

# Лабораторная работа №14

Именованные каналы

Кузнецова София Вадимовна

# Содержание

Цель работы	5
Задание	6
Выполнение лабораторной работы	7
Контрольные вопросы	11
Выводы	15

## Список иллюстраций

0.1	Создание файлов в домашнем каталоге . . . . .	7
0.2	Перенос скрипта для <code>common.h</code> . . . . .	7
0.3	Перенос скрипта для <code>server.c</code> . . . . .	8
0.4	Перенос скрипта для <code>client.c</code> . . . . .	9
0.5	Перенос скрипта для <code>Makefile</code> . . . . .	9
0.6	Проверка выполнения . . . . .	10

## Список таблиц

## Цель работы

Приобретение практических навыков работы с именованными каналами.

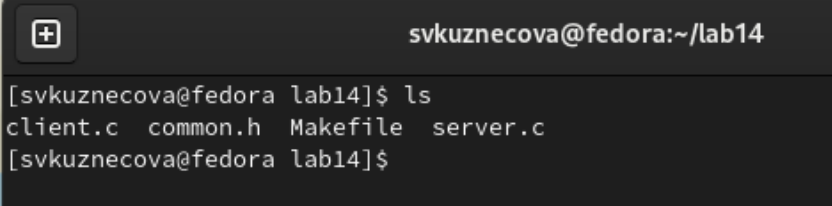
# Задание

Изучите приведённые в тексте программы `server.c` и `client.c`. Взяв данные примеры за образец, напишите 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?

# Выполнение лабораторной работы

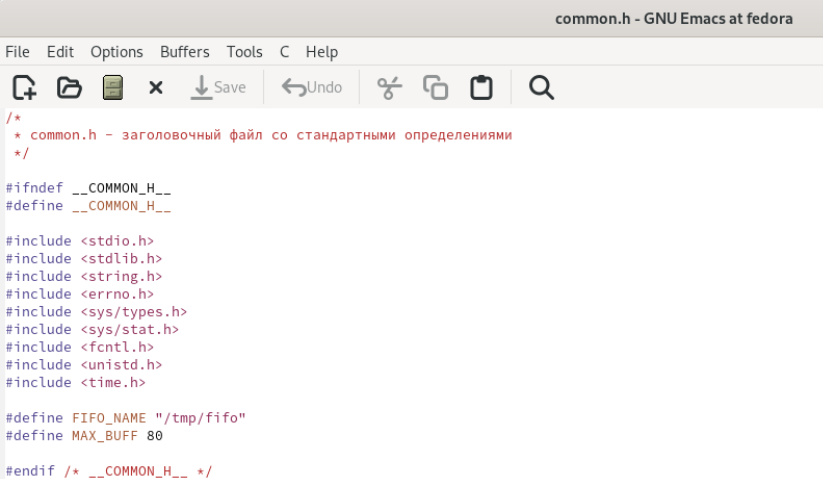
В домашнем каталоге создадим файлы `common.h`, `server.c`, `client.c` и `Makefile`, выполним проверку. Перейдём в `emacs`.

A terminal window with a dark background. The prompt is `svkuznecova@fedora:~/lab14`. The user enters `ls` and the output is `client.c common.h Makefile server.c`. The prompt returns to `[svkuznecova@fedora lab14]$`.

```
[svkuznecova@fedora lab14]$ ls
client.c common.h Makefile server.c
[svkuznecova@fedora lab14]$
```

Рис. 0.1: Создание файлов в домашнем каталоге

В `emacs` откроем созданный файл `common.h` и приступим к переносу в него скрипта из файла, а также внесём в него дополнительные изменения.

A screenshot of the GNU Emacs editor. The title bar says "common.h - GNU Emacs at fedora". The menu bar includes File, Edit, Options, Buffers, Tools, C, and Help. The toolbar has icons for opening, saving, undo, redo, and search. The text area shows the content of `common.h`, which is a header file with standard definitions and includes.

```
/*
 * common.h - заголовочный файл со стандартными определениями
 */

#ifndef __COMMON_H__
#define __COMMON_H__

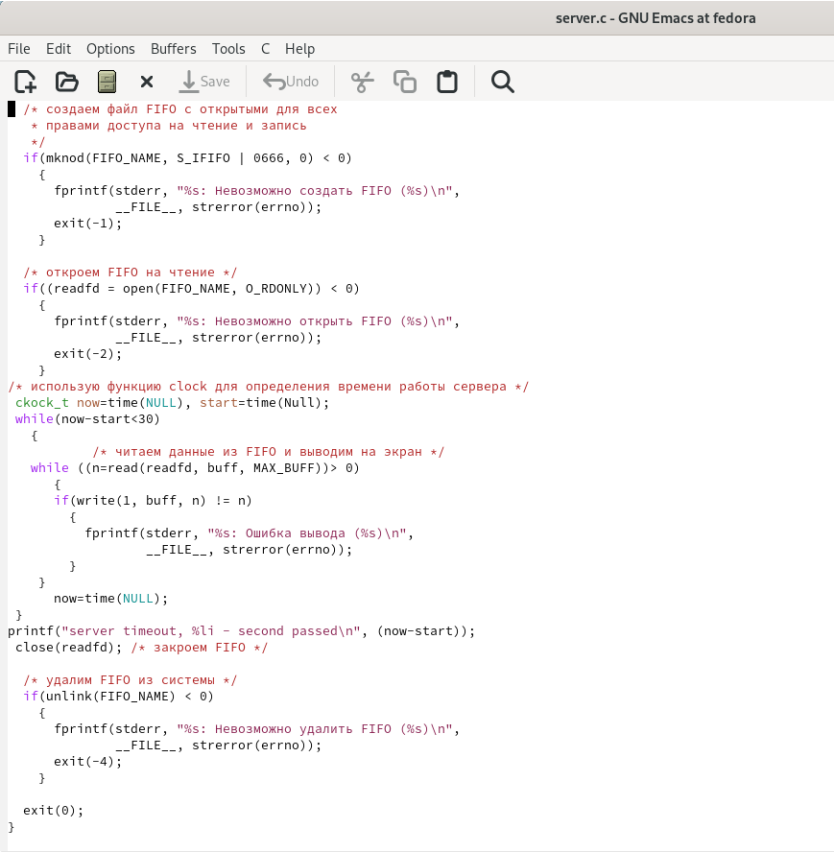
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>

#define FIFO_NAME "/tmp/fifo"
#define MAX_BUFF 80

#endif /* __COMMON_H__ */
```

Рис. 0.2: Перенос скрипта для `common.h`

После того как мы перенесли, изменили и сохранили скрипт для первого файла, открываем файл `server.c` и также переносим в него скрипт с изменениями, но уже для второго файла. Выполняем сохранение.



```
server.c - GNU Emacs at fedora
File Edit Options Buffers Tools C Help
[Icons: Open, Save, Undo, Cut, Copy, Paste, Find]

/* 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",
        __FILE__, strerror(errno));
    exit(-1);
}

/* откроем FIFO на чтение */
if((readfd = open(FIFO_NAME, O_RDONLY)) < 0)
{
    fprintf(stderr, "%s: Невозможно открыть FIFO (%s)\n",
        __FILE__, strerror(errno));
    exit(-2);
}

/* использую функцию clock для определения времени работы сервера */
clock_t now=time(NULL), start=time(NULL);
while(now-start<30)
{
    /* читаем данные из FIFO и выводим на экран */
    while ((n=read(readfd, buff, MAX_BUFF))> 0)
    {
        if(write(1, buff, n) != n)
        {
            fprintf(stderr, "%s: Ошибка вывода (%s)\n",
                __FILE__, strerror(errno));
        }
    }
    now=time(NULL);
}
printf("server timeout, %li - second passed\n", (now-start));
close(readfd); /* закроем FIFO */

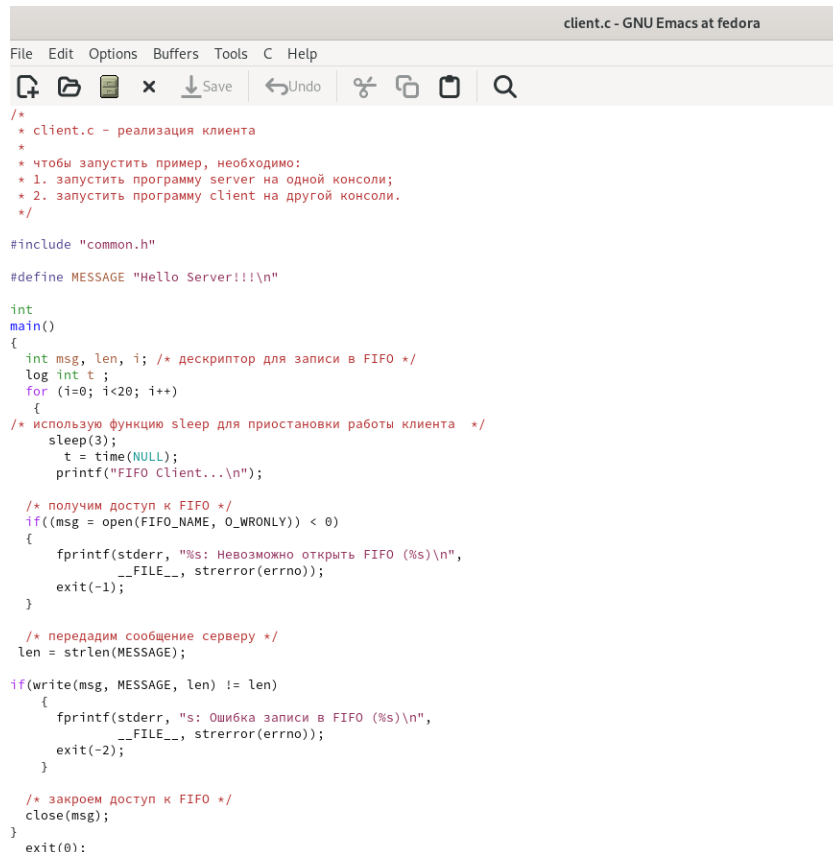
/* удалим FIFO из системы */
if(unlink(FIFO_NAME) < 0)
{
    fprintf(stderr, "%s: Невозможно удалить FIFO (%s)\n",
        __FILE__, strerror(errno));
    exit(-4);
}

exit(0);
}
```

Рис. 0.3: Перенос скрипта для `server.c`

Теперь нам нужно перенести третий скрипт в файл `client.c`. После чего также выполняем сохранение.





```
/*
 * 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 msg, len, i; /* дескриптор для записи в FIFO */
    log int t;
    for (i=0; i<20; i++)
    {
        /* использую функцию sleep для приостановки работы клиента */
        sleep(3);
        t = time(NULL);
        printf("FIFO Client...\n");

        /* получим доступ к FIFO */
        if((msg = open(FIFO_NAME, O_WRONLY)) < 0)
        {
            fprintf(stderr, "%s: Невозможно открыть FIFO (%s)\n",
                    __FILE__, strerror(errno));
            exit(-1);
        }

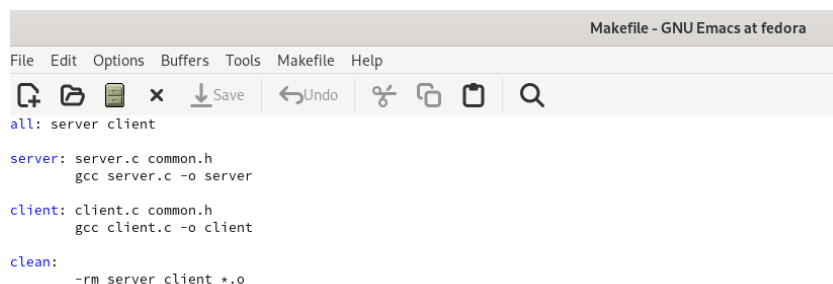
        /* передадим сообщение серверу */
        len = strlen(MESSAGE);

        if(write(msg, MESSAGE, len) != len)
        {
            fprintf(stderr, "%s: Ошибка записи в FIFO (%s)\n",
                    __FILE__, strerror(errno));
            exit(-2);
        }

        /* закроем доступ к FIFO */
        close(msg);
    }
    exit(0);
}
```

Рис. 0.4: Перенос скрипта для client.c

Выполняем перенос скрипта для последнего, четвёртого файла Makefile и закрываем emacs.



```
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. 0.5: Перенос скрипта для Makefile

В терминале выполняем команду `make all`. После чего в одном терминале запустим команду `./server`, а в других `./client` и проверим корректность выполнения.

```
[ismahorin@fedora ~]$ make all
gcc client.c -o client
[ismahorin@fedora ~]$ ./server
FIFO Server...
Hello Server!!!
Hello Server!!!
Hello Server!!!
Hello Server!!!
Hello Server!!!
Hello Server!!!
Hello Server!!!
Hello Server!!!
[ismahorin@fedora ~]$
```

```
[ismahorin@fedora ~]$ ./client
FIFO client...
[ismahorin@fedora ~]$
```

Рис. 0.6: Проверка выполнения

# Контрольные вопросы

1. В чем ключевое отличие именованных каналов от неименованных?

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. Возможно ли создание неименованного канала из командной строки?

Чтобы создать неименованный канал из командной строки нужно использовать символ `|`, служащий для объединения двух и более процессов: процесс 1 процесс 2 процесс 3

3. Возможно ли создание именованного канала из командной строки?

Чтобы создать именованный канал из командной строки нужно использовать либо команду `mknod`, либо команду `mkfifo`.

4. Опишите функцию языка C, создающую неименованный канал.

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. Опишите функцию языка C, создающую именованный канал.

Файлы именованных каналов создаются функцией `mkfifo()` или функцией `mknod`:  
1. «`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),  
2. «`mknod (namefile, IFIFO | 0666, 0)`», где `namefile` имя канала, `0666` к каналу разрешен доступ на запись и на чтение любому запросившему процессу),  
3. «`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. Что будет в случае прочтения из `fifo` меньшего числа байтов, чем находится в канале? Большее числа байтов?

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. Аналогично, что будет в случае записи в `fifo` меньшего числа байтов, чем позволяет буфер? Большого числа байтов?

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. Могут ли два и более процессов читать или записывать в канал?

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. Опишите функцию `write` (тип возвращаемого значения, аргументы и логику работы). Что означает 1 (единица) в вызове этой функции в программе `server.c` (строка 42)?

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` к файлу, `ENOSPC` на устройстве нет свободного места. Единичка в вызове функции `write` в программе `server.c` означает идентификатор (дескриптор потока) стандартного потока вывода.

10. Опишите функцию `strerror`.

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.

## Выводы

В ходе выполнения лабораторной работы были приобретены практические навыки работы с именованными каналами.