

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, состоящую из двух процессов — родительского и дочернего, взаимодействующих через неименованные каналы.

Родительский процесс должен:

- Запрашивать у пользователя имя файла и передавать его дочернему процессу;
- Принимать от пользователя строки и передавать их дочернему процессу;
- Получать от дочернего процесса сообщения о результатах обработки строк и выводить их на экран.

Дочерний процесс должен:

- Получить от родительского процесса имя файла и открыть его для записи;
- Принимать строки от родительского процесса;
- Проверять, начинается ли каждая строка с заглавной буквы;
- Если строка начинается с заглавной буквы — записывать её в файл;
- Если строка не начинается с заглавной буквы — отправлять сообщение об ошибке родительскому процессу;
- Завершать работу после получения пустой строки.

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

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4
5  #include "cross_platform.h"
6  #include "string_utils.h"
7
8  #define BUFFER_SIZE 1024
9
10 int main(void) {
11     process_t child;
12     char line[BUFFER_SIZE];
13     char childBuf[BUFFER_SIZE];
14     int bytes;
15
16     memset(&child, 0, sizeof(child));
17
18     const char *childPath = CpGetChildProcessName("child");
19
20     if (CpProcessCreate(&child, childPath) != 0) {
21         fprintf(stderr, "Error: failed to create child process\n");
22         return EXIT_FAILURE;
23     }
24
25     printf("Enter file name: ");
26     if (!fgets(line, sizeof(line), stdin)) {
27         fprintf(stderr, "Error: failed to read file name\n");
28         CpProcessClose(&child);
29         return EXIT_FAILURE;
30     }
31     TrimNewline(line);
32
33     if (CpProcessWrite(&child, line, strlen(line)) < 0 ||
34         CpProcessWrite(&child, "\n", 1) < 0) {
35         fprintf(stderr, "Error: failed to send file name to child process\n");
36         CpProcessClose(&child);
37         return EXIT_FAILURE;
38     }
39
40     bytes = CpProcessRead(&child, childBuf, (int)sizeof(childBuf) - 1);
41     if (bytes > 0) {
42         if (bytes > (int)sizeof(childBuf) - 1) bytes = (int)sizeof(childBuf) - 1;
43         childBuf[bytes] = '\0';
44         printf("%s", childBuf);
45         if (CpStringContains(childBuf, "Error:")) {
46             CpProcessClose(&child);
47             return EXIT_FAILURE;
48         }
49     }
50
51     while (1) {
52         printf("Enter string (empty string to exit): ");
53         if (!fgets(line, sizeof(line), stdin)) break;
54         TrimNewline(line);
55
56         if (CpStringLength(line) == 0) {
```

```

57         CpProcessWrite(&child, "\n", 1);
58         break;
59     }
60
61     if (CpProcessWrite(&child, line, strlen(line)) < 0 ||
62         CpProcessWrite(&child, "\n", 1) < 0) {
63         fprintf(stderr, "Error: failed to send string to child process\n");
64         break;
65     }
66
67     bytes = CpProcessRead(&child, childBuf, (int)sizeof(childBuf) - 1);
68     if (bytes > 0) {
69         if (bytes > (int)sizeof(childBuf) - 1) bytes = (int)sizeof(childBuf) - 1;
70         childBuf[bytes] = '\0';
71         printf("%s", childBuf);
72     }
73 }
74
75 CpProcessClose(&child);
76 printf("Parent process finished.\n");
77 return EXIT_SUCCESS;
78 }

```

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

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "string_utils.h"
5
6  #define BUFFER_SIZE 1024
7
8  int main(void) {
9      char buffer[BUFFER_SIZE];
10     FILE* file = NULL;
11
12     if (!fgets(buffer, sizeof(buffer), stdin)) {
13         const char* err = "Error: failed to read file name\n";
14         printf("%s", err); fflush(stdout);
15         return EXIT_FAILURE;
16     }
17     TrimNewline(buffer);
18
19     file = fopen(buffer, "w");
20     if (file == NULL) {
21         char err[BUFFER_SIZE];
22         snprintf(err, sizeof(err), "Error: cannot open file '%s' for writing\n", buffer);
23         printf("%s", err); fflush(stdout);
24         return EXIT_FAILURE;
25     }
26
27     printf("File opened successfully\n");
28     fflush(stdout);
29
30     while (fgets(buffer, sizeof(buffer), stdin)) {
31         TrimNewline(buffer);
32         if (strlen(buffer) == 0) {

```



```

33         break;
34     }
35
36     if (IsCapitalStart(buffer)) {
37         fprintf(file, "%s\n", buffer);
38         fflush(file);
39         char ok[BUFFER_SIZE];
40         snprintf(ok, sizeof(ok), "String written to file: '%s'\n", buffer);
41         printf("%s", ok);
42         fflush(stdout);
43     } else {
44         char err[BUFFER_SIZE];
45         snprintf(err, sizeof(err), "Error: string must start with capital letter -
46             '%s'\n", buffer);
47         printf("%s", err);
48         fflush(stdout);
49     }
50
51     if (file) fclose(file);
52     printf("Child process finished\n");
53     fflush(stdout);
54     return EXIT_SUCCESS;
55 }

```

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

```

1  #include "stringutils.h"
2  #include <string.h>
3  #include <ctype.h>
4
5  void TrimNewline(char* str) {
6      if (!str) return;
7      size_t len = strlen(str);
8      if (len > 0 && str[len - 1] == '\n') str[len - 1] = '\0';
9  }
10
11 int IsCapitalStart(const char* str) {
12     if (!str || *str == '\0') return 0;
13     return isupper((unsigned char)str[0]) != 0;
14 }
15
16 size_t CpStringLength(const char* str) {
17     return strlen(str);
18 }
19
20 int CpStringContains(const char* str, const char* substr) {
21     if (!str || !substr) return 0;
22     return strstr(str, substr) != NULL;
23 }

```

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

```

1  #include "crossplatform.h"
2  #include <stdlib.h>
3
4  #ifdef _WIN32

```

```

5  #include <windows.h>
6
7  int CpProcessCreate(process_t* proc, const char* path) {
8      if (!proc || !path) return -1;
9
10     SECURITY_ATTRIBUTES sa;
11     sa.nLength = sizeof(SECURITY_ATTRIBUTES);
12     sa.bInheritHandle = TRUE;
13     sa.lpSecurityDescriptor = NULL;
14
15     HANDLE childStdoutRead = NULL;
16     HANDLE childStdoutWrite = NULL;
17     HANDLE childStdinRead = NULL;
18     HANDLE childStdinWrite = NULL;
19
20     if (!CreatePipe(&childStdoutRead, &childStdoutWrite, &sa, 0)) return -1;
21     if (!CreatePipe(&childStdinRead, &childStdinWrite, &sa, 0)) {
22         CloseHandle(childStdoutRead);
23         CloseHandle(childStdoutWrite);
24         return -1;
25     }
26
27     SetHandleInformation(childStdoutRead, HANDLE_FLAG_INHERIT, 0);
28     SetHandleInformation(childStdinWrite, HANDLE_FLAG_INHERIT, 0);
29
30     STARTUPINFOA si;
31     PROCESS_INFORMATION pi;
32     ZeroMemory(&si, sizeof(si));
33     si.cb = sizeof(si);
34     si.hStdError = childStdoutWrite;
35     si.hStdOutput = childStdoutWrite;
36     si.hStdInput = childStdinRead;
37     si.dwFlags |= STARTF_USESTDHANDLES;
38
39     char cmdline[1024];
40     strncpy(cmdline, path, sizeof(cmdline)-1);
41     cmdline[sizeof(cmdline)-1] = '\\0';
42
43     if (!CreateProcessA(NULL, cmdline, NULL, NULL, TRUE, 0, NULL, NULL, &si, &pi)) {
44         CloseHandle(childStdoutRead);
45         CloseHandle(childStdoutWrite);
46         CloseHandle(childStdinRead);
47         CloseHandle(childStdinWrite);
48         return -1;
49     }
50
51     CloseHandle(childStdoutWrite);
52     CloseHandle(childStdinRead);
53
54     proc->handle = pi.hProcess;
55     proc->stdinWrite = childStdinWrite;
56     proc->stdoutRead = childStdoutRead;
57
58     CloseHandle(pi.hThread);
59     return 0;
60 }
61
62 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

```

```

121     pid_t pid = fork();
122     if (pid == -1) {
123         close(inp[0]); close(inp[1]);
124         close(out[0]); close(out[1]);
125         return -1;
126     }
127
128     if (pid == 0) {
129         dup2(inp[0], STDIN_FILENO);
130         dup2(out[1], STDOUT_FILENO);
131
132         close(inp[0]); close(inp[1]);
133         close(out[0]); close(out[1]);
134
135         execl(path, path, (char*)NULL);
136         _exit(127);
137     } else {
138         close(inp[0]);
139         close(out[1]);
140         proc->pid = pid;
141         proc->stdin_fd = inp[1];
142         proc->stdout_fd = out[0];
143         return 0;
144     }
145 }
146
147 int CpProcessWrite(process_t* proc, const char* data, size_t size) {
148     if (!proc || !data) return -1;
149     ssize_t n = write(proc->stdin_fd, data, size);
150     if (n == -1) return -1;
151     return (int)n;
152 }
153
154 int CpProcessRead(process_t* proc, char* buffer, size_t size) {
155     if (!proc || !buffer || size == 0) return -1;
156     ssize_t n = read(proc->stdout_fd, buffer, (ssize_t)(size - 1));
157     if (n == -1) return -1;
158     if (n == 0) {
159         buffer[0] = '\0';
160         return 0;
161     }
162     buffer[n] = '\0';
163     return (int)n;
164 }
165
166 int CpProcessClose(process_t* proc) {
167     if (!proc) return -1;
168     int status = -1;
169     if (proc->stdin_fd != -1) {
170         close(proc->stdin_fd);
171         proc->stdin_fd = -1;
172     }
173     if (proc->stdout_fd != -1) {
174         close(proc->stdout_fd);
175         proc->stdout_fd = -1;
176     }
177     if (proc->pid > 0) {
178         waitpid(proc->pid, &status, 0);

```

```

179         if (WIFEXITED(status)) return WEXITSTATUS(status);
180     }
181     return -1;
182 }
183
184 #endif

```

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

```

1  #ifndef STRING_UTILS_H
2  #define STRING_UTILS_H
3
4  #include <stddef.h>
5
6  void TrimNewline(char* str);
7  int IsCapitalStart(const char* str);
8
9  size_t CpStringLength(const char* str);
10 int CpStringContains(const char* str, const char* substr);
11
12 #endif

```

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

```

1  #ifndef CROSS_PLATFORM_H
2  #define CROSS_PLATFORM_H
3
4  #include <stddef.h>
5  #include <stdio.h>
6  #include <string.h>
7
8  #ifdef _WIN32
9  #include <windows.h>
10 #else
11 #include <sys/types.h>
12 #endif
13
14 #ifdef _WIN32
15 typedef struct {
16     HANDLE handle;
17     HANDLE stdinWrite;
18     HANDLE stdoutRead;
19 } process_t;
20 #else
21 typedef struct {
22     pid_t pid;
23     int stdin_fd;
24     int stdout_fd;
25 } process_t;
26 #endif
27
28 int CpProcessCreate(process_t* proc, const char* path);
29 int CpProcessWrite(process_t* proc, const char* data, size_t size);
30 int CpProcessRead(process_t* proc, char* buffer, size_t size);
31 int CpProcessClose(process_t* proc);
32
33 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