

### 3. Організація взаємодії процесів через *pipe* і *FIFO* в *Linux*

#### 3.1. Поняття про потік вводу-виводу

Серед всіх категорій засобів комунікації найбільш вживаними є **канали зв'язку**, що забезпечують досить безпечну та інформативну взаємодію процесів.

Існують **дві моделі передачі даних** по каналах зв'язку – **потік вводу-виводу** і **повідомлення**. Серед них найпростішою є потокова модель, у якій для операції прийому/передачі інформації взагалі не важливо, що передається або приймається. Вся інформація в каналі зв'язку розглядається як неперервний потік байтів, що не володіє ніякою внутрішньою структурою. Вивченню механізмів, що забезпечують потокову передачу даних в операційній системі *Linux*, і буде присвячена ця тема.

#### 3.2. Поняття про роботу з файлами через системні виклики і стандартну бібліотеку вводу-виводу для мови *C*

Потокова передача інформації може здійснюватися не тільки між процесами, але й між процесом і пристроєм вводу-виводу, наприклад між процесом і диском, на якому дані зображаються у вигляді файлу. Системні виклики, які використовуються для потокової роботи з файлом, багато в чому відповідають системним викликам, що застосовуються для потокового спілкування процесів. Почнемо розгляд саме з механізму потокового обміну між процесом і файлом.

З курсу програмування мовою *C* Вам відомі функції роботи з файлами зі стандартної бібліотеки вводу-виводу: **fopen()**, **fread()**, **fwrite()**, **fprintf()**, **fscanf()**, **fgets()**. Ці функції входять як невід'ємна частина в стандарт ANSI на мову *C* і дозволяють програмістові одержувати інформацію з файлу або записувати її у файл за умови, що програміст має певні знання про вміст переданих даних. Так, наприклад, функція **fgets()** використовується для введення з файлу послідовності символів, що закінчується символом '\n' – переходу на новий рядок. Функція **fscanf()** здійснює ввід інформації, що відповідає заданому формату. В потоковій моделі операції, що визначаються функціями стандартної бібліотеки вводу-виводу, не є потоковими операціями, тому що кожна з них вимагає наявності деякої структури переданих даних.

В операційній системі *Linux* ці функції є надбудовою – сервісним інтерфейсом – над системними викликами, що здійснюють прямі потокові операції обміну інформацією між процесом і файлом, і не потребують ніяких знань про те, що вона містить. Трохи пізніше коротко познайомимося із системними викликами **open()**, **read()**, **write()** і **close()**, які застосовуються для такого обміну, але спочатку введемо ще одне поняття – поняття файлового дескриптора.

### 3.3. Файловий дескриптор

Інформація про файли, які використовуються процесом, входить до складу його системного контексту і зберігається в його блоці керування – **PCB**. Можна спрощено вважати, що в операційній системі UNIX інформація про файли, з якими процес здійснює операції потокового обміну, разом з інформацією про потокові лінії зв'язку, що з'єднують процес із іншими процесами і пристроями вводу-виводу, зберігається в деякому масиві, який одержав назву **таблиця відкритих файлів** або **таблиця файлових дескрипторів**. Індекс елемента цього масиву, що відповідає певному потоку вводу-виводу, одержав назву **файлового дескриптора** для цього потоку. Таким чином, файловий дескриптор – це невелике ціле невід'ємне число, яке для поточного процесу в даний момент часу однозначно визначає деякий діючий канал вводу-виводу. Деякі файлові дескриптори на етапі старту будь-якої програми асоціюються із стандартними потоками вводу-виводу. Так, наприклад, файловий дескриптор 0 відповідає стандартному потоку введення, файловий дескриптор 1 – стандартному потоку виводу, файловий дескриптор 2 – стандартному потоку для виводу помилок. У нормальному інтерактивному режимі роботи стандартний потік введення зв'язує процес із клавіатурою, а стандартні потоки виводу і виводу помилок – з поточним терміналом.

Детальніше будову структур даних, що містять інформацію про потоки вводу-виводу, асоційованих з процесом, будемо розглядати пізніше, при вивченні організації файлових систем в Linux (теми 7 і 8).

### 3.4. Відкриття файлу. Системний виклик `open()`

Файловий дескриптор використовується як параметр, що описує потік вводу-виводу для системних викликів, які виконують операції над цим потоком. Тому перш ніж здійснювати операції читання даних з файлу і запису їх у файл, потрібно помістити інформацію про файл у таблицю відкритих файлів і визначити відповідний файловий дескриптор. Для цього застосовується процедура відкриття файлу, яка здійснюється системним викликом `open()`.

Системний виклик `open`

Прототип системного виклику

```
#include <fcntl.h>
int open(char *path, int flags);
int open(char *path, int flags, int mode);
```

Опис системного виклику

Системний виклик `open` призначений для виконання операції відкриття файлу `i`, у випадку її вдалого здійснення, повертає файловий дескриптор відкритого файлу (невелике невід'ємне ціле число, яке використовується надалі для інших операцій із цим файлом).

Параметр **path** є вказівником на рядок, який містить повне або відносне ім'я файлу.

Параметр **flags** може приймати одне з наступних трьох значень:

- **O\_RDONLY** – якщо над файлом надалі будуть відбуватися тільки операції читання;
- **O\_WRONLY** – якщо над файлом надалі будуть здійснюватися тільки операції запису;
- **O\_RDWR** – якщо над файлом будуть здійснюватися операції читання і операції запису.

Кожне із цих значень може бути скомбіноване за допомогою операції "побітове або (|)" з одним або декількома прапорцями:

- **O\_CREAT** – якщо файлу із зазначеним іменем не існує, він повинен бути створений;
- **O\_EXCL** – застосовується разом із прапорцем **O\_CREAT**. При спільному їхньому використанні та існуванні файлу із зазначеним іменем, відкриття файлу не здійснюється і виникає помилка;
- **O\_NDELAY** – забороняє переведення процесу в стан очікування при виконанні операції відкриття і будь-яких наступних операціях над цим файлом;
- **O\_APPEND** – при відкритті файлу і перед виконанням кожної операції запису (якщо вона, звичайно, дозволена) покажчик поточної позиції у файлі встановлюється на кінець файлу;
- **O\_TRUNC** – якщо файл існує, зменшити його розмір до 0, зі збереженням існуючих атрибутів файлу, крім, можливо, часу останнього доступу до файлу і його останньої модифікації.

Крім того, у деяких версіях операційної системи UNIX можуть застосовуватися додаткові значення прапорців:

- **O\_SYNC** – будь-яка операція запису у файл буде блокуватися (тобто процес буде переведений у стан очікування) доти, поки записана інформація не буде фізично розміщена на відповідний нижній рівень – hardware;
- **O\_NOCTTY** – якщо ім'я файлу належить до термінального пристрою, воно не стає керуючим терміналом процесу, навіть якщо до цього процес не мав керуючого терміналу.

Параметр **mode** встановлює атрибути прав доступу різних категорій користувачів до нового файлу при його створенні. Він обов'язковий, якщо серед заданих прапорців присутній прапорець **O\_CREAT**, і може бути опущений у протилежному випадку. Цей параметр задається як сума наступних вісімкових значень:

- 0400 – дозволене читання для користувача, що створив файл;
- 0200 – дозволений запис для користувача, що створив файл;
- 0100 – дозволене виконання для користувача, що створив файл;
- 0040 – дозволене читання для групи користувача, що створив файл;

- 0020 – дозволений запис для групи користувача, що створив файл;
- 0010 – дозволене виконання для групи користувача, що створив файл;
- 0004 – дозволене читання для всіх інших користувачів;
- 0002 – дозволений запис для всіх інших користувачів;
- 0001 – дозволене виконання для всіх інших користувачів.

При створенні файлу реально встановлювані права доступу одержуються зі стандартної комбінації параметра **mode** і маски створення файлів поточного процесу **umask**, а саме – вони рівні **mode & ~umask**.

При відкритті файлів типу **FIFO** системний виклик має деякі особливості поведінки в порівнянні з відкриттям файлів інших типів. Якщо **FIFO** відкривається тільки для читання, і не заданий прапорець **O\_NDELAY**, то процес, що здійснив системний виклик, блокується доти, поки який-небудь інший процес не відкриє **FIFO** на запис. Якщо прапорець **O\_NDELAY** заданий, то повертається значення файлового дескриптора, асоційованого з **FIFO**. Якщо **FIFO** відкривається тільки для запису, і не заданий прапорець **O\_NDELAY**, то процес, що здійснив системний виклик, блокується доти, поки який-небудь інший процес не відкриє **FIFO** для читання. Якщо прапорець **O\_NDELAY** заданий, то констатується виникнення помилки і повертається значення -1.

### Значення, що повертається

Системний виклик повертає значення файлового дескриптора для відкритого файлу при нормальному завершенні і значення -1 при виникненні помилки.

Системний виклик **open()** використовує набір прапорців для того, щоб вказати операції, які передбачається застосовувати до файлу надалі або які повинні бути виконані безпосередньо в момент відкриття файлу. Із усього можливого набору прапорців спочатку нас будуть цікавити тільки прапорці **O\_RDONLY**, **O\_WRONLY**, **O\_RDWR**, **O\_CREAT** і **O\_EXCL**. Перші три прапорці є взаємовиключними: хоча б один з них повинен бути застосований і наявність одного з них не допускає наявності двох інших. Ці прапорці описують набір операцій, які, при успішному відкритті файлу, будуть дозволені над файлом надалі: тільки читання, тільки запис, читання і запис. З матеріалів теми 1 відомо, що у кожного файлу існують атрибути прав доступу для різних категорій користувачів. Якщо файл із заданим іменем існує на диску, і права доступу до нього для користувача, від імені якого працює поточний процес, не суперечать потрібному набору операцій, то операційна система сканує таблицю відкритих файлів від її початку до кінця в пошуках першого вільного елементу, заповнює його і повертає індекс цього елемента як файловий дескриптор відкритого файлу. Якщо файлу на диску немає, не вистачає прав або відсутнє вільне місце в таблиці відкритих файлів, то констатується виникнення помилки.

У випадку, коли ми припускаємо, що файл на диску може бути

відсутній, і хочемо, щоб він був створений, прапорець для набору операцій повинен використовуватися в комбінації із прапорцем **O\_CREAT**. Якщо файл існує, то все відбувається за розглянутим вище сценарієм. Якщо файлу немає, спочатку відбувається створення файлу з набором прав, вказаних у параметрах системного виклику. Перевірка відповідності набору операцій оголошеним правам доступу може і не відбуватися (як, наприклад, в Linux).

У випадку, коли потрібно, щоб файл на диску був відсутній і був створений у момент відкриття, прапорець для набору операцій повинен використовуватися в комбінації із прапорцями **O\_CREAT** і **O\_EXCL**.

Роботу системного виклику **open()** із прапорцями **O\_APPEND** і **O\_TRUNC** розглянемо під час вивчення теми 7, яка присвячена організації файлових систем в Linux.

### 3.5. Системні виклики **read()**, **write()**, **close()**

Для здійснення потокових операцій читання інформації з файлу і її запису у файл застосовуються системні виклики **read()** і **write()**.

#### Системні виклики **read** і **write**

##### Прототипи системних викликів

```
#include <sys/types.h>
#include <unistd.h>
size_t read(int fd, void *addr, size_t nbytes);
size_t write(int fd, void *addr, size_t nbytes);
```

##### Опис системних викликів

Системні виклики **read** і **write** призначені для здійснення потокових операцій введення (читання) і виводу (запису) інформації над каналами зв'язку, які описуються файловими дескрипторами, тобто для **файлів, pipe, FIFO і socket**.

Параметр **fd** є файловим дескриптором створеного раніше потокового каналу зв'язку, через який буде відсилатися або одержуватися інформація, тобто значення, яке повернув один із системних викликів **open()**, **pipe()** або **socket()**.

Параметр **addr** – адреса області пам'яті, починаючи з якої буде одержуватися інформація для передачі або розміщення прийнятої інформації.

Параметр **nbytes** для системного виклику **write** визначає кількість байт, яка повинна бути передана, починаючи з адреси пам'яті **addr**. Параметр **nbytes** для системного виклику **read** визначає кількість байт, яку ми хочемо одержати з каналу зв'язку і розмістити в пам'яті, починаючи з адреси **addr**.

##### Значення, що повертається

У випадку успішного завершення системний виклик повертає

кількість реально відісланих або прийнятих байт. Відмітимо, що це значення (більше або рівне 0) може не збігатися із заданим значенням параметра **nbytes**, а може бути меншим у зв'язку з відсутністю місця на диску або в лінії зв'язку при передачі даних або відсутності інформації при її прийомі. При виникненні якої-небудь помилки повертається від'ємне значення.

### Особливості поведінки при роботі з файлами

При роботі з файлами інформація записується у файл або читається з файлу, починаючи з місця, яке визначається покажчиком поточної позиції у файлі. Значення покажчика збільшується на кількість реально прочитаних або записаних байт. При читанні інформації з файлу вона не пропадає з нього. Якщо системний виклик **read** повертає значення 0, то це означає, що файл прочитаний до кінця.

Зараз не будемо акцентувати увагу на понятті покажчика поточної позиції у файлі і взаємному впливі значення цього покажчика та поведінки системних викликів. Це питання розглянемо під час вивчення теми 7.

Після завершення поточкових операцій процес повинен виконати операцію закриття потоку вводу-виводу, під час якої відбудеться остаточне скидання буферів на лінії зв'язку, звільняться виділені ресурси операційної системи, і елемент таблиці відкритих файлів, що відповідає файловому дескриптору, буде позначений як вільний. За ці дії відповідає системний виклик **close()**. Відзначимо, що при завершенні роботи процесу (див. тему 2) за допомогою явного або неявного виклику функції **exit()** відбувається автоматичне закриття всіх відкритих потоків вводу-виводу.

### Системний виклик close

#### Прототип системного виклику

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

#### Опис системного виклику

Системний виклик **close** призначений для коректного завершення роботи з файлами та іншими об'єктами вводу-виводу, які описуються в операційній системі через файлові дескриптори: **pipe**, **FIFO**, **socket**.

Параметр **fd** є дескриптором відповідного об'єкта, тобто значенням, що повернув один із системних викликів **open()**, **pipe()** або **socket()**.

#### Значення, що повертається

Системний виклик повертає значення 0 при нормальному завершенні і значення -1 при виникненні помилки.

## 3.6. Програма для запису інформації у файл

```

#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
int main(){
    int fd;
    size_t size;
    char string[] = "Hello, world!";
    /* Зануляємо маску створення файлів поточного процесу
       для того, щоб права доступу в створюваного
       файлу точно відповідали параметру виклику open()
    */
    (void)umask(0);
    /*Спробуємо відкрити файл із іменем myfile у
    поточній директорії тільки для операцій виводу. Якщо
    файл не існує, спробуємо його створити із правами
    доступу 0666, тобто read-write для всіх категорій
    користувачів */
    if((fd = open("myfile", O_WRONLY | O_CREAT,
        0666)) < 0){
        /* Якщо файл відкрити не вдалося, друкуємо про
           це повідомлення і припиняємо роботу */
        printf("Can't open file\n");
        exit(-1);
    }
    /* Намагаємось записати у файл 14 байт із масиву,
    тобто весь рядок "Hello, world!" разом з ознакою
    кінця рядка */
    size = write(fd, string, 14);
    if(size != 14){
        /* Якщо записалася менша кількість байт,
           повідомляємо про помилку */
        printf("Can't write all string\n");
        exit(-1);
    }
    /* Закриваємо файл */
    if(close(fd) < 0){
        printf("Can't close file\n");
    }
    return 0;
}

```

*Лістинг 3.1. Програма 03-1.c, що ілюструє використання системних викликів open(), write() і close() для запису інформації у файл*

### 3.7. Поняття про pipe. Системний виклик pipe()

Найпростішим способом для передачі інформації за допомогою потокової моделі між різними процесами або навіть усередині одного процесу в операційній системі Linux є **pipe** (канал, труба, конвеєр).

**Важлива відмінність pipe від файлу полягає в тому, що прочитана інформація негайно видаляється з нього і не може бути прочитана повторно.**

**Pipe** можна уявити собі у вигляді труби обмеженої ємності, розташованої всередині адресного простору операційної системи, доступ до вхідного і вихідного отвору якої здійснюється за допомогою системних викликів. Насправді **pipe** – область пам'яті, недоступна користувачьким процесам напряду, найчастіше організована у вигляді кільцевого буфера (хоча існують і інші види організації). При операціях читання і запису по буферу переміщуються два покажчики, що відповідають вхідному і вихідному потокам. При цьому вихідний покажчик ніколи не може перегнати вхідний і навпаки. Для створення нового екземпляра такого кільцевого буфера всередині операційної системи використовується системний виклик **pipe()**.

#### Системний виклик pipe

##### Прототип системного виклику

```
#include <unistd.h>
int pipe(int *fd);
```

##### Опис системного виклику

Системний виклик **pipe** призначений для створення **pipe** усередині операційної системи.

Параметр **fd** є покажчиком на масив із двох цілих змінних. При нормальному завершенні виклику в перший елемент масиву – **fd[0]** – буде занесено файловий дескриптор, що відповідає вихідному потоку даних **pipe**, який дозволяє виконувати тільки операцію читання, а в другий елемент масиву – **fd[1]** – буде занесено файловий дескриптор, що відповідає вхідному потоку даних і дозволяє виконувати тільки операцію запису.

##### Значення, що повертається

Системний виклик повертає значення 0 при нормальному завершенні і значення -1 при виникненні помилок.

У процесі роботи системний виклик організовує виділення ділянки пам'яті під буфер і покажчики та заносить інформацію, що відповідає вхідному і вихідному потокам даних, у два елементи таблиці відкритих



файлів, зв'язуючи тим самим з кожним **pipe** два файлових дескриптори. Для одного з них дозволена тільки операція читання з **pipe**, а для іншого – тільки операція запису в **pipe**. Для виконання цих операцій ми можемо використати ті ж самі системні виклики **read()** і **write()**, що й при роботі з файлами. Після закінчення використання вхідного або/і вихідного потоку даних, потрібно закрити відповідний потік за допомогою системного виклику **close()** для звільнення системних ресурсів. Відзначимо, що, коли всі процеси, які використовують **pipe**, закривають всі асоційовані з ним файлові дескриптори, операційна система ліквідує **pipe**. Таким чином, час існування **pipe** в системі не може перевищувати час життя процесів, що працюють з ним.

### 3.8. Програма для роботи з **pipe** в одному процесі

Досить яскравою ілюстрацією дій із створення **pipe**, запису в нього даних, читання з нього і звільнення виділених ресурсів може бути програма, що організує роботу з **pipe** у рамках одного процесу, яка наведена нижче:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(){
    int fd[2];
    size_t size;
    char string[] = "Hello, world!";
    char resstring[14];
    /* Спробуємо створити pipe */
    if(pipe(fd) < 0){
        /* Якщо створити pipe не вдалося, друкуємо про це
           повідомлення і припиняємо роботу */
        printf("Can\'t create pipe\n");
        exit(-1);
    }
    /* Намагаємось записати в pipe 14 байт із нашого
       масиву, тобто весь рядок "Hello, world!" разом з
       ознакою кінця рядка */
    size = write(fd[1], string, 14);
    if(size != 14){
        /* Якщо записалася менша кількість байт,
           повідомляємо про помилку */
        printf("Can\'t write all string\n");
        exit(-1);
    }
    /* Намагаємось прочитати з pipe 14 байт в інший
       масив, тобто весь записаний рядок */
    size = read(fd[0], resstring, 14);
    if(size < 0){
```

```

    /* Якщо прочитати не змогли, повідомляємо про
    помилку */
    printf("Can\'t read string\n");
    exit(-1);
}
/* Друкуємо прочитаний рядок */
printf("%s\n", resstring);
/* Закриваємо вхідний потік*/
if(close(fd[0]) < 0){
    printf("Can\'t close input stream\n");
}
/* Закриваємо вихідний потік*/
if(close(fd[1]) < 0){
    printf("Can\'t close output stream\n");
}
return 0;
}

```

*Лістинг 3.2. Програма 03-2.c, що ілюструє роботу з pipe у рамках одного процесу*

### 3.9. Організація зв'язку через pipe між процесом-батьком і процесом-нащадком. Успадкування файлових дескрипторів при викликах `fork()` і `exec()`

Зрозуміло, що якби вся перевага **pipe** зводилася до заміни функції копіювання з пам'яті в пам'ять всередині одного процесу на пересилання інформації через операційну систему, то це нічого б не вартувало. Однак таблиця відкритих файлів успадковується процесом-нащадком при породженні нового процесу системним викликом **fork()** і входить до складу незмінної частини системного контексту процесу при системному виклику **exec()** (за винятком тих потоків даних, для файлових дескрипторів яких була спеціальними засобами виставлена ознака, що спонукає операційну систему закрити їх при виконанні **exec()**, однак їхній розгляд виходить за рамки нашого курсу). Ця обставина дозволяє організувати передачу інформації через **pipe** між рідними процесами, що мають спільного предка, який створив **pipe**.

Розглянемо програму, яка здійснює однонаправлений зв'язок між процесом-батьком і процесом-нащадком:

```

#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(){
    int fd[2], result;

```

```

size_t size;
char resstring[14];
/* Спробуємо створити pipe */
if(pipe(fd) < 0){
    /* Якщо створити pipe не вдалося, друкуємо про
       це повідомлення і припиняємо роботу */
    printf("Can\'t create pipe\n");
    exit(-1);
}
/* Породжуємо новий процес */
result = fork();
if(result){
    /* Якщо створити процес не вдалося, сповіщаємо
       про це і завершуємо роботу */
    printf("Can\'t fork child\n");
    exit(-1);
} else if (result > 0) {
    /* Ми перебуваємо в батьківському процесі, що
       буде передавати інформацію процесу-нащадку. У
       цьому процесі вихідний потік даних нам не
       знадобиться, тому закриваємо його.*/
    close(fd[0]);
    /* Пробуємо записати в pipe 14 байт, тобто весь
       рядок "Hello, world!" разом з ознакою кінця
       рядка */
    size = write(fd[1], "Hello, world!", 14);
    if(size != 14){
        /* Якщо записалася менша кількість байт,
           повідомляємо про помилку і завершуємо роботу
           */
        printf("Can\'t write all string\n");
        exit(-1);
    }
    /* Закриваємо вхідний потік даних.
       Батько припиняє роботу */
    close(fd[1]);
    printf("Parent exit\n");
} else {
    /* Перебуваємо в породженому процесі, що буде
       одержувати інформацію від процесу-батька. Він
       успадкував від батька таблицю відкритих файлів
       і, знаючи файлові дескриптори, що відповідають
       pipe, може його використовувати. У цьому процесі
       вхідний потік даних нам не буде потрібний, тому
       закриваємо його.*/
    close(fd[0]);
    /* Намагаємось прочитати з pipe 14 байт у масив,
       тобто весь записаний рядок */

```

```

size = read(fd[0], resstring, 14);
if(size < 0){
    /* Якщо прочитати не змогли, повідомляємо
    про помилку і завершуємо роботу */
    printf("Can't read string\n");
    exit(-1);
}
/* Друкуємо прочитаний рядок */
printf("%s\n", resstring);
/* Закриваємо вхідний потік і завершуємо роботу*/
close(fd[0]);
}
return 0;
}

```

*Лістинг 3.3. Програма 03-3.c, що здійснює однонаправлений зв'язок через pipe між процесом-батьком і процесом-нащадком*

**Pipe** призначений для організації однонаправленого або симплексного зв'язку. Якби в попередньому прикладі ми спробували організувати через **pipe** двосторонній зв'язок, коли процес-батько пише інформацію в **pipe**, припускаючи, що її одержить процес-нащадок, а потім читає інформацію з **pipe**, припускаючи, що її записав породжений процес, то могла б виникнути ситуація, у якій процес-батько прочитав би власну інформацію, а процес-нащадок не одержав би нічого. Для використання одного **pipe** у двох напрямках необхідні спеціальні засоби синхронізації процесів. Найпростіший спосіб організації двонаправленого зв'язку між рідними процесами полягає у використанні двох **pipe**.

Відзначимо, що в деяких UNIX-подібних системах (наприклад, в Solaris2) реалізовані повністю дуплексні **pipe**. У таких системах для обох файлових дескрипторів, асоційованих з **pipe**, дозволені і операція читання, і операція запису. Однак таке поведіння не характерне для **pipe** і не переноситься на всі системи.

### 3.10. Особливості поведінки викликів read() і write() для pipe

Системні виклики **read()** і **write()** мають певні особливості виконання при роботі з **pipe**, пов'язані з його обмеженим розміром, затримками в передачі даних і можливістю блокування процесів, що обмінюються інформацією. Організація заборони блокування цих викликів для **pipe** виходить за рамки нашого курсу.

Будьте уважні при написанні програм, що обмінюються великими обсягами інформації через **pipe**. Пам'ятайте, що за один раз із **pipe** може прочитатися і записатися менше інформації, ніж Ви хотіли. Перевіряйте

### значення, що повертаються викликами!

Одна з особливостей виконання системного виклику `read()`, що блокується, пов'язана зі спробою читання з порожнього `pipe`. Якщо є процеси, у яких цей `pipe` відкритий для запису, то системний виклик блокується і чекає появи інформації. Якщо таких процесів немає, він поверне значення 0 без блокування процесу. Ця особливість приводить до **необхідності закриття** файлового дескриптора, **асоційованого із вхідним кінцем `pipe`, у процесі, що буде використовувати `pipe` для читання** (`close(fd[1])` у процесі-нащадку в лістингу програми 3.3). Аналогічною особливістю виконання за відсутності процесів, у яких `pipe` відкритий для читання, володіє і системний виклик `write()`, із чим зв'язана **необхідність закриття** файлового дескриптора, **асоційованого з вихідним кінцем `pipe`, у процесі, що буде використовувати `pipe` для запису** (`close(fd[0])` у процесі-батькові в тій же програмі).

#### Системні виклики `read` і `write` (продовження)

Особливості виконання при роботі з `pipe`, `FIFO` і `socket`

#### Системний виклик `read`

Ситуація	Поведінка
Спроба прочитати менше байт, ніж є наявності в каналі зв'язку.	Читає необхідну кількість байт і повертає значення, що відповідає прочитаній кількості. Прочитана інформація вилучається з каналу зв'язку.
У каналі зв'язку є менше байт, ніж потрібно, але не нульова кількість.	Читає все, що є в каналі зв'язку, і повертає значення, що відповідає прочитаній кількості. Прочитана інформація вилучається з каналу зв'язку.
Спроба прочитати з каналу зв'язку, у якому немає інформації. Блокування виклику дозволене.	Виклик блокується доти, поки не з'явиться інформація в каналі зв'язку і поки існує процес, що може передати в нього інформацію. Якщо інформація з'явилася, то процес розблокується, і поведінка виклику визначається двома попередніми рядками таблиці. Якщо в канал нікому передати дані (немає жодного процесу, у якого цей канал зв'язку відкритий для запису), то виклик повертає значення 0. Якщо канал зв'язку повністю закривається для запису під час блокування процесу, що читає дані, то процес розблоковується, і системний виклик

<p>Спроба прочитати з каналу зв'язку, у якому немає інформації. Блокування виклику не дозволене.</p>	<p>повертає значення 0.</p> <p>Якщо є процеси, у яких канал зв'язку відкритий для запису, системний виклик повертає значення -1 і встановлює змінну <b>errno</b> у значення <b>EAGAIN</b>. Якщо таких процесів немає, системний виклик повертає значення 0.</p>
<p>Системний виклик write</p>	
<p>Ситуація</p> <p>Спроба записати в канал зв'язку менше байт, ніж залишилося до його заповнення.</p> <p>Спроба записати в канал зв'язку більше байт, ніж залишилося до його заповнення. Блокування виклику дозволене.</p> <p>Спроба записати в канал зв'язку більше байт, ніж залишилося до його заповнення, але менше, ніж розмір буфера каналу зв'язку. Блокування виклику заборонене.</p> <p>У каналі зв'язку є місце. Спроба записати в канал зв'язку більше байт, ніж залишилося до його заповнення, і більше, ніж розмір буфера каналу зв'язку. Блокування виклику заборонене.</p> <p>Спроба запису в канал</p>	<p>Поведінка</p> <p>Необхідна кількість байт поміщається в канал зв'язку, повертається записана кількість байт.</p> <p>Виклик блокується доти, поки всі дані не будуть поміщені в канал зв'язку. Якщо розмір буфера каналу зв'язку менше, ніж передана кількість інформації, то виклик буде чекати, поки частина інформації не буде зчитана з каналу зв'язку. Повертається записана кількість байт.</p> <p>Системний виклик повертає значення -1 і встановлює змінну <b>errno</b> у значення <b>EAGAIN</b>.</p> <p>Записується стільки байт, скільки залишилося до заповнення каналу. Системний виклик повертає кількість записаних байт.</p> <p>Системний виклик повертає значення -1 і</p>

зв'язку, у якому немає місця. Блокування виклику не дозволене.

Спроба запису в канал зв'язку, з якого нікому більше читати, або повне закриття каналу на читання під час блокування системного виклику.

встановлює змінну **errno** у значення **EAGAIN**.

Якщо виклик був заблокований, то він розблокується. Процес одержує сигнал **SIGPIPE**. Якщо цей сигнал обробляється користувачем, то системний виклик поверне значення -1 і встановить змінну **errno** у значення **EPIPE**.

Необхідно відзначити додаткову особливість системного виклику **write** при роботі з **pipe** і **FIFO**. Запис інформації, розмір якої не перевищує розмір буфера, повинен здійснюватися атомарно – одним шматком. Цим пояснюється ряд блокувань і помилок у попередньому переліку.

### 3.11. Поняття про FIFO. Використання системного виклику **mknod()** для створення FIFO. Функція **mkfifo()**

Доступ до інформації про розташування **pipe** в операційній системі і його стан може бути здійснений тільки через таблицю відкритих файлів процесу, що створив **pipe**, і через успадковані від нього таблиці відкритих файлів процесів-нащадків. Тому викладений вище механізм обміну інформацією через **pipe** справедливий лише для рідних процесів, що мають загального предка, який ініціював системний виклик **pipe()**, або для таких процесів і самого предка і не може використовуватися для потокового спілкування з іншими процесами. В операційній системі Linux існує можливість використання **pipe** для взаємодії інших процесів, але її реалізація складна і не розглядається в межах курсу.

Для організації потокової взаємодії будь-яких процесів в операційній системі Linux застосовується засіб зв'язку, що одержав назву **FIFO** (від First Input First Output) або іменованний **pipe**. **FIFO** подібний до **pipe**. Але дані про розташування **FIFO** в адресному просторі ядра і його стан процеси можуть одержувати не через родинні зв'язки, а через файлову систему. Для цього при створенні іменованого **pipe** на диску створюється файл спеціального типу, звертаючись до якого процеси можуть одержати необхідну інформацію. Для створення **FIFO** використовується системний виклик **mknod()** або існуюча в деяких версіях UNIX функція **mkfifo()**.

Зазначимо, що при їхній роботі не відбувається реального виділення області адресного простору операційної системи під іменованний **pipe**, а тільки створюється файл-мітка, існування якої дозволяє здійснити реальну організацію **FIFO** у пам'яті при його відкритті за допомогою вже відомого системного виклику **open()**.

Після відкриття іменованій **pipe** поводитьсь точно так само, як і неіменованій. Для подальшої роботи з ним застосовуються системні виклики **read()**, **write()** і **close()**. Час існування **FIFO** в адресному просторі ядра операційної системи, як і у випадку з **pipe**, не може перевищувати час життя останнього з процесів, що його використали. Коли всі процеси, що працюють із **FIFO**, закривають всі файлові дескриптори, асоційовані з ним, система звільняє ресурси, виділені під **FIFO**. Вся непрочитана інформація втрачається. Файл-мітка залишається на диску і може використовуватися для нової реальної організації **FIFO** у подальшому.

## Використання системного виклику **mknod** для створення **FIFO**

### Прототип системного виклику

```
#include <sys/stat.h>
#include <unistd.h>
int mknod(char *path, int mode, int dev);
```

### Опис системного виклику

Оскільки метою є не повний опис системного виклику **mknod**, а тільки опис його використання для створення **FIFO**, будемо розглядати не всі можливі варіанти задання параметрів, а тільки ті з них, які відповідають цій специфічній потребі.

Параметр **dev** не є важливим для нашого випадку, а тому будемо завжди задавати його рівним 0.

Параметр **path** є покажчиком на рядок, що містить повне або відносне ім'я файлу, який буде міткою **FIFO** на диску. Для успішного створення **FIFO** файлу з таким іменем перед викликом не повинно існувати.

Параметр **mode** встановлює атрибути прав доступу різних категорій користувачів до **FIFO**. Цей параметр задається як результат побітової операції "або" значення **S\_IFIFO**, який вказує, що системний виклик повинен створити **FIFO**, і деякої суми наступних вісімкових значень:

- 0400 – дозволене читання для користувача, що створив **FIFO**;
- 0200 – дозволений запис для користувача, що створив **FIFO**;
- 0040 – дозволене читання для групи користувача, що створив **FIFO**;
- 0020 – дозволений запис для групи користувача, що створив **FIFO**;
- 0004 – дозволене читання для всіх інших користувачів;
- 0002 – дозволений запис для всіх інших користувачів.

При створенні **FIFO** реально встановлювані права доступу одержуються зі стандартної комбінації параметра **mode** і маски створення файлів поточного процесу **umask**, а саме – вони рівні **(0777 & mode) & ~umask**.

Значення, що повертається



При успішному створенні **FIFO** системний виклик повертає значення 0, при неуспішному – від'ємне значення.

## Функція mkfifo

### Прототип функції

```
#include <sys/stat.h>
#include <unistd.h>
int mkfifo(char *path, int mode);
```

### Опис функції

Функція **mkfifo** призначена для створення **FIFO** в операційній системі.

Параметр **path** є покажчиком на рядок, що містить повне або відносне ім'я файлу, який буде міткою **FIFO** на диску. Для успішного створення **FIFO** файлу з таким іменем перед викликом функції не повинно існувати.

Параметр **mode** встановлює атрибути прав доступу різних категорій користувачів до **FIFO**. Цей параметр задається як деяка сума наступних вісімкових значень, які описані вище для **mknod**.

При створенні **FIFO** реально встановлювані права доступу одержуються зі стандартної комбінації параметра **mode** і маски створення файлів поточного процесу **umask**, а саме – вони рівні **(0777 & mode) & ~umask**.

### Значення, що повертається

При успішному створенні **FIFO** функція повертає значення 0, при неуспішному – від'ємне значення.

Важливо розуміти, що файл типу **FIFO** не призначений для розміщення на диску інформації, яка записується в іменованій **pipe**. Ця інформація розташовується всередині адресного простору операційної системи, а файл є тільки міткою, яка створює передумови для її розміщення.

Не намагайтеся переглянути вміст цього файлу за допомогою Midnight Commander (mc)! Це приведе до його повного зависання!

## 3.12. Особливості поведінки виклику open() при відкритті FIFO

Системні виклики **read()** і **write()** при роботі з **FIFO** мають ті ж особливості поведінки, що й при роботі з **pipe**. Системний виклик **open()** при відкритті **FIFO** також поводить трохи інакше, ніж при відкритті інших типів файлів, що пов'язане з можливістю блокування процесів, які його виконують. Якщо **FIFO** відкривається тільки для читання, і прапорець **O\_NDELAY** не заданий, то процес, що здійснив системний виклик, блокується

доти, поки який-небудь інший процес не відкриє **FIFO** для запису. Якщо прапорець **O\_NDELAY** заданий, то повертається значення файлового дескриптора, асоційованого з **FIFO**. Якщо **FIFO** відкривається тільки для запису, і прапорець **O\_NDELAY** не заданий, то процес, який здійснив системний виклик, блокується доти, поки який-небудь інший процес не відкриє **FIFO** для читання. Якщо прапорець **O\_NDELAY** заданий, то констатується виникнення помилки і повертається значення **-1**. Задання прапорця **O\_NDELAY** у параметрах системного виклику **open()** приводить і до того, що процесу, який відкрив **FIFO**, забороняється блокування при виконанні наступних операцій читання із цього потоку даних і запису в нього.

### 3.13. Програма з використанням FIFO у рідних процесах

Для ілюстрації взаємодії процесів через **FIFO** розглянемо таку програму:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main(){
    int fd, result;
    size_t size;
    char resstring[14];
    char name[]="aaa.fifo";
    /* Зануляємо маску створення файлів поточного
    процесу для того, щоб права доступу в створюваного
    FIFO точно відповідали параметру виклику mknod() */
    (void)umask(0);
    /* Спробуємо створити FIFO з іменем aaa.fifo у
    поточній директорії */
    if(mknod(name, S_IFIFO | 0666, 0) < 0){
        /* Якщо створити FIFO не вдалося, друкуємо про
        це повідомлення і припиняємо роботу */
        printf("Can\'t create FIFO\n");
        exit(-1);
    }
    /* Породжуємо новий процес */
    if((result = fork()) < 0){
        /* Якщо створити процес не вдалося, сповіщаємо
        про це і завершуємо роботу */
        printf("Can\'t fork child\n");
        exit(-1);
    } else if (result > 0) {
        /* Перебуваємо в батьківському процесі, який
        буде передавати інформацію процесу-нащадку. У
```

```

        цьому процесі відкриваємо FIFO на запис.*/
        if((fd = open(name, O_WRONLY)) < 0){
            /* Якщо відкрити FIFO не вдалося, друкуємо
               про це повідомлення і припиняємо роботу */
            printf("Can\'t open FIFO for writing\n");
            exit(-1);
        }
        /* Намагаємось записати в FIFO 14 байт, тобто
           весь рядок "Hello, world!" разом з ознакою кінця
           рядка */
        size = write(fd, "Hello, world!", 14);
        if(size != 14){
            /* Якщо записалася менша кількість байт, то
               повідомляємо про помилку і завершуємо роботу */
            printf("Can\'t write all string to
                   FIFO\n");
            exit(-1);
        }
        /* Закриваємо вхідний потік даних і на цьому
           батько припиняє роботу */
        close(fd);
        printf("Parent exit\n");
    } else {
        /* Перебуваємо в породженому процесі, що буде
           одержувати інформацію від процесу-батька.
           Відкриваємо FIFO на читання.*/
        if((fd = open(name, O_RDONLY)) < 0){
            /* Якщо відкрити FIFO не вдалося, друкуємо
               про це повідомлення і припиняємо роботу */
            printf("Can\'t open FIFO for reading\n");
            exit(-1);
        }
        /* Намагаємось прочитати з FIFO 14 байт у масив,
           тобто весь записаний рядок */
        size = read(fd, resstring, 14);
        if(size < 0){
            /* Якщо прочитати не змогли, повідомляємо
               про помилку і завершуємо роботу */
            printf("Can\'t read string\n");
            exit(-1);
        }
        /* Друкуємо прочитаний рядок */
        printf("%s\n", resstring);
        /* Закриваємо вхідний потік і завершуємо роботу */
        close(fd);
    }
    return 0;
}

```

### *Лістинг 3.4. Програма 03-4.c, що здійснює однонаправлений зв'язок через FIFO між процесом-батьком і процесом-нащадком*

У цій програмі інформацією між собою обмінюються процес-батько і процес-нащадок. Зверніть увагу, що повторний запуск цієї програми приведе до помилки при спробі створення **FIFO**, тому що файл із заданим іменем вже існує. Тут потрібно або вилучати його перед кожним запуском програми з диска вручну, або після першого запуску модифікувати вихідний текст, виключивши з нього все, пов'язане із системним викликом **mknod()**. Із системним викликом, призначеним для вилучення файлу при роботі процесу, ми познайомимося пізніше (тема 7) при вивченні файлових систем.

### **Завдання для самостійної роботи**

Самостійно опрацюйте матеріали теми 3 до початку практичного заняття, а також такі питання:

1. Процеси, що взаємодіють. Особливості.
2. Категорії засобів обміну інформацією.
3. Логічна організація механізму передачі інформації (встановлення зв'язку, інформаційна валентність процесів і засобів зв'язку, особливості передачі інформації за допомогою ліній зв'язку (буферизація, потік вводу/виводу і повідомлень), надійність ліній зв'язку, завершення зв'язку).

### **Завдання для лабораторної роботи**

1. Що таке потокова модель передачі даних? У яких випадках вона використовується?
2. Опишіть потокову передачу інформації між процесом і пристроєм вводу-виводу.
3. Для чого призначений файловий дескриптор? Яка його структура?
4. Як відбувається відкриття файлу? Системний виклик **open()** та його параметри для відкриття файлу.
5. Як здійснюються потокові операції читання інформації з файлу і її запису у файл?
6. За які дії відповідає системний виклик **close()**?
7. Наберіть, відкомпілюйте програму **03-1.c** і запустіть її на виконання. Зверніть увагу на використання системного виклику **umask()** з параметром 0 для того, щоб права доступу до створеного файлу точно відповідали зазначеним у системному виклику **open()**.
8. Змініть програму **03-1.c** так, щоб вона читала записану раніше у файл інформацію і друкувала її на екран. Всі зайві оператори бажано вилучити.

9. Який Ви знаєте найпростіший спосіб для передачі інформації за допомогою потокової моделі між різними процесами або навіть усередині одного процесу в операційній системі Linux?
10. У чому полягає основна відмінність **pipe** від файлу?
11. Які дії виконує системний виклик **pipe()**? Що він повертає? Чи створюються реальні файли, що відповідають файловим дескрипторам вхідного та вихідного потоків? Як здійснюється читання з **pipe** та запис в нього?
12. Наберіть програму **03-2.c**, що ілюструє роботу з **pipe** у рамках одного процесу, відкомпілюйте її і запустіть на виконання. Поясніть одержані результати.
13. Для чого використовуються **pipe**? Яка перевага від його використання у порівнянні з використанням функції копіювання з пам'яті в пам'ять всередині одного процесу?
14. Наберіть програму **03-3.c**, що здійснює однонаправлений зв'язок через **pipe** між процесом-батьком і процесом-нащадком, відкомпілюйте її і запустіть на виконання. Поясніть одержані результати.
15. Модифікуйте програму **03-3.c** для зв'язку між собою двох рідних процесів, що виконують різні програми. Поясніть одержані результати та відмінності між двома програмами.
16. Модифікуйте програму **03-3.c** для організації двостороннього зв'язку через **pipe** між процесом-батьком і процесом-нащадком, відкомпілюйте її і запустіть на виконання. Поясніть одержані результати та відмінності між двома програмами.
17. Які особливості поведінки викликів **read()** і **write()** для **pipe**?
18. Який засіб зв'язку застосовується для організації потокової взаємодії будь-яких процесів в операційній системі Linux? У чому полягає основна відмінність **FIFO** від **pipe**?
19. Які системні виклики використовуються для створення **FIFO**? Які особливості поведінки виклику **open()** при відкритті **FIFO**?
20. Наберіть програму **03-4.c**, що здійснює однонаправлений зв'язок через **FIFO** між процесом-батьком і процесом-нащадком, відкомпілюйте її і запустіть на виконання.
21. Використовуючи попередній приклад напишіть дві програми, одна із яких записує інформацію в **FIFO**, а друга – читає з нього, так щоб між ними не було яскраво виражених родинних зв'язків (тобто щоб жодна з них не була нащадком іншої).
22. Визначите розмір **pipe** для вашої операційної системи.