|  |
| --- |
| «Санкт-Петербургский государственный электротехнический университет |
| «ЛЭТИ» им. В.И.Ульянова(Ленина)» |
| (СПбГЭТУ «ЛЭТИ») Кафедра вычислительной техники |
|  |
| **Отчет**  **по лабораторным работам № 8**  **по дисциплине** **«Организация процессов и программирование в среде Linux»** |
| Тема: «Обмен данными через очередь» |
|  |

|  |
| --- |
|  |

|  |  |
| --- | --- |
| Студент гр. 9307 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_Брызгалова Е. А. |
|  |  |
|  |  |
| Преподаватель | \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Разумовский Г. В. |

Оглавление

[Цель работы: 2](#_Toc118034666)

[Задание 2](#_Toc118034667)

[Распечатки входных и выходного файлов. 2](#_Toc118034668)

[Текст программы main.cpp 3](#_Toc118034669)

[Вывод 6](#_Toc118034670)

Цель работы:

Знакомство с механизмом обмена данными через программный канал и системными вызовами, обеспечивающими такой обмен.

Задание

1) Написать программу, которая обменивается данными через канал с двумя потомками. Программа открывает входной файл, построчно читает из него данные и записывает их в канал. Потомки выполняют свои программы и поочередно читают символы из канала и записывают их в свои выходные файлы: первый потомок – нечетные символы, а второй – четные. Синхронизация работы потомков должна осуществляться напрямую с использованием сигналов SIGUSR1 и SIGUSR2. Об окончании записи файла в канал программа оповещает потомков сигналом SIGQUIT и ожидает завершения работы потомков. Когда они заканчивают работу, программа закрывает канал.

2) Откомпилировать все программы и запустить их

Распечатки входных и выходного файлов.

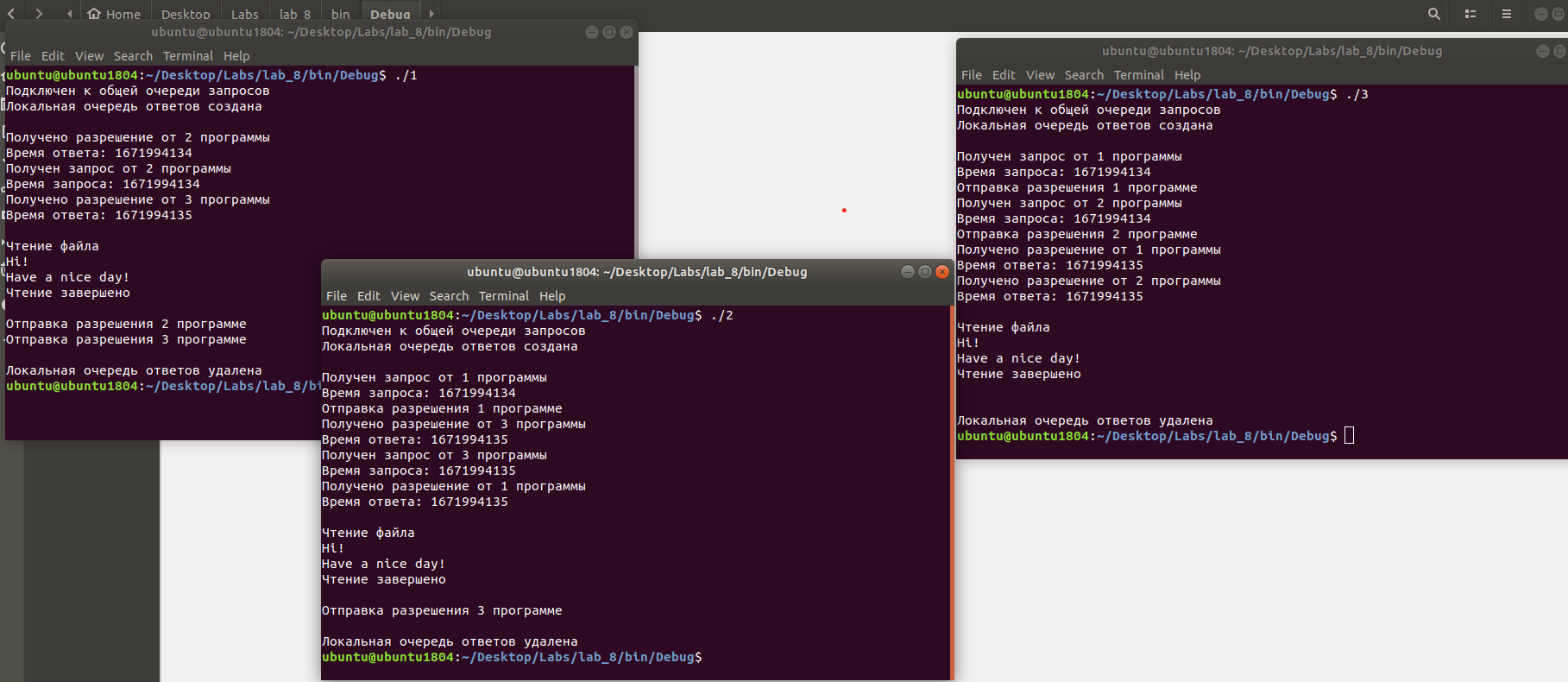


Рисунок 1. Файл input

# **Текст программы 1.cpp**

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include <iostream>

#include <fstream>

#include <cstring>

#include <fcntl.h>

#include <signal.h>

#include <sys/types.h>

#include <string.h>

#include <assert.h>

#include <sys/wait.h>

using namespace std;

signed int fd[2];

char buf[1];

int pid1, pid2;

int i;

sigset\_t sig\_m1;

void ex(int sig)

{

cout<<"Finised"<<endl;

exit(0);

}

int main()

{ bool f1,f2;

ifstream file1;

// игнорируем сигналы синхронизации потомков

signal(SIGUSR1, SIG\_IGN);

signal(SIGUSR2, SIG\_IGN);

signal(SIGQUIT, SIG\_IGN);

pipe(fd);

const int flags = fcntl(fd[0], F\_GETFL, 0);

fcntl(fd[0], F\_SETFL, flags | O\_NONBLOCK);

switch(pid1 = fork())

{

case -1 : fprintf(stderr, "Ошибка fork()\n");

exit(0);

case 0 :

{

close(fd[1]);

execl("c1","c1", "output1.txt", &pipefd[0], NULL);

}

default :

switch( pid2 = fork())

{

case -1 : fprintf(stderr, "Ошибка fork()\n");

exit(0);

case 0 :

{

close(pipefd[1]);

execl("c2","c2", "output1.txt", &pipefd[0], NULL);

}

default :

close(fd[0]);

file1.open("text.txt");

string s;

while(getline(file1, s)){

cout<<"Parent: "<<s<< endl;

write(fd[1], &s[0], strlen(s.c\_str()));

}

kill(pid1, SIGQUIT);

kill(pid2, SIGQUIT);

file1.close();

}

}

waitpid(pid1, NULL, 0);

waitpid(pid2, NULL, 0);

close (fd[0]);

close (fd[1]);

return 0;

}

# **Текст программы 1.cpp**

#include <stdio.h>

#include <stdbool.h>

#include <time.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

typedef struct request\_message{

long mtype;

int sender;

int response\_queue;

int time;

} request\_msg;

typedef struct response\_message{

long mtype;

int sender;

} response\_msg;

int main(){

bool is\_creator; // метка сервера

int prog\_id = 1; // метка программы

int request\_queue; // общая очередь запросов

int response\_queue; // локальная очередь ответов

int request\_time; // время отправки запроса

request\_msg req\_msg; // отправляемый запрос

request\_msg req\_buf[2]; // принимаемые запросы

int resp\_no\_count = 0; // запросов без ответа

response\_msg resp\_msg; // отправляемый ответ

response\_msg resp\_buf; // принимаемый ответ

int permissions\_get = 0; // разрешений получено

int permissions\_send = 0; // разрешений отправлено

int readers\_now = 3; // количество текущих пользователей очереди запросов

//создание общей очереди запросов

request\_queue = msgget(121, 0606|IPC\_CREAT|IPC\_EXCL);

if(request\_queue != -1){

is\_creator = true;

puts("Общая очередь запросов создана");

}else{

request\_queue = msgget(121, IPC\_CREAT);

if(request\_queue == -1){

puts("Не удается подключиться к общей очереди запросов");

return 1;

}else{

puts("Подключен к общей очереди запросов");

}

}

//создание локальной очереди ответов

response\_queue = msgget(IPC\_PRIVATE, 0606|IPC\_CREAT);

if(response\_queue == -1){

puts("Не удается создать локальную очередь ответов");

if(is\_creator){

msgctl(request\_queue, IPC\_RMID, 0);

return 2;

}

}else{

puts("Локальная очередь ответов создана\n");

}

request\_time = time(0);

req\_msg.time = request\_time;

req\_msg.mtype = 2;

req\_msg.sender = prog\_id;

req\_msg.response\_queue = response\_queue;

msgsnd(request\_queue, &req\_msg, sizeof(request\_msg), 0); // 1 -> 2

req\_msg.mtype = 3;

msgsnd(request\_queue, &req\_msg, sizeof(request\_msg), 0); // 1 -> 3

resp\_msg.mtype = 1;

resp\_msg.sender = prog\_id;

// ожидание получения разрешений от других программ

while(permissions\_get < 2){

//проверка сообщений в общей очереди запросов

if(msgrcv(request\_queue, &req\_buf[resp\_no\_count], sizeof(request\_msg), prog\_id, IPC\_NOWAIT) != -1){

printf("Получен запрос от %d программы\n", req\_buf[resp\_no\_count].sender);

printf("Время запроса: %d\n", req\_buf[resp\_no\_count].time);

if(req\_buf[resp\_no\_count].time < request\_time

|| (req\_buf[resp\_no\_count].time == request\_time && req\_buf[resp\_no\_count].sender < prog\_id)){

msgsnd(req\_buf[resp\_no\_count].response\_queue, &resp\_msg, sizeof(response\_msg), 0);

permissions\_send++;

printf("Отправка разрешения %d программе\n", req\_buf[resp\_no\_count].sender);

}else{

resp\_no\_count++;

}

}

//проверка сообщений в локальной очереди

if(msgrcv(response\_queue, &resp\_buf, sizeof(response\_msg), 1, IPC\_NOWAIT) != -1)

{

permissions\_get++;

printf("Получено разрешение от %d программы\n", resp\_buf.sender);

printf("Время ответа: %ld\n", time(0));

}

}

// чтение файла

FILE \*fp = fopen("text.txt", "r");

char arr[124];

printf("\nЧтение файла\n");

while (fgets(arr, 124, fp) != NULL)

{

printf("%s", arr);

}

puts("Чтение завершено\n");

fclose(fp);

// обслуживание запросов без ответа

while(resp\_no\_count > 0){

msgsnd(req\_buf[resp\_no\_count - 1].response\_queue, &resp\_msg, sizeof(response\_msg), 0);

permissions\_send++;

printf("Отправка разрешения %d программе\n", req\_buf[resp\_no\_count - 1].sender);

resp\_no\_count--;

}

while(permissions\_send < 2){

if(msgrcv(request\_queue, &req\_buf[resp\_no\_count], sizeof(request\_msg), prog\_id, IPC\_NOWAIT) != -1){

msgsnd(req\_buf[resp\_no\_count].response\_queue, &resp\_msg, sizeof(response\_msg), 0);

permissions\_send++;

printf("Отправка разрешения %d программе\n", req\_buf[resp\_no\_count].sender);

}

}

//удаление очередей

req\_msg.mtype = 4;

msgsnd(request\_queue, &req\_msg, sizeof(request\_msg), 0);

msgctl(response\_queue, IPC\_RMID, 0);

puts("\nЛокальная очередь ответов удалена");

if(is\_creator){

while(readers\_now > 0){

if(msgrcv(request\_queue, &req\_msg, sizeof(request\_msg), 4, 0) != -1){

readers\_now--;

}

}

msgctl(request\_queue, IPC\_RMID, 0);

puts("Общая очередь запросов удалена");

}

return 0;

}

# **Текст программы 2.cpp**

#include <stdio.h>

#include <stdbool.h>

#include <time.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

typedef struct request\_message{

long mtype;

int sender;

int response\_queue;

int time;

} request\_msg;

typedef struct response\_message{

long mtype;

int sender;

} response\_msg;

int main(){

bool is\_creator; // метка сервера

int prog\_id = 2; // метка программы

int request\_queue; // общая очередь запросов

int response\_queue; // локальная очередь ответов

int request\_time; // время отправки запроса

request\_msg req\_msg; // отправляемый запрос

request\_msg req\_buf[2]; // принимаемые запросы

int resp\_no\_count = 0; // запросов без ответа

response\_msg resp\_msg; // отправляемый ответ

response\_msg resp\_buf; // принимаемый ответ

int permissions\_get = 0; // разрешений получено

int permissions\_send = 0; // разрешений отправлено

int readers\_now = 3; // количество текущих пользователей очереди запросов

//создание общей очереди запросов

request\_queue = msgget(121, 0606|IPC\_CREAT|IPC\_EXCL);

if(request\_queue != -1){

is\_creator = true;

puts("Общая очередь запросов создана");

}else{

request\_queue = msgget(121, IPC\_CREAT);

if(request\_queue == -1){

puts("Не удается подключиться к общей очереди запросов");

return 1;

}else{

puts("Подключен к общей очереди запросов");

}

}

//создание локальной очереди ответов

response\_queue = msgget(IPC\_PRIVATE, 0606|IPC\_CREAT);

if(response\_queue == -1){

puts("Не удается создать локальную очередь ответов");

if(is\_creator){

msgctl(request\_queue, IPC\_RMID, 0);

return 2;

}

}else{

puts("Локальная очередь ответов создана\n");

}

request\_time = time(0);

req\_msg.time = request\_time;

req\_msg.mtype = 2;

req\_msg.sender = prog\_id;

req\_msg.response\_queue = response\_queue;

msgsnd(request\_queue, &req\_msg, sizeof(request\_msg), 0); // 1 -> 2

req\_msg.mtype = 3;

msgsnd(request\_queue, &req\_msg, sizeof(request\_msg), 0); // 1 -> 3

resp\_msg.mtype = 1;

resp\_msg.sender = prog\_id;

// ожидание получения разрешений от других программ

while(permissions\_get < 2){

//проверка сообщений в общей очереди запросов

if(msgrcv(request\_queue, &req\_buf[resp\_no\_count], sizeof(request\_msg), prog\_id, IPC\_NOWAIT) != -1){

printf("Получен запрос от %d программы\n", req\_buf[resp\_no\_count].sender);

printf("Время запроса: %d\n", req\_buf[resp\_no\_count].time);

if(req\_buf[resp\_no\_count].time < request\_time

|| (req\_buf[resp\_no\_count].time == request\_time && req\_buf[resp\_no\_count].sender < prog\_id)){

msgsnd(req\_buf[resp\_no\_count].response\_queue, &resp\_msg, sizeof(response\_msg), 0);

permissions\_send++;

printf("Отправка разрешения %d программе\n", req\_buf[resp\_no\_count].sender);

}else{

resp\_no\_count++;

}

}

//проверка сообщений в локальной очереди

if(msgrcv(response\_queue, &resp\_buf, sizeof(response\_msg), 1, IPC\_NOWAIT) != -1)

{

permissions\_get++;

printf("Получено разрешение от %d программы\n", resp\_buf.sender);

printf("Время ответа: %ld\n", time(0));

}

}

// чтение файла

FILE \*fp = fopen("text.txt", "r");

char arr[124];

printf("\nЧтение файла\n");

while (fgets(arr, 124, fp) != NULL)

{

printf("%s", arr);

}

puts("Чтение завершено\n");

fclose(fp);

// обслуживание запросов без ответа

while(resp\_no\_count > 0){

msgsnd(req\_buf[resp\_no\_count - 1].response\_queue, &resp\_msg, sizeof(response\_msg), 0);

permissions\_send++;

printf("Отправка разрешения %d программе\n", req\_buf[resp\_no\_count - 1].sender);

resp\_no\_count--;

}

while(permissions\_send < 2){

if(msgrcv(request\_queue, &req\_buf[resp\_no\_count], sizeof(request\_msg), prog\_id, IPC\_NOWAIT) != -1){

msgsnd(req\_buf[resp\_no\_count].response\_queue, &resp\_msg, sizeof(response\_msg), 0);

permissions\_send++;

printf("Отправка разрешения %d программе\n", req\_buf[resp\_no\_count].sender);

}

}

//удаление очередей

req\_msg.mtype = 4;

msgsnd(request\_queue, &req\_msg, sizeof(request\_msg), 0);

msgctl(response\_queue, IPC\_RMID, 0);

puts("\nЛокальная очередь ответов удалена");

if(is\_creator){

while(readers\_now > 0){

if(msgrcv(request\_queue, &req\_msg, sizeof(request\_msg), 4, 0) != -1){

readers\_now--;

}

}

msgctl(request\_queue, IPC\_RMID, 0);

puts("Общая очередь запросов удалена");

}

return 0;

}

# **Текст программы 3.cpp**

#include <stdio.h>

#include <stdbool.h>

#include <time.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

typedef struct request\_message{

long mtype;

int sender;

int response\_queue;

int time;

} request\_msg;

typedef struct response\_message{

long mtype;

int sender;

} response\_msg;

int main(){

bool is\_creator; // метка сервера

int prog\_id = 3; // метка программы

int request\_queue; // общая очередь запросов

int response\_queue; // локальная очередь ответов

int request\_time; // время отправки запроса

request\_msg req\_msg; // отправляемый запрос

request\_msg req\_buf[2]; // принимаемые запросы

int resp\_no\_count = 0; // запросов без ответа

response\_msg resp\_msg; // отправляемый ответ

response\_msg resp\_buf; // принимаемый ответ

int permissions\_get = 0; // разрешений получено

int permissions\_send = 0; // разрешений отправлено

int readers\_now = 3; // количество текущих пользователей очереди запросов

//создание общей очереди запросов

request\_queue = msgget(121, 0606|IPC\_CREAT|IPC\_EXCL);

if(request\_queue != -1){

is\_creator = true;

puts("Общая очередь запросов создана");

}else{

request\_queue = msgget(121, IPC\_CREAT);

if(request\_queue == -1){

puts("Не удается подключиться к общей очереди запросов");

return 1;

}else{

puts("Подключен к общей очереди запросов");

}

}

//создание локальной очереди ответов

response\_queue = msgget(IPC\_PRIVATE, 0606|IPC\_CREAT);

if(response\_queue == -1){

puts("Не удается создать локальную очередь ответов");

if(is\_creator){

msgctl(request\_queue, IPC\_RMID, 0);

return 2;

}

}else{

puts("Локальная очередь ответов создана\n");

}

request\_time = time(0);

req\_msg.time = request\_time;

req\_msg.mtype = 2;

req\_msg.sender = prog\_id;

req\_msg.response\_queue = response\_queue;

msgsnd(request\_queue, &req\_msg, sizeof(request\_msg), 0); // 1 -> 2

req\_msg.mtype = 1;

msgsnd(request\_queue, &req\_msg, sizeof(request\_msg), 0); // 1 -> 3

resp\_msg.mtype = 1;

resp\_msg.sender = prog\_id;

// ожидание получения разрешений от других программ

while(permissions\_get < 2){

//проверка сообщений в общей очереди запросов

if(msgrcv(request\_queue, &req\_buf[resp\_no\_count], sizeof(request\_msg), prog\_id, IPC\_NOWAIT) != -1){

printf("Получен запрос от %d программы\n", req\_buf[resp\_no\_count].sender);

printf("Время запроса: %d\n", req\_buf[resp\_no\_count].time);

if(req\_buf[resp\_no\_count].time < request\_time

|| (req\_buf[resp\_no\_count].time == request\_time && req\_buf[resp\_no\_count].sender < prog\_id)){

msgsnd(req\_buf[resp\_no\_count].response\_queue, &resp\_msg, sizeof(response\_msg), 0);

permissions\_send++;

printf("Отправка разрешения %d программе\n", req\_buf[resp\_no\_count].sender);

}else{

resp\_no\_count++;

}

}

//проверка сообщений в локальной очереди

if(msgrcv(response\_queue, &resp\_buf, sizeof(response\_msg), 1, IPC\_NOWAIT) != -1)

{

permissions\_get++;

printf("Получено разрешение от %d программы\n", resp\_buf.sender);

printf("Время ответа: %ld\n", time(0));

}

}

// чтение файла

FILE \*fp = fopen("text.txt", "r");

char arr[124];

printf("\nЧтение файла\n");

while (fgets(arr, 124, fp) != NULL)

{

printf("%s", arr);

}

puts("Чтение завершено\n");

fclose(fp);

// обслуживание запросов без ответа

while(resp\_no\_count > 0){

msgsnd(req\_buf[resp\_no\_count - 1].response\_queue, &resp\_msg, sizeof(response\_msg), 0);

permissions\_send++;

printf("Отправка разрешения %d программе\n", req\_buf[resp\_no\_count - 1].sender);

resp\_no\_count--;

}

while(permissions\_send < 2){

if(msgrcv(request\_queue, &req\_buf[resp\_no\_count], sizeof(request\_msg), prog\_id, IPC\_NOWAIT) != -1){

msgsnd(req\_buf[resp\_no\_count].response\_queue, &resp\_msg, sizeof(response\_msg), 0);

permissions\_send++;

printf("Отправка разрешения %d программе\n", req\_buf[resp\_no\_count].sender);

}

}

//удаление очередей

req\_msg.mtype = 4;

msgsnd(request\_queue, &req\_msg, sizeof(request\_msg), 0);

msgctl(response\_queue, IPC\_RMID, 0);

puts("\nЛокальная очередь ответов удалена");

if(is\_creator){

while(readers\_now > 0){

if(msgrcv(request\_queue, &req\_msg, sizeof(request\_msg), 4, 0) != -1){

readers\_now--;

}

}

msgctl(request\_queue, IPC\_RMID, 0);

puts("Общая очередь запросов удалена");

}

return 0;

}

Вывод

В процессе выполнения лабораторной работы познакомились с механизмом обмена данными через программный канал и системными вызовами, обеспечивающими такой обмен.