Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №1 по курсу**

**«Операционные системы»**

Студент: Снытко Владислав Александрович

Группа: М8О–203Б–20

Вариант: 20

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2020.

**Постановка задачи**

## Цель работы:

## Приобретение практических навыков в:

## • Управление процессами в ОС

## • Обеспечение обмена данных между процессами посредством каналов

**Задание**:

Составить и отладить программу на языке Си, осуществляющую работу с процессами и взаимодействие между ними в одной из двух операционных систем. В результате работы программа (основной процесс) должен создать для решение задачи один или несколько дочерних процессов. Взаимодействие между процессами осуществляется через системные сигналы/события и/или каналы (pipe).

Необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы.

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.

Вариант 20) Правило фильтрации: строки длины больше 10 символов отправляются в pipe2, иначе в pipe1. Дочерние процессы инвертируют строки.

**Общие сведения о программе**

- Состав проекта и компоновка:

  - Исполняемые файлы: `main` (родитель), `child1`, `child2` (дети). Сборка выполняется CMake, артефакты размещаются в `build/bin`.

  - Исходники: `src/main.c` (родитель), `src/child1.c`, `src/child2.c` (дети), `src/reverse.c/.h` (общая функция реверса строки).

  - Родитель запускает детей через `exec` (см. ниже), поэтому запускать `main` следует из каталога с бинарниками, чтобы `./child1`/`./child2` были доступны по относительному пути.

- Используемые заголовочные файлы (основные):

  - В родителе: `stdio.h`, `stdlib.h`, `string.h`, `errno.h`, `sys/types.h`, `sys/wait.h`, `unistd.h`.

  - В детях: `stdio.h`, `stdlib.h`, `string.h`, `unistd.h`, а также проектный заголовок `reverse.h`.

- Ключевые системные вызовы/функции и их назначение:

  1) `fork()` — порождает новый процесс. В родителе возвращает PID (>0), в ребенке — 0, при ошибке — -1.

  2) `pipe(int fds[2])` — создает анонимный однонаправленный канал: `fds[0]` — конец чтения, `fds[1]` — конец записи.

  3) `dup2(int oldfd, int newfd)` — дублирует файловый дескриптор, переназначая стандартные потоки (например, STDIN) на конец pipe.

  4) `execl(const char \*path, const char \*arg0, ..., NULL)` — заменяет образ текущего процесса внешней программой (`./child1`, `./child2`), наследуя открытые дескрипторы.

  5) `write(int fd, const void \*buf, size\_t count)` — запись байт (родитель пишет введенные строки в соответствующий `pipe`).

  6) `close(int fd)` — закрытие неиспользуемых концов каналов в каждом процессе (критично для возникновения EOF у читающей стороны).

  7) `wait(int \*wstatus)` — ожидание завершения дочерних процессов, чтобы корректно освободить их ресурсы.

  8) `getline(char \*\*linep, size\_t \*np, FILE \*stream)` — удобная построчная функция чтения из `stdin`/pipe (не системный вызов, но ключевая для логики ввода). Возвращает длину строки или -1 при EOF/ошибке.

  9) `fopen`, `fputs`, `fflush`, `fclose` — стандартные функции ввода‑вывода в детях для записи результатов в файлы.

**Общий метод и алгоритм решения**.

Для реализации поставленной задачи выполнены следующие шаги:

1) Изучены принципы порождения процессов и переназначения потоков в POSIX: системные вызовы `fork`, `pipe`, `dup2`, а также загрузка внешних программ через `exec` и ожидание завершения детей через `wait`.

2) Спроектирована архитектура с одним родительским процессом и двумя дочерними: родитель создает два анонимных канала (для «коротких» и «длинных» строк) и маршрутизирует ввод в нужный канал; дети читают из своего stdin, обрабатывают данные и выводят результат.

3) Реализована общая библиотечная функция `reverse` (`src/reverse.c/.h`) для инвертирования строк с сохранением завершающего `\n`.

4) Реализованы программы-дети `child1.c` и `child2.c`, которые читают строки построчно, инвертируют их, печатают в stdout и записывают в указанный файл (передается через `argv[1]`).

5) Реализована программа-родитель `main.c`, которая: вводит имена выходных файлов, создает каналы, порождает детей, переназначает stdin детей на соответствующие концы каналов и маршрутизирует пользовательский ввод по правилу длины (≤10/ >10).

6) Настроена сборка с помощью CMake с раздельными целями (`main`, `child1`, `child2`, статическая библиотека `reverse`), результаты сборки — в `build/bin`.

7) Подготовлены скрипты тестирования (в каталоге `test/`) для быстрой проверки корректности маршрутизации и реверса строк, а также описаны шаги запуска под WSL в отчете.

**Основные файлы программы**

**main.c:**

|  |
| --- |
| #define \_GNU\_SOURCE  #include <errno.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <sys/types.h>  #include <sys/wait.h>  #include <unistd.h>  // Заменяет один завершающий '\n' в строке, если он есть.  static void check\_end\_in\_filename(char \*s) {      if (!s) return;      size\_t len = strlen(s);      if (len > 0 && s[len - 1] == '\n') s[len - 1] = '\0';  }  int main(      void) {      char \*filename1 = NULL,           \*filename2 = NULL;      size\_t c1 = 0, c2 = 0;      if (getline(&filename1, &c1, stdin) == -1)          goto end;      if (getline(&filename2, &c2, stdin) == -1)          goto end;      check\_end\_in\_filename(filename1);      check\_end\_in\_filename(filename2);      if (filename1[0] == '\0' || filename2[0] == '\0') {          fprintf(stderr, "empty filename\n");          goto end;      }      int p1[2], p2[2]; //int: [0] = чтение, [1] = запись; p1 для коротких строк, p2 для длинных      if (pipe(p1) == -1 || pipe(p2) == -1) {          perror("pipe error");          goto end;      }      pid\_t a = fork();      if (a == 0) { // ребенок          dup2(p1[0], STDIN\_FILENO); // dup2 создаёт ДУБЛИКАТ: теперь fd p1[0] и                                      // STDIN\_FILENO указывают на один и тот же                                      // открытый объект (конец трубы)          // После dup2 нам больше не нужен исходный номер p1[0] — достаточно          // стандартного ввода (fd=0), поэтому закрываем исходный дескриптор. Это          // уменьшает счётчик ссылок на объект ядра и предотвращает удержание          // лишнего конца трубы.          close(p1[0]); // Закрываем оригинальный дескриптор          close(p1[1]); // ребенок только читает          close(p2[0]); // Полностью не используем вторую трубу          close(p2[1]);          execl("./child1", "child1", filename1, (char \*)NULL);          perror("exec child1"); // Если exec вернулся — произошла ошибка загрузки          \_exit(1);      }      if (a < 0) {          perror("fork1");          goto end;      }      pid\_t b = fork();      if (b == 0) {          dup2(p2[0], STDIN\_FILENO);          close(p2[0]); // Закрываем оригинал дескриптора чтения          close(p2[1]);          close(p1[0]);          close(p1[1]);          execl("./child2", "child2", filename2, (char \*)NULL);          perror("exec child2");          exit(1);      }      if (b < 0) {          perror("fork2");          goto end;      }      close(p1[0]);      close(p2[0]);      char \*line = NULL;      size\_t cap = 0;      ssize\_t n;      while ((n = getline(&line, &cap, stdin)) != -1) { // Читаем до тех пор, пока пользователь не завершит ввод (Ctrl+D / EOF), /n          if (n == 1) break;          size\_t pure\_len = (line[n - 1] == '\n') ? (size\_t)n - 1 : (size\_t)n;          int out\_fd = (pure\_len > 10) ? p2[1] : p1[1];          if (write(out\_fd, line, (size\_t)n) == -1) {              perror("write");              break;          }      }      close(p1[1]);      close(p2[1]);      if (wait(NULL) == -1)          perror("wait");      if (wait(NULL) == -1)          perror("wait");      free(line);  end:      free(filename1);      free(filename2);  } |

**child1:**

|  |
| --- |
| #define \_GNU\_SOURCE  #include <fcntl.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #include "reverse.h"  int main(int argc,           char \*argv[]) { // Точка входа программы; argc — число аргументов,      // argv — массив строк аргументов      if (argc < 2) { // Проверяем, передан ли хотя бы один аргумент (имя файла          fprintf(stderr, "need file\n");          return 1;      }      FILE \*out = fopen(argv[1], "w"); // "w" = create/truncate/write      if (!out) {          perror("fopen");          return 2;      }      char \*line = NULL;      size\_t cap = 0;      ssize\_t len;      while ((len = getline(&line, &cap, stdin)) != -1) {          reverse(line);          fputs(line, out); // использует буфер stdio          fflush(out); // опционально, чтобы данные сразу записывались      }      free(line);      fclose(out);  } |

**child2.c:**

|  |
| --- |
| #define \_GNU\_SOURCE  #include <fcntl.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #include "reverse.h"  int main(int argc, char \*argv[]) {      if (argc < 2) {  // Проверяем, передан ли хотя бы один аргумент (имя файла          fprintf(stderr, "need file\n");          return 1;      }      FILE \*out = fopen(argv[1], "w");  // "w" = create/truncate/write      if (!out) {          perror("fopen");          return 2;      }      char \*line = NULL;      size\_t cap = 0;      while (getline(&line, &cap, stdin) != -1) {          reverse(line);          fputs(line, out);          fflush(out);      }      free(line);      fclose(out);  } |

**reverse.c:**

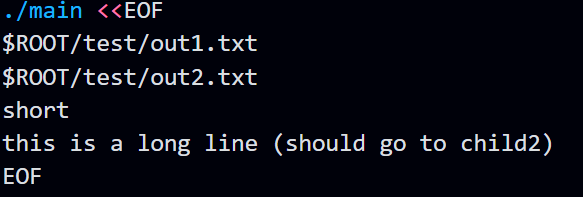
|  |
| --- |
| #include <string.h>  void reverse(char \*s) {      if (!s) return;      size\_t len = strlen(s);      int has\_nl = 0;      if (len && s[len - 1] == '\n') {          has\_nl = 1;          s[--len] = '\0';      }      for (size\_t i = 0; i < len / 2; ++i) {          char tmp = s[i];          s[i] = s[len - 1 - i];          s[len - 1 - i] = tmp;      }      if (has\_nl) {          s[len] = '\n';          s[len + 1] = '\0';      }  } |

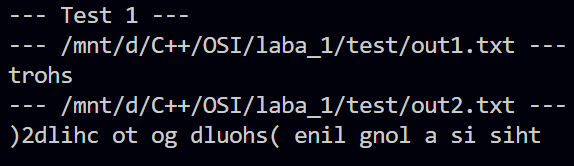
**reverse.h:**

|  |
| --- |
| #ifndef REVERSE\_H  #define REVERSE\_H  #include <stddef.h>  void reverse(char \*s);  #endif |

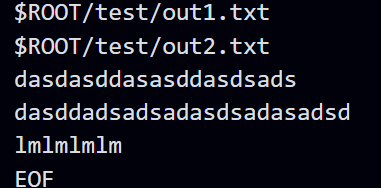
**Пример работы**

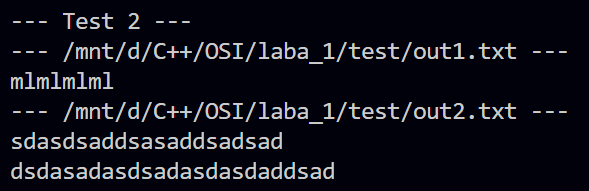
Test1:



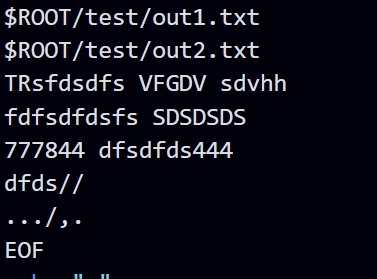


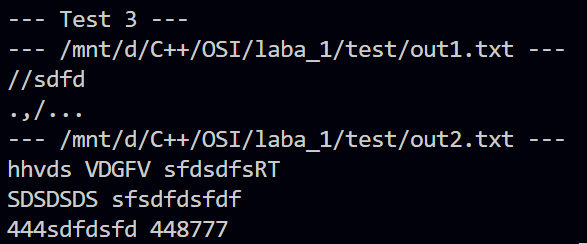
Test2:





Test3:



****

**Вывод**

Требования задания выполнены: основной процесс порождает дочерние, обмен организован через анонимные каналы (pipe), а ошибки системных вызовов обрабатываются. Отработаны ключевые механизмы `fork/pipe/dup2/exec/wait`, реализована маршрутизация строк по длине и реверс с выводом в stdout и записью в файлы; корректность подтверждена сборкой и прогонами тестовых скриптов в WSL.