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: “ИСПОЛЬЗОВАНИЕ ОБЩИХ ФАЙЛОВ, БУФЕРИЗАЦИЯ, СИНХРОНИЗАЦИЯ ”

Выполнил**:** студент группы 10701321

Грудинский К. А.

Принял**:** Кондратёнок Е.В.

Минск 2023

# Лабораторная работа № 15. ИСПОЛЬЗОВАНИЕ ОБЩИХ ФАЙЛОВ, БУФЕРИЗАЦИЯ, СИНХРОНИЗАЦИЯ

# Цель работы: Освоить механизм взаимодействия между процессами на основе использования общей памяти.

**Задание 1**

1.Создайте двух клиентов и один сервер, передайте данные с помощью маппинга файла.

**Решение**

#include <iostream>

#include <fstream>

#include <sys/mman.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/wait.h>

#define FILENAME "shared\_file"

struct SharedData {

int value;

char message[256];

};

void serverProcess() {

int fd = open(FILENAME, O\_CREAT | O\_RDWR, S\_IRUSR | S\_IWUSR);

if (fd == -1) {

std::cerr << "Failed to open the file" << std::endl;

return;

}

ftruncate(fd, sizeof(SharedData));

SharedData\* sharedData = static\_cast<SharedData\*>(mmap(NULL, sizeof(SharedData), PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0));

close(fd);

while (true) {

// Server reads data from the shared memory

std::cout << "Server: Value = " << sharedData->value << ", Message = " << sharedData->message << std::endl;

sleep(1);

}

munmap(sharedData, sizeof(SharedData));

}

void clientProcess(int clientId) {

int fd = open(FILENAME, O\_RDWR);

if (fd == -1) {

std::cerr << "Failed to open the file" << std::endl;

return;

}

SharedData\* sharedData = static\_cast<SharedData\*>(mmap(NULL, sizeof(SharedData), PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0));

close(fd);

int counter = 0;

while (counter < 5) {

// Clients write data to the shared memory

sharedData->value = counter;

snprintf(sharedData->message, sizeof(sharedData->message), "Message from Client %d", clientId);

std::cout << "Client " << clientId << ": Data written" << std::endl;

sleep(2);

counter++;

}

munmap(sharedData, sizeof(SharedData));

}

int main() {

// Create the server process

if (fork() == 0) {

serverProcess();

exit(0);

}

// Create two client processes

if (fork() == 0) {

clientProcess(1);

exit(0);

}

if (fork() == 0) {

clientProcess(2);

exit(0);

}

// Wait for all child processes to complete

for (int i = 0; i < 3; ++i) {

wait(NULL);

}

return 0;

}

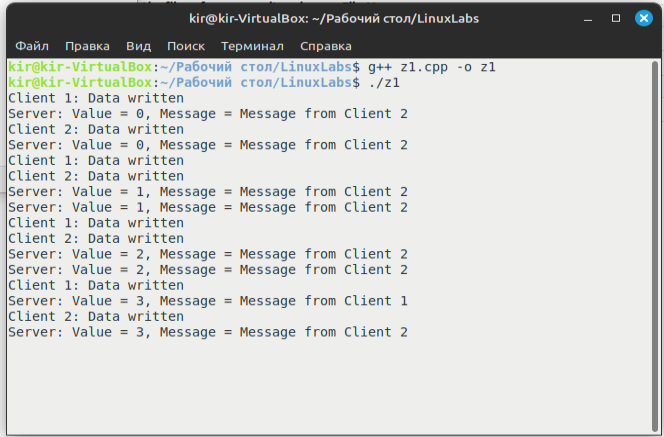


Рисунок 1 – выполнение задания 1

Задание 2.

Создайте два сервера и одного клиента и передайте данные через маппинг файла. Клиент должен определить от какого сервера приходит сообщение.

#include <iostream>

#include <sys/mman.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

#include <cstring>

#include <sys/wait.h>

#define FILENAME "shared\_file"

#define MESSAGE\_SIZE 256

struct SharedData {

int serverId;

char message[MESSAGE\_SIZE];

};

void serverProcess(int id) {

int fd = open(FILENAME, O\_CREAT | O\_RDWR, S\_IRUSR | S\_IWUSR);

if (fd == -1) {

std::cerr << "Failed to open the file" << std::endl;

return;

}

ftruncate(fd, sizeof(SharedData));

SharedData\* sharedData = static\_cast<SharedData\*>(mmap(NULL, sizeof(SharedData), PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0));

close(fd);

int counter = 0;

while (counter < 5) {

sharedData->serverId = id;

snprintf(sharedData->message, sizeof(sharedData->message), "Message from Server %d", id);

std::cout << "Server " << id << ": Data written" << std::endl;

sleep(2);

counter++;

}

munmap(sharedData, sizeof(SharedData));

}

void clientProcess() {

int fd = open(FILENAME, O\_RDONLY);

if (fd == -1) {

std::cerr << "Failed to open the file" << std::endl;

return;

}

SharedData\* sharedData = static\_cast<SharedData\*>(mmap(NULL, sizeof(SharedData), PROT\_READ, MAP\_SHARED, fd, 0));

close(fd);

for (int i = 0; i < 10; ++i) {

std::cout << "Client: Received from Server " << sharedData->serverId << ": " << sharedData->message << std::endl;

sleep(1);

}

munmap(sharedData, sizeof(SharedData));

}

int main() {

// Create two server processes

if (fork() == 0) {

serverProcess(1);

exit(0);

}

if (fork() == 0) {

serverProcess(2);

exit(0);

}

// Create one client process

if (fork() == 0) {

clientProcess();

exit(0);

}

// Wait for all child processes to complete

for (int i = 0; i < 3; ++i) {

wait(NULL);

}

return 0;

}

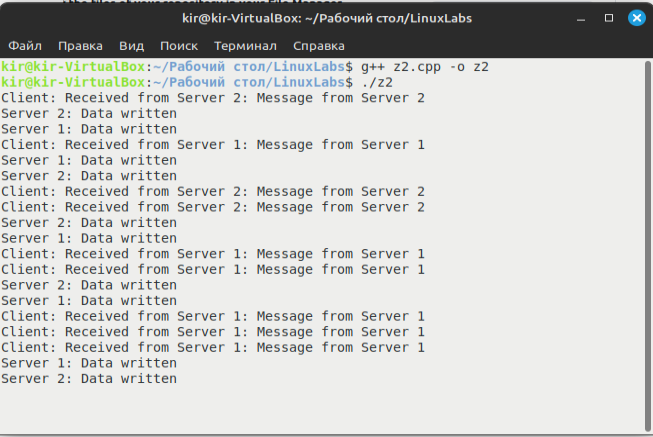


Рисунок 2 – выполнение задания 2

Вывод*:* В ходе выполнения лабораторной работы ознакомились с использованием общих файлов, синхронизацией и буферизацией в Linux.

**Контрольные вопросы**

*Что такое синхронизация процессов?*

Синхронизация процессов — это механизм, используемый в многозадачных и параллельных вычислительных системах для обеспечения правильного взаимодействия между параллельно выполняемыми процессами или потоками. Цель синхронизации заключается в предотвращении возможных проблем, таких как гонки данных, блокировки и другие условия, которые могут привести к непредсказуемому поведению программы или даже к её сбоям.

Вот несколько ключевых аспектов синхронизации процессов:

1. Гонки данных (Race Conditions): Гонка данных возникает, когда несколько процессов или потоков пытаются одновременно обращаться к общим ресурсам (переменным, файлам и т.д.), и хотя бы одна из этих операций включает запись. Результат зависит от того, какие операции завершатся быстрее, что может привести к некорректному состоянию программы.

2. Блокировки (Locks): Использование блокировок — это один из основных способов синхронизации. Процессы или потоки могут блокировать доступ к ресурсам до тех пор, пока не завершат свою работу. Это предотвращает гонки данных, но может также привести к проблемам, связанным с долгими ожиданиями и блокировками других ресурсов.

3. Семафоры: Семафоры — это абстрактные объекты, которые могут использоваться для синхронизации между процессами. Они представляют собой счетчики, которые уменьшаются и увеличиваются. Процессы ждут, пока семафор не станет положительным, прежде чем продолжить выполнение.

4. Мьютексы (Mutex): Мьютексы (от англ. "mutex", сокращение от "mutual exclusion") представляют собой объекты синхронизации, предназначенные для обеспечения взаимного исключения. Только один процесс или поток может владеть мьютексом в определенный момент времени, что предотвращает одновременный доступ к общим ресурсам.

5. События (Events): События используются для уведомления о том, что произошло какое-то событие. Один процесс может ждать, пока другой процесс или поток установит событие, чтобы продолжить выполнение.

Синхронизация процессов становится критически важной в многозадачных и параллельных системах, где несколько потоков или процессов конкурируют за ресурсы и должны взаимодействовать так, чтобы избежать проблем согласованности данных и обеспечить корректное выполнение программы.