**Обмін даними в ОС UNIX за допомогою неіменованих та іменованих каналів.**

## 1. Загальні відомості

**1.1.** Серед усіх категорій засобів комунікації найбільш вживаними є канали зв'язку, що забезпечують досить безпечну і інформативну взаємодію процесів. Існує дві моделі передачі даних по каналах зв'язку - потік вводу-виводу і повідомлення. З них більш простою є потокова модель, у якій вся інформація в каналі зв'язку розглядається як безупинний потік байт, що не володіє ніякою внутрішньою структурою.

**1.2.** Потокова передача інформації може здійснюватися не тільки між процесами, але і між процесом і пристроєм вводу-виводу, наприклад між процесом і диском, на якому дані представляються у вигляді файлу. Оскільки системні виклики, що використовуються для потокової роботи з файлом, багато в чому відповідають системним викликам, застосовуваним для потокового спілкування процесів, ми почнемо наш розгляд саме з механізму потокового обміну між процесом і файлом.

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

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

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

**1.6.** Після завершення потокових 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. При цьому елемент таблиці відкритих файлів, що відповідає файловому дескриптору, буде позначений як вільний. За ці дії відповідає системний виклик close(). Треба відзначити, що при завершенні роботи процесу за допомогою явного чи неявного виклику функції exit() відбувається автоматичне закриття усіх відкритих потоків вводу-виводу.

**1.7.** Найбільш простим способом для передачі інформації за допомогою потокової моделі між різними процесами чи навіть всередині одного процесу в операційній системі UNIX є `pipe` (канал, труба, конвеєр).

Важлива відмінність `pip`'а від файлу полягає в тім, що прочитана інформація негайно видаляється з нього і не може бути прочитана повторно.

**1.8.** `Pipe` можна уявити собі у виді труби обмеженої ємності, розташованої всередині адресного простору операційної системи, доступ до вхідного і вихідного отвору якої здійснюється за допомогою системних викликів. У дійсності `pipe` являє собою область пам'яті, недоступну на пряму процесам користувача, найчастіше організовану у виді кільцевого буфера. При операціях читання і запису по буферу переміщаються два покажчики, що відповідають вхідному і вихідному потокам. При цьому вихідний покажчик ніколи не може перегнати вхідний і навпаки. Для створення нового екземпляру такого кільцевого буфера всередині операційної системи використовується системний виклик `pipe()`.

**1.9.** У процесі роботи системний виклик організує виділення області пам'яті під буфер і покажчики і заносить інформацію, що відповідає вхідному і вихідному потокам даних, у два елементи таблиці відкритих файлів, зв'язуючи тим самим з кожним `pip`'ом два файлових дескриптори. Для одного з них дозволена тільки операція читання з `pip`'а, а для іншого - тільки операція запису в`pipe`. Для виконання цих операцій ми можемо використовувати системні виклики `read()` і `write`. По закінченні використання вхідного чи/і вихідного потоку даних, потрібно закрити відповідний потік за допомогою системного виклику `close()` для звільнення системних ресурсів. Необхідно відзначити, що, коли всі процеси, що використовують `pipe`, закривають всі асоційовані з ним файлові дескриптори, операційна система ліквідує `pipe`. Таким чином, час існування `pip`'а в системі не може перевищувати час життя процесів, що працюють з ним.

**1.10.** Наведений вище механізм обміну інформацією через `pipe` справедливий лише для процесів, що мають загального батька, який ініціював системний виклик `pipe()`, і не може використовуватися для потокового спілкування з іншими процесами. В операційній системі `UNIX` існує можливість використання `pip`'а для взаємодії процесів (які не обов'язково мають загального батька), але її реалізація досить складна і лежить за межами нашого курсу.

**1.11.** Для організації потокової взаємодії процесів (які не обов'язково мають загального батька) в операційній системі `UNIX` застосовується засіб зв'язку, що одержав назву `FIFO` (від `First Input First Output` ) чи іменований `pipe`. `FIFO` в усьому подібний `pip`'у, за одним винятком: 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 `pip`'а на диску створюється файл спеціального типу, звертаючись до якого процеси можуть одержати потрібну їм інформацію. Для створення `FIFO` використовується системний виклик `mknod()` чи існуюча в деяких версіях `UNIX` функція `mkfifo()`.

**1.12.** Слід зазначити, що при їхній роботі не відбувається дійсного виділення області адресного простору операційної системи під іменований `pipe`, а тільки заводиться файл-мітка, існування якої дозволяє здійснити реальну організацію `FIFO` у пам'яті при його відкритті за допомогою системного виклику `open()`.

**1.13.** Після відкриття іменований `pipe` поводиться точно так само, як і неіменований. 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 `pip`'ом, не може перевищувати час життя останнього процесу, що його використовує. Коли всі процеси, що працюють з `FIFO`, закривають усі файлові дескриптори, асоційовані з ним, система звільняє ресурси, виділені під `FIFO`. Уся непрочитана інформація губиться. У той же час файл-мітка залишається на диску і може використовуватися для організації нової `FIFO`.

## 2. Синтаксис та призначення системних викликів

### 2.1. Системний виклик open.

```
#include <fcntl.h>

int open(path, flags)
        char *path;
        int flags;
int open(path, flags, mode);
        char *path;
        int flags;
        int mode;
```

Системний виклик `open` призначений для виконання операції відкриття файлу і, у випадку його вдалого здійснення, повертає файловий дескриптор відкритого файлу

Параметр path є покажчиком на рядок, що містить повне чи відносне ім'я файлу.

Параметр flags може приймати одне з наступних трьох значень:

| | |
|---|---|
| `O_RDONLY` | Відкрити тільки на читання. |
| `O_WRONLY` | Відкрити тільки на запис. |
| `O_RDWR` | Відкрити на читання/запис. |

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

| | |
|---|---|
| `O_CREAT` | Якщо файлу з зазначеним ім'ям не існує, він повинний бути створений. |
| `O_EXCL` | Застосовується разом із прапором `O_CREAT`. При спільному їхньому використанні й існуванні файлу з зазначеним ім'ям, відкриття файлу не відбувається і констатується помилкова ситуація. |
| `O_NDELAY` | Забороняє перехід процесу в стан очікування при виконанні операції відкриття і будь-яких наступних операцій над цим файлом. |
| `O_APPEND` | При відкритті файлу і перед виконанням кожної операції запису 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, часу останнього доступу до файлу і часу його останньої модифікації. |

Параметр mode встановлює атрибути прав доступу різних категорій користувачів до нового файлу при його створенні. Він обов'язковий, якщо серед заданих прапорів присутній прапор O_CREAT. Цей параметр задається як сума наступних значень:

| Тип | Oct | Hex | Призначення |
|---|---|---|---|
| `S_IRUSR` | `0400` | `0x100` | Дозволити читання власнику |
| `S_IWUSR` | `0200` | `0x080` | Дозволити запис власнику |
| `S_IRGRP` | `0040` | `0x020` | Дозволити читання групі |
| `S_IWGRP` | `0020` | `0x010` | Дозволити запис групі |

| | | | |
|---|---|---|---|
| S_IROTH | 0004 | 0x004 | Дозволити читання іншим |
| S_IWOTH | 0002 | 0x002 | Дозволити запис іншим |

При успішному завершенні результатом служить дескриптор файлу; у випадку помилки повертається -1, а змінній errno присвоюється код помилки.

**2.2. Системний виклик pipe.**

```
#include <unistd.h>

int pipe (fildes)
          int fildes [2];
```

Системний виклик `pipe` створює механізм вводу/виводу, який називається неіменованим каналом, і повертає два дескриптори файлу `fildes[0]` і `fildes[1]`. Дескриптор `fildes[0]` відкритий на читання, дескриптор `fildes[1]` – на запис.

Канал буферизує до 5120 байт даних; запис у нього більшої кількості інформації без зчитування призведе до блокування пишучого процесу. За допомогою дескриптора `fildes[0]` інформація читається в тому ж порядку, у якому вона записувалася за допомогою дескриптора `fildes[1]`.

Системний виклик `pipe` завершується невдачею, якщо виконана хоча б одна з наступних умов:

– Перевищується максимальна кількість файлів, відкритих одночасно в одному процесі.

– Переповнено системну таблицю файлів.

При успішному завершенні результат дорівнює 0; у випадку помилки повертається -1, а змінній `errno` присвоюється код помилки.

**2.3. Системні виклики dup та dup2.**

```
#include <unistd.h>

int dup (oldfd)
    int oldfd;
int dup2(oldfd, newfd)
         int oldfd;
         int newfd;
```

Системні виклики `dup` і `dup2` створюють копію файлового дескриптора `oldfd`.

Аргумент `oldfd` - це дескриптор відкритого файлу, а аргумент `newfd` - ціле число, менше константи `NOFILES`. У результаті виконання функції `dup2 newfd` стане дескриптором того ж файлу, що і `oldfd`. Якщо `newfd` вже був дескриптором відкритого файлу, він попередньо закривається.

Таким чином старий дескриптор можна використовувати замість нового і навпаки. Вони спільно блокують файл, використовують покажчики позиції файлу і прапорці. Наприклад, якщо позиція файлу змінюється за допомогою `lseek` в одному з дескрипторів, то вона змінюється також і в іншому. Проте, кожен дескриптор має свій власний прапорець "`close-on-exec`". `dup` надає новому дескриптору найменший вільний номер.

При успішному завершенні `dup` і `dup2` повертають новий дескриптор; у випадку помилки повертається -1, а змінній `errno` присвоюється код помилки.

**2.4. Системний виклик write.**

```
#include <unistd.h>

int write (fildes, buf, nbyte)
           int fildes;
           char *buf;
           unsigned nbyte;
```

Системний виклик `write` призначений для здійснення потокових операцій виводу (запису) інформації над каналами зв'язку, що описуються файловими дескрипторами, тобто для `файлів`, `pipe`, `FIFO` і `socket`.

Аргумент `fildes` - це дескриптор створеного раніше потокового каналу зв'язку, через який буде відсилатися інформація, тобто значення, що повернув один із системних викликів `creat(2)`, `open(2)`, `dup(2)`, `fcntl(2)`, `pipe(2)` чи `socket()`.

Параметр `nbytes` для системного виклику `write` визначає кількість байт, що повинний бути переданий, починаючи з адреси пам'яті `buf`.

Особливості поведінки при роботі з файлами:

При роботі з файлами інформація записується в файл, починаючи з місця, обумовленого покажчиком поточної позиції у файлі. Значення покажчика збільшується на кількість реально записаних байт.

Системний виклик `write()` має також певні особливості поведінки при роботі з `pip`'ом, пов'язані з його обмеженим розміром, затримками в передачі даних і можливістю блокування процесів, що обмінюються інформацією.

Особливості поведінки при роботі з `pipe`, `FIFO` та `socket`:

| Ситуація | Реакція |
|---|---|
| Спроба записати в канал зв'язку менше байт, чим залишилося до його заповнення. | Необхідна кількість байт записується в канал зв'язку, повертається записана кількість байт. |
| Спроба записати в канал зв'язку більше байт, чим залишилося до його заповнення. Блокування виклику дозволене. | 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 тим самим буде чекати, поки частина інформації не буде зчитана з каналу зв'язку. Повертається записана кількість байт. |
| Спроба записати в канал зв'язку більше байт, чим залишилося до його заповнення, але менше, ніж розмір буфера каналу зв'язку. Блокування виклику заборонене. | Системний виклик повертає значення -1 і встановлює змінну `errno` у значення `EAGAIN`. |
| У каналі зв'язку є місце. Спроба записати в канал зв'язку більше байт, чим залишилося до його заповнення, і більше, ніж розмір буфера каналу зв'язку. Блокування виклику заборонене. | Записується стільки байт, скільки залишилося до заповнення каналу. Системний виклик повертає кількість записаних байт. |
| Спроба запису в канал зв'язку, у якому немає місця. Блокування виклику заборонено. | Системний виклик повертає значення -1 і встановлює змінну `errno` у значення `EAGAIN`. |
| Спроба запису в канал зв'язку, з якого нікому більше читати, чи повне закриття каналу на читання під час блокування системного виклику. | Якщо виклик був заблокований, то він розблокується. Процес одержує сигнал `SIGPIPE`. Якщо цей сигнал обробляється користувачем, то системний виклик поверне значення -1 і установить перемінну errno у значення `EPIPE`. |

При успішному завершенні результат дорівнює кількості реально записаних байт; у випадку помилки повертається -1, а змінній `errno` присвоюється код помилки.

**2.5. Системний виклик read.**

```
#include <unistd.h>

int read (fildes, buf, nbyte)
          int fildes;
          char *buf;
          unsigned nbyte;
```

Системний виклик `read` призначений для здійснення потокових операцій вводу (читання) інформації над каналами зв'язку, що описуються файловими дескрипторами, тобто для `файлів`, `pipe`, `FIFO` і `socket`.

Аргумент fildes - це дескриптор створеного раніше потокового каналу зв'язку, через який буде прийматись інформація, тобто значення, що повернув один із системних викликів creat(2), open(2), dup(2), fcntl(2), pipe(2) чи socket().

Системний виклик read намагається прочитати nbyte байт із файлу, асоційованого з дескриптором fildes, у буфер, покажчиком на який є аргумент buf.

При успішному завершенні системного виклику read повертається кількість реально прочитаних байт; ця кількість може бути менша значення аргументу nbyte, якщо файл асоційований з лінією зв'язку або якщо кількість байт, що залишилися у файлі, менша значення аргументу nbyte.

Особливості поведінки при роботі з файлами:

При роботі з файлами інформація читається з файлу, починаючи з місця, обумовленого покажчиком поточної позиції у файлі. Значення покажчика збільшується на кількість реально прочитаних 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.

Системний виклик read() має певні особливості поводження при роботі з pip'ом, зв'язані з його обмеженим розміром, затримками в передачі даних і можливістю блокування процесів, що обмінюються інформацією. Організація заборони блокування системного виклику read() для pipe виходить за рамки нашого курсу.

Особливості поведінки при роботі з pipe, FIFO і socket:

| Ситуація | Реакція |
|---|---|
| Спроба прочитати менше байт, чим є в наявності в каналі зв'язку. | Читає необхідна кількість байт і повертає значення, що відповідає прочитаній кількості. Прочитана інформація видаляється з каналу зв'язку. |
| У каналі зв'язку знаходиться менше байт, чим потрібно для читання, але не нульова кількість. | Читає усе, що є в каналі зв'язку, і повертає значення, що відповідає прочитаній кількості. Прочитана інформація видаляється з каналу зв'язку. |
| Спроба читати 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 поводження виклику визначається двома попередніми рядками таблиці. Якщо в канал нема кому передати дані (немає жодного процесу, у якого цей канал зв'язку відкритий для запису), то виклик повертає значення 0. Якщо канал зв'язку цілком закривається для запису під час блокування читаючого процесу, то процес розблокується, і системний виклик повертає значення 0. |
| Спроба читати з каналу зв'язку, у якому немає інформації. Блокування виклику не дозволене. | Якщо є процеси, у яких канал зв'язку відкритий для запису, системний виклик повертає значення -1 і встановлює змінну errno у значення EAGAIN. Якщо таких процесів нема, системний виклик повертає значення 0. |

При успішному завершенні результат дорівнює кількості реально прочитаних байт; у випадку помилки повертається -1, а змінній errno присвоюється код помилки.

**2.6. Системний виклик close.**

```
#include <unistd.h>

int close (fildes)
           int fildes;
```

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

Аргумент fildes - це дескриптор файлу, отриманий після виконання системних викликів creat(2), open(2), dup(2), fcntl(2), pipe(2) або socket. Системний виклик close закриває цей дескриптор. Останній виклик close для файлу, зв'язаного з дескриптором fildes, приводить до ліквідації потоку.

При успішному завершенні результат дорівнює 0; у випадку помилки повертається -1, а змінній `errno` присвоюється код помилки.

**2.7. Системний виклик mknod.**

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

int mknod (path, mode, dev)
           char* path;
           int mode;
           int dev;
```

Системний виклик `mknod` створює новий файл із маршрутним ім'ям, на яке вказує аргумент `path`. Тип і права доступу до нового файлу визначається аргументом `mode` (побітове `OR` типу і прав доступу).

Деякі типи файлів які створюються системним викликом `mknod`:

| Тип | Oct | Hex | Призначення |
|---|---|---|---|
| S_IFIFO | 0010000 | 0x1000 | Спеціальний іменований канал. |
| S_IFCHR | 0020000 | 0x2000 | Спеціальний символьний файл. |
| S_IFBLK | 0060000 | 0x6000 | Спеціальний блочний файл. |
| S_IFREG | 0100000<br>0000000 | 0x8000<br>0x0000 | Звичайний файл. |

Права доступу трактуються як у системному виклику `open`.

Якщо `mknod` визначає спеціальний блочний чи символьний файл, то аргумент `dev` задає залежну від конфігурації системи специфікацію блочного чи символьного пристрою вводу/виводу; у іншому випадку аргумент `dev` ігнорується. Таким чином параметр `dev` є несуттєвим у нашій ситуації, і ми будемо завжди задавати його рівним 0.

При успішному завершенні системного виклику результат дорівнює 0; у випадку помилки повертається -1, а змінній errno присвоюється код помилки.

**2.8. Функція mkfifo.**

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

int mkfifo (path, mode)
            char* path;
            int mode;
```

Системний виклик `mkfifo` створює новий спеціальний FIFO файл (іменований канал) із маршрутним ім'ям, на яке вказує аргумент `path`. Права доступу до нього визначається аргументом `mode` і трактуються як у системному виклику `open` (див. лабораторну роботу №3).

При успішному завершенні результат дорівнює 0; у випадку помилки повертається -1, а змінній errno присвоюється код помилки.

**2.9. Особливості системних викликів open, read() і write() при роботі з FIFO.**

Системні виклики read() і write() при роботі з FIFO мають ті ж особливості поведінки, що і при роботі з pip'ом. Системний виклик open() при відкритті FIFO поводиться інакше, чим при відкритті інших типів файлів, що зв'язано з можливістю блокування процесів, що його викликають . Якщо FIFO відкривається тільки для читання (O_RDONLY), і прапор O_NDELAY не заданий, то процес, що здійснив системний виклик, блокується доти, поки який-небудь інший процес не відкриє FIFO на запис. Якщо прапор O_NDELAY заданий, то повертається значення файлового дескриптора, асоційованого з FIFO. Якщо FIFO відкривається тільки для запису (O_WRONLY), і прапор O_NDELAY не заданий, то процес, що здійснив системний виклик, блокується доти, поки який-небудь інший процес не відкриє FIFO на читання. Якщо прапор O_NDELAY заданий, то констатується виникнення помилки і повертається значення -1. Прапора O_NDELAY у параметрах системного виклику open() приводить і до того, що

процесу, що відкрив FIFO, забороняється блокування при виконанні наступних операцій читання з цього потоку даних і запису в нього.

2.9. Системний виклик unlink.

```
#include <unistd.h>

int unlink (path)
                char* path;
```

Системний виклик unlink видаляє елемент каталогу, заданий маршрутним ім'ям, на яке вказує аргумент path.

Коли всі посилання на файл вилучені і немає процесу, для якого цей файл є відкритим, unlink знищує файл. В іншому випадку знищення файлу відкладається до моменту закриття його всіма процесами.

При успішному завершенні результат дорівнює 0; у випадку помилки повертається -1, а змінній errno присвоюється код помилки.