

Отчёт по лабораторной работе №14

Именованные каналы

Дарья Сергеевна Кочина

Содержание

1	Цель работы	4
2	Задание	5
3	Теоретическое введение	6
4	Выполнение лабораторной работы	7
5	Выводы	19

Список иллюстраций

4.1	Создание файлов	7
4.2	Файл common.h	8
4.3	Файл server.c	9
4.4	Файл server.c	10
4.5	Файл client.c	11
4.6	Файл client.c	12
4.7	Файл Makefile	13
4.8	Компиляция	13
4.9	Проверка	14
4.10	Проверка	14

1 Цель работы

Целью данной лабораторной работы является приобретение практических навыков работы с именованными каналами.

2 Задание

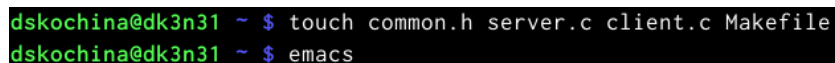
Приобрести практические навыки работы с именованными каналами.

3 Теоретическое введение

Одним из видов взаимодействия между процессами в операционных системах является обмен сообщениями. Под сообщением понимается последовательность байтов, передаваемая от одного процесса другому. В операционных системах типа UNIX есть 3 вида межпроцессорных взаимодействий: общепонимание (именованные каналы, сигналы), System V Interface Definition (SVID — разделяемая память, очередь сообщений, семафоры) и BSD (сокеты). Для передачи данных между неродственными процессами можно использовать механизм именованных каналов (named pipes). Данные передаются по принципу FIFO (First In First Out) (первым записан — первым прочитан), поэтому они называются также FIFO pipes или просто FIFO. 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.

4 Выполнение лабораторной работы

1. Для начала изучили материал лабораторной работы. Далее на основе примеров напишем аналогичные программы, но с изменениями.
2. Создадим необходимые файлы для работы. (рис. [4.1])

A terminal window with a black background and green text. The prompt is 'dskochina@dk3n31 ~ \$'. The first command is 'touch common.h server.c client.c Makefile' and the second is 'emacs'.

```
dskochina@dk3n31 ~ $ touch common.h server.c client.c Makefile
dskochina@dk3n31 ~ $ emacs
```

Рис. 4.1: Создание файлов

3. Затем изменим коды программ, данных в лабораторной работе. В файл `common.h` добавим стандартные заголовочные файлы: “`unistd.h`”, “`time.h`”. Это необходимо для работы других файлов. Этот файл является заголовочным, чтобы в остальных программах не прописывать одно и то же каждый раз. (рис. [4.2])

```

/*
 * 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. 4.2: Файл common.h

4. Затем в файл server.c добавляем цикл “while” для контроля за временем работы сервера. Разница между текущим временем и началом работы не должна превышать 30 секунд. (рис. [4.3], [4.4])


```

/*
 * 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);
    }

    /* 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);
    }
}

```

Рис. 4.3: Файл 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);
}
/* начало отсчёта времени */
clock_t start = time(NULL);

/* цикл работает пока с момента начала отсчёта времени прошло меньше 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);
        }
    }
    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. 4.4: Файл server.c

5. В файл client.c добавим цикл, который отвечает за количество сообщений о текущем времени (4 сообщения). С помощью команды “sleep” приостановим работу клиента на 5 секунд. (рис. [4.5], [4.6])

```

/*
 * 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");

    /* цикл отвечающий за отправку сообщения о текущем времени */
    for(int i=0; i<4; 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);

```

Рис. 4.5: Файл client.c

```

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

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

/* закроем доступ к FIFO */
close(writefd);
exit(0);
}

```

Рис. 4.6: Файл client.c

6. Makefile оставили без изменений. (рис. [4.7])

```
all: server client

server: server.c common.h
    gcc server.c -o server

client: client.c common.h
    gcc client.c -o client

clean:
    rm server client *.o
```

Рис. 4.7: Файл Makefile

7. Далее делаем компиляцию файлов с помощью команды “make all”. (рис. [4.8])

```
dskochina@dk3n31 ~ $ make all
gcc client.c -o client
dskochina@dk3n31 ~ $
```

Рис. 4.8: Компиляция

8. Затем открываем три терминала для проверки работы наших файлов. В первом пишем “./server”, а в остальных “./client”. В результате каждый терминал вывел по 4 сообщения, а по истечению 30 секунд работа сервера была завершена. Всё работает верно. (рис. [4.9])

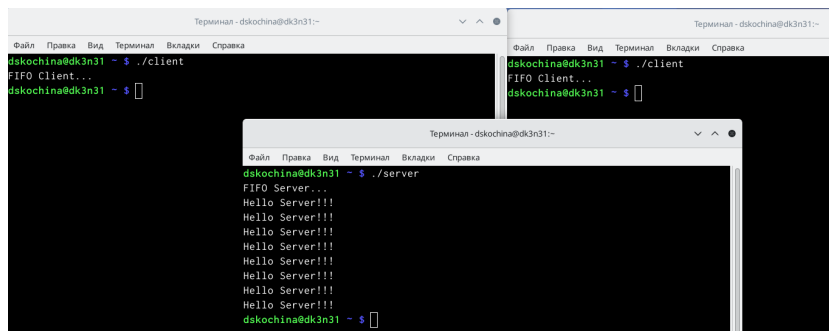


Рис. 4.9: Проверка

9. Проверим длительность работы сервера. Вводим команду “./server” в одном терминале. Он завершил свою работу через 30 секунд. Если сервер завершит свою работу, не закрыва канал, то при повторном запуске появится ошибка “Невозможно создать FIFO”, так как у нас уже есть один канал. (рис. [4.10])

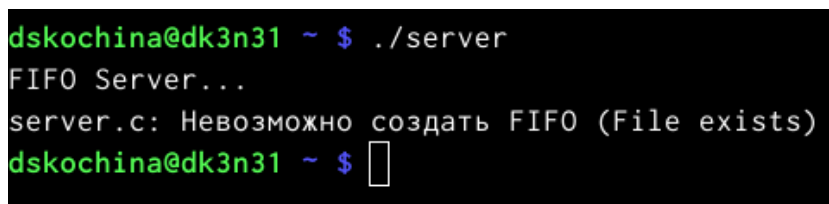


Рис. 4.10: Проверка

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 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 (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 свобод-

ное 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.

5 Выводы

В ходе выполнения данной лабораторной работы я приобрела практические навыки работы с именованными каналами.