

Лабораторная работа No 14.

**Программирование в командном процессоре ОС UNIX. Расширенное
программирование**

Боровиков Даниил Александрович

Содержание

1	Цель работы	5
2	Выполнение лабораторной работы	6
3	Контрольные вопросы	11
4	Выводы	16

Список иллюстраций

2.1	Создание файлов в домашнем каталоге	6
2.2	Перенос скрипта для common.h	7
2.3	Перенос скрипта для server.c	8
2.4	Перенос скрипта для main.c	9
2.5	Перенос скрипта для Makefile	10
2.6	Проверка выполнения	10

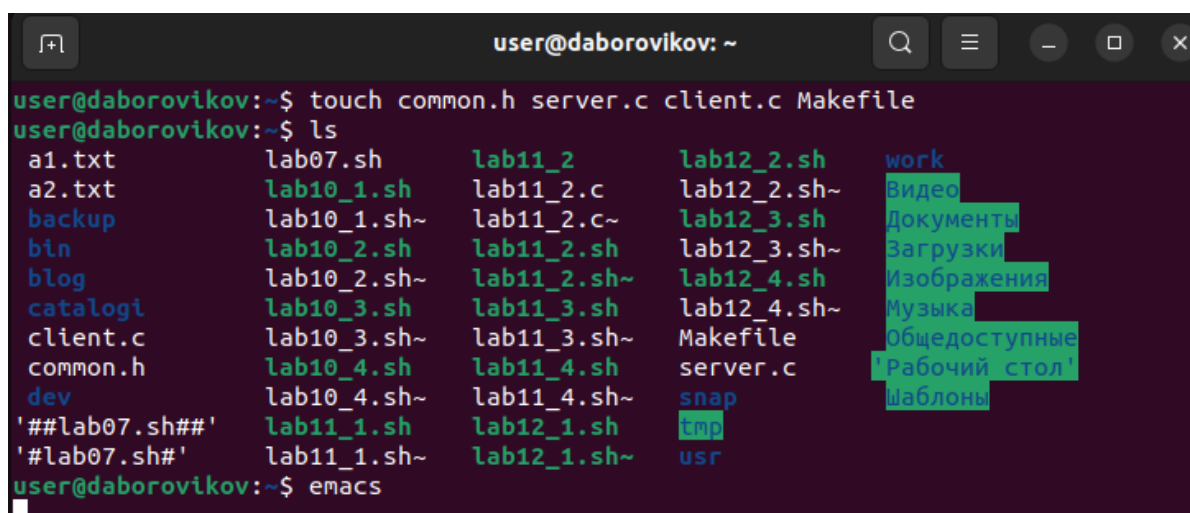
Список таблиц

1 Цель работы

Приобрести простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования С калькулятора с простейшими функциями

2 Выполнение лабораторной работы

В домашнем каталоге создадим файлы `common.h`, `server.c`, `client.c` и `Makefile`, выполним проверку. Перейдём в `emacs` (Рис. 2.1).

A terminal window titled 'user@daborovikov: ~' with standard window controls. The user has executed 'touch common.h server.c client.c Makefile' and then 'ls'. The output of 'ls' is a multi-column listing of files and directories in the home directory, including lab files, source code, and system directories. The prompt is now 'user@daborovikov:~\$ emacs'.

```
user@daborovikov:~$ touch common.h server.c client.c Makefile
user@daborovikov:~$ ls
a1.txt      lab07.sh    lab11_2     lab12_2.sh  work
a2.txt      lab10_1.sh  lab11_2.c   lab12_2.sh~ Видео
backup      lab10_1.sh~ lab11_2.c~   lab12_3.sh  Документы
bin         lab10_2.sh  lab11_2.sh   lab12_3.sh~ Загрузки
blog        lab10_2.sh~ lab11_2.sh~   lab12_4.sh  Изображения
catalogi    lab10_3.sh  lab11_3.sh   lab12_4.sh~ Музыка
client.c    lab10_3.sh~ lab11_3.sh~   Makefile    Общедоступные
common.h    lab10_4.sh  lab11_4.sh   server.c     'Рабочий стол'
dev         lab10_4.sh~ lab11_4.sh~   snap        Шаблоны
'##lab07.sh##' lab11_1.sh  lab12_1.sh   tmp
'#lab07.sh#'   lab11_1.sh~ lab12_1.sh~   usr
user@daborovikov:~$ emacs
```

Рис. 2.1: 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 (Рис. 2.2).

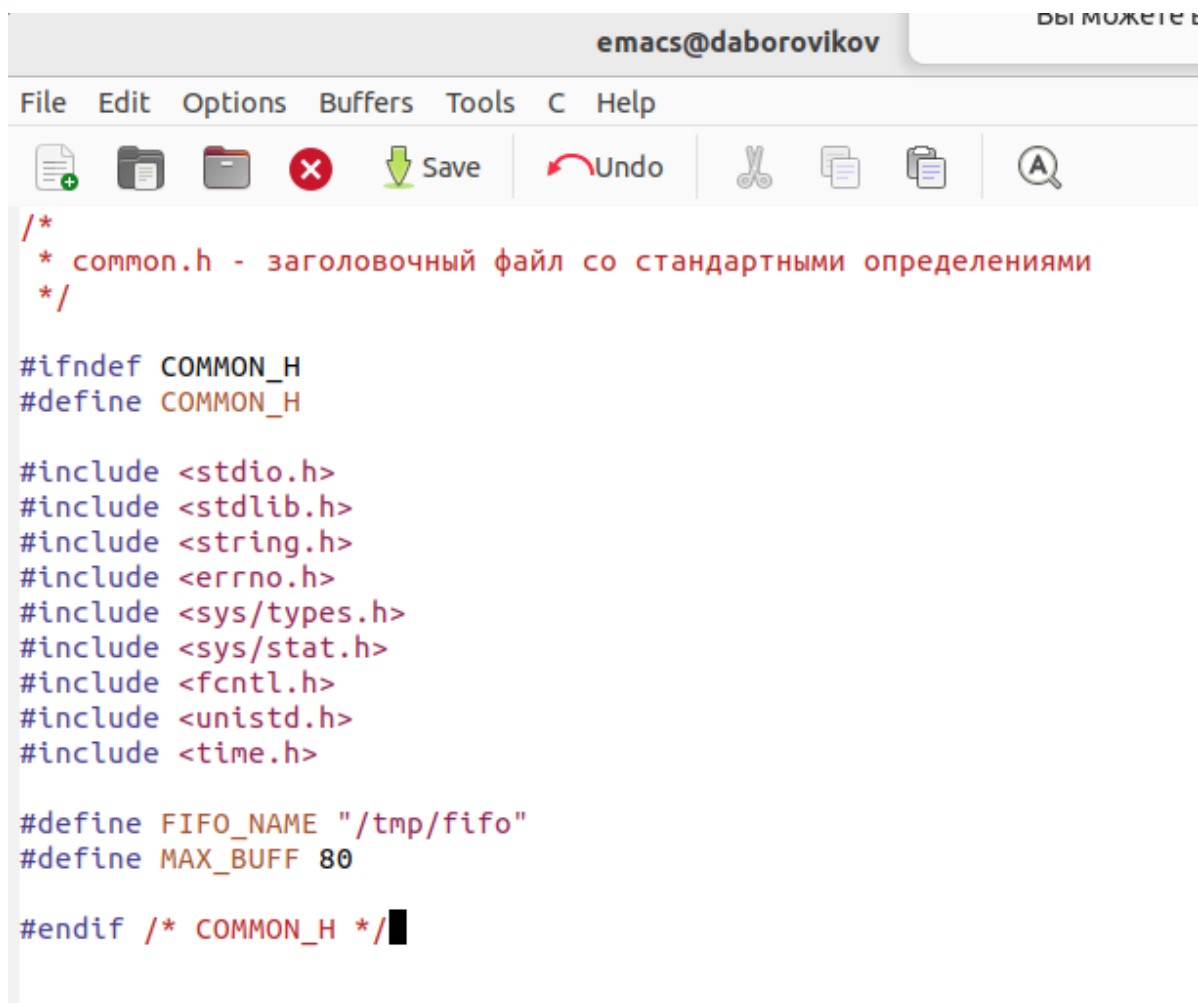
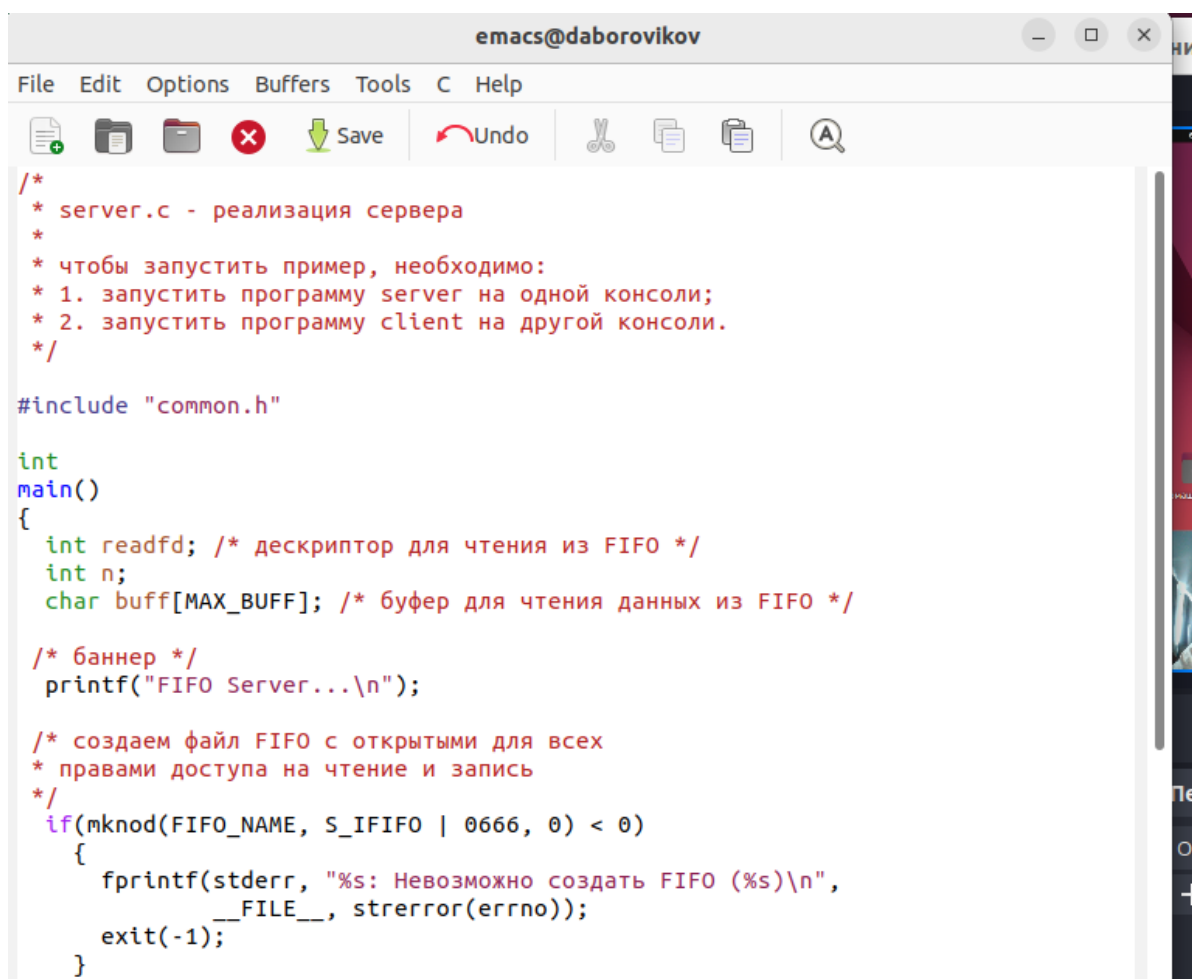


Рис. 2.2: Перенос скрипта для common.h

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 (Рис. 2.3).



```
/*
 * 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"

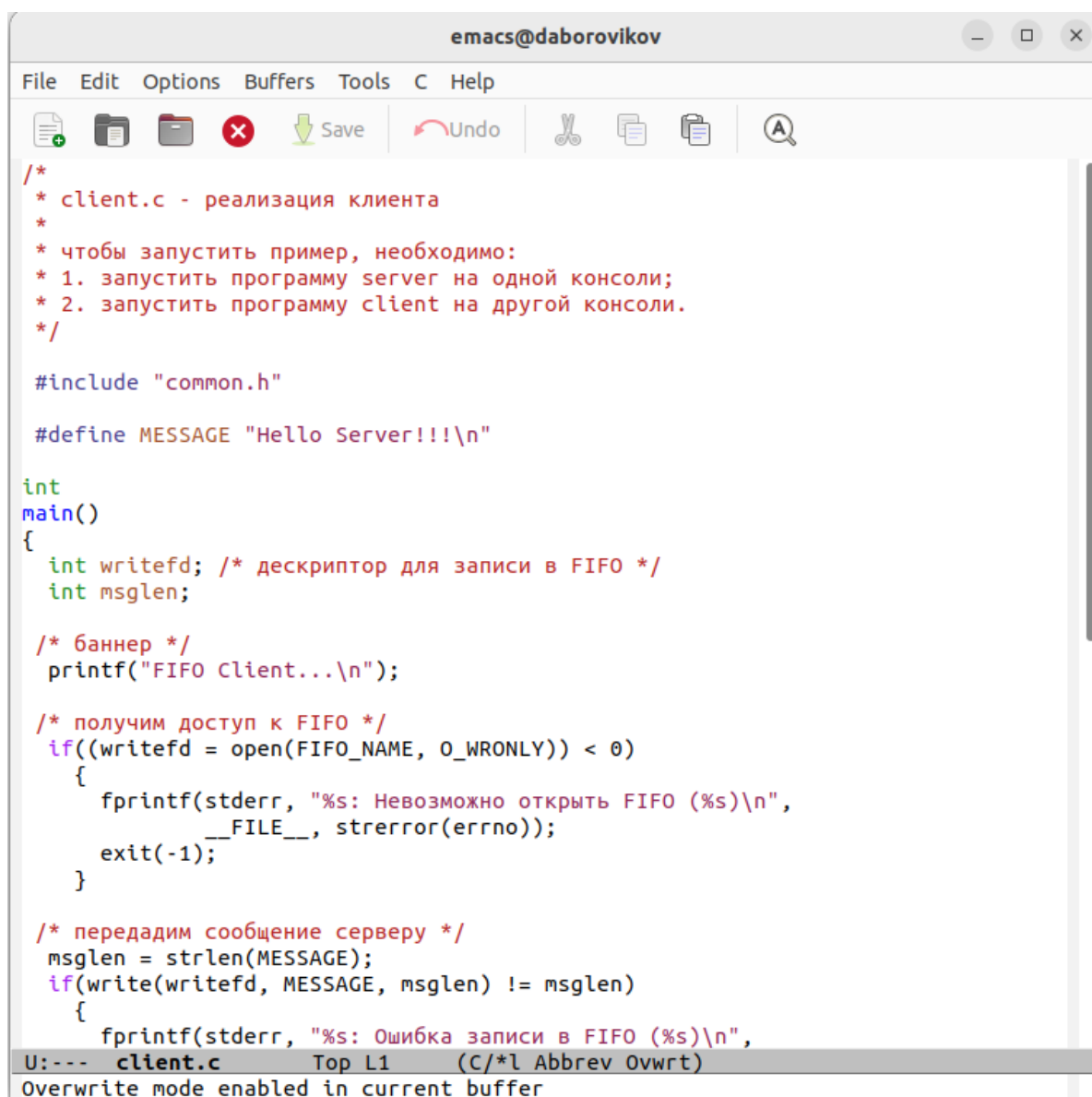
int
main()
{
    int readfd; /* дескриптор для чтения из FIFO */
    int n;
    char buff[MAX_BUFF]; /* буфер для чтения данных из FIFO */

    /* баннер */
    printf("FIFO Server...\n");

    /* 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);
    }
}
```

Рис. 2.3: Перенос скрипта для server.c

Теперь нам нужно перенести третий скрипт в файл client.c. После чего также выполняем сохранение (Рис. 2.4).



```
/*
 * 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...\n");

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

    /* передадим сообщение серверу */
    msglen = strlen(MESSAGE);
    if(write(writefd, MESSAGE, msglen) != msglen)
    {
        fprintf(stderr, "%s: Ошибка записи в FIFO (%s)\n",
                __FILE__, strerror(errno));
        exit(-1);
    }
}
```

U:--- client.c Top L1 (C/*l Abbrev Ovwrt)
Overwrite mode enabled in current buffer

Рис. 2.4: Перенос скрипта для main.c

Выполним перенос скрипта для последнего, четвёртого файла Makefile и закроем emacs (Рис. 2.5).

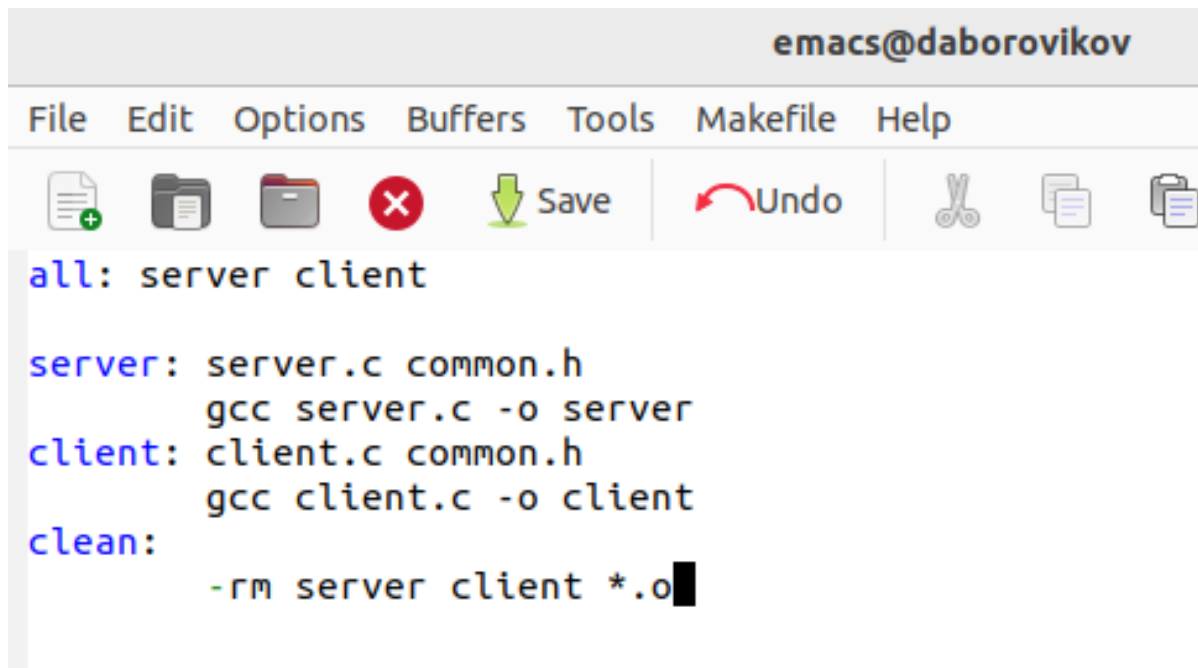


Рис. 2.5: Перенос скрипта для Makefile

В терминале выполним команду `make all`. После чего в одном терминале запустим команду `./server`, а в двух других `./client` и проверим корректность выполнения (Рис. 2.6).

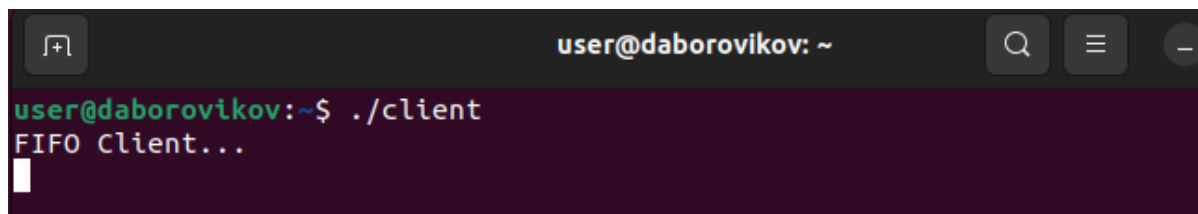


Рис. 2.6: Проверка выполнения

3 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 выполнился 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? Большого 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, воз-

вращается 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` файла, `ENOSPC` на устройстве нет свободного места. Единица 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.

4 Выводы

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