МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

## Факультет информационных технологий и робототехники

Кафедра программного обеспечения информационных систем и технологий

**Отчет по лабораторной работе № 15**

по дисциплине:” Системное программирование”

на тему:***” IPC, очереди сообщений, разделяемая память, синхронизация”***

Вариант 13

Выполнил**:** студент группы 10701222 Демянов Л. А.

Хасаншин А. В.

Приняла**:** пр. Давыденко Н.В.

Минск 2024

# Лабораторная работа № #15. IPC, очереди сообщений, разделяемая память, синхронизация.

**Цель работы:**Освоить механизмы межпроцессного взаимодействия в LINUX на основе системных ресурсов, таких как *очередей сообщений*, *разделяемую память*, а также средства синхронизации общих ресурсов.

Задание 1

### Реализация простого чат-приложения

**Цель:** научиться организовывать взаимодействие нескольких процессов с помощью очередей сообщений.

Напишите программу на C, которая создает очередь сообщений.

Реализуйте простую структуру сообщения, которая будет содержать текстовое сообщение и идентификатор отправителя.

Создайте два процесса: один для отправки сообщений, другой для их получения.

Процесс отправителя должен отправлять сообщения в очередь, а процесс получателя - получать их и выводить на экран.

Подключить к очереди сообщений двух (и более) клиентов (один из них выполняется как поток).

Расширьте предыдущее задание, добавив возможность отправки сообщений от нескольких процессов-отправителей к одному процессу-получателю.

Реализуйте возможность указания имени отправителя (члена команды) в каждом сообщении.

Добавьте в программу обработку сигналов для корректного завершения работы процессов.

Ключ к общей памяти должен задаваться автоматически системой.

**Код Программы :**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <pthread.h>

#include <signal.h>

#include <unistd.h>

#define MAX\_MSG\_SIZE 100

#define MAX\_NAME\_SIZE 30

// Структура сообщения

struct message {

long mtype; // Тип сообщения

char sender[MAX\_NAME\_SIZE]; // Имя отправителя

char mtext[MAX\_MSG\_SIZE]; // Текст сообщения

};

int msgid; // ID очереди сообщений

// Обработчик сигнала для корректного завершения

void signal\_handler(int signo) {

if (signo == SIGINT) {

printf("\nПолучен сигнал завершения. Удаляем очередь сообщений...\n");

msgctl(msgid, IPC\_RMID, NULL);

exit(0);

}

}

// Функция потока-отправителя

void\* sender\_thread(void\* arg) {

struct message msg;

char\* sender\_name = (char\*)arg;

msg.mtype = 1;

strncpy(msg.sender, sender\_name, MAX\_NAME\_SIZE);

while(1) {

printf("Введите сообщение от %s (или 'exit' для выхода): ", sender\_name);

fgets(msg.mtext, MAX\_MSG\_SIZE, stdin);

msg.mtext[strcspn(msg.mtext, "\n")] = 0; // Убираем символ новой строки

if(strcmp(msg.mtext, "exit") == 0) {

break;

}

// Отправляем сообщение

if(msgsnd(msgid, &msg, sizeof(struct message) - sizeof(long), 0) == -1) {

perror("msgsnd");

exit(1);

}

}

return NULL;

}

// Функция процесса-получателя

void receiver\_process() {

struct message msg;

printf("Процесс-получатель запущен...\n");

while(1) {

// Получаем сообщение

if(msgrcv(msgid, &msg, sizeof(struct message) - sizeof(long), 1, 0) == -1) {

perror("msgrcv");

exit(1);

}

printf("Получено сообщение от %s: %s\n", msg.sender, msg.mtext);

}

}

int main() {

// Устанавливаем обработчик сигнала

signal(SIGINT, signal\_handler);

// Создаем очередь сообщений

key\_t key = ftok(".", 'a');

if((msgid = msgget(key, IPC\_CREAT | 0666)) == -1) {

perror("msgget");

exit(1);

}

pid\_t pid = fork();

if(pid < 0) {

perror("fork");

exit(1);

}

else if(pid == 0) { // Дочерний процесс - получатель

receiver\_process();

}

else { // Родительский процесс

pthread\_t sender1, sender2;

char sender1\_name[] = "Отправитель 1";

char sender2\_name[] = "Отправитель 2";

// Создаем потоки-отправители

if(pthread\_create(&sender1, NULL, sender\_thread, (void\*)sender1\_name) != 0) {

perror("pthread\_create");

exit(1);

}

if(pthread\_create(&sender2, NULL, sender\_thread, (void\*)sender2\_name) != 0) {

perror("pthread\_create");

exit(1);

}

// Ожидаем завершения потоков

pthread\_join(sender1, NULL);

pthread\_join(sender2, NULL);

// Удаляем очередь сообщений

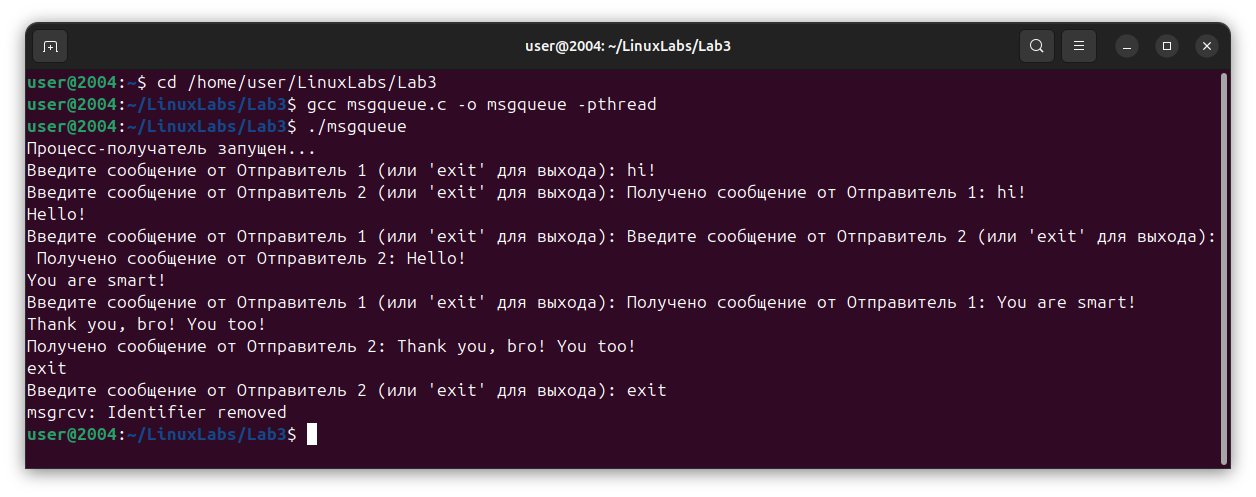
msgctl(msgid, IPC\_RMID, NULL);

}

return 0;

}

**Результаты\_работы:**



## Задание 2.

Создать два (и более) объекта ***общие памяти***. Первый процесс пишет в первый объект и читает из второго объекта памяти, второй процесс, наоборот, читает из первого объекта и пишет во второй (программа-чат).

Ключ общей памяти должен задаваться пользователем, например, это могут быть номера телефонов членов команды.

### Скриншоты результатов

#include <iostream>

#include <sys/mman.h>

#include <fcntl.h>

#include <unistd.h>

#include <cstring>

#include <string>

#include <thread>

#include <atomic>

struct SharedMemory {

char message[256]; // Сообщение

std::atomic<bool> has\_message; // Флаг наличия нового сообщения

};

void readMessages(SharedMemory\* read\_shm) {

while (true) {

if (read\_shm->has\_message) {

std::cout << "\nДругой процесс: " << read\_shm->message << "\n> ";

std::cout.flush();

read\_shm->has\_message = false;

}

usleep(100); // Небольшая пауза, чтобы избежать активного ожидания

}

}

void chatProcess(const std::string& write\_key, const std::string& read\_key) {

// Открытие сегментов общей памяти

int write\_fd = shm\_open(write\_key.c\_str(), O\_CREAT | O\_RDWR, 0666);

int read\_fd = shm\_open(read\_key.c\_str(), O\_CREAT | O\_RDWR, 0666);

if (write\_fd < 0 || read\_fd < 0) {

perror("shm\_open");

exit(EXIT\_FAILURE);

}

// Задание размера сегментов

ftruncate(write\_fd, sizeof(SharedMemory));

ftruncate(read\_fd, sizeof(SharedMemory));

// Маппинг сегментов в память

SharedMemory\* write\_shm = (SharedMemory\*)mmap(nullptr, sizeof(SharedMemory), PROT\_READ | PROT\_WRITE, MAP\_SHARED, write\_fd, 0);

SharedMemory\* read\_shm = (SharedMemory\*)mmap(nullptr, sizeof(SharedMemory), PROT\_READ | PROT\_WRITE, MAP\_SHARED, read\_fd, 0);

if (write\_shm == MAP\_FAILED || read\_shm == MAP\_FAILED) {

perror("mmap");

exit(EXIT\_FAILURE);

}

// Инициализация

write\_shm->has\_message = false;

read\_shm->has\_message = false;

std::cout << "Чат готов! Введите сообщения:\n";

// Запуск потока для чтения сообщений

std::thread readerThread(readMessages, read\_shm);

// Основной поток: отправка сообщений

while (true) {

std::cout << "> ";

std::string input;

std::getline(std::cin, input);

strncpy(write\_shm->message, input.c\_str(), sizeof(write\_shm->message) - 1);

write\_shm->message[sizeof(write\_shm->message) - 1] = '\0';

write\_shm->has\_message = true;

}

// Завершение

readerThread.join();

munmap(write\_shm, sizeof(SharedMemory));

munmap(read\_shm, sizeof(SharedMemory));

close(write\_fd);

close(read\_fd);

shm\_unlink(write\_key.c\_str());

shm\_unlink(read\_key.c\_str());

}

int main() {

std::string phone1, phone2;

std::cout << "Введите номер вашего телефона: ";

std::cin >> phone1;

std::cout << "Введите номер телефона собеседника: ";

std::cin >> phone2;

std::cin.ignore(); // Очистка буфера для последующего ввода

std::cout << "Запущен процесс чата. Используйте номера:\n"

<< "Ваш номер: " << phone1 << "\n"

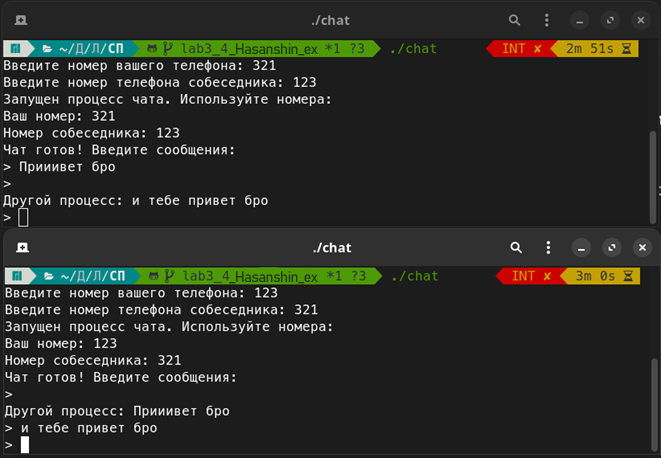
<< "Номер собеседника: " << phone2 << "\n";

// Для этого процесса write\_key = phone1, read\_key = phone2

chatProcess("/shm\_" + phone1, "/shm\_" + phone2);

return 0;

}



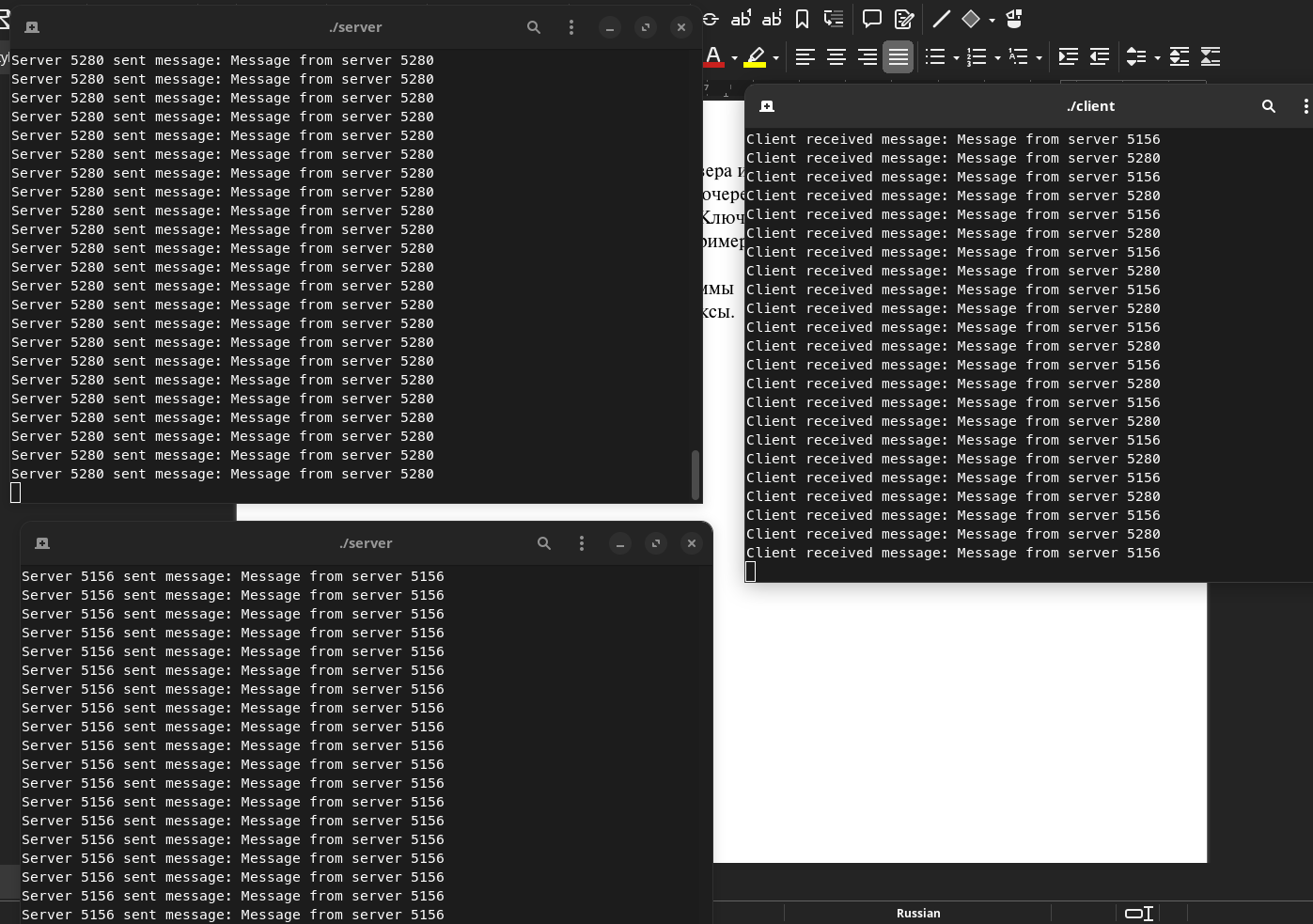
## Задание 3

Сделать два (и более) программы сервера и одного клиента.

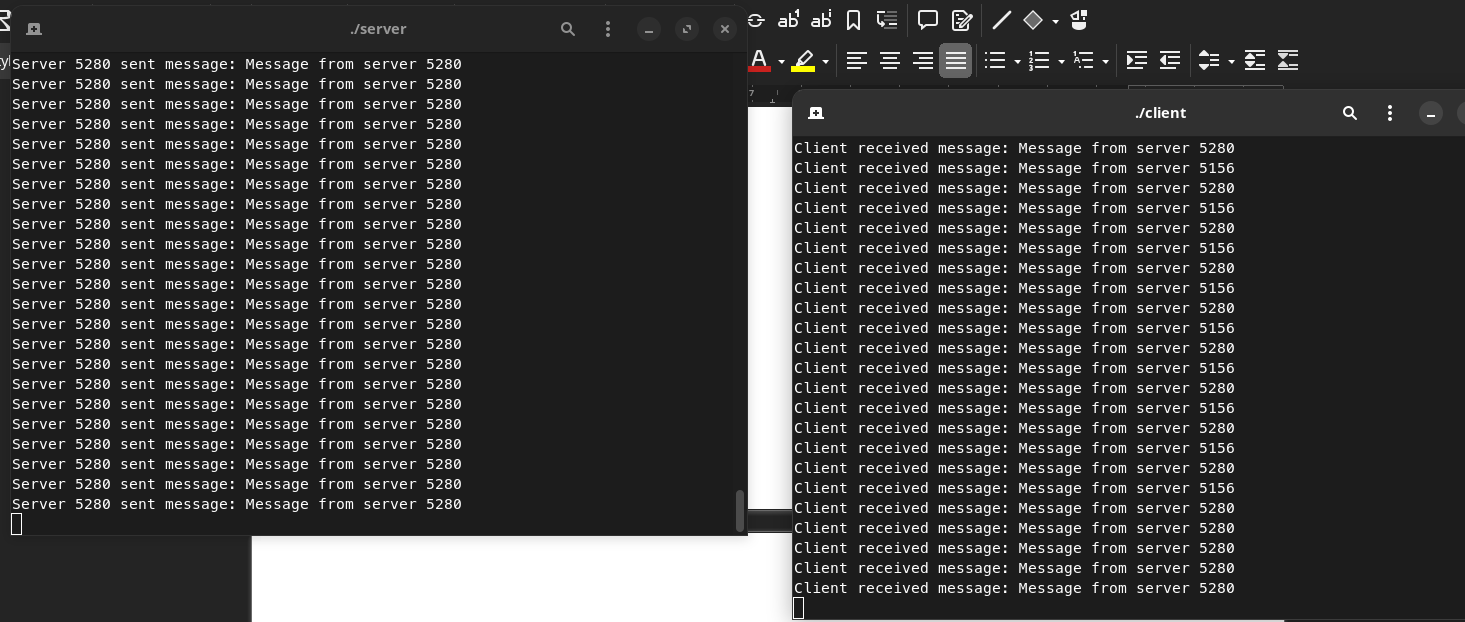
Серверы используют одну и ту же очередь. Клиент читает и определяет, от какого сервера пришло сообщение. Ключ к очереди создается системой либо устанавливается пользователем (например, это могут быть номера телефонов членов команды).

Для совершенствования программы можно использовать средства синхронизации, семафоры или мьютексы.

Запущено 2 сервера:



Запущен 1 сервер:

  
  
Сервер:

#include <iostream>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <unistd.h>

#include <pthread.h>

#include <cstring>

#define QUEUE\_KEY 12345

#define MSG\_SIZE 256

struct message {

long mtype;

char mtext[MSG\_SIZE];

};

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

void send\_message(int server\_id) {

int msgid = msgget(QUEUE\_KEY, 0666 | IPC\_CREAT);

if (msgid == -1) {

perror("msgget failed");

exit(1);

}

message msg;

msg.mtype = 1;

snprintf(msg.mtext, MSG\_SIZE, "Message from server %d", server\_id);

pthread\_mutex\_lock(&mutex);

if (msgsnd(msgid, &msg, sizeof(msg.mtext), 0) == -1) {

perror("msgsnd failed");

pthread\_mutex\_unlock(&mutex);

exit(1);

}

std::cout << "Server " << server\_id << " sent message: " << msg.mtext << std::endl;

pthread\_mutex\_unlock(&mutex);

}

int main() {

int server\_id = getpid(); // Используем идентификатор процесса как уникальный идентификатор сервера

while (true) {

send\_message(server\_id);

sleep(1);

}

return 0;

}

Клиент:

#include <iostream>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <unistd.h>

#define QUEUE\_KEY 12345

#define MSG\_SIZE 256

struct message {

long mtype;

char mtext[MSG\_SIZE];

};

int main() {

int msgid = msgget(QUEUE\_KEY, 0666 | IPC\_CREAT);

if (msgid == -1) {

perror("msgget failed");

exit(1);

}

message msg;

while (true) {

if (msgrcv(msgid, &msg, sizeof(msg.mtext), 1, 0) == -1) {

perror("msgrcv failed");

exit(1);

}

std::cout << "Client received message: " << msg.mtext << std::endl;

}

return 0;

}

Вывод: Были изучены механизмы межпроцессовго взаимодействия на основе системных ресурсов очереди сообщений и разделяемая память в Ubuntu.

## Контрольные вопросы

1. Что такое **семафоры**, для чего они используются?

Семафоры — это примитивы синхронизации, используемые для управления доступом к общим ресурсам в многопоточных и многопроцессорных системах. Они представляют собой счетчики, которые позволяют контролировать, сколько потоков или процессов могут одновременно выполнять определенные действия или доступ к ресурсам.

### Основные функции семафоров:

**Синхронизация процессов**: Семафоры позволяют синхронизировать выполнение процессов, обеспечивая, что только определенное количество процессов может одновременно выполнять критические секции кода или доступ к общим ресурсам.

**Защита критических секций**: С помощью семафоров можно предотвратить одновременный доступ нескольких потоков к разделяемым данным, что помогает избежать гонок данных и других проблем, связанных с параллельным выполнением.

**Управление ресурсами**: Семафоры используются для контроля доступа к ограниченным ресурсам, например, к памяти или аппаратным устройствам, что позволяет избежать ситуации, когда ресурсы используются одновременно несколькими процессами.

**Решение типовых задач**: Семафоры могут быть использованы для реализации различных алгоритмов, таких как задача производителя и потребителя, где один процесс производит данные, а другой их потребляет, или для создания барьеров, которые синхронизируют группы задач в определенных контрольных точках.

Что такое **синхронизация процессов**?

Синхронизация процессов — это механизм упорядочивания выполнения процессов или потоков, чтобы они корректно взаимодействовали друг с другом, избегая конфликтов при доступе к общим ресурсам и обеспечивая согласованность данных.