deskryptorem
 - wejście-wyjście nie jest buforowane
 - dane pliku są reprezentowane jako ciąg bajtów
- Plik jest identyfikowany za pomocą nazwy. Nazwy plików mogą być dowolne i zawierać do 255 znaków. W niektórych systemach plików obowiązują nazwy do 14 znaków.
- Pliki zorganizowane są w hierarchiczną strukturę nazywaną *drzewem katalogowym* (ang. *directory tree*). Na szczycie drzewa znajduje się katalog nazywany katalogiem głównym (ang. *root directory*), oznaczany symbolem ukośnika /. Nazwa pliku musi być unikatowa tylko w obrębie katalogu.
- Pełna nazwa pliku odzwierciedla strukturę drzewa i określana jest jako *bezwzględna nazwa ścieżkowa* (ang. *absolute path name*). Podaje ona sekwencję katalogów prowadzących do pliku czyli określa bezwzględne położenie pliku w drzewie katalogowym.. Zawsze zaczyna się od symbolu /. Przykład:

```
    /usr/bin/ls  
    /home/nowak/bin/program
```
- Każdy proces (urochomiony program):
 - ma przypisany katalog główny. Jest on dziedziczony z procesu macierzystego (np.z shella),
 - ma przypisany *bieżący katalog roboczy* (ang. *current working directory*). Nazwa tego katalogu jest dziedziczona z procesu macierzystego (np. z shella),
 - dziedziczy otwarte w procesie macierzystym pliki (np. w shellu),
- Katalog bieżący jest wykorzystywany do tworzenia *nazw względnych* (ang. *relative pathname*). Jest to każda nazwa, która nie zaczyna się znakiem /. Podaje ona położenie pliku względem bieżącego katalogu roboczego. Przykład:

```
    bin/program
```

Standardowa biblioteka we-wy

Otwieranie i zamykanie pliku

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *type);
int fclose(FILE *stream);
```

- Funkcja `fopen` otwiera plik. Jej argumenty to:
filename - nazwa ścieżkowa otwieranego pliku
type - typ strumienia reprezentującego plik; może to być jedna z wartości:

r	otwórz plik tylko do czytania; plik musi istnieć
w	otwórz plik tylko do pisania; jeśli plik nie istnieje, należy go utworzyć; jeśli plik istnieje, jest skracany do zerowej wielkości
a	otwórz plik tylko do pisania (dopisywanie); jeśli plik nie istnieje, należy go utworzyć; jeśli plik istnieje, dane dopisywane są na koniec
r+	otwórz plik do czytania i pisania; plik musi istnieć
w+	otwórz plik do czytania i pisania; jeśli plik nie istnieje, należy go utworzyć; jeśli plik istnieje, jest skracany do zerowej wielkości
a+	otwórz plik do czytania i pisania; jeśli plik nie istnieje, należy go utworzyć; jeśli plik istnieje, dane dopisywane są na koniec

Jeśli operacja zakończy się powodzeniem, funkcja `fopen` zwraca wskaźnik do otwartego strumienia. W przeciwnym wypadku zwraca `NULL` i kod błędu umieszcza w `errno`.

- Funkcja `fclose` zamyka plik. Powoduje wyczyszczenie bufora `write` poprzez zapisanie jego zawartości na dysku i zwalnia zasoby związane z obsługą pliku przez jądro systemu. Jeśli operacja zakończy się powodzeniem, funkcja `fclose` zwraca zero. W przeciwnym wypadku zwraca stałą `EOF` i kod błędu umieszcza w `errno`.

Stałe

- Korzystając z biblioteki we-wy możemy posługiwać się następującymi stałymi:
 - zdefiniowane w `stdio.h`
 - `EOF` – oznacza koniec pliku, wartość zwracana przez funkcje po napotkaniu końca pliku
 - `BUFSIZ` – domyślna wielkość bufora dla operacji we-wy
 - `FILENAME_MAX` – liczba bajtów potrzebnych do przechowywania najdłuższej ścieżki dozwolonej przez implementację
- Proces uruchamiany przez shell automatycznie otwiera trzy strumienie: standardowe wejście, standardowe wyjście i standardowe wyjście diagnostyczne. Otrzymują one odpowiednio nazwy: `stdin`, `stdout`, `stderr`.

Czytanie i pisanie po znaku

```
#include <stdio.h>
int fgetc(FILE *stream);
int getc(FILE stream); /* zazwyczaj makro */
int getchar(void);

int fputc(int c, FILE *stream);
int putc(int c, FILE stream); / zazwyczaj makro */
int putchar(int c);

int ungetc(int c; FILE *stream);
```

- Funkcje `fgetc` i `getc` zwracają następny znak (bajt) z pliku określonego za pomocą wskaźnika `stream`. Jeśli nie ma znaku do czytania (osiągnięto koniec pliku) lub wystąpił błąd, zwracana jest stała `EOF`.
- Funkcje `fputc` i `putc` dokonują konwersji znaku `c` do `unsigned char` i umieszczają w `stream`. Jeśli operacja się powiedzie, zwracana jest wartość `c`, w przeciwnym wypadku zwracana jest stała `EOF`.
- Funkcja `ungetc` zwraca znak `c` do strumienia `stream`.

Czytanie i pisanie wierszami

```
#include <stdio.h>
char *fgets(char *s, int n, FILE stream);
char *gets(char *s);

int fputs(const char* s, FILE *stream);
int puts(const char *s);
```

- Funkcja `gets` czyta znaki ze `stdin` do znaku nowego wiersza (lub końca pliku) i umieszcza je w `s`. Znak nowego wiersza jest pomijany (wczytywany i zastępowany znakiem końca napisu). Funkcja umieszcza na końcu wczytanego ciągu znaków znak końca napisu `'\0'`. Zwraca `s`, jeśli operacja wczytywania zakończyła się powodzeniem, lub `NULL` gdy napotkany został koniec pliku przed rozpoczęciem czytania.
- Funkcja `fgets` czyta znaki ze `stream` do:
 - znaku nowego wiersza, jest on pamiętany,
 - przeczytania `n-1` znaków
 - lub napotkania znaku końca plikui umieszcza je w `s`.
- Funkcja `puts` zapisuje napis wskazany przez `s` uzupełniony znakiem nowego wiersza do standardowego wyjścia. Jeśli się powiedzie, zwraca ilość wyprowadzonych znaków. W przypadku błędu zwraca `EOF`.
- Funkcja `fputs` zapisuje napis wskazany przez `s` do strumienia `stream` (nie dołącza znaku nowego wiersza). Jeśli się powiedzie, zwraca ilość wyprowadzonych znaków. W przypadku błędu zwraca `EOF`.

Czytanie i pisanie blokami

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream);
```

- Funkcja `fread` czyta *nitems* danych, każda o rozmiarze *size*, ze strumienia *stream* i umieszcza je w tablicy wskazanej przez *ptr*. Zwraca liczbę przeczytanych danych (nie bajtów), 0 - gdy nic nie było przeczytane lub *EOF* gdy napotkany został koniec pliku przed rozpoczęciem czytania.
- Funkcja `fwrite` przesyła *nitems* danych, każda o rozmiarze *size*, ze strumienia *stream* i umieszcza je w tablicy wskazanej przez *ptr*. Zwraca liczbę przesłanych danych (nie bajtów), lub *EOF* gdy wystąpi błąd.

Sprawdzanie statusu strumienia we-wy

```
#include <stdio.h>
int ferror(FILE *stream);
int feof(FILE *stream);
void clearerr(FILE *stream);
```

- Funkcja `ferror` zwraca wartość $\neq 0$, jeśli wystąpi błąd operacji we-wy. W przeciwnym wypadku zwraca 0.
- Funkcja `feof` zwraca wartość $\neq 0$, jeśli podczas czytania ze strumienia napotkany został koniec pliku. W przeciwnym wypadku zwraca 0.
- Funkcja `clearerr` czyści wskaźniki błędu i końca pliku dla strumienia *stream*.

Formatowane wejście-wyjście

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *s, const char *format, ...);

int scanf(const char* format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
```

Przemieszczanie się w pliku

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
void rewind(FILE *stream);
long ftell(FILE *stream);

int fsetpos(FILE *stream, const fpos_t *pos);
int fgetpos(FILE *stream, fpos_t *pos);
```

Buforowanie

- Zachowanie domyślne:
 - Pliki dyskowe są buforowane dużymi porcjami (np. 1024 bajty i więcej).
 - Strumień `stdout`, jeśli dotyczy terminala jest buforowany wierszami; w przeciwnym wypadku jest buforowany tak jak plik.
 - Strumień `stderr` nie jest buforowany.
 - Jeśli `stdin` dotyczy terminala, strumień `stdout` jest automatycznie czyszczony (ang. *flush*) zawsze wtedy, kiedy wykonywane jest czytanie ze `stdin`.
 - Wywołanie funkcji `fseek` lub `rewind` czyści wszystkie bufory wyjścia, które zawierają dane.

- Funkcje obsługi buforowania:

```
#include <stdio.h>
int fflush(FILE *stream);
void setbuf(FILE *stream, char *buf);
void setvbuf(FILE *stream, char *buf, int type, size_t size);
```

- Jeśli `stream` jest otwarty do zapisu, funkcja `fflush` powoduje zapisanie danych czekających w buforze do pliku. Jeśli `stream` jest otwarty do odczytu, funkcja `fflush` powoduje usunięcie z bufora wszystkich nie przeczytanych danych. Jeśli `stream` ma wartość `NULL`, funkcja `fflush` powoduje zapisanie danych do plików ze wszystkich strumieni otwartych do zapisu.
- Funkcja `setbuf` pozwala przydzielić własny bufor (tablica wskazywana przez `buf`). Rozmiar tej tablicy będzie równy stałej `BUFSIZ`. Jeśli wartością `buf` jest `NULL`, strumień nie będzie buforowany.
- Funkcja `setvbuf` pozwala określić własny sposób buforowania. Argument `type` może przyjmować wartości:

<code>_IOFBF</code>	Pełne buforowanie we-wy
<code>_IOLBF</code>	Buforowanie wierszami. Bufor jest czyszczony wtedy, kiedy wpisany zostanie do niego znak nowego wiersza, bufor jest pełny lub żądane jest wprowadzenie nowych danych.
<code>IONBF</code>	Brak buforowania

Buforem jest tablica `buf` o rozmiarze `size`.

Niskopoziomowa biblioteka we-wy

- Biblioteka umożliwiająca wykonywanie operacji wejścia-wyjścia na poziomie funkcji systemowych.
- Otwarty plik jest reprezentowany za pomocą *deskryptora* pliku (ang. *file descriptor*). Deskryptor pliku to nieujemna liczba całkowita, której wartość jest określana przez system.

Podstawowy interfejs we-wy

```
#include <sys/types.h>
#include <sys/stat.h>          /* dla mode_t - praw dostępu */
#include <fcntl.h>            /* dla flags - trybów dostępu */
int open(const char *pathname, int flags, [mode_t mode]);

#include <unistd.h>           /* dla ssize_t */
ssize_t read(int filedes, void *buffer, size_t n);
ssize_t write(int filedes, const void *buffer, size_t n);

#include <unistd.h>
int close(int filedes);
```

Stałe:

- `BUFSIZ` – „optymalny” rozmiar bloku dla operacji we-wy (definiowany w `stdio.h`)
- Przykład: uproszona wersja polecenia `cat`

```
int cat( char *file)
{
    int fd;
    ssize_t rcount, wcount;
    char buffer[BUFSIZ];
    int errors = 0;

    if (strcmp(file, "-") == 0)
        fd = 0;
    else if ((fd = open(file, O_RDONLY)) < 0) {
        perror(file);
        return 1;
    }

    while ((rcount = read(fd, buffer, sizeof buffer)) > 0) {
        wcount = write(1, buffer, rcount);
        if (wcount != rcount) {
            perror(file);
            errors++;
            break;
        }
    }

    if (rcount < 0) {
        perror(file);
        errors++;
    }

    if (fd != 0) {
        if (close(fd) < 0) {
            perror(file);
            errors++;
        }
    }

    return (errors != 0);
}
```

Otwieranie i zamykanie pliku

```
#include <sys/types.h>
#include <sys/stat.h>          /* dla mode_t - praw dostępu */
#include <fcntl.h>            /* dla flags - trybów dostępu */
#include <unistd.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` otwiera plik. Jej argumenty to nazwa pliku (*pathname*) - bezwzględna lub względna, określenie opcji dostępu i przebiegu operacji na pliku (*flags*), oraz praw dostępu do pliku (*mode*) - jeśli plik jest jednocześnie tworzony. Jeśli funkcja wykona się poprawnie, zwraca *deskryptor pliku*. W przeciwnym wypadku zwraca -1 i ustawia zmienną `errno`.
- Funkcja `open` jest używana zarówno do otwarcia istniejącego pliku, jak i do utworzenia nowego pliku.
- Argument prawa dostępu do pliku (*mode*) używany jest tylko wtedy, kiedy tworzony jest plik.
- Funkcja `creat` otwiera plik *do zapisu*. Jej argumenty to nazwa pliku (*pathname*) - bezwzględna lub względna oraz prawa dostępu do pliku (*mode*).
- Funkcja `close` zamyka plik określony deskryptorem. (*filedes*). Jeśli w programie nie umieszcza się tej funkcji, to pliki są automatycznie zamykane wtedy, kiedy proces kończy się wykonywać.

Deskryptor pliku

- Deskryptor pliku jest to liczba całkowita reprezentująca w procesie otwarty plik.
- Deskryptory mają wartości z zakresu 0 do limitu określonego w systemie. Wartość tego limitu można uzyskać za pomocą funkcji `getdtablesize()` lub polecenia `ulimit -u`.

- Przykład:

```
#include <stdio.h>           /* dla printf() */
#include <unistd.h>         /* dla getdtablesize() */
#include <stdlib.h>
int main(int argc, char **argv)
{
    printf("max fds: %d\n", getdtablesize());
    exit(0);
}
```

- Funkcja `open` zwraca deskryptor pliku, który identyfikuje otwarty plik i jest używany przez inne funkcje. Funkcja wybiera jako deskryptor pliku *najmniejszą* nieujemną liczbę całkowitą, nie używaną jeszcze jako deskryptor pliku w procesie dokonującym wywołania.
- Zazwyczaj każdy program uruchamiany jest z otwartymi trzema deskryptorami: 0 (standardowe wejście), 1 (standardowe wyjście) i 2 (standardowe wyjście błędów). Standard Posix definiuje w `<unistd.h>` stałe, którymi można posługiwać się zamiast deskryptorów liczbowych:

Deskryptor	Nazwa	Opis
0	STDIN_FILENO	standardowy strumień wejściowy
1	STDOUT_FILENO	standardowy strumień wyjściowy
2	STDERR_FILENO	standardowy strumień diagnostyczny

Deskryptor i wskaźnik do pliku

```
#include <stdio.h>

int fileno(FILE *fp);

FILE *fdopen(int fd, const char* type)
```

- Funkcja `fileno` zwraca deskryptor pliku odpowiadający wskaźnikowi do pliku `fp`.
- Funkcja `fdopen` zwraca wskaźnik do pliku związanego z deskryptorem `fd`. Argument `type` określa typ dostępu do pliku (patrz opis funkcji `fopen()`).

Opcje dostępu do pliku

- Tryb dostępu opisywany jest stałą zdefiniowaną w pliku `<fcntl.h>`. Podstawowe stałe to:

Nazwa symboliczna	Wartość	Opis
<code>O_RDONLY</code>	0	Otwórz plik tylko do czytania
<code>O_WRONLY</code>	1	Otwórz plik tylko do pisania
<code>O_RDWR</code>	2	Otwórz plik do czytania i zapisu

Tylko *jedna* z tych stałych może być użyta. Plik tylko wtedy zostanie otworzony, jeśli proces będzie miał odpowiednie uprawnienia.

Dodatkowo może być użyta jedna z następujących stałych:

<code>O_APPEND</code>	dołącz do końca pliku - wskaźnik pozycji przesuwany jest na koniec pliku zawsze wtedy, kiedy jądro chce zapisać, czyli przed wykonaniem <code>write</code>
<code>O_CREAT</code>	utwórz pliku, jeśli nie istnieje (o ile proces posiada odpowiednie uprawnienia i istnieje wymagany katalog); otwórz plik jeśli plik istnieje
<code>O_EXCL</code>	używana razem z <code>O_CREAT</code> : zwróć błąd, jeśli plik, który ma być utworzony już istnieje
<code>O_TRUNC</code>	jeśli plik istnieje i został pomyślnie otwarty w trybie do zapisu lub odczytu-zapisu, usuń jego zawartość
<code>O_NOCTTY</code>	jeśli nazwa pliku dotyczy terminala, nie przydzielaj tego terminala jako terminala sterującego tego procesu
<code>O_SYNC</code>	<code>write</code> czeka na zakończenie pełnej operacji zapisu
<code>O_DSYNC</code>	<code>write</code> czeka na zakończenie operacji zapisu danych na dysku
<code>O_RSYNC</code>	<code>read</code> czeka na zakończenie poprzedniej operacji zapisu, jeśli w pamięci cache znajdują się takie dane
<code>O_NONBLOCK</code>	otwórz w trybie nieblokującym (zastosowanie: na przykład potoki)

Prawa dostępu do pliku

- Wyróżniane są trzy typy użytkowników:
 - właściciel pliku
 - grupa pliku
 - pozostali użytkownicy, nie należący do poprzednich kategorii
- Dla każdej kategorii wyróżniane są trzy typy praw dostępu:
 - prawo odczytu z pliku
 - prawo zapisu do pliku
 - prawo wykonywania pliku
- Prawa dostępu do pliku opisywane są za pomocą stałych ósemkowych lub symbolicznych (<sys/stat.h>).

Stała symboliczna	Stała liczbowa	Opis
S_IRUSR	0400	Właściciel ma prawo do czytania
S_IWUSR	0200	Właściciel ma prawo do pisania
S_IXUSR	0100	Właściciel ma prawo do wykonywania
S_IRGRP	0040	Grupa pliku ma prawo do czytania
S_IWGRP	0020	Grupa pliku ma prawo do pisania
S_IXGRP	0010	Grupa pliku ma prawo do pisania
S_IROTH	0004	Pozostali użytkownicy mają prawo do czytania
S_IWOTH	0002	Pozostali użytkownicy mają prawo do pisania
S_IXOTH	0001	Pozostali użytkownicy mają prawo do wykonywania
S_IRWXU	0700	Właściciel ma prawo do czytania, pisania i wykonywania
S_IRWXG	0070	Grupa pliku ma prawo do czytania, pisania i wykonywania
S_IRWXO	0007	Pozostali użytkownicy mają prawo do czytania, pisania i wykonywania

- Przykład: trzy sposoby zapisu praw `rwX r-- ---`

```
mode_t prawa=0;
prawa = 0740;
prawa = S_IRUSR|S_IWUSR|S_IXUSR|S_IRGRP; /* 0740 */
prawa = S_IRWXU|S_IRGRP; /* 0740 */
```
- Przykłady:


```
fd=open("dane.txt",O_RDONLY);
fd=open("dane.txt",O_RDWR|O_CREAT, 0644);
fd=open("blokada",O_WRONLY|O_CREAT|O_EXCL, 0644);
fd=open("wyniki",O_WRONLY|O_CREAT|O_TRUNC, 0644);
```

Modyfikatory praw dostępu do pliku

- Podstawowe prawa dostępu mogą być modyfikowane.

Nazwa symboliczna	Stała	Opis
S_ISUID	04000	Bit set-user-id
S_ISGID	02000	Bit set-group-id
S_ISVTX	01000	Bit sticky

Funcje `open()` i `create()`

- Funkcja `creat` otwiera plik *do zapisu*. Jej argumenty to nazwa pliku (*pathname*) - bezwzględna lub względna oraz prawa dostępu do pliku (*mode*).
- Wywołanie:
`create(pathname, mode)`
jest równoważne wywołaniu funkcji `open` z następującymi parametrami:
`open(pathname, O_WRONLY|O_CREAT|O_TRUNC, mode);`

Przykłady

- Utwórz nowy plik z prawami 0644:

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

int main (int argc, char* argv[]) {
    char* nazwa_pliku = argv[1];
    /* prawa dostępu dla nowego pliku */
    mode_t tryb = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH;
    /* lub po prostu:
    mode_t tryb = 0644;
    */
    /* utwórz nowy plik i otwórz go do pisania */
    int fd = open (nazwa_pliku, O_WRONLY | O_EXCL | O_CREAT, tryb);
    if (fd == -1) {
        perror ("open");
        return 1;
    }
    return 0;
}
```

Maska tworzenia plików

- Każdy proces ma maskę tworzenia pliku (ang. *file creation mask*), która określa bity *wyłączone* podczas tworzenia nowego pliku. Maską jest dziedziczona z procesu macierzystego.
- Maską dotyczy tylko plików zwykłych i katalogów.
- Prawa dostępu do tworzonego pliku wyznaczane są zależnością:

```
prawa_obowiązujące = prawa_żądane & (~maska);
```

gdzie prawa żądane określone są na przykład w funkcji tworzącej plik.

- Odziedziczoną maskę można zmienić za pomocą funkcji `umask`:

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t newmask);
```

Argumentem funkcji jest nowa maska. Funkcja zwraca poprzednią maskę.

- Przykład: przypisanie nowej wartości maski

```
umask(S_IWGRP | S_IRWXO); /* równoważne poleceniu shella umask 027 */
```

- Przykład: przypisanie nowej wartości maski i zapamiętanie poprzedniej wartości

```
mode_t=stara_maska;
stara_maska=umask(S_IWGRP | S_IRWXO);
```

- Przykład: wyświetlenie aktualnie obowiązującej maski

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

mode_t zapytaj_o_maske(void) {
    mode_t stara_maska;
    umask(stara_maska = umask(0)); /* nie chcemy zmieniać maski */
    return stara_maska;
}

int main(int argc, char **argv) {
    printf("umask = %04o\n", zapytaj_o_maske());
    return 0;
}
```

Czytanie z pliku i zapisywanie do pliku

```
#include <unistd.h>
ssize_t read(int filedes, void *buffer, size_t n);
ssize_t write(int filedes, const void *buffer, size_t n);
```

- Funkcja `read` służy do wczytywania określonej liczby bajtów (argument `n`) do bufora (`buffer`) z otwartego uprzednio pliku (`filedes`). Funkcja zwraca liczbę rzeczywiście przeczytanych bajtów. Jeśli wystąpi błąd zwracana jest wartość `-1`. Wartość `0` oznacza koniec pliku.
- Zachowanie funkcji `read` związane jest z typem przetwarzanego pliku. W przypadku *pliku zwykłego*, zwrócona liczba rzeczywiście przeczytanych bajtów może być mniejsza wtedy, kiedy liczba bajtów w tym pliku jest mniejsza od `n`. Następne wywołanie funkcji `read` zwróci `0`.
- Funkcja `write` służy do przesyłania określonej liczby bajtów (argument `n`) z bufora (`buffer`) do otwartego uprzednio pliku (`filedes`). Funkcja zwraca liczbę przesłanych bajtów do wyjścia. Jeśli wystąpi błąd zwracana jest wartość `-1`. Jeśli liczba przesłanych bajtów w przypadku zwykłego pliku nie jest równa `n`, oznacza to, że wystąpił jakiś błąd.
- Z każdym deskryptorem pliku związany jest *wskaźnik odczytu-zapisu* (ang. *read-write pointer*). Rejestruje on położenie *następnego* bajtu w pliku, który powinien być odczytany (lub zapisany). Każde wykonanie funkcji `read` lub `write` powoduje zwiększenie wskaźnika odczytu-zapisu o odpowiednią liczbę bajtów.
- Jeśli plik jest otworzony w trybie `O_APPEND`, wskaźnik pozycji przesuwany jest na koniec pliku zawsze wtedy, kiedy jądro chce zapisać, czyli przed wykonaniem `write`.

- Przykład: program czyta plik po 512 znaków

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

#define ROZMIAR 512

int main() {
    char bufor[ROZMIAR];
    int fd;      /* deskryptor pliku */
    ssize_t n;   /* liczba przeczytanych bajtów przez read*/
    long ile=0;  /* sumaryczna liczba przeczytanych bajtów */
    if ((fd=open("dane.txt", O_RDONLY)) == -1)
    { /* obsługa błędu */ }
    while ((n=read(fd,bufor,ROZMIAR)) > 0)
        ile += n;
    printf("plik dane.txt zawiera %ld bajtów\n",ile);
    close(n);
    exit(0);
}
```

- Przykład: kopiowanie plików

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#define ROZMIAR 512
#define PRAWA 0644

int kopiujplik(const char *nazwaWe, const char *nazwaWy) {
    int plikwe, plikwy;
    ssize_t n;
    char bufor[ROZMIAR];
    if ((plikwe=open(nazwaWe, O_RDONLY)) == -1)
        return -1;
    if ((plikwy=open(nazwaWy, O_WRONLY|O_CREAT|O_TRUNC, PRAWA)) == -1)
    { close(plikwe);
      return -2;
    }
    while ((n=read(plikwe,bufor,ROZMIAR)) > 0) {
        if (write(plikwy,bufor,n)<n) {
            close(plikwe);
            close(plikwy);
            return(-3);
        }
    }
    close(plikwe);
    close(plikwy);
    if (n==-1) return (-4);
    else return 0;
}

int main() {
    kopiujplik("plik1.txt", "plik2.txt");
    return 0;
}
```

Informacja o pliku

- W Uniksie wyróżniane są następujące typy plików:
 - pliki zwykłe (ang. *regular*) - służą do przechowywania danych. Unix nie rozróżnia pliku tekstowego i binarnego. W systemie Uniksowym plik to ciąg bajtów. Dostęp do pliku zwykłego może być zarówno sekwencyjny (ang. *sequential*) jak i swobodny (ang. *random*);
 - katalogi - plik specjalny zawierający listę plików umieszczonych w katalogu (ang. *directory*).
 - łącza (ang. *pipes*) - dostarczają kanały komunikacji między procesami;
 - łącze nienazwane (ang. *unnamed pipes*) - umożliwia komunikację między procesami związanymi ze sobą; łącza te tworzone są w razie potrzeby i likwidowane wtedy, kiedy obydwa ich końce (do czytania i pisania) zostaną zlikwidowane, łącza te nie istnieją zatem w systemie plików
 - łącze nazwane (ang. *named pipes*) - ma przypisaną nazwę pliku, umożliwia komunikację dwóm nie związanym ze sobą procesom;
 - urządzenia (ang. *devices*) - blokowe (ang. *block devices*) i znakowe (ang. *character devices*) – służą do obsługi urządzeń
 - dowiązania symboliczne (ang. *symbolic links*) - specjalne pliki, które zawierają ścieżkę dostępu do innego pliku. W większości poleceń i funkcji system podąża za dowiązaniem - jeśli otworzony zostanie taki plik, system rozpoznaje, że jest to dowiązanie i otwiera plik, do którego się ono odnosi;
 - gniazda (ang. *sockets*) - dostarczają kanały komunikacji między procesami, również uruchomionymi na różnych komputerach.

Nazwa katalogu i pliku

```
#include <libgen.h>

char *dirname(char *path);
char *basename(char *path);
```

- Funkcje `dirname` i `basename` rozbijają zakończony znakiem null łańcuch nazwy ścieżki dostępu na składowe: katalog i nazwę pliku. Funkcja `dirname` zwraca łańcuch aż do ostatniego znaku '/', ale z jego wyłączeniem, a `basename` zwraca składową następującą po ostatnim '/'. Razem funkcje te zwracają pełną nazwę ścieżki dostępu.

- Przykład:

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

int main (int argc, char **argv)
{
    int n;
    char *ptr1, *ptr2;
    char *dir;
    char *base;

    for (n = 1; n < argc; n++) {
        if ((ptr1 = strdup (argv [n])) == NULL)
            { perror("strdup failed"); exit(1); }
        dir = dirname (ptr1);

        if ((ptr2 = strdup (argv [n])) == NULL)
            { perror("strdup failed"); exit(1); }
        base = basename (ptr2);

        printf ("%s:\n", argv [n]);
        printf ("  dirname = %s\n", dir);
        printf ("  basename = %s\n\n", base);

        free (ptr1);
        free (ptr2);
    }
    return (0);
}
```

Przykład użycia:

```
$/prog /usr/bin/zip
/usr/bin/zip:
  dirname = /usr/bin
  basename = zip

$ ./prog /usr/bin
/usr/bin:
  dirname = /usr
  basename = bin
```


Uzyskiwanie informacji o pliku

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

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

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

- Funkcja `stat` zwraca informację o pliku przechowywaną w i-węźle. Argument `pathname` określa nazwę pliku, `buf` - zmienną, do której funkcja wpisze pobraną z i-węzła informację. Jeśli nazwa jest dowiązaniem symbolicznym, zwracana jest informacja o pliku, na który wskazuje dowiązanie. Jeśli taki plik nie istnieje lub nie ma do niego dostępu, zwracany jest błąd.
- Funkcja `lstat` zwraca informację o pliku mającym nazwę `pathname`. Jest ona umieszczona w argumencie `buf`. Jeśli nazwa jest *dowiązaniem symbolicznym*, zwraca jest informacja o samym dowiązaniu (nie zaś o pliku na który wskazuje).
- Funkcja `fstat` używana jest do otwartego pliku.
- Struktura `stat` zawiera takie informacje jak:

```
struct stat {
    mode_t    st_mode;      /* typ pliku i jego prawa dostępu */
    ino_t     st_ino;       /* numer i-węzła */
    dev_t     st_dev;       /* numer urządzenia (system plików) */
    nlink_t   st_nlink;     /* liczba dowiązań twardych*/
    uid_t     st_uid;       /* UID właściciela pliku */
    gid_t     st_gid;       /* GID właściciela pliku */
    off_t     st_size;      /* rozmiar w bajtach, dla pliku zwykłego */
    time_t    st_atime;     /* czas ostatniego dostępu w sek*/
    time_t    st_mtime;     /* czas ostatniej modyfikacji w sek */
    time_t    st_ctime;     /* czas ostatniej zmiany statusu pliku w sek*/
};
```

Czasy są wyrażone w sekundach od początku epoki (1 stycznia 1970 roku).

• Przykład:

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

void show_stat_info(char *, struct stat *);

int main(int argc, char *argv[])
{
    struct stat info;
    if (argc>1)
    {
        if(stat(argv[1], &info) != -1 ){
            show_stat_info( argv[1], &info );
            return 0;
        }
        else
            perror(argv[1]);
    }
    return 1;
}

void show_stat_info(char *fname, struct stat *buf)
/* wyświetla dane pliku ze struktury stat */
{
    printf("typ i prawa: %o\n",  buf->st_mode);
    printf(" dowiazania: %d\n",  buf->st_nlink);
    printf(" wlasciciel: %d\n",  buf->st_uid);
    printf("      grupa: %d\n",  buf->st_gid);
    printf("      rozmiar: %ld\n", buf->st_size);
    printf(" czas modyf: %ld\n", buf->st_mtime);
    printf("      nazwa: %s\n",  fname );
}
```

Wynik działania programu dla przykładowego pliku:

```
typ i prawa: 100644
dowiazania: 1
wlasciciel: 1260
grupa: 101
rozmiar: 1208
czas modyf: 1067011428
nazwa: pl.c
```

Dla porównania informacja o pliku z polecenia `ls -l`:

```
-rw-r--r--  1 piotr  users      1208 paź 11 18:03 pl.c
```

Typ pliku

- Do sprawdzania typu pliku służą odpowiednie makra (`sys/stat.h`):

Makro	Maska symboliczna	Maska liczbowa	Typ pliku
	<code>S_IFMT</code>	0170000	Wybiera bity typu pliku
<code>S_ISREG()</code>	<code>S_IFREG</code>	0100000	Plik zwykły
<code>S_ISDIR()</code>	<code>S_IFDIR</code>	0040000	Katalog
<code>S_ISFIFO()</code>	<code>S_IFIFO</code>	0010000	Łącze (nazwane)
<code>S_ISBLK()</code>	<code>S_IFBLK</code>	0060000	Urządzenie blokowe
<code>S_ISCHR()</code>	<code>S_IFCHR</code>	0020000	Urządzenie znakowe
<code>S_ISLNK()</code>	<code>S_IFLNK</code>	0120000	Dowiązanie symboliczne
<code>S_ISSOCK()</code>	<code>S_IFSOCK</code>	0140000	Gniazdo

- Trzy sposoby sprawdzenia typu pliku:

```
if (S_ISDIR(buf.st_mode)) printf("to jest katalog");
if ( (buf.st_mode & 0170000) == 0040000 printf("to jest katalog");
if ( (buf.st_mode & S_IFMT) == S_IFDIR printf("to jest katalog");
```

- Przykład:

```
int sprawdz(char *nazwa_pliku) {
    struct stat buf;
    char *ptr;

    lstat(nazwa_pliku, &buf);
    if (S_ISREG(buf.st_mode)) ptr = "zwykly";
    else if (S_ISDIR(buf.st_mode)) ptr = "katalog";
    else if (S_ISCHR(buf.st_mode)) ptr = "specjalny znakowy";
    else if (S_ISBLK(buf.st_mode)) ptr = "specjalny blokowy";
    else if (S_ISFIFO(buf.st_mode)) ptr = "fifo";
#ifdef S_ISLNK
    else if (S_ISLNK(buf.st_mode)) ptr = "dowiazanie";
#endif
#ifdef S_ISSOCK
    else if (S_ISSOCK(buf.st_mode)) ptr = "gniazdo";
#endif
    else ptr = "*** nieznany typ pliku ***";
    printf("%s\n", ptr);
    return 0;
}
```

Prawa dostępu

• Przykład 1:

```
#include <stdio.h>
#include <sys/stat.h>

int main (int argc, char* argv[]) {
    const char* const nazwa_pliku = argv[1];
    struct stat buf;
    stat (nazwa_pliku, &buf);
    if (buf.st_mode & S_IWUSR)
        printf ("Wlasciciel ma prawo do zapisu `%s'.\n", nazwa_pliku);
    return 0;
}
```

• Przykład 2:

```
void mode_to_letters( int mode, char str[] )
{
    strcpy( str, "-----" );          /* default=no perms */

    if ( S_ISDIR(mode) ) str[0] = 'd';  /* directory?      */
    if ( S_ISCHR(mode) ) str[0] = 'c';  /* char devices    */
    if ( S_ISBLK(mode) ) str[0] = 'b';  /* block device    */

    if ( mode & S_IRUSR ) str[1] = 'r'; /* 3 bits for user */
    if ( mode & S_IWUSR ) str[2] = 'w';
    if ( mode & S_IXUSR ) str[3] = 'x';

    if ( mode & S_IRGRP ) str[4] = 'r'; /* 3 bits for group */
    if ( mode & S_IWGRP ) str[5] = 'w';
    if ( mode & S_IXGRP ) str[6] = 'x';

    if ( mode & S_IROTH ) str[7] = 'r'; /* 3 bits for other */
    if ( mode & S_IWOTH ) str[8] = 'w';
    if ( mode & S_IXOTH ) str[9] = 'x';
}
```

• Przykład 3:

```
const int N_BITS = 3;
unsigned int mask = 0700;
struct stat buff;
static char *perm[] = {"---", "--x", "-w-", "-wx",
                      "r--", "r-x", "rw-", "rwx"};

int i;
for (int i=3; i; --i) {
    printf("%s\n", perm[(buff.st_mode & mask) >> (i-1)*N_BITS]);
    mask >>= N_BITS;
}
```

Różne operacje na plikach

Poruszanie się po pliku

- Z każdym otwartym plikiem związany jest *wskaźnik odczytu-zapisu* (ang. *read-write pointer*). Rejestruje on położenie *następnego* bajtu w pliku, który powinien być odczytany (lub zapisany).
- Unix pozwala na sekwencyjny i swobodny dostęp do pliku. Domyślnie wybierany jest dostęp sekwencyjny. Oznacza to, że wskaźnik jest zwiększany o przeczytaną liczbę bajtów. Funkcja `lseek` pozwala na dostęp swobodny dzięki temu, że pozwala zmienić pozycję wskaźnika na dowolną inną.

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int filedes, off_t offset, int start_flag);
```

- Argument *filedes* określa deskryptor pliku, którego operacja dotyczy. Argumenty *offset* i *start_flag* określają nową pozycję. Pozycja ta jest wyznaczana następująco: *offset* podaje liczbę bajtów, które trzeba dodać do pozycji początkowej określonej za pomocą *start_flag*. Argument *offset* może być dodatni (przesunięcie w kierunku końca pliku) lub ujemny (przesunięcie w kierunku początku pliku). Argument *start_flag* może przyjmować następujące wartości:

SEEK_SET lub 0	<i>offset</i> jest mierzony od początku pliku
SEEK_CUR lub 1	<i>offset</i> jest mierzony od bieżącej pozycji wskaźnika
SEEK_END lub 2	<i>offset</i> jest mierzony od końca pliku

Funkcja zwraca nową pozycję w pliku. W przypadku błędu zwraca -1.

- Argument *offset* w momencie otworzenia pliku ma wartość domyślną 0, o ile nie określono opcji `O_APPEND`.
- Przykład: odczytanie aktualnej pozycji w pliku o deskrytorze *fd*

```
off_t pozycja = lseek(fd, 0, SEEK_CUR);
```

- Przykład: przesunięcie wskaźnika w pliku o 1024 bajty od początku pliku

```
off_t nowa_pozycja;
nowa_pozycja= lseek(fd, 1024, SEEK_SET);
```

Skracanie pliku

```
#include <unistd.h>
int truncate(const char *pathname, off_t length)
int ftruncate(int filedes, off_t length)
```

- Funkcja `truncate` pozwala usunąć część wskazanego pliku, bez konieczności otwierania go. Parametr `length` wskazuje do jakiego rozmiaru w bajtach należy plik zmniejszyć.
- Funkcja `ftruncate` działa podobnie, ale na otwartym pliku wskazanym za pomocą deskryptora.

Usuwanie pliku

```
#include <unistd.h>
int unlink(const char* pathname);           // POSIX

#include <stdio.h>                            // ISO C
int remove(const char* pathname)
```

- Usunięcie pliku oznacza usunięcie pliku z katalogu i zmniejszenie licznika odwołań do pliku (przechowywanego w i-węźle). Miejsce na dysku zostaje zwolnione dopiero wtedy, kiedy zostanie usunięte ostatnie dowiązanie do tego pliku – licznik odwołań przyjmie wartość 0.
- Funkcja `unlink` usuwa pozycję we wskazanym katalogu odpowiadającą plikowi. Potrzebne jest do tego prawo pisania do katalogu. Jeśli operacja się powiedzie zwraca 0, w przeciwnym wypadku -1.
- Wywołanie `unlink` nie oznacza to, że natychmiast zostanie usunięty plik. Plik jest usuwany dopiero wtedy, kiedy licznik odwołań do niego przyjmie wartość 0 i plik nie będzie używany przez żaden proces.
- Przykład:

```
fd=open("/tmp/plik", O_CREAT|O_EXCL|O_TRUNC|O_RDWR, 0000);
unlink("/tmp/plik");
... nadal można korzystać z pliku ...
close(fd);           // zamknij plik, zwolnij miejsce na dysku
```
- Funkcja `remove` usuwa plik lub pusty katalog, w zależności od tego czym jest jej argument. Zwraca 0 jeśli powiedzie się, -1 w przeciwnym wypadku. Jest to funkcja standardowej biblioteki `we-wy`.

Dowiązanie pliku i zmiana jego nazwy

```
#include <unistd.h>
int link(const char* existingpath, const char *newpath);

int rename(const char* oldname, const char *newname);
```

- Funkcja `link` tworzy dowiązanie do istniejącego pliku. Jest to dowiązanie twarde. Nowa nazwa (*newpath*) musi znajdować się w tym samym fizycznym systemie plików.
- Funkcja `rename` pozwala zmienić nazwę pliku na inną. Obydwie nazwy muszą znajdować się w tym samym fizycznym systemie plików. Jeśli istnieje plik o podanej nazwie, to przed wykonaniem zmiany nazwa ta zostanie usunięta.

Przykład

- Polecenie `mv` przesuwa plik do innego katalogu. Zakładamy, że jest ono wykonywane w tym samym systemie plików. Na przykład: `mv ./a.out ./bin/aplikacja`. Implementacja tego polecenia mogłaby wyglądać następująco, jeśli nie jest dostępna funkcja `rename`:

```
if (link("./a.out", "./bin/apilkacja") == -1) {
    fprintf(stderr, "%s: link(2)\n", strerror(errno));
    abort();
}
if (unlink("./a.out") == -1) {
    fprintf(stderr, "%s: unlink(2)\n", strerror(errno));
    abort();
}
```

Problem: *operacja nie jest atomowa*, nie jest wykonywana jako jedna nieprzerywalna operacja. Rozwiązanie: wprowadzono funkcję `rename`. Patrz opis tej funkcji: `man 2 rename`.

Zmiana atrybutów pliku

Sprawdzenie praw dostępu

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

int stat(const char *pathname, struct stat *buf);
int lstat(const char *pathname, struct stat *buf);
int fstat(int filedes, struct stat *buf);
```

- Opis funkcji stat, lstat, fstat - patrz Informacje o pliku

```
#include <unistd.h>
int access(const char *pathname, int amode);
```

- Funkcja access pozwala sprawdzić czy proces ma określone prawa dostępu do pliku. Funkcja uwzględnia *rzeczywisty* UID i *rzeczywisty* GID procesu. Zwraca 0, jeśli określony tryb dostępu jest dozwolony. Argumentem funkcji może być:

Tryb dostępu	Opis
F_OK	plik istnieje
R_OK	proces może czytać z pliku
W_OK	proces może pisać do pliku
X_OK	proces może wykonywać plik (przeszukiwać katalogi)

- Przykład:

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
int main (int argc, char* argv[]){
    char* sciezka = argv[1];
    int prawa;
    prawa = access (sciezka, F_OK);
    if (prawa == 0)
        printf ("plik %s istnieje\n", sciezka);
    else {
        if (errno == ENOENT) printf ("plik %s nie istnieje\n", sciezka);
        else if (errno == EACCES) printf ("plik %s jest niedostepny\n", sciezka);
        return 0;
    }
    prawa = access (sciezka, R_OK);
    if (prawa == 0)
        printf ("%s: prawa do czytania\n", sciezka);
    else
        printf ("%s: brak praw do czytania\n", sciezka);
    prawa = access (sciezka, W_OK);
    if (prawa == 0)
        printf ("%s: prawa do pisania\n", sciezka);
    else if (errno == EACCES)
        printf ("%s brak praw do pisania (odmowa dostepu)\n", sciezka);
    else if (errno == EROFS)
        printf ("%s brak praw do pisania (zamontowany tylko do czytania)\n",
            sciezka);
    return 0;
}
```


Zmiana praw dostępu do pliku

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *pathname, mode_t newmode);
int fchmod(int filedes, mode_t newmode);
```

- Zmiana praw dostępu dotyczy i-węzła, mimo, że podaje się nazwę. Prawa mogą być podane w postaci ósemkowej lub w postaci kombinacji praw dostępu i modyfikatorów połączonych alternatywą bitową.
- Tylko właściciel pliku lub administrator może zmieniać prawa dostępu do pliku. Czyli EUID procesu musi być taki sam jak EUID pliku lub proces musi mieć uprawnienia administratora.

Zmiana właściciela i grupy pliku

```
#include <unistd.h>
int chown(const char *pathname, uid_t owner_id, gid_t group_id);
int fchown(int filedes, uid_t owner_id, gid_t group_id);
int lchown(const char *pathname, uid_t owner_id, gid_t group_id);
```

- Każdy plik jest własnością jednego z użytkowników systemu. Funkcja `chown` pozwala zmieniać właściciela pliku.
- W wielu systemach Uniksowych (należy do nich Linuks) właściciela pliku może zmieniać tylko administrator systemu. Użytkownik może zmieniać grupę pliku, ale tylko na taką, do której sam należy. Zmiana właściciela i grupy pliku dotyczy i-węzła.
- Parametry `owner_id` i `group_id` wskazują nowego właściciela i nową grupę pliku. Wpisanie w jednej z tych pozycji `-1`, oznacza, że dana wartość nie ma ulec zmianie.

Zmiana znaczników czasowych pliku

- Z każdym plikiem związane są trzy czasy.

Pole w strukturze stat	Opis	Przykład funkcji, która zmienia wartość pola	Opcja polecenia ls
st_atime	ostatni dostęp do pliku	read	-u
st_mtime	ostatnia modyfikacja zawartości pliku	write	domyślnie
st_ctime	ostatnia modyfikacja atrybutów pliku	chmod, chown, write	-c

- Funkcja `utime` pozwala zmienić czas dostępu i modyfikacji zawartości pliku.

```
#include <sys/types.h>
#include <utime.h>
int utime(const char *pathname, const struct utimbuf *times);

struct utimbuf {
    time_t actime; /* czas dostępu */
    time_t modtime; /* czas modyfikacji zawartości*/
};
```

Czasy są podawane w postaci liczby sekund, które upłynęły od 1 stycznia 1970 godziny 0:00:00 UTC (*Coordinated Universal Time*). Jeśli zmiana czasu się powiedzie, zwracana jest wartość 0, w przeciwnym wypadku wartość -1.

- Jeśli argument `times` jest równy `NULL`, to obydwie czasy będą ustawione na czas bieżący. EUID procesu musi być równe EUID pliku, lub proces musi mieć prawo zapisu dla pliku.
- Jeśli argument `times` jest wskaźnikiem do struktury podającej wartości czasu, to EUID procesu musi być równe EUID pliku lub proces musi mieć uprawnienia administratora. Prawa zapisu do pliku nie są brane pod uwagę.
- Przykład:

```
struct utimbuf ut;
struct stat info;
time_t now;
time(&now); // pobierz bieżący czas
stat("plik", &info); // pobierz informacje o czasach pliku
ut.actime=info.st_atime; // tego czasu nie zmieniaj
ut.modtime=now-(24*60*60); // ten czas przesun o dzień do tyłu
utime("plik", &ut); // ustaw czas
```

Pliki tymczasowe

- Plik tymczasowy – plik tworzony na czas działania programu.
- Tworząc plik tymczasowy należy pamiętać o tym, że:
 - jednocześnie może być uruchomionych wiele programów i każdy z nich może tworzyć pliki tymczasowe
 - prawa dostępu do pliku tymczasowego powinny być tak ustawione, aby nieautoryzowani użytkownicy nie mogli zmienić wykonywania programu drogą zmiany pliku tymczasowego lub jego zastąpienia innym
 - nazwy plików tymczasowych powinny być tak generowane, aby nie można było nich z góry przewidzieć
- Przykładowe funkcje tworzenia plików tymczasowych:

```
#include <stdlib.h>
char *mkstemp(char *template);
```

```
#include <stdio.h>
FILE *tmpfile(void);
```

- Funkcja `mkstemp` generuje nazwę tymczasowego pliku na podstawie podanego wzorca, tworzy i otwiera ten plik. Wzorec budowany jest w ten sposób, że jako ostatnie 6 znaków nazwy wpisywany jest znak X. Ciąg ten zostanie zastąpiony przez funkcję unikatową sekwencją znaków. Funkcja zwraca deskryptor otworzonego pliku. Plik jest tworzony z prawami 0600 (r i w dla właściciela) - ostateczne prawa uwzględniają maskę i dostępem `O_EXCL`.
- Funkcja `tmpfile` tworzy i otwiera plik tymczasowy, zwraca wskaźnik do pliku. Plik jest automatycznie odłączany, tak więc jest automatycznie usuwany wtedy, kiedy deskryptor zostanie zamknięty lub program się zakończy. Plik otrzymuje prawa "w+b".
- Przykład 1:

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

```
int pisz_do_tymcz (char* bufor, size_t dlugosc) {
    char plik_tymcz_nazwa[] = "/tmp/temp_file.XXXXXXX";
    int fd = mkstemp (plik_tymcz_nazwa); /* utworz plik tymczasowy */
    unlink (plik_tymcz_nazwa);          /* odłącz plik tymczasowy */
    write (fd, &dlugosc, sizeof (dlugosc));
    write (fd, bufor, dlugosc);
    return fd;
}
```

```
char* czytaj_z_tymcz (int plik_tymcz, size_t* dlugosc) {
    char* bufor;
    int fd = plik_tymcz;
    lseek (fd, 0, SEEK_SET);
    read (fd, dlugosc, sizeof (*dlugosc));
    bufor = (char*) malloc (*dlugosc);
    read (fd, bufor, *dlugosc);
    close (fd); /* teraz plik tymczasowy zostanie usunięty */
    return bufor;
}
```

- Przykład 2.

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

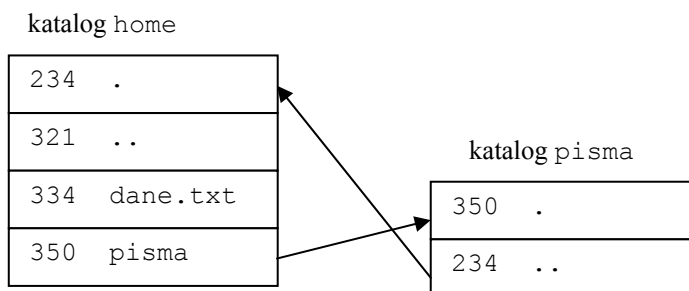
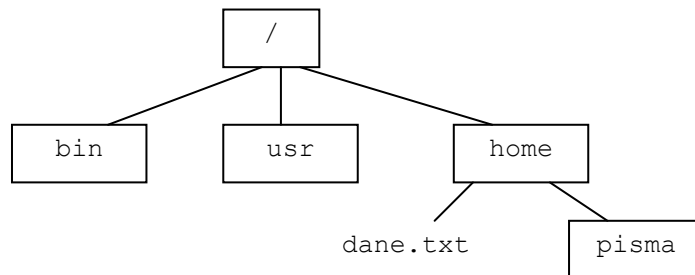
int main(int argc, char *argv[]) {
    FILE *tmpf = 0;
    char buf[128];
    if ( !(tmpf = tmpfile()) ) {
        fprintf(stderr, "%s: plik tymczasowy.\n",
                strerror(errno));
        abort();
    }

    fprintf(tmpf, "proces PID %ld zapisywal do pliku.\n",
            (long) getpid());
    fflush(tmpf);

    rewind(tmpf);
    fgets(buf, sizeof buf, tmpf);
    printf("Wczytalem: %s\n", buf);
    fclose(tmpf);
    return 0;
}
```

Działania na katalogach

- Katalogi służą Uniksowi do organizacji plików.
- Katalog jest to plik, o specjalnej zawartości.
- Katalog składa się z pozycji opisujących pliki zawarte w katalogu.
- Do katalogu można stosować wiele z funkcji działających na plikach zwykłych. Podstawowe różnice narzucone przez system to:
 - katalog ma swoją własną funkcję tworzenia `mkdir`
 - nie można zapisywać do katalogu za pomocą funkcji `write` - plik jest automatycznie umieszczony w katalogu za pomocą funkcji, która go tworzy



Pozycja katalogu zawiera numer i-wężła oraz nazwę pliku lub podkatalogu

Odczytywanie zawartości katalogu

```
#include <dirent.h>
#include <sys/types.h>
DIR *opendir(const char *dirname);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
void rewinddir(DIR *dirp);

struct dirent {
    ino_t d_ino; /* numer i-węzła */
    char d_name[NAME_MAX+1]; /* nazwa pliku w postaci napisu */
};
```

- Funkcja `opendir` otwiera katalog `dirname` do czytania. Zwraca wskaźnik do struktury `DIR`. Jest to wewnętrzna struktura używana przez funkcje działające na katalogu. Zwraca `NULL` wtedy, kiedy wystąpił błąd.
- Funkcja `readdir` zwraca wskaźnik do opisu kolejnej pozycji w katalogu (`struct dirent`). Funkcja zwraca `NULL` wtedy, kiedy wystąpił błąd oraz wtedy, gdy nie ma więcej pozycji w katalogu. Kolejność umieszczania nazw w katalogu zależy od implementacji. Również od implementacji zależy postać struktury `dirent`. W każdej z implementacji musi wystąpić pole `d_name`, zawierające nazwę pliku.
- Funkcja `closedir` zamyka katalog.
- Funkcja `rewinddir` ustawia strukturę `DIR` w taki sposób, aby następne wywołanie `readdir` zwróciło pierwszy plik w katalogu.
- Przykład: wypisanie wszystkich plików z bieżącego katalogu

```
#include <errno.h>
#include <dirent.h>
#include <stdio.h>

int main() {
    DIR * katalog;
    struct dirent * pozycja;
    if (!(katalog = opendir("."))) {
        perror("opendir");
        return 1;
    }
    /* zmienna errno jest ustawiana tylko w przypadku błędu */
    errno = 0;
    while ((pozycja = readdir(katalog))) {
        puts(pozycja->d_name);
        errno = 0;
    }
    if (errno) {
        perror("readdir");
        return 1;
    }
    closedir(katalog);
    return 0;
}
```

- Przykład: Chcemy uzyskać następujący wykaz dla podanego katalogu:

```
typ pliku          : pełna nazwa pliku
directory         : /home/.
directory         : /home/..
regular file      : /home/plik

#include <assert.h>
#include <dirent.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

const char* get_file_type (const char* path) {
    struct stat st;
    lstat (path, &st);
    if (S_ISLNK (st.st_mode))      return "symbolic link";
    else if (S_ISDIR (st.st_mode)) return "directory";
    else if (S_ISCHR (st.st_mode)) return "character device";
    else if (S_ISBLK (st.st_mode)) return "block device";
    else if (S_ISFIFO (st.st_mode)) return "fifo";
    else if (S_ISSOCK (st.st_mode)) return "socket";
    else if (S_ISREG (st.st_mode)) return "regular file";
    else assert (0);
}

int main (int argc, char* argv[]) {
    char* dir_path;
    DIR* dir;
    struct dirent* entry;
    char entry_path[PATH_MAX + 1];
    size_t path_len;

    if (argc >= 2)
        /* Jesli podano katalog, szukaj w tym katalogu. */
        dir_path = argv[1];
    else
        /* W przeciwnym wypadku weź katalog bieżący. */
        dir_path = ".";

    strncpy (entry_path, dir_path, sizeof (entry_path));
    path_len = strlen (dir_path);
    if (entry_path[path_len - 1] != '/') {
        entry_path[path_len] = '/';
        entry_path[path_len + 1] = '\0';
        ++path_len;
    }

    dir = opendir (dir_path);
    while ((entry = readdir (dir)) != NULL) {
        const char* type;
        strncpy (entry_path + path_len, entry->d_name,
                sizeof (entry_path) - path_len);
        type = get_file_type (entry_path);
        printf ("%18s: %s\n", type, entry_path);
    }

    closedir (dir);
    return 0;
}
```

Tworzenie i usuwanie katalogów

```
#include <fcntl.h>
#include <unistd.h>
int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);
```

- Funkcja `mkdir` tworzy nowy katalog o podanej nazwie (*pathname*) i prawach (*mode*). Prawa są modyfikowane przez maskę procesu. Wywołanie funkcji zakończy się niepowodzeniem, jeśli podana nazwa istnieje albo któraś ze składowych ścieżki nie jest katalogiem.
- Każdy nowoutworzony katalog zawiera dwie pozycje:
 - `.` (kropka) - jest to nazwa odnosząca się do bieżącego katalogu
 - `..` (dwie kropki) - katalog macierzysty
- Funkcja `rmdir` usuwa katalog o podanej nazwie (*pathname*). Katalog musi być pusty (to znaczy może zawierać tylko pozycje `.` oraz `..`).

Zmiana katalogu

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

```
char *getcwd(char *name, size_t size);
int chdir(const char *path);
int fchdir(int filedes);
int chroot(const char *pathname);
```

- Funkcja `getcwd` ustala pełną nazwę katalogu bieżącego. Jako argumenty tej funkcji podaje się nazwę bufora, do którego nazwa zostanie wpisana (*name*) oraz jego rozmiar (*size*). Jeśli nazwa przekracza `size-1`, funkcja zwraca `NULL` i ustawia błąd `ERANGE`. W przeciwnym razie zwraca wskaźnik do bufora *name*.
- Funkcje `chdir` i `fchdir` zmienia bieżący katalog roboczy. Funkcja może zakończyć się niepowodzeniem, jeśli argument określa plik, który nie jest katalogiem lub proces nie ma odpowiednich uprawnień.

- Przykład:

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

```
int main()
{
    char buf[BUFSIZ];
    char *cp;

    cp = getcwd(buf, sizeof(buf));    /* bieżący katalog roboczy */
    printf("Bieżący katalog: %s\n", buf);

    printf("Zmieniam na ..\n");
    chdir("..");                      /* wykonaj: cd .. */

    cp = getcwd(buf, sizeof(buf));    /* nowy bieżący katalog roboczy */
    printf("Bieżący katalog: %s\n", buf);

    return 0;
}
```

- Funkcja `chroot` zmienia katalog główny. System ma tylko jeden katalog główny, ale znaczenie / może być inne dla każdego procesu w systemie. Parametr *pathname* określa nowy katalog główny dla procesu. Funkcja `chroot` nie zmienia bieżącego katalogu! O ile nazwy bezwzględne są interpretowane względem nowego katalogu głównego, o tyle posługując się nazwą względną typu `../..katalog` nadal można dotrzeć do katalogu powyżej nowego katalogu głównego. Zatem wraz ze zmianą katalogu głównego należy zmienić również katalog bieżący.

- Przykład:

```
if (chroot("/nowy/hatalog/glowny) < 0) /* nowy katalog glowny / */
/* obsluga bledow */

if (chdir("/jakis/katalog") < 0)    /* nazwa wzgledem nowego / */
/* obsluga bledow */
```

Przeglądanie drzewa katalogów

- Przeglądanie drzewa katalogowego można zrealizować za pomocą otwierania kolejnych katalogów i wywoływania funkcji `stat()` dla jego pozycji, czynność tę należy powtarzać rekurencyjnie dla podkatalogów.
- Alternatywa: funkcja „file tree walk”

```
#include <ftw.h>

int ftw(const char *dir,                // katalog początkowy
        int (*fn)(const char *file,   // wskaźnik do funkcji
                  const struct stat *sb, int flag), // wywoływanej dla każdego
                                                // obiektu w katalogu
        int nopenfd);                 // maksymalna liczba otwartych katalogów

int nftw(const char *dir,              // katalog początkowy
         int (*fn)(const char *file,  // wskaźnik do funkcji
                   const struct stat *sb, // wywoływanej dla każdego
                   int flag, struct FTW *s), // obiektu w katalogu
         int nopenfd,                 // maksymalna liczba otwartych katalogów
         int flags);                  // flagi określające działanie funkcji
```

- Przykład 1: wyświetlanie fragmentu drzewa katalogów

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

int print (const char *name, const struct stat *stat_buf, int flags);

int main (int argc, char **argv)
{
    if (ftw (argv [1], print, 64) != 0)
    {
        perror("ftw failed");
        return 1;
    }
    return 0;
}

int print (const char *name, const struct stat *stat_buf, int flags)
{
    printf ("%s\n", name);
    return 0;
}
```

- Przykład 2: zliczanie plików wg typu w drzewie katalogowym

```
#define _XOPEN_SOURCE 1          /* Wymagane dla nftw() w GLIBC */
#define _XOPEN_SOURCE_EXTENDED 1 /* j.w. */

#include <stdio.h>
#include <sys/types.h>
#include <ftw.h>
#include <sys/param.h>
#include <sys/stat.h>
#include <dirent.h>

static int num_reg;
static int num_dir;
static int num_cspec;
static int num_bspec;
static int num_fifo;
static int num_sock;
static int num_symlink;
static int num_door;

int process (const char *path, const struct stat *stat_buf, int type,
            struct FTW *ftwp);

int main (int argc, char **argv)
{
    int i;

    for (i = 1; i < argc; i++) {
        num_reg = num_dir = num_cspec = num_bspec = 0;
        num_fifo = num_sock = num_symlink = num_door = 0;

        nftw (argv [i], process, FOPEN_MAX, FTW_PHYS);

        printf ("Totals for %s:\n", argv [i]);
        printf (" Regular files: %d\n", num_reg);
        printf (" Directories: %d\n", num_dir);
        printf (" Character special files: %d\n", num_cspec);
        printf (" Block special files: %d\n", num_bspec);
        printf (" FIFOs: %d\n", num_fifo);
        printf (" Sockets: %d\n", num_sock);
        printf (" Symbolic links: %d\n", num_symlink);
        printf (" Doors: %d\n", num_door);
    }

    return (0);
}

int process (const char *path, const struct stat *stat_buf, int type,
            struct FTW *ftwp)
{
    switch (type) {
        case FTW_F:
            switch (stat_buf -> st_mode & S_IFMT) {
                case S_IFREG:
                    num_reg++;
                    break;
            }
    }
}
```

```

    case S_IFCHR:
        num_cspec++;
        break;

    case S_IFBLK:
        num_bspec++;
        break;

    case S_IFIFO:
        num_fifo++;
        break;

}
break;

case FTW_D:
    num_dir++;
    break;

case FTW_SL:
case FTW_SLN:
    num_symlink++;
    break;

case FTW_DNR:
    printf("Can't read directory: %s\n", path);
    break;

case FTW_NS:
    printf("Can't stat %s\n", path);
    break;
}

return (0);
}

```

Przykład 3: wyświetlanie fragmentu drzewa katalogów wraz z określeniem typu pliku

```

#define _XOPEN_SOURCE 1          /* Wymagane dla nftw() w GLIBC */
#define _XOPEN_SOURCE_EXTENDED 1 /* j.w. */

#include <stdio.h>
#include <errno.h>
#include <getopt.h>
#include <ftw.h>
#include <limits.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

#define SPARE_FDS 5 /* fds for use by other functions, see text */

int process(const char *file, const struct stat *sb,
            int flag, struct FTW *s);

/* usage --- print message and die */

void usage(const char *name)
{
    fprintf(stderr, "usage: %s [-c] directory ...\n", name);
    exit(1);
}

```

```
/* main --- call nftw() on each command-line argument */

int main(int argc, char **argv)
{
    int i, c, nfd;
    int errors = 0;
    int flags = FTW_PHYS;
    char start[PATH_MAX], finish[PATH_MAX];

    while ((c = getopt(argc, argv, "c")) != -1) {
        switch (c) {
            case 'c':
                flags |= FTW_CHDIR;
                break;
            default:
                usage(argv[0]);
                break;
        }
    }

    if (optind == argc)
        usage(argv[0]);

    getcwd(start, sizeof start);

    nfd = getdtablesize() - SPARE_FDS; /* leave some spare descriptors */
    for (i = optind; i < argc; i++) {
        if (nftw(argv[i], process, nfd, flags) != 0) {
            fprintf(stderr, "%s: %s: stopped early\n",
                argv[0], argv[i]);
            errors++;
        }
    }

    if ((flags & FTW_CHDIR) != 0) {
        getcwd(finish, sizeof finish);
        printf("Starting dir: %s\n", start);
        printf("Finishing dir: %s\n", finish);
    }

    return (errors != 0);
}

/* process --- print out each file at the right level */

int process(const char *file, const struct stat *sb,
            int flag, struct FTW *s)
{
    int retval = 0;
    const char *name = file + s->base;

    printf("%*s", s->level * 4, ""); /* indent over */

    switch (flag) {
        case FTW_F:
            printf("%s (file)\n", name);
            break;
        case FTW_D:
            printf("%s (directory)\n", name);
            break;
        case FTW_DNR:
            printf("%s (unreadable directory)\n", name);
            break;
    }
}
```

```
case FTW_SL:
    printf("%s (symbolic link)\n", name);
    break;
case FTW_NS:
    printf("%s (stat failed): %s\n", name, strerror(errno));
    break;
case FTW_DP:
case FTW_SLN:
    printf("%s: FTW_DP or FTW_SLN: can't happen!\n", name);
    retval = 1;
    break;
default:
    printf("%s: unknown flag %d: can't happen!\n", name, flag);
    retval = 1;
    break;
}

return retval;
}
```

Zarządzanie deskryptorami plików

Tworzenie kopii deskryptora otwartego pliku

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

- Funkcja `dup` zwraca deskryptor związany z tym samym plikiem co `oldfd`. Jako wartość deskryptora wybiera *najmniejszą dostępną liczbę*. W przypadku niepowodzenia zwraca wartość `-1`.

W funkcji `dup2` można dodatkowo określić wartość nowego deskryptora (argument `newfd`). Jeśli plik związany z tym deskryptorem jest już otwarty, najpierw zostanie zamknięty.

Przykład zastosowania - przeadresowanie

- Zadanie: w programie początkowo skorzystamy ze standardowego wejścia, po pewnym czasie chcemy przełączyć standardowe wejście na czytanie z pliku.

Metoda 1: `close` i `open`

- Korzystamy z tego, że podczas otwierania pliku jest mu przypisywany najniższy dostępny deskryptor.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
    int fd;
    char wiersz[100];

    /* czytaj i drukuj trzy wiersze: domyslne przypisanie stdin */
    fgets( wiersz, 100, stdin ); printf("%s", wiersz );
    fgets( wiersz, 100, stdin ); printf("%s", wiersz );
    fgets( wiersz, 100, stdin ); printf("%s", wiersz );

    /* przeadresuj wejście */
    close(0);
    fd = open("dane.txt", O_RDONLY);
    if ( fd != 0 ){
        fprintf(stderr, "Nie udało się otworzyć pliku jako fd 0\n");
        exit(1);
    }

    /* czytaj i drukuj trzy wiersze: stdin przypisane do pliku */
    fgets( wiersz, 100, stdin ); printf("%s", wiersz );
    fgets( wiersz, 100, stdin ); printf("%s", wiersz );
    fgets( wiersz, 100, stdin ); printf("%s", wiersz );
    return 0;
}
```

Metoda 2: open ..close ..dup ..close

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
    int fd,nfd ;
    char wiersz[100];

    /* czytaj i drukuj trzy wiersze:
       domyslne przypisanie stdin */
    fgets( wiersz, 100, stdin ); printf("%s", wiersz );
    fgets( wiersz, 100, stdin ); printf("%s", wiersz );
    fgets( wiersz, 100, stdin ); printf("%s", wiersz );

    /* przeadresuj wejscie */
    fd = open("dane.txt", O_RDONLY);
    close(0);
    nfd=dup(fd);
    if ( nfd != 0 ){
        fprintf(stderr,"Nie udalo sie otworzyc plikui jako fd 0\n");
        exit(1);
    }
    close(fd);

    /* czytaj i drukuj trzy wiersze:
       stdin przypisane do pliku*/
    fgets( wiersz, 100, stdin ); printf("%s", wiersz );
    fgets( wiersz, 100, stdin ); printf("%s", wiersz );
    fgets( wiersz, 100, stdin ); printf("%s", wiersz );

    return 0;
}
```


Metoda 3: open .. dup2 ..close

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
    int  fd,nfd ;
    char  wiersz[100];

    /* czytaj i drukuj trzy wiersze:
       domyslne przypisanie stdin */
    fgets( wiersz, 100, stdin ); printf("%s", wiersz );
    fgets( wiersz, 100, stdin ); printf("%s", wiersz );
    fgets( wiersz, 100, stdin ); printf("%s", wiersz );

    /* przeadresuj wejscie */
    fd = open("dane.txt", O_RDONLY);
    nfd=dup2(fd,0);
    if ( nfd != 0 ){
        fprintf(stderr,"Nie udalo sie otworzyc plikui jako fd 0\n");
        exit(1);
    }
    close(fd);

    /* czytaj i drukuj trzy wiersze:
       stdin przypisane do pliku*/
    fgets( wiersz, 100, stdin ); printf("%s", wiersz );
    fgets( wiersz, 100, stdin ); printf("%s", wiersz );
    fgets( wiersz, 100, stdin ); printf("%s", wiersz );
    return 0;
}
```