

Отчёт по лабораторной работе №14

Дисциплина: Операционные системы

Шмырин Михаил Сергеевич

Содержание

1	Цель работы	5
2	Выполнение лабораторной работы	6
3	Выводы	13
4	Ответы на контрольные вопросы	14

Список таблиц

Список иллюстраций

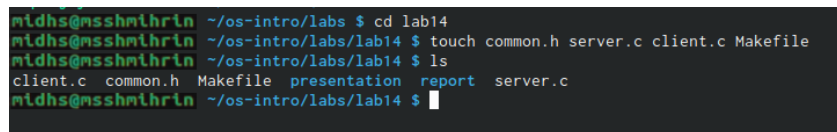
2.1	Создание файлов	6
2.2	common.h	7
2.3	server.c	8
2.4	server.c	9
2.5	client.c	10
2.6	client.c	10
2.7	Makefile	11
2.8	Компиляция файлов	11
2.9	Работа сервера	12
2.10	Работа сервера	12
2.11	Работа сервера	12

1 Цель работы

Цель данной лабораторной работы - приобрести практические навыки работы с именованными каналами.

2 Выполнение лабораторной работы

1. Для выполнения лабораторной работы создал четыре файл с помощью команды `touch` и откроем их в `emacs` для редактирования (рис. 2.1)

A terminal window with a dark background and green text. The prompt is 'midhs@msshm1hr1n'. The user enters 'cd lab14'. The prompt changes to 'midhs@msshm1hr1n ~/os-intro/labs/lab14'. The user enters 'touch common.h server.c client.c Makefile'. The prompt changes to 'midhs@msshm1hr1n ~/os-intro/labs/lab14'. The user enters 'ls'. The output is 'client.c common.h Makefile presentation report server.c'. The prompt changes to 'midhs@msshm1hr1n ~/os-intro/labs/lab14' with a cursor at the end.

```
midhs@msshm1hr1n ~/os-intro/labs $ cd lab14
midhs@msshm1hr1n ~/os-intro/labs/lab14 $ touch common.h server.c client.c Makefile
midhs@msshm1hr1n ~/os-intro/labs/lab14 $ ls
client.c common.h Makefile presentation report server.c
midhs@msshm1hr1n ~/os-intro/labs/lab14 $
```

Рис. 2.1: Создание файлов

2. Изменим код программ, предоставленных в тексте задания лабораторной работы. В файл `common.h` добавил стандартные заголовочные файлы `unistd.h` и `time.h`, которые необходимы для работы других файлов (рис. 2.2). Этот файл предназначен для заголовочных файлов, чтобы не прописывать их в других программах каждый раз

```

/*
 * 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. 2.2: common.h

Затем в файл server.c добавим цикл while для контроля за временем работы сервера (рис. 2.3, 2.4), причем время от начал работы сервера до настоящего не должно превышать 30 секунд

```

1 /*
2 * server.c - реализация сервера
3 *
4 * чтобы запустить пример, необходимо:
5 * 1. запустить программу server на одной консоли;
6 * 2. запустить программу client на другой консоли.
7 */
8
9 #include "common.h"
10
11 int
12 main()
13 {
14     int readfd; /* дескриптор для чтения из FIFO */
15     int n;
16     char buff[MAX_BUFF]; /* буфер для чтения данных из FIFO */
17
18     /* баннер */
19     printf("FIFO Server...\n");
20
21     /* создаем файл FIFO с открытыми для всех
22      * правами доступа на чтение и запись
23      */
24     if(mknod(FIFO_NAME, S_IFIFO | 0666, 0) < 0)
25     {
26         fprintf(stderr, "%s: Невозможно создать FIFO (%s)\n",
27             __FILE__, strerror(errno));
28         exit(-1);
29     }
30 }

```

Рис. 2.3: server.c


```

/* 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);
}

clock_t start = time(NULL);

/* читаем данные из FIFO и выводим на экран */
while(time(NULL)-start < 30)
{
    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);
        }
    }
}
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. 2.4: server.c

В файл client.c добавим цикл for который будет отвечать за количество сообщений о текущем времени (4 сообщения), и команду sleep(5) для остановки работы клиента через 5 секунд. Также я изменил выводимое сообщение на текущее время (рис. 2.5, 2.6)

```

/*
 * 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");
    for (int i=0; i<4; i++)
    {
        if((writefd = open(FIFO_NAME, O_WRONLY)) < 0)
        {
            fprintf(stderr, "%s: Невозможно открыть FIFO (%s)\n",
                __FILE__, strerror(errno));
            exit(-1);
        }
        long int Time = time(NULL);
        char* text = ctime(&Time);

```

Рис. 2.5: client.c

```

/* 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(-2);
}
sleep (5);
}

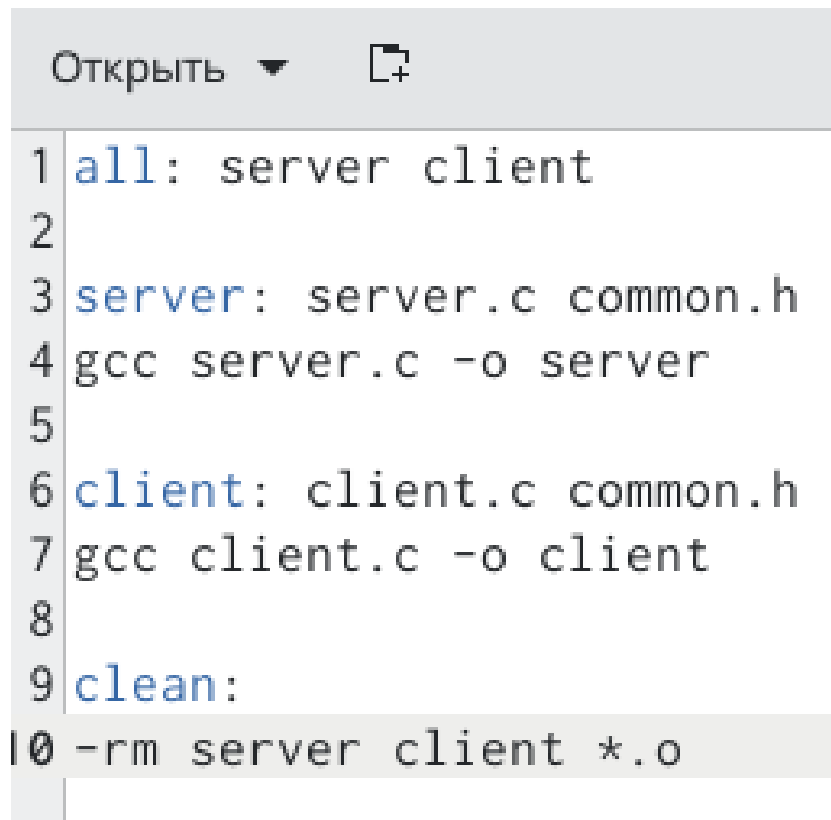
/* закроем доступ к FIFO */
close(writefd);

exit(0);
}

```

Рис. 2.6: client.c

Makefile оставил без изменений (рис. 2.7)

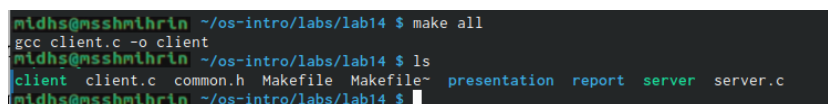


The image shows a text editor window with a menu bar containing 'Открыть' (Open) and a file icon. The editor displays the following Makefile content:

```
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
```

Рис. 2.7: Makefile

3. Используя команду `make all` (рис. 2.8), скомпилировал необходимые для работы файлы



The image shows a terminal window with the following commands and output:

```
midhs@msshmihrin ~/os-intro/labs/lab14 $ make all
gcc client.c -o client
midhs@msshmihrin ~/os-intro/labs/lab14 $ ls
client  client.c  common.h  Makefile  Makefile~  presentation  report  server  server.c
midhs@msshmihrin ~/os-intro/labs/lab14 $
```

Рис. 2.8: Компиляция файлов

4. Проверим работу написанного кода

Открыл три терминала, в первом окне запустил программу `./server`, во втором и третьем `./client`. В результате каждый терминал-клиент вывел по четыре сообщения о текущем времени. Спустя 30 секунд работы сервера был прекращена (рис. 2.9, 2.10, 2.11)

```

midhs@msshmihrin ~/os-intro/labs/lab14 $ cd ~/os-intro/labs/lab14
midhs@msshmihrin ~/os-intro/labs/lab14 $ ./server
FIFO Server...
Hello Server!!!
Hello Server!!!
Hello Server!!!
Hello Server!!!
Hello Server!!!
Hello Server!!!
Hello Server!!!
Hello Server!!!
midhs@msshmihrin ~/os-intro/labs/lab14 $

```

Рис. 2.9: Работа сервера

```

midhs@msshmihrin ~/os-intro/labs/lab14 $ cd ~/os-intro/labs/lab14
midhs@msshmihrin ~/os-intro/labs/lab14 $ ./client
FIFO Client...
midhs@msshmihrin ~/os-intro/labs/lab14 $

```

Рис. 2.10: Работа сервера

```

midhs@msshmihrin ~/os-intro/labs/lab14 $ cd ~/os-intro/labs/lab14
midhs@msshmihrin ~/os-intro/labs/lab14 $ ./client
FIFO Client...
midhs@msshmihrin ~/os-intro/labs/lab14 $

```

Рис. 2.11: Работа сервера

Если клиент завершит свою работу, не закрыв канал, то при повторном запуске сервера появится ошибка “Невозможно закрыть FIFO”, так как уже существует один канал.

3 Выводы

Я приобрел практические навыки по работе с именованными каналами.

4 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 канал 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`: 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 канал уже существует, `mkfifo()` возвращает -1. После создания файл канал 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 канал 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 числ `count` (например, когда размер для за-

писи 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.