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-210БВ-24

Студент: Белков А.Д.

Преподаватель: Бахарев В.Д.

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: 02.10.25

Москва, 2025

**Постановка задачи**

**Вариант 11.**

Родительский процесс создаёт два дочерних процесса. Перенаправление стандартных потоков ввода-вывода показано на картинке выше. Child1 и Child2 можно ”соединить” между собой дополнительным каналом. Родительский и дочерний процесс должны быть представлены разными программами. Родительский процесс принимает от пользователя строки произвольной длины и пересылает их в pipe1. Процесс child1 и child2 производят работу над строками. Child2 пересылает результат своей работы родительскому процессу. Родительский процесс полученный результат выводит в стандартный поток вывода.

Child 1 переводит строки в верхний регистр. Child2 превращает все пробельные символы в символ ”\_”.

**Общий метод и алгоритм решения**

Использованные системные вызовы:

* pid\_t fork(void); – создает дочерний процесс.
* int pipe(int \*fd); – создаёт канал (pipe), позволяющий процессам обмениваться данными через файловые дескрипторы. Возвращает -1, если возникла ошибка при создании. Заполняет массив fd.

fd[0] – файловый дескриптор, использующийся для чтения.

fd[1] – файловый дескриптор, использующийся для записи.

* ssize\_t readlink(const char \*path, char\* buf, size\_t bufsize); – считывает содержание символической ссылки и записывает в buf.
* int write(int fd, void \*buf, size\_t count); – записывает данные из буфера по файловому дескриптору в файл или канал, возвращает количество реально записанных байт.
* int read(int fd, void \*buf, size\_t count); – считывает данные из файла или канала по файловому дескриптору.
* int close(int fd); – закрывает файловый дескриптор.
* int dup2(int oldfd, int newfd); – перенаправляет файловый дескриптор, позволяя процессу использовать канал (pipe) вместо стандартного ввода/вывода.
* int execv(const char \*path, char \*const argv[]); – выполняет замену текущего процесса новым, загружая и исполняя указанную программу. Существующая программа, запущенная в процессе удаляется, а в текущий процесс загружаются новые стек, данные и куча.
* pid\_t waitpid(pid\_t pid, int \*wstatus, int options); – ожидает завершения конкретного дочернего процесса.

Я реализовал межпроцессное взаимодействие с помощью системных вызовов. Есть родительский процесс, который порождает два дочерних процесса. Первый преобразует все символы в передаваемой строке в верхний регистр, второй заменяет пробельные символы на символ ”\_”. Взаимодействие между порождёнными процессами (fork) происходит посредством канала, созданного функцией pipe.

Описание работы программы:  
1. Родительский процесс создаёт три канала:

* pipe1: отвечает за передачу данных от родителя к первому дочернему процессу.
* pipe\_child: отвечает за передачу данных от первого дочернего процесса ко второму.
* pipe2: отвечает за передачу данных от второго дочернего процесса обратно к родителю.

2. Родитель порождает два дочерних процесса с помощью функции fork():

* Child1 перенаправляет стандартный ввод на pipe1, а вывод - на pipe\_child. После чего с помощью функции execv() загружает программу child с параметром 1.
* Child2 перенаправляет ввод на pipe\_child, а вывод - на pipe2. После чего с помощью функции execv() загружает программу child, но уже с параметром 2.

3. child.c:

* Если параметр равен 1, то процесс преобразует все символы во входном потоке в верхний регистр.
* Если параметр равен 2, то процесс заменяет все пробельные символы (’ ’ и ’\t’) на символ подчёркивания ’\_’.

4. Пользователь взаимодействует только с родительским процессом:

* Ввод данных с клавиатуры.
* Родитель перенаправляет введённый текст по цепочке: родитель -> child1 -> child2 -> родитель.
* На выходе на экран выводится уже преобразованный текст

5. После завершения работы все файловые дескрипторы закрываются, а родительский процесс дожидается завершения дочерних процессов с помощью функции waitpid().

**Код программы**

**client.c**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdint.h>

#include <ctype.h>

#include <sys/wait.h>

static char CHILD\_PROGRAM\_NAME[] = "child";

int main(int argc, char\* argv[]){

char prograth[1024];

{

ssize\_t len = readlink("/proc/self/exe", prograth, sizeof(prograth) -1);

if (len == -1){

const char msg[] = "error: failed to read full program path\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

}

while (prograth[len] != '/')

--len;

prograth[len] = '\0';

}

// создание каналов

int pipe1[2]; // parent -> child1

int pipe2[2]; // child2 -> parent

// обработали ошибку создания pipe

if (pipe(pipe1) == -1 || pipe(pipe2) == -1){

const char msg[] = "error: failed to create pipe\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

}

// создание child1

int pipe\_child[2]; // pipe child1 -> child2

if (pipe(pipe\_child) == -1){

const char msg[] = "error: failed to create internal pipe\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

}

pid\_t child1 = fork();

if (child1 == -1){

const char msg[] = "error: failed to spawn child1\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

}

else if (child1 == 0){

// перенаправляем stdin child1 на pipe1

dup2(pipe1[0], STDIN\_FILENO);

dup2(pipe\_child[1], STDOUT\_FILENO);

// закрываем лишние концы

close(pipe1[0]);

close(pipe1[1]);

close(pipe2[0]);

close(pipe2[1]);

close(pipe\_child[0]);

close(pipe\_child[1]);

char path[1024];

snprintf(path, sizeof(path), "%s/%s", prograth, CHILD\_PROGRAM\_NAME);

char \*const args[] = {CHILD\_PROGRAM\_NAME, "1", NULL};

execv(path, args);

const char msg[] = "error: failed to exec child\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

}

// создание child2

pid\_t child2 = fork();

if (child2 == -1){

const char msg[] = "error: failed to spawn child1\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

}

else if (child2 == 0){

// перенастраиваем ввод/вывод для второго ребёнка

dup2(pipe\_child[0], STDIN\_FILENO);

dup2(pipe2[1], STDOUT\_FILENO);

close(pipe1[0]);

close(pipe1[1]);

close(pipe2[0]);

close(pipe2[1]);

close(pipe\_child[0]);

close(pipe\_child[1]);

char path[1024];

snprintf(path, sizeof(path), "%s/%s", prograth, CHILD\_PROGRAM\_NAME);

char \*const args[] = {CHILD\_PROGRAM\_NAME, "2", NULL};

execv(path, args);

const char msg[] = "error: failed to exec child2\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

}

// parent

close(pipe1[0]); // читающий конец

close(pipe2[1]); // записывающий конец

close(pipe\_child[0]);

close(pipe\_child[1]); // pipe для детей

char buf[4096];

ssize\_t sz;

while ((sz = read(STDIN\_FILENO, buf, sizeof(buf))) > 0) {

write(pipe1[1], buf, sz);

sz = read(pipe2[0], buf, sizeof(buf));

write(STDOUT\_FILENO, buf, sz);

}

// закрываем pipe'ы

close(pipe1[1]);

close(pipe2[0]);

// ждём пока дети закончат свою работу

waitpid(child1, NULL, 0);

waitpid(child2, NULL, 0);

return 0;

}

**child.c**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdint.h>

#include <ctype.h>

int main(int argc, char\* argv[]){

if (argc < 2){

const char msg[] = "error: not enough arguments, specify child id(1 or 2)\n";

write(STDERR\_FILENO, msg, sizeof(msg));

exit(EXIT\_FAILURE);

}

// argv[1][0] ([0] - "./child" ; [1] - 1 or 2)

int child\_number = argv[1][0] - '0'; // получаем 1-ый или 2-ой дочерний процесс

char buf[4096];

ssize\_t sz;

while ((sz = read(STDIN\_FILENO, buf, sizeof(buf))) > 0){

if (child\_number == 1){

// перевод в верхний регистр

for (ssize\_t i = 0; i < sz; ++i){

buf[i] = toupper(buf[i]);

}

write(STDOUT\_FILENO, buf, sz);

}

else {

// замена пробельных символов на '\_'

for (ssize\_t i = 0; i < sz; ++i){

if (buf[i] == ' ' || buf[i] == '\t')

buf[i] = '\_';

}

write(STDOUT\_FILENO, buf, sz);

}

}

return 0;

}

**Протокол работы программы**

**Тестирование:**

artmlink@pop-os:~/2\_course\_MAI/MAI-OS-Labs-2025/lab-1$ ./client

string1

STRING1\_

sTrIng2

STRING2\_\_\_\_\_\_

make America great again

MAKE\_AMERICA\_GREAT\_AGAIN\_\_\_

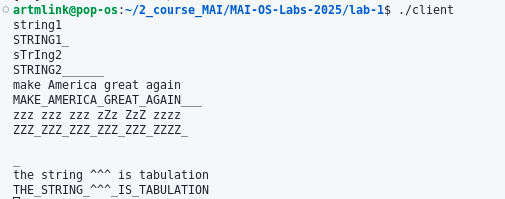
zzz zzz zzz zZz ZzZ zzzz

ZZZ\_ZZZ\_ZZZ\_ZZZ\_ZZZ\_ZZZZ\_

\_

the string ^^^ is tabulation

THE\_STRING\_^^^\_IS\_TABULATION



**Вывод**

В ходе выполнения данной лабораторной работы было изучено взаимодействие процессов через каналы (pipe) и механизмы их работы при создании дочерних процессов с помощью fork(). В процессе выполнения возникали трудности с правильным расположением вызова pipe(), а также с пониманием того, как правильно закрывать неиспользуемые дескрипторы. В дальнейшем хотелось бы пожелать, чтобы при выдаче лабораторных работ также прилагались ссылки на дополнительные материалы (статьи, видео и т.п.), которые облегчат изучение материала и дадут исчерпывающие примеры использования необходимых для реализации лабораторных работ функций.