

Лабораторная работа №14

Операционные системы

Пашаев Юсиф Юнусович

Содержание

1	Цель работы	5
2	Выполнение лабораторной работы	6
3	Выводы	12
4	Контрольные вопросы	13

Список иллюстраций

2.1	Создаем файлы	6
2.2	Программы	7
2.3	Программы	8
2.4	Программы	9
2.5	Программы	10
2.6	Проверка написаного кода	11
2.7	Проверка написаного кода	11

Список таблиц

1 Цель работы

Приобретение практических навыков работы с именованными каналами

2 Выполнение лабораторной работы

1. Создаем необходимые нам файлы (рис. 2.1).

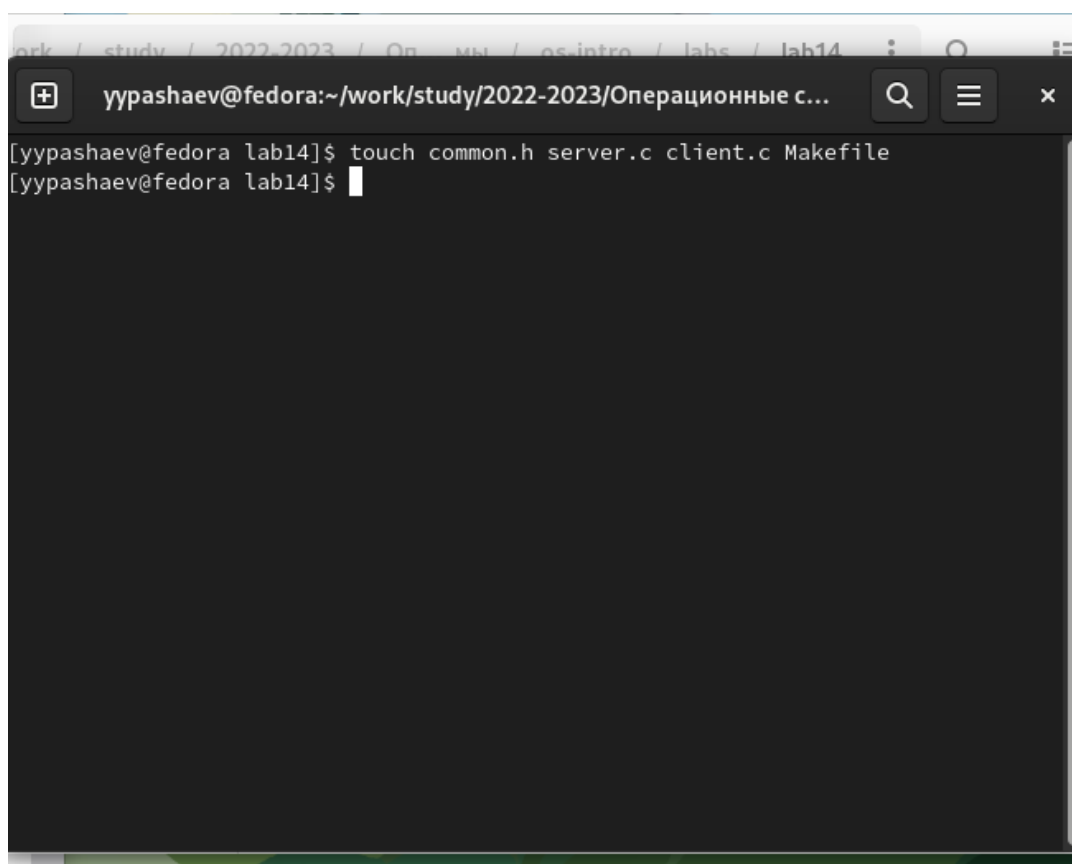


Рис. 2.1: Создаем файлы

2. Прописываем в созданных файлах программы



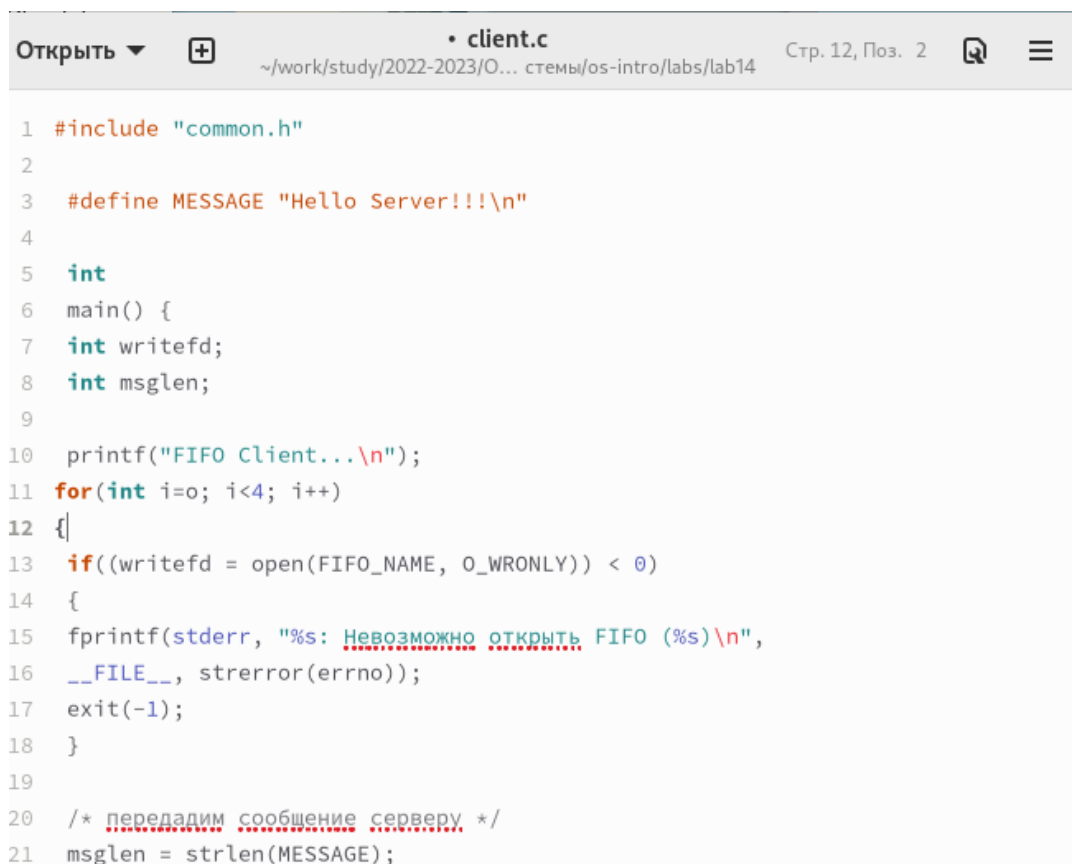
```
1  #ifndef __COMMON_H__
2  #define __COMMON_H__
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <errno.h>
8  #include <sys/types.h>
9  #include <sys/stat.h>
10 #include <fcntl.h>
11 #include <unistd.h>
12 #include <time.h>
13
14 #define FIFO_NAME "/tmp/fifo"
15 #define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

Рис. 2.2: Программы



```
12  /* создаем файл FIFO с открытием для всех
13  * добавим дескриптор на чтение и запись
14  */
15  if(mknod(FIFO_NAME, S_IFIFO | 0666, 0) < 0)
16  {
17      fprintf(stderr, "%s: Невозможно создать FIFO (%s)\n",
18              __FILE__, strerror(errno));
19      exit(-1);
20  }
21
22  /* открываем FIFO на чтение */
23  if((readfd = open(FIFO_NAME, O_RDONLY)) < 0)
24  {
25      fprintf(stderr, "%s: Невозможно открыть FIFO (%s)\n",
26              __FILE__, strerror(errno));
27      exit(-2);
28  }
29
30  while(time(NULL)-start < 30)
31  {
32      while((n = read(readfd, buff, MAX_BUFF)) > 0)
33      {
34          if(write(1, buff, n) != n)
35          {
36              fprintf(stderr, "%s: Ошибка вывода (%s)\n",
37                      __FILE__, strerror(errno));
38              exit(-3);
39          }
40      }
```

Рис. 2.3: Программы



The image shows a code editor window with a title bar containing a plus icon, the filename "client.c", and the path "~/work/study/2022-2023/O... стемы/os-intro/labs/lab14". The editor displays the source code for "client.c" with line numbers 1 through 21. The code includes a header file, defines a message, and implements a main function that attempts to open a FIFO and send a message to a server.

```
1 #include "common.h"
2
3 #define MESSAGE "Hello Server!!!\n"
4
5 int
6 main() {
7     int writefd;
8     int msglen;
9
10    printf("FIFO Client...\n");
11    for(int i=0; i<4; i++)
12    {
13        if((writefd = open(FIFO_NAME, O_WRONLY)) < 0)
14        {
15            fprintf(stderr, "%s: Невозможно открыть FIFO (%s)\n",
16                __FILE__, strerror(errno));
17            exit(-1);
18        }
19
20        /* передадим сообщение серверу */
21        msglen = strlen(MESSAGE);
```

Рис. 2.4: Программы

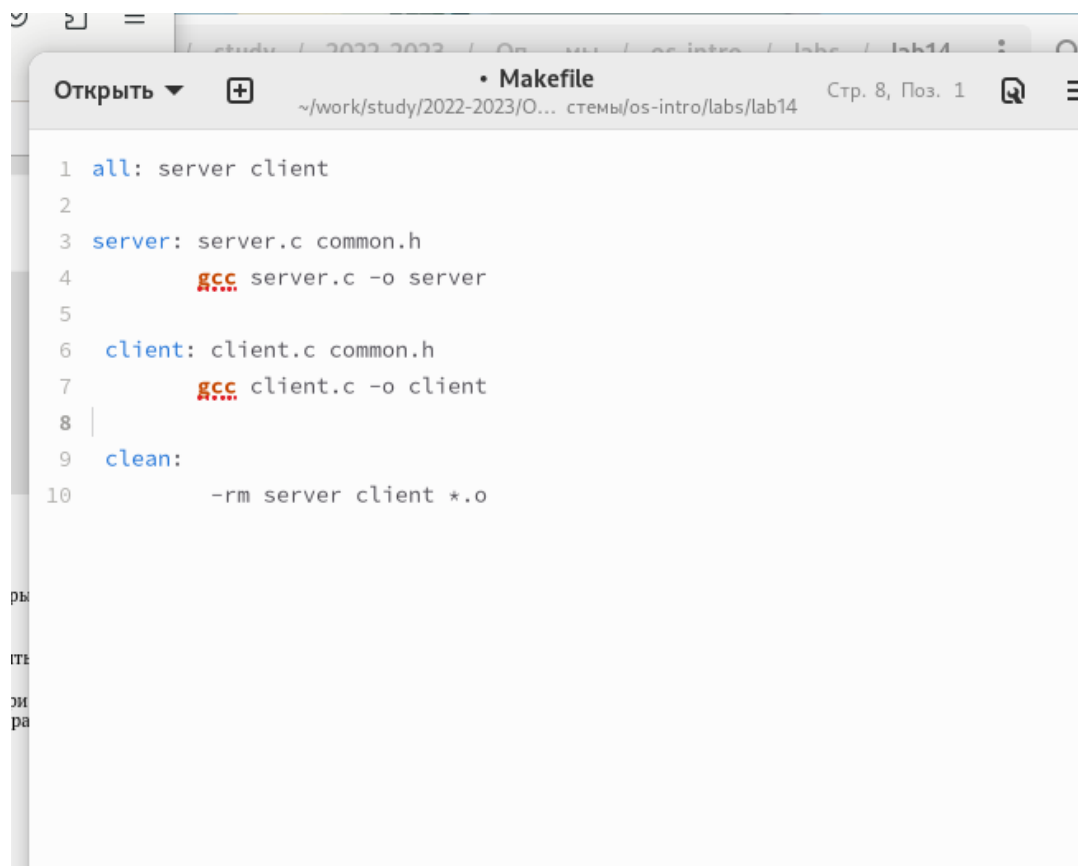
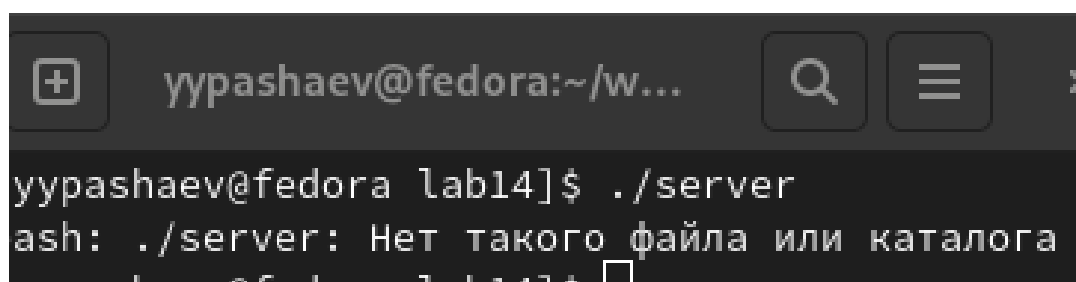


Рис. 2.5: Программы

3. После написания кодов, я, используя команду «make all», скомпилировала необходимые файлы. Далее я проверила работу написанного кода. Открыла 3 консоли (терминала) и запустила: в первом терминале – «./server», в остальных двух – «./client». В результате каждый терминал-клиент вывел по 4 сообщения. Спустя 30 секунд работа сервера была прекращена.



A terminal window with a dark background. The title bar shows a plus icon, the text 'yypashaev@fedora:~/w...', a search icon, a menu icon, and a close icon. The terminal content shows the prompt 'yypashaev@fedora lab14]\$' followed by the command './server'. The next line shows the error message 'bash: ./server: Нет такого файла или каталога'.

```
yypashaev@fedora lab14]$ ./server
bash: ./server: Нет такого файла или каталога
```

Рис. 2.6: Проверка написанного кода



A terminal window with a dark background. The title bar shows a plus icon, the text 'yypashaev@fedora:~/...', a search icon, a menu icon, and a close icon. The terminal content shows the prompt '[yypashaev@fedora lab14]\$' followed by the command './client'. The next line shows the error message 'bash: ./client: Нет такого файла или каталога'.

```
[yypashaev@fedora lab14]$ ./client
bash: ./client: Нет такого файла или каталога
```

Рис. 2.7: Проверка написанного кода

3 Выводы

Я приобрел практические навыки работы с именованными каналами

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 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 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 обрабатывать 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 мак-

симум 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 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.

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 # Список литературы{.unnumbered}