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: Д. И. Шнайдер
Группа: М8О-208БВ-24
Преподаватель: Е. С. Миронов

Москва, 2025

Условие

Родительский процесс создает дочерний процесс. Первой строкой пользователь в консоль родительского процесса вводит имя файла, которое будет использовано для открытия File с таким именем на запись. Перенаправление стандартных потоков ввода-вывода показано на картинке выше. Родительский и дочерний процесс должны быть представлены разными программами. Родительский процесс принимает от пользователя строки произвольной длины и пересылает их в pipe1. Процесс child проверяет строки на валидность правилу. Если строка соответствует правилу, то она выводится в стандартный поток вывода дочернего процесса, иначе в pipe2 выводится информация об ошибке. Родительский процесс полученные от child ошибки выводит в стандартный поток вывода.

Цель работы: Приобретение практических навыков управления процессами в операционных системах семейства Windows и Linux/Unix, а также организация межпроцессного взаимодействия с использованием неименованных каналов (pipes). Дополнительной целью являлась разработка кроссплатформенного решения, абстрагирующего особенности системных API.

Задание: Разработать программу из трёх процессов: родительского и двух дочерних, взаимодействующих через неименованные каналы.

Родительский процесс должен:

- Запрашивать у пользователя имя файла и передавать его дочерним процессам
- Принимать от пользователя строки и распределять их по чётности:
  - Нечётные строки (по порядку ввода) отправлять в pipe1 первому дочернему процессу
  - Чётные строки отправлять в pipe2 второму дочернему процессу

Дочерние процессы должны:

- Получать строки из соответствующих каналов
- Инвертировать полученные строки (записывать символы в обратном порядке)
- Записывать результат в общий выходной файл

Вариант: 15

Метод решения

Для решения задачи применена архитектура с двумя процессами (родительским и дочерним), взаимодействующими через два неименованных канала (pipe).

Основной алгоритм работы

1. **Инициализация:** Создание двух каналов - pipe1 для передачи данных от родителя к потомку, pipe2 для обратной связи (сообщения об ошибках)
2. **Запуск процесса:** Создание дочернего процесса с перенаправлением стандартных потоков ввода/вывода

3. **Передача параметров:** Отправка имени файла через pipe1 как первого сообщения
4. **Обработка данных:**
    - Родительский процесс читает строки от пользователя и передает через pipe1
    - Дочерний процесс проверяет каждую строку на соответствие критерию (начало с заглавной буквы)
    - Валидные строки записываются в файл, сообщения об ошибках отправляются через pipe2
5. **Завершение работы:** Корректное закрытие каналов и процессов при получении пустой строки

Особенности реализации

Для обеспечения кроссплатформенности разработан уровень абстракции, скрывающий различия между API Windows и Unix-систем. Реализована поддержка как латинских, так и кириллических символов при проверке заглавных букв.

Описание программы

Программа реализована в модульном стиле и состоит из четырех основных компонентов.

Модуль parent.c

Реализует логику родительского процесса:

- Создание и управление каналами связи
- Запуск и контроль дочернего процесса
- Взаимодействие с пользователем (ввод строк)
- Координация передачи данных между процессами
- Обработка сообщений об ошибках от дочернего процесса

Модуль child.c

Содержит бизнес-логику дочернего процесса:

- Чтение входных данных из канала pipe1
- Валидация строк по критерию (начало с заглавной буквы)
- Запись валидных строк в выходной файл
- Формирование и отправка сообщений об ошибках через pipe2
- Управление файловыми операциями

Модуль cross_platform.h/c

Предоставляет кроссплатформенные абстракции:

- **Структуры данных:** pipe_t (для каналов), process_t (для процессов)
- **Функции работы с каналами:** создание, закрытие, чтение, запись
- **Функции управления процессами:** создание, ожидание завершения
- **Вспомогательные функции:** перенаправление потоков, работа с памятью

Модуль string_utils.h/c

Содержит функции обработки строк:

- `is_capital_start()` - проверка начала строки с заглавной буквы с поддержкой латиницы и кириллицы
- `trim_newline()` - удаление символов новой строки

Используемые системные вызовы

- **Windows:** CreateProcess, CreatePipe, ReadFile, WriteFile, CloseHandle
- **Unix:** fork, pipe, dup2, read, write, close, waitpid
- **Кроссплатформенные:** fopen, fclose, fgets, fprintf, fflush

Архитектура программы обеспечивает четкое разделение ответственности между модулями и поддерживает работу в различных операционных средах.

Результаты

В результате работы была разработана кроссплатформенная программа для межпроцессного взаимодействия, успешно функционирующая как в Windows, так и в Unix-подобных операционных системах.

Ключевые особенности реализации

- **Кроссплатформенная архитектура:** Программа использует единый код для различных ОС благодаря системе абстракций в модуле `cross_platform`
- **Поддержка Unicode:** Реализована проверка заглавных букв как для латинского алфавита (ASCII), так и для кириллицы (UTF-8)
- **Асинхронная обработка ошибок:** Родительский процесс проверяет канал ошибок без блокировки основного потока выполнения
- **Корректное управление ресурсами:** Обеспечено правильное закрытие дескрипторов каналов и процессов при завершении работы

Пример работы программы

```
Enter file name: output.txt
Enter string (empty string to exit): hello world
Child: Error: string must start with capital letter -'hello world'
Enter string (empty string to exit): MAI
Enter string (empty string to exit): 123Start
Child: Error: string must start with capital letter -'123Start'
Enter string (empty string to exit): Aviation
Enter string (empty string to exit):
Parent process finished.
```

Содержимое файла output.txt:

```
MAI
Aviation
```

Производительность

Программа демонстрирует стабильную работу при обработке строк различной длины. Время отклика системы на ввод пользователя практически не отличается от времени работы обычных консольных приложений, что подтверждает эффективность выбранного подхода к организации межпроцессного взаимодействия.

Выводы

В ходе выполнения лабораторной работы были успешно достигнуты все поставленные цели и решены основные задачи:

1. **Освоены механизмы управления процессами:** На практике применены системные вызовы для создания и управления процессами в различных операционных системах (`fork()`, `waitpid()` в Unix и `CreateProcess()`, `WaitForSingleObject()` в Windows)

2. **Реализовано межпроцессное взаимодействие:** Организован эффективный обмен данными между независимыми процессами с использованием неименованных каналов (pipe), что позволило обеспечить разделение функциональности между родительским и дочерним процессами

3. **Создано кроссплатформенное решение:** Разработана система абстракций, позволяющая программе компилироваться и работать в различных операционных системах без изменения бизнес-логики приложений

4. **Решены практические проблемы:**

   - Реализована поддержка многобайтовых кодировок (UTF-8) для корректной обработки кириллических символов
   - Организовано асинхронное чтение из каналов для своевременного получения сообщений об ошибках

- Обеспечено корректное освобождение системных ресурсов (дескрипторов файлов и процессов)

Работа продемонстрировала важность создания переносимого и устойчивого к ошибкам программного обеспечения, а также необходимость тщательного проектирования архитектуры приложений, использующих межпроцессное взаимодействие. Полученные навыки могут быть применены при разработке более сложных распределенных систем и приложений, требующих параллельной обработки данных.

Исходная программа

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "cross_platform.h"
#include "string_utils.h"

#define BUFFER_SIZE 1024

int main(void) {
    process_t child;
    char line[BUFFER_SIZE];
    char childBuf[BUFFER_SIZE];
    int bytes;

    memset(&child, 0, sizeof(child));

    const char *childPath = CpGetChildProcessName("child");

    if (CpProcessCreate(&child, childPath) != 0) {
        fprintf(stderr, "Error: failed to create child process\n");
        return EXIT_FAILURE;
    }

    printf("Enter file name: ");
    if (!fgets(line, sizeof(line), stdin)) {
        fprintf(stderr, "Error: failed to read file name\n");
        CpProcessClose(&child);
        return EXIT_FAILURE;
    }
    TrimNewline(line);

    if (CpProcessWrite(&child, line, strlen(line)) < 0 ||
        CpProcessWrite(&child, "\n", 1) < 0) {
        fprintf(stderr, "Error: failed to send file name to child process\n");
        CpProcessClose(&child);
        return EXIT_FAILURE;
    }

    bytes = CpProcessRead(&child, childBuf, (int)sizeof(childBuf) - 1);
    if (bytes > 0) {
        if (bytes > (int)sizeof(childBuf) - 1) bytes = (int)sizeof(childBuf) - 1;
        childBuf[bytes] = '\0';
        printf("%s", childBuf);
        if (CpStringContains(childBuf, "Error:")) {
            CpProcessClose(&child);
            return EXIT_FAILURE;
        }
    }

    while (1) {
        printf("Enter string (empty string to exit): ");
        if (!fgets(line, sizeof(line), stdin)) break;
        TrimNewline(line);

        if (CpStringLength(line) == 0) {
```

```
            CpProcessWrite(&child, "\n", 1);
            break;
        }

        if (CpProcessWrite(&child, line, strlen(line)) < 0 ||
            CpProcessWrite(&child, "\n", 1) < 0) {
            fprintf(stderr, "Error: failed to send string to child process\n");
            break;
        }

        bytes = CpProcessRead(&child, childBuf, (int)sizeof(childBuf) - 1);
        if (bytes > 0) {
            if (bytes > (int)sizeof(childBuf) - 1) bytes = (int)sizeof(childBuf) - 1;
            childBuf[bytes] = '\0';
            printf("%s", childBuf);
        }
    }

    CpProcessClose(&child);
    printf("Parent process finished.\n");
    return EXIT_SUCCESS;
}
```

Листинг 1: parent.c - Родительский процесс, который запускает дочерний, передаёт ему имя файла и строки для записи

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "string_utils.h"

#define BUFFER_SIZE 1024

int main(void) {
    char buffer[BUFFER_SIZE];
    FILE* file = NULL;

    if (!fgets(buffer, sizeof(buffer), stdin)) {
        const char* err = "Error: failed to read file name\n";
        printf("%s", err); fflush(stdout);
        return EXIT_FAILURE;
    }
    TrimNewline(buffer);

    file = fopen(buffer, "w");
    if (file == NULL) {
        char err[BUFFER_SIZE];
        snprintf(err, sizeof(err), "Error: cannot open file '%s' for writing\n", buffer
            );
        printf("%s", err); fflush(stdout);
        return EXIT_FAILURE;
    }

    printf("File opened successfully\n");
    fflush(stdout);

    while (fgets(buffer, sizeof(buffer), stdin)) {
        TrimNewline(buffer);
        if (strlen(buffer) == 0) {
```

```
            break;
        }

        if (IsCapitalStart(buffer)) {
            fprintf(file, "%s\n", buffer);
            fflush(file);
            char ok[BUFFER_SIZE];
            snprintf(ok, sizeof(ok), "String written to file: '%s'\n", buffer);
            printf("%s", ok);
            fflush(stdout);
        } else {
            char err[BUFFER_SIZE];
            snprintf(err, sizeof(err), "Error: string must start with capital letter -
                '%s'\n", buffer);
            printf("%s", err);
            fflush(stdout);
        }
    }

    if (file) fclose(file);
    printf("Child process finished\n");
    fflush(stdout);
    return EXIT_SUCCESS;
}
```

Листинг 2: child.c - Дочерний процесс, который получает от родителя имя файла и записывает в него строки, начинающиеся с заглавной буквы

```
#include "stringutils.h"
#include <string.h>
#include <ctype.h>

void TrimNewline(char* str) {
    if (!str) return;
    size_t len = strlen(str);
    if (len > 0 && str[len - 1] == '\n') str[len - 1] = '\0';
}

int IsCapitalStart(const char* str) {
    if (!str || *str == '\0') return 0;
    return isupper((unsigned char)str[0]) != 0;
}

size_t CpStringLength(const char* str) {
    return strlen(str);
}

int CpStringContains(const char* str, const char* substr) {
    if (!str || !substr) return 0;
    return strstr(str, substr) != NULL;
}
```

Листинг 3: stringutils.c - Реализация функций для обработки строк и проверки условий форматирования

```
#include "crossplatform.h"
#include <stdlib.h>

#ifdef _WIN32
```

```c
#include <windows.h>

int CpProcessCreate(process_t* proc, const char* path) {
    if (!proc || !path) return -1;

    SECURITY_ATTRIBUTES sa;
    sa.nLength = sizeof(SECURITY_ATTRIBUTES);
    sa.bInheritHandle = TRUE;
    sa.lpSecurityDescriptor = NULL;

    HANDLE childStdoutRead = NULL;
    HANDLE childStdoutWrite = NULL;
    HANDLE childStdinRead = NULL;
    HANDLE childStdinWrite = NULL;

    if (!CreatePipe(&childStdoutRead, &childStdoutWrite, &sa, 0)) return -1;
    if (!CreatePipe(&childStdinRead, &childStdinWrite, &sa, 0)) {
        CloseHandle(childStdoutRead);
        CloseHandle(childStdoutWrite);
        return -1;
    }

    SetHandleInformation(childStdoutRead, HANDLE_FLAG_INHERIT, 0);
    SetHandleInformation(childStdinWrite, HANDLE_FLAG_INHERIT, 0);

    STARTUPINFOA si;
    PROCESS_INFORMATION pi;
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    si.hStdError = childStdoutWrite;
    si.hStdOutput = childStdoutWrite;
    si.hStdInput = childStdinRead;
    si.dwFlags |= STARTF_USESTDHANDLES;

    char cmdline[1024];
    strncpy(cmdline, path, sizeof(cmdline)-1);
    cmdline[sizeof(cmdline)-1] = '\0';

    if (!CreateProcessA(NULL, cmdline, NULL, NULL, TRUE, 0, NULL, NULL, &si, &pi)) {
        CloseHandle(childStdoutRead);
        CloseHandle(childStdoutWrite);
        CloseHandle(childStdinRead);
        CloseHandle(childStdinWrite);
        return -1;
    }

    CloseHandle(childStdoutWrite);
    CloseHandle(childStdinRead);

    proc->handle = pi.hProcess;
    proc->stdinWrite = childStdinWrite;
    proc->stdoutRead = childStdoutRead;

    CloseHandle(pi.hThread);
    return 0;
}

int CpProcessWrite(process_t* proc, const char* data, size_t size) {
```

```
63      if (!proc || !data) return -1;
64      DWORD written = 0;
65      if (!WriteFile(proc->stdinWrite, data, (DWORD)size, &written, NULL)) {
66          return -1;
67      }
68      return (int)written;
69  }
70
71  int CpProcessRead(process_t* proc, char* buffer, size_t size) {
72      if (!proc || !buffer || size == 0) return -1;
73      DWORD readBytes = 0;
74      if (!ReadFile(proc->stdoutRead, buffer, (DWORD)(size - 1), &readBytes, NULL)) {
75          return -1;
76      }
77      buffer[readBytes] = '\0';
78      return (int)readBytes;
79  }
80
81  int CpProcessClose(process_t* proc) {
82      if (!proc) return -1;
83      int exitCode = -1;
84      if (proc->stdinWrite) {
85          CloseHandle(proc->stdinWrite);
86          proc->stdinWrite = NULL;
87      }
88      if (proc->stdoutRead) {
89          CloseHandle(proc->stdoutRead);
90          proc->stdoutRead = NULL;
91      }
92      if (proc->handle) {
93          WaitForSingleObject(proc->handle, INFINITE);
94          DWORD code;
95          if (GetExitCodeProcess(proc->handle, &code)) {
96              exitCode = (int)code;
97          }
98          CloseHandle(proc->handle);
99          proc->handle = NULL;
100     }
101     return exitCode;
102 }
103
104 #else
105
106 #include <unistd.h>
107 #include <sys/wait.h>
108 #include <errno.h>
109
110 int CpProcessCreate(process_t* proc, const char* path) {
111     if (!proc || !path) return -1;
112     int inpipe[2];
113     int outpipe[2];
114
115     if (pipe(inpipe) == -1) return -1;
116     if (pipe(outpipe) == -1) {
117         close(inpipe[0]); close(inpipe[1]);
118         return -1;
119     }
120
```

```
    pid_t pid = fork();
    if (pid == -1) {
        close(inpipe[0]); close(inpipe[1]);
        close(outpipe[0]); close(outpipe[1]);
        return -1;
    }

    if (pid == 0) {
        dup2(inpipe[0], STDIN_FILENO);
        dup2(outpipe[1], STDOUT_FILENO);

        close(inpipe[0]); close(inpipe[1]);
        close(outpipe[0]); close(outpipe[1]);

        execl(path, path, (char*)NULL);
        _exit(127);
    } else {
        close(inpipe[0]);
        close(outpipe[1]);
        proc->pid = pid;
        proc->stdin_fd = inpipe[1];
        proc->stdout_fd = outpipe[0];
        return 0;
    }
}

int CpProcessWrite(process_t* proc, const char* data, size_t size) {
    if (!proc || !data) return -1;
    ssize_t n = write(proc->stdin_fd, data, size);
    if (n == -1) return -1;
    return (int)n;
}

int CpProcessRead(process_t* proc, char* buffer, size_t size) {
    if (!proc || !buffer || size == 0) return -1;
    ssize_t n = read(proc->stdout_fd, buffer, (ssize_t)(size - 1));
    if (n == -1) return -1;
    if (n == 0) {
        buffer[0] = '\0';
        return 0;
    }
    buffer[n] = '\0';
    return (int)n;
}

int CpProcessClose(process_t* proc) {
    if (!proc) return -1;
    int status = -1;
    if (proc->stdin_fd != -1) {
        close(proc->stdin_fd);
        proc->stdin_fd = -1;
    }
    if (proc->stdout_fd != -1) {
        close(proc->stdout_fd);
        proc->stdout_fd = -1;
    }
    if (proc->pid > 0) {
        waitpid(proc->pid, &status, 0);
```

```
        if (WIFEXITED(status)) return WEXITSTATUS(status);
    }
    return -1;
}

#endif
```

Листинг 4: crossplatform.c - Реализация создания дочернего процесса с перенаправлением ввода-вывода для Windows и Linux

```
#ifndef STRING_UTILS_H
#define STRING_UTILS_H

#include <stddef.h>

void TrimNewline(char* str);
int IsCapitalStart(const char* str);

size_t CpStringLength(const char* str);
int CpStringContains(const char* str, const char* substr);

#endif
```

Листинг 5: stringutils.h - Заголовок с утилитами для обработки строк: обрезка переводов строк и проверка заглавной буквы

```
#ifndef CROSS_PLATFORM_H
#define CROSS_PLATFORM_H

#include <stddef.h>
#include <stdio.h>
#include <string.h>

#ifdef _WIN32
#include <windows.h>
#else
#include <sys/types.h>
#endif

#ifdef _WIN32
typedef struct {
    HANDLE handle;
    HANDLE stdinWrite;
    HANDLE stdoutRead;
} process_t;
#else
typedef struct {
    pid_t pid;
    int stdin_fd;
    int stdout_fd;
} process_t;
#endif

int CpProcessCreate(process_t* proc, const char* path);
int CpProcessWrite(process_t* proc, const char* data, size_t size);
int CpProcessRead(process_t* proc, char* buffer, size_t size);
int CpProcessClose(process_t* proc);

size_t CpStringLength(const char* str);
```

```
34 int CpStringContains(const char* str, const char* substr);
35
36 static inline const char* CpGetChildProcessName(const char* baseName) {
37     (void)baseName;
38 #ifdef _WIN32
39     return "child.exe";
40 #else
41     return "./child";
42 #endif
43 }
44
45 #define CpWriteStdout(data, size) ((int)fwrite((data), 1, (size), stdout))
46 #define CpWriteStderr(data, size) ((int)fwrite((data), 1, (size), stderr))
47
48 #endif
```

Листинг 6: crossplatform.h - Кроссплатформенный заголовок с API для работы с процессами и строками в Windows/POSIX