Spis treści

Na podstawie:

- *M.J.Rochkind - Programowanie w systemie UNIX dla zaawansowanych (Advanced UNIX Programming)*
- `https://ai.ia.agh.edu.pl/wiki/pl:dydaktyka:so:2017:labs:lab_intro`
- *Graham Glass, King Ables - Linux dla programistów i użytkowników*

1 Podstawowe pojęcia zarządzania plikami

1.1. Deskryptory

Deskryptor

Każdy proces ma powiązanie ze sobą dane systemowe jak zostało omówione w Lab1. Daną taką stanowi również zbiór deskryptorów plików. Deskryptor pliku to najzwyczajniej mała liczba całkowita (max. wartość można sprawdzić przez wywołanie sysconf(_SC_OPEN_MAX));) przypisana do pliku który został otworzony przez proces (jedna dla każdego pliku).
Defaultowo, otwarte są 3 deskryptory gdy proces się zaczyna, standardowe wejście, wyjście i wyjście błędów (deskryptory 0 do 2). W programie w języku C deskryptor reprezentuje się po prostu przez zmienną typu int.
Aby uzyskać deskryptor posługujemy się funkcjami takimi jak open(można nią otworzyć pliki regularne, specjalne, FIFO), pipe(dla nienazwanych potoków), socket, accept, connect(w programowaniu sieciowym).
Kilka różnych deskryptorów może wskazywać ten sam plik, także wewnątrz jednego procesu.

1.2. Prawa dostępu

Jesteśmy przyzwyczajeni do reprezentowania praw przy pomocy 9 liter np.
-rwxr-xr-x
lub równoważnie przy pomocy cyfr:
755
Takie reprezentacje są widoczne po wykonaniu polecenia ls.
W wywołaniach systemowych możliwe jest użycie innej reprezentacji, która jest w formie:

S_I**pwww**

p - pozwolenie (R, W lub X)
www - dla kogo (USR, GRP lub OTH)
Daje nam to 9 możliwych symboli. Dla przykładu 755 jest reprezentowane jako:
S_IRUSR | S_IWUSR | S_IXUSR | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH

Można też przyznać wszystkie prawa dla danej grupy korzystając ze składni

S_I**RWXw**

gdzie w zastępujemy przez U, G lub O. 755 można alternatywnie zapisać jako:
S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH
Jako, że często używamy tylko dwóch rodzajów pozwoleń - dla katalogów i dla plików - warto zdefiniować makra, które im odpowiadają, np.:

```
1 #define PERM_DIRECTORY    S_IRWXU
2 #define PERM_FILE         (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
```

1.3. Maska praw dostępu

Maska to 4 cyfry których ósemkowa reprezentacja jest negowana, a następnie poddawana koniunkcji z prawami dostępu. Tylko prawa które uzyskały 1 są nadawane. Służy to ograniczeniu możliwości nadania pewnych praw. Maskę można sprawdzić przy pomocy umask w konsoli.
Łatwiej zrozumieć to w ten sposób, że jeśli w masce na jakimś miejscu występuje 1, to prawo odpowiadające temu miejscu nie zostanie nadane choćbyśmy je jawnie przydzielili. np. dla maski 0002, nie zostanie nadane prawo zapisywanie dla 'innych', nawet jeśli wykonamy instrukcję przydzielającą prawa z prawami 0777.

2 Wywołania systemowe

2.1. open

open służy do otworzenia(regularnego, specjalnego, FIFO) lub stworzenia pliku(tylko pliki regularne). Pliki specjalne tworzy mknod, a FIFO funkcja mkfifo. Po otwarciu/utworzeniu pliku w przypadku powodzenia zwracany jest deskryptor pliku, który można użyć w wywołaniach innych funkcji. W przypadku niepowodzenia zwracana jest wartość -1 i ustawiana jest zmienna errno.

```
int open(
      const char *path, /* pathname */
      int flags, /* flags */
      mode_t perms /* permissions (when creating) */
); /* Returns file descriptor or -1 on error (sets errno) */
```

2.2. open - otwieranie istniejącego pliku

Przy otwieraniu istniejącego pliku określamy plik podając ścieżkę do niego oraz flagi. Flaga określa cel otwierania pliku. Są trzy możliwe flagi:

- O_RDONLY - tylko odczyt
- O_WRONLY - tylko zapis
- O_RDWR - zapis i odczyt

Dla danego procesu sprawdzane jest czy może on wykonać open na pliku na podstawie praw, które przypisane są do userID procesu otwierającego plik. Przy operacji otwierania pliku, tylko dwa argumenty są podawane do funkcji. offset w czytaniu i zapisywaniu jest ustawiony na pierwszy bajt pliku. Przykład

```
fd = open("/home/marc/oldfile", O_RDONLY);
```

2.3. open - tworzenie nowego pliku

Jeśli przekazany do open plik nie istnieje, to open stworzy nowy plik, pod warunkiem, że podamy flagę O_CREAT. Oprócz tego można podać inne flagi konfiguracyjne:

- O_RDONLY - tylko odczyt
- O_WRONLY - tylko zapis
- O_RDWR - zapis i odczyt
- O_APPEND - ustaw wskaźnik pliku na końcu przed każdym write()
- O_EXCL - jeśli ustawione jest O_CREAT i plik istnieje, to wywołanie się nie powiedzie

- O_TRUNC - jeśli plik istnieje to zostanie obcięty do zerowej długości

Przy tworzeniu pliku niezbędne jest podanie trzeciego argumentu określającego prawa dostępu (argument ten nie ma żadnego wpływu jeśli wołamy open dla istniejącego pliku). Co jeśli utworzymy plik do zapisu i odczytu, ale trzeci argument na to nie zezwoli ? Wówczas jako, że plik jest nowy, to odczyt i zapis są dostępne zaraz po stworzeniu. Przy ponownym otwarciu pliku prawa będą już zgodne z trzecim argumentem. Oczywiście przy nadawaniu praw w ten sposób należy wziąć pod uwagę maskę, której wartość jest aplikowana w tym procesie. Przykład

```
fd = open("/home/marc/oldfile", O_RDONLY | O_CREAT, 0600);
```

2.4. creat

Kombinacja flag O_CREAT | O_WRONLY | O_TRUNC występuje bardzo często przy tworzeniu plików, dlatego stworzono dla niej specjalną funkcję systemową creat.

```
int creat(
  const char *path, /* pathname */
  mode_t perms /* permissions */
);
```

2.5. Właściciele nowego pliku

Pytanie, które może się nasuwać to kto jest właścicielem nowo powstałego pliku. userID jest ustawione na takie jak userID procesu, a groupID jest ustawiane takie jak katalogu lub groupID procesu, o tym która z tych metod została wybrana dla groupID można się dowiedzieć przez wywołanie funkcji stat.

2.6. umask - ustawianie maski dla procesu

Wywołanie systemowe umask pozwala na ustawienie maski dla praw dostępu, dla aktualnego procesu.

```
mode_t umask(
  mode_t cmask /* new mask */
);
```

Funkcja ta zwraca zawsze wartość poprzedniej maski, nie nowo ustawionej. Aby sprawdzić przy jej pomocy aktualną maskę należy zamienić maskę na dowolną inną, a następnie z powrotem na taką jaka była. Typowa wartość maski dla procesu to S_IWGRP | S_IWOTH (czyli 022). Przykład wywołania:

```
  umask(S_IWGRP | S_IWOTH);
```

2.7. unlink

Jak wiemy w systemach unixowych funkcjonuje pojęcie link, które oznacza odwołanie do innego pliku lub katalogu za pomocą nazwy, co jest niewidoczne na poziomie aplikacji. Takich odwołań do tego samego pliku może być wiele. Funkcja systemowa unlink usuwa link z katalogu redukując liczbę linków dla danego pliku o 1, jeśli jest to ostatni link wskazujący plik, to usuwa cały plik wraz z jego i-node.

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

Funkcja ta nie powinna być używana dla katalogów, dla katalogów przeznaczona jest funkcja rmdir.
Jeśli funkcja unlink będzie miała usunąć plik, który został otwarty w jakimś procesie, to usunięcie nastąpi dopiero gdy proces ten zamknie plik lub sam proces przestanie istnieć. Ta cecha powoduje, że unlink jest często używane do usuwania plików tymczasowych, które mają zostać usunięte po zakończeniu procesu. Przykład

```
    fd = open("temp", O_RDWR | O_CREAT | O_TRUNC | O_EXCL, 0);
    unlink("temp");
```

2.8. tmpnam

Funkcja ta ma zwracań unikalną nazwę, która może być nazwą pliku tymczasowego (po to aby nie istniały dwa pliki tymczasowe o tej samej nazwie), mimo, że zwraca ona unikalną wartość to i tak istnieje pewne prawdopodobieństwo, że pomiędzy jej wywołaniem, a wywołaniem open nastąpi utworzenie innego pliku tymczasowego o tej samej nazwie przez inny proces. Przykład wywołania

```
    pathname = tmpnam(NULL);
    /* tu moze nastapic stworzenie pliku o tej nazwie przez inny proces */
    fd = open(pathname, O_RDWR | O_CREAT | O_TRUNC | O_EXCL, 0);
    unlink(pathname)
```

Nie ma gwarancji unikalności nazwy, aby rozwiązać ten problem powstała funkcja mkstemp.

2.9. mkstemp

mkstemp gwarantuje, że plik zostanie stworzony z unikalną nazwą.

```
int mkstemp(
    char *template /* template for file name */
); /* Returns open file descriptor or -1 on error (may not set errno) */
```

plik jest otwierany do zapisu i odczytu. Funkcja zwraca -1 w przypadku niepowodzenia lub wartość deskryptora w przypadku powodzenia. Do funkcji należy przekazać argument będący ciągiem znaków, który na końcu ma 6 znaków X. Przykład

```
char pathname[] = "/tmp/dataXXXXXX";
fd = mkstemp(pathname)
unlink(pathname)
printf("%s\n", pathname);
```

2.10. O_APPEND i lseek

Jak wiemy O_APPEND jest flagą powiązaną z offsetem dla pliku. Jeśli jej nie ustawimy to przy czytaniu lub pisaniu do pliku zaczynamy od jego początku. Offset jest wspólny dla czytania i zapisywania. Jeśli zapiszemy do danego pliku, a później zaczniemy czytać, to rozpoczniemy czytanie tam gdzie skończyliśmy zapisywanie.
lseek to funkcja systemowa pozwalająca na sprawdzenie gdzie jest offset, a także na ustawienie offsetu. Funkcja lseek jest automatycznie wołana gdy ustawiona jest flaga O_APPEND przy każdym zapisie do pliku, powoduje to, że każdy zapis jest dodawany na koniec pliku.

```
off_t lseek(int fd, off_t przesuniecie, int tryb)
```

Trzy możliwe tryby przesunięcia to:

- SEEK_SET - względem początku pliku
- SEEK_CUT - względem bieżącej pozycji w pliku
- SEEK_END - względem końca pliku

lseek zwraca bieżącą pozycję w pliku jeśli się powiedzie lub -1 w przypadku niepowodzenia(pozycja nieosiągalna, np. przed początkiem pliku).

2.11. write

```
ssize_t write(
  int fd, /* file descriptor */
  const void *buf, /* data to write */
  size_t nbytes /* amount to write */
);
```

write zapisuje nbytes bajtów do pliku wskazywanego przez deskryptor fd. Zapis rozpoczyna się od obecnej pozycji offsetu i jest zwiększany po zapisie. Funkcja zwraca liczbę zapisanych bajtów lub -1 w przypadku niepowodzenia. Jeśli ustawiona została flaga O_APPEND to zapis rozpoczynany jest od końca pliku. write może być użyte do zapisu nie tylko do plików regularnych, ale składania jest wtedy nieco inna.
Wywołania write są bardzo szybkie, ponieważ po wywołaniu write tak naprawdę dane nie są od razu zapisywane do pliku docelowego, są one buforowane i czekają gdy kernel będzie 'miał czas' aby je zapisać. Jeśli przed zapisem jakiś proces zacznie czytać plik, to dane zostaną mu udostępnione z bufora.

2.12. read

```
ssize_t read(
  int fd, /* file descriptor */
  void *buf, /* address to receive data */
  size_t nbytes /* amount to read */
); /* Returns number of bytes read or -1 on error (sets errno) */
```

read jest przeciwieństwem write. Czyta nbytes bajtów do buf, z pliki reprezentowanego przed fd. read zaczyna na aktualnej pozycji offsetu pliku, a następnie offset jest zwiększany o liczbę przeczytanych bajtów. read zwraca liczbę przeczytanych bajtów, 0 gdy napotka koniec pliku oraz -1 w przypadku błędu.

2.13. close

```
int close(
  int fd /* file descriptor */
); /* Returns 0 on success or -1 on error (sets errno) */
```

close sprawia, że deskryptor powiązany z plikiem jest odkładany do ponownego użycia.

2.14. pread i pwrite

```
ssize_t pread(
  int fd, /* file descriptor */
  void *buf, /* address to receive data */
  size_t nbytes, /* amount to read */
  off_t offset /* where to read */
); /* Returns number of bytes read or -1 on error (sets errno) */
```

```
ssize_t pwrite(
  int fd, /* file descriptor */
  const void *buf, /* data to write */
  size_t nbytes, /* amount to write */
  off_t offset /* where to write */
); /* Returns number of bytes written or -1 on error (sets errno) */
```

pread i pwrite są podobne do read i write z tym, że są poprzedzane wywołaniem lseek. Flaga O_APPEND nie ma żadnego wpływu na pwrite. offset pliku nie jest wcale używany, ponieważ przekazywany jest bezpośrednio przez te funkcje. Wywołania pread i pwrite eliminują problem możliwej zmiany offsetu pomiędzy wywołaniami lseek oraz write/read.

3 Buforowane I/O

...