Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу
«Операционные системы»

Процессы операционных систем

Студент: Шевлякова София Сергеевна
Группа: М8О–208Б–21
Вариант: 16
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2022

Постановка задачи

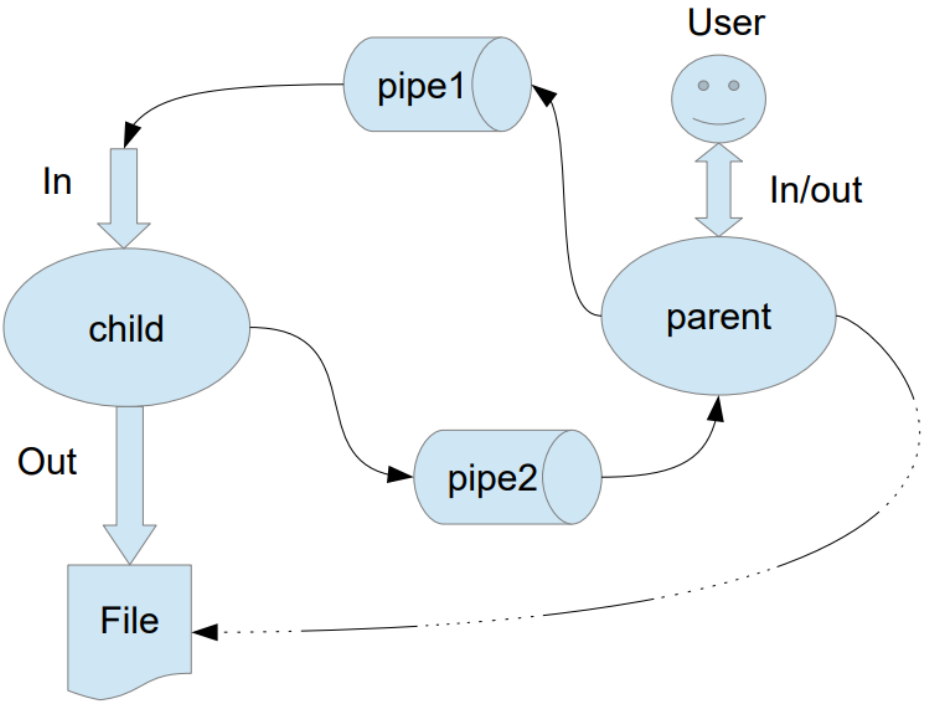
Цель работы

Приобретение практических навыков в:

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

Задание

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. Правило проверки: строка должна оканчиваться на «.» или «;».



Общие сведения о программе

Программа компилируется из файла main.c. Также используется заголовочные файлы: stdio.h, stdlib.h, unistd.h, sys/wait.h, fcntl.h. В программе используются следующие системные вызовы:

1. **write () -** переписывает count байт из буффера в файл. Возвращает количество записанных байт или -1;
2. **read () -** считывает count байт из файла в буффер. Возвращает количество считаных байт (оно может быть меньше count) или -1;
3. **pipe () -** создаёт канал между двумя процессами. Создаёт и помещает в массив 2 файловых дескриптора для чтения и для записи. Возвращает 0 или -1;
4. **open () -** открывает или создаёт файл при необходимости. Возвращает дескриптор открытого файла или -1;
5. **close() -** закрывает файловый дескриптор, который больше не ссылается ни на один файл, возвращает 0 или -1;
6. **fork () -** порождается процесс-потомок. Весь код после fork() выполняется дважды, как в процессе-потомке, так и в процессе-родителе. Процесс-потомок и процесс-родитель получают разные коды возврата после вызова fork(). Процесс-родители возвращает идентификатор pid потомка или -1. Процесс-потомок возвращает 0 или -1;
7. **dup2 () –** переназначение файлового дескриптора, старый и новый файловые дескрипторы являются взаимозаменяемыми, указывают на одно и то же. Возвращает новый дескриптор или -1**;**
8. **execv () -** заменяет текущий образ процесса новым образом процесса. Новая программа наследует от вызывавшего процесса идентификатор и открытые файловые дескрипторы;
9. **waitpid () -** ожидание завершения дочериного процесса или сигнала, приостанавливает на время родительский процесс. Возвращает идентификатор дочернего процесса, который завершил выполнение или -1;

Общий метод и алгоритм решения

Используя системный вызов pipe() 2 раза, создадим 2 канала между процессами: в первый канал будет записывать родительский процесс и читать дочерний, а во второй канал – наоборот. Родительский процесс из стандартного потока ввода прочитает первую строчку, введённую пользователем, и создаст файл с таким именем только на запись. При помощи системного вызова fork() мы создадим дочерний процесс. Для дочернего процесса fork() вернет 0, а для родительского – идентификатор дочернего процесса (число > 0). Находясь в дочернем процессе, закроем ненужные файловый дескрипторы, чтобы избежать ошибок. Так как дочерний и родительский процесс должны быть представлены разными программами, то с помощью execv() заменим текущий образ процесса образом файла child.

У родительского процесса так же закрываем ненужные файловые дескрипторы. Считываем строчку, которую ввел пользователь в консоль, и записываем сначала ее длину, а потом и всю строчку в pipe1. Дочерний процесс сможет прочитать эту строчку, проверить на валидность правилу и отправить через pipe2 родительскому процессу или длину этой строчки, или же -1. Родительский процесс обрабатывает только -1 и выводит в стандартный поток выводы сообщение об ошибке. Так как в задании не указано условие окончание ввода, то пусть это будет просто переход пользователя на следующую строку – `\n`, такая строка будет содержать только один символ. После окончания ввода закрываются все файловые дескрипторы в родительском и дочернем процессе, также используется системный вызов waitpid(), чтобы родительский процесс не завершился раньше дочернего.

Основные файлы программы

========== main.c ==========

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

#define WRITE_END 1
#define READ_END 0
#define TRUE 1

void send_error_and_stop(char *massage, int code) {
    int i = 0;
    while (massage[i] != '\0') {
        i++;
    }
    write(STDERR_FILENO, massage, sizeof(char) * i);
    exit(code);
}

int read_line(char **ptr) {
    if (ptr == NULL) {
        return -1;
    }
    int capacity = 10;
    (*ptr) = malloc(sizeof (char) * capacity);
    if ((*ptr) == NULL) {
        return -1;
    }
    char *new_ptr;
    char c;
    int i = 0;
    while (c != EOF && c !='\n') {
        if (read(STDIN_FILENO, &c, sizeof(char)) == -1) {
            send_error_and_stop("Cannot read symbol\n", 1);
            }
        if (c == '\n') break;
        (*ptr)[i++] = (char) c;
        if (i >= capacity / 2) {
            new_ptr = realloc((*ptr), sizeof (char) * capacity * 2);
            if (new_ptr == NULL) {
                return -1;
            }
```

```
            (*ptr) = new_ptr;
            capacity *= 2;
        }
    }
    (*ptr)[i++] = '\0';
    return i;
}

int main(int argc, const char *argv[]) {
    char *FileName;
    if (read_line(&FileName) == -1) {
        send_error_and_stop("Cannot read file name to open\n", 1);
    }

    int fd1[2];
    int fd2[2];
    int p1 = pipe(fd1);
    int p2 = pipe(fd2);
    if (p1 == -1 || p2 == -1) {
        send_error_and_stop("Pipe error\n", 1);
    }

    int fd;
    if ((fd = open(FileName, O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR)) == -1) {
        send_error_and_stop("Cannot open file\n", 2);
    }

    free(FileName);
    FileName = NULL;

    int pid = fork();
    if (pid == -1) {
        send_error_and_stop("Fork error\n", 3);
    } else if (pid == 0) {
//======================  child  ====================== //
        if (close(fd1[WRITE_END]) == -1 || close(fd2[READ_END]) == -1) {
            send_error_and_stop("Cannot close fd\n", 4);
        }

        char fd_file [10];
        char fd_read [10];
        char fd_write [10];

        sprintf(fd_file, "%d", fd);
        sprintf(fd_read, "%d", fd1[READ_END]);
```

```
        sprintf(fd_write, "%d", fd2[WRITE_END]);

        char *Child_args[] = {"child", fd_file, fd_read, fd_write, NULL};
        if (execv("child", Child_args) == -1) {
            send_error_and_stop("Cannot call exec child\n", 5);
        }
    } else {
// ========================  parent ====================== //
        if (close(fd1[READ_END]) == -1 || close(fd2[WRITE_END]) == -1) {
                send_error_and_stop("Cannot close fd\n", 6);
            }

            int len;
            int child_answer;
            char *line;
            while (TRUE) {
                len = read_line(&line);
                if (write(fd1[WRITE_END], &len, sizeof(int)) == -1) {
                    send_error_and_stop("Cannot write to fd\n", 7);
                }
                if (len <3 - 1) {
                    free(line);
                    line = NULL;
                    break;
                }
                if (write(fd1[WRITE_END], line, sizeof(char) * len) == -1) {
                    send_error_and_stop("Cannot write to fd\n", 7);
                }
                if (read(fd2[READ_END], &child_answer, sizeof(int)) == -1) {
                    send_error_and_stop("Cannot read from fd\n", 8);
                }
                if (child_answer == -1) {
                    printf("%s has no \';\' or \'.\' in the end\n", line);
                }

                free(line);
                line = NULL;
            }

            if (close(fd1[WRITE_END]) == -1 || close(fd2[READ_END]) == -1 || close(fd) == -1) {
                send_error_and_stop("Cannot close fd\n", 9);
            }

            if (waitpid(pid, NULL, 0) == -1) {
                send_error_and_stop("Waiting error\n",10);
```

```
                }
                return 0;
                }
}
======================== child.c ========================
  #include <stdio.h>
  #include <stdlib.h>
  #include <unistd.h>

  #define WRITE_END 1
  #define READ_END 0
  #define TRUE 1

  void send_error_and_stop(char *massage, int code) {
      int i = 0;
      while (massage[i] != '\0') {
          i++;
      }
      write(STDERR_FILENO, massage, sizeof(char) * i);
      exit(code);
  }

  int main(int argc, const char *argv[]) {
      if (argc < 4) {
          send_error_and_stop("Arguments missing\n", 1);
      }

      int fd;
      int fd1[2];
      fd = atoi(argv[1]);
      fd1[READ_END] = atoi(argv[2]);
      fd1[WRITE_END] = atoi(argv[3]);

      if (dup2(fd, STDOUT_FILENO) == -1) {
          send_error_and_stop("Cannot do dup2\n", 2);
      }

      int  len;
      char *str;
      int error_code = -1;
      while(TRUE) {

          if (read(fd1[READ_END], &len, sizeof (int)) == -1) {
              send_error_and_stop("Cannot read from file\n", 3);
          }
```

```
        if (len < 2) break;
        str = malloc(sizeof(char) * len);
        if (str == NULL) {
            send_error_and_stop("Cannot allocate memory\n", 4);
        }

        if (read(fd1[READ_END], str, sizeof (char) * len) == -1) {
            send_error_and_stop("Cannot read from file\n", 5);
        }

        if (str[len - 2] == '.' || str[len - 2] == ';') {
            if (printf("%s\n", str) == -1) {
                send_error_and_stop("Cannot write to file\n", 6);
            }

            if (write(fd1[WRITE_END], &len, sizeof(int)) == -1) {
                send_error_and_stop("Cannot write to fd\n", 7);
            }

        } else {
            if (write(fd1[WRITE_END], &error_code, sizeof(int)) == -1) {
                send_error_and_stop("Cannot write to fd\n", 8);
            }
        }

        free(str);
        str = NULL;
    }
    if (close(fd1[WRITE_END]) == -1 || close(fd1[READ_END]) == -1 || close(fd) == -1) {
        send_error_and_stop("Cannot close fd\n", 9);
    }
    return 0;
}
```

```
======================== test1.txt ========================
123
sdfds;
saaaaaaaaaa;
aaaa
111111.l

======================== test2.txt ========================
meow
qqqqqqqq;qqq;.q
```

qqq
111112232321FHNGJ.....

========== test3.txt ==========
.
dsasddd;;;
dsassdd
meowmeow
1343212321346788

Пример работы

```
sonikxx@LAPTOP-9UGJH447:~/OS/lab2_var166$ make
gcc main.c
gcc -c child.c
gcc child.o -o child
sonikxx@LAPTOP-9UGJH447:~/OS/lab2_var166$ ./a.out < test1.txt
aaaa has no ';' or '.' in the end
111111.1 has no ';' or '.' in the end
sonikxx@LAPTOP-9UGJH447:~/OS/lab2_var166$ ls
123  Makefile  a.out  child  child.c  child.o  main.c  test1.txt  test2.txt  test3.txt
sonikxx@LAPTOP-9UGJH447:~/OS/lab2_var166$ cat 123
sdfds;
saaaaaaaaaa;
sonikxx@LAPTOP-9UGJH447:~/OS/lab2_var166$ ./a.out < test2.txt
qqqqqqqq;qqq;.q has no ';' or '.' in the end
qqq has no ';' or '.' in the end
sonikxx@LAPTOP-9UGJH447:~/OS/lab2_var166$ ls
123  Makefile  a.out  child  child.c  child.o  main.c  meow  test1.txt  test2.txt  test3.txt
sonikxx@LAPTOP-9UGJH447:~/OS/lab2_var166$ cat meow
11112232321FHNGJ......
sonikxx@LAPTOP-9UGJH447:~/OS/lab2_var166$ ./a.out < test3.txt
Cannot open file
sonikxx@LAPTOP-9UGJH447:~/OS/lab2_var166$ ./a.out
doc
1242311111.;1
1242311111.;1 has no ';' or '.' in the end
2111111.
ssaaasaaaaaa
ssaaasaaaaaa has no ';' or '.' in the end

sonikxx@LAPTOP-9UGJH447:~/OS/lab2_var166$ ls
123  Makefile  a.out  child  child.c  child.o  doc  main.c  meow  test1.txt  test2.txt  test3.txt
sonikxx@LAPTOP-9UGJH447:~/OS/lab2_var166$ cat doc
2111111.
sonikxx@LAPTOP-9UGJH447:~/OS/lab2_var166$
```

Вывод

В ходе данной лабораторной работы я научилась управлять процессами при помощи системных вызовов и обеспечивать обмен данными между процессами при помощи неименованных каналов.

Системные вызовы необходимы для управления процессами, файлами и каталогами, а также каналами ввода и вывода данных. Одним из способов создания дочернего процесса является системный вызов fork(), он создает точную копию исходного процесса, включая все дескрипторы файлов, регистры и т. п. После выполнения вызова fork() исходный процесс и его копия (родительский и дочерний процессы) выполняются независимо друг от друга. Благодаря систему вызову pipe можно создать канал (трубу) между двумя

процессами, в которой один процесс сможем писать поток байтов, а другой процесс сможет его читать, так мы переопределяем потоки ввода-вывода.

Благодаря системным вызовам можно упростить программу или выполнить действия, запрещенные в пользовательском режиме.