Министерство образования и науки РФ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

Тульский государственный университет

КАФЕДРА АВТОМАТИКИ И ТЕЛЕМЕХАНИКИ

ПЕРЕДАЧА ДАННЫХ МЕЖДУ ПРОЦЕССАМИ. ИСПОЛЬЗОВАНИЕ КАНАЛОВ

Лабораторная работа № 4
по курсу «Операционные системы»

Вариант № 3

Выполнил: студент группы 220601 _____ Белым А.А.
(подпись)
Проверил: _____ Попов А.И.
(подпись)

Тула 2012

Цель работы

Целью работы состоит в том, чтобы изучить принципы передачи данных между процессами, научиться применять изученные принципы на практике.

Задание

Написать приложение (клиент), которое передает математическое выражение, а второе (сервер) вычисляет его и передает первому его значение.

Теоретическая справка

Именованные каналы являются объектами ядра ОС Windows, позволяющими организовать межпроцессный обмен не только в изолированной вычислительной системе, но и в локальной сети. Они обеспечивают дуплексную связь и позволяют использовать как потоковую модель, так и модель, ориентированную на сообщения. Обмен данными может быть синхронным и асинхронным.

Каналы должны иметь уникальные в рамках сети имена в соответствии с правилами именования ресурсов в сетях Windows (Universal Naming Convention, UNC), например, \\ServerName\pipe\PipeName. Для общения внутри одного компьютера имя записывается в форме \\.\pipe\PipeName, где "." обозначает локальную машину. Слово "pipe" в составе имени фиксировано, а PipeName - имя, задаваемое пользователем. Эти имена, подобно именам открытых файлов, не являются именами объектов. Они относятся к пространству имен под управлением драйверов файловых систем именованных каналов (\Winnt\System32\Drivers\Npfs.sys).

Через канал можно передавать данные только между двумя процессами. Один из процессов создает канал, другой открывает его. После этого оба процесса могут передавать данные через канал в одну или обе стороны, используя для этого функции, предназначенные для работы с файлами, такие как ReadFile и WriteFile. Заметим, что приложения могут выполнять над каналами Pipe синхронные или асинхронные операции, аналогично тому, как это можно делать с файлами. В случае использования асинхронных операций необходимо отдельно побеспокоиться об организации синхронизации.

Каналы работают, используя следующие принципы:

1. При чтении меньшего числа байт из канала, чем в нем находится, возвращается требуемое число байт, а остальные хранятся в канале;
2. При чтении большего числа байт из канала возвращается доступное число байт и указатель на действительно прочитанное число байт;
3. При чтении из пустого канала возвращается 0 байт. При использовании блокирующих функций READ блокируется до появления данных.
4. Запись данных объемом меньшим, чем объем канала выполняется атомарно.
5. Запись данных объемом большим, чем объем канала блокируем операцию WRITE до освобождения места в канале.

Существуют две разновидности каналов Pipe - именованные (Named Pipes) и анонимные (Anonymous Pipes). Как видно из названия, именованным каналам при создании присваивается имя, которое доступно для других процессов. Зная имя какой-либо рабочей станции в сети, процесс может получить доступ к каналу, созданному на этой рабочей станции.

Анонимные каналы обычно используются для организации передачи данных между родительскими и дочерними процессами, запущенными на одной рабочей станции или на “отдельно стоящем” компьютере.

В простейшем случае один серверный процесс создает один канал (точнее говоря, одну реализацию канала) для работы с одним клиентским процессом. Однако часто требуется организовать взаимодействие одного серверного процесса с несколькими клиентскими. Например, сервер базы данных может принимать от клиентов запросы и рассылать ответы на них. В случае такой необходимости серверный процесс может создать несколько реализаций канала, по одной реализации для каждого клиентского процесса.

Каналы предусматривают несколько режимов работы: блокирующий и неблокирующий, синхронная передача данных и асинхронная.

Для канала, созданного в синхронном блокирующем режиме (с использованием константы PIPE_WAIT), функция **ConnectNamedPipe** переходит в состояние ожидания соединения с клиентским процессом. Если канал создан в синхронном не блокирующем режиме (с использованием константы PIPE_NOWAIT), функция **ConnectNamedPipe** немедленно возвращает управление с кодом TRUE, если только клиент был отключен от данной реализации канала и возможно подключение этого клиента. В противном случае возвращается значение FALSE. Дальнейший анализ необходимо выполнять с помощью функции GetLastError.

Если параметр функции **ConnectNamedPipe** pOverlapped указан как NULL, функция выполняется в синхронном режиме. В противном случае используется асинхронный режим. Для этого в функции **CreateNamedPipe** нужно установить атрибут FILE_FLAG_OVERLAPPED, а в функции **ConnectNamedPipe** указать адрес структуры pOverlapped.

Неименованные каналы используются для передачи данных только в одном направлении (только чтение или только запись). Это может понадобиться для того, чтобы запустить какую-нибудь внешнюю утилиту и управлять ее поведением из своей программы.

Инструкция пользователю

Данный комплекс программ (клиент(ы) + сервер) позволяет вычислять математические выражения.

Приступая работе, запустите программу-сервер. После этого можно приступать к запуску клиентов. Перед сеансом работы необходимо подключится к серверу, локальному или удаленному. Если сервер относительно клиента находится на удаленной машине, необходимо ввести его сетевое имя. После подключения вводите ваше выражение в комбинированное поле ввода. Для вычисления выражения нажмите специальную кнопку или клавишу <Enter>.

Инструкция программисту

Серверная часть.

```
DWORD WINAPI MainServerThread(LPVOID PARAMS)
```

Главный поток сервера. Создает новые каналы, ожидает подключение клиента, при подключении создает новый обслуживающий поток. Параметр игнорируется.

```
DWORD WINAPI ServiceThread(LPVOID lpvParam)
```

Обслуживающий поток, в качестве параметра принимает дескриптор обслуживаемого канала. Принимает сообщения клиента, создает ответное сообщение с помощью функции `GenerateReply`, и отправляет его клиенту.

```
VOID GenerateReply(LPTSTR pchRequest,LPTSTR pchReply, LPDWORD pchBytes )
```

Функция принимает выражение `pchRequest` от клиента, пробует его вычислить. Если вычисление прошло успешно, возвращается результат, преобразованный в строку по указателю `pchReply`, если нет – возвращается строковое сообщение об ошибке (по тому же указателю). Также возвращает размер результата по указателю `pchBytes`.

Клиентская часть.

```
int MainWindow::OpenPipe(LPTSTR lpszPipename);
```

Открывает канал с именем `lpszPipename`. Указанное имя – чистое имя канала, без родительских директорий в виртуальной ФС. Возвращает в случае ошибки значение функции `GetLastError`, иначе возвращает 0.

```
int MainWindow::Communicate(LPTSTR lpvMessage);
```

Отправляет сообщение `lpvMessage` в канал, ранее открытый процедурой `MainWindow::OpenPipe`. Возвращает в случае ошибки значение функции `GetLastError`, иначе возвращает 0.

Текст программы

Ниже представлен текст программ для вычисления математического выражения с использованием каналов для его передачи и написанных на языке C++, в среде Qt Creator 2.5.2 + MinGW-GCC 4.6 с использованием библиотеки Qt.

Серверная часть.

mainwindow.h:

```
#ifndef MAINWINDOW_H
```

```
#define MAINWINDOW_H

#include <QMainWindow>
#define WINVER 0x0620

extern "C"{
#include <windows.h>
}
#include <muParser.h>
const int BUFSIZE=512;

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private:
    Ui::MainWindow *ui;
private slots:
    void addToLog(QString msg);
};

#endif // MAINWINDOW_H
```

mainwindow.cpp:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QDebug>
#include <wchar.h>
#include "threadmanager.h"
#define ADD_TO_LOG(msg) manager.emitAddToLog(QString()<<msg);

QString &operator<<(QString s,const QString &other){
    return s+=other;
}

QString &operator<<(QString s,const char* other){
    return s+=QString(other);
}

QString &operator<<(QString s,const int other){
    QString tmp;
    return s+=tmp.sprintf("%d",other);
}

DWORD WINAPI ServiceThread(LPVOID);
VOID GenerateReply(LPTSTR, LPTSTR, LPDWORD);

ThreadManager manager;

DWORD WINAPI MainServerThread(LPVOID PARAMS){
    BOOL   fConnected = FALSE;
    DWORD  dwThreadId = 0;
    HANDLE hPipe = INVALID_HANDLE_VALUE, hThread = NULL;
    LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\MythematicaPipe");
    for(;;){
        SECURITY_ATTRIBUTES m_pSecAttrib;
        SECURITY_DESCRIPTOR m_pSecDesc;
```

```
        InitializeSecurityDescriptor(&m_pSecDesc,
        SECURITY_DESCRIPTOR_REVISION);


        SetSecurityDescriptorDacl(&m_pSecDesc,TRUE,(PACL)NULL,FALSE);

        m_pSecAttrib.nLength = sizeof(SECURITY_ATTRIBUTES);
        m_pSecAttrib.bInheritHandle = TRUE;
        m_pSecAttrib.lpSecurityDescriptor = &m_pSecDesc;
        hPipe=CreateNamedPipe(lpszPipename,
                              PIPE_ACCESS_DUPLEX,
                              PIPE_TYPE_MESSAGE|PIPE_READMODE_MESSAGE|PIPE_WAIT,
                              PIPE_UNLIMITED_INSTANCES,
                              BUFSIZE,                  // output buffer size
                              BUFSIZE,                  // input buffer size
                              0,                        // client time-out
                              &m_pSecAttrib);
        if (hPipe == INVALID_HANDLE_VALUE)
         {
             ADD_TO_LOG("CreateNamedPipe failed, GLE="<<GetLastError()<<".\n");
             return -1;
         }

         // Wait for the client to connect; if it succeeds,
         // the function returns a nonzero value. If the function
         // returns zero, GetLastError returns ERROR_PIPE_CONNECTED.

         fConnected = ConnectNamedPipe(hPipe, NULL) ?
            TRUE : (GetLastError() == ERROR_PIPE_CONNECTED);

         if (fConnected)
         {
            ADD_TO_LOG("Клиент подключен, запускается обслуживающий поток.\n");

            // Create a thread for this client.
            hThread = CreateThread(
               NULL,              // no security attribute
               0,                 // default stack size
               ServiceThread,    // thread proc
               (LPVOID) hPipe,    // thread parameter
               0,                 // not suspended
               &dwThreadId);      // returns thread ID

            if (hThread == NULL)
            {
               ADD_TO_LOG("CreateThread failed, GLE="<<GetLastError()<<".\n");
               return -1;
            }
            else CloseHandle(hThread);
          }
         else
           // The client could not connect, so close the pipe.
            CloseHandle(hPipe);
    }

    return 0;
}

DWORD WINAPI ServiceThread(LPVOID lpvParam)
// This routine is a thread processing function to read from and reply to a client
// via the open pipe connection passed from the main loop. Note this allows
// the main loop to continue executing, potentially creating more threads of
// of this procedure to run concurrently, depending on the number of incoming
// client connections.
{
```

```
  HANDLE hHeap      = GetProcessHeap();
  TCHAR* pchRequest = (TCHAR*)HeapAlloc(hHeap, 0, BUFSIZE*sizeof(TCHAR));
  TCHAR* pchReply   = (TCHAR*)HeapAlloc(hHeap, 0, BUFSIZE*sizeof(TCHAR));

  DWORD cbBytesRead = 0, cbReplyBytes = 0, cbWritten = 0;
  BOOL fSuccess = FALSE;
  HANDLE hPipe  = NULL;

  // Do some extra error checking since the app will keep running even if this
  // thread fails.

  if (lpvParam == NULL)
  {
      ADD_TO_LOG( "\nERROR - Pipe Server Failure:\n");
      ADD_TO_LOG( "   InstanceThread got an unexpected NULL value in
lpvParam.\n");
      ADD_TO_LOG( "   InstanceThread exitting.\n");
      if (pchReply != NULL) HeapFree(hHeap, 0, pchReply);
      if (pchRequest != NULL) HeapFree(hHeap, 0, pchRequest);
      return (DWORD)-1;
  }

  if (pchRequest == NULL)
  {
      ADD_TO_LOG( "\nERROR - Pipe Server Failure:\n");
      ADD_TO_LOG( "   InstanceThread got an unexpected NULL heap allocation.\n");
      ADD_TO_LOG( "   InstanceThread exitting.\n");
      if (pchReply != NULL) HeapFree(hHeap, 0, pchReply);
      return (DWORD)-1;
  }

  if (pchReply == NULL)
  {
      ADD_TO_LOG( "\nERROR - Pipe Server Failure:\n");
      ADD_TO_LOG( "   InstanceThread got an unexpected NULL heap allocation.\n");
      ADD_TO_LOG( "   InstanceThread exitting.\n");
      if (pchRequest != NULL) HeapFree(hHeap, 0, pchRequest);
      return (DWORD)-1;
  }

  // Print verbose messages. In production code, this should be for debugging
only.
  ADD_TO_LOG("Обслуживающий поток создан, ожидает сообщение.\n");

// The thread's parameter is a handle to a pipe object instance.

  hPipe = (HANDLE) lpvParam;

// Loop until done reading
  while (1)
  {
  // Read client requests from the pipe. This simplistic code only allows messages
  // up to BUFSIZE characters in length.
     fSuccess = ReadFile(
        hPipe,        // handle to pipe
        pchRequest,    // buffer to receive data
        BUFSIZE*sizeof(TCHAR), // size of buffer
        &cbBytesRead, // number of bytes read
        NULL);        // not overlapped I/O

     if (!fSuccess || cbBytesRead == 0)
     {
         if (GetLastError() == ERROR_BROKEN_PIPE)
         {
             ADD_TO_LOG("Обслуживающий поток: клиент отключен, ошибка
#"<<GetLastError()<<".\n");
```

```
            }
            else
            {
                ADD_TO_LOG("Обслуживающий поток: ошибка ReadFile #"<<GetLastEr-
ror()<<".\n");
            }
            break;
        }

   // Process the incoming message.
      GenerateReply(pchRequest, pchReply, &cbReplyBytes);

   // Write the reply to the pipe.
      fSuccess = WriteFile(
         hPipe,        // handle to pipe
         pchReply,     // buffer to write from
         cbReplyBytes, // number of bytes to write
         &cbWritten,   // number of bytes written
         NULL);        // not overlapped I/O

      if (!fSuccess || cbReplyBytes != cbWritten)
      {
           ADD_TO_LOG("Обслуживающий поток: ошибка WriteFile #"<<GetLastEr-
ror()<<".\n");
           break;
      }
  }

// Flush the pipe to allow the client to read the pipe's contents
// before disconnecting. Then disconnect the pipe, and close the
// handle to this pipe instance.

   FlushFileBuffers(hPipe);
   DisconnectNamedPipe(hPipe);
   CloseHandle(hPipe);

   HeapFree(hHeap, 0, pchRequest);
   HeapFree(hHeap, 0, pchReply);

   ADD_TO_LOG("InstanceThread exitting.\n");
   return 1;
}

VOID GenerateReply(LPTSTR pchRequest,
                       LPTSTR pchReply,
                       LPDWORD pchBytes )
// This routine is a simple function to print the client request to the console
// and populate the reply buffer with a default data string. This is where you
// would put the actual client request processing code that runs in the context
// of an instance thread. Keep in mind the main thread will continue to wait for
// and receive other client connections while the instance thread is working.
{
    mu::Parser parser;
    QString tmp= QString::fromWCharArray(pchRequest);
   ADD_TO_LOG("Получен запрос от клиента:\""<<tmp<<"\"\n");
   parser.SetExpr(tmp.toAscii().constData());
   QString res;
   try{
       res.sprintf("%f",parser.Eval());
   } catch (mu::Parser::exception_type &e){
       res=QString(e.GetMsg().c_str());
       ADD_TO_LOG("Ошибка парсера:"<<res);
   }
   LPTSTR buf=new wchar_t[res.length()+1];
   res.toWCharArray(buf);
   buf[res.length()]='\0';
```

```
    // Check the outgoing message to make sure it's not too long for the buffer.
    if (!wcscpy( pchReply, buf))
    {
        *pchBytes = 0;
        pchReply[0] = 0;
        ADD_TO_LOG("Ошибка при копировании ответного сообщения.\n");
        return;
    }
    *pchBytes = (lstrlen(pchReply)+1)*sizeof(TCHAR);
    delete buf;
}



MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    connect(&manager,SIG-
NAL(addToLog(QString)),this,SLOT(addToLog(QString)),Qt::BlockingQueuedConnection);
    CreateThread(NULL,
                 0,
                 &MainServerThread,
                 (LPVOID)NULL,
                 0,
                 (PDWORD)NULL);
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::addToLog(QString msg){
    ui->plainTextEdit->appendPlainText(msg);
}
```

Клиентская часть.

mainwindow.h:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QCompleter>
#include <QLabel>
#define WINVER 0x0620

extern "C"{
#include <windows.h>
}
const int BUFSIZE=512;
const char pipe_name[]="\\\\%s\\pipe\\MythematicaPipe";
namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
```

```
private slots:

    void on_comboBox_activated(const QString &arg1);

    void on_pushButton_2_clicked();

    void on_action_eval_triggered();

    void on_pushButton_clicked();

private:
    HANDLE hPipe;
    QCompleter complete;

    Ui::MainWindow *ui;
    int OpenPipe(LPTSTR lpszPipename);
    int Communicate(LPTSTR lpvMessage);
};

#endif // MAINWINDOW_H
```

mainwindow.cpp:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

#include <QDebug>
#include <wchar.h>

#define ADD_TO_LOG(msg) ui->plainTextEdit->appendPlainText(QString()<<msg);

QString &operator<<(QString s,const QString &other){
    return s+=other;
}

QString &operator<<(QString s,const char* other){
    return s+=QString(other);
}

QString &operator<<(QString s,const int other){
    QString tmp;
    return s+=tmp.sprintf("%d",other);
}

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    hPipe=INVALID_HANDLE_VALUE;
    ui->setupUi(this);
    ui->dockWidget->setFloating(true);
    ui->dockWidget->close();
    statusBar()->showMessage("Не подключено.");
    complete.setCaseSensitivity(Qt::CaseSensitive);
    ui->comboBox->setCompleter(&complete);
}

MainWindow::~MainWindow()
{
    delete ui;
}

int MainWindow::OpenPipe(LPTSTR lpszPipename){
    // Try to open a named pipe; wait for it, if necessary.
    if(hPipe!=INVALID_HANDLE_VALUE)
```

```
        CloseHandle(hPipe);
    DWORD dwMode;
    BOOL fSuccess = FALSE;
    int e;
       while (1)
       {

          hPipe = CreateFile(
             lpszPipename,   // pipe name
             GENERIC_READ |  // read and write access
             GENERIC_WRITE,
             0,              // no sharing
             NULL,           // default security attributes
             OPEN_EXISTING,  // opens existing pipe
             0,              // default attributes
             NULL);          // no template file

       // Break if the pipe handle is valid.

          if (hPipe != INVALID_HANDLE_VALUE)
             break;

          // Exit if an error other than ERROR_PIPE_BUSY occurs.

          if ((e=GetLastError()) != ERROR_PIPE_BUSY)
          {
             ADD_TO_LOG("Could not open pipe. GLE="<<e<<".\n");
             return e;
          }

          // All pipe instances are busy, so wait for 20 seconds.

          if ( ! WaitNamedPipe(lpszPipename, 20000))
          {
             ADD_TO_LOG("Could not open pipe: 20 second wait timed out.");
             return WAIT_TIMEOUT;
          }
       }

    // The pipe connected; change to message-read mode.

       dwMode = PIPE_READMODE_MESSAGE;
       fSuccess = SetNamedPipeHandleState(
          hPipe,    // pipe handle
          &dwMode,  // new pipe mode
          NULL,     // don't set maximum bytes
          NULL);    // don't set maximum time
       if ( ! fSuccess)
       {
          e=GetLastError();
          ADD_TO_LOG("SetNamedPipeHandleState failed. GLE="<<e<<".\n");
          return e;
       }
       return 0;

}

int MainWindow::Communicate(LPTSTR lpvMessage){
       int e;
       BOOL fSuccess = FALSE;
       TCHAR  chBuf[BUFSIZE];
       DWORD  cbRead, cbToWrite, cbWritten;
       cbToWrite = (lstrlen(lpvMessage)+1)*sizeof(TCHAR);
       ADD_TO_LOG("Sending "<<cbToWrite<<" byte message: \""<<QString::fromWCha-
rArray(lpvMessage)<<"\"\n");
       fSuccess = WriteFile(
```

```
            hPipe,                  // pipe handle
            lpvMessage,             // message
            cbToWrite,              // message length
            &cbWritten,             // bytes written
            NULL);                  // not overlapped

        if ( ! fSuccess)
        {
            e=GetLastError();
            ADD_TO_LOG("WriteFile to pipe failed. GLE="<<e<<".\n");
            return e;
        }

        ADD_TO_LOG("\nMessage sent to server, receiving reply as follows:\n");

        do
        {
        // Read from the pipe.

            fSuccess = ReadFile(
               hPipe,    // pipe handle
               chBuf,    // buffer to receive reply
               BUFSIZE*sizeof(TCHAR),  // size of buffer
               &cbRead,  // number of bytes read
               NULL);    // not overlapped

            if ( ! fSuccess && GetLastError() != ERROR_MORE_DATA )
               break;
            QString res=QString::fromWCharArray(chBuf);
            ADD_TO_LOG("Answer received from server:\""<<res<<"\".\n");
            ui->textBrowser->append(res);
        } while ( ! fSuccess);  // repeat loop if ERROR_MORE_DATA

        if ( ! fSuccess)
        {
            e=GetLastError();
            ADD_TO_LOG("ReadFile from pipe failed. GLE="<<e<<".\n");
            return e;
        }
        return 0;

}

void MainWindow::on_comboBox_activated(const QString &arg1)
{
    ui->action_eval->trigger();
}

void MainWindow::on_pushButton_2_clicked()
{
    QString server;
    if(ui->radioButton->isChecked()){
        const char *serv_name=ui->lineEdit->text().toAscii().constData();
        server.sprintf(pipe_name,serv_name);
        QString tmp;
        statusBar()->showMessage(tmp.sprintf("Подключение к %s...",serv_name));
    } else {
        server.sprintf(pipe_name,".");
        statusBar()->showMessage("Подключение к локальному серверу...");
    }
    qDebug()<<server;
    LPTSTR buf=new wchar_t[server.length()+1];
    server.toWCharArray(buf);
    buf[server.length()]='\0';
    int e;
    if(!(e=OpenPipe(buf))){
```

```
        if(ui->radioButton->isChecked()){
            const char *serv_name=ui->lineEdit->text().toAscii().constData();
            QString tmp;
            statusBar()->showMessage(tmp.sprintf("Подключено к %s.",serv_name));
        } else {
            statusBar()->showMessage("Подключено к локальному серверу.");
        }
    } else {
        QString tmp;
        statusBar()->showMessage(tmp.sprintf("Ошибка подключения #%d.",e));
    }
    delete buf;
}

void MainWindow::on_action_eval_triggered()
{
    QString expr=ui->comboBox->currentText();
    LPTSTR buf=new wchar_t[expr.length()+1];
    expr.toWCharArray(buf);
    buf[expr.length()]='\0';
    Communicate(buf);
    delete buf;
}

void MainWindow::on_pushButton_clicked()
{
    ui->comboBox->addItem(ui->comboBox->currentText());
    ui->action_eval->trigger();
}
```

Тестовый пример

На рисунке 1 представлен пример работы комплекса программ для вычисления математических выражений.

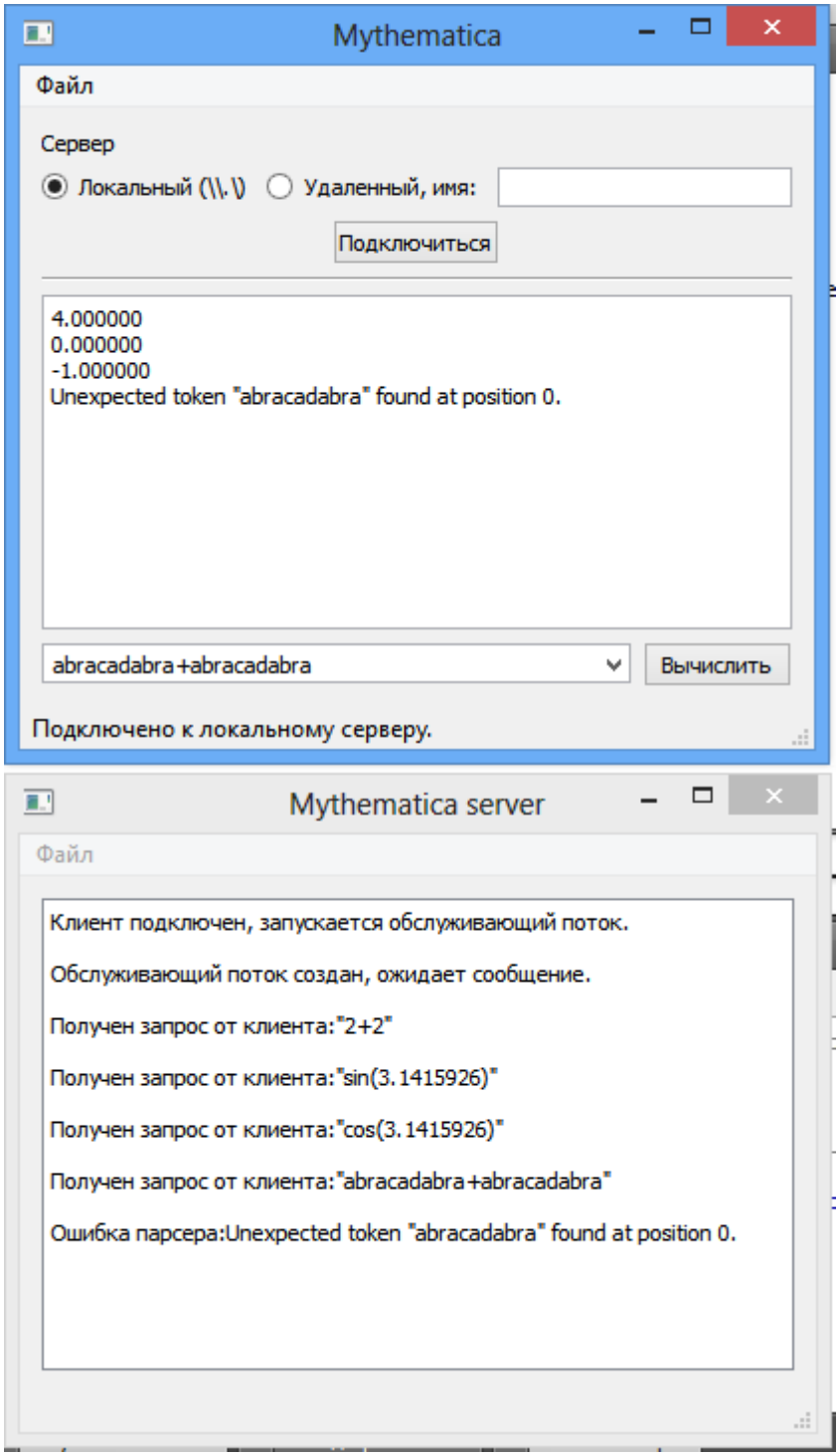

Рисунок 1— Пример вычисления математического выражения

Вывод

Именованные каналы являются объектами ядра ОС Windows, позволяющими организовать межпроцессный обмен не только в изолированной вычислительной системе, но и в локальной сети. Анонимные (неименованные) каналы позволяют передавать данные в одну сторону на локальной машине. Каналы являются классическим средством межпроцессного взаимодействия.