

Systemy operacyjne

Sprawozdanie - laboratorium 2 „Podstawowe operacje na plikach”

Andrzej Kołakowski
296586

1) Podstawy obsługi plików w systemie UNIX

1. Odpowiedzi na pytania

- Co to są deskrytory plików?

Deskryptor pliku – identyfikator pliku wykorzystywany przez system operacyjny. Po wykonaniu operacji otwarcia pliku, deskryptor pliku może być wykorzystywany wielokrotnie przez wywołanie systemowe w operacjach wejścia/wyjścia. Zwracany przez funkcje systemowe z rodziny `open`. Zgodnie z POSIX deskryptor pliku to liczba całkowita, czyli wartość typu `int` z języka C.

- Jakie są standardowe deskrytory otwierane dla procesów?

Domyślnie każdy proces po uruchomieniu ma otwarte 3 standardowe deskrytory plików: 0 - `stdin`, 1 - `stdout`, 2 - `stderr`.

- Jakie flagi trzeba ustawić w funkcji `open` aby otrzymać funkcjonalność funkcji `creat`?

Wywołanie `creat` odpowiada wywołaniu `open` z ustawionymi flagami

`O_CREAT|O_WRONLY|O_TRUNC`.

- W wyniku wykonania polecenia `umask` otrzymano 0022. Jakie prawa dostępu będzie miał plik otwarty w następujący sposób: `open(pathname, O_RDWR | O_CREAT, S_IRWXU | S_IRWXG | S_IRWXO)`

Dla nowo tworzonych plików prawa dostępu są równe: $(\text{mode} \& \sim \text{umask})$, gdzie: `&` - bitowe AND, `~` - bitowa negacja.

```
111 111 111 == S_IRWXU | S_IRWXG | S_IRWXO == mode
111 101 101 == ~umask
111 101 101 == mode & ~umask
```

Zatem plik będzie miał prawa dostępu 755.

- Co oznaczają flagi: `O_WRONLY` | `O_CREAT` | `O_TRUNC`?

`O_WRONLY` - tylko do zapisu

`O_CREAT` - jeżeli plik nie istnieje to zostanie utworzony
`O_TRUNC` - jeżeli plik istnieje to zawartość zostanie wyczyszczona

- Co oznacza flaga `O_APPEND`?

`O_APPEND` - zapis na koniec pliku

- Co oznacza zapis: `S_IRUSR | S_IWUSR`?

Uprawnienia do odczytu i zapisu dla właściciela.

3. Program rozbudowany o obsługę błędów

copy1.c

2) Operacje pisania i czytania z pliku

- Czy w momencie powrotu z funkcji `write` dane są już zapisane na urządzenie wyjściowe?

Nie koniecznie, wciąż mogą znajdować się w systemowym buforze.

2. Program z sekcji 1 rozbudowany o sprawdzenie rezultatów wywołania funkcji `read` i `write`

copy2.c

3. Analiza programu z funkcją `writeall`

- Co robi ta funkcja?

Funkcja zapisuje określoną ilość bajtów podaną w parametrze.

- Jakiej sytuacji dotyczy wartość `EINTR`?

`EINTR` oznacza odebranie sygnału zanim jakiegokolwiek dane zostały zapisane, w tym wypadku funkcja ponawia zapis.

- Proszę napisać funkcję `readall`

readall.c

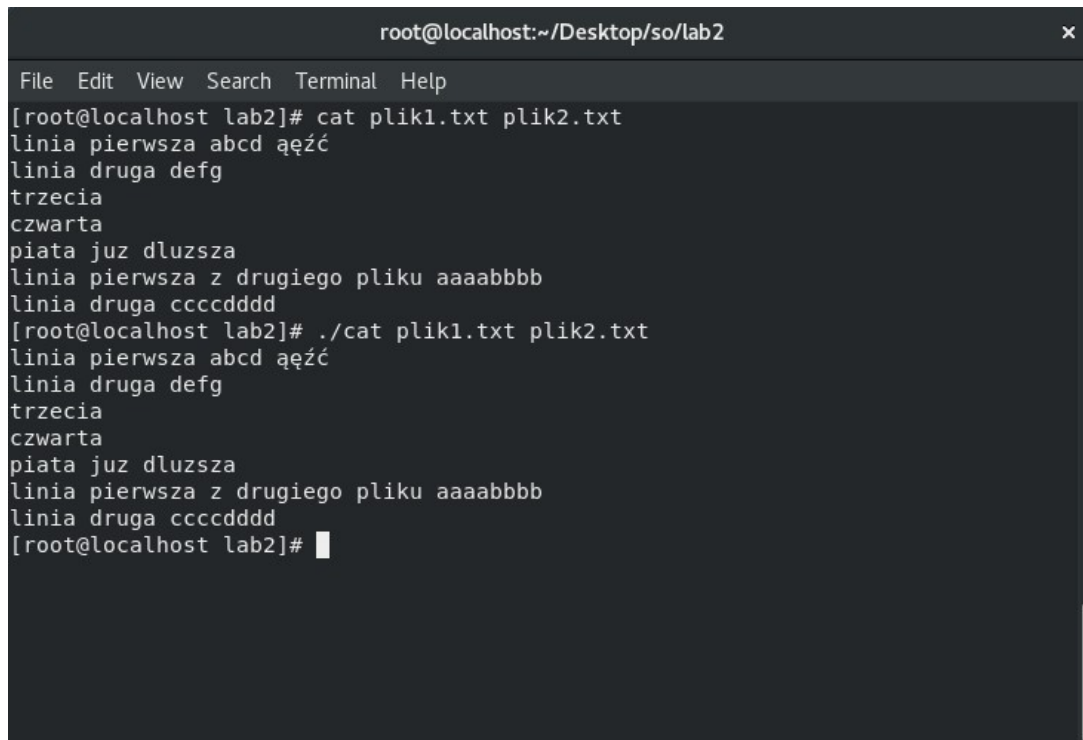
- Proszę zmodyfikować program z pierwszej sekcji włączając do niego zawartość funkcji `writeall`

copy3.c

- Proszę napisać program realizujący funkcjonalność polecenia `cat` bez opcji.
 - Polecenie powinno działać dla podanego pliku/plików oraz bez podania pliku (czytać ze standardowego wejścia)
 - Rozbudować program dodając obsługę wybranych opcji polecenia `cat`.

Dodano obsługę opcji `-n` (numerowanie linii) oraz `-E` (wyświetlanie `$` na końcu linii).
Możliwe jest łączenie opcji na przykład: `./cat -n -E` lub `./cat -nE`.

Porównanie „zwykłego” `cat` z moim:



```
root@localhost:~/Desktop/so/lab2
File Edit View Search Terminal Help
[root@localhost lab2]# cat plik1.txt plik2.txt
linia pierwsza abcd aęźć
linia druga defg
trzecia
czwarta
piata juz dluzsza
linia pierwsza z drugiego pliku aaaabbbb
linia druga ccccd
[root@localhost lab2]# ./cat plik1.txt plik2.txt
linia pierwsza abcd aęźć
linia druga defg
trzecia
czwarta
piata juz dluzsza
linia pierwsza z drugiego pliku aaaabbbb
linia druga ccccd
[root@localhost lab2]#
```

```
root@localhost:~/Desktop/so/lab2
File Edit View Search Terminal Help
[root@localhost lab2]# cat -n -E plik1.txt plik2.txt
 1 linia pierwsza abcd aęźć$
 2 linia druga defg$
 3 trzecia$
 4 czwarta$
 5 piata juz dluzsza$
 6 linia pierwsza z drugiego pliku aaaabbbb$
 7 linia druga cccddddd$
[root@localhost lab2]# ./cat -n -E plik1.txt plik2.txt
 1 linia pierwsza abcd aęźć$
 2 linia druga defg$
 3 trzecia$
 4 czwarta$
 5 piata juz dluzsza$
 6 linia pierwsza z drugiego pliku aaaabbbb$
 7 linia druga cccddddd$
[root@localhost lab2]#
```

```
root@localhost:~/Desktop/so/lab2
File Edit View Search Terminal Help
[root@localhost lab2]# cat
marek
marek
pawel
pawel
piotrek
piotrek
^C
[root@localhost lab2]# ./cat
marek
marek
pawel
pawel
piotrek
piotrek
^C
[root@localhost lab2]#
```

cat.c

3) Wskaźnik pliku i sygnalizator `O_APPEND`

- Dwa deskryptory: `fd1` i `fd2` użyto do otwarcia pliku podając tą samą ścieżkę dostępu do pliku. Wskaźnik pliku ustawiony jest na początku pliku. Następnie korzystając z deskryptora `fd1` wykonano operację zapisania 100b do pliku. Następnie przy użyciu

deskryptora `fd2` wykonano operację czytania z pliku. Pytanie: Na jakiej pozycji jest wskaźnik pliku? Jakie dane odczytano przy użyciu `fd2`?

Wskaźnik pliku jest na początku, odczytano te dane, które zostały zapisane przez `fd1`.

- Do otwarcia pliku użyto jednego deskryptora `fd3`. Następnie wykonano kolejno operację pisania 100b i czytania 100b. Na jakiej pozycji jest wskaźnik pliku? Co zostało przeczytane?

Wskaźnik pliku jest na pozycji 200b, przeczytany został zakres 100-200b.

- Czy każdorazowe poprzedzenie operacji pisania ustawieniem wskaźnika pliku na końcu pliku za pomocą funkcji `lseek` daje taki sam rezultat jak otwarcie pliku w trybie z ustawioną flagą `O_APPEND`? Odpowiedź uzasadnij.

Tak, flaga `O_APPEND` powoduje ustawienie wskaźnika pliku na końcu pliku przed każdą operacją `write`.

- Jak wygląda wywołanie funkcji `lseek` które:
 - ustawia wskaźnik na zadanej pozycji:
`lseek(deskryptor, pozycja, SEEK_SET)`
 - znajduje koniec pliku:
`lseek(deskryptor, 0, SEEK_END)`
 - zwraca bieżącą pozycję wskaźnika:
`lseek(deskryptor, 0, SEEK_CUR)`

5. Program testujący funkcję `backward`

backward.c

6. Program testujący funkcję `backward`, (`lseek`, `read`) -> `pread`

backward2.c, taka zamiana jest równoważna.

7. Program realizujący funkcjonalność polecenia `tail`

tail.c

4) Buforowanie operacji I/O

3. Program testujący ile czasu zajmują funkcje kopiujące `copy2` i `copy3` (z sekcji 4)

copy2io.c, copy3io.c

Wyniki:

```
[root@localhost io]# ./copy2io plik plik1
(null):
    "Total (user/sys/real)", 0, 4, 5
    "Child (user/sys)", 0, 0

[root@localhost io]# ./copy3io plik plik1
(null):
    "Total (user/sys/real)", 3, 3, 9
    "Child (user/sys)", 0, 0
```

Można zauważyć, że funkcja `copy2`, korzystająca z systemowych funkcji `read` i `write` wykorzystuje najwięcej czasu procesora w trybie jądra. Funkcja `copy3` wykorzystuje funkcje biblioteczne, zwykle wygodniejsze w użyciu, jednak kosztem dodatkowego czasu spędzonego w trybie użytkownika.

4. Program liczący czas wykonania funkcji `copy2` przy różnych rozmiarach bufora

copy2buf.c

Wyniki:

```
[root@localhost io]# ./copy2buf plik plik1
(1):
    "Total (user/sys/real)", 36, 1198, 1276
    "Child (user/sys)", 0, 0
(512):
    "Total (user/sys/real)", 1, 4, 4
    "Child (user/sys)", 0, 0
(1024):
    "Total (user/sys/real)", 0, 2, 11
    "Child (user/sys)", 0, 0
(1100):
    "Total (user/sys/real)", 0, 3, 12
    "Child (user/sys)", 0, 0
```

Największy przeskok w wykorzystanym czasie procesora zauważalny jest między buforem 1-bajtowym a 512-bajtowym. Przy większych rozmiarach różnice są nieznaczne.