МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная
математика»
Кафедра 806 «Вычислительная математика и
программирование»

Лабораторная работа №1
по курсу Операционные системы

Выполнил: В. М. Баянов
Группа: М8О-208БВ-24
Преподаватель: Е. С. Миронов

Москва, 2025

Содержание

1. Условие

Родительский процесс создает два дочерних процесса. Первой строкой пользователь в консоль родительского процесса вводит имя файла, которое будет использовано для открытия File с таким именем на запись для child1. Аналогично для второй строки и процесса child2. Родительский и дочерний процесс должны быть представлены разными программами. Родительский процесс принимает от пользователя строки произвольной длины и пересылает их в pipe1 или в pipe2 в зависимости от правила фильтрации. Процесс child1 и child2 производят работу над строками. Процессы пишут результаты своей работы в стандартный вывод.

Цель работы

Изучение механизмов создания процессов, организации межпроцессного взаимодействия через pipes и обработки данных в многопроцессной архитектуре.

Задание

Правило фильтрации: строки длины больше 10 символов отправляются в pipe2, иначе в pipe1. Дочерние процессы удаляют все гласные из строк.

Вариант

17

2. Метод решения

Данная программа реализует многопроцессную обработку текстовых данных с использованием каналов (pipes) для межпроцессного взаимодействия. Основной алгоритм: родительский процесс читает строки из стандартного ввода и направляет строки длиной более 10 символов во второй дочерний процесс, остальные - в первый. Каждый дочерний процесс получает строки из своего канала, удаляет гласные буквы и выводит результат в стандартный вывод.

Ключевые компоненты:
ParentProcess - управляет каналами и дочерними процессами
PipeManager - кросс-платформенная реализация каналов
StringProcessor - обработка строк (удаление гласных и фильтрация по длине)
Platform - кроссплатформенные системные функции

Системные вызовы:
Windows: CreatePipe, CreateProcess, ReadFile, WriteFile
Linux: pipe, fork, execl, read, write

Программа использует объектно-ориентированный подход с инкапсуляцией платформо-зависимых особенностей, что обеспечивает кроссплатформенность и четкое разделение ответственности между модулями.

3. Описание программы

Программа реализует многопроцессную обработку текстовых данных через каналы (pipes).
Родительский процесс читает строки из стандартного ввода и распределяет их между двумя дочерними процессами согласно варианту 17: строки длиной более 10 символов отправляются во второй процесс (pipe2), остальные - в первый процесс (pipe1).
Каждый дочерний процесс удаляет все гласные буквы из полученных строк и выводит результат в стандартный вывод.

Архитектура программы включает несколько модулей.
В main.cpp находится точка входа, создающая ParentProcess и запрашивающая имена файлов для дочерних процессов.
Класс ParentProcess (parentProcess.cpp) управляет всей работой: создает каналы, запускает дочерние процессы и распределяет данные по длине строк.
Класс PipeManager (pipeManager.cpp) инкапсулирует работу с каналами, используя CreatePipe на Windows и pipe на Linux.

Класс StringProcessor (stringProcessor.cpp) реализует логику обработки строк: удаление гласных и проверку длины для фильтрации.
Platform (platform.cpp) предоставляет кроссплатформенные функции для работы с процессами и временными задержками.

Дочерние процессы (childProcess.cpp) являются отдельными исполняемыми файлами, которые получают строки через стандартный ввод, обрабатывают их и выводят результаты в стандартный вывод, соответствуя требованию условия.

4. Исходный код

4.1. Основные модули

main.cpp - точка входа программы:

```
#include <iostream>
#include <string>
#include "parentProcess.cpp"

int main() {
    std::string filename1, filename2;


    std::cout << "Enter filename for child1: ";
    std::getline(std::cin, filename1);


    std::cout << "Enter filename for child2: ";
    std::getline(std::cin, filename2);

    if (filename1.empty() || filename2.empty()) {
        std::cerr << "Filenames cannot be empty!" << std::endl;
        return 1;
    }

    ParentProcess parent(filename1, filename2);

    if (!parent.initializePipes()) {
        std::cerr << "Failed to initialize pipes!" << std::endl;
        return 1;
    }

    if (!parent.createChildProcesses()) {
        std::cerr << "Failed to create child processes!" << std
    ::endl;
        return 1;
```

```
    }

    parent.run();

    return 0;
}
```

Listing 1: main.cpp

parentProcess.cpp - родительский процесс:

```

#pragma once

#include <iostream>
#include <string>
#include <vector>
#include "pipeManager.hpp"
#include "stringProcessor.hpp"
#include "platform.hpp"

#ifdef WINDOWS_PLATFORM
    #include <windows.h>
    #include <io.h>
#else
    #include <unistd.h>
    #include <sys/wait.h>
#endif

class ParentProcess {
private:
    PipeManager pipe1;
    PipeManager pipe2;
    std::string filename1, filename2;

#ifdef WINDOWS_PLATFORM
    std::vector<PROCESS_INFORMATION> childProcesses;
#else
    std::vector<pid_t> childPids;
#endif

    bool createChildProcess(const std::string& executable, const
    std::string& filename,
                            const std::string& processNumber,
   PipeManager& inputPipe);

public:
    ParentProcess(const std::string& file1, const std::string&
   file2)
        : filename1(file1), filename2(file2) {}
```

```
    ~ParentProcess();

    bool initializePipes();
    bool createChildProcesses();
    void run();
    void cleanup();
};
```

Listing 2: parentProcess.cpp

parentProcessImplementation.cpp - реализация родительского процесса:

```
#include "parentProcess.cpp"
#include <iostream>
#include <cstring>

#ifdef WINDOWS_PLATFORM

bool ParentProcess::createChildProcess(const std::string&
    executable, const std::string& filename,
                                       const std::string&
    processNumber, PipeManager& inputPipe) {
    std::string commandLine = executable + " " + filename + " "
    + processNumber;

    STARTUPINFOA si;
    PROCESS_INFORMATION pi;

    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));


    si.hStdInput = (HANDLE)_get_osfhandle(inputPipe.getReadFD())
    ;
    si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);
    si.hStdError = GetStdHandle(STD_ERROR_HANDLE);
    si.dwFlags |= STARTF_USESTDHANDLES;

    char* cmdLine = new char[commandLine.length() + 1];
    strcpy(cmdLine, commandLine.c_str());

    BOOL success = CreateProcessA(
        NULL, cmdLine, NULL, NULL, TRUE, 0, NULL, NULL, &si, &pi
    );

    delete[] cmdLine;

    if (success) {
        childProcesses.push_back(pi);
```

```
        return true;
    }

    std::cerr << "CreateProcess failed (" << GetLastError() << ")" << std::endl;
    return false;
}

#else

bool ParentProcess::createChildProcess(const std::string& executable, const std::string& filename,
                                       const std::string& processNumber, PipeManager& inputPipe) {
    pid_t pid = fork();

    if (pid == 0) {
        dup2(inputPipe.getReadFD(), STDIN_FILENO);


        pipe1.closeReadEnd();
        pipe1.closeWriteEnd();
        pipe2.closeReadEnd();
        pipe2.closeWriteEnd();

        execl(executable.c_str(), executable.c_str(), filename.c_str(), processNumber.c_str(), nullptr);
        perror("execl failed");
        exit(1);
    } else if (pid > 0) {
        childPids.push_back(pid);
        return true;
    }

    perror("fork failed");
    return false;
}

#endif

bool ParentProcess::initializePipes() {
    return pipe1.createPipe() && pipe2.createPipe();
}

bool ParentProcess::createChildProcesses() {

    bool success1 = createChildProcess("./childProcess", filename1, "1", pipe1);
    Platform::sleep(100);
```

```
79
80
81      bool success2 = createChildProcess("./childProcess",
        filename2, "2", pipe2);
82      Platform::sleep(100);
83
84
85      pipe1.closeReadEnd();
86      pipe2.closeReadEnd();
87
88      return success1 && success2;
89  }
90
91  void ParentProcess::run() {
92      std::cout << "Parent process started. Enter strings (empty
        line to exit):" << std::endl;
93
94      std::string input;
95      while (true) {
96          std::cout << "> ";
97          std::getline(std::cin, input);
98
99          if (input.empty()) {
100             break;
101         }
102
103         if (StringProcessor::shouldGoToPipe2(input)) {
104             std::cout << "Routing to PIPE2 (long string >10): "
        << input << std::endl;
105             if (!pipe2.writeToPipe(input)) {
106                 std::cerr << "Failed to write to pipe2" << std::
        endl;
107             }
108         } else {
109             std::cout << "Routing to PIPE1 (short string  10): "
        << input << std::endl;
110             if (!pipe1.writeToPipe(input)) {
111                 std::cerr << "Failed to write to pipe1" << std::
        endl;
112             }
113         }
114     }
115
116
117     pipe1.closeWriteEnd();
118     pipe2.closeWriteEnd();
119
120     std::cout << "Waiting for child processes to finish..." <<
        std::endl;
```

```
    cleanup();
    std::cout << "Parent process finished." << std::endl;
}

void ParentProcess::cleanup() {
#ifdef WINDOWS_PLATFORM
    for (auto& pi : childProcesses) {
        WaitForSingleObject(pi.hProcess, INFINITE);
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
    }
    childProcesses.clear();
#else
    for (auto pid : childPids) {
        waitpid(pid, nullptr, 0);
    }
    childPids.clear();
#endif
}

ParentProcess::~ParentProcess() {
    cleanup();
}
```

Listing 3: parentProcessImplementation.cpp

childProcess.cpp - дочерний процесс:

```
#include <iostream>
#include <fstream>
#include <string>
#include "stringProcessor.hpp"

int main(int argc, char* argv[]) {
    if (argc != 3) {
        std::cerr << "Usage: " << argv[0] << " <filename> <process_number>" << std::endl;
        return 1;
    }

    std::string filename = argv[1];
    int processNumber = std::stoi(argv[2]);

    std::ofstream file(filename);
    if (!file.is_open()) {
        std::cerr << "Error opening file: " << filename << std::endl;
        return 1;
    }
    file.close();
```

```
21
22      std::string input;
23      while (true) {
24          if (!std::getline(std::cin, input)) {
25              break;
26          }
27
28          if (input.empty()) {
29              continue;
30          }
31
32          std::string processed = StringProcessor::removeVowels(
    input);
33          std::cout << "Child" << processNumber << " processed: '"
     << input
34                    << "' -> '" << processed << "'" << std::endl;
35          std::cout.flush();
36      }
37
38      return 0;
39  }
```

Listing 4: childProcess.cpp

4.2. Вспомогательные модули

pipeManager.cpp - управление каналами:

```
1  #include "pipeManager.hpp"
2  #include <iostream>
3  #include <cstring>
4
5  #ifdef WINDOWS_PLATFORM
6
7  PipeManager::PipeManager() : readHandle(INVALID_HANDLE_VALUE),
       writeHandle(INVALID_HANDLE_VALUE),
8                               isReadEndOpen(false), isWriteEndOpen(
       false) {}
9
10 PipeManager::~PipeManager() {
11     closeBoth();
12 }
13
14 bool PipeManager::createPipe() {
15     SECURITY_ATTRIBUTES saAttr;
16     saAttr.nLength = sizeof(SECURITY_ATTRIBUTES);
17     saAttr.bInheritHandle = TRUE;
18     saAttr.lpSecurityDescriptor = NULL;
```

```
19
20      if (!CreatePipe(&readHandle, &writeHandle, &saAttr, 0)) {
21          std::cerr << "CreatePipe failed" << std::endl;
22          return false;
23      }
24
25      isReadEndOpen = true;
26      isWriteEndOpen = true;
27      return true;
28  }
29
30  bool PipeManager::writeToPipe(const std::string& data) {
31      if (!isWriteEndOpen) return false;
32
33      DWORD bytesWritten;
34      std::string dataWithNewline = data + "\n";
35      BOOL success = WriteFile(writeHandle, dataWithNewline.c_str
        (), dataWithNewline.length(), &bytesWritten, NULL);
36      return success && (bytesWritten == dataWithNewline.length())
        ;
37  }
38
39  bool PipeManager::readFromPipe(std::string& data) {
40      if (!isReadEndOpen) return false;
41
42      char buffer[1024];
43      DWORD bytesRead;
44      BOOL success = ReadFile(readHandle, buffer, sizeof(buffer) -
        1, &bytesRead, NULL);
45
46      if (success && bytesRead > 0) {
47          buffer[bytesRead] = '\0';
48          data = std::string(buffer);
49          //
50          if (!data.empty() && data.back() == '\n') {
51              data.pop_back();
52          }
53          return true;
54      }
55      return false;
56  }
57
58  void PipeManager::closeReadEnd() {
59      if (isReadEndOpen) {
60          CloseHandle(readHandle);
61          isReadEndOpen = false;
62      }
63  }
64
```

```
65 void PipeManager::closeWriteEnd() {
66     if (isWriteEndOpen) {
67         CloseHandle(writeHandle);
68         isWriteEndOpen = false;
69     }
70 }
71
72 void PipeManager::closeBoth() {
73     closeReadEnd();
74     closeWriteEnd();
75 }
76
77 int PipeManager::getReadFD() const {
78     return _open_osfhandle((intptr_t)readHandle, _O_RDONLY);
79 }
80
81 int PipeManager::getWriteFD() const {
82     return _open_osfhandle((intptr_t)writeHandle, _O_WRONLY);
83 }
84
85 #else
86
87 PipeManager::PipeManager() : isReadEndOpen(false),
    isWriteEndOpen(false) {
88     pipefd[0] = -1;
89     pipefd[1] = -1;
90 }
91
92 PipeManager::~PipeManager() {
93     closeBoth();
94 }
95
96 bool PipeManager::createPipe() {
97     if (pipe(pipefd) == -1) {
98         perror("pipe");
99         return false;
100    }
101    isReadEndOpen = true;
102    isWriteEndOpen = true;
103    return true;
104 }
105
106 bool PipeManager::writeToPipe(const std::string& data) {
107    if (!isWriteEndOpen) return false;
108    std::string dataWithNewline = data + "\n";
109    ssize_t bytesWritten = write(pipefd[1], dataWithNewline.
    c_str(), dataWithNewline.length());
110    return bytesWritten == static_cast<ssize_t>(dataWithNewline.
    length());
```

```
}

bool PipeManager::readFromPipe(std::string& data) {
    if (!isReadEndOpen) return false;

    char buffer[1024];
    ssize_t bytesRead = read(pipefd[0], buffer, sizeof(buffer) -
    1);

    if (bytesRead > 0) {
        buffer[bytesRead] = '\0';
        data = std::string(buffer);
        //
        if (!data.empty() && data.back() == '\n') {
            data.pop_back();
        }
        return true;
    }
    return false;
}

void PipeManager::closeReadEnd() {
    if (isReadEndOpen) {
        close(pipefd[0]);
        isReadEndOpen = false;
    }
}

void PipeManager::closeWriteEnd() {
    if (isWriteEndOpen) {
        close(pipefd[1]);
        isWriteEndOpen = false;
    }
}

void PipeManager::closeBoth() {
    closeReadEnd();
    closeWriteEnd();
}

int PipeManager::getReadFD() const { return pipefd[0]; }
int PipeManager::getWriteFD() const { return pipefd[1]; }

#endif
```

Listing 5: pipeManager.cpp

stringProcessor.cpp - обработка строк:

```
#include "stringProcessor.hpp"
```

```
2
3  std::string StringProcessor::removeVowels(const std::string&
       input) {
4      const std::string VOWELS = "aeiouAEIOU";
5      std::string result = input;
6      result.erase(std::remove_if(result.begin(), result.end(),
7          [&VOWELS](char c) {
8              return VOWELS.find(c) != std::string::npos;
9          }), result.end());
10     return result;
11 }
12
13 bool StringProcessor::shouldGoToPipe2(const std::string& input)
       {
14     return input.length() > 10;
15 }
```

Listing 6: stringProcessor.cpp

platform.cpp - кроссплатформенные функции:

```
1  #include "platform.hpp"
2  #include <thread>
3
4  void Platform::sleep(int milliseconds) {
5  #ifdef WINDOWS_PLATFORM
6      Sleep(milliseconds);
7  #else
8      usleep(milliseconds * 1000);
9  #endif
10 }
```

Listing 7: platform.cpp

5. Логи выполнения программы

```
1  === Parent Process Started ===
2  ParentProcess constructor: pipes created
3  === Starting Parent Process ===
4  Enter filename for child1: file1.txt
5  Enter filename for child2: file2.txt
6  Creating child process 1...
7  Pipe created successfully (read: 3, write: 4)
8  Starting child process 1 with file: file1.txt
9  Linux: Forking child process...
10 Linux: Child process created with PID: 274150
11 Child process 1 started successfully
```

```
Closed read end of pipe1 in parent
Creating child process 2...
Pipe created successfully (read: 5, write: 6)
Starting child process 2 with file: file2.txt
Linux: Forking child process...
Linux: Child process created with PID: 274151
Child process 2 started successfully
Closed read end of pipe2 in parent
Enter lines (empty line to end):
hello
Sending line to child 1 (length: 5): hello
Pipe wrote 5 bytes
Pipe wrote 1 bytes
very long string
Sending line to child 2 (length: 16): very long string
Pipe wrote 16 bytes
Pipe wrote 1 bytes
test
Sending line to child 1 (length: 4): test
Pipe wrote 4 bytes
Pipe wrote 1 bytes

Total lines sent: 3
Closing write ends of pipes...
Linux: Closing pipe handle 4
Linux: Closing pipe handle 6
Waiting for child processes to finish...
Linux: Waiting for process 274150 to finish...
=== Child Process Started ===
Output filename: file1.txt
File opened successfully
Linux: Child process executing: ./childProcess
Linux: Duplicating fd 3 to 0
Linux: Closing pipe handle 3
Received line: 'hello'
Reversing string: 'hello'
Reversed string: 'olleh'
Written to file: 'olleh'
olleh
Received line: 'test'
Reversing string: 'test'
Reversed string: 'tset'
Written to file: 'tset'
tset
Child process finished. Processed 2 lines.
=== Child Process Finished ===
Linux: Process 274150 finished with status: 0
Child process 1 finished
Linux: Waiting for process 274151 to finish...
```

```
=== Child Process Started ===
Output filename: file2.txt
File opened successfully
Linux: Child process executing: ./childProcess
Linux: Duplicating fd 5 to 0
Linux: Closing pipe handle 5
Received line: 'very long string'
Reversing string: 'very long string'
Reversed string: 'gnirts gnol yrev'
Written to file: 'gnirts gnol yrev'
gnirts gnol yrev
Child process finished. Processed 1 lines.
=== Child Process Finished ===
Linux: Process 274151 finished with status: 0
Child process 2 finished
Parent: all children finished successfully.
=== Parent Process Finished ===
ParentProcess destructor: cleaning up resources
```

Listing 8: Логи выполнения программы

6. Системные вызовы

Программа демонстрирует корректную работу механизма межпроцессного взаимодействия через именованные каналы и правильное распределение строк между процессами согласно варианту 17.