

# Отчет по лабораторной работе 14

## Приобретение практических навыков с именованными каналами

Межидов Хамзат Лечаевчи

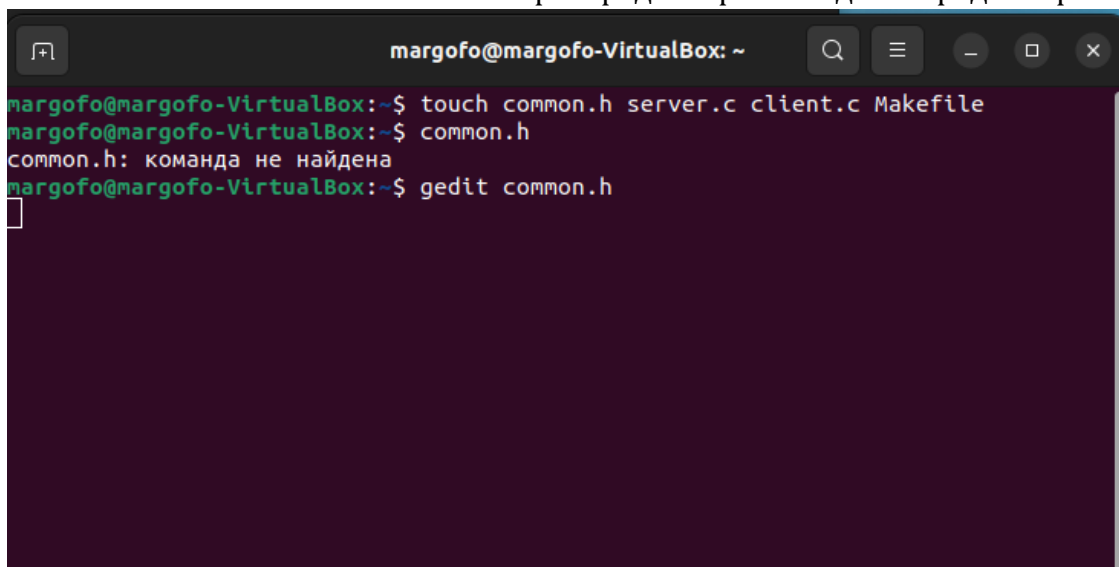
### Содержание

### Цель работы

- Приобретение практических навыков работы с именованными каналами.

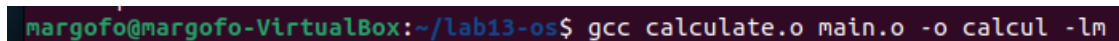
### Ход работы

- Для начала я создал необходимые файлы с помощью команды “touch common.h server.c client.c Makefile” и открыл редактор emacs для их редактирования.



```
margofo@margofo-VirtualBox: ~  
margofo@margofo-VirtualBox:~$ touch common.h server.c client.c Makefile  
margofo@margofo-VirtualBox:~$ common.h  
common.h: команда не найдена  
margofo@margofo-VirtualBox:~$ gedit common.h
```

1



```
margofo@margofo-VirtualBox:~/lab13-os$ gcc calculate.o main.o -o calcul -lm
```

2

- Далее я изменил коды программ, представленных в тексте лабораторной работы. В файл unistd.h и time.h, необходимые для работы кодов других файлов. Common.h Предназначен для заголовочных файлов, чтобы в остальных программах их не прописывать каждый раз.

```
margofo@margofo-VirtualBox: ~/lab14-os
margofo@margofo-VirtualBox:~$ mkdir lab14-os
margofo@margofo-VirtualBox:~$ cd lab14-os
margofo@margofo-VirtualBox:~/lab14-os$ touch common.h
margofo@margofo-VirtualBox:~/lab14-os$ gedit common.h
```

3

```
Открыть ▾ *common.h
~/lab14-os
1 #ifndef __COMMON_H__
2 #define __COMMON_H__
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <string.h>
7 #include <errno.h>
8 #include <sys/types.h>
9 #include <sys/stat.h>
10 #include <fcntl.h>
11 #include <unistd.h>
12 #include <time.h>
13
14 #define FIFO_NAME "/tmp/fifo"
15 #define MAX_BUFF 80
16
17 #endif
```

4

- В файл Server.c добавил цикл while для контроля за временем работы сервера. Разница между текущим временем time (NULL) и временем начала работы clock\_t start=time(NULL) не должна превышать 30 секунд

```
margofo@margofo-VirtualBox:~/lab14-os$ gedit server.c
```

5

```
server.c
~/lab14-os
Сохранить

2
3 int main() {
4     int readfd;
5     int n;
6     char buff[MAX_BUFF];
7
8     printf("FIFO Server...\n");
9
10    if(mknod(FIFO_NAME, S_IFIFO | 0666, 0) < 0)
11    {
12        fprintf(stderr, "%s: Невозможно создать FIFO (%s)\n", __FILE__, strerror(errno));
13        exit(-1);
14    }
15
16    if((readfd = open(FIFO_NAME, O_RDONLY)) < 0)
17    {
18        fprintf(stderr, "%s: Невозможно открыть FIFO (%s)\n", __FILE__, strerror(errno));
19        exit(-2);
20    }
21
22    clock_t start = time(NULL);
23
24    while(time(NULL)-start < 30)
25    {
26        while((n = read(readfd, buff, MAX_BUFF)) > 0)
27        {
28            if(write(1, buff, n) != n)
29            {
30                fprintf(stderr, "%s: Ошибка вывода (%s)\n", __FILE__, strerror(errno));
31                exit(-3);
32            }
33        }
34    }
35
36    close(readfd);
37
38    if(unlink(FIFO_NAME) < 0)
39    {
40        fprintf(stderr, "%s: Невозможно удалить FIFO (%s)\n", __FILE__, strerror(errno));
41        exit(-4);
42    }
43
44    exit(0);
45 }
```

6

- В файл client.c добавил цикл, который отвечает за количество сообщений о текущем времени, которое получается в результате выполнения команд и команду для остановки работы клиента на 5 секунд

```
margof@margof-VirtualBox:~/lab14-os$ gedit server.c
margof@margof-VirtualBox:~/lab14-os$ gedit client.c
```

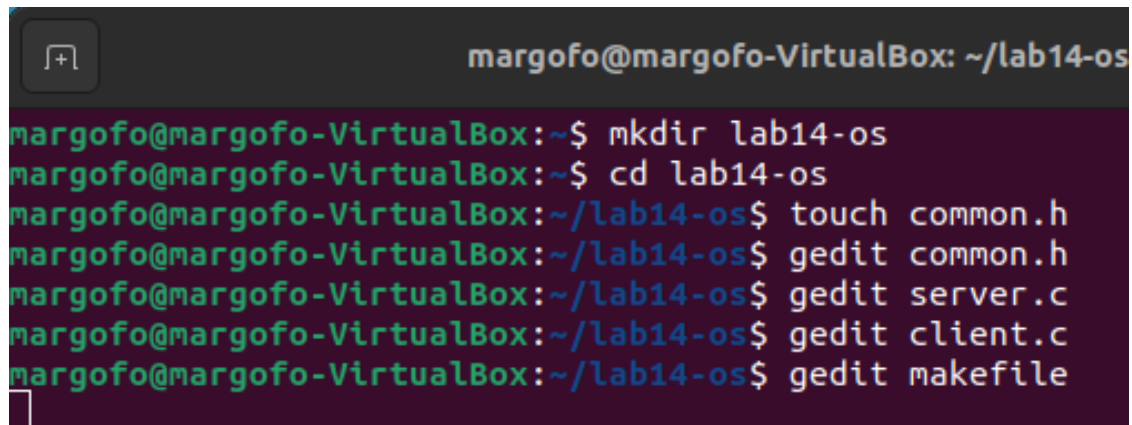
7

A screenshot of a code editor window titled 'client.c' with the path '~/.lab14-os'. The editor has a dark theme and includes buttons for 'Открыть' (Open), 'Сохранить' (Save), and a menu icon. The code is a C program for a FIFO client. It includes 'common.h', defines 'main()', and uses 'writefd' and 'msglen' variables. It prints 'FIFO Client...\n', loops 4 times, attempts to open a FIFO, writes a timestamp, and sleeps for 5 seconds. Error handling is present for both opening and writing to the FIFO.

```
1 #include "common.h"
2
3 int main() {
4     int writefd;
5     int msglen;
6
7     printf("FIFO Client...\n");
8
9     for(int i=0; i<4; i++)
10    {
11
12        if((writefd = open(FIFO_NAME, O_WRONLY)) < 0)
13        {
14            fprintf(stderr, "%s: Невозможно открыть FIFO (%s)\n", __FILE__, strerror(errno));
15            exit(-1);
16        }
17
18        long int ttime = time(NULL);
19        char* text = ctime(&ttime);
20
21        msglen = strlen(text);
22        if(write(writefd, text, msglen) != msglen)
23        {
24            fprintf(stderr, "%s: Ошибка записи в FIFO (%s)\n", __FILE__, strerror(errno));
25            exit(-2);
26        }
27
28        sleep (5);
29    }
30
31    close(writefd);
32
33    exit(0);
34 }
35
```

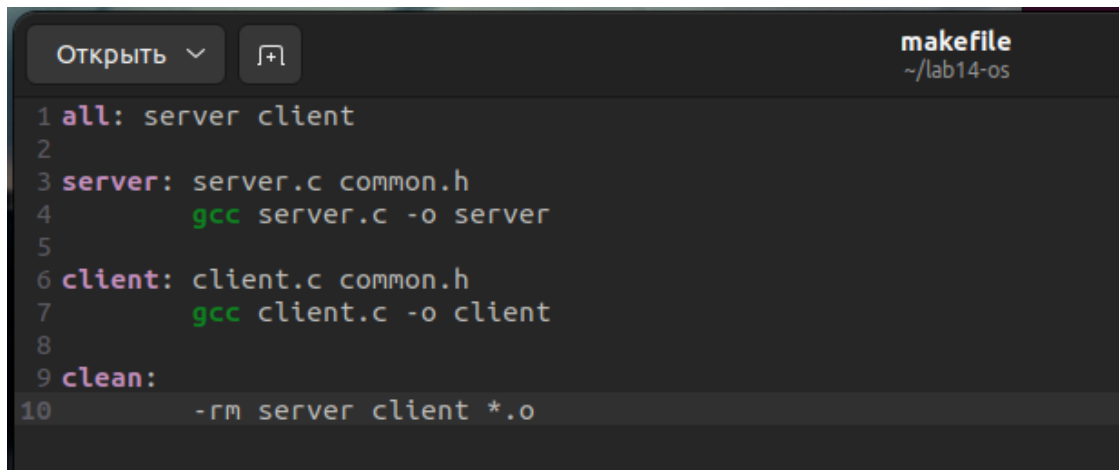
8

- Makefile не изменялся

A screenshot of a terminal window with the prompt 'margof@ margof-VirtualBox: ~/lab14-os'. The terminal shows a series of commands to create the directory structure and files for the lab: 'mkdir lab14-os', 'cd lab14-os', 'touch common.h', and four 'gedit' commands to create 'common.h', 'server.c', 'client.c', and 'makefile'.

```
margof@ margof-VirtualBox: ~/lab14-os
margof@ margof-VirtualBox:~$ mkdir lab14-os
margof@ margof-VirtualBox:~$ cd lab14-os
margof@ margof-VirtualBox:~/lab14-os$ touch common.h
margof@ margof-VirtualBox:~/lab14-os$ gedit common.h
margof@ margof-VirtualBox:~/lab14-os$ gedit server.c
margof@ margof-VirtualBox:~/lab14-os$ gedit client.c
margof@ margof-VirtualBox:~/lab14-os$ gedit makefile
```

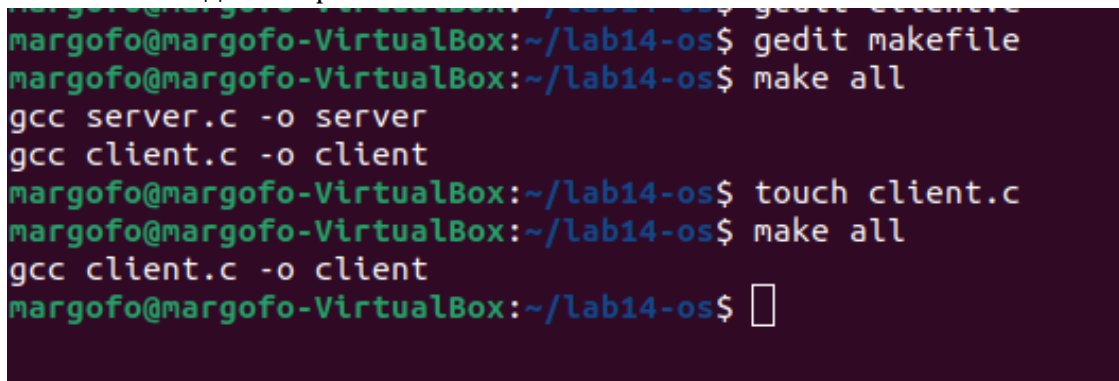
9

A screenshot of a text editor window titled "makefile" with the path "~/lab14-os". The editor shows a Makefile with the following content:

```
1 all: server client
2
3 server: server.c common.h
4     gcc server.c -o server
5
6 client: client.c common.h
7     gcc client.c -o client
8
9 clean:
10     -rm server client *.o
```

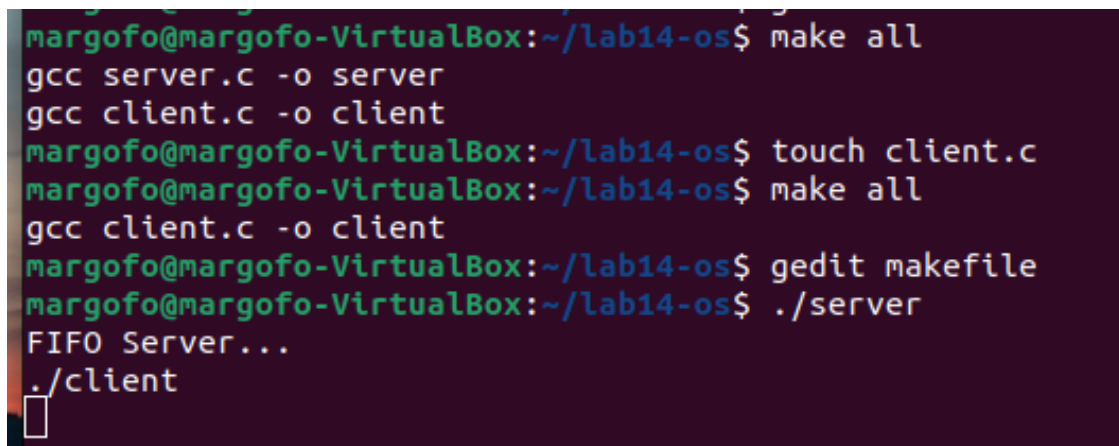
10

- После написания кодов я использую команду “make all”, скомпилировал необходимые файлы

A screenshot of a terminal window with the prompt "margofo@margofo-VirtualBox:~/lab14-os\$". The terminal shows the following commands and output:

```
margofo@margofo-VirtualBox:~/lab14-os$ gedit makefile
margofo@margofo-VirtualBox:~/lab14-os$ make all
gcc server.c -o server
gcc client.c -o client
margofo@margofo-VirtualBox:~/lab14-os$ touch client.c
margofo@margofo-VirtualBox:~/lab14-os$ make all
gcc client.c -o client
margofo@margofo-VirtualBox:~/lab14-os$
```

11

A screenshot of a terminal window with the prompt "margofo@margofo-VirtualBox:~/lab14-os\$". The terminal shows the following commands and output:

```
margofo@margofo-VirtualBox:~/lab14-os$ make all
gcc server.c -o server
gcc client.c -o client
margofo@margofo-VirtualBox:~/lab14-os$ touch client.c
margofo@margofo-VirtualBox:~/lab14-os$ make all
gcc client.c -o client
margofo@margofo-VirtualBox:~/lab14-os$ gedit makefile
margofo@margofo-VirtualBox:~/lab14-os$ ./server
FIFO Server...
margofo@margofo-VirtualBox:~/lab14-os$ ./client
```

12

## Вывод

- В ходе выполнения данной лабораторной работы я приобрел практические навыки работы с именнованными каналами

## Контрольные вопросы

- Именнованные каналы отличаются от неименованных наличием идентификатора канала, который представлен как специальный файл. Поскольку файл находится на локальной файловой системе, данное IPC используется внутри одной системы
- Чтоб создать неименованный канал из командной строки нужно использовать символ `|`, служащий для объединения двух и более процессов: `процесс_1 | процесс_2 | процесс_3...`
- Чтобы создать именнованный канал из командной строки нужно использовать либо команду `"mknod"`, либо команду `"mkfifo"`.
- Неименованный канал является средством взаимодействия между связанными процессами - родительским и дочерним. Родительский процесс создает канал при помощи системного вызова: `"int pipe"`; Массив из двух целых чисел является выходным параметром этого системного вызова. Если вызов выполняется 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, 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, 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
- Файлы именнованных каналов создаются функцией `mkfifo()` или функцией `mknod`: `"int mkfifo(const char pathname, mode_t mode);"` где первый параметр путь где будет располагаться FIFO, `"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()` создаёт 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

- При чтении меньшего числа байтов, чем находится в канале или FIFO, возвращается требуемое число байтов, остаток сохраняется до последующих чтений. При чтении большего числа байтов, чем находится в канале или FIFO, возвращается ядоступное число байтов. Процесс, читающий из канала, должен соответствующим образом обработать ситуацию, когда прочитано меньше, чем заказано.
- Запись числа байтов, меньшего ёмкости 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).
- Количество процессов, которые могут параллельно присоединяться к любому концу канала, не ограничено. Однако если два и более процесса записывают в канал данные одновременно, каждый процесс за один раз может записать максимум PIPE BUF байтов данных. Предположим, процесс (назовём его А) пытается записать X байтов данных в канал, в котором имеется место 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. 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 поочерёдно двумя процессами. Аналогичным образом, если два или более процессов одновременно попытаются прочитать данные из канала, может случиться так, что каждый из них прочитает только часть необходимых данных.
- 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. 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.

- 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 возникаюь 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 поймёт, как исправить ошибку, прочитав сообщение функции `strerror`. Возвращённый 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.