

# **Отчёт по лабораторной работе № 14**

**Именованные каналы**

Нати Франсиску Бунда

# Содержание

<b>1</b>	<b>Цель работы</b>	<b>4</b>
<b>2</b>	<b>Задание</b>	<b>5</b>
<b>3</b>	<b>Выполнение лабораторной работы</b>	<b>6</b>
3.1	Внесение изменений в программы . . . . .	6
<b>4</b>	<b>Выводы</b>	<b>10</b>
<b>5</b>	<b>Ответы на контрольные вопросы</b>	<b>11</b>

## Список иллюстраций

3.1	Создание файлов . . . . .	6
3.2	Файл “common.h” . . . . .	7
3.3	Файл “server.c” . . . . .	7
3.4	Файл “client.c” . . . . .	8
3.5	Файл “Makefile” . . . . .	8
3.6	Компиляция . . . . .	8
3.7	Проверка . . . . .	9
3.8	Проверка . . . . .	9

# **1 Цель работы**

Приобретение практических навыков работы с именованными каналами.

## 2 Задание

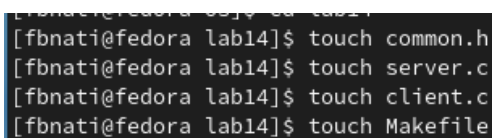
Изучить приведённые в тексте программы `server.c` и `client.c`. Взяв данные примеры за образец, написать аналогичные программы, внося следующие изменения: 1. Работает не 1 клиент, а несколько (например, два). 2. Клиенты передают текущее время с некоторой периодичностью (например, раз в пять секунд). Используйте функцию `sleep()` для приостановки работы клиента. 3. Сервер работает не бесконечно, а прекращает работу через некоторое время (например, 30 сек). Используйте функцию `clock()` для определения времени работы сервера. Что будет в случае, если сервер завершит работу, не закрыв канал?

## 3 Выполнение лабораторной работы

### 3.1 Внесение изменений в программы

Для начала изучили материал лабораторной работы. Далее на основе примеров написали аналогичные программы, но с изменениями.

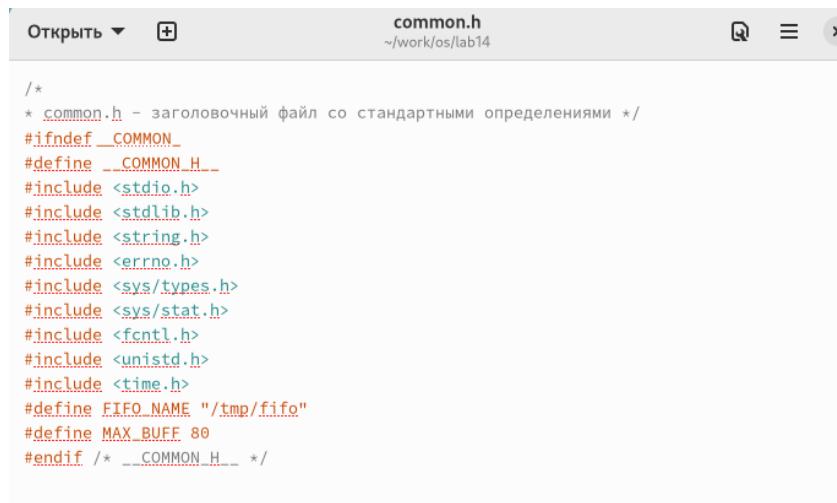
Для начала создали необходимые файлы для работы. (рис. 3.1).



```
[fbnati@fedora lab14]$ touch common.h
[fbnati@fedora lab14]$ touch server.c
[fbnati@fedora lab14]$ touch client.c
[fbnati@fedora lab14]$ touch Makefile
```

Рис. 3.1: Создание файлов

Затем изменили коды программ, данных в лабораторной работе. В файл `common.h` добавили стандартные заголовочные файлы: “`unistd.h`”, “`time.h`”. Это необходимо для работы других файлов. Этот файл является заголовочным, чтобы в остальных программах не прописывать одно и то же каждый раз.(рис. 3.2).



```

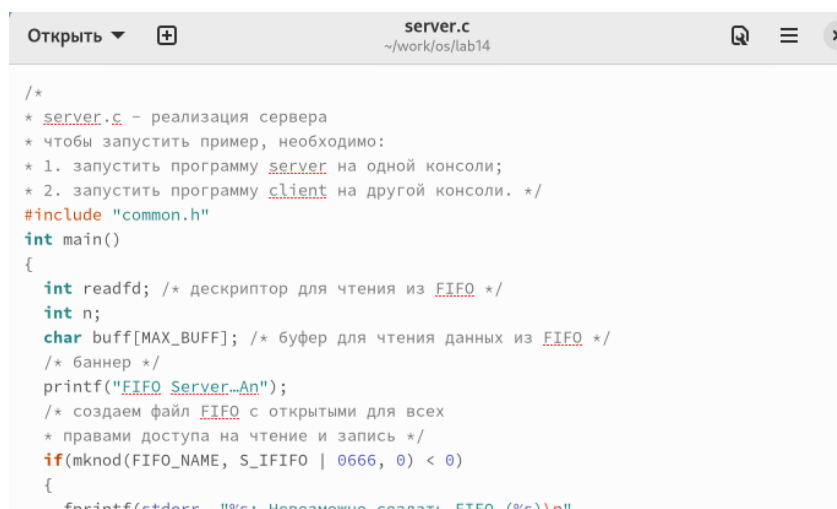
Открыть ▾ + common.h
~/work/os/lab14

/*
 * common.h - заголовочный файл со стандартными определениями */
#ifndef __COMMON__
#define __COMMON__
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#define FIFO_NAME "/tmp/fifo"
#define MAX_BUFF 80
#endif /* __COMMON__ */

```

Рис. 3.2: Файл “common.h”

Затем в файл server.c добавляем цикл “while” для контроля за временем работы сервера. Разница между текущим временем и началом работы не должна превышать 30 секунд.(рис. 3.3).



```

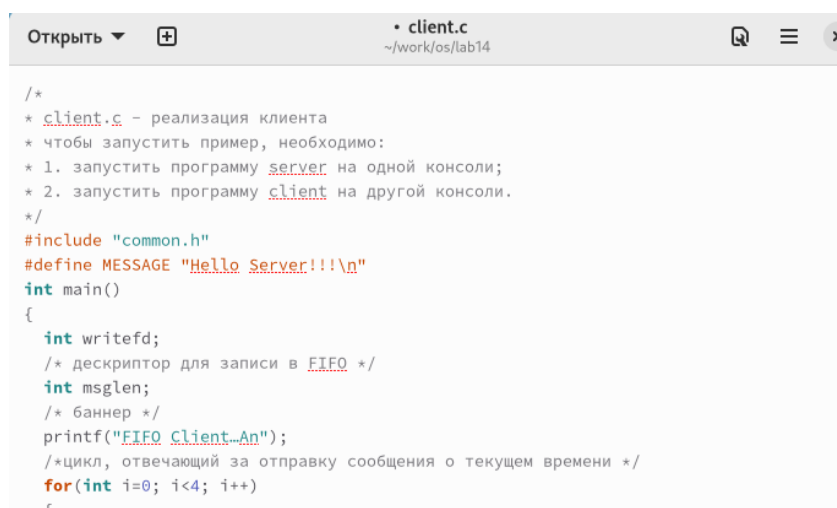
Открыть ▾ + server.c
~/work/os/lab14

/*
 * server.c - реализация сервера
 * чтобы запустить пример, необходимо:
 * 1. запустить программу server на одной консоли;
 * 2. запустить программу client на другой консоли. */
#include "common.h"
int main()
{
    int readfd; /* дескриптор для чтения из FIFO */
    int n;
    char buff[MAX_BUFF]; /* буфер для чтения данных из FIFO */
    /* баннер */
    printf("FIFO Server-An");
    /* создаем файл FIFO с открытыми для всех
     * правами доступа на чтение и запись */
    if(mknod(FIFO_NAME, S_IFIFO | 0666, 0) < 0)
    {
        fprintf(stderr, "%s: Невозможно создать FIFO (%s)\n",

```

Рис. 3.3: Файл “server.c”

В файл client.c добавим цикл, который отвечает за количество сообщений о текущем времени (4 сообщения). С помощью команды “sleep” приостановим работу клиента на 5 секунд.рис. 3.4).



```

/*
 * client.c - реализация клиента
 * чтобы запустить пример, необходимо:
 * 1. запустить программу server на одной консоли;
 * 2. запустить программу client на другой консоли.
 */
#include "common.h"
#define MESSAGE "Hello Server!!!\n"
int main()
{
    int writefd;
    /* дескриптор для записи в FIFO */
    int msglen;
    /* баннер */
    printf("FIFO Client..An");
    /*цикл, отвечающий за отправку сообщения о текущем времени */
    for(int i=0; i<4; i++)
    {

```

Рис. 3.4: Файл “client.c”

Makefile оставили без изменений.(рис. 3.5).



```

all: server client

server: server.c common.h
    gcc server.c -o server

client: client.c common.h
    gcc client.c -o client

clean:
    -rm server client *.o

```

Рис. 3.5: Файл “Makefile”

Далее делаем компиляцию файлов с помощью команды “make all”.(рис. 3.6).



```

gcc server.c -o server
gcc client.c -o client

```

Рис. 3.6: Компиляция



Затем открываем три терминала для проверки работы наших файлов. В первом пишем “./server”, а в остальных “./client”. В результате каждый терминал вывел по 4 сообщения, а по истечении 30 секунд работа сервера была завершена. Всё работает верно. (рис. 3.7).

```

[fbnati@fedora lab14]$ ./client
client.c: Невозможно открыть FIFO (No such file or directory)
FIFO Client_An[fbnati@fedora lab14]$ ./client
FIFO Client_An[fbnati@fedora lab14]$

[fbnati@fedora lab14]$ ./server
Hello Server!!!
Hello Server!!!
Hello Server!!!
Hello Server!!!
Hello Server!!!
Hello Server!!!
Hello Server!!!
FIFO Server_An[fbnati@fedora lab14]$

[fbnati@fedora lab14]$ ./client
Client_An[fbnati@fedora lab14]$

```

Рис. 3.7: Проверка

Проверим длительность работы сервера. Вводим команду “./server” в одном терминале. Он завершил свою работу через 30 секунд. Если сервер завершит свою работу, не закрывая канал, то при повторном запуске появится ошибка “Невозможно создать FIFO”, так как у нас уже есть один канал.(рис. 3.8).

```

[fbnati@fedora lab14]$ ./server
server.c: Невозможно создать FIFO (File exists)
FIFO Server_An[fbnati@fedora lab14]$

```

Рис. 3.8: Проверка

## **4 Выводы**

В ходе выполнения были приобретены навыки работы с именованными каналами.

## 5 Ответы на контрольные вопросы

1. Именованные каналы отличаются от неименованных наличием идентификатора канала, который представлен как специальный файл (соответственно имя именованного канала – это имя файла). Поскольку файл находится на локальной файловой системе, данное IPC используется внутри одной системы.
2. Чтобы создать неименованный канал из командной строки нужно использовать символ |, служащий для объединения двух и более процессов: процесс\_1 | процесс\_2 | процесс\_3...
3. Чтобы создать именованный канал из командной строки нужно использовать либо команду «mknod», либо команду «mkfifo».
4. Неименованный канал является средством взаимодействия между связанными процессами – родительским и дочерним. Родительский процесс создает канал при помощи системного вызова: «int pipe(int fd[2]);». Массив из двух целых чисел является выходным параметром этого системного вызова. Если вызов выполнен нормально, то этот массив содержит два файловых дескриптора. fd[0] является дескриптором для чтения из канала, fd[1] – дескриптором для записи в канал. Когда процесс порождает другой процесс, дескрипторы родительского процесса наследуются дочерним процессом, и, таким образом, прокладывается трубопровод между двумя процессами. Естественно, что один из процессов использует канал только для чтения, а другой – только для записи. Поэтому, если, например, через канал должны

передаваться данные из родительского процесса в дочерний, родительский процесс сразу после запуска дочернего процесса закрывает дескриптор канала для чтения, а дочерний процесс закрывает дескриптор для записи. Если нужен двунаправленный обмен данными между процессами, то родительский процесс создает два канала, один из которых используется для передачи данных в одну сторону, а другой – в другую.

5. Файлы именованных каналов создаются функцией `mkfifo()` или функцией `mknod`:

«`int mkfifo(const char *pathname, mode_t mode);`», где первый параметр – путь, где будет располагаться FIFO (имя файла, идентифицирующего канал), второй параметр определяет режим работы с FIFO (маска прав доступа к файлу),

«`mknod (namefile, IFIFO | 0666, 0)`», где `namefile` – имя канала, `0666` – к каналу разрешен доступ на запись и на чтение любому запросившему процессу),

«`int mknod(const char *pathname, mode_t mode, dev_t dev);`». Функция `mkfifo()` создает канал и файл соответствующего типа. Если указанный файл канала уже существует, `mkfifo()` возвращает -1. После создания файла канала процессы, участвующие в обмене данными, должны открыть этот файл либо для записи, либо для чтения.

6. При чтении меньшего числа байтов, чем находится в канале или FIFO, возвращается требуемое число байтов, остаток сохраняется для последующих чтений. При чтении большего числа байтов, чем находится в канале или FIFO, возвращается доступное число байтов. Процесс, читающий из канала, должен соответствующим образом обработать ситуацию, когда прочитано меньше, чем заказано.
7. Запись числа байтов, меньшего емкости канала или FIFO, гарантированно атомарно. Это означает, что в случае, когда несколько процессов одновременно записывают в канал, порции данных от этих процессов не перемешиваются. При записи большего числа байтов, чем это позволяет канал или

FIFO, вызов `write(2)` блокируется до освобождения требуемого места. При этом атомарность операции не гарантируется. Если процесс пытается записать данные в канал, не открытый ни одним процессом на чтение, процессу генерируется сигнал `SIGPIPE`, а вызов `write(2)` возвращает 0 с установкой ошибки (`errno=ERRPIPE`) (если процесс не установил обработки сигнала `SIGPIPE`, производится обработка по умолчанию – процесс завершается).

8. Количество процессов, которые могут параллельно присоединяться к любому концу канала, не ограничено. Однако если два или более процесса записывают в канал данные одновременно, каждый процесс за один раз может записать максимум `PIPE BUF` байтов данных. Предположим, процесс (назовем его А) пытается записать X байтов данных в канал, в котором имеется место для Y байтов данных. Если X больше, чем Y, только первые Y байтов данных записываются в канал, и процесс блокируется. Запускается другой процесс (например, В); в это время в канале появляется свободное пространство (благодаря третьему процессу, считывающему данные из канала). Процесс В записывает данные в канал. Затем, когда выполнение процесса А возобновляется, он записывает оставшиеся X-Y байтов данных в канал. В результате данные в канал записываются поочередно двумя процессами. Аналогичным образом, если два (или более) процесса одновременно попытаются прочитать данные из канала, может случиться так, что каждый из них прочитает только часть необходимых данных.
9. Функция `write` записывает байты `count` из буфера `buffer` в файл, связанный с `handle`. Операции `write` начинаются с текущей позиции указателя на файл (указатель ассоциирован с заданным файлом). Если файл открыт для добавления, операции выполняются в конец файла. После осуществления операций записи указатель на файл (если он есть) увеличивается на количество действительно записанных байтов. Функция `write` возвращает число действительно записанных байтов. Возвращаемое значение должно быть

положительным, но меньше числа count (например, когда размер для записи count байтов выходит за пределы пространства на диске). Возвращаемое значение -1 указывает на ошибку; errno устанавливается в одно из следующих значений: EACCES – файл открыт для чтения или закрыт для записи, EBADF – неверный handle-р файла, ENOSPC – на устройстве нет свободного места. Единица в вызове функции write в программе server.c означает идентификатор (дескриптор потока) стандартного потока вывода.

10. Прототип функции strerror: «char \* strerror( int errornum );». Функция strerror интерпретирует номер ошибки, передаваемый в функцию в качестве аргумента -errornum, в понятное для человека текстовое сообщение (строку). Откуда берутся эти ошибки? Ошибки эти возникают при вызове функций стандартных Си-библиотек. То есть хорошим тоном программирования будет – использование этой функции в паре с другой, и если возникнет ошибка, то пользователь или программист поймет, как исправить ошибку, прочитав сообщение функции strerror. Возвращенный указатель ссылается на статическую строку с ошибкой, которая не должна быть изменена программой. Дальнейшие вызовы функции strerror перезапишут содержание этой строки. Интерпретированные сообщения об ошибках могут различаться, это зависит от платформы и компилятора.