2 Pliki

2.1 Podstawowe funkcje

1. Otworzenie pliku umożliwia funkcja systemowa `open`. Jej pierwszym argumentem jest ścieżka do pliku, który ma być otworzony. Drugim argumentem są flagi określające tryb otwarcia pliku. Trzeci (opcjonalny) argument określa prawa dostępu do nowego pliku, jeżeli ma on powstać w wyniku otwierania pliku. Najczęściej stosowane flagi dla funkcji `open` to:

   `O_RDONLY` otwarcie pliku tylko do odczytu,
   `O_WRONLY` otwarcie pliku tylko do zapisu,
   `O_RDWR` otwarcie pliku do odczytu i zapisu,
   `O_CREAT` flaga powodująca utworzenie otwieranego pliku w przypadku stwierdzenia jego braku.

   Oto przykładowe wywołanie funkcji `open`:

   ```
   fd = open("przyklad.txt", O_WRONLY | O_CREAT, 0644);
   ```

   Funkcja systemowa `open` zwraca *deskryptor pliku*. Deskryptor pliku jest liczbą, która powinna być interpretowana jako indeks do tablicy otwartych plików utrzymywanej przez system dla każdego procesu. Tablica ta jest niedostępna bezpośrednio dla programisty i zmiany do niej można wprowadzać jedynie poprzez wywołania odpowiednich funkcji systemowych. Poniższy przedstawiono przykładową zawartość tablicy otwartych plików:

0	`stdin`	standardowe wejście
1	`stdout`	standardowe wyjście
2	`stderr`	standardowe wyjście diagnostyczne
3	`przykład.txt`	otwarty plik
4		
...	...	

   Jak widać pierwsze 3 pozycje w tej tabeli są już wstępnie zainicjowane i reprezentują standardowe strumienie procesu. Funkcja `open` zajmuje zawsze pierwszą wolną pozycję w tablicy otwartych plików. Dla nowego procesu jest to więc pozycja o indeksie 3.

2. Zamknięcie pliku:

```
close(fd);
```

Funkcja systemowa `close` wymaga podania deskryptora otwartego pliku jako argumentu. Niezamknięte pliki są automatycznie zamykane przez system w momencie kończenia procesu.

3. Zapis do pliku:

```
write(fd, "Czesc", 5);
```

Drugi argument funkcji systemowej `write` jest adresem miejsca w pamięci, z którego będą pobierane dane do zapisu. W powyższym przypadku drugi argument reprezentuje adres statycznego napisu. Ostatni argument określa liczbę bajtów do zapisu. Funkcja zwraca liczbę faktycznie zapisanych bajtów.

4. Odczyt z pliku:

```
char buf[20];
...
n = read(fd, buf, 20);
```

Funkcja systemowa `read` ma te same argumenty co funkcja `write`. W tym przypadku jednak użyto tablicy znaków jako drugiego argumentu (bufor danych). Ostatni argument wskazuje na liczbę bajtów jakie mają być odczytane. Funkcja `read` zwraca liczbę bajtów jakie faktycznie udało się przeczytać (wartość większa od zera) lub `0` jeżeli wskaźnik plikowy dotarł do końca pliku.

5. Odczyt całego pliku wymaga wielokrotnego wywołania funkcji `read`, ponieważ rozmiar pliku z reguły nie jest znany z góry. Poniższy przykład pokazuje przykładową pętlę odczytującą zawartość pliku i wyświetlającą odczytane dane na standardowym wyjściu, a więc kierując te dane do deskryptora o numerze 1. Wypisywanie na ekranie dotyczy takiej liczby bajtów jak faktycznie udało się przeczytać.

```
while((n=read(fd, buf, 20)) > 0)
{
  write(1, buf, n);
}
```

6. Funkcje systemowe zwracają wartość liczbową wskazującą na status wykonania zleconej operacji. Z reguły wartość -1 sygnalizuje wystąpienie błędu. Szczegółowy kod błędu można odczytać badając wartość globalnej zmiennej `errno` typu `int`. Do obsługi błędów przydatna będzie funkcja biblioteczna `perror`, która bada wartość zmiennej `errno` i wyświetla tekstowy opis błędu, który wystąpił. Typowo więc obsługa błędów wygląda więc następująco:

```
fd = open("przyklad.txt", O_RDONLY);
if (fd == -1)
{
  printf("Kod: %d\n", errno);
  perror("Otwarcie pliku");
  exit(1);
}
```

Odwołanie do zmiennej `errno` wymaga załączenia pliku nagłówkowego `errno.h`.

Przetestuj obsługę błędów na nieistniejącym pliku oraz na pliku, do którego nie ma prawa do odczytu.

7. Przetestuj wyświetlanie systemowych komunikatów w języku polskim. W tym celu należy ustawić odpowiednie *locale* na początku programu:

```
setlocale(LC_ALL, "pl_PL.UTF-8");
```

8. Napisz program dopisujący dane na końcu istniejącego pliku. Można w tym celu wykorzystać otwieranie pliku w trybie „dopisywania”:

```
fd = open("przyklad.txt", O_WRONLY | O_APPEND | O_CREAT, 0644);
```

9. Usuń istniejący plik:

```
fd = unlink("przyklad.txt");
```

   Usuwanie pliku nie wymaga jego otwierania.

10. Skasuj zawartość pliku funkcją systemową `ftruncate`:

```
fd = open("przyklad.txt", O_WRONLY);
ftruncate(fd, 0);
```

11. Maska dla nowotworzonych plików:

```
# umask
022
# umask 077
```

   (zobacz również opis funkcji systemowej `umask`)

12. Przećwicz przemieszczanie się wewnątrz pliku funkcją `lseek`, której nagłówek jest następujący:

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

   Parametr `offset` określa przesunięcie a parametr `whence` określa punkt odniesienia: `SEEK_SET` — względem początku, `SEEK_CUR` — względem bieżącej pozycji, `SEEK_END` — względem końca pliku. Funkcja — w przypadku poprawnego wykonania — zwraca przesunięcie w bajtach względem początku pliku.

   Przykładowe zlecenie:

```
lseek(fd, -1, SEEK_END);
```

   spowoduje przesunięcie wskaźnika plikowego na pozycję przed ostatnim bajtem w pliku.

2.2 Implementacja prostych narzędzi systemu Unix

1. Zaproponuj implementację programu `cat` do wyświetlania danych ze wskazanego pliku na standardowym wyjściu. Dodaj możliwość czytania z wielu plików. Przykład użycia programu:

```
# mycat a.txt b.txt c.txt
```

2. Napisz program kopiujący zawartość pliku wskazanego pierwszym argumentem do pliku wskazanego drugim argumentem — analogicznie do standardowej komendy `cp`.
3. Rozważ implementację programu `tee` powielającego dane ze standardowego wejścia na standardowe wyjście i do pliku. Przykład użycia programu:

```
# ps ax | mytee wynik.log
```

4. Napisz program obliczający liczbę znaków i linii w pliku — analogicznie do standardowej komendy `wc`.
5. Napisz program rozpoznawający czy plik dany argumentem jest plikiem tekstowym, a więc zawierającym tylko znaki o kodach od 32 do 127.
6. Napisz program wyświetlający zawartość wskazanego pliku wielkimi literami. Zastosuj funkcję biblioteczną `toupper`. Program powinien więc działać analogicznie do zlecenia:

   ```
   # cat plik.txt | tr a-z A-Z
   ```

7. Napisz program sprawdzający czy zawartość dwóch plików wskazanych argumentami jest identyczna — analogicznie do standardowej komendy `cmp`.
8. Napisz program wyświetlający rozmiar pliku wskazanego argumentem. Do badania rozmiaru wykorzystaj funkcję `lseek`.
9. Napisz program wypisujący od końca (znak po znaku) zawartość wskazanego pliku. Zaproponuj rozwiązanie, które będzie gwarantowało wysoką wydajność.

2.3 Blokowanie dostępu do plików

1. Współbieżny dostęp do plików wymaga blokowania dostępu do nich w celu zabezpieczenia się przed niepoprawnymi wynikami. W najprostszym przypadku blokowanie można zrealizować na poziomie całego pliku przy użyciu funkcji `flock`:

   ```
   int fd = open(...);
   flock(fd, LOCK_EX);
   ...
   flock(fd, LOCK_UN);
   ```

   Blokada typu `LOCK_EX` jest blokadą wyłączną (do zapisu). Blokada dzielona (do odczytu) jest zakładana flagą `LOCK_SH`.

2. Blokowanie dostępu do pliku na poziomie poszczególnych bajtów można zrealizować funkcją `fcntl`. Poniższy fragment kodu zakłada i zdejmuje blokadę na pliku:

   ```
   struct flock lck;
   int fd = open(...);
   lck.l_type = F_WRLCK;
   lck.l_start = 10;
   lck.l_len = 20;
   lck.l_whence = SEEK_SET;
   fcntl(fd, F_SETLKW, &lock);
   ...
   lck.l_type = F_UNLCK;
   fcntl(fd, F_SETLKW, &lock);
   ```

   Blokada w powyższym przykładzie jest wyłączna (typ `F_WRLCK`). Blokada dzielona jest uzyskiwana poprzez ustawienie pola `l_type` na `F_RDLCK`.