

Отчёт лабораторной работы №15

Дисциплина: Операционные системы

Касьянов Даниил Владимирович

Содержание

| | | |
|----------|---------------------------------------------|-----------|
| 1 | Цель работы | 5 |
| 2 | Последовательность выполнения работы | 6 |
| 3 | Выполнение лабораторной работы | 7 |
| 4 | Контрольные вопросы | 14 |
| 5 | Выводы | 18 |
| 6 | Библиография | 19 |

Список таблиц

Список иллюстраций

1 Цель работы

Приобретение практических навыков работы с именованными каналами.

2 Последовательность выполнения работы

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?

3 Выполнение лабораторной работы

В домашнем каталоге создаю файлы **common.h**, **server.c**, **client.c**, **Makefile**. Открываю их в редакторе **emacs** и копирую в них коды из лабораторной работы (Рисунок 1).

```
dvkasjyanov@dvkasjyanov:~$ touch common.h server.c client.c Makefile
dvkasjyanov@dvkasjyanov:~$ emacs &
```

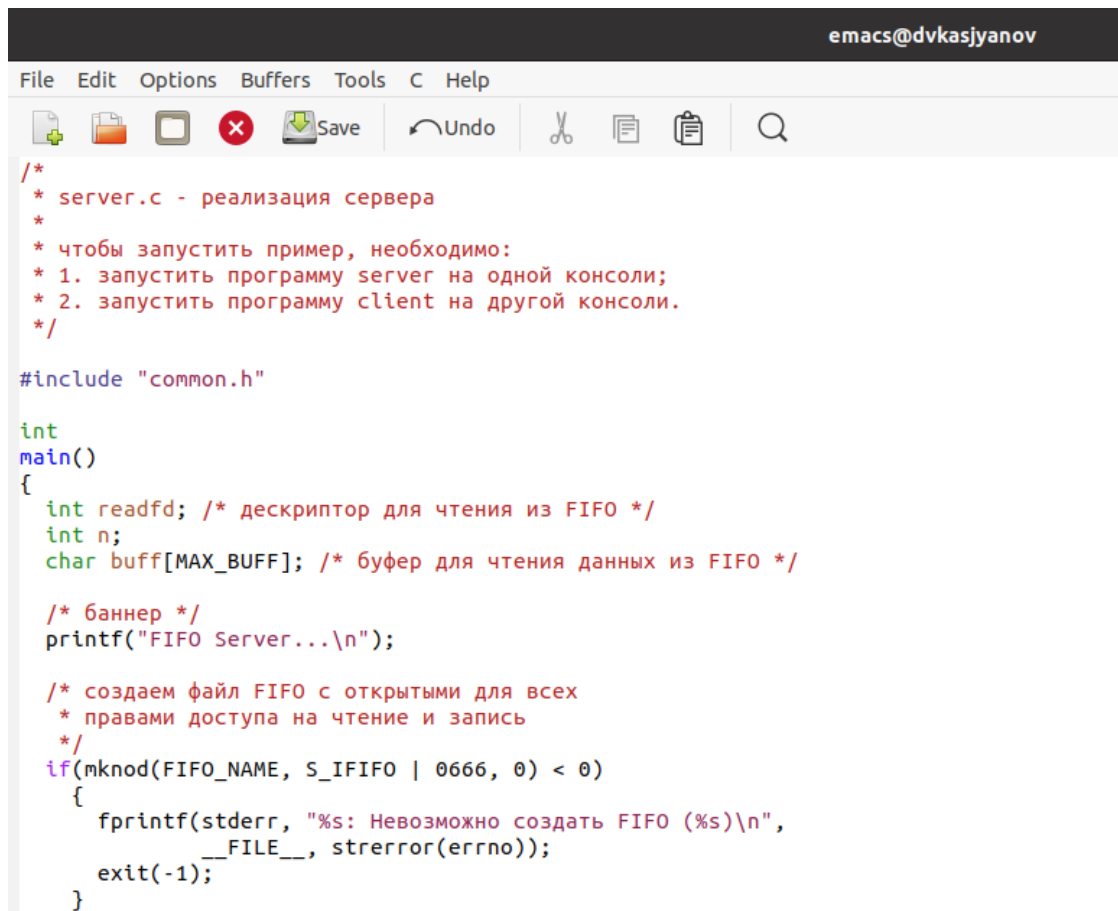
(Рисунок 1)

Вношу изменения в файлы.

- **common.h**: добавляю стандартные заголовочные файлы **unistd.h** и **time.h**, необходимые для работы кодов других файлов (Рисунок 2).

(Рисунок 2)

- **server.c**: добавляю цикл **while** для контроля времени работы сервера. Разница между текущим временем **time(NULL)** и временем начала работы **clock_t start=time(NULL)** не превышает 30 секунд (Рис. 3, 4, 5).



```
/*
 * server.c - реализация сервера
 *
 * чтобы запустить пример, необходимо:
 * 1. запустить программу server на одной консоли;
 * 2. запустить программу client на другой консоли.
 */

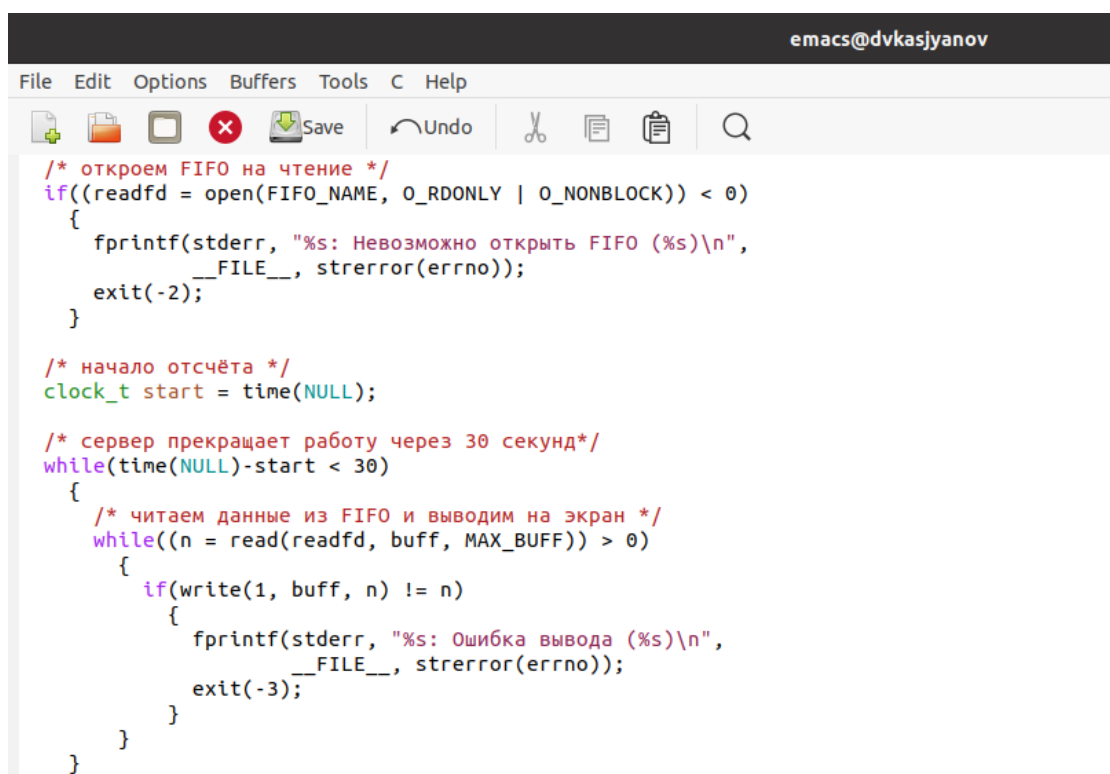
#include "common.h"

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);
    }
}
```

(Рисунок 3)



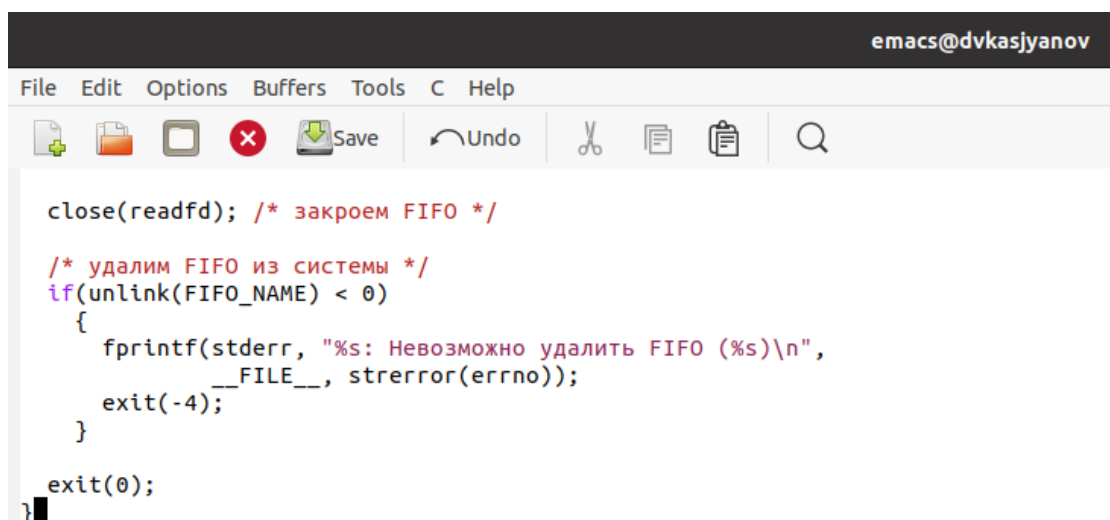
The screenshot shows the Emacs editor interface with the title bar 'emacs@dvkasjyanov'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'C', and 'Help'. The toolbar contains icons for file operations and editing. The code in the buffer is as follows:

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

/* начало отсчёта */
clock_t start = time(NULL);

/* сервер прекращает работу через 30 секунд*/
while(time(NULL)-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));
            exit(-3);
        }
    }
}
```

(Рисунок 4)



The screenshot shows the Emacs editor interface with the title bar 'emacs@dvkasjyanov'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'C', and 'Help'. The toolbar contains icons for file operations and editing. The code in the buffer is as follows:

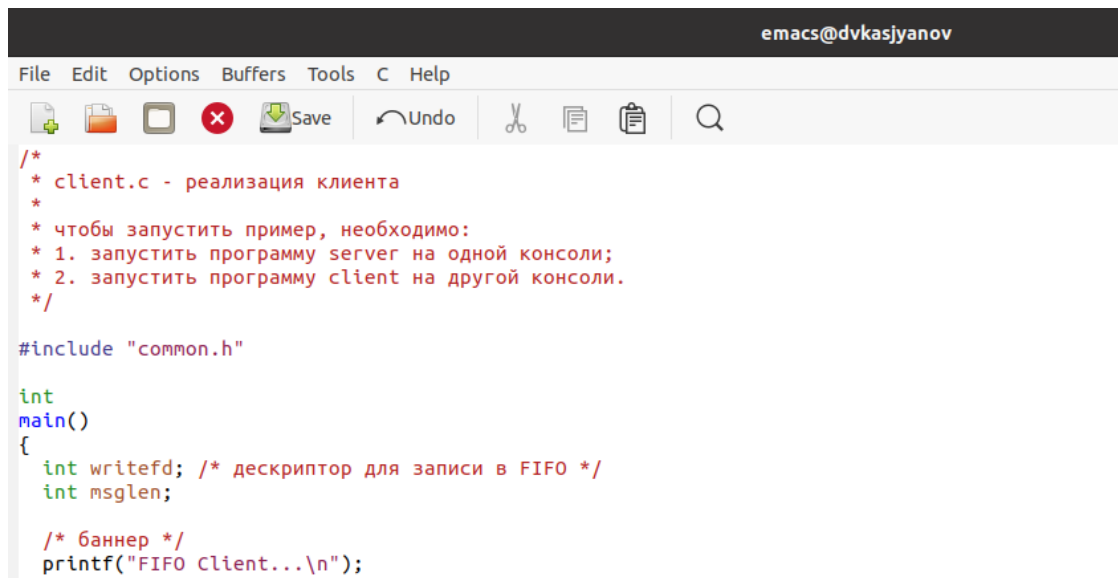
```
close(readfd); /* закроем FIFO */

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

exit(0);
}
```

(Рисунок 5)

- **cclient.c**: удаляю строку с переменной **MESSAGE**, добавляю цикл, который отвечает за количество сообщений о текущем времени (6 сообщений) и команду **sleep(5)** для приостановки работы клиента на 5 секунд (Рис. 6, 7).



The image shows a screenshot of the Emacs text editor. The title bar at the top reads "emacs@dvkasjyanov". The menu bar includes "File", "Edit", "Options", "Buffers", "Tools", "C", and "Help". The toolbar contains icons for saving, undo, and search, along with text labels "Save" and "Undo". The main editing area displays C code for a client implementation. The code includes comments in Russian explaining how to run the program, followed by a preprocessor directive to include "common.h", and the start of a main function with variable declarations and an initial printf statement.

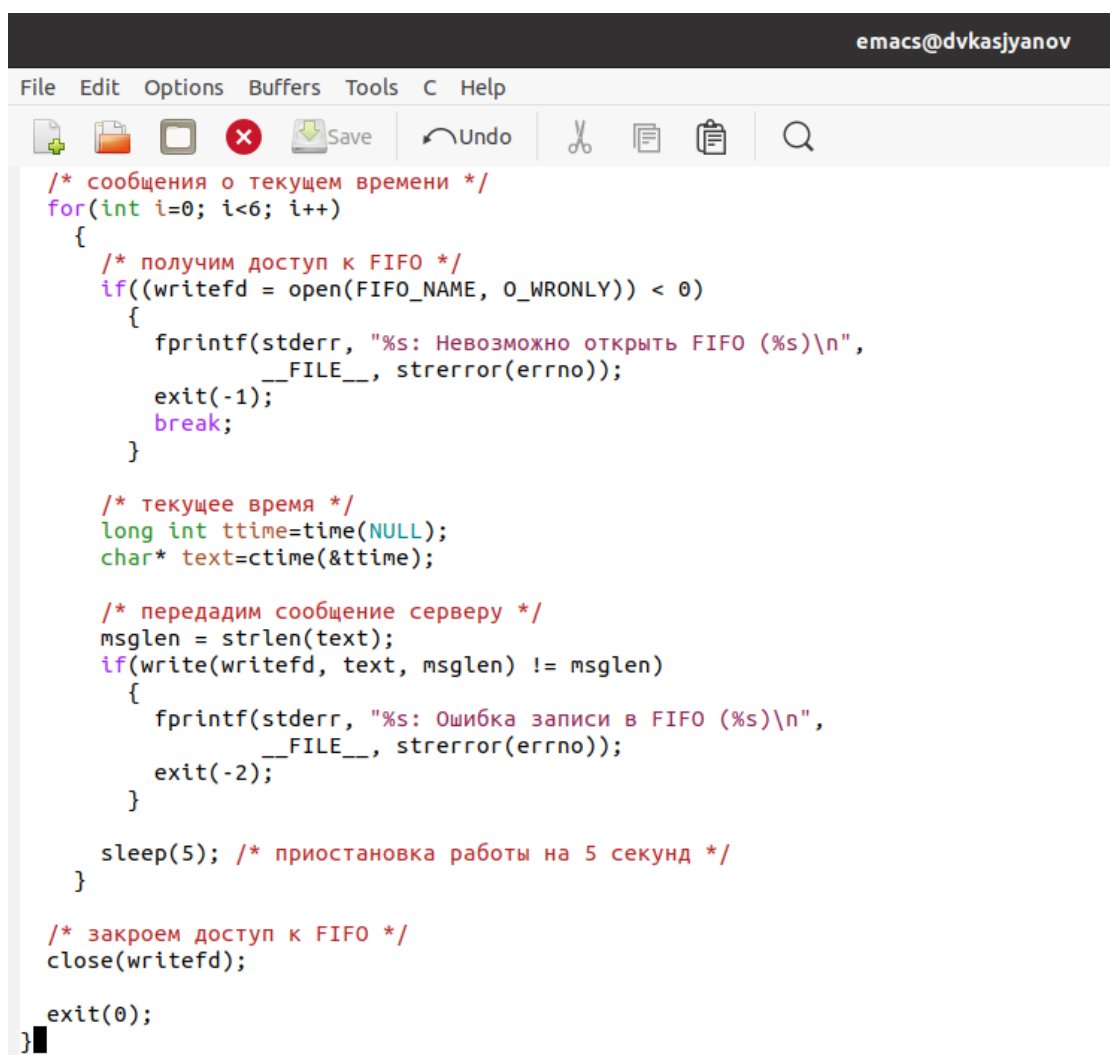
```
/*
 * client.c - реализация клиента
 *
 * чтобы запустить пример, необходимо:
 * 1. запустить программу server на одной консоли;
 * 2. запустить программу client на другой консоли.
 */

#include "common.h"

int
main()
{
    int writefd; /* дескриптор для записи в FIFO */
    int msglen;

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

(Рисунок 6)



The screenshot shows an Emacs editor window with the title bar 'emacs@dvkasjyanov'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'C', and 'Help'. The toolbar contains icons for file operations (new, open, save, close), editing (undo, redo, cut, copy, paste), and search. The main text area contains the following C code:

```
/* сообщения о текущем времени */
for(int i=0; i<6; i++)
{
    /* 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);
        break;
    }

    /* текущее время */
    long int ttime=time(NULL);
    char* text=ctime(&ttime);

    /* передадим сообщение серверу */
    msglen = strlen(text);
    if(write(writefd, text, msglen) != msglen)
    {
        fprintf(stderr, "%s: Ошибка записи в FIFO (%s)\n",
            __FILE__, strerror(errno));
        exit(-2);
    }

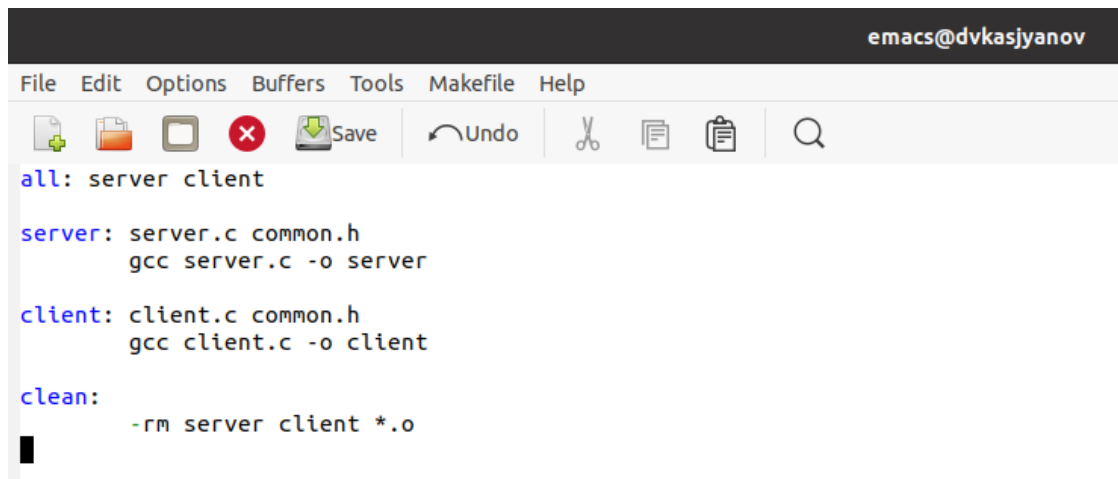
    sleep(5); /* приостановка работы на 5 секунд */
}

/* закроем доступ к FIFO */
close(writefd);

exit(0);
}
```

(Рисунок 7)

- **Makefile:** оставляю файл без изменений (Рисунок 8).



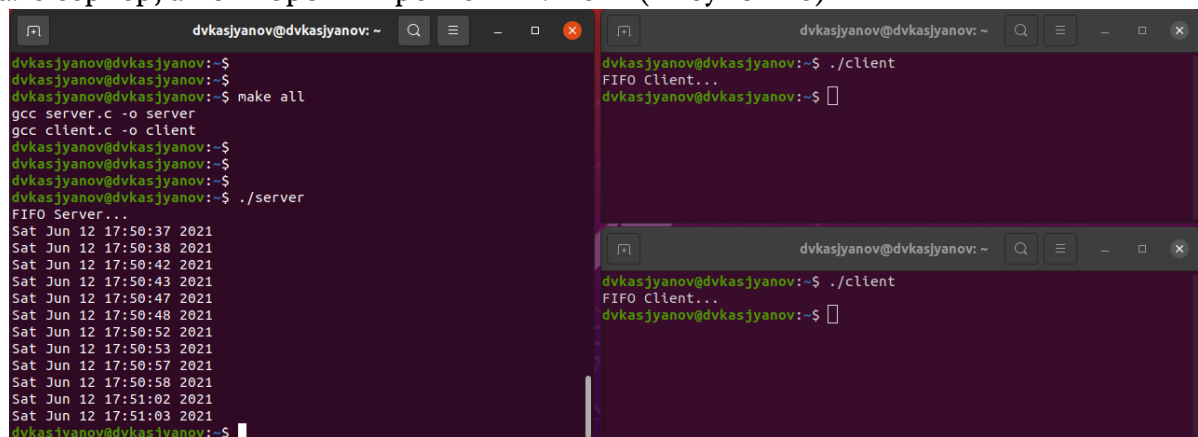
(Рисунок 8)

Используя команду `make all`, компилирую файлы (Рисунок 9).

```
dvkasjyanov@dvkasjyanov:~$ make all
gcc server.c -o server
gcc client.c -o client
dvkasjyanov@dvkasjyanov:~$
```

(Рисунок 9)

Открываю 3 терминала. Проверяю работу скрипта, открывая в первом терминале сервер, а во втором и третьем - клиент (Рисунок 10)

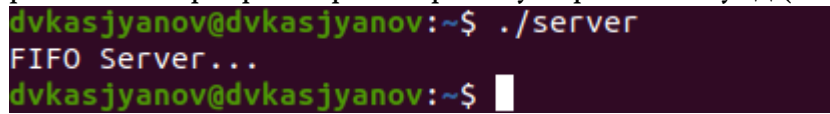


(Рисунок 10)

В результате терминалы-клиенты вывели по 6 сообщений, спустя 30 секунд работа сервера была прекращена.

Программа работает корректно.

Проверяю длительность работы сервера, введя команду `./server` в одном терминале. Сервер завершает работу через 30 секунд (Рисунок 11).



```
dvkasjyanov@dvkasjyanov:~$ ./server
FIFO Server...
dvkasjyanov@dvkasjyanov:~$
```

(Рисунок 11)

Если сервер завершит свою работу, не закрыв канал, то при очередном запуске сервер, появится ошибка, так как один канал уже существует.

4 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 **ИРС** 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 <имя_файла>`, либо команду `mkfifo <имя_файла>`.
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 (**A**) пытается записать **X** байтов данных в канал, в котором имеется место для **Y** байтов данных. Если **X** больше, чем **Y**, то тогда только первые **Y** байтов данных записываются в канал, и процесс блокируется. Запускается другой процесс (**B**); в это время в канале появляется свободное пространство (благодаря третьему процессу, считывающему данные из канала). Процесс **B** записывает данные в канал. Затем, когда выполнение процесса **A** 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-p** 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). Эти ошибки возникают при вызове функций стандартных библиотек **C**. 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 **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.

5 Выводы

Я приобрёл практические навыки работы с именованными каналами.

6 Библиография

Лабораторная работа №15 - “Именованные каналы”

Каналы (pipe, fifo)

Структура ядра и системные вызовы. Программирование