**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

Факультет прикладной математики и информатики

Кафедра информационных систем управления

**Пивоварчик Александр Витальевич**

**Сравнительный анализ реализации каналов в ОС Unix-Linux и Windows**

Курсовой проект

студента 3 курса 2 группы

**Руководитель:**

Безверхий Александр Анатольевич,

старший преподаватель кафедры   
информационных систем управления

Минск, 2016

**Аннотация**

В курсовом проекте рассмотрены механизмы межпроцессного взаимодействия “Анонимный канал” и “Именованный канал”. Проведен сравнительный анализ их реализации на операционных системах ”Linux”и ”Windows”. Реализованы практические примеры.

**Анатацыя**

У курсавым праекце разгледжаны механізмы міжпрацэснага ўзаемадзеяння "Ананімны канал" і "Найменный канал". Праведзены параўнальны аналіз іх рэалізацыі на аперацыйных сістэмах "Linux" і "Windows". Рэалізаваны практычныя прыклады.

**Annotation**

Mechanisms of interprocess communication "Anonymous pipe" and "Named pipe" are overviewed in the course project. Comparative analysis of their implementation in the operational systems "Linux" and "Windows" are done. Practical examples are implemented.

**Реферат**

Курсовой проект, 29 стр., 3 рис., 9 источников.

Сравнительный анализ реализации каналов в ОС Unix-Linux и Windows

Объектом исследования являются механизмы межпроцессного взаимодействия “Анонимный канал” и “Именованный канал”.

Цель работы: исследование и сравнительный анализ реализаций рассматриваемых механизмов в ОС “Linux” и “Windows”, реализация практических примеров.

Методология проведения работы: системный подход, открытые системы, инженерия знаний, технологии разработки компьютерных систем.

Результаты работы: отчет по сравнительному анализу механизмов, практические примеры.

Область применения результатов: разработка программного обеспечения для ОС ”Linux” и “Windows”.

При форматировании текста учитывались требования ВАК РБ.

**СОДЕРЖАНИЕ**

**ВВЕДЕНИЕ**

**ГЛАВА 1. Анонимные каналы**

1. Общие сведения
2. Практический пример
3. Различия реализаций в ОС ”Linux”и ”Windows”

**ГЛАВА 2. Именованные каналы**

1. Общие сведения
2. Практический пример
3. Различия реализаций в ОС ”Linux”и ”Windows”

**ЗАКЛЮЧЕНИЕ**

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

**ПРИЛОЖЕНИЯ**

**Введение**

С течением времени постоянно нарастала тенденция реализации проектов на основе ряда взаимодействующих процессов, в том числе и на основе процессов запущенных на различных машинах. Соответственно это требовало такого же интенсивного развития механизмов общения процессов. На данный момент как операционная система “Linux”, так и операционная система ”Windows” предоставляют широкий спектр инструментов для организации взаимодействия процессов между собой. Это такие технологии, как:

1. Анонимный канал
2. Именованный канал
3. Разделяемый сегмент памяти
4. Сетевой сокет
5. Проецируемый в память файл
6. И другие…

Задача данной работы является:

1. Детальное рассмотрение механизмов ”Анонимный канал” и ”Именованный канал”.
2. Сравнение их реализаций предоставляемых ОС “Linux” и ”Windows”.
3. Реализации примеров “клиент – сервер” взаимодействия между процессами отдельно для операционной системы “Windows” и отдельно для операционной системы ”Linux”.

**Глава 1- Анонимные каналы**

**1.1 Основные сведения**

Каналы - это старейший из инструментов межпроцессного взаимодействия, существующий приблизительно со времени появления самых ранних версий оперативной системы UNIX.

Каналы представляют собой однонаправленный неструктурированный поток данных фиксированного максимального размера, работающий по принципу очереди («первым пришел, первым вышел»). Отправитель добавляет данные в конец канала, получатель извлекает их из его начала. После того как данные будут прочитаны, они сразу же удаляются из канала и больше недоступны для чтения.

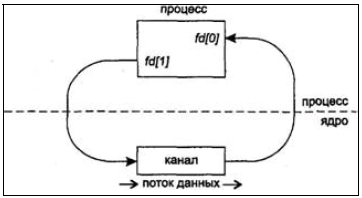
Различают два вида каналов: именованные каналы и анонимные каналы, которые идентичны между собой во всем, кроме способа первоначального обращения к ним процессов.

Как видно из названия анонимные каналы не имеют имени. Таким образом, основной особенностью анонимных каналов является то, что работать с ними могут только родственные процессы, получившие доступ через механизм наследования.

Наиболее общим применением анонимного канала является перенаправление вывода одной программы на вход другой. Это нашло широкое применение в терминалах ”Linux” и ”Windows”. Для этого пользователи объединяют две программы конвейера при помощи оператора «|». Например:

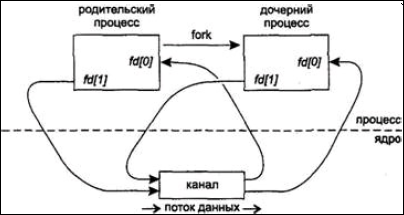
*$ ls | grep ‘cpp’* – вывести на экран список файлов исходного кода.

Вот как это все организуется на уровне ядра операционной системы. Когда процесс создает канал, ядро устанавливает два файловых дескриптора для пользования этим каналом. Один такой дескриптор используется, чтобы открыть путь ввода в канал (запись), в то время как другой применяется для получения данных из канала (чтение). В этом смысле, канал мало применим практически, так как создающий его процесс может использовать канал только для взаимодействия с самим собой. Если процесс посылает данные через канал (fd0), он имеет возможность получить эту информацию из fd1. Хотя канал первоначально связывает процесс с самим собой, данные, идущие через канал, проходят через ядро.



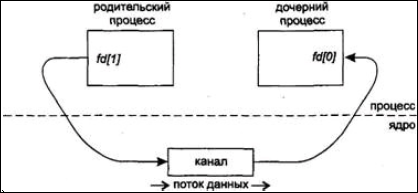
*Рисунок 1 - схема анонимного канала после создания*

Далее процесс, создающий канал, обычно порождает дочерний процесс. Как только дочерний процесс унаследует какой-нибудь открытый файловый дескриптор от родителя, мы получаем базу для межпроцессного общения.



*Рисунок 2 - схема анонимного канала после создания дочернего процесса*

Теперь мы видим, что оба процесса имеют доступ к файловым дескрипторам, которые основывают канал. На этой стадии должно быть принято критическое решение “В каком направлении мы хотим запустить данные?”. Два процесса взаимно согласовываются и "закрывают" неиспользуемый конец канала.



*Рисунок 3 - схема анонимного канала после задания направления*

Конструкция канала теперь полная и готова к передаче данных.

Далее будут описаны основные этапы организации взаимодействия «от родительского к дочернему» через анонимный канал. Общение в обратном направлении организуется аналогично.

1. Родительский процесс создает анонимный канал
2. Родительский процесс порождает дочерний процесс, дескрипторы анонимного канала передается через механизм наследования дескрипторов.
3. Родительский процесс закрывает дескриптор на чтений, таким образам задавая направление общения «от родительского к дочернему».
4. Дочерний процесс заканчивает инициализацию.
5. Дочерний процесс закрывает дескриптор на запись, довершая структуру анонимного канала.
6. Родительский процесс передает данные дочернему.
7. По окончанию общения оба процесса закрывают оставшиеся дескрипторы.

**1.2 Практический пример**

Для демонстрации работы технологии ”Анонимный канал” в ОС ”Linux” и ”Windows” был реализован следующий пример. Родительский процесс создает анонимный канал и затем пораждает дочерний процесс. Дочерний процесс инициализируется и, используя механизм наследования дескрипторов, открывает анонимный канал для общения. Далее серверный процесс читает с консоли целочисленные значения и отправляет их через канал дочернему проессу. Дочерний процесс в свою очередь читает целочисленные сообщения из канала, умножает их на минус один (обрабатывает данные) и выводит на консоль. Общение заканчивается, когда родительский процесс посылает дочернему запрос на остановку.

Код программы для ОС “Linux” находтся в приложении 1. Демонстрация работы приложния для ОС “Linux” находится в приложении 2.

Код программы для “Windows” находтся в приложении 3. Демонстрация работы приложния для ОС “Windows” находится в приложении 4.

**1.3 Различия реализаций в ОС “Linux” и ”Windows”**

Функционально анонимные каналы устроены одинаково на обеих ОС. Основные различия заключаются в программном интерфейсе. Далее приведена таблица соответствия основных функций вовлеченных в организацию взаимодействия посредством анонимного канала.

|  |  |
| --- | --- |
| **Linux** | **Windows** |
| int pipe(int pipe\_handlers[2]) | bool CreatePipe(Handler rh, Handler wh, …) |
| pid\_t fork(void) | bool CreateProcess(…) |
| size\_t read(int fh, void\* buf, size\_t num) | bool ReadFile(Handler fh, void\* buf, …) |
| size\_t write(int fh, void\* buf, size\_t num) | bool WriteFile(Handler fh, void\* buf, …) |
| int close(int handler) | bool CloseHandle(Handle handler) |

**Глава 2 – Именованные каналы**

**2.1 Основные сведения**

Именованный канал во многом работает как анонимный канал, но имеет одно важнейшее отличие. Как видно из названия, именованный канал имеет имя. Из этого следует ряд значимых отличий от анонимного канала:

1. Процессы различного происхождения могут разделять данные через именованный канал, обратившись к нему по имени. Это отличие снимает основное ограничение анонимного канала на родственную связь между взаимодействующими процессами и позволяет строить более гибкие системы взаимодействия.
2. Именованные каналы постоянно существуют в виде специального файла устройства в файловой системе, из которой они удаляются посредством специального системного вызова. Анонимные каналы в свою очередь являются временными объектами ядра и удаляются системой, когда все процессы заканчивают работу с каналом. Таким образом, именованные каналы могут быть легко найдены пользователем непосредственно в файловой системе. Например, в ОС “Linux” это выглядит так:  
   *$ls -l test\_fifo  
   ^prw-r--r-- 1 root root 0 Dec 12 22:15 test\_fifo*Здесь буква “p” в спецификации файла обозначает “pipe” (канал).
3. Именованный канал остается в файловой системе для дальнейшего использования и после того, как весь ввод/вывод сделан. Таким образом, общение по этому же каналу может быть возобновлено через некоторое время. Однако некорректная работа с именованными каналами может приводить к утечке ресурсов.
4. Именованные каналы более уязвимы, чем анонимные каналы. В отличие от анонимных каналов, которые доступны только для родственных процессов, в работу именованного канала может вмешаться совершенно любой процесс, который знает имя канала, и это практически невозможно предотвратить.

Таким образом, именованный канал является некоторым симбиозом анонимного канала и файл. Операции открытия, закрытия, чтения из и записи в именованный канал аналогичны операциям для работы с файлами.

Далее будут описаны основные этапы организации взаимодействия «от клиента к серверу» через именованный канал. Общение в обратном направлении организуется аналогично.

1. Серверный процесс создает именованный канал.
2. Серверный процесс пробует открыть канал для чтения, ожидает подключения клиента.
3. Клиентский процесс ожидает появления свободного канала с указанным именем.
4. Клиентский процесс открывает канал для записи.
5. Клиентский процесс последовательно передает данные серверному процессу.
6. По окончанию взаимодействия оба процесса закрывают дескриптор канала.
7. Серверный процесс удаляет канал из файловой системы.

**2.2 Практический пример**

Для демонстрации работы технологии ”Именованный канал” в ОС ”Linux” и ”Windows” был реализован следующий пример. Серверный процесс создает именованный канал и ожидает клиентского процесса. Клиентский процесс подключается к именованному каналу по известному имени. Далее клиентский процесс читает с консоли целочисленные значения и отправляет их через канал серверому проессу. Серверный процесс в свою очередь читает целочисленные сообщения из канала и выводит их на консоль. Общение заканчивается, когда клиентский процесс посылает серверному запрос на остановку.

Код программы для ОС “Linux” находтся в приложении 5. Демонстрация работы приложния для ОС “Linux” находится в приложении 6.

Код программы для “Windows” находтся в приложении 7. Демонстрация работы приложния для ОС “Windows” находится в приложении 8.

**2.3 Различия реализаций в ОС “Linux” и ”Windows”**

Функциональные отличия:

1. Именованные каналы в ОС “Windows” являются двунаправленными, таким образом, для организации полноценного общения хватает всего одного канала, в то время как в ОС “Linux”нужно создавать два однонаправленных.
2. Именованные каналы в ОС “Windows” позволяют производить обмен целыми сообщениями, таким образом, процесс выполняющий чтение, может считывать сообщения переменной длины в том виде, в котором они были посланы процессом, выполняющим запись.
3. Именованные каналы в ОС “Windows” позволяют организовывать взаимодействие между процессами запущенными на различных машинах в сети. Для этого в имя канала включается адрес машины, где он был создан. Таким образом, именованные каналы в ОС “Windows” похожи на сокеты.
4. Также ОС “Windows” предоставляет возможность создавать несколько независимых экземпляров канала, имеющих одинаковые имена. Например, с единственной серверной системой могут связываться одновременно несколько клиентов, использующих каналы с одним и тем же именем. Каждый клиент может иметь собственный экземпляр именованного канала, и сервер может использовать этот же канал для отправки ответа клиенту.

Как мы видим, функционал именованных каналов в ОС “Windows” значительно расширен и предоставляет более удобный интерфейс для реализации полноценного «сервер – клиент» взаимодействия.

Отличия в программном интерфейсе. Далее приведена таблица соответствия основных функций вовлеченных в организацию взаимодействия посредством именованного канала.

|  |  |
| --- | --- |
| **Linux** | **Windows** |
| int mkfifo(const char\* pathname, mode\_t m) | Handler CreateNamedPipe(LPCTSTR name, …) |
| int open(const char\* pathname, int flag, …) | bool ConnectNamedPipe(Handler handler, …) |
| -//- | Handler CreateFile(LPCTSTR name, …) |
| -//- | bool WaitNamedPipe(LPCTSTR name, …) |
| size\_t read(int fh, void\* buf, size\_t num) | bool ReadFile(Handler fh, void\* buf, …) |
| size\_t write(int fh, void\* buf, size\_t num) | bool WriteFile(Handler fh, void\* buf, …) |
| int close(int handler) | bool CloseHandle(Handle handler) |
| int unlink(const char\* pathname) | bool DisconnectNamedPipe(Handler handler) |

**Заключение**

Из проведенного исследования стало ясно, что механизмы межпроцессного взаимодействия “Анонимный канал” и ”Именованный канал” являются удобными средствами организации общения процессов, являются актуальными на данный момент и находят практическое применения в своей нише.

Сравнительного анализа реализаций данных механизмов на ОС “Linux” и “Windows” показал, что функционально анонимный канал устроен одинаково на обеих ОС и имеет только интерфейсные отличия. Именованные канала в свою очередь имеют ряд отличий, в основном в сторону расширения функционала в ОС “Windows”. Это такие отличия как возможность организации взаимодействия через сеть, работа с несколькими реализациями одного именованного канала, передача данных сообщениями и так далее. Таким образом, механизм “Именованный канал” в ОС “Windows” является более развитой технологией и более удобен для реализации «клиент-серверного» взаимодействия.

**Список использованных источников**

1. Вахалия Ю., Unix изнутри, СПб, 2003г., ISBN 5-94723-013-5
2. Лав Р., Linux. Системное программирование. 2-е изд., СПб, 2014г., ISBN 978-5-496-00747-4
3. Теренс Чан, Системное программирование на С++, Киев, 1997г., ISBN 5-7315-0013-4
4. Рихтер Дж., Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows, 4-е изд., СПб, 2001г., ISBN 5-272-00384-5
5. Харт, Джонсон, М., Системное программирование в среде Windows, 3-е издание, М., 2005г., ISBN 5-8459-0879-5
6. Стивенс У., UNIX: взаимодействие процессов, СПб, 2003г., ISBN 5-318-00534-9
7. <http://www.linuxlib.ru/unixarch/glava_47.htm>
8. <http://citforum.ru/operating_systems/linux_pg/lpg_02.shtml>
9. <http://www.frolov-lib.ru/books/bsp/v27/ch2_3.htm>

**Приложение 1**

linux\_anonymous\_pipe.cpp

#include <unistd.h>

#include <stdio.h>

#include <sys/wait.h>

#define STOP\_MESSAGE -1

int main(int argc, char \*\* argv)

{

pid\_t child\_pid;

// Pair anonymous pipe handlers

int pipe\_handlers[2];

int \*write\_handler\_ptr = &(pipe\_handlers[1]);

int \*read\_handler\_ptr = &(pipe\_handlers[0]);

// Create anonymous pipe

pipe(pipe\_handlers);

// Create child process

if((child\_pid = fork()) == -1)

{

printf("Failed to create child process");

return 1;

}

else if(child\_pid == 0)

{

// Child process

printf("Child process is started\n");

// Close excess anonymous pipe write handler

close(\*write\_handler\_ptr);

// Communicatoion with parent process

int message;

int n\_bytes;

while(true)

{

// Read from pipe

n\_bytes = read(\*read\_handler\_ptr, &message, sizeof(int));

if(n\_bytes <= 0)

{

printf("Failed to read from anonymous pipe\n");

return 1;

}

if(message == STOP\_MESSAGE)

{

printf("Child process has received stop message\n");

break;

}

printf("Child output (multiply by -1): %d\n", -1 \* message);

}

// Close anonymous pipe read handler

close(\*read\_handler\_ptr);

printf("Child process is finished\n");

}

else

{

// Parent

printf("Parent process is started\n");

// Close excess anonypous pipe read handler

close(\*read\_handler\_ptr);

// Communication with child process

int message;

int n\_bytes;

while(true)

{

// Some delay: 10ms // To separate processes outputs

usleep(1000 \* 10);

printf("Parent input (message or %d to stop): ", STOP\_MESSAGE);

scanf("%d", &message);

// Write to pipe

n\_bytes = write(\*write\_handler\_ptr, &message, sizeof(int));

if(n\_bytes <= 0)

{

printf("Failed to write to anonymous pipe\n");

break;

}

if(message == STOP\_MESSAGE)

{

printf("Communication is stopped by user\n");

break;

}

}

// Close anonymous pipe write handler, reading from pipe will be stopped

close(\*write\_handler\_ptr);

printf("Waiting until child process is not finished\n");

waitpid(child\_pid, NULL, 0);

printf("Parent process is finished\n");

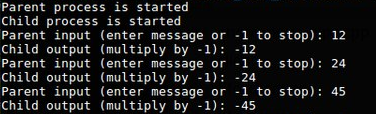
}

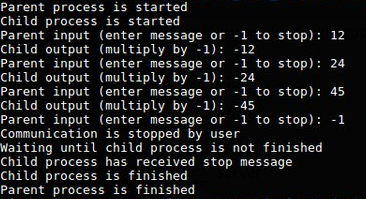
return 0;

}

**Приложение 2**

  
*Рисунок 4 - старт родительского процесса, создание анонимного канала, порождение дочернего*

  
*Рисунок 5 – общение между дочерним и родительским процессом*

  
*Рисунок 6 – остановка дочернего и серверного процессов*

**Приложение 3**

win\_anonymous\_pipe.cpp

#include <stdio.h>

#include <string.h>

#include <windows.h>

#define STOP\_MESSAGE -1

int main(int argc, char \*\* argv)

{

int message;

unsigned long n\_bytes;

bool is\_success;

if (argc == 1)

{

// Parent process // There are not any command line parameters

printf("Parent process is started\n");

// Read and write anonymous pipe handlers

HANDLE hReadPipe, hWritePipe;

// Set pipe configuration, that handlers will be inheritable

SECURITY\_ATTRIBUTES PipeSA;

PipeSA.bInheritHandle = TRUE;

PipeSA.lpSecurityDescriptor = NULL;

PipeSA.nLength = sizeof(SECURITY\_ATTRIBUTES);

// Create anonymous pipe

is\_success = CreatePipe(&hReadPipe, &hWritePipe, &PipeSA, 0);

if (is\_success == false)

{

printf("Failed to create anonymous pipe: %d\n", GetLastError());

system("pause");

return 1;

}

printf("Anonimous pipe is created\n");

// Set child process configurations

char CommandLine[MAX\_PATH] = { '\n' };

strcat(CommandLine, argv[0]);

strcat(CommandLine, " Client");

PROCESS\_INFORMATION ProcInfo;

STARTUPINFO StartInfo;

GetStartupInfo(&StartInfo);

// Transfer anonymous pipe read and write handler to child process

StartInfo.hStdInput = hReadPipe;

StartInfo.hStdError = hWritePipe;

StartInfo.hStdOutput = GetStdHandle(STD\_OUTPUT\_HANDLE);

StartInfo.dwFlags = STARTF\_USESTDHANDLES;

// Create child process from same executable (something like Unix fork())

is\_success = CreateProcess(argv[0], (LPTSTR)CommandLine, NULL, NULL, TRUE, 0,

NULL, NULL, &StartInfo, &ProcInfo);

if (is\_success == false)

{

printf("Failed to create child process: %d\n", GetLastError());

system("pause");

return 1;

}

// Child process handler

HANDLE hChildProcess = ProcInfo.hProcess;

printf("Child process is created\n");

// Close excess anonymous pipe read handler

CloseHandle(hReadPipe);

// Communication with child process

while (true)

{

Sleep(100); // Some delay: 100 ms

printf("Parent input (enter message or %d to stop): ", STOP\_MESSAGE);

scanf("%d", &message);

// Write to pipe

is\_success = WriteFile(hWritePipe, &message, sizeof(int), &n\_bytes, NULL);

if (is\_success == false || n\_bytes != sizeof(int))

{

printf("Failed to write to anonymous pipe\n");

break;

}

if (message == STOP\_MESSAGE)

{

printf("Communication is stopped by user\n");

break;

}

}

// Close anonymous pipe write handler, reading from file will be stopped

CloseHandle(hWritePipe);

printf("Waiting until child process is not finished\n");

WaitForSingleObject(hChildProcess, INFINITE);

printf("Parent process is finished\n");

system("pause");

}

else if (argc == 2 && strcmp(argv[1], "Client") == 0)

{

// Child process // There is one command line parameter and is is equal to "Client"

printf("Child process is started\n");

// Get anonymous pipe read and write handlers

HANDLE hReadPipe = GetStdHandle(STD\_INPUT\_HANDLE);

HANDLE hWritePipe = GetStdHandle(STD\_ERROR\_HANDLE);

// Close excess anonymos pipe write handler

CloseHandle(hWritePipe);

// Communication with parent process

while (true)

{

// Read from pipe

is\_success = ReadFile(hReadPipe, &message, sizeof(int), &n\_bytes, NULL);

if (is\_success == false || n\_bytes != sizeof(int))

{

printf("Failed to read from anonymous pipe\n");

return 1;

}

if (message == STOP\_MESSAGE)

{

printf("Chiled process has received stop message\n");

break;

}

printf("Child output (multiply by -1): %d\n", -1 \* message);

}

// Close anonymous pipe read handler

CloseHandle(hReadPipe);

printf("Child process is finished\n");

}

else

{

printf("Invalid parameters");

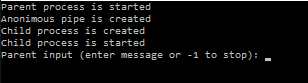
return 1;

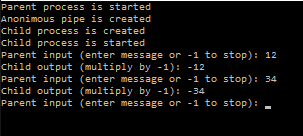
}

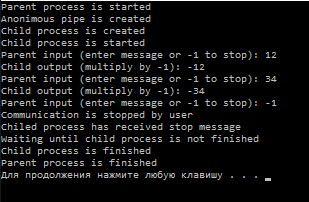
return 0;

}

**Приложение 4**

*****Рисунок 7 - старт родительского процесса, создание анонимного канала, порождение дочернего*

*****Рисунок 8 – общение между родительским и дочерним процессами*

*****Рисунок 9 – остановка дочернего и родительского процессов*

**Приложение 5**

linux\_fifo\_server.cpp

#include <unistd.h>

#include <fcntl.h>

#include <stdio.h>

#include <sys/stat.h>

#define STOP\_MESSAGE -1

int main(int argc, char \*\*argv)

{

// Buffer for fifo file name

char fifo\_name[256];

printf("Enter fifo name (full path) ");

scanf("%s", fifo\_name);

// Create fifo file

if(mkfifo(fifo\_name, S\_IRWXO | S\_IRWXG | S\_IRWXU) != 0)

{

printf("Failed to create fifo: %s\n", fifo\_name);

return 1;

}

printf("Server started, fifo: %s\n", fifo\_name);

// Try to open fifo for reading, blocked until isn`t opened at other side

printf("Start waiting for client\n");

int fifo\_handler = open(fifo\_name, O\_RDONLY, 0);

if(fifo\_handler < 0)

{

printf("Failed to open fifo: %s\n", fifo\_name);

// Release fifo resources

close(fifo\_handler);

unlink(fifo\_name);

return 1;

}

printf("Client is connected\n");

// Communication with client

int message;

int n\_bytes;

while(true)

{

// Read from fifo

n\_bytes = read(fifo\_handler, &message, sizeof(int));

if(n\_bytes <= 0)

{

printf("Failed to read from fifo\n");

break;

}

if(message == STOP\_MESSAGE)

{

printf("Server have received stop message\n");

break;

}

printf("Received: %d\n", message);

}

// Release fifo resources

close(fifo\_handler);

unlink(fifo\_name);

printf("Server is stopped\n");

return 0;

}

linux\_fifo\_client.cpp

#include <stdio.h>

#include <unistd.h>

#include <sys/stat.h>

#include <fcntl.h>

#define STOP\_MESSAGE -1

int main(int argc, char \*\*argv)

{

// Buffer for name of fifo file

char fifo\_name[256];

printf("Enter fifo name (full path) ");

scanf("%255s", fifo\_name);

// Try to open fifo for writing

int fifo\_handler = open(fifo\_name, O\_WRONLY, 0);

if(fifo\_handler < 0)

{

printf("Failed to open fifo: %s\n", fifo\_name);

return 1;

}

printf("Client is connected to fifo: %s\n", fifo\_name);

// Communication with server

int message;

int n\_bytes;

while(true)

{

printf("Sent (message or %d to stop): ", STOP\_MESSAGE);

scanf("%d", &message);

// Write to fifo

n\_bytes = write(fifo\_handler, &message, sizeof(message));

if(n\_bytes != sizeof(message))

{

printf("Failed to write to fifo\n");

break;

}

if(message == STOP\_MESSAGE)

{

printf("Communication is stopped by user\n");

break;

}

}

// Close fifo hanler

close(fifo\_handler);

printf("Client is stopped\n");

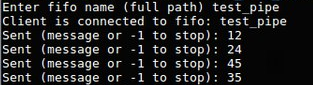
return 0;

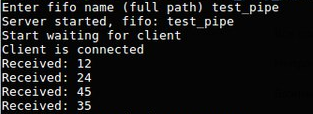
}

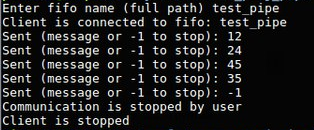
**Приложение 6**

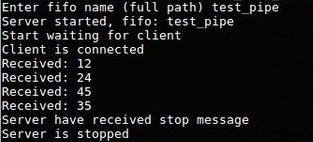
  
*Рисунок 10 - старт сервера, создание именованного канала*

  
*Рисунок 11 – старт клиента, подключение к именованному каналу*

  
*Рисунок12 – запись в канал клиентом*

  
*Рисунок 13 – чтение из канала сервером*

  
*Рисунок 14 – остановка клиента*

  
*Рисунок 15 – остановка сервера*

**Приложение 7**

win\_fifo\_server.cpp

#include <windows.h>

#include <stdio.h>

#include <string.h>

#define STOP\_MESSAGE -1

int main(int argc, char \*\*argv)

{

bool is\_success;

// Buffer for name of and path to fifo file

char fifo\_name[256];

char fifo\_path[256];

printf("Enter fifo name: ");

scanf("%200s", fifo\_name);

sprintf(fifo\_path, "\\\\.\\pipe\\%s", fifo\_name);

// Handler for fifo

HANDLE hFifo;

// Create fifo

hFifo = CreateNamedPipe(fifo\_path, PIPE\_ACCESS\_INBOUND, PIPE\_TYPE\_BYTE | PIPE\_WAIT,

1, 1000, 1000, 100, NULL);

if (hFifo == INVALID\_HANDLE\_VALUE)

{

printf("Failed to create fifo: %s\n", fifo\_name);

system("pause");

return 1;

}

printf("Server started, fifo: %s\n", fifo\_name);

// Try to connect to client

printf("Start waiting for client\n");

is\_success = ConnectNamedPipe(hFifo, NULL);

if (is\_success == false)

{

printf("Waiting of client is failed\n");

system("pause");

return 1;

}

printf("Client is connected\n");

// Communication with client

int message;

unsigned long n\_bytes;

while (true)

{

// Read from fifo

is\_success = ReadFile(hFifo, &message, sizeof(message), &n\_bytes, NULL);

if (is\_success == false || n\_bytes != sizeof(message))

{

printf("Failed to read from fifo\n");

break;

}

if (message == STOP\_MESSAGE)

{

printf("Server has received stop message\n");

break;

}

printf("Received: %d\n", message);

}

// Disconnect client

DisconnectNamedPipe(hFifo);

// Release fifo resources

CloseHandle(hFifo);

printf("Server is stopped\n");

system("pause");

return 0;

}

win\_fifo\_client.cpp

#include <stdio.h>

#include <windows.h>

#include <string.h>

#define STOP\_MESSAGE -1

int main(int argc, char \*\*argv)

{

bool is\_success;

// Buffers for name of and full path to fifo file

char fifo\_name[256];

char fifo\_path[256];

printf("Enter fifo name: ");

scanf("%200s", fifo\_name);

sprintf(fifo\_path, "\\\\.\\pipe\\%s", fifo\_name);

// Handler for fifo

HANDLE hFifo;

// Wait available fifo

is\_success = WaitNamedPipe(fifo\_path, 100);

if (is\_success == false)

{

printf("Failed to find available fifo: %s\n", fifo\_name);

system("pause");

return 1;

}

// Open fifo for writing

hFifo = CreateFile(fifo\_path, GENERIC\_WRITE, 0, NULL, OPEN\_EXISTING,

FILE\_ATTRIBUTE\_NORMAL, NULL);

if (hFifo == INVALID\_HANDLE\_VALUE)

{

printf("Failed to open fifo: %s\n", fifo\_name);

system("pause");

return 1;

}

printf("Client is connected to fifo: %s\n", fifo\_name);

// Communication with server

int message;

unsigned long n\_bytes;

while (true)

{

printf("Sent (message or %d to stop): ", STOP\_MESSAGE);

scanf("%d", &message);

// Write to fifo

is\_success = WriteFile(hFifo, &message, sizeof(message), &n\_bytes, NULL);

if (is\_success == false || n\_bytes != sizeof(message))

{

printf("Failed to write to fifo\n");

break;

}

if (message == STOP\_MESSAGE)

{

printf("Communication is stopped by user\n");

break;

}

}

// Close fifo handle

CloseHandle(hFifo);

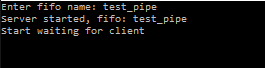
printf("Client is stopped\n");

system("pause");

return 0;

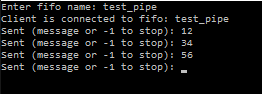
}

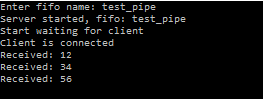
**Приложение 8**

  
*Рисунок 16 – старт сервера, создание именованного канала*

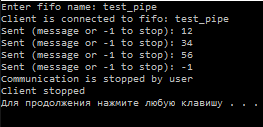
****

*Рисунок 17 – старт клиента, подключение к именованному каналу*

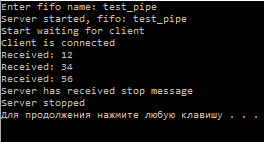
  
*Рисунок 18 – запись в канал клиентом*

****

*Рисунок 19 – чтение из канала сервером*

****

*Рисунок 20 – остановка клиента*

****

*Рисунок 21 – остановка сервера*