Операційні системи. Практикум

3. Організація взаємодії процесів через pipe і FIFO в UNIX

План заняття

Поняття про потік вводу-виводу

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

Існує дві моделі передачі даних по каналах зв'язку -- потік вводу-виводу і повідомлення. З них простішою є потокова модель, у якій операції передачі/прийому інформації взагалі не цікавляться вмістом того, що передається або приймається. Вся інформація в каналі зв'язку розглядається як неперервний потік байт, що не володіють ніякою внутрішньою структурою. Вивченню механізмів, що забезпечують потокову передачу даних в операційній системі UNIX, і буде присвячений цей семінар.

Поняття про роботу з файлами через системні виклики і стандартну бібліотеку вводу-виводу для мови С

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. Оскільки поняття файлу повинне бути знайомим тим хто вивчає цей курс, а 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.

Як ми сподіваємося, з курсу програмування мовою С вам відомі функції роботи з файлами зі стандартної бібліотеки вводу-висновку, такі як fopen(), fread(), fwrite(), fprintf(), fscanf(), fgets() і т.д. Ці функції входять як невід'ємна частина в стандарт ANSI на мову С 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() використається для введення з файлу послідовності символів, що закінчується символом '\n' -- переведення каретки. Функція fscanf() робить ввід інформації, що відповідає заданому формату, і т.д. З погляду потокової 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 UNIX 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 ми коротко познайомимося із системними викликами `open()`, `read()`, `write()` і `close()`, які застосовуються для такого обміну, але спочатку нам потрібно ввести ще одне поняття -- поняття файлового дескриптора.

Файловий дескриптор

На лекції 2 ми говорили, що 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-висновку. 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, ми будемо розглядати пізніше, при вивченні організації файлових систем в UNIX (семінари 11-12 і 13-14).

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, ми повинні 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 і, 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 іменм 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, часів 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 відпповідний 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 термінала.

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 поводження 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 прапорів на поточному рівні знань 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. Як вам відомо з матеріалів семінарів 1-2, 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 потрібного 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 ми **вимагаємо**, 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.

Докладніше про операції відкриття файлу і її місця серед набору всіх файлових операцій буде розповідатися на лекції 7 "Файлова система з точки зору користувача". Роботу системного виклику open() із прапорцями O_APPEND і O_TRUNC ми розберемо на семінарах 11-12, присвячених організації файлових систем в UNIX.

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 описуваними 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 одержуватись інформація, тобто значенням, яке повернув один із системних викликів `open()`, `pipe()` або `socket()`.

Параметр `addr` -- адреса області пам'яті, починаючи з якої буде братися 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, що ми хочемо одержати з каналу зв'язку і розмістити в пам'яті, починаючи з адреси `addr`.

Значення, що повертаються

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 менше, ніж воно, в 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 негативне значення.

Особливості поводження при роботі з файлами

При роботі з файлами інформація записується у файл або читається з файлу, починаючи з місця, яке изначається 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.

Ми зараз не акцентуємо увагу на понятті покажчика поточної позиції у файлі і взаємному впливі значення цього покажчика та поведінки системних викликів. Це питання розглянемо на семінарах 11-12.

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, і 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 (див. семінар 3-4) 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 повертаються

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

Програми для запису інформації у файл

Для ілюстрації розглянутого вище давайте розглянемо наступну програму:

```
/*Програма 03-1.c, що ілюструє використання системних викликів
open(), write() і close() 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 нашого масиву, тобто ввесь
    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

Наберіть, відкомпілюйте цю програму 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()`.

Написання, компіляція і запуск програми для читання інформації з файлу

Змініть програму з попереднього розділу так, щоб вона читала записану раніше у файл інформацію і друкувала її на екран. Всі зайві оператори бажано видалити.

Поняття про pipe. Системний виклик pipe()

Найбільш простим способом для передачі інформації за допомогою потокової моделі між різними процесами або навіть усередині 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.

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 користувальницьким процесам напряму, найчастіше організована у вигляді кільцевого буфера (хоча існують і інші види організації). По буфері при операціях читання і запису переміщаються 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 усередині 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 pip'а 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 pip'а, 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 повертаються

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 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 використати ті ж самі системні виклики read() і write(), що й при роботі з файлами. Природно, по закінченні використання вхідного або/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.

Програма для pipe в одному процесі

Досить яскравою ілюстрацією дій по створенню pip'а, запису в нього даних, читанню з нього і звільненню виділених ресурсів може служити програма, що організує роботу з pip'ом у рамках одного процесу, наведена нижче:

```
/* Програма 03-2.c, що ілюструє роботу з pip'ом у рамках одного процесу
*/
#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 pip'a 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 pip'ом у рамках одного процесу

Наберіть програму, відкомпілюйте її і запустіть на виконання.

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

Зрозуміло, що якби вся перевага pip'ів зводилася б до заміни функції копіювання з пам'яті в пам'ять усередині 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 предкака, що створив `pipe`.

Програма для організації однонапрямленого зв'язку між рідними процесами через pipe

Давайте розглянемо програму, що здійснює однонапрямлений зв'язок між процесом-батьком і процесом-дитиною:

```
/* Програма 03-3.c, що здійснює однонапрямлений зв'язок через pipe
між процесом-батьком і процесом-дитиною */
#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-дитині. 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, на цьому
        батько припиняє роботу */
        close(fd[1]);
        printf("Parent exit\n");
    } else {
        /* Ми 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, и може його використовувати.
        У цьому процесі вхідний потік даних нам не буде потрібний,
        тому закриваємо його.*/
        close(fd[1]);
        /* Намагаємось прочитати з pip'а 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 однонапрямлений зв'язок через ріре між процесом-батьком і процесом-дитиною

Наберіть програму, відкомпілюйте її і запустіть на виконання.

Завдання підвищеної складності: модифікуйте цей приклад для зв'язку між собою двох рідних процесів, що виконують різні програми.

Написання, компіляція і запуск програми для організації двонаправленого зв'язку між родинними процесами через ріре

`Pipe` служить для організації однонапряямленого або симплексного зв'язку. Якби в попередньому прикладі ми спробували організувати через ріре двосторонній зв'язок, коли процес-батько пише інформацію в ріре, припускаючи, що її одержить процес-дитина, а потім читає інформацію з рір'а, припускаючи, що її записав породжений процес, то могла б виникнути ситуація, у якій процес-батько прочитав би власну інформацію, а процес-дитина не одержала б нічого. Для використання одного рір'а у двох напрямках необхідні спеціальні засоби синхронізації процесів, про які мова йде в лекціях "Алгоритми синхронізації" (лекція 5) і "Механізми синхронізації" (лекція 6). Простіший спосіб організації двонаправленого зв'язку між родинними процесами полягає у використанні двох ріре. Модифікуйте програму з попереднього приклада (розділ "Програма для організації односпрямованого зв'язку між родинними процесами через ріре") для організації такого двостороннього зв'язку, відкомпілюйте її і запустіть на виконання.

Необхідно відзначити, що в деяких UNIX-подібних системах (наприклад, в Solaris2) реалізовані повністю дуплексні рір’и [27]. У таких системах для обох файлових дескрипторів, асоційованих з рір'ом, дозволені і операція читання, і операція запису. Однак таке поводження не характерне для рір'ів і не переноситься на всі системи.

Особливості поведінки викликів read() і write() для рір'а

Системні виклики `read()` і `write()` мають певні особливості виконання при роботі з рір'ом, 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 рір’а може прочитатися менше інформації, ніж ви намагалися записати, і за один раз в** `pipe` **може 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 рір'а. 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 рір'а, у процесі, що буде використовувати** ріре **для читання (**`close(fd[1]`**)** у процесі-дитині в програмі з розділу "Програма для організації однонапрямленого зв'язку між родинними процесами через ріре"). Аналогічною особливістю виконання при відсутності процесів, у яких ріре відкритий для читання, володіє і системний виклик `write()`, із чим зв'язана **необхідність закриття** файлового дескриптора, **асоційованого з вихідним кінцем рір'а, 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 в наявності в каналі зв'язку.	Читає необхідну кількість байт і повертає значення, що відповідає прочитаній кількості. Прочитана інформація видаляється 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 видаляється 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 повертає значення 0.
Спроба прочитати 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 немає, системний виклик повертає значення 0

Системний виклик write

Ситуація	Поведінка
Спроба записати в канал зв'язку менше байт, ніж залишилося до його заповнення.	Необхідна кількість байт поміщається в канал зв'язку, повертається записана кількість байт.
Спроба записати в канал зв'язку більше байт, ніж 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 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	Записується стільки байт, скільки залишилося до заповнення каналу. Системний виклик повертає кількість

канал зв'язку більше байт, чим залишилося до його заповнення, і більше, ніж розмір буфера каналу зв'язку. Блокування виклику заборонене.	записаних байт.
Спроба запису в канал зв'язку, у якому немає місця. Блокування виклику не дозволене.	Системний виклик повертає значення -1 і встановлює змінну `errno` у значення `EAGAIN`.
Спроба запису в канал зв'язку, з якого нікому більше читати, або 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 встановить змінну `errno` у значення `EPIPE`.

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

Завдання підвищеної складності: визначите розмір `pipe` для вашої операційної системи.

Поняття FIFO. Використання системного виклику mknod() для створення FIFO. Функція mkfifo()

Як ми вже з'ясували, доступ до інформації про розташування рір'а 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 прабатька, що ініціював системний виклик `pipe()`, або для таких процесів і самого прабатька і не може використовуватися для потокового спілкування з іншими процесами. В операційній системі UNIX існує можливість використання рір'а для взаємодії інших процесів, але її реалізація досить складна і лежить далеко за межами наших занять.

Для організації потокової взаємодії будь-яких процесів в операційній системі UNIX застосовується засіб зв'язку, що одержав назву FIFO (від First Input First Output) або іменований `pipe`. FIFO у всьому подібний рір'у, за одним виключенням: 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 рір'а на диску заводиться 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()`.

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 вже відомого нам ситемного виклику** `open()`**.**

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 рір'ом, не може перевищувати час життя останнього з його процесів, що 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. У той же час 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

Нашою метою не є повний опис системного виклику `mknod`, а тільки опис його використання для створення FIFO. Тому ми 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 повертаються

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:

- 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 повертаються

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 іменований `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!

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 pip'ом. Системний виклик `open()` при відкритті FIFO також поводиться трохи інакше, ніж при відкритті інших типів файлів, що пов'язане з можливістю блокування виконуючих його процесів. 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 на запис. 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 на читання. Якщо прапорець `O_NDELAY` заданий, то констатується виникнення помилки і повертається значення -1. Завдання 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.

Програма с FIFO у рідних процесах

Для ілюстрації взаємодії процесів через FIFO розглянемо таку програму:

```
/* Програма 03-4.c, що здійснює однонапрямлений зв'язок через
FIFO між процесом-батьком і процесом-дитиною */
#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) {
        /* Ми перебуваємо в батьківському процесі, що буде
        передавати інформацію процесу-дитині. 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 {
        /* Ми 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 однонапрямлений зв'язок через FIFO між процесом-батьком і процесом-дитиною

Наберіть програму, відкомпілюйте її і запустіть на виконання. У цій програмі інформацією між собою обмінюються процес-батько і процес-дитина. 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 видаляти його перед кожним прогоном 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 видалення файлу при роботі процесу, ми познайомимося пізніше (на семінарах 11-12) при вивченні файлових систем.

Написання, компіляція і запуск програми з FIFO у неспоріднених процесах

Для закріплення отриманих знань напишіть на базі попереднього приклада дві програми, одна із яких пише 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).

Непрацюючий приклад для зв'язку процесів на різних комп'ютерах

Якщо у вас є можливість, знайдіть два комп'ютери, що мають розділювану файлову систему (наприклад, змонтовану за допомогою NFS), і запустіть на них програми з попереднього розділу так, щоб кожна програма працювала на своєму комп'ютері, а FIFO створювався на розділюваній файловій системі. Хоча обидва процеси бачать той самий файл із типом FIFO, взаємодії між ними не відбувається, тому що вони функціонують у фізично різних адресних просторах і намагаються відкрити FIFO усередині різних операційних систем.