

## ЛАБОРАТОРНАЯ РАБОТА №6. LINUX API – ВВЕДЕНИЕ В МЕЖПРОЦЕССНОЕ ВЗАИМОДЕЙСТВИЕ. КАНАЛЫ.

Наличие в Unix-системах простых и эффективных средств взаимодействия между процессами оказало на программирование в Unix не менее важное влияние, чем представление объектов системы в виде файлов. Благодаря межпроцессному взаимодействию (**Inter-Process Communication, IPC**) разработчик (и пользователь) может разбить решение сложной задачи на несколько простых операций, каждая из которых доверяется отдельной небольшой программе.

Последовательная обработка одной задачи несколькими простыми программами очень похожа на конвейерное производство (среди многих значений английского **pipeline** есть и «конвейер», но в этой работе для перевода слова **pipe** будет использоваться принятое в отечественной литературе термином «канал».



Альтернативой конвейерному подходу являются большие монолитные пакеты, построенные по принципу «все в одном». Использование набора простых утилит для решения одной сложной задачи требует несколько большей квалификации со стороны пользователя, но взамен предоставляет гибкость, не достижимую при использовании монолитных «монстров». Наборы утилит, использующих открытые протоколы IPC, легко наращивать и модифицировать. Разбиение сложных задач на сравнительно небольшие подзадачи также позволяет снизить количество ошибок, допускаемых программистами. Помимо всего этого у **IPC** есть еще одно важное преимущество. Программы, использующие

**IPC**, могут «общаться» друг с другом практически также эффективно, как и с пользователем, в результате чего появляется возможность автоматизировать выполнение сложных задач. Могущество скриптовых языков Unix и Linux во многом основано на возможностях IPC.

## 6.1 Неименованные каналы

Чаще всего внутри-программные каналы используются тогда, когда программа запускает другую программу и считывает данные, которые та выводит в свой стандартный поток вывода. С помощью этого трюка разработчик может использовать в своей программе функциональность другой программы, не вмешиваясь во внутренние детали ее работы. Для решения этой задачи используются функции **popen (3)** и **pclose (3)**. Формально эти функции подобны функциям **fopen (3)** и **fclose(3)**.

### Синтаксис

```
#include <stdio.h>

FILE *popen(const char *command, const char *type);

int pclose(FILE *stream);
```

### Описание

Функция **popen()** открывает процесс, создавая канал, производя **fork** и вызывая командную оболочку. Так как канал задается однонаправленным, аргумент **type** может указать только на режим чтения или записи, но не на оба одновременно.

Аргумент **command** - это указатель на C-строку, содержащую командную строку для оболочки. Эта команда передается **/bin/sh** с помощью флага **-c** . Интерпретация, если она необходима, выполняется самой оболочкой. Аргумент **type** - это указатель на C-строку, содержащую символ **`r'** для чтения или **`w'** для записи.

Возвращаемое значение **popen()** - это обычный поток ввода-вывода (за исключением того, что он должен быть закрыт только функцией **pclose()** , а не **fclose()**). Запись в канал передается на стандартный поток ввода команды, стандартный поток вывода команды передается в канал, кроме случаев, когда потоки вывода-вывода переопределены самой командой.

Заметьте, что выходной поток, возвращаемый **popen**, по умолчанию полностью буферизирован.

Функция **pclose** ожидает завершения ассоциированного процесса и возвращает код выхода так же, как и функция **wait4**.

### Возвращаемые значения

Функция **popen** возвращает **NULL**, если вызовы **fork(2)** или **pipe(2)** завершились ошибкой или если невозможно выделить необходимый для этого объем памяти.

Функция **pclose** возвращает **-1**, если **wait4** возвращает ошибку или если была обнаружена какая-либо другая ошибка.

Функция **popen()** запускает внешнюю программу и возвращает вызвавшему ее приложению указатель на структуру **FILE**, связанную либо со стандартным потоком ввода, либо со стандартным потоком вывода запущенного процесса. Функция **pclose()** завершает работу с внешним приложением и закрывает канал. Для демонстрации работы с функциями **popen()/pclose()** мы напишем небольшую программу writelog.cpp

### Работа с функциями popen()/pclose()

---

```
/*
   Popen/Pclose Demo .
*/

#include <stdio.h>
#include <errno.h>

#define BUF_SIZE 0x100

int main(int argc, char * argv[])
{
    FILE * f;
    FILE * o;
    int len;
    char buf[BUF_SIZE];
    if (argc != 2)
    {
        printf("использование: makelog \"<command>\"\n");
        return -1;
    }
    f = popen(argv[1], "r");
    if (f == NULL)
    {
        perror("ошибка:\n");
```

```

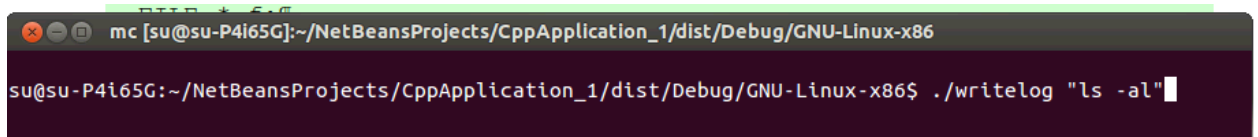
    return -1;
}
o = fopen("log.txt", "w");
while ((len = fread(buf, 1, BUF_SIZE, f)) != 0)
{
    write(1, buf, len);
    fwrite(buf, 1, len, o);
}
pclose(f);
fclose(o);
return 0;
}

```

Программа writelog.cpp выполняет команду оболочки, переданную ей в качестве параметра, и записывает данные, выводимые этой командой, одновременно на стандартный терминал и в файл log.txt (аналогичными функциями обладает стандартная команда **tee**).

На экране терминала будут распечатаны данные, выводимые командой оболочки **ls -al**, а в рабочей директории программы writelog будет создан файл log.txt, содержащий те же данные. Кавычки вокруг команды оболочки нужны для того, чтобы программа writelog получала строку вызова команды как один параметр командной строки.

Запустите программу, как показано на рисунке ниже.

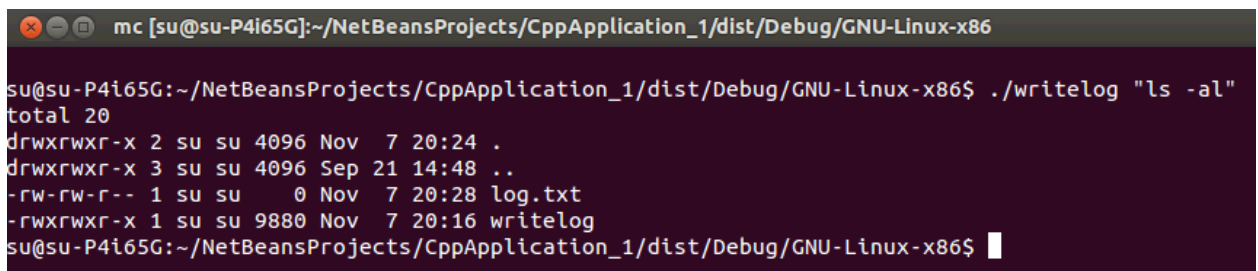


```

mc [su@su-P4i65G]:~/NetBeansProjects/CppApplication_1/dist/Debug/GNU-Linux-x86
su@su-P4i65G:~/NetBeansProjects/CppApplication_1/dist/Debug/GNU-Linux-x86$ ./writelog "ls -al"

```

Результат работы программы будет выглядеть как на рисунке ниже, аналогичные данные будут сохранены в файл log.txt



```

mc [su@su-P4i65G]:~/NetBeansProjects/CppApplication_1/dist/Debug/GNU-Linux-x86
su@su-P4i65G:~/NetBeansProjects/CppApplication_1/dist/Debug/GNU-Linux-x86$ ./writelog "ls -al"
total 20
drwxrwxr-x 2 su su 4096 Nov  7 20:24 .
drwxrwxr-x 3 su su 4096 Sep 21 14:48 ..
-rw-rw-r-- 1 su su   0 Nov  7 20:28 log.txt
-rwxrwxr-x 1 su su 9880 Nov  7 20:16 writelog
su@su-P4i65G:~/NetBeansProjects/CppApplication_1/dist/Debug/GNU-Linux-x86$

```

Рассмотрим фрагмент исходного текста программы:

```
f = popen(argv[1], "r");
```

Эта операция очень похожа на открытие обычного файла для чтения. Переменная **f** имеет тип **FILE \***, но в параметре **argv[1]** функции **popen()** передается не имя файла, а команда на запуск программы или команда оболочки, например, **"ls -al"**. Если вызов **popen()** был успешен, мы можем считывать данные, выводимые запущенной командой, с помощью обычной функции **fread(3)**.

## Синтаксис

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE
*stream);
```

## Описание

Функция **fread** считывает элементы данных **nmemb** (с размером каждого **size** байтов) с потока, на который указывает **stream**, и сохраняет их в позиции, на которую указывает **ptr**.

Функция **fwrite** записывает элементы данных **nmemb** (с размером каждого **size** байтов) в поток, на который указывает **stream**, при получении элементов с той позиции, на которую указывает **ptr**.

## Возвращаемые значения

**fread** и **fwrite** возвращают количество элементов, успешно считанных или записанных (то есть не количество символов). В случае ошибки или по достижении конца файла возвращаемое значение станет равным **"short item count"** (или нулю).

**fread** не определяет, действительно ли произошла ошибка или достигнут конец файла; для точного определения необходимо вызывать функции **feof(3)** и **ferror(3)**.

```
fread(buf, 1, BUF_SIZE, f)
```

Особенность функции **popen()** заключается в том, что эта функция не возвращает **NULL**, даже если переданная ей команда не является корректной.

Самый простой способ обнаружить ошибку в этой ситуации - попытаться прочесть данные из потока вывода. Если в потоке вывода нет данных (**fread()** возвращает значение

0), значит произошла ошибка. Для вывода данных, прочитанных помощью **fread()**, на терминал мы используем функцию **write()** с указанием дескриптора стандартного потока вывода:

```
write(1, buf, len);
```

Параллельно эти же данные записываются в файл на диске. По окончании чтения данных открытый канал нужно закрыть:

```
pclose(f);
```

Следует иметь в виду, что **pclose()** вернет управление вызывающему потоку только после того как запущенное с помощью **popen()** приложение завершит свою работу.

В заключение отметим еще одну особенность функции **popen()**. Для выполнения переданной ей команды **popen()** сперва запускает собственный экземпляр оболочки, что с одной стороны хорошо, а с другой - не очень. Хорошо это потому, что при вызове **popen()** автоматически выполняются внутренние операции оболочки (такие как обработка шаблонов имен файлов), используются переменные окружения типа **PATH** и **HOME** и т.п. Отрицательная сторона подхода, применяемого **popen()**, связана с дополнительными накладными расходами на запуск процесса оболочки, которые оказываются лишними в том случае, когда для выполнения внешней программы сама оболочка не нужна.

Для обмена данными с внешним приложением функция **popen()** использует каналы неявным образом. Наиболее распространенный тип каналов, - неименованные однонаправленные каналы (**anonymous pipes**), создаваемые функцией **pipe(2)**. На уровне интерфейса программирования такой канал представляется двумя дескрипторами файлов, один из которых служит для чтения данных, а другой – для записи. Каналы не поддерживают произвольный доступ, т. е. данные могут считываться только в том же порядке, в котором они записывались.

## Синтаксис

```
#include <unistd.h>
int pipe(int filedes[2]);
```

## Описание

**pipe** создает пару файловых дескрипторов, указывающих на запись **inode** именowanego канала, и помещает их в массив, на который указывает **filedes**. **filedes[0]** предназначен для чтения, а **filedes[1]** предназначен для записи.

## Возвращаемые значения

При удачном завершении вызова возвращаемое значение равно нулю. При ошибке возвращается **-1**, а переменной **errno** присваивается номер ошибки.

Неименованные каналы используются преимущественно вместе с функцией **fork(2)** и служат для обмена данными между родительским и дочерним процессами. Для организации подобного обмена данными, сначала, с помощью функции **pipe()**, создается канал. Функции **pipe()** передается единственный параметр – массив типа **int**, состоящий из двух элементов. В первом элементе массива функция возвращает дескриптор файла, служащий для чтения данных из канала (выход канала), а во втором - дескриптор для записи (вход). Затем, с помощью функции **fork()** процесс «раздваивается». Дочерний процесс наследует от родительского процесса оба дескриптора, открытых с помощью **pipe()**, но, также как и родительский процесс, он должен использовать только один из дескрипторов.

Направление передачи данных между родительским и дочерним процессом определяется тем, какой дескриптор будет использоваться родительским процессом, а какой - дочерним. Продемонстрируем изложенное на простом примере программы **pipes.cpp**, использующей функции **pipe()** и **fork()**.

## Использование неименованных каналов

---

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
int main (int argc, char * argv[])
{
    int pipedes[2];
    pid_t pid;
    pipe(pipedes);
    pid = fork();
    printf ("Open pipes and write string\n");
    if ( pid > 0 ) {
        char *str = "String passed via pipe\n";
        printf ("Close pipe\n");
        close(pipedes[0]);
        write(pipedes[1], (void *) str, strlen(str) + 1);
        close(pipedes[1]);
    } else {
        char buf[1024];
        int len;
        close(pipedes[1]);
        while ((len = read(pipedes[0], buf, 1024)) != 0)
            write(2, buf, len);
        close(pipedes[0]);
    }
    return 0;
}
```

Оба дескриптора канала хранятся в переменной **pipedes**. После вызова **fork()** процесс раздваивается и родительский процесс (тот, в котором **fork()** вернула ненулевое значение, равное PID дочернего процесса) закрывает дескриптор, открытый для чтения, и записывает данные в канал, используя дескриптор, открытый для записи (**pipedes[1]**). Дочерний процесс (в котором **fork()** вернула 0) закрывает дескриптор, открытый для записи, и затем считывает данные из канала, используя дескриптор, открытый для чтения (**pipedes[0]**). Назначение дескрипторов легко запомнить, сопоставив их с аббревиатурой **I/O** (первый дескриптор - для чтения (**input**), второй - для записи (**output**)). Стандарт POSIX предписывает, чтобы каждый процесс, получивший оба канальных дескриптора, закрывал тот дескриптор, который ему не нужен, перед тем, как начать работу с другим дескриптором, и хотя в системе Linux этим требованием можно пренебречь, лучше все же придерживаться строгих правил.

Запустите файл и просмотрите результат.



```
su@su-P4i65G:~/NetBeansProjects/CppApplication_1/dist/Debug/GNU-Linux-x86$ ./pipes
Open Pipes and write string
Close pipes
su@su-P4i65G:~/NetBeansProjects/CppApplication_1/dist/Debug/GNU-Linux-x86$ Open Pipes and read string
String passed via pipe
Close pipes
```

В нашем примере нам не нужно беспокоиться о синхронизации передачи данных, поскольку ядро системы выполнит всю трудную работу за нас. Но в жизни встречаются и не столь тривиальные случаи. Например, ничто не мешает нам создать несколько дочерних процессов с помощью нескольких вызовов `fork()`. Все эти процессы могут использовать один и тот же канал, при условии, что каждый процесс использует только один из дескрипторов **pipdes**, согласно его назначению. В этой ситуации нам пришлось бы выполнять синхронизацию передачи данных явным образом.

## 6.2 Передача данных через канал

Для передачи данных с помощью каналов используются специальные области памяти (созданные ядром системы), называемые буферами каналов (**pipe buffers**). Одна из важных особенностей буферов каналов заключается в том, что даже если предыдущая запись заполнила буфер не полностью, повторная запись данных в буфер становится возможной только после того, как прежде записанные данные будут прочитаны. Это означает, что если разные процессы, пишущие данные в один и тот же канал, передают данные блоками, размеры которых не превышают объем буферов, данные из блоков, записанных разными процессами, не будут перемешиваться между собой. Использование этой особенности каналов существенно упрощает синхронизацию передачи данных.

Узнать размер буфера можно с помощью вызова функции **fpathconf(pipedes, \_PC\_PIPE\_BUF)**

## Синтаксис

```
#include <unistd.h>

long fpathconf(int filedes, int name);
long pathconf(char *path, int name);
```

## Описание

**fpathconf()** получает значение параметра настроек **name** для открытого описателя файла **filedes**.

**pathconf()** получает значение параметра настроек **name** для имени файла **path**.

Соответствующий макрос, определенный в **<unistd.h>**, выдает минимальные значения; если приложение собирается использовать изменяющиеся значения, то необходимо вызывать функции **fpathconf()** или **pathconf()**, которые выдают результаты более гибким образом.

Если приравнивать **name** следующим константам, то выдаются, соответственно, их режимы работы:

<b>_PC_LINK_MAX</b>	возвращает максимальное количество ссылок на файл. Если <b>filedes</b> или <b>path</b> обращаются к каталогу, то значение применяется ко всему каталогу. Соответствующий макрос - <b>_POSIX_LINK_MAX</b> .
<b>_PC_MAX_CANON</b>	возвращает максимальную длину отформатированной строки ввода, причем, <b>filedes</b> или <b>path</b> должны обращаться к терминалу. Соответствующий макрос - <b>_POSIX_MAX_CANON</b> .
<b>_PC_MAX_INPUT</b>	возвращает максимальную длину строки ввода, причем, <b>filedes</b> или <b>path</b> должны

	обращаться к терминалу. Соответствующий макрос - <b>_POSIX_MAX_INPUT</b> .
<b>_PC_NAME_MAX</b>	возвращает максимальную длину имени файла в каталоге <b>path</b> или <b>filedes</b> , которую процесс разрешает создать. Соответствующий макрос - <b>_POSIX_NAME_MAX</b> .
<b>_PC_PATH_MAX</b>	возвращает максимальную длину относительного имени файла, где <b>path</b> или <b>filedes</b> являются текущими рабочими каталогами. Соответствующий макрос - <b>_POSIX_PATH_MAX</b> .
<b>_PC_PIPE_BUF</b>	возвращает размер буфера каналов, где <b>filedes</b> должно обращаться к каналу или к каналу <b>FIFO</b> , и <b>path</b> должно обращаться к каналу <b>FIFO</b> . Соответствующий макрос - <b>_POSIX_PIPE_BUF</b> .

### Возвращаемые значения

Возвращаются ограничения, если таковые существуют. Если система не имеет ограничений для требуемого ресурса, то возвращается **-1** и переменная **errno** не изменяется. Если есть ошибка, то возвращается **-1**, но в переменную **errno** записывается код ошибки.

### 6.3 Именованные каналы

Хотя в приведенном выше примере неименованные каналы используются только для передачи данных между процессами, связанными «родственными узлами», существует возможность использовать их и для передачи данных между совершенно разными процессами. Для этого нужно организовать передачу дескрипторов канала между неродственными процессами. Для передачи данных между неродственными процессами мы воспользуемся механизмом именованных каналов (**named pipes**), который позволяет каждому процессу получить свой, «законный» дескриптор канала. Передача данных в этих

каналах (как, впрочем, и в однонаправленных неименованных каналах) подчиняется принципу **FIFO** (первым записано - первым прочитано), поэтому в англоязычной литературе иногда можно встретить названия **FIFO pipes** или просто **FIFOs**. Именованные каналы отличаются от неименованных наличием имени, то есть идентификатора канала, потенциально видимого всем процессам системы. Для идентификации именованного канала создается файл специального типа **pipe**. Это еще один представитель семейства виртуальных файлов Unix, не предназначенных для хранения данных (размер файла канала всегда равен нулю). Файлы именованных каналов являются элементами VFS, как и обычные файлы Linux, и для них действуют те же правила контроля доступа. Для создания файлов именованных каналов можно воспользоваться функцией **mkfifo(3)**.

## Синтаксис

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

## Описание

Функция **mkfifo** создает особый **FIFO**-файл с названием **pathname**. **mode** определяет уровни доступа для **FIFO**. Они меняются с помощью процесса **umask** обычным путем: уровни доступа для созданного файла есть (**mode & ~umask**).

Особый **FIFO**-файл похож на обычный канал, только он создается другим путем. Вместо того, чтобы быть анонимным каналом связи, особый **FIFO**-файл подключается к системе с помощью вызова **mkfifo**.

Как только таким образом создан особый **FIFO**-файл, любой процесс может открыть его для чтения или записи так же, как и любой обычный файл. Тем не менее, он должен быть открытым в обоих состояниях одновременно, прежде чем Вы захотите провести в нем операции ввода или вывода. Однако, открытие **FIFO** для чтения обычно блокирует его до тех пор, пока другой процесс не откроет этот же **FIFO** для записи, и наоборот.

## Возвращаемые значения

Обычно возвращаемое значение при успешном завершении работы **mkfifo** равно **0**. В случае ошибки возвращается **-1**, при этом значение переменной **errno** изменяется соответственно ошибке.

После создания файла канала, процессы, участвующие в обмене данными, должны открыть этот файл либо для записи, либо для чтения. Обратите внимание на то, что после закрытия файла канала файл (и соответствующий ему канал) продолжают существовать. Для того чтобы закрыть сам канал, нужно удалить его файл, например с помощью последовательных вызовов **unlink(2)**.

## Синтаксис

```
#include <unistd.h>
int unlink(const char *pathname);
```

## Описание

**unlink** удаляет имя из файловой системы. Если это имя было последней ссылкой на файл и больше нет процессов, которые держат этот файл открытым, данный файл удаляется и место, которое он занимает освобождается для дальнейшего использования.

Если имя было последней ссылкой на файл, но какие-либо процессы всё ещё держат этот файл открытым, файл будет оставлен пока последний файловый дескриптор, указывающий на него, не будет закрыт. Если имя указывает на символическую ссылку, ссылка будет удалена. Если имя указывает на сокет, FIFO или устройство, имя будет удалено, но процессы, которые открыли любой из этих объектов могут продолжать его использовать.

## Возвращаемое значение

В случае успеха возвращается **ноль**. В случае ошибки возвращается **-1** и значение **errno** устанавливается соответствующим образом.

Рассмотрим работу именованного канала на примере простой системы клиент-сервер. Программа-сервер создает канал и передает в него текст, вводимый пользователем с клавиатуры. Программа-клиент читает текст и выводит его на терминал.

Программы из этого примера можно рассматривать как упрощенный вариант системы мгновенного обмена сообщениями между пользователями многопользовательской ОС.

## Сервер для работы с именованными каналами

---

```
/*
    Named Pipe Demo
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

#define FIFO_NAME "./fifofile"

int main(int argc, char * argv[])
{
    FILE * f;
    char ch;
    mkfifo(FIFO_NAME, 0600);
    f = fopen(FIFO_NAME, "w");
    if (f == NULL)
    {
        printf("Error open file\n");
        return -1;
    }
    printf ("Type some text message for sending to client\n");
    printf ("Type 'q' for exit\n");
    do
    {
        ch = getchar();
        fputc(ch, f);
        if (ch == 10) fflush(f);
    } while (ch != 'q');
    fclose(f);
    unlink(FIFO_NAME);
    return 0;
}
```

Вызов функции **mkfifo()** создает файл-идентификатор канала в рабочей директории программы:

```
mkfifo(FIFO_NAME, 0600);
```

где **FIFO\_NAME** - макрос, задающий имя файла канала (в нашем случае - **"/fifofile"**).

В качестве маски доступа мы используем восьмеричное значение **0600**, разрешающее процессу с аналогичными реквизитами пользователя чтение и запись. Для краткости мы не проверяем значение, возвращенное **mkfifo()**, на предмет ошибок. В результате вызова **mkfifo()** с заданными параметрами в рабочей директории программы должен появиться специальный файл **fifofile**. Далее в программе-сервере мы просто открываем созданный файл для записи:

```
f = fopen(FIFO_NAME, "w");
```

Считывание данных, вводимых пользователем, выполняется с помощью **getchar()**, а с помощью функции **fputc()** данные передаются в канал. Работа сервера завершается, когда пользователь вводит символ **"q"**. Исходный текст программы-клиента представлен ниже.

### Клиент для работы с именованным каналом

---

```
/*
    Named Pipe Demo
*/

#include <stdio.h>

#define FIFO_NAME "./fifofile"

int main ()
{
    FILE * f;
    char ch;
    printf ("Start client and wait for server messages...\n");
    f = fopen(FIFO_NAME, "r");
    do
    {
        ch = fgetc(f);
        putchar(ch);
    } while (ch != 'q');
    fclose(f);
    unlink(FIFO_NAME);
    return 0;
}
```

Клиент открывает файл **fifofile** для чтения как обычный файл:

```
f = fopen(FIFO_NAME, "r");
```

Символы, передаваемые по каналу, считываются с помощью функции **fgetc()** и выводятся на экран терминала с помощью **putchar()**. Каждый раз, когда пользователь сервера нажимает ввод, функция **fflush()**, вызываемая сервером, выполняет принудительную очистку буферов канала, в результате чего клиент считывает все переданные символы. Получение символа "q" завершает работу клиента.

Скомпилируйте программы `server.cpp` и `client.cpp` в одной директории. Запустите сначала сервер, потом клиент в разных окнах терминала. Печатайте текст в окне сервера. После каждого нажатия клавиши [Enter] клиент должен отображать строку, напечатанную на сервере.

Одной из сильных сторон Unix/Linux IPC является возможность организовывать взаимодействие между программами, которые не только ничего не знают друг о друге, но и используют разные механизмы ввода/вывода.

Окно терминала с запущенным клиентом

```
su@su-P4i65G:~/NetBeansProjects/CppApplication_1/dist/Debug/GNU-Linux-x86$ ./client
Start client and wait for server messages...
□
```

Окно терминала с запущенным сервером

```
su@su-P4i65G: ~/NetBeansProjects/CppApplication_1/dist/Debug/GNU-Linux-x86
su@su-P4i65G:~/NetBeansProjects/CppApplication_1/dist/Debug/GNU-Linux-x86$ ./server
Type some text message for sending to client.
For exit type 'q'.
```

Результат передачи данных от сервер к клиенту

```
su@su-P4i65G:~/NetBeansProjects/CppApplication_1/dist/Debug/GNU-Linux-x86$ ./client
Start client and wait for server messages...
send some message to client
□

su@su-P4i65G: ~/NetBeansProjects/CppApplication_1/dist/Debug/GNU-Linux-x86
su@su-P4i65G:~/NetBeansProjects/CppApplication_1/dist/Debug/GNU-Linux-x86$ ./server
Type some text message for sending to client.
For exit type 'q'.
send some message to client
```



Каналы представляют собой простое и удобное средство передачи данных, которое, однако, подходит не во всех ситуациях. Например, с помощью каналов довольно трудно организовать обмен асинхронными сообщениями между процессами.

#### 6.4 Последовательность выполнения работы

1. Ознакомиться с теоретическим материалом.
2. Получить у преподавателя индивидуальный вариант задания, который должен предусматривать разработку двух программ (сервер и клиент) и взаимодействие между ними посредством почтовых ящиков или конвейеров.
3. Разработать и отладить сервер и клиент в соответствии с полученным заданием.
4. Написать отчет и представить его к защите вместе с исполняемыми модулями программ и их исходными текстами.

##### 6.4.1 Требования к отчету

Отчет должен содержать следующие разделы:

1. Титульный лист, оформленный согласно утвержденному образцу.
2. Цели выполняемой лабораторной работы.
3. Задание на лабораторную работу.
4. Исходные тексты созданных программ.
5. Результаты работы программ с использованием средств межпроцессорного взаимодействия.
6. Выводы

##### 6.4.2 Дополнительные индивидуальные задания

###### **Вариант №1**

Разработать две программы – сервер и клиент. Клиент отправляет серверу через именованный канал два числа  $L$  и  $U$ , введенные пользователем, где  $L$  – это нижняя граница диапазона,  $U$  – верхняя граница диапазона. Сервер принимает значения границ диапазона из почтового ящика, вычисляет сумму и произведение чисел от  $L$  до  $U$  и выводит полученные значения на экран.

### **Вариант №2**

Разработать две программы – сервер и клиент. Клиент отправляет серверу введенный пользователем номер числа Фибоначчи через именованный канал. Сервер принимает из именованного конвейера номер, вычисляет число Фибоначчи с этим номером, по формуле  $F_i = F_{i-1} + F_{i-2}$ ,  $F_0 = F_1 = 1$  и выводит его на экран.

### **Вариант №3**

Разработать две программы – сервер и клиент. Клиент отправляет серверу через неименованный канал введенную пользователем строку, хранящую знаковое целое число. Сервер принимает из почтового ящика строку, хранящую знаковое целое число, и выводит на экран строковый эквивалент этого числа прописью (например, ввод «-1211» должен приводить к выводу «минус тысяча двести одиннадцать»).

### **Вариант №4**

Разработать две программы – сервер и клиент. Клиент отправляет серверу через именованный канал введенную пользователем строку, хранящую число со знаком и плавающей точкой. Сервер принимает из именованного конвейера строку, хранящую число со знаком и плавающей точкой, и выводит на экран строковый эквивалент этого числа прописью (например, ввод «-12.11» должен приводить к выводу «минус двенадцать целых одиннадцать сотых»).

### **Вариант №5**

Разработать две программы – сервер и клиент. Клиент отправляет серверу через неименованный канал две строки, введенные пользователем. Сервер принимает из почтового ящика две строки. Далее, если обе строки хранят целые числа со знаком, то на экран выводится сумма чисел, в противном случае – конкатенация двух введенных строк.

### **Вариант №6**

Разработать две программы – сервер и клиент. Клиент отправляет серверу через именованный канал элементы двух прямоугольных матриц, введенные пользователем. Сервер принимает из именованного конвейера две прямоугольные матрицы, а затем выводит на экран их сумму и произведение.

#### **Вариант №7**

Разработать две программы – сервер и клиент. Клиент отправляет серверу через неименованный канал элементы вектора (одномерного целочисленного массива), введенные пользователем. Сервер принимает вектор из почтового ящика, упорядочивает его по возрастанию любым из так называемых «улучшенных алгоритмов» сортировки массивов и выводит на экран.

#### **Вариант №8**

Разработать две программы – сервер и клиент. Клиент отправляет серверу через именованный канал элементы вектора (одномерного массива чисел с плавающей точкой), введенные пользователем. Сервер принимает вектор из именованного конвейера, упорядочивает его по возрастанию любым из так называемых «улучшенных алгоритмов» сортировки массивов и выводит на экран.

#### **Вариант №9**

Разработать две программы – сервер и клиент. Клиент отправляет серверу через неименованный канал элементы вектора (одномерного массива строк), введенные пользователем. Сервер принимает вектор из почтового ящика, упорядочивает его по возрастанию любым из так называемых «улучшенных алгоритмов» сортировки массивов и выводит на экран.

#### **Вариант №10**

Разработать две программы – сервер и клиент. Клиент отправляет серверу через именованный канал элементы введенной пользователем квадратной матрицы. Сервер принимает матрицу из именованного конвейера, затем вычисляет сумму элементов, лежащих на главной и побочной диагоналях, и выводит на экран.

#### **Вариант №11**

Разработать две программы – сервер и клиент. Клиент отправляет серверу через неименованный канал элементы введенной пользователем квадратной матрицы. Сервер принимает матрицу из почтового ящика, затем вычисляет сумму элементов, не лежащих на главной и побочной диагоналях, и выводит на экран.

### **Вариант №12**

Разработать две программы – сервер и клиент. Клиент принимает от пользователя две даты – строки вида ЦЦ.ЦЦ.ЦЦЦЦ, где Ц – это любая цифра из диапазона [0-9] и отправляет серверу через именованный канал. Сервер принимает даты из именованного конвейера, вычисляет полное количество дней, прошедших между двумя полученными датами, и выводит его на экран.

### **Вариант №13**

Разработать две программы – сервер и клиент. Клиент принимает от пользователя два значения времени – строки вида ЦЦ.ЦЦ.ЦЦ, где Ц – это любая цифра из диапазона [0-9], и отправляет серверу через неименованный канал. Сервер принимает из почтового ящика обе строки, вычисляет полное количество секунд, прошедших между двумя значениями времени, и выводит его на экран.

### **Вариант №14**

Разработать две программы – сервер и клиент. Клиент отправляет серверу через именованный канал две строки, введенные пользователем. Сервер принимает из именованного конвейера две строки, осуществляет поиск вхождения второй строки в первую любым известным методом, кроме прямого (алгоритм Кнута-Мориса-Пратта, алгоритм Боуэра-Мура), и выводит на экран значение индекса элемента первой строки, с которого началось совпадение, или -1 в противном случае.

### **Вариант №15**

Разработать две программы – сервер и клиент. Клиент отправляет серверу через неименованный канал две строки, введенные пользователем. Сервер принимает из почтового ящика две строки, осуществляет поиск количества вхождений второй строки в первую любым известным методом, кроме прямого (алгоритм Кнута-Мориса-Пратта, алгоритм Боуэра-Мура), и выводит на экран полученное значение.