

Лабораторна робота №12

Тема: «Програмування міжпроцесної взаємодії з використанням каналів, черг повідомлень та програмування багатопоточної взаємодії»

Викладач: Олександр А. Блажко,

доцент кафедри ІС Одеської політехніки, blazhko@ieee.org

Мета роботи: отримати навички обміну даними між процесами за допомогою іменованих каналів, черг повідомлень, а також отримати навички створення потоків.

1 Теоретичні відомості

1.1 Особливості міжпроцесної взаємодії

Взаємодія між процесами (*Inter-Process Communication, IPC*) – набір засобів обміну даними (повідомленнями) між процесами.

Обмін даними між процесами може бути синхронним або асинхронним.

Синхронізація (др.-грец. слова «разом» + «час») – встановлення та підтримання тимчасових співвідношень між двома і більше системами, які беруть участь у передачі цифрових даних. Основним недоліком синхронної передачі даних є можливість виникнення ситуації, коли приймач і передавач даних працюють не в такт (частота формування сигналу в канал зв'язку не збігається з частотою опитування даних на приймальній стороні), що призводить до виникнення помилок у прийнятих даних.

Синхронний обмін даними базується на синхронізації дій між процесами.

Асинхронний обмін даними є протилежним синхронному, коли приймач і передавач даних працюють не в такт, але це не призводить до помилок, бо враховується системою.

В межах одного комп'ютера під управлінням однієї ОС використовують наступні засоби взаємодії через обмін даними:

- файли (*files*) для обміну даними між процесами через загальний ресурс, один процес записує у файл, інший процес – читає з файлу;
- сигнали (*signals*) для асинхронного обміну неочікуваними повідомленнями, які не передають дані між процесами, а тільки сповіщають їх про якусь подію, коли процес має відреагувати на неї виконанням наперед визначеної дії (функції або команди);
- неіменовані канали (*anonymous pipes, unnamed pipes*) у вигляді програмного конвеєру для синхронного обміну повідомленнями між процесами-родичами з використанням ланцюжку *Stdin/Stdout*-потоків операцій запису у файл та читання з файлу;

- іменовані канали (*named pipes, FIFO-pipes*) для синхронного обміну очікуваних повідомлень між будь-якими процесами, а не тільки між процесами-родичами, з використанням ланцюжку *Stdin/Stdout* потоків операцій запису у файл та читання з файлу, коли відправлення повідомлення відбувається подібно до операції запису в файл, а отримання повідомлення – подібно до читання даних з файлу, а якщо канал порожній – процес, який очікує дані, призупиняється до надходження даних в канал;
- черги повідомлень (*message queues*) – пакети даних, які асинхронно передаються між процесами з повідомленням отримувача про надходження пакету;
- сегменти загальної, колективної пам'яті (*shared memory*) – засіб, що дозволяє кільком процесам сумісно використовувати (поділяти) фрагмент оперативної пам'яті з метою обміну даними; відправлення даних відбувається шляхом запису в пам'ять, отримання – читанням з пам'яті.

1.2 Особливості роботи неіменованих та іменованих каналів

1.2.1 Особливості роботи з неіменованими каналами

В *Unix*-подібних ОС обмін даними між спорідненими процесами (*parent*-процес та *child*-процес) відбувається через так звані неіменовані канали (*unnamed pipes*), які забезпечують передачу даних так, що вихідний потік кожного процесу (*stdout*) безпосередньо з'єднується зі стандартним потоком вводу (*stdin*) наступного. В цьому випадку імена каналів створюються автоматично самою ОС, про що користувач або процес, який їх створює, нічого не знає.

Важливою особливістю реалізації каналів в ОС *Unix* є застосування буферизації під час передачі даних. Завдяки буферизації, записування та зчитування даних у каналі може відбуватись без звернення до зовнішніх пристроїв та з різною швидкістю без втрати даних.

Канал використовується для зв'язку між двома процесами за протоколом взаємодії:

- перший процес ініціалізує канал, викликаючи функцію *fork* та створюючи свою копію – *child*-процес;
- *parent*-процес закриває відкритий для читання кінець каналу, а *child*-процес, у свою чергу, закриває відкритий на запис кінець каналу.

Відомим прикладом використання неіменованого каналу є конвеєрна обробка даних, коли результати обробки, отримані в одному процесі (команді), передаються в інший, використовуючи наступну команду конвеєризації, коли інтерпретатор підключає відкритий для читання кінець кожного каналу до стандартного потоку *Stdin*, а відкритий на запис – до стандартного потоку *Stdout*.

1.2.2 Програмна реалізація неіменованих каналів

Неіменований канал програмно створюється функцією *pipe* і надає можливість однонаправленої передачі даних між двома процесами.

На відміну від функції створення звичайного файлу, яка створює один дескриптор файлу, функція *pipe* створює зразу два дескриптори:

- *fd[0]* – дескриптор файлу, відкритий для читання (*Stdin*);
- *fd[1]* – дескриптор файлу, відкритий для запису (*Stdout*).

Доступ до даних у каналі, як у звичайному файлі, відбувається за допомогою:

- стандартної функції *read* читання з файлу;
- стандартної функції *write* запису у файл.

На рисунку 1 наведено приклад програми *pipe_write*, яка створює неіменований канал між двома процесами: *parent*-процесом та *child*-процесом.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main () {
    int data_processed;
    int file_pipes[2];
    const char message[] = "Laboratory work of Błazhko";
    char buffer[BUFSIZ + 1];
    pid_t pid;
    char* program_args[] = {"read_data",buffer,NULL};
    memset(buffer, '\0', sizeof(buffer));
    if (pipe(file_pipes) == 0) {
        pid = fork();
        if (pid == 0) {
            sprintf(buffer,"%d", file_pipes[0]);
            execv("./pipe_read", program_args);
        }
        else {
            data_processed = write(file_pipes[1],message,strlen(message));
            printf("I'm parent with PID=%d.\n", getpid());
            printf("StdIn fd[0]=%d, Stdout fd[1]=%d\n",
                file_pipes[0],file_pipes[1]);
            printf("I wrote %d bytes of message: %s.\n",
                data_processed,message);
            sleep(1);
        }
    }
    return EXIT_SUCCESS;
}
```

Рис. 1 - Приклад програми *pipe_write*

Нижче прокоментовано окремі рядки програми:

1) масив *file_pipes* містить два дескриптори, пов'язані з потоками *Stdin* та *Stdout*, які в подальшому будуть ініціалізовані викликом функції *pipe*;

- 2) для роботи з функціями *read* та *write* необхідно ініціалізувати буфер даних *buffer*, розмір якого зберігається у константі *BUFSIZ* з файлу *stdio.h*, та, найчастіше = 8192 байт;
- 3) для очистки буферу даних *buffer* кодом `\0` використовується функція *memset*;
- 4) після успішного створення каналу функцією *pipe* створюється *child*-процес;
- 5) функція *sprintf* зчитує з дескриптору *file_pipes[0]* дані до буферу;
- 6) *child*-процес через функцію *execv* завантажує програмний образ з програмою *read_data* з аргументом *buffer*, визначеним у масиві *program_args*;
- 7) паралельно *parent*-процес починає запис даних повідомлення через функцію *write* у файл з дескриптором *file_pipes[1]*.

На рисунку 2 наведено приклад програми *pipe_read*.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char *argv[]) {
    int data_processed;
    char buffer[BUFSIZ + 1];
    int file_descriptor;
    memset(buffer, '\0', sizeof(buffer));
    sscanf(argv[1], "%d", &file_descriptor);
    data_processed = read(file_descriptor, buffer, BUFSIZ);
    printf("I'm child with PID=%d.\n", getpid());
    printf("StdIn fd[0]=%d\n", file_descriptor);
    printf("I read %d bytes: %s. \n", data_processed, buffer);
    return EXIT_SUCCESS;
}
```

Рис. 2 – Приклад програми *pipe_read* для отримання даних з неіменованого каналу

Нижче прокоментовано окремі рядки програми:

- 1) функція *sscanf* отримує буфер, який є першим елементом масиву *argv* при запуску програми *read_data* з програми *write_data*;
- 2) результат змінює посилання на файловий дескриптор *file_descriptor*;
- 3) функція *read* зчитує *count* байт з файлу, визначеного файлового дескриптору, у буфер *buffer*, та повертає кількість зчитаних байт;
- 4) значення буферу містить отримане повідомлення з неіменованого каналу.

На рисунку 3 наведено результат виконання програми *pipe_write*.

```
[blazhko@vps61efe OS-Labwork12-Examples]$ ./pipe_write
I'm parent with PID=2140255.
StdIn fd[0]=3, Stdout fd[1]=4
I wrote 26 bytes of message: Laboratory work of Blazhko.
I'm child with PID=2140256.
StdIn fd[0]=3
I read 26 bytes: Laboratory work of Blazhko.
```

Рис. 3 – Результат виконання програми *pipe_write*

1.2.3 Робота з іменованими каналами через інтерпретатор командного рядку

В *UNIX System III* (1982 рік) було додано канали *FIFO*, які називаються *іменованими каналами (named pipes)*. Іменовані канали схожі за поведінкою на неіменовані канали, але забезпечують обмін даними між неспорідненими процесами. В *Unix*-подібних ОС на відміну від неіменованих каналів іменований канал постійно зберігається у файловій системі як спеціальний файл *pipe*-типу, тому він має ім'я.

У командній оболонці *Bash* для створення файлів-каналів є команда *mkfifo*:

mkfifo канал

Наприклад, для створення каналу на ім'я *pipe1* виконується наступна команда:

mkfifo pipe1

Після створення каналу його файл відображається у переліку файлів каталога, в якому він був створений, з першою буквою *p* в атрибуті файлу, наприклад:

```
prw-r--r-- 1 user group          0 2010-11-17 01:13 pipe1
```

Особливістю файлу є те, що його розмір завжди = 0 байт.

Після створення файлу каналу можна створити процес, який буде записувати в канал. Наприклад, через перенаправлення виводу результату команди *ls*:

```
ls > pipe1
```

Цей процес буде "чекати", поки канал не прочитає передані йому дані, але канал сам ці дані читати не може. Для читання даних з каналу треба перейти в інший псевдотермінал та створити процес, який буде читати з каналу, наприклад:

```
cat pipe1
```

Наведемо приклад створення каналу, який буде стискати (архівувати) все, що туди потрапляє, використовуючи утиліту *gzip*:

```
mkfifo archive_me
```

```
gzip -c < archive_me > archive_file.gz
```

```
cat file.txt > archive_me
```

У файл *archive_file.gz* запишуться передані дані у заархівованому вигляді.

Для перевірки результатів архівування можна виконати команду:

```
gunzip -c archive_file.gz
```

Видаляти канали можна як і звичайні файли через команду *rm*:

```
rm pipe1
```

1.2.4 Програмна реалізація іменованих каналів

Для програмного створення іменованого каналу використовується функція:

```
int mkfifo(const char *pathname, mode_t mode);
```

Аргументи функції:

- *pathname* – назва каналу як повний шлях до файлу каналу;
- *mode* – права доступу, які визначаються за форматом, сумісним з форматом команди *chmod*.

Для відкриття каналу використовується стандартна функція *open* з відкриття файлу за вказаною назвою, співпадаючою з назвою каналу. Для читання з каналу використовується стандартна функція *read*. Для видалення каналу використовується стандартна функція *remove* з видалення файлу за вказаною назвою, співпадаючою з назвою каналу. На рисунку 4 наведено приклад програми *fifo* створення каналу з назвою */tmp/my_named_pipe*

```
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>

#define NAMEDPIPE_NAME "/tmp/my_named_pipe"
#define BUFSIZE 50

int main (int argc, char ** argv) {
    int fd, len;
    char buf[BUFSIZE];

    if ( mkfifo(NAMEDPIPE_NAME, 0777) ) {
        fprintf(stderr, "Error in mkfifo!");
        return 1;
    }
    printf("%s is created\n", NAMEDPIPE_NAME);

    if ( (fd = open(NAMEDPIPE_NAME, O_RDONLY)) <= 0 ) {
        fprintf(stderr, "Error in open!");
        return 1;
    }
    printf("%s is opened\n", NAMEDPIPE_NAME);

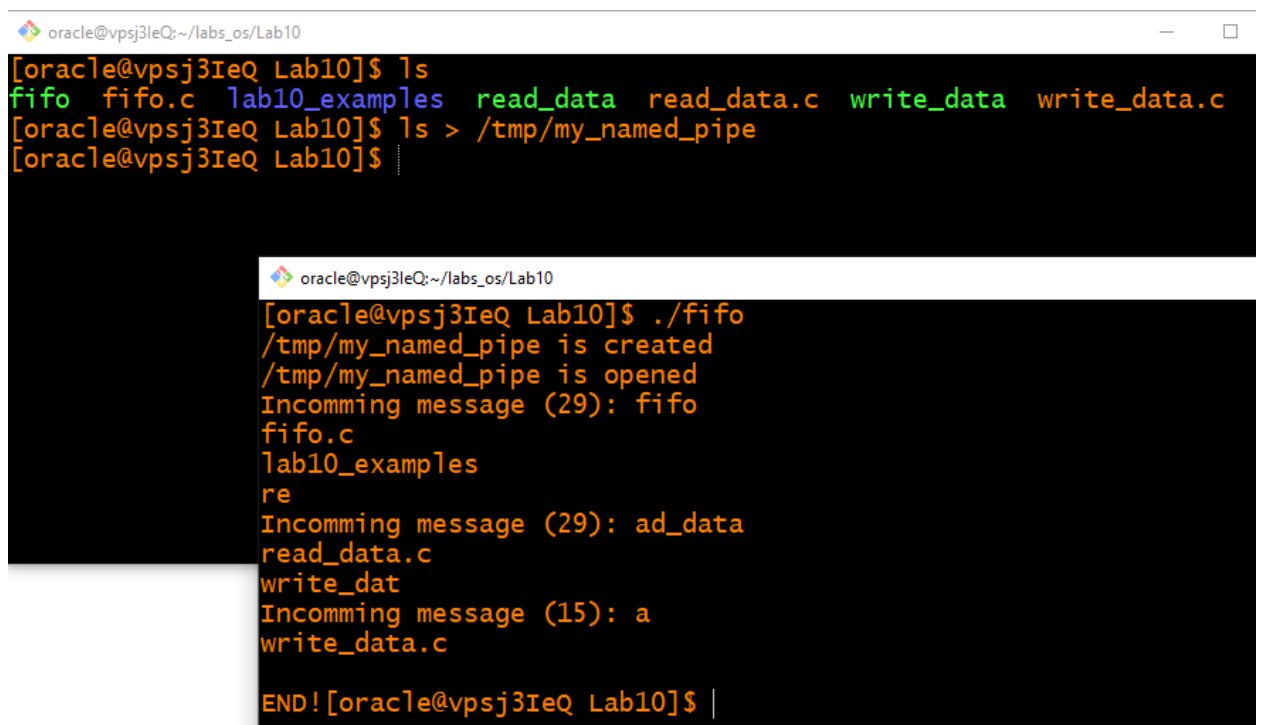
    do {
        memset(buf, '\0', BUFSIZE);
        if ( (len = read(fd, buf, BUFSIZE-1)) <= 0 ) {
            printf("END!");
            close(fd);
            remove(NAMEDPIPE_NAME);
            return 0;
        }
        printf("Incomming message (%d): %s\n", len, buf);
    } while ( 1 );
}
```

Рис. 4 – Приклад програми *fifo* створення іменованого каналу

На рисунку 5 наведено приклади виконання програми, яка створює іменованний канал в одному псевдотерміналі. В іншому псевдотерміналі виконується команда *ls*, яка передає результат своєї роботи в іменованний канал */tmp/my_named_pipe*

Як видно з рисунку 5:

- 1) на початку передачі даних через канал відкривається файл каналу, про що свідчить відповідне повідомлення;
- 2) після початку передачі даних через канал вони виводяться на екран у другому псевдотерміналі порціями по 29 байт у відповідності з розміром буферу в програмі *fifo*.
- 3) дані виводяться на екран у циклі поки не буде завершено їх передачу через канал;
- 4) коли дані у каналі завершуються, файл закривається функцією *close* та видаляється канал функцією *remove*.



```
oracle@vpsj3IeQ:~/labs_os/Lab10
[oracle@vpsj3IeQ Lab10]$ ls
fifo fifo.c lab10_examples read_data read_data.c write_data write_data.c
[oracle@vpsj3IeQ Lab10]$ ls > /tmp/my_named_pipe
[oracle@vpsj3IeQ Lab10]$

oracle@vpsj3IeQ:~/labs_os/Lab10
[oracle@vpsj3IeQ Lab10]$ ./fifo
/tmp/my_named_pipe is created
/tmp/my_named_pipe is opened
Incomming message (29): fifo
fifo.c
lab10_examples
re
Incomming message (29): ad_data
read_data.c
write_dat
Incomming message (15): a
write_data.c

END! [oracle@vpsj3IeQ Lab10]$
```

Рис. 5 – Фрагмент псевдотерміналів з виконанням програм, які використовують іменований канал

1.3 Керування чергами повідомлень між процесами

Механізм обміну повідомленнями між процесами на основі черг враховує наявність:

- процесу-постачальника, який передає повідомлення;
- процесу-споживача, який отримує повідомлення;
- черга повідомлень для їх зберігання доки їх не обробить споживач.

1.3.1 Використання черг повідомлень для процесів, які взаємодіють з однією БД під керуванням серверу СКБД *PostgreSQL*

СКБД *PostgreSQL* надає простий механізм міжпроцесної взаємодії для множини процесів як копій серверу СКБД, що працюють з однією БД, використовуючи команди *LISTEN/NOTIFY*, для очікування приходу повідомлень від процесів або пересилання процесам повідомлень (рисунок 6).

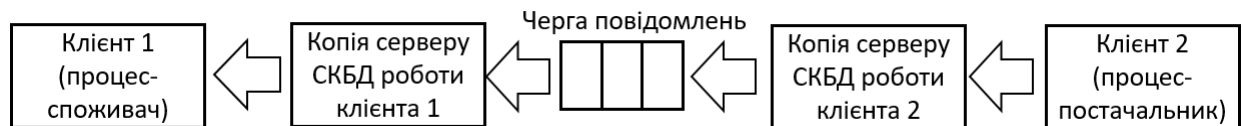


Рис. 6 – Особливості обміну повідомленнями між процесами-клієнтами через копії серверу СКБД *PostgreSQL*

Особливості механізму повідомлень *LISTEN/NOTIFY*:

- механізм повідомлень нагадує механізм сигналів як асинхронних повідомлень, що передаються зарезервованим каналом за вказаною назвою;
- через чергу повідомлень разом із повідомленням один серверний процес може передати рядок повідомлення, а додаткові дані можна передати через таблиці БД, які будуть зчитуватися іншим серверним процесом;
- повідомлення включає унікальну назву черги всередині однієї БД, *PID* серверного процесу, який надіслав повідомлення, та рядок повідомлення;
- при виборі назви черги повідомлення можна враховувати назву таблиці БД або будь-яку іншу інформацію, яка описує особливості міжпроцесної взаємодії.

Команда *LISTEN* починає процес очікування на отримання повідомлення, використовуючи синтаксис:

```
LISTEN channel_name
```

Наприклад:

```
LISTEN bills;
```

Команда *NOTIFY* надсилає повідомлення за допомогою синтаксису:

```
NOTIFY channel_name [, message]
```

Наприклад:

```
NOTIFY bills, 'Hello'
```

Коли викликається команда *NOTIFY* в поточному або іншому псевдотерміналі взаємодії з сервером СКБД, підключеному до тієї ж БД, всі копії-сервери, що очікують

повідомлення через вказану чергу повідомлень, отримують повідомлення і кожен сервер передає повідомлення підключеним до нього програмі-споживача, наприклад, утиліті *psql*.

Можна відмовитись від отримання повідомлень з черги за допомогою команди *UNLISTEN*, використовуючи синтаксис:

```
UNLISTEN channel_name | *
```

Спеціальний знак *** скасовує очікування від всіх каналів.

Якщо клієнт, який очікує повідомлення за вказаною чергою, сам виконує *NOTIFY* до цієї черги, тоді він отримає своє повідомлення, як і будь-який інший процес, що очікує повідомлення. Таку ситуацію можна контролювати, якщо перевірити, чи не збігається *PID* сигналізуючого процесу із власним *PID* процесу. Якщо вони збігаються, повідомлення можна ігнорувати.

Для отримання *PID* серверного процесу використовується функція *pg_backend_pid()*, яку можна використовувати у команді *SELECT*:

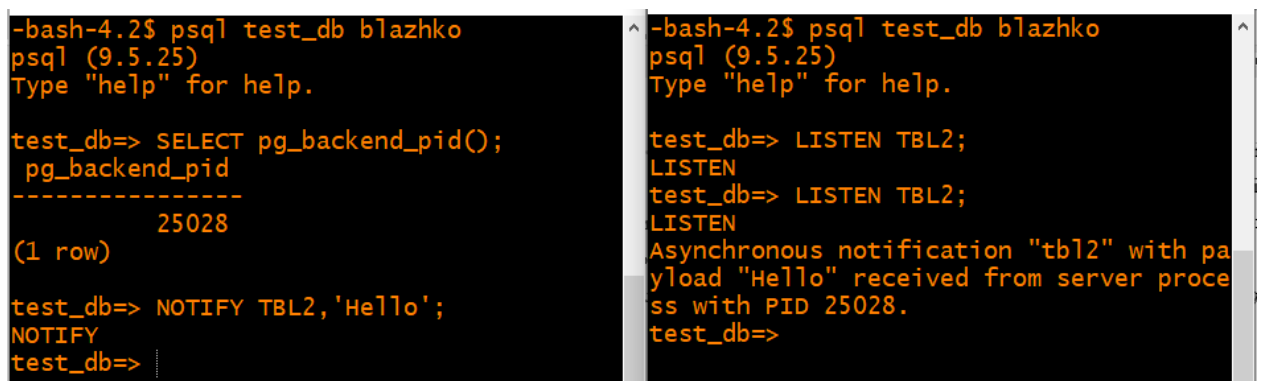
```
SELECT pg_backend_pid();
```

Список черг повідомлень можна отримати за допомогою функції *pg_listening_channels()*, наприклад:

```
SELECT pg_listening_channels();
```

На рисунку 7 наведено приклад виконання команд *LISTEN/NOTIFY*:

- у 1-му терміналі (зліва) виконується команда *SELECT pg_backend_pid()*;
- у 2-му терміналі виконується команда *LISTEN TBL2*;
- у 1-му терміналі виконується команда *NOTIFY TBL2, 'Hello'*
- у 2-му терміналі виконується команда *LISTEN TBL2*;
- у 2-му терміналі з'являється рядок з повідомленням від серверного процесу з *PID=25028*.



```
-bash-4.2$ psql test_db blazhko
psql (9.5.25)
Type "help" for help.

test_db=> SELECT pg_backend_pid();
 pg_backend_pid
-----
          25028
(1 row)

test_db=> NOTIFY TBL2,'Hello';
NOTIFY
test_db=>

^
-bash-4.2$ psql test_db blazhko
psql (9.5.25)
Type "help" for help.

test_db=> LISTEN TBL2;
LISTEN
test_db=> LISTEN TBL2;
LISTEN
Asynchronous notification "tbl2" with payload "Hello" received from server process with PID 25028.
test_db=>
```

Рис. 7 – Приклад виконання команд *LISTEN/NOTIFY*

1.3.2 Програмування черги повідомлень

Черги повідомлень з'явилися в ОС *AT&T UNIX System (UNIX System V)* як бібліотека функцій, основними з яких є функції:

- *msgget* – відкриття (створення) черги повідомлень;
- *msgsnd* – внесення повідомлення до черги;
- *msgrcv* – зчитування повідомлення з черги;
- *msgctl* – контроль черги повідомлень.

Для відкриття (створення) черги повідомлень використовується функція *msgget*:

```
int msgget(key_t key, int msgflg),
```

де *key* – унікальний ідентифікатор повідомлень;

msgflg – права доступу, які визначаються за форматом, сумісним з форматом команди *chmod*, враховуючи, що для створення нової черги повідомлень необхідно ці права об'єднати порозрядною операцією *OR* зі спеціальним бітом зі значенням *IPC_CREAT*, але якщо черга вже існує, тоді значення *IPC_CREAT* ігнорується.

Для внесення повідомлення до черги черги повідомлень використовується функція *msgsnd*:

```
int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz,  
int msgflg);
```

де *msqid* – унікальний ідентифікатор повідомлень, який було визначено функцією *msgget*;

msg_ptr – посилання на структуру повідомлення (показчик), прикладом якої може бути така : *struct my_msg_st { long int my_msg_type; char some_text[MAX_TEXT]; };*

msg_sz – довжина повідомлення;

int msgflg – флаг керування діями, які вживаються під час заповнення поточної черги повідомлень або досягненні загальносистемного обмеження для черг повідомлень, за замовчуванням = 0.

На рисунку 8 наведено приклад програми *msgsnd* для розміщення повідомлення в черзі повідомлень.

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/msg.h>
#define MAX_TEXT 512

struct my_msg_st {
    long int my_msg_type;
    char some_text[MAX_TEXT];
};

int main(void) {
    struct my_msg_st my_data;
    int msgid;
    char buffer[BUFSIZ] = "Message from Blazhko";
    msgid = msgget( (key_t)1234, 0777 | IPC_CREAT);
    if (msgid == -1) {
        fprintf(stderr, "msgget failed with error: %dn", errno);
        return EXIT_FAILURE;
    }
    my_data.my_msg_type = 1;
    strcpy(my_data.some_text, buffer);
    if (msgsnd(msgid, (void*)&my_data, MAX_TEXT, 0) == -1) {
        fprintf(stderr, "msgsnd failedn");
        return EXIT_FAILURE;
    }
    else
        printf("I sent message: %s.\n", my_data.some_text);
    return EXIT_SUCCESS;
}

```

Рис. 8 – Приклад програми *msgsnd* для розміщення повідомлення в черзі повідомлень

Для зчитування повідомлення з черги повідомлень використовується функція:

```
int msgrcv (int msqid, void * msg_ptr, size_t msg_sz, long
int msgtype, int msgflg);
```

де *msqid* – ідентифікатор черги повідомлень, який визначено функцією *msgget*;

msg_ptr – посилання на структуру повідомлення (покажчик), яка була описана раніше у функції *msgsnd*;

msg_sz – довжина повідомлення;

msgtype – прапорець визначення порядку отримання повідомлень, якщо значення *msgtype* = 0, тоді береться перше доступне повідомлення у черзі.

Для видалення черги повідомлень використовується функція *msgctl*, в якій значення другого аргументу = *IPC_RMID*, а значення третього аргументу = 0:

```
int msgctl(int msqid; int command, struct msqid_ds *buf);
```

На рисунку 9 наведено приклад програми *msgsnd* для отримання повідомлення з черги повідомлень.

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/msg.h>

struct my_msg_st {
    long int my_msg_type;
    char some_text[BUFSIZ];
};

int main(void) {
    int msgid;
    struct my_msg_st my_data;
    msgid = msgget((key_t)1234, 0777 | IPC_CREAT);
    if (msgid == -1) {
        fprintf(stderr, "msgget failed with error: %dn", errno);
        return EXIT_FAILURE;
    }
    if (msgrcv(msgid, (void *)&my_data, BUFSIZ, 0, 0) == -1) {
        fprintf(stderr, "msgrcv failed with error: %dn", errno);
        return EXIT_FAILURE;
    }
    printf("I recieved message: %s\n", my_data.some_text);
    if (msgctl(msgid, IPC_RMID, 0) == -1) {
        fprintf(stderr, "msgctl(IPC_RMID) failedn");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}

```

Рис. 9 – Приклад програми *msgrcv* для отримання повідомлення з черги повідомлень

На рисунку 10 наведено приклад виконання програм обміну повідомленнями:

- спочатку запускається програма *msgrcv*, яка починає чекати повідомлення;
- потім запускається програма *msgsnd*, яка передає повідомлення.

```

blazhko@vps6iefe:~/OS-LabWork12-Examples
[blazhko@vps6iefe OS-LabWork12-Examples]$ ./msgrcv
I recieved message: Message from Blazhko
[blazhko@vps6iefe OS-LabWork12-Examples]$

blazhko@vps6iefe:~/OS-LabWork12-Examples
[blazhko@vps6iefe OS-LabWork12-Examples]$ gcc msgrcv.c -o msgrcv
[blazhko@vps6iefe OS-LabWork12-Examples]$ gcc msgsnd.c -o msgsnd
[blazhko@vps6iefe OS-LabWork12-Examples]$ ./msgsnd
I sent message: Message from Blazhko.
[blazhko@vps6iefe OS-LabWork12-Examples]$

```

Рис. 10 – Приклад виконання програм обміну повідомленнями

1.4 Процеси та потоки

1.4.1 Особливості використання потоків

Потоки (threads) - це можливість одночасного виконання в одному і тому же середовищі процесу декількох програм, які будуть в достатній мірі незалежними один від одного. Потоки поділяють адресний простір, відкриті файли і інші ресурси – спрощені процеси. Термін *багатопоточність* використовується для опису декількох потоків в одному процесі.

При запуску багатопоточного процесу в системі з одним процесором потоки працюють по черзі. Ілюзія паралельної роботи створюється шляхом перемикання процесора між потоками. У разі трьох обмежених продуктивністю процесора потоків, кожному буде відповідати віртуальний процесор з швидкодією рівним однієї третини швидкодії реального процесора.

Один і той же адресний простір означає спільне використання глобальних змінних, оскільки будь-який потік має доступ до будь-якої адреси оперативної пам'яті в адресному просторі процесу.

У багатопотокових застосуваннях процес спочатку трактується як один потік. Цей потік може створювати інші потоки. Ієрархія потоків відсутня – всі потоки є рівнозначними.

Переваги використання потоків:

1) легкість створення і знищення (не вимагають ніяких ресурсів), приблизно в 100 разів менше часу, ніж на створення процесу;

2) можливість спільного використання паралельними програмами адресного простору і всіх його даних.

Наприклад, в текстовому редакторі один потік виводить текст на екран, інший переформатує відповідно до змін, що надійшли, а третій періодично зберігає нові версії файлу на диску. Якби програма була однопоточною, то всі команди з клавіатури і миші ігнорувалися до закінчення дискової операції. У користувача – враження низької продуктивності. Очевидно, що в цьому випадку модель з трьома процесами не підійде, оскільки необхідна робота з одним і тим же документом. Три потоки використовують спільну пам'ять і всі три мають доступ до документу.

1.4.2 Програмування потоків

Функція *pthread_create* створює новий потік в адресному просторі процесу.

Формат виклику функції:

```
int pthread_create(pthread_t, const pthread_attr_t *attr,  
void* (*start_routine)(void*), void *arg);
```

Функція отримує в якості аргументів:

- показчик на потік, змінну типу *pthread_t*, в яку, в разі вдалого завершення зберігає id потоку;
- *pthread_attr_t* - атрибути потоку, у разі якщо використовуються атрибути за замовчуванням, то можна передавати NULL;
- *start_routine* – показчик на функцію, яка буде виконуватися в новому потоці;
- *arg* – показчик на структуру з аргументами, які будуть передані функції.

Функція *pthread_exit* завершує виконання потоку та визивається у функції, яка пов'язана з потоком.

Функція *pthread_join* дозволяє почекати, поки не завершиться інший потік, звернувшись до системного виклику та, якщо очікуваний потік вже завершив свою роботу, системний виклик *pthread_join* виконується миттєво, інакше потік, що виконав цю функцію, блокується.

Опис вказаних функцій міститься у заголовочному файлі *pthread.h*

На рисунку 11 наведено приклад програми зі створення двох потоків, які виконують дії у відповідності з кодом функцій *f1* та *f2*.

Кожна функція виводить в циклі повідомлення, при цьому кількість ітерацій передається як параметр створення потоків.

Для компіляції програми необхідно підключити додаткову бібліотеку *pthread*:

```
gcc thread.c -o thread -lpthread
```

```
// компіляція з підключенням бібліотеки -lpthread

#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/types.h>

int main(void) {
    pthread_t f2_thread, f1_thread;
    void *f2(), *f1();
    int i1 = 10, i2 = 10;
    pthread_create(&f1_thread, NULL, f1, &i1);
    pthread_create(&f2_thread, NULL, f2, &i2);
    pthread_join(f1_thread, NULL);
    pthread_join(f2_thread, NULL);
    return 0;
}

void *f1(int *x) {
    int i, n;
    n = *x;
    for (i=1; i<n; i++) {
        printf("f1: %d\n", i);
        sleep(1);
    }
    pthread_exit(0);
}

void *f2(int *x) {
    int i, n;
    n = *x;
    for (i=1; i<n; i++) {
        printf("f2: %d\n", i);
        sleep(1);
    }
    pthread_exit(0);
}
```

Рис. 11 – Приклад програми зі створення двох потоків

Після запуску програми буде запущено процес, який запустить два потоки. Обидва потоки будуть виводити на екран повідомлення зі своїх функцій, як показано на рисунку 12.

```
[oracle@vpsj3IeQ Lab10]$ ./thread
f2: 1
f1: 1
f2: 2
f1: 2
f2: 3
f1: 3
f2: 4
```

Рис. 12 – Приклад програми зі створення двох потоків

2 Завдання до лабораторної роботи

Виконати наступні дії з підготовки до виконання завдань роботи:

- 1) встановити з'єднання з *Linux*-сервером з *IP*-адресою = 46.175.148.116;
- 2) перейти до каталогу *Git*-репозиторія;
- 3) створити нову *Git*-гілку з назвою «*Laboratory-work-12*»;
- 4) перейти до роботи зі створеною гілкою;
- 5) створити каталог з назвою «*Laboratory-work-12*»;
- 6) перейти до каталогу «*Laboratory-work-12*»;
- 7) в каталозі «*Laboratory-work-12*» створити файл *README.md* та додати до файлу рядок тексту із темою лабораторної роботи «Програмування міжпроцесної взаємодії з використанням каналів, черг повідомлень та програмування багатопоточної взаємодії» як заголовок 2-го рівня *Markdown*-форматування;
- 8) виконати операції з оновлення *GitHub*-репозиторію змінами *Git*-репозиторія через послідовність *Git*-команд *add*, *commit* із коментарем «*Changed by Local Git*» та *push*;
- 9) на веб-сервісі *GitHub* перейти до створеної гілки «*Laboratory-work-12*»;
- 10) перейти до каталогу «*Laboratory-work-12*» та розпочати процес редагування файлу *README.md* ;

в подальшому за результатами рішень кожного наступного розділу завдань до файлу *README.md* додавати рядки як заголовки 3-го рівня *Markdown*-форматування з назвами розділу та знімки екрані з підписами до знімків екранів з описом пунктів завдань.

2.1 Використання черг повідомлень для процесів, які взаємодіють з однією БД під керуванням серверу СКБД *PostgreSQL*

Забезпечити обмін повідомленням між двома процесами-копіями серверів СКБД *PostgreSQL*, використовуючи команду *LISTEN* для програми-споживача та команду *NOTIFY* для програми-постачальника через утиліту *psql* та враховуючи, що:

- назва черги повідомлень співпадає з вашим прізвищем в транслітерації, наприклад, *blazhko*;
- рядок повідомлення = "*Hello, process of Surname*", наприклад, "*Hello, process of Blazhko*";
- у псевдотерміналі програми-постачальника отримати PID процесу-копії серверу СКБД;
- у псевдотерміналі програми-споживача отримати перелік назв черг повідомлень, до яких програма може отримувати доступ.

2.2 Програмна реалізація неіменованих каналів

2.2.1 Створити програму на мові C з назвою *pipe_read* яка виконує дії:

- отримує повідомлення з неіменованого каналу з використанням значення, яке передано до програми через аргумент командного рядку;
- виводить на екран повідомлення типу *"I'm child process of Surname with pid="*, наприклад, *" I'm child process of Blazhko with pid="* зі значенням свого PID;
- виводить на екран повідомлення, в якому вказано розмір повідомлення, отриманого раніше з каналу.

2.2.2 Створити програму на мові C з назвою *pipe_write*, яка виконує дії:

- створює неіменований канал;
- створює *child*-процес, в якому завантажується новий програмний образ з викликом програми *pipe_read* та її аргументом;
- *parent*-процес передає у канал повідомлення типу *"The Laboratory Work of Surname"*, наприклад, *" The Laboratory Work of Blazhko"*;
- *parent*-процес виводить на екран повідомлення типу *"I'm parent process of Surname with pid="*, наприклад, *" I'm parent process of Blazhko with pid="* зі значенням свого PID;
- *parent*-процес виводить на екран повідомлення, в якому вказано розмір повідомлення, переданого раніше у канал.

2.2.3 Скомпілювати програми та перевірити їх роботу.

Проаналізувати роботу програм та надати висновок щодо синхронності або асинхронності їх дій.

2.3 Робота з іменованими каналами через інтерпретатор командного рядку

2.3.1 Створити іменований канал з використанням команди *mkfifo*:

- назва каналу співпадає з вашим прізвищем у транслітерації;
- права доступу до каналу = можна лише читати та писати власнику та групі власника.

Переглянути властивості створеного каналу як файлу.

2.3.2 Підключити до іменованого каналу процес, який буде в нього записувати за командою, визначеною з таблиці за вашим варіантом.

Таблиця 3 – Варіанти команд

№ варіанту	Приклад команди з аргументами
1.	<code>ls -l -a</code>
2.	<code>sum -r /etc/passwd</code>
3.	<code>who -b -t</code>
4.	<code>ps -o pid,cmd</code>
5.	<code>pstree -u -s</code>
6.	<code>cat -E /etc/hosts</code>
7.	<code>free -k -w</code>
8.	<code>pwd -L -P</code>
9.	<code>who -u -p</code>
10.	<code>ps -o pid,ppid</code>
11.	<code>stat -t /etc/passwd</code>
12.	<code>pstree -p -a</code>
13.	<code>cat /etc/hosts /etc/hosts</code>
14.	<code>head --lines=3 /etc/group</code>
15.	<code>tail --lines=3 /etc/group</code>
16.	<code>command -v tail</code>
17.	<code>date -d -u</code>
18.	<code>seq 1 5</code>
19.	<code>wc -c /etc/group</code>
20.	<code>uniq --help</code>
21.	<code>uname -s -n</code>
22.	<code>ls -l -i</code>
23.	<code>sum --help</code>
24.	<code>pwd --help</code>
25.	<code>who --help</code>
26.	<code>command --help</code>
27.	<code>head --help</code>
28.	<code>tail --help</code>
29.	<code>uname --help</code>

2.3.3 Перейти до нового (2-го) псевдотерміналу роботи з ОС *Linux* та створити процес, який буде читати зі створеного раніше каналу.

Проаналізувати роботу процесів та надати висновок щодо синхронності або асинхронності їх дій.

2.3.4 Видалити іменований канал.

2.4 Програмна реалізація іменованих каналів

2.4.1 Створити програму на мові *C*, яка буде створювати іменований канал та виконувати дії у відповідності з пунктами попередніх завдань 2.3.1, 2.3.2, 2.3.4

2.4.2 Скомпілювати програму та перевірити її роботу, враховуючи пункт завдання 2.3.3.

2.5 Програмування черги повідомлень

2.2.1 Створити програму на мові C з назвою *msgsnd*, яка виконує дії:

- створює чергу, номер якої $= (1000 + 100*m+n)$, де $m = \{1,2,3,4,5,6\}$ – остання цифра у номері вашої групи, n – номер вашого варіанту (важливо, щоб цей номер ще не використовувався іншими студентами, інакше можлива помилка у зв'язку із обмеженням прав доступу на вже кимось створену чергу, тоді оберіть інший номер);

- передає до черги повідомлення типу "*Message from Surname*", наприклад, "*Message from Blazhko*";

2.2.2 Створити програму на мові C з назвою *msgrcv*, яка отримує повідомлення з черги, номер якої $= (1000 + 100*m+n)$, де $m = \{1,2,3,4,5,6\}$ – остання цифра у номері вашої групи, n – номер вашого варіанту.

2.2.3 Скомпілювати програми та перевірити їх роботу.

Проаналізувати роботу програм та надати висновок щодо синхронності або асинхронності їх дій.

2.6 Програмування потоків

2.6.1 Створити програму на мові C за прикладом з рисунку 11, в якій функції двох потоків:

- виводять на екран повідомлення, які враховують ваше прізвище транслітерацією та номер функції;

- повідомлення виводиться $(n+5)$ разів, де n – номер варіанту.

2.6.2 Скомпілювати програму та перевірити її роботу.

2.7 Підготовка процесу *Code Review* для надання рішень завдань лабораторної роботи на перевірку викладачем

Примітка: Рішення на завдання 2.1-2.3 можуть бути надані під час *Online*-заняття для отримання відповідних балів. За межами *Online*-заняття виконуються всі рішення.

На веб-сервісі *GitHub* зафіксувати зміни у файлі *README.md*.

Скопіювати файли, які було створено у попередніх розділах завдань, в каталог «*Laboratory-work-12*» *Git*-репозиторію.

Оновити *Git*-репозиторій змінами нової гілки «*Laboratory-work-12*» з *GitHub*-репозиторію.

Оновити *GitHub*-репозиторій змінами нової гілки «*Laboratory-work-12*» *Git*-репозиторію. Виконати запит *Pull Request*.

Примітка: Увага! Не натискайте кнопку «Merge pull request»!

Це повинен зробити лише рецензент, який є вашим викладачем!

Може бути виконано два запити *Pull Request*: під час *Online*-заняття та за межами *Online*-заняття.

Якщо запит *Pull Request* було зроблено під час *Online*-заняття, тоді рецензент-викладач перегляне рішення, надасть оцінку та закриє *Pull Request* без операції *Merge*.

Коли буде зроблено остаточний запит *Pull Request*, тоді рецензент-викладач перегляне ваше рішення та виконає злиття нової гілки та основної гілки через операцію *Merge*. Якщо рецензент знайде помилки, він повідомить про це у коментарях, які з'являться на сторінці *Pull request*.

2.8 Оцінка результатів виконання завдань лабораторної роботи

Оцінка	Умови
+2 бали	під час <i>Online</i> -заняття виконано правильні рішення завдань 2.1-2.3 або на наступному <i>Online</i> -занятті чи на консультації отримано правильну відповідь на два запитання, які стосуються рішень
+2 бали	1) всі рішення роботи відповідають завданням 2) <i>Pull Request</i> представлено не пізніше найближчої консультації після офіційного заняття із захисту лабораторної роботи
-0.5 балів за кожну помилку	в рішенні є помилка, про яку вказано в <i>Code Review</i>
-1 бал	<i>Pull Request</i> представлено пізніше часу завершення найближчої консультації після офіційного заняття із захисту лабораторної роботи за кожний тиждень запізнення
+2 бали	на наступному <i>Online</i> -занятті або на консультації отримано правильну відповідь на два запитання, які стосуються рішень

Література

1. Блажко О.А. Відео-запис лекції «Програмування міжпроцесної взаємодії з використанням каналів, черг повідомлень та програмування багатопоточної взаємодії». URL: <https://youtu.be/0kghg0gD3WA>