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»

Группа: М8О-213БВ-24

Студент: Гриханов А.Е.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 03.10.25

Москва, 2025

Постановка задачи

Вариант 6.

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<endline>». Дочерний процесс считает их сумму и выводит результат в стандартный поток вывода. Числа имеют тип int. Количество чисел может быть произвольным.

Общий метод и алгоритм решения

Использованные системные вызовы:

- pid_t fork(void); – создает дочерний процесс.
- int pipe(int *fd); – создает однонаправленный канал данных и возвращает два файловых дескриптора: для чтения и для записи.
- int execv(const char *path, char *const argv) - заменяет образ текущего процесса новым процессом, загруженным из исполняемого файла.
- pid_t waitpid(pid_t pid, int *status, int options) - приостанавливает родительский процесс до тех пор, пока дочерний не завершится.
- void _exit(int status) – немедленно завершает процесс, не выполняя стандартных процедур очистки.
- int open(const char *path, int oflag) – открывает файл и возвращает файловый дескриптор для него.
- ssize_t read(int fd, void *buf, size_t count) – читает данные из файлового дескриптора в буффер.
- ssize_t write(int fd, const void *buf, size_t count) – записывает данные из буфера в файловый дескриптор.
- int close(int fd) – закрывает файловый дескриптор, освобождая его.
- int dup2(int oldfd, int newfd) – дублирует файловый дескриптор, позволяя перенаправлять потоки ввода-вывода.

В рамках задания было создано многопроцессное приложение для демонстрации взаимодействия процессов через неименованные каналы (pipe) с использованием системных вызовов POSIX.

Программа состоит из двух частей: родительского процесса и дочернего процесса.

1. Родительский процесс сначала запрашивает у пользователя имя файла с данными и создает канал (pipe) для обмена информацией.

2. С помощью системного вызова fork() создается дочерний процесс.

3. В дочернем процессе стандартный поток (stdin) связывается с файлом пользователя, а стандартный вывод (stdout) с записывающим концом канала. Это делается с помощью вызова dup2().

4. После этого дочерний процесс, используя execv(), замещает свой код на код отдельной программы-вычислителя. Эта программа теперь просто читает данные из своего stdin (не зная что это файл) и записывает результат в stdout (не зная что это канал).

5. В это время родительский процесс ожидает данные на читающем конце канала. Получив результаты от дочернего процесса, он выводит их в консоль и ожидает завершения работы ‘потомка’.

Таким образом, была реализована классическая модель взаимодействия, где одна программа управляет потоком данных, а другая выполняет основную вычислительную работую

Код программы

child.c

```
#include <stddef.h>
#include <stdio.h>
#include <unistd.h>


static int my_atoi(const char *str) {
  int result = 0;
  int sign = 1;
  int i = 0;
  if (str[0] == '-') {
    sign = -1;
    i++;
  }
  while (str[i] >= '0' && str[i] <= '9') {
    result = result * 10 + (str[i] - '0');
    i++;
  }
  return result * sign;
}
```

```
static void process_line(char *line_buf) {
  char num1_str[20], num2_str[20];
  size_t i = 0, j = 0;

  while (line_buf[i] != ' ' && line_buf[i] != '\0') {
    if (j < sizeof(num1_str) - 1)
      num1_str[j++] = line_buf[i];
    i++;
  }
  num1_str[j] = '\0';
  if (line_buf[i] == ' ')
    i++;

  j = 0;
  while (line_buf[i] != '\0') {
    if (j < sizeof(num2_str) - 1)
      num2_str[j++] = line_buf[i];
    i++;
  }
  num2_str[j] = '\0';

  int sum = my_atoi(num1_str) + my_atoi(num2_str);
  char result_buf[32];
  int len = snprintf(result_buf, sizeof(result_buf), "%d\n", sum);
  write(STDOUT_FILENO, result_buf, len);
}

int main(void) {
  char line_buf[128];
  size_t line_pos = 0;
```

```
  char current_char;

  while (read(STDIN_FILENO, &current_char, 1) > 0) {
    if (current_char != '\n') {
      if (line_pos < sizeof(line_buf) - 1) {
        line_buf[line_pos++] = current_char;
      }
    } else {
      if (line_pos > 0) {
        line_buf[line_pos] = '\0';
        process_line(line_buf);
      }
      line_pos = 0;
    }
  }

  if (line_pos > 0) {
    line_buf[line_pos] = '\0';
    process_line(line_buf);
  }

  return 0;
}
```

parent.c

```
#include <fcntl.h>
#include <sys/wait.h>
#include <unistd.h>

static void report_error_and_exit(const char *msg) {
```

```
  int i = 0;
  while (msg[i])
    i++;
  write(STDERR_FILENO, msg, i);
  write(STDERR_FILENO, "\n", 1);
  _exit(1);
}

int main(void) {
  const char *prompt = "Enter filename: ";
  write(STDERR_FILENO, prompt, 16);

  char filename[256];
  ssize_t bytes_read = read(STDIN_FILENO, filename, sizeof(filename) - 1);
  if (bytes_read <= 0) {
    report_error_and_exit("Parent: Failed to read filename.");
  }

  if (filename[bytes_read - 1] == '\n') {
    filename[bytes_read - 1] = '\0';
  } else {
    filename[bytes_read] = '\0';
  }

  int pipefd[2];
  if (pipe(pipefd) == -1) {
    report_error_and_exit("Parent: Failed to create pipe.");
  }

  pid_t pid = fork();
```

```
if (pid == -1) {
  report_error_and_exit("Parent: Fork failed.");
}

if (pid == 0) {

  int file_fd = open(filename, O_RDONLY);
  if (file_fd == -1) {
    report_error_and_exit("Child: Failed to open file.");
  }

  if (dup2(file_fd, STDIN_FILENO) == -1) {
    report_error_and_exit("Child: dup2 for stdin failed.");
  }

  if (dup2(pipefd[1], STDOUT_FILENO) == -1) {
    report_error_and_exit("Child: dup2 for stdout failed.");
  }

  close(file_fd);
  close(pipefd[0]);
  close(pipefd[1]);

  char *argv[] = {"./child", NULL};
  execv(argv[0], argv);

  report_error_and_exit("Child: execv failed.");

} else {
  close(pipefd[1]);
```

```
    char buffer[256];
    ssize_t count;
    while ((count = read(pipefd[0], buffer, sizeof(buffer))) > 0) {
      write(STDOUT_FILENO, buffer, count);
    }
    close(pipefd[0]);

    waitpid(pid, NULL, 0);
  }

  return 0;
}
```

Протокол работы программы

```
└─$ echo -e "10 25\n100 1\n5 3" > commands.txt


┌─agrikhanov@MacBook-Air-Artem-4 ~/Desktop/labs25-26-3sem/oci/lr1 ‹main●›
└─$ ./parent
Enter filename: commands.txt
35
101
8

└─$ echo -e "-10 5\n-5 -5\n10 0" > advanced.txt
┌─agrikhanov@MacBook-Air-Artem-4 ~/Desktop/labs25-26-3sem/oci/lr1 ‹main●›
└─$ ./parent
Enter filename: advanced.txt
-5
-10
10

┌─agrikhanov@MacBook-Air-Artem-4 ~/Desktop/labs25-26-3sem/oci/lr1 ‹main●›
└─$ touch empty.txt
┌─agrikhanov@MacBook-Air-Artem-4 ~/Desktop/labs25-26-3sem/oci/lr1 ‹main●›
└─$ ./parent
Enter filename: empty.txt
┌─agrikhanov@MacBook-Air-Artem-4 ~/Desktop/labs25-26-3sem/oci/lr1 ‹main●›
└─$

└─$ ./parent
Enter filename: fshdfhdfh
Child: Failed to open file.
```

Вывод

В ходе лабораторной работы я освоил ключевые системные вызовы, такие как fork(), execv(), pipe(), для создания многопроцессного приложения. Также научился организовывать взаимодействие между родительским и дочерним процессами, используя каналы и перенаправление потоков ввода-вывода для эффективного распределения задач.