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-211БВ-24

Студент: Бабицкий И.А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 02.10.25

Москва, 2025

Постановка задачи

Вариант 22.

Родительский процесс создает два дочерних процесса. Первой строкой пользователь в консоль родительского процесса вводит имя файла, которое будет использовано для открытия `File` с таким именем на запись для `child1`. Аналогично для второй строки и процесса `child2`. Родительский и дочерний процесс должны быть представлены разными программами.

Родительский процесс принимает от пользователя строки произвольной длины и пересылает их в `pipe1` или в `pipe2` в зависимости от правила фильтрации. Процесс `child1` и `child2` производят работу над строками. Процессы пишут результаты своей работы в стандартный вывод.

Правило фильтрации: с вероятностью 80% строки отправляются в `pipe1`, иначе в `pipe2`. Дочерние процессы инвертируют строки.

Общий метод и алгоритм решения

Использованные системные вызовы:

- `pid_t fork()` — создание дочернего процесса;
- `int execv(const char *filename, char *const argv[])` — замена образа памяти процесса;
- `void exit(int status)` — завершения выполнения процесса и возвращение статуса;
- `int pipe(int pipefd[2])` — создание неименованного канала для передачи данных между процессами;
- `int dup2(int oldfd, int newfd)` — переназначение файлового дескриптора;
- `int open(const char *pathname, int flags, mode_t mode)` — открытие\создание файла;
- `int close(int fd)` — закрыть файл;
- `pid_t waitpid(pid_t pid)` — ожидание завершения дочернего процесса.

Принцип работы:

1. Создаём два неименнованных канала для передачи данных.
2. Считываем имена дочерних файлов и открываем сами файлы.
3. Создаём два дочерних процесса, в каждом из которых заменяем поток ввода на `stdin`.
4. Определение случайным образом файла, в который будут записаны входные данные.
5. Разворот и запись данных в выбранный файл.
6. Закрытие файлов и завершение работы дочерних процессов.

Код программы

parent.c

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>
```

```c
#include <time.h>
#include <stdbool.h>
#include "../include/errors.h"
#include "../include/colors.h"

#define PIPE_ERROR -1
#define IS_CHILD 0
#define FORK_ERROR -1
#define EXECV_ERROR -1
#define FILE_NAME_ERROR -1
#define FILE_NAME_SIZE 256
#define MESSAGE_SIZE 128
#define BUFFER_SIZE 1024
#define AFTER_FORK_ERROR -1
#define EMPTY_BUFFER -1


int getFileName(char* fileName, const int fileNumber) {
  char message[MESSAGE_SIZE];
  int messageLen = sprintf(
    message,
    COLOR_BOLD_CYAN "Enter name of the %d child file: " COLOR_WHITE,
    fileNumber
  );
  write(STDOUT_FILENO, message, messageLen);

  ssize_t fileNameLen = read(STDIN_FILENO, fileName, sizeof(fileName) - 1);
  if (fileNameLen < 1) {
    errorInvalidFileName();
    return FILE_NAME_ERROR;
  }
  fileName[fileNameLen - 1] = '\0';
  return 0;
}


int afterFork(const int pipe[], const int otherPipe[], char *fileName) {
  dup2(pipe[0], STDIN_FILENO);
  close(pipe[0]);
  close(pipe[1]);
  close(otherPipe[0]);
  close(otherPipe[1]);

  char *const args[] = {"child", fileName, NULL};
  if (execv("./child", args) == EXECV_ERROR) {
    errorExecv();
    return -1;
  }
```

```
  return 0;
}


int readData(char buffer[], ssize_t *bufferLen) {
  *bufferLen = read(STDIN_FILENO, buffer, BUFFER_SIZE);

  if (*bufferLen <= 0) {
    return -1;
  }

  if (buffer[0] == '\n') {
    return -1;
  }

  return 0;
}


void writeData(const char buffer[], const ssize_t bufferLen, const int pipe1[],
const int pipe2[]) {
  if ((rand() % 100) < 80) {
    write(pipe1[1], buffer, bufferLen);
    return;
  }

  write(pipe2[1], buffer, bufferLen);
}


int main() {
  int pipe1[2], pipe2[2];
  if (pipe(pipe1) == PIPE_ERROR || pipe(pipe2) == PIPE_ERROR) {
    errorPipe();
    return 0;
  }

  char fileName1[FILE_NAME_SIZE], fileName2[FILE_NAME_SIZE];
  if (getFileName(fileName1, 1) == FILE_NAME_ERROR) {
    return 0;
  }
  if (getFileName(fileName2, 2) == FILE_NAME_ERROR) {
    return 0;
  }

  pid_t child1 = fork();
  switch (child1) {
    case FORK_ERROR: {
      errorFork();
```

```
      return 0;
    }
    case IS_CHILD: {
      if (afterFork(pipe1, pipe2, fileName1) == AFTER_FORK_ERROR) {
        return 0;
      }
    }
  }

  pid_t child2 = fork();
  switch (child2) {
    case FORK_ERROR: {
      errorFork();
      return 0;
    }
    case IS_CHILD: {
      if (afterFork(pipe2, pipe1, fileName2) == AFTER_FORK_ERROR) {
        return 0;
      }
    }
  }

  close(pipe1[0]);
  close(pipe2[0]);

  char buffer[BUFFER_SIZE];
  ssize_t bufferLen;
  srand(time(NULL));
  while (true) {
    if (readData(buffer, &bufferLen) == EMPTY_BUFFER) {
      break;
    }

    writeData(buffer, bufferLen, pipe1, pipe2);
  }

  close(pipe1[1]);
  close(pipe2[1]);

  wait(NULL);
  wait(NULL);

  return 0;
}
```

child.c

```
#include <stdint.h>
```

```c
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include "../include/errors.h"

#define BUFFER_SIZE 1024
#define OPEN_ERROR -1


void reverse(char *str, ssize_t len) {
  if (len > 0 && str[len-1] == '\n') {
    len--;
  }

  for (ssize_t i = 0; i < len / 2; i++) {
    char tmp = str[i];
    str[i] = str[len - 1 - i];
    str[len - 1 - i] = tmp;
  }
}


int main(int argc, char *argv[]) {
  if (argc < 2) {
    errorInvalidCountOfAgruments();
    return 0;
  }

  int file = open(argv[1], O_WRONLY | O_CREAT | O_TRUNC, 0600);
  if (file == OPEN_ERROR) {
    errorOpenFile();
    return 0;
  }

  char buffer[BUFFER_SIZE];
  ssize_t bufferLen;
  while ((bufferLen = read(STDIN_FILENO, buffer, sizeof(buffer))) > 0) {
    reverse(buffer, bufferLen);
    write(file, buffer, bufferLen);
    write(STDOUT_FILENO, buffer, bufferLen);
  }

  close(file);
  return 0;
}
```

errors.c

```
#include "../include/errors.h"
#include "../include/colors.h"
#include <stdlib.h>
#include <string.h>
#include <unistd.h>


void errorPipe() {
  const char *message = COLOR_BOLD_RED "error: " COLOR_WHITE "pipe failed\n";
  write(STDERR_FILENO, message, strlen(message));
  exit(EXIT_FAILURE);
}


void errorFork() {
  const char *message = COLOR_BOLD_RED "error: " COLOR_WHITE "fork failed\n";
  write(STDERR_FILENO, message, strlen(message));
  exit(EXIT_FAILURE);
}


void errorExecv() {
  const char *message = COLOR_BOLD_RED "error: " COLOR_WHITE "exec child\n";
  write(STDERR_FILENO, message, strlen(message));
  exit(EXIT_FAILURE);
}


void errorInvalidCountOfAgruments() {
  const char *message = COLOR_BOLD_CYAN "usage: " COLOR_WHITE "child filename\n";
  write(STDERR_FILENO, message, strlen(message));
  exit(EXIT_FAILURE);
}


void errorOpenFile() {
  const char *message = COLOR_BOLD_CYAN "error: " COLOR_WHITE "cannot open file\n";
  write(STDERR_FILENO, message, strlen(message));
  exit(EXIT_FAILURE);
}


void errorInvalidFileName() {
  const char *message = COLOR_BOLD_CYAN "error: " COLOR_WHITE "invalid file name\n";
  write(STDERR_FILENO, message, strlen(message));
  exit(EXIT_FAILURE);
}
```

Протокол работы программы

```
$ ./parent
child1.txt
child2.txt
Hello
Ola
54321
$ cat < child1
olleH
12345
$ cat < child2
alO
```

Вывод

В результате выполнения лабораторной работы мне удалось отработать на практике знания по работе прерываний и процессов. В реализации программы удалось использовать приёмы в программировании, изученные в ходе выполнение лабораторных работ по дисциплине “Фундаментальные алгоритмы”.