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: Е. Г. Туймуков
Группа: М8О-208БВ-24
Преподаватель: Е. С. Миронов

Москва, 2025

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 инвертируют строки.

Вариант: 20

Метод решения

Данная программа реализует многопроцессную обработку текстовых данных с использованием каналов (pipes) для межпроцессного взаимодействия. Основной алгоритм: родительский процесс запрашивает имена двух файлов, читает строки из стандартного ввода и направляет их в два дочерних процесса по правилу фильтрации: строки длиной более 10 символов отправляются второму процессу, остальные - первому. Каждый дочерний процесс получает строки из своего канала, инвертирует их (переворачивает задом наперёд) и записывает результаты в указанный файл, а также выводит в стандартный вывод (stdout).

Ключевые компоненты: ParentProcess - управляет каналами и дочерними процессами Pipe - кроссплатформенная реализация каналов ChildProcess - запускает дочерние процессы utils - содержит функцию для инверсии строк

Системные вызовы: Windows: CreatePipe, CreateProcess, ReadFile, WriteFile, CloseHandle, WaitForSingleObject Linux/macOS: pipe, fork, execvp, read, write, close, waitpid, dup2

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

Описание программы

Программа реализует многопроцессную обработку текстовых данных через каналы (pipes). Родительский процесс запрашивает у пользователя имена двух файлов для записи результатов, читает строки из стандартного ввода и распределяет их между двумя дочерними процессами: строки длиной более 10 символов отправляются второму процессу, остальные - первому. Каждый дочерний процесс читает строки из своего канала, переворачивает их задом наперёд и записывает результат в указанный файл, а также выводит в стандартный вывод (stdout).

Архитектура программы включает несколько модулей. В parent.cpp находится точка входа, создающая объект ParentProcess. Класс ParentProcess (parentprocess.cpp) управляет всей работой: создает каналы, запрашивает имена файлов, запускает дочерние процессы и распределяет строки по правилу фильтрации. Класс Pipe (pipe.cpp) инкапсулирует работу с каналами, используя CreatePipe на Windows и pipe на Linux/macOS. Класс ChildProcess (childprocess.cpp) отвечает за запуск дочерних процессов через CreateProcess (Windows)

или fork/execvp (Linux/macOS). Файл child.cpp реализует дочерний процесс, который читает строки из стандартного ввода (подключенного к каналу), инвертирует их с помощью функции reverseString из utils.cpp и записывает в файл и stdout. Модули oslinux.cpp и oswin.cpp обеспечивают кроссплатформенность, реализуя системные вызовы для соответствующих платформ.

Результаты

Разработанная программа успешно реализует многопроцессную архитектуру для параллельной обработки текстовых данных.
В ходе решения были достигнуты следующие ключевые результаты:
Корректная работа системы межпроцессного взаимодействия
Реализованы два независимых канала передачи данных между родительским и дочерними процессами
Обеспечено четкое распределение строк по длине строк
Достигнута синхронизация процессов через блокирующие операции чтения/записи
Кросс-платформенная функциональность
Программа корректно работает как в Windows, так и в Linux/Unix системах
Реализована унифицированная абстракция для работы с каналами через класс Pipe
Обеспечен единый интерфейс для создания процессов на разных платформах

Выводы

В ходе лабораторной работы успешно разработана многопроцессная система обработки текстовых данных с использованием межпроцессного взаимодействия через каналы. Программа демонстрирует корректную работу в кросс-платформенном режиме на Windows и Unix системах.

Исходная программа

childprocess.cpp

```
#include "../include/child_process.hpp"
#include "../include/os.h"

ChildProcess::ChildProcess(Pipe* p,const std::string& f,bool is_c1)
: pipe(p),file_name(f),is_child1(is_c1),pid(INVALID_PIPE_HANDLE) {}

void ChildProcess::execute() {
#ifdef _WIN32
const char* exe = "child.exe";
#else
const char* exe = "./child";
#endif
char* argv[] = { (char*)"child",(char*)file_name.c_str(),NULL };
pid = CreateChildProcess(exe,argv,pipe->getReadFd());
}
```

child.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include "../include/utils.h"

int main(int argc,char** argv) {
if (argc <2) {
std::cerr <<"No filename provided" <<std::endl;
return 1;
}
std::string filename = argv[1];
std::ofstream file(filename);
if (!file.is_open()) {
std::cerr <<"Failed to open file: " <<filename <<std::endl;
return 1;
}

std::string line;
while (std::getline(std::cin,line)) {
std::string reversed = reverseString(line);
file <<reversed <<std::endl;
std::cout <<reversed <<std::endl;  // Вывод в stdout
}

file.close();
return 0;
}
```

os_linux.cpp

```
#include "../include/os.h"
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>
#include <cstdio> // Для perror

int CreatePipe(PipeHandle fd[2]) {
return pipe(fd);
}

int PipeRead(PipeHandle fd,char* buf,size_t size) {
return read(fd,buf,size);
}

int PipeWrite(PipeHandle fd,const char* buf,size_t size) {
return write(fd,buf,size);
}

int ClosePipe(PipeHandle fd) {
return close(fd);
}

ProcessHandle CreateChildProcess(const char* exe,char* const* argv,PipeHandle
stdin_handle) {
pid_t pid = fork();
if (pid == -1) {
perror("fork");
return -1;
}
if (pid == 0) {
// Child
if (stdin_handle != INVALID_PIPE_HANDLE) {
Dup2(stdin_handle,STDIN_FILENO);
ClosePipe(stdin_handle);
}
// Закрыть write end,но поскольку pipe в parent,child не имеет его
execvp(exe,argv);
perror("execvp");
_exit(1);
}
return pid;
}

int WaitProcess(ProcessHandle pid) {
int status;
return waitpid(pid,&status,0);
}
```

```
int Dup2(PipeHandle oldfd,int newfd) {
return dup2(oldfd,newfd);
}

os_win.cpp
#include "../include/os.h"
#include <iostream> // Для std::cerr

int CreatePipe(PipeHandle fd[2]) {
SECURITY_ATTRIBUTES saAttr;
saAttr.nLength = sizeof(SECURITY_ATTRIBUTES);
saAttr.bInheritHandle = TRUE;
saAttr.lpSecurityDescriptor = NULL;

if (!CreatePipe(&fd[0],&fd[1],&saAttr,0)) {
std::cerr <<"CreatePipe failed" <<std::endl;
return -1;
}

// Make write end non-inheritable
if (!SetHandleInformation(fd[1],HANDLE_FLAG_INHERIT,0)) {
std::cerr <<"SetHandleInformation failed" <<std::endl;
return -1;
}

return 0;
}

int PipeRead(PipeHandle fd,char* buf,size_t size) {
DWORD bytesRead;
if (!ReadFile(fd,buf,size,&bytesRead,NULL)) {
return -1;
}
return bytesRead;
}

int PipeWrite(PipeHandle fd,const char* buf,size_t size) {
DWORD bytesWritten;
if (!WriteFile(fd,buf,size,&bytesWritten,NULL)) {
return -1;
}
return bytesWritten;
}

int ClosePipe(PipeHandle fd) {
return !CloseHandle(fd);
}
```

```
ProcessHandle CreateChildProcess(const char* exe,char* const* argv,PipeHandle
stdin_handle) {
PROCESS_INFORMATION pi;
STARTUPINFO si;
ZeroMemory(&si,sizeof(si));
si.cb = sizeof(si);
ZeroMemory(&pi,sizeof(pi));

si.dwFlags |= STARTF_USESTDHANDLES;
si.hStdInput = stdin_handle;
si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);  // Оставляем stdout как есть
si.hStdError = GetStdHandle(STD_ERROR_HANDLE);

std::string cmdline;
for (int i = 0; argv[i]; ++i) {
if (i >0) cmdline += " ";
cmdline += argv[i];
}

if (!CreateProcessA(NULL,(LPSTR)cmdline.c_str(),NULL,NULL,TRUE,0,NULL,NULL,&si,&pi))
{
std::cerr <<"CreateProcess failed (" <<GetLastError() <<")" <<std::endl;
return INVALID_HANDLE_VALUE;
}

// Close thread handle
CloseHandle(pi.hThread);

return pi.hProcess;
}

int WaitProcess(ProcessHandle pid) {
WaitForSingleObject(pid,INFINITE);
CloseHandle(pid);
return 0;
}

int Dup2(PipeHandle oldfd,int newfd) {
// No-op on Windows,since we set in STARTUPINFO
return 0;
}

parentprocess.cpp
#include "../include/parent_process.hpp"
#include <iostream>

ParentProcess::ParentProcess()
: pipe1(new Pipe()),pipe2(new Pipe()),child1(nullptr),child2(nullptr) {}
```

```
ParentProcess::~ParentProcess() {
delete pipe1;
delete pipe2;
delete child1;
delete child2;
}

void ParentProcess::start() {
std::cout <<"Enter filename for child1: ";
std::getline(std::cin,file1);
std::cout <<"Enter filename for child2: ";
std::getline(std::cin,file2);

child1 = new ChildProcess(pipe1,file1,true);
child1->execute();
// Close read end in parent
ClosePipe(pipe1->getReadFd());

child2 = new ChildProcess(pipe2,file2,false);
child2->execute();
// Close read end in parent
ClosePipe(pipe2->getReadFd());

std::cout <<"Enter lines (empty line to end):" <<std::endl;
std::string line;
while (std::getline(std::cin,line)) {
if (line.empty()) break;
Pipe* target_pipe = (line.length() >10) ? pipe2 : pipe1;
target_pipe->write(line.c_str(),line.length());
target_pipe->write("\n",1);
}

ClosePipe(pipe1->getWriteFd());
ClosePipe(pipe2->getWriteFd());

WaitProcess(child1->getPid());
WaitProcess(child2->getPid());

std::cout <<"Parent: children finished." <<std::endl;
}

parent.cpp
#include "../include/parent_process.hpp"

int main() {
ParentProcess parent;
parent.start();
```

```
return 0;
}
```

pipe.cpp

```
#include "../include/pipe.hpp"

Pipe::Pipe() {
if (CreatePipe(fd) != 0) {
// Обработка ошибки,но для простоты пропустим
}
}

Pipe::~Pipe() {
ClosePipe(fd[0]);
ClosePipe(fd[1]);
}

ssize_t Pipe::read(char* buf,size_t size) {
return PipeRead(fd[0],buf,size);
}

ssize_t Pipe::write(const char* buf,size_t size) {
return PipeWrite(fd[1],buf,size);
}
```

cmake

```
#include "../include/utils.h"

std::string reverseString(const std::string& s) {
std::string rev;

// Просто идем с конца строки и добавляем символы в новую строку
for (int i = s.size() -1; i >= 0; i--) {
rev += s[i];
}

return rev;
}
```

utils.cpp

```
cmake_minimum_required(VERSION 3.10)
project(lab1)

set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

include_directories(include)
```

```
set(UTILS_SOURCES src/utils.cpp)

if(UNIX)
set(PLATFORM_SOURCES src/os_linux.cpp)
elseif(WIN32)
set(PLATFORM_SOURCES src/os_win.cpp)
endif()

add_executable(parent
src/parent.cpp
src/parent_process.cpp
src/pipe.cpp
src/child_process.cpp
${UTILS_SOURCES}
${PLATFORM_SOURCES}
)

add_executable(child
src/child.cpp
${UTILS_SOURCES}
)

if(WIN32)
target_link_libraries(parent ws2_32)
endif()
```