Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу
«Операционные системы»

Студент: Шатунова Юлия Викторовна
Группа: М8О-208Б-20
Вариант: 13
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2021

Содержание

Репозиторий

https://github.com/s0bakkaa/OS/tree/main/lab2

Постановка задачи

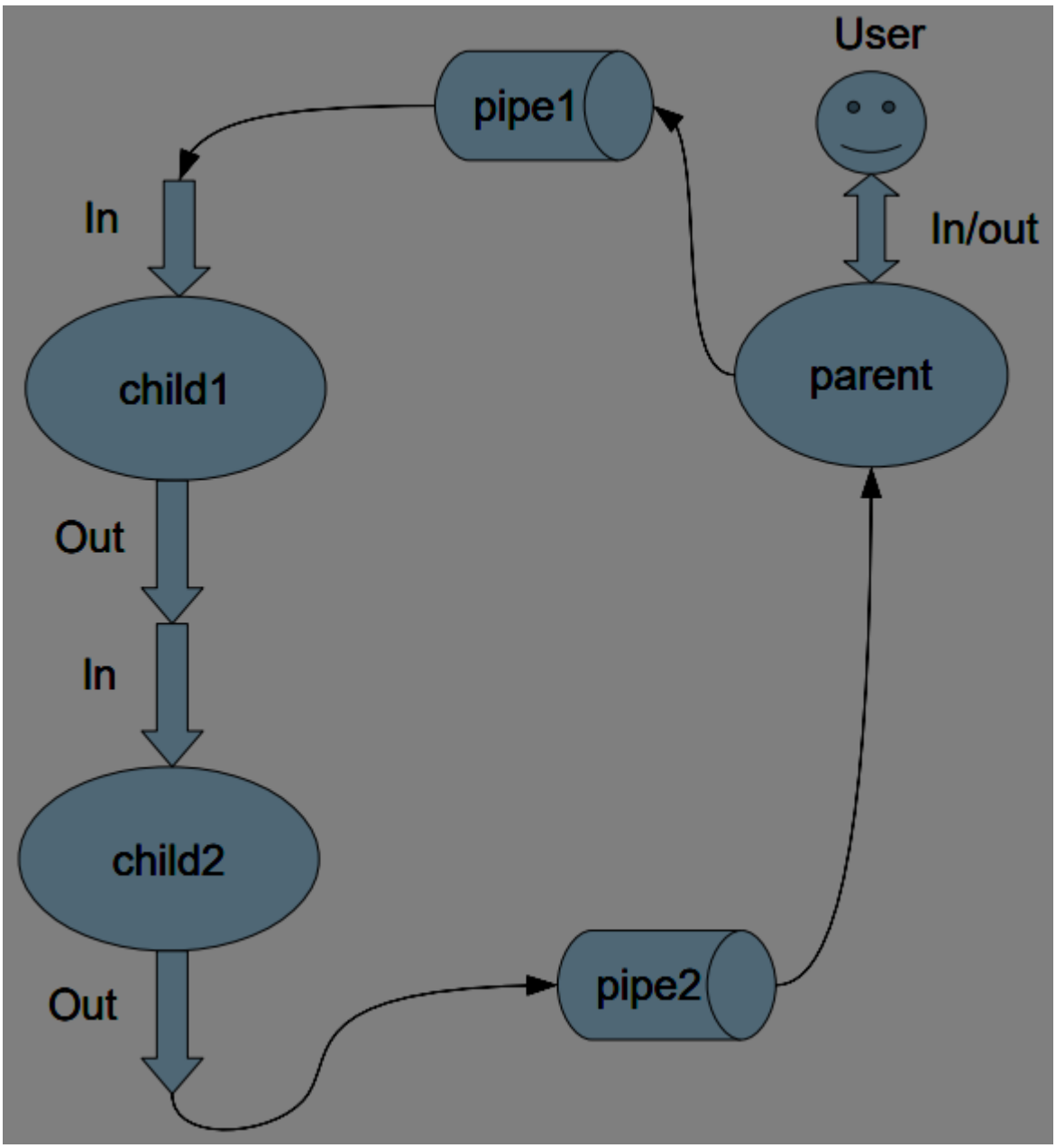
Цель работы

Приобретение практических навыков в:

1. Управление процессами в ОС
2. 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.


User
pipe1
In
In/out
child1
parent
Out
In
child2
Out
pipe2

13 вариант) Child1 переводит строки в нижний регистр. Child2 превращает все пробельные символы в символ «_».

Общие сведения о программе

Программа компилируется из файла main.cpp. Также используется заголовочные файлы: unistd.h, stdio.h , stdlib.h, ctype.h. В программе используются следующие системные вызовы:

1. **fork** - создает копию текущего процесса, который является дочерним процессом для текущего процесса
2. **pipe** - создаёт однонаправленный канал данных, который можно использовать для взаимодействия между процессами.
3. **fflush** - если поток связан с файлом, открытым для записи, то вызов приводит к физической записи содержимого буфера в файл. Если же поток указывает на вводимый файл, то очищается входной буфер.
4. **close** - закрывает файл.
5. **read** - читает количество байт(третий аргумент) из файла с файловым дескриптором(первый аргумент) в область памяти(второй агрумент).
6. **write -** записывает в файл с файловым дескриптором(первый аргумент) из области памяти(второй аргумент) количество байт(третий аргумент).
7. **perror –** вывод сообщения об ошибке.

Общий метод и алгоритм решения

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы fork, pipe, fflush, close, read, write.
2. Написать программу, которая будет работать с 3-мя процессами: один родительский и два дочерних, процессы связываются между собой при помощи pipe-ов.

   Организовать работу с выделением памяти под строку неопределенной длины и запись длины в массив строки в качестве первого элемента для передачи между процессами через pipe.

Исходный код

main.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>

int main() {
   int pipefd[2];  // child1->child2
   int pipefd_1[2];  // parent->child1
   int pipefd_2[2];  // child2->parent
   int errno = 0;

   pid_t pid_child1 = 0;
   pid_t pid_child2 = 0;

        // pipes: child1->child2, parent->child1, child2->parent
        if (pipe(pipefd) == -1 || pipe(pipefd_1) == -1 || pipe(pipefd_2) == -1) {
                perror("pipe error");
                exit(EXIT_FAILURE);
        }
```

```
if ((pid_child1 = fork()) > 0) {  // create the 1st process
   if ((pid_child2 = fork()) > 0) {  // create the 2nd process
      // parent
      char *str_in = (char *) malloc(sizeof(char) * 2);
      str_in[0] = 0;
      char c;
      while ((c = getchar()) != EOF) {
         str_in[0] += 1;
         str_in[str_in[0]] = c;
         str_in = (char *) realloc(str_in, (str_in[0] + 2) * sizeof(char));
      }
      str_in[str_in[0]] = '\0';
      errno = write(pipefd_1[1], str_in, (str_in[0] + 2) * sizeof(char));
                        if (errno == -1) {
                                 perror ("write error");
                                 exit(EXIT_FAILURE);
                        }
      char *str_out = (char *) malloc(sizeof(char));
      errno = read(pipefd_2[0], &str_out[0], sizeof(char));
                        if (errno == -1) {
                                 perror("read error");
                                 exit(EXIT_FAILURE);
                        }
      str_out = (char *) realloc(str_out, (str_out[0] + 2) * sizeof(char));
      for (int i = 1; i < str_out[0] + 1; ++i) {
         errno = read(pipefd_2[0], &str_out[i], sizeof(char));
                                 if (errno == -1) {
                                          perror("read error");
                                          exit(EXIT_FAILURE);
                                 }
         printf("%c", str_out[i]);
      }
      printf("\n");
```

```
    close(pipefd_2[0]);
    close(pipefd_1[1]);
    free(str_in);
    free(str_out);
}
else if (pid_child2 == 0) { // child2
    fflush(stdin);
    fflush(stdout);
    char *str_in = (char *) malloc(sizeof(char));
    errno = read(pipefd[0], &str_in[0], sizeof(char));
                        if (errno == -1) {
                                perror("read error");
                                exit(EXIT_FAILURE);
                        }
    str_in = (char *) realloc(str_in, (str_in[0] + 2) * sizeof(char));
    for (int i = 1; i < str_in[0] + 1; i++) {
        errno = read(pipefd[0], &str_in[i], sizeof(char));
                                if (errno == -1) {
                                        perror("read error");
                                        exit(EXIT_FAILURE);
                                }
    }
    char *str_out = (char *) malloc(2 * sizeof(char));
    str_out[0] = str_in[0];;
    for (int i = 1; i < str_in[0]; i++) {  // ' '->'_'
        if (str_in[i] == ' ') {
            str_out[i] = '_';
        }
        else {
                str_out[i] = str_in[i];
        }
        str_out = (char *) realloc(str_out, (str_out[0] + 2) * sizeof(str_out));
    }
```

```
        str_out[0]++;
        str_out[str_out[0]] = '\0';
        errno = write(pipefd_2[1], str_out, (str_out[0] + 2) * (sizeof(char)));
                        if (errno == -1) {
                                perror ("write error");
                                exit(EXIT_FAILURE);
                        }
        fflush(stdout);
        close(pipefd_2[1]);
        close(pipefd[0]);
        free(str_in);
        free(str_out);
    }
                else {
                        perror("fork #2 error");
                        exit(EXIT_FAILURE);
                }
}
else if (pid_child1 == 0) {  // child1
    char *str_in = (char *) malloc(sizeof(char));
    errno = read(pipefd_1[0], &str_in[0], sizeof(char));
                if (errno == -1) {
                        perror("read error");
                        exit(EXIT_FAILURE);
                }
    str_in = (char *) realloc(str_in, (str_in[0] + 2) * sizeof(char));
    char *str_out = (char *) malloc((str_in[0] + 2) * sizeof(char));
    str_out[0] = str_in[0];
    for (int i = 1; i < str_in[0] + 1; i++) {  // UPPER->lower
        errno = read(pipefd_1[0], &str_in[i], sizeof(char));
                        if (errno == -1) {
                                perror("read error");
                                exit(EXIT_FAILURE);
```

```
                        }
            str_out[i] = tolower(str_in[i]);
        }
        errno = write(pipefd[1], str_out, (str_out[0] + 2) * sizeof(char));
                if (errno == -1) {
                        perror("write error");
                        exit(EXIT_FAILURE);
                }
        close(pipefd_1[0]);
        close(pipefd[1]);
        free(str_in);
        free(str_out);
    }
    else {
                perror("fork #1 error");
                exit(EXIT_FAILURE);
    }
          return EXIT_SUCCESS;
}
```

Демонстрация работы программы

```
[yulia@andromeda lab2]$ cat test.txt
Take Care Of Nature
WASH YOUR HANDS
Beware  Of  PEOPLE
[yulia@andromeda lab2]$ gcc main.c
[yulia@andromeda lab2]$ ./a.out < test.txt
take_care_of_nature
wash_your_hands
beware__of__people
```

Выводы

Существуют специальные системные вызовы(fork) для создания процессов, также существуют специальные каналы pipe, которые позволяют связать

процессы и обмениваться данными при помощи этих pipe-ов. При использовании fork важно помнить, что фактически создается копию вашего текущего процесса и неправильная работа может привести к неожиданным результатам и последствиям, однако создание процессов очень удобно, когда вам нужно выполнять несколько действий параллельно. Также у каждого процесса есть свой id, по которому его можно определить. Также важно работать с чтением и записью из канала, помня что read, write возвращает количество успешно считанных/записанных байт и оно не обязательно равно тому значению, которое вы указали. Также важно не забывать закрывать pipe после завершения работы.