Національна академія управління

МЕТОДИЧНІ ВКАЗІВКИ

щодо виконання практичних занять

з курсу

“ОПЕРАЦІЙНІ СИСТЕМИ”

Частина ІІ

КИЇВ 2009

Системне програмування і операційні системи

Методичні вказівки щодо виконання практичних занять. Частина ІІ

Укладачі:

к.т.н., доцент **Баклан Ігор Всеволодович**

Степанкова Ганна Анатоліївна

Затверджено на засіданні Ради факультету комп'ютерних наук. Протокол №7 від 29 серпня 2008 р.

ЗМІСТ

ЗАНЯТТЯ №4.

ОРГАНІЗАЦІЯ ВЗАЄМОДІЇ ПРОЦЕСІВ ЧЕРЕЗ PIPE І FIFO У UNIX.....5

Поняття про потік введення-виведення 5
Поняття про роботу з файлами через системні виклики та стандартну бібліотеку введення-виведення для мови С 5
Файловий дескриптор.....6
Відкриття файлу. Системний виклик open().....7
Системні виклики read(), write(), close().....10
Прогін програми для запису інформації до файла.....12
Поняття про pipe. Системний виклик pipe().....14
Прогін програми для pipe в одному процесі.....15
Організація зв'язку через pipe між процесом-батьком і процесом-нащадком. Спадкування файлових дескрипторів при викликах fork() і exec().....17
Прогін програми для організації односпрямованого зв'язку між родинними процесами через pipe 17
Особливості поведінки викликів read() і write() для pipe.....20
Поняття FIFO. Використання системного виклику mknod() для створення FIFO. Функція mkfifo().....23
Особливості поведінки виклику open() при відкритті FIFO.....26
Прогін програми с FIFO у родинних процесах.....27

ЗАНЯТТЯ №5,6,7 (6 ГОДИН) 30

ОРГАНІЗАЦІЯ РОБОТИ З ПОДІЛЮВАНОЮ ПАМ'ЯТТЮ В UNIX. ПОНЯТТЯ НИТОК ВИКОНАННЯ (THREAD). СЕМАФОРИ В UNIX У ЯКОСТІ ЗАСОБУ СИНХРОНІЗАЦІЇ ПРОЦЕСІВ. ЧЕРГИ ПОВІДОМЛЕНЬ У UNIX.....30

Переваги і недоліки потокового обміну даними. 31
Поняття про System V IPC 31
Простір імен. Адресація в System V IPC. Функція ftok() 32
Дескриптори System V IPC 34
Поділювана пам'ять у UNIX. Системні виклики shmget(), shmat(), shmdt() 35
Прогін програм з використанням поділюваної пам'яті.....39
Команди ipcs і ipcrm 45
Використання системного виклику shmctl() для звільнення ресурсу 47

Поділювана пам'ять і системні виклики fork(), exec() і функція exit() 48
Поняття про нитки виконання (thread) у UNIX. Ідентифікатор нитки виконання. Функція pthread_self() 48
Створення та завершення thread'а. Функції pthread_create(), pthread_exit(), pthread_join() 50
Прогін програми з використанням двох ниток виконання 53
Необхідність синхронізації процесів і ниток виконання, які використовують загальну пам'ять 55
Семафори в UNIX. Відмінність операцій над UNIX-семафорами від класичних операцій 62
Створення масиву семафорів або доступ до вже існуючого. Системний виклик semget() 63
Виконання операцій над семафорами. Системний виклик semop() 65
Прогін приклада з використанням семафора 67
Зміна попереднього приклада 71
Видалення набору семафорів із системи за допомогою команди ipcrm або системного виклику semctl() 71
Поняття про POSIX-семафори 73
Повідомлення у якості засобу зв'язку та засобу синхронізації процесів 73
Черги повідомлень у UNIX у якості складової частини System V IPC 73
Створення черги повідомлень або доступ до вже існуючої. Системний виклик msgget() 74
Реалізація примітивів send і receive. Системні виклики msgsnd() і msgrcv() 76
Видалення черги повідомлень із системи за допомогою команди ipcrm або системного виклику msgctl() 81
Прогін приклада з односпрямованою передачею текстової інформації 82
Модифікація попереднього приклада для передачі числової інформації 87
Поняття мультиплексування. Мультиплексування повідомлень. Модель взаємодії процесів клієнт-сервер. Нерівноправність клієнта та сервера 88
Використання черг повідомлень для синхронізації роботи процесів 89

ЗАНЯТТЯ №4

ОРГАНІЗАЦІЯ ВЗАЄМОДІЇ ПРОЦЕСІВ ЧЕРЕЗ PIPE І FIFO У UNIX

Поняття потоку введення-виведення. Уявлення про роботу з файлами через системні виклики та стандартну бібліотеку введення-виведення. Поняття файлового дескриптора. Відкриття файлу. Системний виклик open(). Системні виклики close(), read(), write(). Поняття pipe. Системний виклик pipe(). Організація зв'язку через pipe між процесом-батьком і процесом-нащадком. Спадкування файлових дескрипторів при викликах fork() і exec(). Особливості поведінки викликів read() і write() для pip'a. Поняття FIFO. Використання системного виклику mknod() для створення FIFO. Функція mkfifo(). Особливості поведінки виклику open() при відкритті FIFO.

Поняття про потік введення-виведення

Серед усіх категорій засобів комунікації найбільш вживаними є канали зв'язку, які забезпечують достатньо безпечну та достатньо інформативну взаємодію процесів.

Існує дві моделі передачі даних за каналами зв'язку – *потік введення-виведення* та *повідомлення*. З них найбільш простою є потокова модель, в якій операції передавання-приймання інформації взагалі не цікавляться вмістом того, що передається або приймається. Вся інформація до каналуі зв'язку розглядається як безперервний потік байтів, який не має ніякої внутрішньої структури. Вивченню механізмів, які забезпечують потокову передачу даних в операційній системі UNIX, і буде присвячений це практичне заняття.

Поняття про роботу з файлами через системні виклики та стандартну бібліотеку введення-виведення для мови С

Потокова передача інформації може здійснюватися не тільки між процесами, але й між процесом і пристроєм введення-виведення, наприклад між процесом і диском, на якому дані подані у вигляді файлу. Оскільки поняття файлу повинне бути знайомо читачу, а системні виклики, які використовуються для потокової роботи з файлом, багато в чому відповідають системним викликам, які застосовуються 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 стандартною бібліотекою введення-виведення, такі як `fopen()`, `fread()`, `fwrite()`, `fprintf()`, `fscanf()`, `fgets()` і т.ін. Ці функції - невід'ємна частина в стандарті ANSI для мови С і дозволяють програмісту отримувати інформацію з файлу або записувати її до файлу за умови, що програміст має визначені знання про вміст переданих даних. Так, наприклад, функція `fgets()` використовується для введення з файлу послідовності символів, які закінчується символом `'\n'` – переведення каретки. Функція `fscanf()` робить введення інформації, яка відповідає заданому формату, і т.ін. З погляду потокової моделі операції, зумовлені функціями стандартної бібліотеки введення-виведення, не є потоковими операціями, тому що кожна з них вимагає наявності деякої структури переданих даних.

В операційній системі UNIX ці функції утворюють певну надбудову – сервісний інтерфейс – над системними викликами, які здійснюють прямі потокові операції обміну інформацією між процесом і файлом й не потребуючих 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 використовувані процесом файли, входить до складу його системного контексту та зберігається в його блоці керування – PCB. В операційній системі UNIX можна спрощено уявляти, що інформація про файли, з якими процес здійснює операції потокового обміну, поряд з інформацією про потокові лінії зв'язку, які з'єднують процес з іншими процесами та пристроями введення-виведення, зберігається в деякому масиві, який і отримав назву *таблиці відкритих файлів* або *таблиці файлових дескрипторів*. Індекс елемента цього масиву, який відповідає визначеному потокові введення-виведення, отримав назву *файлового дескриптора* для цього потоку. Таким чином, файловий дескриптор є невеликим цілим не негативним числом, яке для поточного процесу в даний момент часу однозначно визначає деякий діючий канал введення-виведення. Деякі файлові дескриптори на етапі старту будь-якої програми асоціюються зі стандартними потоками введення-виведення. Так, наприклад, файловий дескриптор 0 відповідає стандартному потоку введення, файловий дескриптор 1 – стандартному потоку виведення, файловий дескриптор 2

– стандартному потоку для виведення помилок. У нормальному інтерактивному режимі роботи стандартний потік введення пов'язує процес із клавіатурою, а стандартні потоки виведення і виведення помилок – з поточним терміналом.

Більш детальна побудова структур даних, які містять інформацію про потоки введення-виведення, асоційованих із процесом, ми будемо розглядати пізніше, при вивченні організації файлових систем у UNIX (семінари 10–11 і 12–13).

Відкриття файлу. Системний виклик open()

Файловий дескриптор використовується як параметр, який описує потік введення-виведення, для системних викликів, які виконують операції над цим потоком. Тому перш ніж робити операції читання даних з файлу та запису їх у файл, ми повинні помістити інформацію про файл до таблиці відкритих файлів і визначити відповідний файловий дескриптор. Для цього застосовується процедура відкриття файлу, здійснювана 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 та, у випадку її вдалого здійснення, повертає файловий дескриптор відкритого файлу (невелике ненегативне ціле число, яке використовується надалі для інших операцій з цим файлом).

Параметр path є покажчиком 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 только операції запису;
- O_RDWR – якщо над файлом будуть здійснюватися й операції читання, і операції запису.

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

- O_CREAT – якщо файлу із зазначеним ім'ям не існує, він повинний бути створений;
- O_EXCL – застосовується разом із прапором O_CREAT. При спільному їхньому використанні й існуванні файлу із зазначеним ім'ям, відкриття файлу не робиться та констатується помилкова ситуація;
- O_NDELAY – забороняє переведення процесу до стану очікування при виконанні операції відкриття та будь-яких наступних операцій над цим файлом;
- O_APPEND – при відкритті файлу та 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, із збереженням існуючих атрибутів файлу, крім часу останнього доступу до файлу та 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 до стану очікування) доти, поки записана інформація не буде фізично поміщена на відповідний нижчий рівень hardware;
- O_NOCTTY – якщо ім'я файлу відноситься до термінального пристрою, він 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 прапорів присутній прапорець O_CREAT, і може бути опущений у противному випадку. Цей параметр задається як сума наступних восьмеричних значень:

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

створив файл;

- 0040 – дозволене читання для групи користувача, який створив файл;
- 0020 – дозволений запис для групи користувача, який створив файл;
- 0010 – дозволене виконання для групи користувача, який 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 встановлюваного права доступу виходять із стандартної комбінації параметра mode і маски створення файлів поточного процесу umask, а саме – вони дорівнюють 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, який здійснив системний виклик, блокується доти, поки будь-який 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, який здійснив системний виклик, блокується до тих пір, поки будь-який інший процес не відкриє FIFO на читання. Якщо прапорець O_NDELAY заданий, то констатується виникнення помилки та повертається значення -1.

Значення, що повертається

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

Системний виклик open() використовує набір прапорів для того, щоб специфицировать операції, які передбачається застосовувати до файлу надалі або які повинні бути виконані безпосередньо в момент відкриття файлу. З усього можливого набору прапорів на поточному рівні знань нас будуть цікавити тільки прапори O_RDONLY,

O_WRONLY, O_RDWR, O_CREAT і O_EXCL. Перші три прапори є взаємовиключними: хоча б один з них повинний бути застосований та наявність одного з них не припускає наявності двох інших. Ці прапори описують набір операцій, які при успішному відкритті файлу будуть дозволені над файлом надалі: тільки читання, тільки запис, читання та запис. Як вам відомо з матеріалів семінару 1, у кожного файлу існують атрибути прав доступу для різних категорій користувачів. Якщо файл із заданим ім'ям існує на диску, і права доступу до нього для користувача, від імені якого працює поточний процес, не суперечать запитаному набору операцій, то операційна система сканує таблицю відкритих файлів від її початку до кінця в пошуках першого вільного елемента, заповнює його та 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 відсутнім, і хочемо, щоб він був створений, прапорець для набору операцій повинний використовуватися в комбінації з прапором O_CREAT. Якщо файл існує, то усе відбувається за розглянутим вище сценарієм. Якщо файлу немає, спочатку виконується створення файлу з набором прав, зазначеним у параметрах системного виклику. Перевірка відповідності набору операцій оголошеним правам доступу може й не вироблятися (як, наприклад, у Linux).

У випадку, коли ми **вимагаємо**, щоб файл на диску був відсутній та був створений у момент відкриття, прапорець для набору операцій повинний використовуватися в комбінації з прапорцями O_CREAT і O_EXCL.

Докладніше про операції відкриття файлу і її місці серед набору усіх файлових операцій буде розповідатися на лекції «Файлова система з погляду користувача». Роботу системного виклику open() із прапорцями O_APPEND і O_TRUNC ми розберемо на семінарах 10–11, присвячених організації файлових систем у UNIX.

Системні виклики read(), write(), close()

Для здійснення потокових операцій читання інформації з файлу та її запису до файлу 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 виведення (запису) інформації над каналами зв'язку, 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 виходити інформація, тобто значенням, яке повернув один із системних викликів `open()`, `pipe()` або `socket()`.

Параметр `addr` являє собою адреса області пам'яті, починаючи з якого буде братися інформація для передачі або розміщатися прийнята інформація.

Параметр `nbytes` для системного виклику `write` визначає кількість байт, що повинний бути переданий, починаючи з адреси пам'яті `addr`. Параметр `nbytes` для системного виклику `read` визначає кількість байт, що ми хочемо отримати з каналу зв'язку і розмістити в пам'яті, починаючи з адреси `addr`.

Значення, які повертаються

У випадку успішного завершення системний виклик повертає кількість реально надісланих або прийнятих байтів. Зауважимо, що це значення (більшим або дорівнюючим `0`) може не збігатися із заданим значенням параметра `nbytes`, а бути менше, ніж воно, у силу відсутності місця на диску або в лінії зв'язку при передаванні даних або відсутності інформації при її прийманні. При виникненні будь-якої помилки повертається негативне значення.

Особливості поведінки при роботі з файлами

При роботі з файлами інформація записується у файл або читається з файлу, починаючи з місця, зумовленого покажчиком поточної позиції у файлі. Значення покажчика збільшується на кількість реально прочитаних або записаних байтів. При читанні інформації з файлу вона не зникає з нього. Якщо системний виклик `read` повертає значення `0`, то це говорить про те, що файл прочитаний до кінця.

Ми зараз не акцентуємо увагу на понятті покажчика поточної позиції у файлі і взаємному впливі значення цього покажчика і поведінки системних викликів. Це питання буде обговорюватися надалі на семінарах 10–11.

Після завершення потокових операцій процес повинен виконати операцію закриття потоку введення-виведення, під час якої відбудеться остаточне скидання буферів на лінії зв'язку, звільняться виділені ресурси операційної системи, і елемент таблиці відкритих файлів, який відповідає файловому дескрипторові, буде відзначений як вільний. За ці дії відповідає системний виклик close(). Треба зауважити, що при завершенні роботи процесу (див. семінар 2–3) за допомогою явного або неявного виклику функції exit() відбувається автоматичне закриття усіх відкритих потоків введення-виведення.

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 й іншими об'єктами введення-виведення, які описуються в операційній системі через файлові дескриптори: pipe, FIFO, socket.

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

Значення, які повертаються

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

Прогін програми для запису інформації до файла

Для ілюстрації сказаного давайте розглянемо наступну програму:

```
/*Програма 05-1.c, яка ілюструє використання
системних викликів
open(), write() і close() для запису інформації до
файла */
#include <sys/types.h>
#include <fcntl.h>
```

```
#include <stdio.h>
int main(){
    int fd;
    size_t size;
    char string[] = "Hello, world!";
     /* Обнуляємо маску створення файлів поточного
процесу для того, щоб права доступу створюваного
файла точно відповідали параметру виклику open()
*/
    (void)umask(0);
     /* Спробуємо відкрити файл з ім'ям myfile у
поточній директорії тільки для операцій виведення.
Якщо файлу не існує, спробуємо його створити з
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 та припиняємо роботу */
        printf("Can\'t open file\n");
        exit(-1);
    }
     /* Пробуємо записати до файла 14 байтів із
нашого масиву, тобто весь рядок "Hello, world!"
разом з ознакою кінця рядка */
    size = write(fd, string, 14);
    if(size != 14){
           /* Якщо була записаною 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 4.1. Програма 04-1.c, яка ілюструє використання системних викликів open(), write() і close() для запису інформації до файла

ЗАВДАННЯ 1: Наберіть, відкомпілюйте цю програму та 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 та запуск програми для читання інформації з файлу

ЗАВДАННЯ 2: Змініть програму з попереднього розділу таким чином, щоб вона читала записану раніше до файла інформацію і друкувала неї на екрані. Усі 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 всередині одного процесу в операційній системі UNIX є `pipe` (канал, труба, конвеєр).

Важлива відмінність рір'а від файлу полягає в тому, що прочитана інформація негайно зникає з нього та не може бути прочитаною другий раз.

`Pipe` можна уявити собі у вигляді труби обмеженої ємності, розташованої всередині адресного простору операційної системи, доступ до вхідного та вихідного отвору якої здійснюється за допомогою системних викликів. У дійсності `pipe` - це область пам'яті, недоступна користувацьким процесам прямо, найчастіше організована у вигляді кільцевого буфера (хоча існують й інші види організації). За буфером при операціях читання та запису переміщуються два покажчики, які відповідають вхідному та вихідному потокам. При цьому вихідний покажчик ніколи не може перегнати вхідний та 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 всередині операційної системи.

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

Значення, які повертаються

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

У процесі роботи системний виклик організує виділення області пам'яті під буфер і покажчики та заносить інформацію, яка відповідає вхідному та вихідному потокам даних, у два елементи таблиці відкритих файлів, пов'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 до pipe. Для виконання цих операцій ми можемо використовувати ті ж самі системні виклики read() і write(), що і при роботі з файлами. Звичайно, по закінченні використання вхідного або вихідного потоку даних, потрібно закрити відповідний потік за допомогою системного виклику close() для звільнення системних ресурсів. Треба зауважити, 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, які працюють з ним.

Прогін програми для pipe в одному процесі

Достатньо яркою ілюстрацією дій по створенню pipe, запису в нього даних, читанню з нього і звільненню виділених ресурсів може служити програма, що організує роботу з рір'ом у рамках одного процесу, приведена нижче:

```
/* Програма 04-2.c, яка ілюструє роботу з pipe 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 та припиняємо роботу */
        printf("Can\'t create pipe\n");
        exit(-1);
    }
    /* Пробуємо 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 байтів,
повідомляємо про помилку */
        printf("Can\'t write all string\n");
        exit(-1);
    }
    /* Пробуємо прочитати з pipe 14 байт до іншого
масиву, 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;
}
```

Листинг 4.2. Програма 04-2.c, що ілюструє роботу з pip'ом у рамках одного процесу

ЗАВДАННЯ 3: Наберіть програму, відкомпілюйте її та запустіть на виконання.

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

Зрозуміло, що якби все достоїнство `pipe` зводилося до заміни функції копіювання з пам'яті до пам'яті всередині одного процесу на пересилання інформації через операційну систему, то овчинка не коштувала би вироблення. Однак `таблиця відкритих файлів` успадковується процесом-дитиною 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, був спеціальними засобами виставлена ознака, яка спонукує операційну систему закрити їх при виконанні `exec()`, однак їх розгляд виходить за межі нашого курсу). Ця обставина дозволяє організувати передачу інформації через `pipe` між родинними процесами, які мають загального прабатька, що створив `pipe`.

Прогін програми для організації односпрямованого зв'язку між родинними процесами через pipe

Давайте розглянемо програму, яка здійснює односпрямований зв'язок між процесом-батьком і процесом-дитиною:

```
/* Програма 04-3.c, що здійснює односпрямований
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 удалося, друкуємо
про це повідомлення та припиняємо роботу */
```

```
        printf("Can\'t create pipe\n");
        exit(-1);
    }
    /* Породжуємо новий процес */
    result = fork();
    if(result){
            /* Якщо створити процес не удалося,
сповіщаємо про це та завершуємо роботу */
        printf("Can\'t fork child\n");
        exit(-1);
    } else if (result > 0) {
        /* Ми знаходимося в батьківському процесі,
який 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
байтів, повідомляємо про помилку та 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 знаходимося в породженому процесі,
який буде отримувати інформацію від процесу-
батька. Він успадкував від батька таблицю
відкритих файлів і, знаючи файлові дескриптори,
які відповідають pipe, може його використовувати.
У цьому процесі вхідний потік даних нам не
знадобится, тому закриваємо його.*/
        close(fd[1]);
        /* Пробуємо прочитати з pipe 14 байтів до
масива, тобто весь записаний рядок */
        size = read(fd[0], resstring, 14);
```

```
        if(size < 0){

                        /* Якщо прочитати не змогли,
повідомляємо про помилку та 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;
}
```

Листинг 4.3. Програма 04-3.c, що здійснює односпрямований зв'язок через pipe між процесом-батьком і процесом-дитиною

ЗАВДАННЯ 4: Наберіть програму, відкомпілюйте її і запустите на виконання.

ЗАВДАННЯ 5: Модифікуйте цей приклад для зв'язку між собою двох родинних процесів, які виконують різні програми.

Написання, компіляція та запуск програми для організації двунаправленного зв'язку між родинними процесами через pipe

`Pipe` служить для організації односпрямованого 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 отримає процес-дитина, 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-дитина не отримала б нічого. Для використання одного `pipe` в двох напрямках необхідні спеціальні засоби синхронізації процесів, про які мова йде в лекціях «Алгоритми синхронізації» (лекція 5) і «Механізми синхронізації» (лекція 6). Більш простий спосіб організації двунаправленного зв'язку між родинними процесами полягає у використанні двох *pipe*.

ЗАВДАННЯ 6: Модифікуйте програму з попереднього приклада (розділ «Прогін програми для організації односпрямованого зв'язку між родинними процесами через pipe») для організації такого двостороннього зв'язку, відкомпілюйте її та запустите на виконання.

Необхідно зауважити, що в деяких UNIX-подібних системах (наприклад, у Solaris2) реалізовані цілком дуплексні `pipe`. У таких системах для обох `файлових дескрипторів`, асоційованих з `pipe`, дозволені й операція читання, й операція запису. Однак така поведінка не характерна для `pipe`.

Особливості поведінки викликів read() і write() для pipe

Системні виклики `read()` і `write()` мають певні особливості поведінки при роботі з `pipe`, зв'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 межі нашого курсу.

Будьте уважні при написанні програм, які обмінюються великими обсягами інформації через `pipe`. Пом'ятайте, що за один раз з pip'a може прочитатися менше інформації, чим ви запитували, і за один раз у `pipe` може записатися менше інформації, чим вам хотілося б. Перевіряйте значення, які повертаються викликами!

Одна з особливостей поведінки заблокованого системного виклику `read()` пов'язана зі спробою читання з порожнього pip*e*. Якщо є процеси, у яких цей `pipe` відкритий для запису, то системний виклик блокується nf чекає появи інформації. Якщо таких процесів немає, він поверне значення `0` без блокування процесу. Ця особливість призводить до **необхідності закриття** `файлового дескриптора`, **асоційованого з вхідним кінцем `pipe`, у процесі, який буде використовувати** `pipe` **для читання (**`close(fd[1]`**)** у процесі-дитині в програмі з розділу «Прогін програми для організації односпрямованого зв'язку між родинними процесами через `pipe`»). Аналогічною особливістю поведінки при відсутності процесів, у яких `pipe` відкритий для читання, володіє й системний виклик `write()`, з чим пов'язана **необхідність закриття** `файлового дескриптора`, **асоційованого з вихідним кінцем pip'a, у процесі, які буде використовувати** `pipe` **для запису (**`close(fd[0]`**)** у процесі-батьку в тій же програмі).

Системні виклики read і write (продовження)

Особливості поведінки при роботі з pipe, FIFO і socket	
Системний виклик read	
Ситуація	**Поведінка**
Спроба прочитати менше байтів, чим є в наявності до каналуі зв'язку.	Читає необхідну кількість байтів і повертає значення, яке відповідає прочитаній кількості. Прочитана інформація вилучається з каналу зв'язку.
У каналі зв'язку знаходиться менше байтів, чим викликано, але не нульову кількість.	Читає усе, що є до каналуі зв'язку, і повертає значення, що відповідає прочитаній кількості. Прочитана інформація вилучається 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 до каналу зв'язку та поки існує процес, який може передати до нього інформацію. Якщо інформація з'явилася, то процес розблоковується, та поведінка виклику визначається двома попередніми рядками таблиці. Якщо до каналу комусь 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, та системний виклик повертає значення `0`.
Спроба читати з каналу зв'язку, в 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	**Поведінки**
Спроба записати до	Необхідна кількість байт міститься до

каналу зв'язку менше байтів, чим залишилося до його заповнення.	каналу зв'язку, повертається записана кількість байт.
Спроба записати до каналу зв'язку більше байтів, чим залишилося до його заповнення. Блокування виклику дозволене.	Виклик блокується доти, поки всі дані не будуть поміщені до каналу 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 до каналу зв'язку більше байтів, 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 до каналу зв'язку більше байтів, чим залишилося до його заповнення, і більше, ніж розмір буфера каналу зв'язку. Блокування виклику заборонене.	Записується стільки байтів, скільки залишилося до заповнення каналу. Системний виклик повертає кількість записаних байт.
Спроба запису до каналу зв'язку, у якому немає місця. Блокування виклику не дозволене.	Системний виклик повертає значення `-1` і встановлює змінну `errno` у значення `EAGAIN`.
Спроба запису до каналу зв'язку, з якого більше читали, або повне закриття каналу на читання під час блокування системного виклику.	Якщо виклик був заблокований, то він розблокується. Процес отримує сигнал `SIGPIPE`. Якщо цей сигнал обробляється користувачем, то системний виклик поверне значення `-1` і установить змінну `errno` у значення `EPIPE`.

Необхідно зауважити додаткову особливість системного виклику `write` при роботі з pipe і FIFO. Запис інформації, розмір якої не перевищує розмір буфера, повинна здійснюватися атомарно – одним підряд лежачим шматком. Цим зумовлний ряд блокувань і помилок у попередньому переліку.

ЗАВДАННЯ 7: **визначите розмір pipe для вашої операційної системи.**

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

Як ми з'ясували, доступ до інформації про розташування *pipe* в операційній системі та його стани може бути здійснений тільки через `таблицю відкритих файлів процесу`, який створив `pipe`, і через успадковані від нього `таблиці відкритих файлів` процесів-нащадків. Тому викладений вище механізм обміну інформацією через `pipe` справедливий лише для родинних процесів, які мають загального прабатька, який ініціювали системний виклик `pipe()`, або для таких процесів і самого прабатька та не може використовуватися для потокового спілкування з іншими процесами. В операційній системі UNIX існує можливість використання `pipe` для взаємодії інших процесів, але її реалізація досить складна та лежить далеко за межами наших занять.

Для організації потокової взаємодії будь-яких процесів в операційній системі UNIX застосовується засіб зв'язку, який отримав назву FIFO (від First Input First Output) або іменований `pipe`. `FIFO` в усьому подібний `pipe`, за одним виключенням: дані про розташування `FIFO` в адресному просторі ядра і його стані процеси можуть отримувати не через родинні зв'язки, а через файлову систему. Для цього при створенні іменованого `pipe` на диску заводиться файл спеціального типу, звертаючи до якого процеси можуть отримати цікавлячу їх інформацію. Для створення `FIFO` використовується системний виклик `mknod()` або існуюча в деяких версіях UNIX функція `mkfifo()`.

Зауважимо, що при їх роботі не відбувається реального 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 системного виклику** `open()`**.**

Після відкриття іменований pipe поводиться точно так само, як і нейменований. 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 його процесів, що використовували. 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 та може далі використовуватися для нової реальної організації FIFO.

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

Наша мета - не повний опис системного виклику mknod, а тільки опис його використання для створення FIFO. Тому ми будемо розглядати не всі можливі варіанти завдання параметрів, а тільки ті з них, які відповідають цієї специфічної діяльності.

Параметр dev є несуттєвим у нашій ситуації, і ми будемо завжди задавати його рівним 0.

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

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 повинний створити FIFO, і деякої суми наступних вісімкових значень:

- 0400 – дозволене читання для користувача, який створив FIFO;
- 0200 – дозволений запис для користувача, який створив FIFO;
- 0040 – дозволене читання для групи користувача, який створив FIFO;
- 0020 – дозволений запис для групи користувача, який створив FIFO;

- 0004 – дозволене читання для всіх інших користувачів;
- 0002 – дозволений запис для всіх інших користувачів.

При створенні FIFO реально встановлюваного права доступу виходять із стандартної комбінації параметра `mode` і маски створення файлів поточного процесу `umask`, а саме – вони дорівнюють `(0777 & mode) & ~umask`.

Значення, які повертаються

При успішному створенні FIFO системний виклик повертає значення `0`, при неуспішному – негативне 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, який має повне або відносне ім'я файлу, що буде міткою FIFO на диску. Для успішного створення FIFO файлу з таким ім'ям перед викликом функції не повинне існувати.

Параметр `mode` встановлює атрибути прав доступу різних категорій користувачів до FIFO. Цей параметр задається як деяка сума наступних восьмеричних значень:

1. 0400 – дозволене читання для користувача, який створив FIFO;
2. 0200 – дозволений запис для користувача, який створив FIFO;
3. 0040 – дозволене читання для групи користувача, який створив FIFO;
4. 0020 – дозволений запис для групи користувача, який створив FIFO;
5. 0004 – дозволене читання для всіх інших користувачів;
6. 0002 – дозволений запис для всіх інших

користувачів.

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

Значення, які повертаються

При успішному створенні FIFO функція повертає значення 0, при неуспішному – негативне значення.

Важливо розуміти, що файл типу FIFO не служить для розміщення на диску інформації, який 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 глибокого 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 pipe. Системний виклик open() при відкритті FIFO також поводиться трохи інакше, чим при відкритті інших типів файлів, що зв'язано з можливістю блокування виконуючих його процесів. Якщо FIFO відкривається тільки для читання, і прапорець O_NDELAY не заданий, то процес, який здійснив системний виклик, блокується доти, поки будь-який 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, що здійснив системний виклик, блокується доти, поки будь-який інший процес не відкриє FIFO на читання. Якщо прапорець O_NDELAY заданий, то констатується виникнення помилки та повертається значення –1. Завдання прапора O_NDELAY у параметрах системного виклику open() призводить і до того, що процесу, який відкрив FIFO, забороняється блокування при виконанні наступних операцій читання з цього потоку даних і запису до нього.

Прогін програми с FIFO у родинних процесах

Для ілюстрації взаємодії процесів через FIFO розглянемо таку програму:

```
/* Програма 04-4.c, яка здійснює односпрямований зв'язок за
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";
     /* Обнуляємо 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 ім'ям aaa.fifo у поточної директорії */
    if(mknod(name, S_IFIFO | 0666, 0) < 0){
         /* Якщо створити FIFO не удалося, друкуємо про цьому повідомлення і припиняємо роботу */
        printf("Can\'t create FIFO\n");
        exit(-1);
    }
    /* Породжуємо новий процес */
    if((result = fork()) < 0){
              /* Якщо створити процес не удалося, сповіщаємо про це і завершуємо роботу */
        printf("Can\'t fork child\n");
        exit(-1);
    } else if (result > 0) {
         /* Ми знаходимося в батьківському процесі, який 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 цьому повідомлення та припиняємо
роботу */
                    printf("Can\'t open FIFO for
writing\n");
            exit(-1);
        }
           /* Пробуємо записати до FIFO 14 байт,
тобто весь рядок "Hello, world!" разом з ознакою
кінця рядку */
        size = write(fd, "Hello, world!", 14);
        if(size != 14){
          /* Якщо записалася менша кількість байт,
то повідомляємо про помилку та завершуємо роботу
*/
                printf("Can\'t write all string to
FIFO\n");
            exit(-1);
        }
          /* Закриваємо вхідний потік даних та на
цьому батько припиняє роботу */
        close(fd);
        printf("Parent exit\n");
    } else {
          /* Ми знаходимося в породженому процесі,
який буде отримувати інформацію від процесу-
батька. Відкриваємо FIFO на читання.*/
        if((fd = open(name, O_RDONLY)) < 0){
                /* Якщо відкрити FIFO не вдалося,
друкуємо про цьому повідомлення та припиняємо
роботу */
                    printf("Can\'t open FIFO for
reading\n");
            exit(-1);
        }
          /* Пробуємо прочитати з FIFO 14 байт до
масива, тобто весь записаний рядок */
        size = read(fd, resstring, 14);
        if(size < 0){
                    /* Якщо прочитати не змогли,
повідомляємо про помилку та 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;
}
```

Листинг 4.4. Програма 04-4.c, яка здійснює односпрямований зв'язок через FIFO між процесом-батьком і процесом-дитиною

<u>ЗАВДАННЯ 8:</u> Наберіть програму, відкомпілюйте її та запустите на виконання. У цій програмі інформацією між собою обмінюються процес-батько та процес-дитина. Зверніть увагу, що повторний запуск цієї програми приведе до помилки при спробі створення FIFO, тому що файл із заданим ім'ям 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, позв'язане із системним викликом mknod(). Із системним викликом, призначеним для видалення файлу при роботі процесу, ми познайомимося пізніше (на семінарах 10–11) при вивченні файлових систем.

Написання, компіляція і запуск програми з FIFO у неспоріднених процесах

<u>ЗАВДАННЯ 9,10:</u> Для закріплення отриманих знань напишіть на базі попереднього приклада дві програми, одна з 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).

ЗАНЯТТЯ №5,6,7 (6 ГОДИН)

ОРГАНІЗАЦІЯ РОБОТИ З ПОДІЛЮВАНОЮ ПАМ'ЯТТЮ В UNIX. ПОНЯТТЯ НИТОК ВИКОНАННЯ (THREAD). СЕМАФОРИ В UNIX У ЯКОСТІ ЗАСОБУ СИНХРОНІЗАЦІЇ ПРОЦЕСІВ. ЧЕРГИ ПОВІДОМЛЕНЬ У UNIX.

Переваги і недоліки потокового обміну даними. Поняття System V IPC. Простір імен. Адресація в System V IPC. Функція ftok(). Дескриптори System V IPC. Поділювана пам'ять у UNIX. Системні виклики shmget(), shmat(), shmdt(). Команди ipc і ipcrm. Використання системного виклику shmctl() для звільнення ресурсу. Поділювана пам'ять і системні виклики fork(), exec() і функція exit(). Поняття про нитці виконання (thread) у UNIX. Ідентифікатор нитки виконання. Функція pthread_self(). Створення і завершення thread'a. Функції pthread_create(), pthread_exit(), pthread_join(). Необхідність синхронізації процесів і ниток виконання, що використовують загальну пам'ять.

Семафори в UNIX. Відмінність операцій над UNIX-семафорами від класичних операцій. Створення масиву семафорів або доступ до вже існуючого масиву. Системний виклик semget(). Виконання операцій над семафорами. Системний виклик semop(). Видалення набору семафорів із системи за допомогою команди ipcrm або системного виклику semctl(). Поняття про POSIX-семафори.

Повідомлення у якості засобу зв'язку і засобу синхронізації процесів. Черги повідомлень у UNIX У якості складова частина System V IPC. Створення черги повідомлень або доступ до вже існуючій. Системний виклик msgget(). Реалізація примітивів send і receive. Системні виклики msgsnd() і msgrcv(). Видалення черги повідомлень із системи за допомогою команди ipcrm або системного виклику msgctl(). Поняття мультиплексування. Мультиплексирование повідомлень. Модель взаємодії процесів сервер^-сервер-клієнт-сервер. Нерівноправність клієнта і сервера. *Використання черг повідомлень для синхронізації роботи процесів.*

Переваги і недоліки потокового обміну даними.

На попередньому занятті ми познайомилися з механізмами, які забезпечують потокову передачу даних між процесами в операційній системі UNIX, а саме з pip'ами та FIFO. Потокові механізми досить прості в реалізації таі зручні для використання, але мають ряд істотних недоліків:

- Операції читання та запису не аналізують вміст переданих даних. Процес, який прочитав 20 байтів із потоку, не може сказати, чи були вони записані одним процесом або декількома, чи записувалися вони за один раз або було, наприклад, виконано 4 операції запису по 5 байт. Дані в потоці ніУ якості не інтерпретуються системою. Якщо потрібно будь-яка інтерпретація даних, то передавальний і приймаючий процеси повинні заздалегідь узгодити свої дії та вміти здійснювати її самостійно.
- Для передачі інформації від одного процесу до іншого потрібно, У якості мінімум, дві операції копіювання даних: перший раз – з адресного простору передавального процесу до системного буфера, другий раз – із системного буфера до адресного простору приймаючого процесу.
- Процеси, які обмінюються інформацією, повинні одночасно існувати в обчислювальній системі. Не можна записати інформацію до потоку за допомогою одного процесу, завершити його, а потім, через якийсь час, запустити інший процес і прочитати записану інформацію.

Поняття про System V IPC

Наведені вище недоліки потоків даних привели до розробки інших механізмів передачі інформації між процесами. Частина цих механізмів,

яка вперше з'явилися в UNIX System V і згодом перекочували відтіля практично до усіх сучасних версій операційної системи UNIX, отримала загальну назву System V IPC (IPC – скорочення від interprocess communications). У групу System V IPC входять: черги повідомлень, поділювана пам'ять і семафори. Ці засоби організації взаємодії процесів пов'язані не тільки спільністю походження, але і мають схожий інтерфейс для виконання подібних операцій, наприклад, для виграділення та звільнення відповідного ресурсу в системі. Ми будемо розглядати їх у порядку від менш семантично навантажених з погляду операційної системи до більш семантично навантажених. Інше кажучи, чим пізніше ми почнемо займатися будь-яким механізмом з `System V IPC`, тим більше дій щодо інтерпретації переданої інформації прийдеться виконувати операційній системі при використанні цього механізму.

Простір імен. Адресація в System V IPC. Функція ftok()

Усі засоби зв'язку з System V IPC, У якості і вже розглянуті нами ріре і FIFO, є засобами зв'язку з непрямою адресацією. У якості ми встановили на попередньому занятті, для організації взаємодії неспоріднених процесів за допомогою засобу зв'язку з непрямою адресацією необхідно, щоб цей засіб зв'язку мав ім'я. Відсутність імен у рір'ов дозволяє процесам отримувати інформацію про розташування рір'а в системі та його стани тільки через родинні зв'язки. Наявність асоційованого імені в FIFO – імені спеціалізованого файлу у файловій системі – дозволяє неспорідненим процесам отримувати цю інформацію через інтерфейс файлової системи.

Безліч усіх можливих імен для об'єктів будь-якого вигляду прийнято називати простором імен відповідного вигляду об'єктів. Для FIFO `простором імен` є безліч усіх припустимих імен файлів у файловій системі. Для всіх об'єктів з `System V IPC` таким `простором імен` є безліч значень деякого цілочисельного типу даних – `key_t` – ключа. Причому програмісту не дозволено прямо привласнювати значення ключа, це значення задається опосередковано: через комбінацію імені будь-якого файлу, який вже існує у файловій системі, та невеликого цілого числа – наприклад, номера екземпляра засобу зв'язку.

Такий хитрий спосіб отримання значення ключа зв'язаний із двома розуміннями:

- Якщо дозволити програмістам самим привласнювати значення ключа для ідентифікації засобів зв'язку, то не виключено, що два програмісти випадково скористаються однаковим значенням, не підозрюючи про це. Тоді їх процеси будуть несанкціоновано

взаємодіяти через той самий засіб комунікації, який може привести до нестандартного поводження цих процесів. Тому основним компонентом значення ключа є перетворене до числового значення повне ім'я деякого файлу, доступ до якого на читання дозволений процесу. Кожний програміст має можливість використовувати для цієї мети свій специфічний файл, файл, який наприклад виконується, зв'язаний з одним із взаємодіючих процесів. Слід зазначити, що перетворення з текстового імені файлу до числа ґрунтується на розташуванні зазначеного файлу на твердому диску або іншому фізичному носії. Тому для утворення ключа варто застосовувати файли, які не змінюють свого положення протягом часу організації взаємодії процесів;

- Другий компонент значення ключа використовується для того, щоб дозволити програмісту пов'язати з тим самим ім'ям файлу більш за один екземпляр кожного засобу зв'язку. У якості такого компонента можна задавати порядковий номер відповідного екземпляра.

Отримання значення ключа з двох компонентів здійснюється функцією `ftok()`.

Функція для генерації ключа System V IPC

Прототип функції

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(char *pathname, char proj);
```

Опис функції

Функція `ftok` служить для перетворення імені існуючого файла та невеликого цілого числа, наприклад, порядкового номера екземпляру засобів зв'язку, до ключа System V IPC.

Параметр `pathname` повинний бути покажчиком на ім'я існуючого файлу, доступного для процесу, який викликає функцію.

Параметр `proj` – це невелике ціле число, яке характеризує екземпляр засобу зв'язку.

У випадку неможливості генерації ключа функція повертає негативне значення, у противному випадку вона повертає значення сгенерованого ключа. Тип даних `key_t` за звичаєм є 32-бітовим цілим.

Ще раз підкреслимо три важливих моменти, пов'язаних з використанням імені файлу для отримання ключа. По-перше, необхідно вказувати ім'я файлу, яке **вже існує** у файловій системі та для якого

процес **має право доступу на читання** (не плутайте з завданням імені файлу при створенні FIFO, де вказувалося ім'я **для знову створюваного** спеціального файлу). По-друге, зазначений файл повинний **зберігати своє положення на диску** до тих пір, поки всі процеси, які беруть участь у взаємодії, не отримають `ключ System V IPC`. По-третє, завдання імені файлу, У якості одного з компонентів для отримання ключа, ні в якому разі не означає, що інформація, передана за допомогою асоційованого засобу зв'язку, буде розташовуватися в цьому файлі. Інформація буде зберігатися **всередині адресного простору операційної системи**, а задане ім'я файлу лише дозволяє різним процесам згенерувати ідентичні ключі.

Дескриптори System V IPC

Ми говорили (див. попереднє заняття розділ «Файловий дескриптор»), що інформацію про потоки введення-виведення, з якими має справу поточний процес, зокрема про pip'ах і FIFO, операційна система зберігає в таблиці відкритих файлів процесу. Системні виклики, які здійснюють операції над потоком, використовують у якості параметра індексу елемента таблиці відкритих файлів, який відповідає потокові, – файловий дескриптор. Використання файлових дескрипторів для ідентифікації потоків усередині процесу дозволяє застосовувати до них вже існуючий інтерфейс для роботи з файлами, але в той же час приводить до автоматичного закриття потоків при завершенні процесу. Цим, зокрема, зумовлений один з перерахованих вище недоліків потокової передачі інформації.

При реалізації компонентів `System V IPC` була прийнята інша концепція. Ядро операційної системи зберігає інформацію про всі засоби `System V IPC`, які використовуються в системі, поза контекстом користувацьких процесів. При створенні нового засобу зв'язку або отриманні доступу до вже існуючий процес отримує ненегативне ціле число – дескриптор (ідентифікатор) **цього засобу зв'язку**, який однозначно ідентифікує його у всій обчислювальній системі. Цей `дескриптор` повинний передаватися у якості параметра всім системним викликам, які здійснюють подальші операції над відповідним засобом `System V IPC`.

Подібна концепція дозволяє усунути один із самих істотних недоліків, який властивий потоковим засобам зв'язку – вимога одночасного існування взаємодіючих процесів, але в той же час вимагає підвищеної обережності для того, щоб процес, який отримує інформацію, не прийняв замість нових старі дані, випадково залишені в механізмі комунікації.

Поділювана пам'ять у UNIX. Системні виклики shmget(), shmat(), shmdt()

З погляду операційної системи, найменш семантично навантаженим засобом System V IPC є поділювана пам'ять (shared memory). Ми вже згадували про цю категорію засобів зв'язку на лекції. Для поточного заняття нам досить знати, що операційна система може дозволити декільком процесам спільно використовувати деяку область адресного простору. Внутрішні механізми, які дозволяють реалізувати таке використання, будуть докладно розглянуті на лекції, присвяченій сегментній, сторінковій і сегментно-сторінковій організації пам'яті.

Усі засоби зв'язку `System V IPC` вимагають попередніх ініціюючих дій (створення) для організації взаємодії процесів.

Для створення області `поділюваної пам'яті` з визначеним ключем або доступу за ключем до вже існуючої області застосовується системний виклик `shmget()`. Існує два варіанти його використання для створення нової області `поділюваної пам'яті`.

- *Стандартний спосіб*. У якості значення ключа системному викликові ставиться значення, сформоване функцією `ftok()` для деякого імені файлу та номера екземпляра області `поділюваної пам'яті`. У якості прапорців ставиться комбінація прав доступу до створюваного сегмента та прапорця `IPC_CREAT`. Якщо сегмент для даного ключа ще не існує, то система буде намагатися створити його із зазначеними правами доступу. Якщо ж раптом він вже існував, то ми просто отримаємо його дескриптор. Можливе додавання до цієї комбінації прапорців прапорця `IPC_EXCL`. Цей прапорець гарантує нормальне завершення системного виклику тільки в тому випадку, якщо сегмент дійсно був створений (тобто раніше він не існував), якщо ж сегмент існував, то системний виклик завершиться з помилкою, та значення системної змінної `errno`, описаної у файлі `errno.h`, буде встановлене в `EEXIST`.
- *Нестандартний спосіб*. У якості значення ключа вказується спеціальне значення `IPC_PRIVATE`. Використання значення `IPC_PRIVATE` завжди приводить до спроби створення нового сегмента `поділюваної пам'яті` з заданими правами доступу і з ключем, який не збігається зі значенням ключа з жодного вже існуючих сегментів і який не може бути отриманий за допомогою функції `ftok()` ні при одній комбінації її параметрів. Наявність прапорців `IPC_CREAT` і `IPC_EXCL` у цьому випадку ігнорується.

Системний виклик shmget()

Прототип системного виклику

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size,
           int shmflg);
```

Опис системного виклику

Системний виклик `shmget` призначений для виконання операції доступу до сегмента поділюваної пам'яті і, у випадку його успішного завершення, повертає дескриптор System V IPC для цього сегмента (ціле ненегативне число, яке однозначно характеризує сегмент усередині обчислювальної системи та використовується надалі для інших операцій з ним).

Параметр `key` є ключем System V IPC для сегмента, тобто фактично його ім'ям із простору імен System V IPC. У якості значення цього параметра може використовуватися значення ключа, отримане за допомогою функції `ftok()`, або спеціальне значення `IPC_PRIVATE`. Використання значення `IPC_PRIVATE` завжди приводить до спроби створення нового сегмента поділюваної пам'яті з ключем, який не збігається зі значенням ключа з жодного вже існуючих сегментів і який не може бути отриманий за допомогою функції `ftok()` ні при одній комбінації її параметрів.

Параметр `size` визначає розмір створюваного або вже існуючого сегмента в байтах. Якщо сегмент із зазначеним ключем вже існує, але його розмір не збігається з вказаним у параметрі `size`, констатується виникнення помилки.

Параметр `shmflg` – прапорці – відіграє роль тільки при створенні нового сегмента поділюваної пам'яті та визначає права різних користувачів при доступі до сегмента, а також необхідність створення нового сегмента та поводження системного виклику при спробі створення. Він є деякою комбінацією (за допомогою операції побітове або – «|») наступних визначених значень і вісімкових прав доступу:

- `IPC_CREAT` – якщо сегмента для зазначеного ключа не існує, він повинний бути створений;
- `IPC_EXCL` – застосовується разом із прапорцем `IPC_CREAT`. При спільному їхньому використанні й

існуванні сегмента із зазначеним ключем, доступ до сегмента не створюється і констатується помилкова ситуація, при цьому змінна errno, описана у файлі <errno.h>, приймає значення EEXIST;

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

Значення, яке повертається

Системний виклик повертає значення дескриптора System V IPC для сегмента поділюваної пам'яті при нормальному завершенні та значення -1 при виникненні помилки.

Доступ до створеної області поділюваної пам'яті надалі забезпечується її дескриптором, який поверне системний виклик shmget(). Доступ до вже існуючої області також може здійснюватися двома способами:

- Якщо ми знаємо її ключ, то, використовуючи виклик shmget(), можемо отримати її дескриптор. У цьому випадку не можна вказувати у якості складеної частини прапорців прапорець IPC_EXCL, а значення ключа, природно, не може бути IPC_PRIVATE. Права доступу ігноруються, а розмір області повинний збігатися з розміром, зазначеним при її створенні.
- Або ми можемо скористатися тим, що дескриптор System V IPC дійсний у межах всієї операційної системи, і передати його значення від процесу, який створив поділювана пам'ять чинному процесу. Зауважимо, що при створенні поділюваної пам'яті за допомогою значення IPC_PRIVATE – це єдино можливий спосіб.

Після отримання дескриптора необхідно включити область поділюваної пам'яті в адресний простір поточного процесу. Це здійснюється за допомогою системного виклику shmat(). При

нормальному завершенні він поверне адресу `поділюваної пам'яті` в адресному просторі поточного процесу. Подальший доступ до цієї пам'яті здійснюється за допомогою звичайних засобів мови програмування.

Системний виклик shmat()

Прототип системного виклику

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
char *shmat(int shmid, char *shmaddr,
            int shmflg);
```

Опис системного виклику

Системний виклик `shmat` призначений для включення області поділюваної пам'яті до адресного простору поточного процесу. Даний опис не є повним описом системного виклику, а обмежується межами цього курсу. Для повного опису зверніться до UNIX Manual.

Параметр `shmid` є дескриптором System V IPC для сегмента поділюваної пам'яті, тобто значенням, що повернув системний виклик `shmget()` при створенні сегмента або при його пошуку за ключем.

У якості параметр `shmaddr` у межах нашого курсу ми завжди будемо передавати значення `NULL`, дозволяючи операційній системі самій розмістити поділювану пам'ять до адресного простору нашого процесу.

Параметр `shmflg` у нашому курсі може приймати тільки два значення: `0` – для здійснення операцій читання та записи над сегментом і `SHM_RDONLY` – якщо ми хочемо тільки читати з нього. При цьому процес повинний мати відповідні права доступу до сегмента.

Значення, яке повертається

Системний виклик повертає адресу сегмента поділюваної пам'яті в адресному просторі процесу при нормальному завершенні та значення `-1` при виникненні помилки.

Після закінчення використання `поділюваної пам'яті` процес може зменшити розмір свого адресного простору, виключивши з нього цю область за допомогою системного виклику `shmdt()`. Відзначимо, що у якості параметра системного виклика `shmdt()` вимагає адреса початку області `поділюваної пам'яті` до адресного простору процесу, тобто значення, яке повернув системний виклик `shmat()`, тому дане значення варто зберігати протягом усього часу використання `поділюваної пам'яті`.

Системний виклик shmdt()

Прототип системного виклику

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt(char *shmaddr);
```

Опис системного виклику

Системний виклик `shmdt` призначений для виключення області поділюваної пам'яті з адресного простору поточного процесу.

Параметр `shmaddr` є адресою сегмента поділюваної пам'яті, тобто значенням, яке повернув системний виклик `shmat()`.

Значення, яке повертається

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

Прогін програм з використанням поділюваної пам'яті

Для ілюстрації використання `поділюваної пам'яті` давайте розглянемо дві взаємодіючі програми.

```
/* Програма 1 (05-1a.c) для ілюстрації роботи з
поділюваною пам'яттю */
/* Ми організуємо поділювану пам'ять для масиву з
трьох цілих чисел.
Перший елемент масиву є лічильником числа запусків
програми 1,
т. е. даної програми, другий елемент масиву –
лічильником числа запусків
програми 2, третій елемент масиву – лічильником
числа запусків обох
програм */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <errno.h>
int main()
{
      int  *array;       /* Покажчик на поділювану
пам'ять */
```

```
    int shmid;      /* IPC дескриптор для області
поділюваної пам'яті */
     int new = 1;     /* Прапорець необхідності
ініціалізації елементів масиву */
    char pathname[] = "05-1a.c"; /* Ім'я файлу,
яке
       використовується для генерації ключа. Файл
із таким і'ям повинний існувати в поточній
директорії */
    key_t key;     /* IPC ключ */
    /* Генеруємо IPC ключ з імені файлу 05-1a.c у
         поточної директорії та номера екземпляра
області
       поділюваної пам'яті 0 */
    if((key = ftok(pathname,0)) < 0){
        printf("Can\'t generate key\n");
        exit(-1);
    }
    /* Намагаємося ексклюзивно створити поділювану
пам'ять для сгенерированного ключа, тобто якщо для
цього ключа вона вже існує, системний виклик
поверне негативне        значення. Розмір пам'яті
визначаємо у якості розмір масиву з трьох цілих
змінних, права доступу 0666 – читання та запис
дозволу для всіх */
    if((shmid = shmget(key, 3*sizeof(int),
    0666|IPC_CREAT|IPC_EXCL)) < 0){
    /* У випадку помилки намагаємося визначити: чи
виникла вона через те, що сегмент поділюваної
пам'яті вже існує
    або з іншої причини */
        if(errno != EEXIST){
             /* Якщо з іншої причини – припиняємо
роботу */
                  printf("Can\'t  create  shared
memory\n");
           exit(-1);
        } else {
                /* Якщо через те, що поділювана
пам'ять уже
           існує, те намагаємося отримати її IPC
```

```
                дескриптор і, у випадку удачі,
скидаємо прапорець необхідності ініціалізації
елементів масиву */

            if((shmid = shmget(key, 3*sizeof(int),
0)) < 0){
                printf("Can\'t find shared memory\
n");
                exit(-1);
            }
            new = 0;
        }
    }
     /* Намагаємося відобразити поділювану пам'ять
до адресного простору поточного процесу. Зверніть
увагу на те, що для правильного порівняння ми явно
перетворюємо значення -1 до покажчика на
ціле.*/

     if((array = (int *)shmat(shmid, NULL, 0)) ==
(int *)(-1)){
        printf("Can't attach shared memory\n");
        exit(-1);
    }
    /* У залежності від значення прапорця new або
ініціюємо масив, або збільшуємо відповідні
лічильники */
    if(new){
        array[0] = 1;
        array[1] = 0;
        array[2] = 1;
    } else {
        array[0] += 1;
        array[2] += 1;
    }
        /* Друкуємо нові значення лічильників,
видаляємо
    поділювану пам'ять з адресного простору
    поточного процесу та завершуємо роботу */
    printf("Program 1 was spawn %d times,
        program 2 - %d times, total - %d times\n",
        array[0], array[1], array[2]);
    if(shmdt(array) < 0){
```

```
        printf("Can't detach shared memory\n");
        exit(-1);
    }
    return 0;
}
```

Лістінг 5.1а. Програма 1 (05-1a.c) для ілюстрації роботи з поділюваною пам'яттю

```
/* Програма 2 (05-1b.c) для ілюстрації роботи з
поділюваною пам'яттю */
/* Ми організуємо поділювану пам'ять для масиву з
трьох цілих чисел. Перший елемент масиву є
лічильником числа запусків програми 1, тобто даної
програми, другий елемент масиву – лічильником
числа
запусків програми 2, третій елемент масиву –
лічильником числа запусків обох програм */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <errno.h>
int main()
{
      int *array;     /* Покажчик на поділювану
пам'ять */
     int shmid;     /* IPC дескриптор для області
поділюваної пам'яті */
      int new = 1;    /* Прапорець необхідності
ініціалізації елементів масиву */
     char pathname[] = "05-1a.c"; /* Ім'я файлу,
яке
        використовується для генерації ключа. Файл
із таким ім'ям повинний існувати в поточній
директорії */
    key_t key;    /* IPC ключ */
    /* Генеруємо IPC ключ з імені файлу 05-1a.c у
       поточної директорії і номера екземпляра
області
    поділюваної пам'яті 0 */
    if((key = ftok(pathname,0)) < 0){
        printf("Can\'t generate key\n");
        exit(-1);
```

```
    }
    /* Намагаємося ексклюзивно створити поділювану
пам'ять
    для сгенерованого ключа, тобто якщо для цього
    ключа вона вже існує, системний виклик поверне
    негативне значення. Розмір пам'яті визначаємо
     у якості розмір масиву з трьох цілих змінних,
права
     доступу 0666 – читання та запис дозволені для
всіх */
    if((shmid = shmget(key, 3*sizeof(int),
        0666|IPC_CREAT|IPC_EXCL)) < 0){
      /* У випадку виникнення помилки намагаємося
визначити:
        чи виникла вона через те, що сегмент
поділюваної
    пам'яті вже існує або з іншої причини */
        if(errno != EEXIST){
             /* Якщо з іншої причини – припиняємо
роботу */
                    printf("Can\'t create shared
memory\n");
            exit(-1);
        } else {
                /* Якщо через те, що поділювана
пам'ять уже
             існує, те намагаємося отримати її IPC
дескриптор і, у випадку удачі, скидаємо прапорець
необхідності ініціалізації елементів масиву */
            if((shmid = shmget(key, 3*sizeof(int),
0)) < 0){
                printf("Can\'t find shared memory\
n");
                exit(-1);
            }
            new = 0;
        }
    }
     /* Намагаємося відобразити поділювану пам'ять
до адресного простору поточного процесу. Зверніть
увагу на те, що для правильного порівняння ми явно
перетворюємо значення -1 до покажчика на
ціле.*/
```

```
    if((array = (int *)shmat(shmid, NULL, 0)) ==
        (int *)(-1)){
        printf("Can't attach shared memory\n");
        exit(-1);
    }
    /* У залежності від значення прапорця new або
    ініціюємо масив, або збільшуємо
    відповідні лічильники */
    if(new){
        array[0] = 0;
        array[1] = 1;
        array[2] = 1;
    } else {
        array[1] += 1;
        array[2] += 1;
    }
        /* Друкуємо нові значення лічильників,
видаляємо поділювану пам'ять з адресного простору
поточного процесу і завершуємо роботу */
    printf("Program 1 was spawn %d times,
        program 2 - %d times, total - %d times\n",
    array[0], array[1], array[2]);
    if(shmdt(array) < 0){
        printf("Can't detach shared memory\n");
        exit(-1);
    }
    return 0;
}
```

Лістінг 5.1b. Програма 2 (05-1b.c) для ілюстрації роботи з поділюваною пам'яттю

Ці програми дуже схожі одна на одну та використовують поділювану пам'ять для збереження числа запусків кожної з програм і їхньої суми. У `поділюваній пам'яті` розміщається масив із трьох цілих чисел. Перший елемент масиву використовується У якості лічильника для програми 1, другий елемент – для програми 2, третій елемент – для обох програм сумарно. Додатковий нюанс у програмах виникає через необхідність ініціалізації елементів масиву при створенні `поділюваної пам'яті`. Для цього нам потрібно, щоб програми могли розрізнювати випадок, коли вони створили її, і випадок, коли вона вже існувала. Ми добиваємося розходження, використовуючи спочатку системний виклик `shmget()` із прапорцями `IPC_CREAT` і `IPC_EXCL`. Якщо виклик завершується нормально, то ми створили

поділювану пам'ять. Якщо виклик завершується з констатацією помилки та значення змінної errno дорівнює EEXIST, то, виходить, поділювана пам'ять вже існує, і ми можемо отримати її IPC дескриптор, застосовуючи той же самий виклик з нульовим значенням прапорців.

ЗАВДАННЯ 1: Наберіть програми, збережете під іменами 05-1a.c і 05-1b.c відповідно, відкомпілюйте їх і запустіть кілька разів. Проаналізуйте отримані результати.

Команди ipcs і ipcrm

Як ми бачили з попереднього приклада, створена область поділюваної пам'яті зберігається в операційній системі навіть тоді, коли немає жодного процесу, який включає її до свого адресного простору. З одного боку, це має визначені переваги, оскільки не вимагає одночасного існування взаємодіючих процесів, з іншого боку, може заподіювати істотні незручності. Допустимо, що попередні програми ми хочемо використовувати таким чином, щоб підраховувати кількість запусків протягом одного поточного сеансу роботи в системі. Однак у створеному сегменті поділюваної пам'яті залишається інформація від попереднього сеансу, і програми будуть видавати загальну кількість запусків за увесь час роботи з моменту завантаження операційної системи. Можна було б створювати для нового сеансу новий сегмент поділюваної пам'яті, але кількість ресурсів у системі не безмежна. Нас рятує те, що існують способи видаляти невикористовувані ресурси System V IPC як за допомогою команд операційної системи, так і за допомогою системних викликів. Усі засоби System V IPC вимагають визначених дій для звільнення займаних ресурсів після закінчення взаємодії процесів. Для того щоб видаляти ресурси System V IPC з командного рядка, нам стануть в нагоді дві команди, ipcs і ipcrm.

Команда ipcs видає інформацію про всі засоби System V IPC, які існують у системі, для яких користувач має права на читання: областях поділюваної пам'яті, семафорах і чергах повідомлень.

Команда ipcs

Синтаксис команди

```
ipcs [-asmq] [-tclup]
ipcs [-smq] -i id
ipcs -h
```

Опис команди

Команда `ipcs` призначена для отримання інформації про засоби System V IPC, до яких користувач має право доступу на читання.

Опція `-i` дозволяє вказати ідентифікатор ресурсів. Буде видаватися тільки інформація для ресурсів, які мають цей ідентифікатор.

Види IPC ресурсів можуть бути задані за допомогою наступних опцій:

`-s` для семафорів;

`-m` для сегментів поділюваної пам'яті;

`-q` для черг повідомлень;

`-a` для всіх ресурсів (за замовчуванням).

Опції `[-tclup]` використовуються для зміни складу вихідної інформації. За замовчуванням для кожного засобу виводяться його ключ, ідентифікатор IPC, ідентифікатор власника, права доступу та ряд інших характеристик. Застосування опцій дозволяє вивести:

- `-t` час здійснення останніх операцій над засобами IPC;
- `-p` ідентифікатори процесу, який створив ресурс, і процесу, який зробив над ним останню операцію;
- `-c` ідентифікатори користувача та групи для творця ресурсу та його власника;
- `-l` системні обмеження для засобів System V IPC;
- `-u` загальний стан IPC ресурсів у системі.

Опція `-h` використовується для отримання короткої довідкової інформації.

З усього різноманіття виведеної інформації нас будуть цікавити тільки IPC ідентифікатори для засобів, створених вами. Ці ідентифікатори будуть використовуватися в команді `ipcrm`, що дозволяє видалити необхідний ресурс із системи. Для видалення сегмента `поділюваної пам'яті` ця команда має вигляд:

```
ipcrm shm <IPC ідентифікатор>
```

ЗАВДАННЯ 2: Видаліть створений вами сегмент поділюваної пам'яті з операційної системи, використовуючи ці команди.

Команда ipcrm

Синтаксис команди

```
ipcrm [shm | msg | sem] id
```

Опис команди

Команда `ipcrm` призначена для видалення ресурсу System V IPC з операційної системи. Параметр `id` задає IPC ідентифікатор для ресурсу, що видаляється, `параметр` shm використовується для сегментів поділюваної пам'яті, `параметр` msg - для черг повідомлень, `параметр` sem - для семафорів.

Якщо поводження програм, які використовують засоби `System V IPC`, базується на припущенні, що ці засоби були створені при їх роботі, не забувайте перед їх запуском видаляти вже існуючі ресурси.

Використання системного виклику shmctl() для звільнення ресурсу

Для тієї ж мети – видалити область `поділюваної пам'яті` із системи – можна скористатися й системним викликом `shmctl()`. Цей системний виклик дозволяє цілком ліквідувати область `поділюваної пам'яті` в операційній системі за заданим дескриптором засобу IPC, якщо, звичайно, у вас вистачає для цього повноважень. Системний виклик `shmctl()` дозволяє виконувати й інші дії над сегментом `поділюваної пам'яті`, але їхнє вивчення лежить за межами нашого курсу.

Системний виклик shmctl()

Прототип системного виклику

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd,
           struct shmid_ds *buf);
```

Опис системного виклику

Системний виклик `shmctl` призначений для отримання інформації про області поділюваної пам'яті, зміни її атрибутів і видалення із системи. Даний опис не є повним описом системного виклику, а обмежується межами цього курсу. Для вивчення повного опису зверніться до UNIX Manual.

У нашому курсі ми будемо користуватися системним викликом `shmctl` тільки для видалення області поділюваної пам'яті із системи.

Параметр shmid є дескриптором System V IPC для сегмента поділюваної пам'яті, тобто значенням, яке повернув системний виклик shmget() при створенні сегмента або при його пошуку за ключем.

У якості параметр cmd у межах нашого курсу ми завжди будемо передавати значення IPC_RMID – команду для видалення сегмента поділюваної пам'яті із заданим ідентифікатором. Параметр buf для цієї команди не використовується, тому ми завжди будемо підставляти туди значення NULL.

Значення, яке повертається

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

Поділювана пам'ять і системні виклики fork(), exec() і функція exit()

Важливим питанням є поводження сегментів поділюваної пам'яті при виконанні процесом системних викликів fork(), exec() і функції exit().

При виконанні системного виклику fork() всі області поділюваної пам'яті, розміщені в адресному просторі процесу, успадковуються породженим процесом.

При виконанні системних викликів exec() і функції exit() всі області поділюваної пам'яті, розміщені в адресному просторі процесу, виключаються з його адресного простору, але продовжують існувати в операційній системі.

Самостійне написання, компіляція та запуск програми для організації зв'язку двох процесів через поділювану пам'ять.

ЗАВДАННЯ 3: Напишіть дві програми, які здійснюють взаємодію через поділювану пам'ять. Перша програма повинна створювати сегмент поділюваної пам'яті та копіювати туди власний вихідний текст, друга програма повинна брати відтіля цей текст, друкувати його на екрані та видаляти сегмент поділюваної пам'яті із системи.

Поняття про нитки виконання (thread) у UNIX. Ідентифікатор нитки виконання. Функція pthread_self()

На лекції 4 ми говорили, що в багатьох сучасних операційних системах існує розширена реалізація поняття процес, коли процес являє собою

сукупність вигляділенихйому ресурсів і набору ниток виконання. Нитки процесу розподілюють його програмний код, глобальні змінні та системні ресурси, але кожна нитка має власний програмний лічильник, свій уміст регістрів і свій стек. Оскільки глобальні змінні в нитках виконання є загальними, вони можуть використовувати їх у якості елементів поділюваної пам'яті, не прибігаючи до механізму, наведеному вище.

У різних версіях операційної системи UNIX існують різні інтерфейси, які забезпечують роботу з нитками виконання. Ми коротко ознайомимося з деякими функціями, які дозволяють розділити процес на thread'и та керувати їх поведінкою, у відповідності зі стандартом POSIX. Нитки виконання, що задовольняють стандартові POSIX, прийнято називати POSIX thread'ами або, стисло, pthread'ами.

На жаль, операційна система Linux не цілком підтримує нитки виконання на рівні ядра системи. При створенні нового thread'а запускається новий традиційний процес, який розподілює з батьківським традиційним процесом його ресурси, програмний код і дані, розташовані поза стек, тобто фактично дійсно створюється новий thread, але ядро не вміє визначати, які ці thread'и є складовими частинами одного цілого. Це «знає» тільки спеціальний процес-координатор, що працює на користувацькому рівні та стартує при першому виклику функцій, які забезпечують POSIX інтерфейс для ниток виконання. Тому ми зможемо спостерігати не всі переваги використання ниток виконання (зокрема, прискорити рішення задачі на однопроцесорній машині з їх допомогою навряд чи вийде), але навіть у цьому випадку thread'и можна задіяти у якості дуже зручного способу для створення процесів із загальними ресурсами, програмним кодом і поділюваною пам'яттю.

Кожна нитка виконання, як і процес, має в системі свій унікальний номер – ідентифікатор thread'а. Оскільки традиційний процес у концепції ниток виконання трактується у якості процеса, який містить єдину нитку виконання, ми можемо довідатися про ідентифікатор цієї нитки та для будь-якого звичайного процесу. Для цього використовується функція pthread_self(). Нитка виконання, створювана при народженні нового процесу, прийнято називати **початковою** або **головною** ниткою виконання цього процесу.

Функція pthread_self()
Прототип функції

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Опис функції

Функція `pthread_self` повертає ідентифікатор поточної нитки виконання.

Тип даних `pthread_t` є синонімом для одного з цілочисельних типів мови С.

Створення та завершення thread'a. Функції pthread_create(), pthread_exit(), pthread_join()

`Нитки виконання`, як і традиційні процеси, можуть породжувати нитки-нащадки, щоправда, тільки всередині свого процесу. Кожний майбутній thread усередині програми повинний являти собою функцію з прототипом

```
void *thread(void *arg);
```

Параметр `arg` передається цієї функції при створенні thread'a і може, до деякої міри, розглядатися У якості аналог параметрів функції `main()`, про які ми говорили на попередніх практичних заняттях. Значення, яке повертається функцією, може інтерпретуватися у якості аналога інформації, яку батьківський процес може отримати після завершення процесу-дитини. Для створення нової `нитки виконання` застосовується функція `pthread_create()`.

Функція для створення нитки виконання

Прототип функції

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
    pthread_attr_t *attr,
    void * (*start_routine)(void *),
    void *arg);
```

Опис функції

Функція `pthread_create` служить для створення нової нитки виконання (thread'a) усередині поточного процесу. Дійсний опис не є повним описом функції, а служить тільки цілям даного курсу. Для вивчення повного опису зверніться до UNIX Manual.

Новий `thread` буде виконувати функцію `start_routine` із прототипом

```
void *start_routine(void *)
```

передаючи їй У якості аргумент параметр `arg`. Якщо потрібно

передати більш одного параметра, вони збираються в структуру, і передається адреса цієї структури. Значення, яке повертається функцією `start_routine` не повинне вказувати на динамічний об'єкт даного thread'a.

Параметр `attr` служить для завдання різних атрибутів створюваного thread'a. Їхній опис виходить за межі нашого курсу, і ми завжди будемо вважати їх заданими за замовчуванням, підставляючи у якості аргумента значення `NULL`.

Значення, які повертаються

При вдалому завершенні функція повертає значення `0` і розміщує ідентифікатор нової нитки виконання за адресою, на який указує параметр `thread`. У випадку помилки повертається **позитивне значення** (а не негативне, як у більшості системних викликів і функцій!), яке визначає код помилки, описаний у файлі `<errno.h>`. Значення системної змінної `errno` при цьому не встановлюється.

Ми не будемо розглядати її в повному обсязі, тому що детальне вивчення програмування з використанням thread'ов не є метою нашого курсу.

Важливою відмінністю цієї функції від більшості інших системних викликів і функцій є те, що у випадку невдалого завершення вона **повертає не негативне, а позитивне значення**, яке визначає код помилки, описаний у файлі `<errno.h>`. Значення системної змінної `errno` при цьому не встановлюється. Результатом виконання цієї функції є поява в системі нової `нитки виконання`, яка буде виконувати функцію, асоційовану з thread'ом, передавши їй специфікований параметр, паралельно з вже існуючими `нитками виконання` процесу.

Створений `thread` може завершити свою діяльність трьома способами:

- За допомогою виконання функції pthread_exit(). Функція ніколи не повертається до її нитки виконання, яка викликала. Об'єкт, на який вказує параметр цієї функції, може бути вивчений в іншій нитці виконання, наприклад, у тій, що породила завершений thread. Цей параметр, отже, повинний указувати на об'єкт, який не є локальним для завершеного thread'a, наприклад, на статичну змінну;
- За допомогою повернення з функції, асоційованої з ниткою виконання. Об'єкт, на який вказує адреса, яка повертається функцією, як і в попередньому випадку, може бути вивчений в

іншій нитці виконання, наприклад, у тій, що породила завершений thread, і повинний указувати на об'єкт, який не є локальним для завершеного thread'a;

- Якщо в процесі виконується повернення з функції main() або де-небудь у процесі (у будь-якій нитці виконання) здійснюється виклик функції exit(), це приводить до завершення всіх thread'ов процесу.

Функція для завершення нитки виконання
Прототип функції

```
#include <pthread.h>
void pthread_exit(void *status);
```

Опис функції

Функція `pthread_exit` служить для завершення нитки виконання (`thread`) поточного процесу.

Функція ніколи не повертається до `thread, що` її викликав. Об'єкт, на який вказує параметр `status`, може бути згодом вивчений в іншій нитці виконання, наприклад у нитці, що породила нитку, яка завершилася. Тому він не повинний вказувати на динамічний об'єкт завершеного thread'a.

Одним з варіантів отримання адреси, яке повертається завершеним thread'ом, з одночасним чеканням його завершення є використання функції `pthread_join()`. `Нитка виконання`, яка викликала цю функцію, переходить до стану **чекання** до завершення заданого thread'a. Функція дозволяє також отримати покажчик, який повернув завершений thread до операційної системи.

Функція pthread_join()
Прототип функції

```
#include <pthread.h>
int pthread_join (pthread_t thread,
void **status_addr);
```

Опис функції

Функція `pthread_join` блокує роботу її нитки, яка викликала, виконання до завершення thread'a з `ідентифікатором` thread. Після розблокування в покажчик, розташований за адресою `status_addr`, заноситься адреса, яку повернув завершений thread або при виході з асоційованої з ним функції, або при виконанні функції

pthread_exit(). Якщо нас не цікавить, що повернула нам нитка виконання, у якості цього параметра можна використовувати значення NULL.

Значення, які повертаються

Функція повертає значення 0 при успішному завершенні. У випадку помилки повертається **позитивне значення** (а не негативне, У якості у більшості системних викликів і функцій!), що визначає код помилки, описаний у файлі <errno.h>. Значення системної змінної errno при цьому не встановлюється.

Прогін програми з використанням двох ниток виконання

Для ілюстрації вищенаведеного давайте розглянемо програму, у якій працюють дві нитки виконання.

```
/* Програма 05-2.c для ілюстрації роботи двох
ниток виконання.
Кожна нитка виконання просто збільшує на 1
поділювану
змінну a. */
#include <pthread.h>
#include <stdio.h>
int a = 0;
/* Змінна a є глобальною статичною для всієї
програми, тому вона буде розділятися обома нитками
виконання.*/
/* Нижче випливає текст функції, яка буде
асоційована з 2-м thread'ом */
void *mythread(void *dummy)
/*   Параметр   dummy   у   нашій   функції   не
використовується та
є присутнім тільки для сумісності типів даних. По
тієї
же причині функція повертає значення void *, хоча
це ні яким чином не використовується в програмі.*/
{
     pthread_t mythid; /* Для ідентифікатора нитки
виконання */
    /* Зауважимо, що змінна mythid є
динамічної локальної змінної функції mythread(),
т. е. міститься в стеку і, отже, не розділяється
нитками виконання. */
```

```
    /* Запитуємо ідентифікатор thread'a */
    mythid = pthread_self();
    a = a+1;
    printf("Thread %d, Calculation result = %d\n",
        mythid, a);
    return NULL;
}
/* Функція main() – вона ж асоційована функція
головного
thread'a */
int main()
{
    pthread_t thid, mythid;
    int result;
    /* Намагаємося створити нову нитку виконання,
асоційовану з функцією mythread(). Передаємо їй
у якості параметр значення NULL. У випадку удачі в
змінну thid буде занесений ідентифікатор нового
thread'a.
Якщо виникне помилка, то припинимо роботу. */
    result = pthread_create( &thid,
        (pthread_attr_t *)NULL, mythread, NULL);
    if(result != 0){
    printf ("Error on thread create,
        return value = %d\n", result);
    exit(-1);
    }
    printf("Thread created, thid = %d\n", thid);
     /* Запитуємо ідентифікатор головного thread'a
*/
    mythid = pthread_self();
    a = a+1;
    printf("Thread %d, Calculation result = %d\n",
        mythid, a);
      /* Очікуємо завершення породженого thread'a,
не
     цікавлячи, яке значення він нам поверне. Якщо
не
       виконати виклик цієї функції, те можлива
ситуація,
     коли ми завершимо функцію main() до того, У
якості виконається
```

```
      породжений thread, що автоматично спричинить
за
     собою його завершення, спотворивши результати.
*/
    pthread_join(thid, (void **)NULL);
    return 0;
}
```

Лістінг 5.2. Програма 05-2.c для ілюстрації роботи двох ниток виконання.

Для зборки файлу, який виконується, при роботі редактора зв'язків **необхідно явно підключити бібліотеку функцій для роботи з pthread'ами**, яка не підключається автоматично. Це робиться за допомогою додавання до команди компіляції та редагування зв'язків параметра `-lpthread` – підключити бібліотеку `pthread`.

ЗАВДАННЯ 4: Наберіть текст, відкомпілюйте цю програму та запустіть на виконання.

Зверніть увагу на відмінність результатів цієї програми від схожої програми, що ілюструвала створення нового процесу (розділ «Прогін програми з `fork()` з однаковою роботою батька та дитини»), які ми розглядали на попередніх практичних заняттях. Програма, що створювала новий процес, друкувала двічі однакові значення для змінної `a`, тому що адресні простори різних процесів незалежні, і кожний процес додавав 1 до своєї власної змінного `a`. Розглянута програма друкує два різних значення, тому що змінна `a` є поділюваної, і кожний thread додає 1 до одній і тієї ж змінній.

Написання, компіляція і прогін програми з використанням трьох ниток виконання.

ЗАВДАННЯ 5: Модифікуйте попередню програму, додавши до неї третю нитку виконання.

Необхідність синхронізації процесів і ниток виконання, які використовують загальну пам'ять

Усі розглянуті на цьому семінарі приклади є не зовсім коректними. У більшості випадків вони працюють правильно, однак можливі ситуації, коли спільна діяльність цих процесів або ниток виконання приводить до невірних і несподіваних результатів. Це зв'язано з тим, що будь-які

неатомарні операції, пов'язані зі зміною вмісту поділюваної пам'яті, являють собою критичну секцію процесу або нитки виконання. Згадайте розгляд критичних секцій у лекції 5.

Повернемося до розгляду програм з розділу «Прогін програм з використанням `поділюваної пам'яті`». При одночасному існуванні двох процесів в операційній системі може виникнути наступна послідовність виконання операцій у часі:

```
...
Процес 1: array[0] += 1;
Процес 2: array[1] += 1;
Процес 1: array[2] += 1;
Процес 1: printf(
   "Program 1 was spawn %d times,
   program 2 - %d times, total - %d times\n",
   array[0], array[1], array[2]);
...
```

Тоді роздрук буде давати неправильні результати. Природно, що відтворити подібну послідовність дій практично нереально. Ми не зможемо підібрати необхідний час старту процесів і ступінь завантаженості обчислювальної системи. Але ми можемо змоделювати цю ситуацію, додавши до обох програм досить тривалі порожні цикли перед оператором `array[2] += 1;` Це зроблено в наступних програмах.

```
/* Програма 1 (05-3a.c) для ілюстрації
некоректної роботи з поділюваною пам'яттю */
/* Ми організуємо поділювану пам'ять для масиву з
трьох цілих
чисел. Перший елемент масиву є лічильником числа
запусків програми 1, тобто даної програми, другий
елемент
масиву – лічильником числа запусків програми 2,
третій
елемент масиву – лічильником числа запусків обох
програм */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <errno.h>
int main()
{
```

```
     int *array; /* Покажчик на поділювану пам'ять
*/
    int shmid; /* IPC дескриптор для області
    поділюваної пам'яті */
       int new = 1; /* Прапорець необхідності
ініціалізації
        елементів масиву */
    char pathname[] = "05-3a.c"; /* Ім'я файлу,
         який використовується для генерації ключа.
Файл із таким
             ім'ям повинний існувати в поточній
директорії */
    key_t key; /* IPC ключ */
    long i;
    /* Генеруємо IPC ключ з імені файлу 05-3a.c у
       поточній директорії та номера екземпляра
області
    поділюваної пам'яті 0 */
    if((key = ftok(pathname,0)) < 0){
        printf("Can\'t generate key\n");
        exit(-1);
    }
     /* Намагаємося ексклюзивно створити поділювану
пам'ять для сгенерированного ключа, тобто якщо для
цього ключа вона вже існує, системний виклик
поверне негативне
      значення. Розмір пам'яті визначаємо у якості
розміру масиву з 3-х цілих змінних, права доступу
0666 - читання і запис дозволені для всіх */
    if((shmid = shmget(key, 3*sizeof(int),
        0666|IPC_CREAT|IPC_EXCL)) < 0){
      /* У випадку виникнення помилки намагаємося
визначити:
        чи виникла вона через те, що сегмент
поділюваної
    пам'яті вже існує або з іншої причини */
        if(errno != EEXIST){
             /* Якщо з іншої причини - припиняємо
роботу */
                  printf("Can\'t create shared
memory\n");
            exit(-1);
        } else {
```

```
                /* Якщо через те, що поділювана
пам'ять уже
              існує – намагаємося отримати її IPC
дескриптор і у випадку удачі, скидаємо прапорець
необхідності ініціалізації елементів масиву */
            if((shmid = shmget(key, 3*sizeof(int),
0)) < 0){
                printf("Can\'t find shared memory\
n");
               exit(-1);
            }
            new = 0;
        }
    }
     /* Намагаємося відобразити поділювану пам'ять
до адресного простору поточного процесу. Зверніть
увагу на те, що для правильного порівняння ми явно
перетворюємо         значення -1 до покажчика на
ціле.*/
    if((array = (int *)shmat(shmid, NULL, 0)) ==
        (int *)(-1)){
        printf("Can't attach shared memory\n");
        exit(-1);
    }
/* У залежності від значення прапорця new або
ініціюємо масив, або збільшуємо
відповідні лічильники */
    if(new){
        array[0] = 1;
        array[1] = 0;
        array[2] = 1;
    } else {
        array[0] += 1;
        for(i=0; i<1000000000L; i++);
              /* Граничне значення для і може
змінюватися в залежності від продуктивності
комп'ютера */
        array[2] += 1;
    }
        /* Друкуємо нові значення лічильників,
видаляємо поділювану пам'ять з адресного простору
поточного процесу та завершуємо роботу */
    printf("Program 1 was spawn %d times,
```

```
        program 2 - %d times, total - %d times\n",
        array[0], array[1], array[2]);
    if(shmdt(array) < 0){
        printf("Can't detach shared memory\n");
        exit(-1);
    }
    return 0;
}
```

Лістінг 5.3а. Програма 1 (05-3a.c) для ілюстрації некоректної роботи з поділюваною пам'яттю.

```
/* Програма 2 (05-3b.c) для ілюстрації
некоректної роботи з поділюваною пам'яттю */
/* Ми організуємо поділювану пам'ять для масиву з
трьох
цілих чисел. Перший елемент масиву є лічильником
числа запусків програми 1, тобто даної програми,
другий елемент масиву – лічильником числа запусків
програми 2, третій елемент масиву – лічильником
числа
запусків обох програм */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <errno.h>
int main()
{
int *array; /* Покажчик на поділювану пам'ять */
    int shmid; /* IPC дескриптор для області
        поділюваної пам'яті */
       int new = 1; /* Прапорець необхідності
ініціалізації
    елементів масиву */
      char pathname[] = "05-3a.c"; /* Ім'я файлу,
який використовується для генерації ключа. Файл із
таким ім'ям повинний існувати в поточній
директорії */
    key_t key; /* IPC ключ */
    long i;
```

```
    /* Генеруємо IPC ключ з імені файлу 05-3a.c у
поточній директорії та номери екземпляра області
поділюваної пам'яті 0 */
    if((key = ftok(pathname,0)) < 0){
        printf("Can\'t generate key\n");
        exit(-1);
    }
    /* Намагаємося ексклюзивно створити поділювану
пам'ять для сгенерованого ключа, тобто якщо для
цього ключа вона вже існує, системний виклик
поверне негативне значення. Розмір пам'яті
визначаємо у якості розміру масиву з трьох цілих
змінних, права доступу 0666 - читання і запис
дозволені для всіх */
    if((shmid = shmget(key, 3*sizeof(int),
        0666|IPC_CREAT|IPC_EXCL)) < 0){
    /* У випадку помилки намагаємося визначити, чи
виникла вона через те, що сегмент поділюваної
пам'яті вже існує або з іншої причини */
        if(errno != EEXIST){
              /* Якщо з іншої причини - припиняємо
роботу */
                    printf("Can\'t create shared
memory\n");
            exit(-1);
        } else {
                 /* Якщо через те, що поділювана
пам'ять уже
             існує - намагаємося отримати її IPC
дескриптор і, у випадку удачі, скидаємо прапорець
необхідності ініціалізації елементів масиву */
            if((shmid = shmget(key,
                3*sizeof(int), 0)) < 0){
                 printf("Can\'t find shared memory\
n");
                exit(-1);
            }
            new = 0;
        }
    }
    /* Намагаємося відобразити поділювану пам'ять
до адресного простору поточного процесу. Зверніть
```

```
увагу на те, що для правильного порівняння ми явно
перетворюємо
    значення -1 до покажчика на ціле.*/
    if((array = (int *)shmat(shmid, NULL, 0)) ==
        (int *)(-1)){
        printf("Can't attach shared memory\n");
        exit(-1);
    }
    /* У залежності від значення прапорця new або
ініціюємо масив, або збільшуємо
відповідні лічильники */
    if(new){
        array[0] = 0;
        array[1] = 1;
        array[2] = 1;
    } else {
        array[1] += 1;
        for(i=0; i<1000000000L; i++);
               /* Граничне значення для і може
змінюватися в залежності від продуктивності
комп'ютера */
        array[2] += 1;
    }
        /* Друкуємо нові значення лічильників,
видаляємо поділювану пам'ять з адресного простору
поточного процесу та завершуємо роботу */
    printf("Program 1 was spawn %d times,
        program 2 - %d times, total - %d times\n",
        array[0], array[1], array[2]);
        if(shmdt(array) < 0){
                    printf("Can't detach shared
memory\n");
            exit(-1);
        }
        return 0;
}
```

Лістінг 5.3b. Програма 2 (05-3b.c) для ілюстрації некоректної роботи з поділюваною пам'яттю.

ЗАВДАННЯ 5: Наберіть програми, збережіть під іменами 05-3a.c і 05-3b.c відповідно, відкомпілюйте їх і запустіть кожну з них один раз для створення й ініціалізації поділюваної пам'яті. Потім запустите іншу і, поки вона

знаходиться в циклі, запустіть, наприклад, з іншого віртуального термінала, знову першу програму. Ви отримаєте несподіваний результат: кількість запусків по окремості не буде відповідати кількості запусків разом.

Як ми бачимо, для написання коректно працюючих програм необхідно забезпечувати взаємовиключення при роботі з `поділюваною пам'яттю` і, може бути, взаємну черговість доступу до неї. Це можна зробити за допомогою розглянутих у лекції 6 алгоритмів синхронізації, наприклад, алгоритму Петерсона або алгоритму булочної.

ЗАВДАННЯ 6: модифікуйте програми з цього розділу для коректної роботи за допомогою алгоритму Петерсона.

Надалі ми розглянемо семафори, які є засобом System V IPC, призначеним для синхронізації процесів.

Семафори в UNIX. Відмінність операцій над UNIX-семафорами від класичних операцій

У попередніх матеріалах мова йшла про необхідність синхронізації роботи процесів для їхньої коректної взаємодії через поділювану пам'ять. Як згадувалося в лекції 6, одним з перших механізмів, запропонованих для синхронізації поводження процесів, стали семафори, концепцію яких описав Дейкстра (Dijkstra) у 1965 році. При розробці засобів `System V IPC` семафори ввійшли до їх складу у якості невід'ємної частини. Слід зазначити, що набір операцій над `семафорами System V IPC` відрізняється від класичного набору операцій `{P, V}`, запропонованого Дейкстрой. Він включає три операції:

- A(S, n) – збільшити значення семафора `S` на величину `n`;
- D(S, n) – поки значення семафора `S < n`, процес блокується. Далі `S = S - n;`
- Z(S) – процес блокується доти, поки значення семафора `S` не стане рівним `0`.

Споконвічно всі IPC-семафори ініціюються нульовим значенням.

Ми бачимо, що класичній операції `P(S)` відповідає операція `D(S,1)`, а класичній операції `V(S)` відповідає операція `A(S,1)`. Аналогом ненульової ініціалізації семафорів Дейкстри значенням n може служити виконання операції `A(S,n)` відразу після створення семафора `S`, із забезпеченням атомарності створення семафора та її виконання за

допомогою іншого семафора. Ми показали, що класичні семафори реалізуються через семафори System V IPC. Зворотне не є вірним. Використовуючи операції P(S) і V(S), ми не зуміємо реалізувати операцію Z(S).

Оскільки IPC-семафори є складовою частиною засобів System V IPC, то для них вірно усе, що говорилося про ці засоби вище. IPC-семафори є засобом зв'язку з непрямою адресацією, вимагають ініціалізації для організації взаємодії процесів і спеціальних дій для звільнення системних ресурсів по його закінченні. Простором імен IPC-семафорів є безліч значень ключа, які генеруються за допомогою функції ftok(). Для здійснення операцій над семафорами системним викликам у якості параметра передаються IPC- дескриптори семафорів, які однозначно ідентифікують їх у всій обчислювальній системі, а вся інформація про семафори розташовується в адресному просторі ядра операційної системи. Це дозволяє організовувати через семафори взаємодію процесів, які навіть не знаходяться в системі одночасно.

Створення масиву семафорів або доступ до вже існуючого. Системний виклик semget()

З метою економії системних ресурсів операційна система UNIX дозволяє створювати не по одному семафору для кожного конкретного значення ключа, а зв'язувати з ключем цілий масив семафорів (у Linux – до 500 семафорів у масиві, хоча ця кількість може бути зменшеною системним адміністратором). Для створення масиву семафорів, асоційованого з визначеним ключем, або доступу за ключем до вже існуючого масиву використовується системний виклик semget(), що є аналогом системного виклику shmget() для поділюваної пам'яті, який повертає значення IPC-дескриптора для цього масива. При цьому застосовуються ті ж способи створення та доступу (див. розділ «Поділювана пам'ять у UNIX. Системні виклики shmget(), shmat(), shmdt()»), що і для поділюваної пам'яті. Знову створені семафори ініціюються нульовим значенням.

Системний виклик semget()

Прототип системного виклику

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems,
           int semflg);
```

Опис системного виклику

Системний виклик `semget` призначений для виконання операції доступу до масиву IPC-семафорів і, у випадку її успішного завершення, повертає дескриптор System V IPC для цього масиву (ціле ненегативне число, яке однозначно характеризує масив семафорів усередині обчислювальної системи і використовується надалі для інших операцій з ним).

Параметр `key` є ключем System V IPC для масиву семафорів, тобто фактично його ім'ям із простору імен System V IPC. У якості значення цього параметра може використовуватися значення ключа, отримане за допомогою функції `ftok()`, або спеціальне значення `IPC_PRIVATE`. Використання значення `IPC_PRIVATE` **завжди** приводить до спроби створення нового масиву семафорів із ключем, який не збігається зі значенням ключа жодного із вже існуючих масивів і не може бути отриманий за допомогою функції `ftok()` ні при одній комбінації її параметрів.

Параметр `nsems` визначає кількість семафорів у створюваному або вже існуючому масиві. У випадку, якщо масив із зазначеним ключем вже є, але його розмір не збігається із зазначеним у параметрі `nsems`, констатується виникнення помилки.

Параметр `semflg` – прапорці – відіграє роль тільки при створенні нового масиву семафорів і визначає права різних користувачів при доступі до масиву, а також необхідність створення нового масиву та поведінку системного виклику при спробі створення. Він є деякою комбінацією (за допомогою операції побітове або – «|») наступних визначених значень і вісімкових прав доступу:

`IPC_CREAT` — якщо масива для зазначеного ключа не існує, він повинний бути створений

`IPC_EXCL` — застосовується разом із прапорцем `IPC_CREAT`. При спільному їх використанні й існуванні масиву із зазначеним ключем, доступ до масиву не виконується та констатується помилка, при цьому змінна `errno`, описана у файлі `<errno.h>`, прийме значення `EEXIST`

- `0400` — дозволене читання для користувача, який створив масив
- `0200` — дозволений запис для користувача, який створив масив
- `0040` — дозволене читання для групи користувача, який створив масив
- `0020` — дозволений запис для групи користувача, який

створив масив

- `0004` — дозволене читання для всіх інших користувачів
- `0002` — дозволений запис для всіх інших користувачів

Знову створені семафори ініціюються нульовим значенням.

Значення, яке повертається

Системний виклик повертає значення дескриптора System V IPC для масиву семафорів при нормальному завершенні та значення `-1` при виникненні помилки.

Виконання операцій над семафорами. Системний виклик semop()

Для виконання операцій `A`, `D` і `Z` над семафорами з `масиву` використовується системний виклик semop(), який володіє досить складною семантикою. Розроблювачі System V IPC явно перевантажили цей виклик, застосовуючи його не тільки для виконання всіх трьох операцій, але ще й для декількох семафорів у масиві IPC-семафорів одночасно. Для правильного використання цього виклику необхідно виконати наступні дії:

- Визначитися, для яких семафорів з `масиву` треба виконати операції. Необхідно мати на увазі, що всі операції реально відбуваються тільки перед успішним поверненням із системного виклику, тобто якщо ви хочете виконати операції `A(S1,5)` і `Z(S2)` в одному виклику та виявилося, що `S2 != 0`, то значення семафора `S1` не буде зміненим доти, поки значення `S2` не стане рівним `0`. Порядок виконання операцій у випадку, коли процес не переходить до стану **чекання**, не визначене. Так, наприклад, при одночасному виконанні операцій `A(S1,1)` і `D(S2,1)` у випадку `S2 > 1` невідомо, що відбудеться раніш – зменшиться значення семафора `S2` або збільшиться значення семафора `S1`. Якщо порядок для вас важливий, краще застосувати кілька викликів замість одного.
- Після того як ви визначилися з кількістю семафорів і чинних операцій, необхідно завести в програмі масив з елементів типу `struct sembuf` з розмірністю, рівною визначеній кількості семафорів (якщо операція відбувається тільки над одним семафором, можна, природно, обійтися просто змінною). Кожний елемент цього масиву буде відповідати операції над одним семафором.

- Заповнити елементи масиву. До поля `sem_flg` кожного елемента потрібно занести значення `0` (інші значення прапорців у семінарах ми розглядати не будемо). До полів `sem_num` і `sem_op` варто занести номера семафорів у масиві IPC семафорів і відповідні коди операцій. Семафори нумеруються, починаючи з `0`. Якщо у вас у масиві всего один семафор, то він буде мати номер `0`. Операції кодуються так:
    - для виконання операції `A(S,n)` значення поля `sem_op` повинне дорівнювати `n`;
    - для виконання операції `D(S,n)` значення поля `sem_op` повинне дорівнювати `-n`;
    - для виконання операції `Z(S)` значення поля `sem_op` повинне дорівнювати `0`.
- У якості другий параметр системного виклику `semop()` вказати адреса заповненого масиву, а у якості третього параметра – раніше визначену кількість семафорів, над якими відбуваються операції.

Системний виклик semop()

Прототип системного виклику

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops,
          int nsops);
```

Опис системного виклику

`Системний виклик semop` призначений для виконання операцій `A`, `D` і `Z` (див. опис операцій над семафорами з масиву IPC семафорів – розділ «Створення масиву семафорів або доступ до вже існуючого. Системний виклик `semget()`» цього заняття). Для повного опису зверніться до UNIX Manual.

Параметр `semid` є дескриптором System V IPC для набору семафорів, тобто значенням, який повернув системний виклик `semget()` при створенні набору семафорів або при його пошуку за ключем.

Кожний з `nsops` елементів масиву, на який указує параметр sops, визначає операцію, яка повинна бути зроблена над яким-небудь семафором з масиву IPC семафорів, і має тип структури `struct sembuf`, до якої входять наступні змінні:

- short sem_num — номер семафора в масиві IPC семафорів (нумеруються, починаючи з 0);
- short sem_op — виконувана операція;
- short sem_flg — прапорці для виконання операції. У нашому курсі завжди будемо вважати цю змінну рівної 0.

Значення елемента структури sem_op визначається в такий спосіб:

- для виконання операції A(S,n) значення повинне дорівнювати n;
- для виконання операції D(S,n) значення повинне дорівнювати -n;
- для виконання операції Z(S) значення повинне дорівнювати 0.

Семантика системного виклику має на увазі, що всі операції будуть у реальності виконані над семафорами тільки перед успішним поверненням із системного виклику. Якщо при виконанні операцій D або Z процес перейшов до стану **чекання**, то він може бути виведений з цього стану при виникненні наступних форс-мажорних ситуацій:

- масив семафорів був вилучений із системи;
- процес отримав сигнал, який повинний бути оброблений.

У цьому випадку відбувається повернення із системного виклику з констатацією помилкової ситуації.

Значення, яке повертається

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

Прогін приклада з використанням семафора

Для ілюстрації сказаного розглянемо найпростіші програми, що синхронізують свої дії за допомогою семафорів

```
 /* Програма 05-4a.c для ілюстрації роботи з
семафорами */
/* Ця програма отримує доступ до одного системного
семафора,
чекає, поки його значення не стане більше або
рівним 1
після запусків програми 05-4b.c,а потім зменшує
його на 1*/
#include <sys/types.h>
```

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
int main()
{
   int semid; /* IPC дескриптор для масиву IPC
семафорів */
   char pathname[] = "05-4a.c"; /* Ім'я файлу,
         що використовується для генерації ключа.
Файл із таким ім'ям повинний існувати в поточній
директорії */
   key_t key; /* IPC ключ */
   struct sembuf mybuf; /* Структура для завдання
      операції над семафором */
    /* Генеруємо IPC-ключ з імені файлу 05-4a.c у
поточній директорії та номери екземпляра масиву
семафорів 0 */
   if((key = ftok(pathname,0)) < 0){
      printf("Can\'t generate key\n");
      exit(-1);
   }
     /* Намагаємося отримати доступ за ключем до
масиву
семафорів, якщо він існує, або створити його з
одного
семафора, якщо його ще не існує, із правами
доступу
read & write для всіх користувачів */
   if((semid = semget(key, 1, 0666 | IPC_CREAT)) <
0){
      printf("Can\'t get semid\n");
      exit(-1);
   }
     /* Виконаємо операцію D(semid1,1) для нашого
масиву
     семафорів. Для цього спочатку заповнимо нашу
структуру.
      Прапорець дорівнює 0. Наш масив семафорів
складається з одного семафора з номером 0. Код
операції -1.*/
   mybuf.sem_op = -1;
   mybuf.sem_flg = 0;
   mybuf.sem_num = 0;
```

```
   if(semop(semid, &mybuf, 1) < 0){
      printf("Can\'t wait for condition\n");
      exit(-1);
   }
   printf("Condition is present\n");
   return 0;
}
```

Лістінг 5.4а. Програма 05-4a.c для ілюстрації роботи із семафорами

```
/* Програма 05-4b.c для ілюстрації роботи з
семафорами */
/* Ця програма отримує доступ до одного системного
семафора
і збільшує його на 1*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
int main()
{
   int semid; /* IPC дескриптор для масиву IPC
      семафорів */
   char pathname[] = "05-4a.c"; /* Ім'я файлу,
        який використовується для генерації ключа.
Файл із таким ім'ям повинний існувати в поточній
директорії */
   key_t key; /* IPC ключ */
    struct sembuf mybuf; /* Структура для завдання
операції над семафором */
    /* Генеруємо IPC ключ з імені файлу 05-4a.c у
поточній
      директорії та номери екземпляра масива
семафорів 0 */
   if((key = ftok(pathname,0)) < 0){
      printf("Can\'t generate key\n");
      exit(-1);
   }
```

```
     /* Намагаємося отримати доступ за ключем до
масиву
семафорів, якщо він існує, або створити його з
одного семафора, якщо його ще не існує, із правами
доступу
read & write для всіх користувачів */
   if((semid = semget(key, 1, 0666 | IPC_CREAT)) <
0){
      printf("Can\'t get semid\n");
      exit(-1);
   }
     /* Виконуємо операцію A(semid1,1) для нашого
масиву
     семафорів. Для цього спочатку заповнимо нашу
структуру.
     Прапор, У якості звичайно, думаємо рівним 0.
Наш масив
      семафорів складається з одного семафора з
номером 0.
   Код операції 1.*/
   mybuf.sem_op = 1;
   mybuf.sem_flg = 0;
   mybuf.sem_num = 0;
   if(semop(semid, &mybuf, 1) < 0){
      printf("Can\'t wait for condition\n");
      exit(-1);
   }
   printf("Condition is set\n");
   return 0;
}
```

Лістінг 5.4b. Програма 05-4b.c для ілюстрації роботи із семафорами

Перша програма виконує над семафором S операцію D(S,1), друга програма виконує над тим же семафором операцію A(S,1). Якщо семафора в системі не існує, будь-яка програма створює його перед виконанням операції. Оскільки при створенні семафор завжди ініціюється 0, то програма 1 може працювати без блокування тільки після запуску програми 2.

ЗАВДАННЯ 7: Наберіть програми, збережете під іменами 05-4a.c і 05-4b.c відповідно, відкомпілюйте та перевірте правильність їхнього поводження.

Зміна попереднього приклада

ЗАВДАННЯ 8: Зміните програми з попереднього розділу так, щоб перша програма могла працювати без блокування після не менш 5 запусків другої програми.

Видалення набору семафорів із системи за допомогою команди ipcrm або системного виклику semctl()

Як ми бачили в попередніх прикладах, масив семафорів може продовжувати існувати в системі й після завершення його виконаних процесів, а семафори будуть зберігати своє значення. Це може привести до некоректного поводження програм, які припускають, що семафори були тільки що створені і, отже, мають нульове значення. Необхідно видаляти семафори із системи перед запуском таких програм або перед їх завершенням. Для видалення семафорів можна скористатися командами ipcs і ipcrm, які ми розглянули вище. Команда ipcrm у цьому випадку повинна мати вигляд

```
ipcrm sem <IPC ідентифікатор>
```

Для цієї ж мети ми можемо застосовувати системний виклик semctl(), який вміє виконувати й інші операції над масивом семафорів, але їхній розгляд виходить за межі нашого курсу.

Системний виклик semctl()

Прототип системного виклику

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd,
           union semun arg);
```

Опис системного виклику

Системний виклик semctl призначений для отримання інформації про масив IPC семафорів, зміни його атрибутів і видалення його із системи. Даний опис не є повним описом системного виклику. Для вивчення повного опису зверніться до UNIX Manual.

У нашому курсі ми будемо застосовувати системний виклик semctl

тільки для видалення масиву семафорів із системи. Параметр `semid` є дескриптором System V IPC для масиву семафорів, тобто значенням, яке повертає системний виклик `semget()` при створенні масиву або при його пошуку за ключем.

У якості параметр `cmd` у межах нашого курсу ми завжди будемо передавати значення `IPC_RMID` – команду для видалення сегмента поділюваної пам'яті із заданим ідентифікатором. Параметри `semnum` і `arg` для цієї команди не використовуються, тому ми завжди будемо підставляти замість них значення `0`.

Якщо які-небудь процеси знаходилися в стані чекання для семафорів з масиву, що видаляється, при виконанні системного виклику `semop()`, то вони будуть розблоковані та повернуться з виклику `semop()` з індикацією помилки.

Значення, яке повертається

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

Написання, компіляція та прогін програми з організацією взаимовиключення за допомогою семафорів для двох процесів, взаємодіючих через поділювану пам'ять

У матеріалах вище було показано, що будь-які неатомарні операції, зв'язані зі зміною вмісту поділюваної пам'яті, являють собою критичну секцію процесу або нитки виконання. Модифікуйте програми з розділу «Необхідність синхронізації процесів і ниток виконання, що використовують загальну пам'ять» семінарів 6–7, що ілюстрували некоректну роботу через поділювану пам'ять, забезпечивши за допомогою семафорів взаимовиключення для їхньої правильної роботи.

Написання, компіляція і прогін програми з організацією взаємної черговості за допомогою семафорів для двох процесів, взаємодіючих через ріре

У матеріалах практичного заняття 4, коли мова йшла про зв'язок родинних процесів через ріре, відзначалося, що ріре є односпрямованим каналом зв'язку, і що для організації зв'язку через один ріре у двох напрямках необхідно використовувати механізми взаємної синхронізації процесів. Організуйте двосторонній почерговий зв'язок процесу-батька і дитину-процесу-дитини через ріре, використовуючи для синхронізації семафори, модифікувавши програму з розділу «Прогін програми для організації односпрямованого зв'язку між родинними процесами через ріре» практичного заняття 4.

Поняття про POSIX-семафори

У стандарті POSIX вводяться інші семафори, цілком аналогічні семафорам Дейкстри. Для ініціалізації значення таких семафорів застосовується функція `sem_init()`, аналогом операції `P` служить функція `sem_wait()`, а аналогом операції `V` – функція `sem_post()`. На жаль, у Linux такі семафори реалізовані тільки для ниток виконання одного процесу, й тому докладно ми на них зупинятися не будемо.

Повідомлення у якості засобу зв'язку та засобу синхронізації процесів

У матеріалах попередніх семінарів були представлені такі засоби організації взаємодії процесів зі складу засобів System V IPC, як поділювана пам'ять і семафори. Третім і останнім, найбільш семантично навантаженим засобом, що входить до System V IPC, є черги повідомлень. У лекції 6 говорилося про моделі повідомлень як про засіб взаємодії процесів через лінії зв'язку, у якому на передану інформацію накладається певна структура, так що процес, який приймає дані, може чітко визначити, де закінчується одна порція інформації та починається інша. Така модель дозволяє задіяти ту саму лінію зв'язку для передачі даних у двох напрямках між декількома процесами. Ми також розглядали можливість використання повідомлень з вбудованими механізмами взаимовиключення та блокування при читанні з порожнього буфера та запису до переповненого буферу для організації синхронізації процесів.

Надалі мова йтиме про використання `черг повідомлень System V IPC` для забезпечення обох названих функцій.

Черги повідомлень у UNIX у якості складової частини System V IPC

Тому що `черги повідомлень` входять до складу засобів System V IPC, для них вірно усе, що ми казали раніше про ці засоби в цілому та вже знайомо нам. `Черги повідомлень`, як і семафори, і поділювана пам'ять, є засобом зв'язку з непрямою адресацією, вимагають ініціалізації для організації взаємодії процесів і спеціальних дій для звільнення системних ресурсів по закінченні взаємодії. Простором імен `черг повідомлень` є та ж сама безліч значень ключа, згенерованих за допомогою функції `ftok()` (див. розділ «Простір імен. Адресація в System V IPC. Функція `ftok()`»). Для виконання примітивів `send` і `receive`, введених у лекції 6, які відповідають системним викликам У якості параметр передаються IPC-дескриптори (див. розділ

«Дескриптори System V IPC») черг повідомлень, які однозначно ідентифікують їх по всій обчислювальній системі.

Черги повідомлень розташовуються в адресному просторі ядра операційної системи у вигляді односпрямованих списків і мають обмеження за обсягом інформації, який зберігається в кожній черзі. Кожний елемент списку є окремим повідомленням. Повідомлення мають атрибут, який має назву типа повідомлення. Вибірка повідомлень з черги (виконання примітива receive) може здійснюватися трьома способами:

1) у порядку FIFO, незалежно від типу повідомлення;
2) у порядку FIFO для повідомлень конкретного типу;
3) першим вибирається повідомлення з мінімальним типом, яке не перевищує деякого заданого значення, що прийшло раніш за інші повідомлення з тим же типом.

Реалізація примітивів send і receive забезпечує приховане від користувача взаимовиключене під час переміщення повідомлення до черги або його отримання з черги. Також вона забезпечує блокування процесу при спробі виконати примітив receive над порожньою чергою або чергою, у якій відсутні повідомлення запитаного типу, або при спробі виконати примітив send для черги, у якій немає вільного місця.

Черги повідомлень, як й інші засоби System V IPC, дозволяють організувати взаємодію процесів, які не знаходяться одночасно в обчислювальній системі.

Створення черги повідомлень або доступ до вже існуючої. Системний виклик msgget()

Для створення черги повідомлень, асоційованої з визначеним ключем, або доступу за ключем до вже існуючої черги використовується системний виклик msgget(), що є аналогом системних викликів shmget() для поділюваної пам'яті та semget() для масиву семафорів, які повертає значення IPC-дескриптора для цієї черги. При цьому існують ті ж способи створення й доступу, що й для поділюваної пам'яті або семафорів (розділ «Поділювана пам'ять у UNIX. Системні виклики shmget(), shmat(), shmdt()» і див. розділ «Створення масиву семафорів або доступ до вже існуючого. Системний виклик semget()», відповідно).

Системний виклик msgget()

Прототип системного виклику

```
#include <types.h>
#include <ipc.h>
#include <msg.h>
int msgget(key_t key, int msgflg);
```

Опис системного виклику

Системний виклик `msgget` призначений для виконання операції доступу до черги повідомлень і, у випадку її успішного завершення, повертає дескриптор System V IPC для цієї черги (ціле ненегативне число, яке однозначно характеризує черга повідомлень усередині обчислювальної системи і використовувані надалі для інших операцій з нею).

Параметр `key` є ключем System V IPC для черги повідомлень, тобто фактично її ім'ям із простору імен System V IPC. У якості значення цього параметра може бути використане значення ключа, отримане за допомогою функції `ftok()`, або спеціальне значення `IPC_PRIVATE`. Використання значення `IPC_PRIVATE` завжди приводить до спроби створення нової черги повідомлень із ключем, який не збігається зі значенням ключа ні однієї з вже існуючих черг і не може бути отриманий за допомогою функції `ftok()` ні при одній комбінації її параметрів.

Параметр `msgflg` – прапорець – відіграє роль тільки при створенні нової черги повідомлень і визначає права різних користувачів при доступі до черги, а також необхідність створення нової черги та поводження системного виклику при спробі створення. Він є деякою комбінацією (за допомогою операції побітове або – «|») наступних визначених значень і вісімкових прав доступу:

- `IPC_CREAT` — якщо черги для зазначеного ключа не існує, вона повинна бути створена;
- `IPC_EXCL` — застосовується разом із прапорцем `IPC_CREAT`. При спільному їх використанні й існуванні масиву з зазначеним ключем доступ до черги не виробляється та констатується помилкова ситуація, при цьому змінна `errno`, описана у файлі `<errno.h>`, прийме значення `EEXIST`;
- `0400` — дозволене читання для користувача, який створив чергу;
- `0200` — дозволений запис для користувача, який створив чергу;
- `0040` — дозволене читання для групи користувача, який

створив чергу;

- 0020 — дозволений запис для групи користувача, який створив чергу;
- 0004 — дозволене читання для всіх інших користувачів;
- 0002 — дозволений запис для всіх інших користувачів;

Черга повідомлень має обмеження за загальною кількостю збереженої інформації, яка може бути змінена адміністратором системи. Поточне значення обмеження можна довідатися за допомогою команди

```
ipcs -l
```

Значення, яке повертається

Системний виклик повертає значення дескриптора System V IPC для черги повідомлень при нормальному завершенні та значення -1 при виникненні помилки.

Реалізація примітивів send і receive. Системні виклики msgsnd() і msgrcv()

Для виконання примітива send використовується системний виклик msgsnd(), який копіює користувацьке повідомлення до черги повідомлень, заданої IPC-дескриптором. При вивченні опису цього виклику звернете особливу увагу на наступні моменти:

- Тип даних struct msgbuf не є типом даних для користувацьких повідомлень, а являє собою лише шаблон для створення таких типів. Користувач сам повинний створити структуру для своїх повідомлень, у якій першим полем повинна бути змінна типу long, що містить позитивне значення типу повідомлення.
- У якості третього параметру – довжини повідомлення – вказується не вся довжина структури даних, яка відповідає повідомленню, а тільки довжина корисної інформації, тобто інформації, яка розташовується в структурі даних після типу повідомлення. Це значення може бути рівним 0 у випадку, коли вся корисна інформація полягає в самому факті приходу повідомлення (повідомлення використовується У якості сигнальний засіб зв'язку).
- У практичних заняттях ми, як правило, будемо використовувати нульове значення прапорця системного виклику, яке приводить до блокування процесу при відсутності вільного місця в черзі повідомлень.

Системний виклик msgsnd()

Прототип системного виклику

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, struct msgbuf *ptr,
int length, int flag);
```

Опис системного виклику

Системний виклик `msgsnd` призначений для приміщення повідомлення до черги повідомлень, тобто є реалізацією примітива send.

Параметр `msqid` є дескриптором System V IPC для черги, до якої відправляється повідомлення, тобто значенням, яке повернув системний виклик `msgget()` при створенні черги або при її пошуку за ключем.

Структура `struct msgbuf` описана у файлі `<sys/msg.h>` як

```
struct msgbuf {
long mtype;
char mtext[1];
};
```

Вона має з себе деякий шаблон структури повідомлення користувача. Повідомлення користувача – це структура, перший елемент якої обов'язково має тип long і містить тип повідомлення, а далі йде інформативна частина теоретично довільної довжини (практично в Linux вона обмежена розміром 4080 байт і може бути ще зменшена системним адміністратором), який містить власне суть повідомлення. Наприклад:

```
struct mymsgbuf {
long mtype;
char mtext[1024];
} mybuf;
```

При цьому інформація зовсім не зобов'язана бути текстовою, наприклад:

```
struct mymsgbuf {
long mtype;
struct {
int iinfo;
float finfo;
} info;
} mybuf;
```

Тип повідомлення повинний бути строго позитивним числом. Дійсна довжина корисної частини інформації (тобто інформації, розташованої у структурі після типу повідомлення) повинна бути передана системному викликові у якості параметра `length`. Цей параметр може дорівнювати і `0`, якщо вся корисна інформація полягає в самому факті наявності повідомлення. Системний виклик копіює повідомлення, розташоване за адресою, на який указує параметр `ptr`, у чергу повідомлень, задану дескриптором `msqid`.

Параметр `flag` може приймати два значення: `0` і `IPC_NOWAIT`. Якщо значення прапорця дорівнює `0`, і в черзі не вистачає місця для того, щоб помістити повідомлення, то системний виклик блокується доти, поки не звільниться місце. При значенні прапорця `IPC_NOWAIT` системний виклик у цій ситуації не блокується, а констатує виникнення помилки з установленням значення змінної `errno`, описаної у файлі `<errno.h>`, рівним `EAGAIN`.

Значення, яке повертається

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

Примітив `receive` реалізується системним викликом msgrcv(). При вивченні опису цього виклику потрібно звернути особливу увагу на наступні моменти:

- Тип даних `struct msgbuf`, як для виклику `msgsnd()`, є лише шаблоном для користувацького типу даних.
- Спосіб вибору повідомлення (див. розділ «Черги повідомлень у UNIX у якості складової частини System V IPC» поточного практичного заняття) задається нульовим, позитивним або негативним значенням параметра `type`. Точне значення типу обраного повідомлення можна визначити з відповідного поля структури, у яку системний виклик скопіює повідомлення.
- Системний виклик повертає довжину тільки корисної частини скопійованої інформації, тобто інформації, розташованої в структурі після поля типу повідомлення.
- Обране повідомлення віддаляється з черги повідомлень.
- У якості параметра `length` вказується максимальна довжина корисної частини інформації, яка може бути розміщена в структурі, адресованої параметром `ptr`.

- У матеріалах надалі ми будемо, як правило, користуватися нульовим значенням прапорців для системного виклику, яке приводить до блокування процесу у випадку відсутності в черзі повідомлень із запитаним типом і до помилкової ситуації у випадку, коли довжина інформативної частини обраного повідомлення перевищує довжину, специфіковану в параметрі `length`.

Системний виклик msgrcv()

Прототип системного виклику

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgrcv(int msqid, struct msgbuf *ptr,
int length, long type, int flag);
```

Опис системного виклику

Системний виклик `msgrcv` призначений для отримання повідомлення з черги повідомлень, тобто є реалізацією примітива receive.

Спосіб вибірки	Значення параметра type
У порядку FIFO, незалежно від типу повідомлення	0
У порядку FIFO для повідомлень з типом `n`	n
Першим вибирається повідомлення з мінімальним типом, який не перевищує значення `n`, що прийшло раніше всіх інших повідомлень з тим же типом	-n

Параметр msqid є дескриптором System V IPC для черги, з якої повинне бути отримане повідомлення, тобто значенням, що повернув системний виклик `msgget()` при створенні черги або при її пошуку за ключем.

Параметр `type` визначає спосіб вибірки повідомлення з черги в такий спосіб

Структура `struct msgbuf` описана у файлі `<sys/msg.h>` як

```
struct msgbuf {
long mtype;
```

```
 char mtext[1];
 };
```
Вона являє собою деякий шаблон структури повідомлення користувача. Повідомлення користувача – це структура, перший елемент якої обов'язково має тип `long` і містить тип повідомлення, а далі йде інформативна частина теоретично довільної довжини (практично в Linux вона обмежена розміром 4080 байт і може бути ще зменшена системним адміністратором), яка містить власне суть повідомлення. Наприклад:

```
 struct mymsgbuf {
 long mtype;
 char mtext[1024];
 } mybuf;
```
При цьому інформація зовсім не зобов'язана бути текстовою, наприклад:

```
 struct mymsgbuf {
 long mtype;
 struct {
 int iinfo;
 float finfo;
 } info;
 } mybuf;
```
Параметр `length` повинний мати максимальну довжину корисної частини інформації (тобто інформації, розташованої в структурі після типу повідомлення), яка може бути розміщена в повідомленні.

У випадку удачі системний виклик копіює обране повідомлення з черги повідомлень за адресою, зазначеному в параметрі `ptr`, одночасно видаляючи його з черги повідомлень.

Параметр `flag` може приймати значення `0` або бути якою-небудь комбінацією прапорців `IPC_NOWAIT` і `MSG_NOERROR`. Якщо прапорець `IPC_NOWAIT` не встановлений і черга повідомлень порожня або в ній немає повідомлень із замовленим типом, то системний виклик блокується до появи запитаного повідомлення. При встановленні прапорця `IPC_NOWAIT` системний виклик у цій ситуації не блокується, а констатує виникнення помилки з установленням значення змінної `errno`, описаної у файлі `<errno.h>`, рівним `EAGAIN`. Якщо дійсна довжина корисної частини інформації в обраному повідомленні перевищує значення, зазначене в параметрі `length` і прапорець `MSG_NOERROR` не встановлений, то вибірка повідомлення не формується, і фіксується наявність помилкової ситуації. Якщо

прапорець MSG_NOERROR установлений, то в цьому випадку помилка не виникає, а повідомлення копіюється в скороченому вигляді.

Значення, яке повертається

Системний виклик повертає при нормальному завершенні дійсну довжину корисної частини інформації (тобто інформації, розташованої в структурі після типу повідомлення), скопійованої з черги повідомлень, і значення −1 при виникненні помилки.

Максимально можлива довжина інформативної частини повідомлення в операційній системі Linux складає 4080 байт і може бути зменшена при генерації системи. Поточне значення максимальної довжини можна визначити за допомогою команди

```
ipcs -l
```

Видалення черги повідомлень із системи за допомогою команди ipcrm або системного виклику msgctl()

Після завершення процесів, які використовували чергу повідомлень, вона не віддаляється із системи автоматично, а продовжує зберігатися в системі разом із усіма незатребуваними повідомленнями доти, поки не буде виконані спеціальна команда або спеціальний системний виклик. Для видалення черги повідомлень можна скористатися вже знайомою нам командою ipcrm, яка в цьому випадку прийме вид:

```
ipcrm msg <IPC ідентифікатор>
```

Для отримання IPC ідентифікатора черги повідомлень застосуйте команду ipcs. Можна видалити чергу повідомлень і за допомогою системного виклику msgctl(). Цей виклик уміє виконувати й інші операції над чергою повідомлень, але в рамках даного курсу ми них розглядати не будемо. Якщо будь-який процес знаходився в стані **чекання** при виконанні системного виклику msgrcv() або msgsnd() для черги, яка видаляється, то він буде разблокированій, і системний виклик констатує наявність помилкової ситуації.

Системний виклик msgctl()

Прототип системного виклику

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd,
           struct msqid_ds *buf);
```

Опис системного виклику

Системний виклик `msgctl` призначений для отримання інформації про чергу повідомлень, зміни її атрибутів і видалення із системи. Даний опис не є повним описом системного виклику, а обмежується межами цього курсу. Для вивчення повного опису зверніться до UNIX Manual.

У нашому курсі ми будемо користуватися системним викликом `msgctl` тільки для видалення черги повідомлень із системи. Параметр `msqid` є дескриптором System V IPC для черги повідомлень, тобто значенням, яке повернув системний виклик `msgget()` при створенні черги або при її пошуку за ключем.

У якості параметр `cmd` у межах нашого курсу ми завжди будемо передавати значення `IPC_RMID` – команду для видалення черги повідомлень із заданим ідентифікатором. Параметр `buf` для цієї команди не використовується, тому ми завжди будемо підставляти туди значення `NULL`.

Значення, яке повертається

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

Прогін приклада з односпрямованою передачею текстової інформації

Для ілюстрації розглянемо дві прості програми.

```
/* Програма 05-5a.c для ілюстрації роботи з
чергами повідомлень */
/* Ця програма отримує доступ до черги
повідомлень, відправляє до неї 5 текстових
повідомлень з типом 1 і одне порожнє повідомлення
з типом 255, яке буде служити для програми 05-5b.c
сигналом припинення роботи. */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <stdio.h>
#define LAST_MESSAGE 255 /* Тип повідомлення для
    припинення роботи програми 05-5b.c */
int main()
{
```

```
        int msqid; /* IPC дескриптор для черги
повідомлень */
char pathname[] = "05-5a.c"; /* Ім'я файлу,
         який використовується для генерації ключа.
Файл із таким ім'ям повинний існувати в поточній
директорії */
    key_t key; /* IPC ключ */
    int i,len; /* Лічильник циклу та довжина
        інформативної частини повідомлення */
     /* Нижче випливає користувацька структура для
повідомлення */
    struct mymsgbuf
    {
        long mtype;
        char mtext[81];
    } mybuf;
     /* Генеруємо IPC ключ з імені файлу 05-5a.c у
поточній директорії та номер екземпляра черги
повідомлень 0. */
    if((key = ftok(pathname,0)) < 0){
        printf("Can\'t generate key\n");
        exit(-1);
    }
      /* Намагаємося отримати доступ за ключем до
черги повідомлень, якщо вона існує, або створити
її, із правами доступу
    read & write для всіх користувачів */
     if((msqid = msgget(key, 0666 | IPC_CREAT)) <
0){
        printf("Can\'t get msqid\n");
        exit(-1);
    }
    /* Посилаємо в циклі 5 повідомлень з типом 1
    до черги повідомлень, ідентифіковану msqid.*/
    for (i = 1; i <= 5; i++){
             /* Спочатку заповнюємо структуру для
нашого
               повідомлення та визначаємо довжину
інформативної частини */
        mybuf.mtype = 1;
               strcpy(mybuf.mtext, "This is text
message");
        len = strlen(mybuf.mtext)+1;
```

```
            /* Відсилаємо повідомлення. У випадку
помилки повідомляємо про це та видаляємо чергу
повідомлень із системи. */
             if (msgsnd(msqid, (struct msgbuf *)
&mybuf,
            len, 0) < 0){
                  printf("Can\'t send message to
queue\n");
            msgctl(msqid, IPC_RMID,
                (struct msqid_ds *) NULL);
            exit(-1);
        }
    }
        /* Відсилаємо повідомлення, яке змусить
процес, який його отримує, припинити роботу, з
типом LAST_MESSAGE і довжиною 0 */
    mybuf.mtype = LAST_MESSAGE;
    len = 0;
    if (msgsnd(msqid, (struct msgbuf *) &mybuf,
        len, 0) < 0){
        printf("Can\'t send message to queue\n");
        msgctl(msqid, IPC_RMID,
            (struct msqid_ds *) NULL);
        exit(-1);
    }
    return 0;
}
```

Лістінг 5.5a. Програма 05-5a.c для ілюстрації роботи з чергами повідомлень.

```
/* Програма 05-5b.c для ілюстрації роботи з
чергами повідомлень */
/* Ця програма отримує доступ до черги повідомлень
і читає з неї повідомлення з будь-яким типом у
порядку FIFO доти, поки не отримає повідомлення з
типом 255, яке буде служити сигналом припинення
роботи. */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
```

```
#include <stdio.h>
#define LAST_MESSAGE 255 /* Тип повідомлення для
    припинення роботи */
int main()
{
       int msqid; /* IPC дескриптор для черги
повідомлень */
    char pathname[] = "05-5a.c"; /* Ім'я файлу,
          що використовується для генерації ключа.
Файл із таким ім'ям повинний існувати в поточній
директорії */
    key_t key; /* IPC ключ */
       int len, maxlen; /* Реальна довжина та
максимальна
         довжина інформативної частини повідомлення
*/
     /* Нижче випливає користувацька структура для
повідомлення */
    struct mymsgbuf
    {
        long mtype;
        char mtext[81];
    } mybuf;
     /* Генеруємо IPC ключ з імені файлу 05-5a.c у
поточній директорії та номери екземпляра черги
повідомлень 0 */
    if((key = ftok(pathname,0)) < 0){
        printf("Can\'t generate key\n");
        exit(-1);
    }
      /* Намагаємося отримати доступ за ключем до
черги повідомлень, якщо вона існує, або створити
її, із правами доступу
    read & write для всіх користувачів */
     if((msqid = msgget(key, 0666 | IPC_CREAT)) <
0){
        printf("Can\'t get msqid\n");
        exit(-1);
    }
    while(1){
              /* У нескінченному циклі приймаємо
повідомлення
```

```
будь-якого типу в порядку FIFO з максимальною
довжиною інформативної частини 81 символ доти,
поки не надійде повідомлення з типом
LAST_MESSAGE*/
        maxlen = 81;
        if(( len = msgrcv(msqid,
              (struct msgbuf *) &mybuf, maxlen, 0,
0) < 0){
               printf("Can\'t receive message from
queue\n");
            exit(-1);
        }
           /* Якщо прийняте повідомлення має тип
LAST_MESSAGE, припиняємо роботу та видаляємо чергу
повідомлень з системи. У противному випадку
друкуємо текст прийнятого повідомлення. */
        if (mybuf.mtype == LAST_MESSAGE){
            msgctl(msqid, IPC_RMID,
             (struct msqid_ds *) NULL);
        exit(0);
        }
        printf("message type = %ld, info = %s\n",
        mybuf.mtype, mybuf.mtext);
    }
    return 0; /* Винятково для відсутності
        warning'ов при компіляції. */
}
```

Лістінг 9.1b. Програма 05-5b.c для ілюстрації роботи з чергами повідомлень.

Перша з цих програм посилає п'ять текстових повідомлень з типом 1 й одне повідомлення нульової довжини з типом 255 другій програмі. Друга програма в циклі приймає повідомлення будь-якого типу в порядку FIFO і друкує їх зміст доти, поки не отримає повідомлення з типом 255. Повідомлення з типом 255 служить для неї сигналом до завершення роботи та ліквідації черги повідомлень. Якщо перед запуском кожної з програм черга повідомлень ще була відсутня в системі, то програма створить її.

Зверніть увагу на використання повідомлення з типом 255 У якості сигнал припинення роботи другого процесу. Це повідомлення має

нульову довжину, тому що його інформативність вичерпується самим фактом наявності повідомлення.

ЗАВДАННЯ 9: Наберіть програми, збережете під іменами 05-5a.c і 05-5b.c відповідно, відкомпілюйте та перевірте правильність їхнього поводження.

Модифікація попереднього приклада для передачі числової інформації

В описі системних викликів msgsnd() і msgrcv() говориться про те, що передана інформації не обов'язково повинна являти собою текст.

Ми можемо скористатися `чергами повідомлень` для передачі даних будь-якого вигляду. При передачі різнорідної інформації доцільно інформативну частину поєднувати усередині повідомлення в окрему структуру:

```
struct mymsgbuf {
     long mtype;
     struct {
          short sinfo;
          float finfo;
     } info;
} mybuf;
```

для правильного обчислення довжини інформативної частини. У деяких обчислювальних системах числові дані розміщаються в пам'яті з вирівнюванням на визначені адреси (наприклад, на адреси, кратні 4). Тому реальний розмір пам'яті, необхідної для розміщення декількох числових даних, може виявитися більшим за суму довжин цих даних, тобто в нашому випадку

```
sizeof(info)>=sizeof(short)+sizeof(float)
```

Для повної передачі інформативної частини повідомлення у якості довжини потрібно вказувати не суму довжин полів, а повну довжину структури.

ЗАВДАННЯ 10: Модифікуйте попередні програми 05-5a.c і 05-5b.c з розділу «Прогін приклада з односпрямованою передачею текстової інформації» для передачі нетекстових повідомлень.

Написання, компіляція та прогін програм для здійснення двостороннього зв'язку через одну чергу повідомлень

Наявність у повідомлень типів дозволяє організувати двосторонній зв'язок між процесами через ту саму чергу повідомлень. Процес 1 може посилати процесу 2 повідомлення з типом 1, а отримувати від нього повідомлення з типом 2. При цьому для `вибірки повідомлень` в обох процесах варто користуватися другим способом вибору (див. розділ «Черги повідомлень у UNIX у якості складової частини System V IPC»).

ЗАВДАННЯ 11: Напишіть, відкомпілюйте та виконайте програми, що здійснюють двосторонній зв'язок через одну чергу повідомлень.

Поняття мультиплексування. Мультиплексування повідомлень. Модель взаємодії процесів клієнт-сервер. Нерівноправність клієнта та сервера

Використовуючи техніку з попереднього прикладу, ми можемо організувати отримання повідомлень одним процесом від безлічі інших процесів через одну чергу повідомлень і відправлення їм відповідей через ту ж чергу повідомлень, тобто здійснити мультиплексування повідомлень. Узагалі під мультиплексування інформації розуміють можливість одночасного обміну інформацією з декількома партнерами. Метод мультиплексування широко застосовується в моделі взаємодії процесів клієнт-сервер. У цій моделі один із процесів є сервером. Сервер отримує запити від інших процесів – клієнтів – на виконання деяких дій і відправляє їм результати обробки запитів. Найчастіше модель клієнт-сервер використовується при розробці мережних додатків, з якими ми зіштовхнемося в матеріалах курсу „Мережеві операційні системи". Вона споконвічно припускає, що взаємодіючі процеси нерівноправні:

- Сервер, як правило, працює постійно, на всьому протязі життя додатка, а клієнти можуть працювати епізодично.
- Сервер чекає запиту від клієнтів, ініціатором же взаємодії є клієнт.
- У якості правило, клієнт звертається до одного сервера за раз, у той час У якості до сервера можуть одночасно надходити запити від декількох клієнтів.
- Клієнт повинний знати, як звернутися до сервера (наприклад, якого типу повідомлення він сприймає) перед початком організації запиту до сервера, у той час як сервер може отримати відсутню інформацію про клієнта з запиту, що прийшов.

Розглянемо наступну схему мультиплексування повідомлень через одну чергу повідомлень для моделі клієнт-сервер. Нехай сервер отримує з черги повідомлень тільки повідомлення з типом 1. До складу повідомлень з типом 1, які посилаються серверу, процеси-клієнти включають значення своїх ідентифікаторів процесу. Прийнявши повідомлення з типом 1, сервер аналізує його зміст, виявляє ідентифікатор процесу, який надіслав запит, і відповідає клієнтові, посилаючи повідомлення з типом, рівним ідентифікаторові процесу, який запитував. Процес-клієнт після відправлення запиту очікує відповіді у вигляді повідомлення з типом, рівним своєму ідентифікаторові. Оскільки ідентифікатори процесів у системі різні, і жоден користувацький процес не може мати PID рівний 1, усі повідомлення можуть бути прочитані тільки тими процесами, яким вони адресовані. Якщо обробка запиту забирає тривалий час, сервер може організовувати рівнобіжну обробку запитів, породжуючи для кожного запиту нова процес-дитина або нова нитка виконання.

Написання, компіляція та прогін програм клієнта і сервера

ЗАВДАННЯ 12: Напишіть, відкомпілюйте та виконайте програми сервера і клієнтів для запропонованої схеми мультиплексування повідомлень.

Використання черг повідомлень для синхронізації роботи процесів

У лекції 6 була доведена еквівалентність черг повідомлень і семафорів у системах, де процеси можуть використовувати поділювану пам'ять. Зокрема, було показано, У якості реалізувати семафори за допомогою черг повідомлень. Для цього вводився спеціальний синхронізуючий процес-сервер, що обслуговує змінн-лічильники для кожного семафора. Процеси-клієнти для виконання операції над семафором посилали процесу-серверові запити на виконання операції й очікували відповіді для продовження роботи. Тепер ми знаємо, У якості це можна зробити в операційній системі UNIX і як, отже, можна використовувати `черги повідомлень` для організації взаимовиключенн і взаємної синхронізації процесів.

ЗАВДАННЯ 13: (підвищеної складності) реалізуйте семафори через черги повідомлень.