Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 "Компьютерные науки и прикладная математика"

Кафедра 806 вычислительной математики и программирования

**Лабораторная работа №1 по курсу**

**«Операционные системы»**

**СИСТЕМНЫЕ ВЫЗОВЫ**

Студент: Парастаев Сослан Геннадьевич

Группа: М8О–203БВ–25

Вариант: 8

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_5 \_\_\_\_\_

Дата: \_10.10.2025

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва 2025

**Постановка задачи**

## Цель работы

Приобретение практических навыков в:

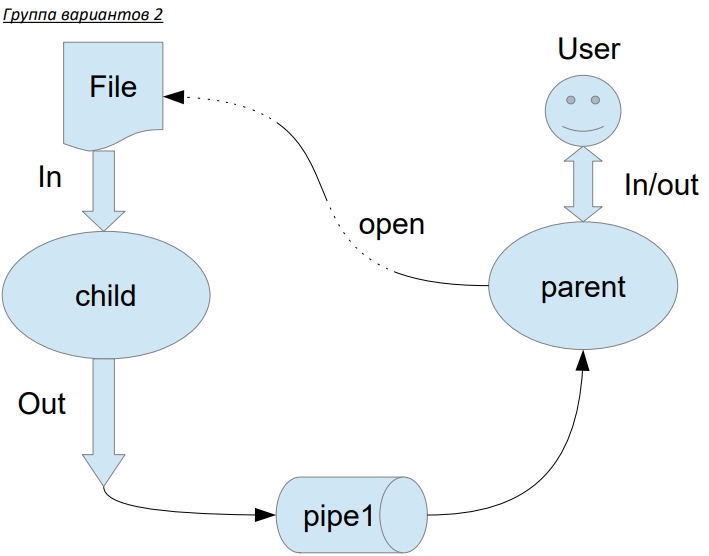
* Управление процессами в ОС
* Обеспечение обмена данных между процессами посредством каналов

## 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.

**Общие сведения о программе**

****

Родительский процесс создает дочерний процесс. Первой строчкой пользователь в консоль родительского процесса вводит имя файла, которое будет использовано для открытия файла с таким именем на чтение. Стандартный поток ввода дочернего процесса переопределяется открытым файлом. Дочерний процесс читает команды из стандартного потока ввода. Стандартный поток вывода дочернего процесса перенаправляется в pipe1. Родительский процесс читает из pipe1 и прочитанное выводит в свой стандартный поток вывода. Родительский и дочерний процесс должны быть представлены разными программами.

В файле записаны команды вида:«число число число». Дочерний процесс производит деление первого числа команда, на последующие числа в команде, а результат выводит в стандартный поток вывода. Если происходит деление на 0, то тогда дочерний и родительский процесс завершают свою работу. Проверка деления на 0 должна осуществляться на стороне дочернего процесса. Числа имеют тип int. Количество чисел может быть произвольным.

В программе используются следующие системные вызовы:

1. **pipe** – Это механизм в операционной системе, который позволяет двум процессам общаться между собой, передавая данные в одном направлении.
2. **open** – Открывает файл и возвращает файловый дескриптор, который используется для последующих операций (чтения, записи). Позволяет указать режим доступа (только чтение, запись и т.д.).
3. **close** – Закрывает открытый файловый дескриптор, освобождая ресурсы системы и делая дескриптор доступным для повторного использования.
4. **write** – Записывает данные из буфера приложения в открытый файл или другой объект, используя его файловый дескриптор.
5. **read** – Читает данные из открытого файлового дескриптора в буфер приложения.
6. **dup2** – Создает копию существующего файлового дескриптора, но с указанным новым номером, что часто используется для перенаправления ввода/вывода (например, в pipe).
7. **waitpid –** Приостанавливает выполнение текущего процесса до тех пор, пока указанный дочерний процесс не завершится или не изменит свое состояние.

**Общий метод и алгоритм решения**.

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы вышенаписанных системных вызовов
2. Подготовить структуру репозитория
3. Объявляем переменные:

* pid\_t pid – для получения id процесса, в котором находимся (либо parent, либо child
* int my\_pipe[2] – будет хранить в 0 индексе дескриптор для чтения(для parent), в 1 индексе дескриптор для записи(для child)
* int status – код, который возвращает child процесс после своего завершения для определения его корректного выполнения
* char filename, path\_to\_filename – введенное имя файла(без расширения) и путь до него(добавление “../test/%s.txt” перед именем файла)

1. Алгоритм описан в комментариях в коде

**Основные файлы программы**

**main.c:**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <sys/wait.h>  #include <fcntl.h>  #include <string.h>  int main(){      pid\_t pid;      int my\_pipe[2]; // pipe содержит 2 дескриптора: pipe[0] - для чтения, pipe[1] - для записи      char filename[50];      int status; // код, который возвращает child процесс после своего завершения для определения его корректного выполнения      char path\_to\_filename[300];      // Создаём pipe, который будет связывать child и parent, где child будет записывать, а parent считывать то, что записывает child      // pipe должен быть направлен в одну сторону, так что для child мы закроем чтение, а для parent закроем запись. Когда закончим чтение/запись, то закроем запись/чтение, чтобы закрыть pipe,      // чтобы был знак, что закончили и процессы не ждали бесконечно      if (pipe(my\_pipe) == -1){          perror("pipe");          exit(1);      }      // Читаем имя файла от пользователя      write(STDOUT\_FILENO, "Введите имя файла(без расширения):\n", 63);      int bytes\_read = read(STDIN\_FILENO, filename, sizeof(filename) - 1);      if (bytes\_read > 0) {          filename[bytes\_read] = '\0';          // Убираем символ новой строки          char \*newline = strchr(filename, '\n');          if (newline) \*newline = '\0';      } else {          perror("read");          exit(1);      }      // Формируем полный путь к файлу      strcpy(path\_to\_filename, "../test/");      strcat(path\_to\_filename, filename);      strcat(path\_to\_filename, ".txt");      // printf("%s", path\_to\_filename);      // Создаём файловый дескриптор      int file\_dt = open(path\_to\_filename, O\_RDONLY);      if (file\_dt == -1){          perror("open");          exit(1);      }      // Создаем дочерний процесс      pid = fork();      if (pid==0){          // child, записывает, значит закрываем чтение в pipe(то есть parent через pipe, не сможет записать что-то в child)          close(my\_pipe[0]);          // "Стандартный поток ввода дочернего процесса переопределяется открытым файлом", так что через dup2() переназначим дескриптор файла на дескриптор ввода для child,          // чтобы для child чтение файла было как stdin          if(dup2(file\_dt, STDIN\_FILENO) == -1) {perror("dup2"); exit(1);}          close(file\_dt);          // Переназначаем дескриптор вывода на запись в parent, чтобы для child запись в pipe(передача данных parent-у) было как stdout          if(dup2(my\_pipe[1], STDOUT\_FILENO) == -1) {perror("dup2"); exit(1);};          close(my\_pipe[1]);          // "Запускаем(замещаем)" дочерний процесс нашим child.          // Первый аргумент - путь до программы, следующие аргументы передаются в main(int argc, char \*argv[]), всегда в конце пишется NULL          execl("./child", "child", NULL);          perror("execl");          exit(1);          }      else{          // parent, читает, значит закрываем запись в pipe(то есть child через pipe, не сможет читать что-то с parent)          close(my\_pipe[1]);          int bytes\_read;          char buffer[100];          while ((bytes\_read = read(my\_pipe[0], buffer, sizeof(buffer))) > 0){              buffer[bytes\_read] = '\0'; // Добавляем нулевой терминатор              write(STDOUT\_FILENO, buffer, bytes\_read);          }          close(my\_pipe[0]);          // Ждем завершения дочернего процесса и получаем его статус          // (для обработки исключения - деления на 0, так как child вернет 1, а её надо передать в вывод main, иначе main вернет 0, будто ошибки не было)            waitpid(pid, &status, 0);          /\*          pid - ID дочернего процесса, который мы получили от fork()          &status - указатель на переменную, куда будет записан статус завершения          0 - флаги ожидания (0 означает обычное ожидание)          Функция блокирует выполнение родительского процесса, пока дочерний не завершится          \*/            // Проверяем, как завершился дочерний процесс          /\*          WIFEXITED(status) - макрос, который возвращает true, если процесс завершился нормально          Если true - процесс вызвал exit() или return из main()\*/          if (WIFEXITED(status)) {              int child\_exit\_code = WEXITSTATUS(status); // Извлекаем код возврата дочернего процесса              if (child\_exit\_code != 0) {                  return child\_exit\_code; // Возвращаем код ошибки дочернего процесса              }      }      return 0;  }  } |

**child.c:**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <sys/wait.h>  #include <fcntl.h>  #include <string.h>  int main(int argc, char\* ARGV[]){      int a, b, c;      double d, e;      char input\_buffer[100];      int bytes\_read;      bytes\_read = read(STDIN\_FILENO, input\_buffer, sizeof(input\_buffer) - 1);      if (bytes\_read > 0) {          input\_buffer[bytes\_read] = '\0';            // Замена scanf на sscanf для чтения из буфера          if (sscanf(input\_buffer, "%d %d %d", &a, &b, &c) == 3){              // Замена printf на write и snprintf              char message[100];              int len = snprintf(message, sizeof(message), "Даны числа: %d %d %d \n", a, b, c);              write(STDOUT\_FILENO, message, len);                if(b == 0 || c == 0){                  char error\_msg[] = "Ошибка: деление на ноль\n";                  write(STDOUT\_FILENO, error\_msg, sizeof(error\_msg) - 1);                  exit(1);              }              d = (double)a / b;              e = (double)a / c;                char result[100];              len = snprintf(result, sizeof(result), "Результат: %.3f %.3f \n", d, e);              write(STDOUT\_FILENO, result, len);          }          else{              char error\_msg[] = "Ошибка чтения чисел\n";              write(STDOUT\_FILENO, error\_msg, sizeof(error\_msg) - 1);              exit(1);          }      }      fflush(stdout);      return 0;  } |

**run.sh:**

|  |
| --- |
| #!/bin/bash  MAIN\_PROGRAM="./main"  TEST\_DIR="../test"  for test\_file in "$TEST\_DIR"/test\*.txt; do  # Пропускаем, если нет файлов, соответствующих шаблону  [ -f "$test\_file" ] || continue    echo "=== Запуск теста с файлом: $test\_file ==="    # Извлекаем имя файла без расширения .txt  filename=$(basename "$test\_file" .txt)    "$MAIN\_PROGRAM" <<< "$filename"    exit\_code=$?  if [ $exit\_code -eq 0 ]; then  echo "✓ Тест $test\_file завершен успешно"  else  echo "✗ Тест $test\_file завершен с ошибкой (код: $exit\_code)"  fi  echo  done  echo "Все тесты завершены" |

Ссылка на github репозиторий:

<https://github.com/SoslanPar/OS_Lab_1>

**Пример работы**

**./main**

**Введите имя файла(без расширения):**

**test1**

**Даны числа: 10 2 5**

**Результат: 5.000 2.000**

**/run.sh**

**=== Запуск теста с файлом: ../test/test1.txt ===**

**Введите имя файла(без расширения):**

**Даны числа: 10 2 5**

**Результат: 5.000 2.000**

**✓ Тест ../test/test1.txt завершен успешно**

**=== Запуск теста с файлом: ../test/test2.txt ===**

**Введите имя файла(без расширения):**

**Даны числа: 50 10 2**

**Результат: 5.000 25.000**

**✓ Тест ../test/test2.txt завершен успешно**

**=== Запуск теста с файлом: ../test/test3.txt ===**

**Введите имя файла(без расширения):**

**Даны числа: 192 3 16**

**Результат: 64.000 12.000**

**✓ Тест ../test/test3.txt завершен успешно**

**=== Запуск теста с файлом: ../test/test4.txt ===**

**Введите имя файла(без расширения):**

**Даны числа: 18 9 3**

**Результат: 2.000 6.000**

**✓ Тест ../test/test4.txt завершен успешно**

**=== Запуск теста с файлом: ../test/test5.txt ===**

**Введите имя файла(без расширения):**

**Даны числа: 8 1 4**

**Результат: 8.000 2.000**

**✓ Тест ../test/test5.txt завершен успешно**

**=== Запуск теста с файлом: ../test/test6.txt ===**

**Введите имя файла(без расширения):**

**Даны числа: 10 3 6**

**=== Запуск теста с файлом: ../test/test7.txt ===**

**Введите имя файла(без расширения):**

**Даны числа: 10 0 5**

**Ошибка: деление на ноль**

**✗ Тест ../test/test7.txt завершен с ошибкой (код: 1)**

**Все тесты завершены**

**Вывод**

В ходе выполнения лабораторной работы были успешно приобретены практические навыки работы с процессами в операционной системе и механизмами межпроцессного взаимодействия. Реализованная программа состоит из двух отдельных процессов (родительского и дочернего), взаимодействующих через pipe.

Основные достижения:

* Организовано корректное взаимодействие между процессами с использованием pipe, dup2, read и write.
* Реализовано перенаправление стандартного ввода дочернего процесса на чтение из файла.
* Обеспечен вывод результатов дочернего процесса в канал с последующим чтением и отображением родительским процессом.
* Добавлена обработка ошибок (деление на ноль, ошибки чтения файла, системные вызовы).
* Реализована корректная передача кода возврата дочернего процесса при ошибках.

Результаты тестирования подтвердили:

* Корректную обработку валидных данных.
* Отлавливание и обработку ошибок (деление на ноль).
* Стабильную работу программы на различных наборах данных.

Программа демонстрирует практическое понимание работы с процессами, pipe и перенаправлением ввода-вывода в UNIX-подобных системах.