**Московский авиационный институт**

**(Национальный исследовательский университет)**

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Операционные системы»

**Лабораторная работа № 2**

Студент: Ваньков Д. А.

Группа: 80-207Б

Преподаватель: Соколов А. А.

Дата: 20.10.2018

Оценка:

Москва, 2018

1. **Постановка задачи**

Вариант 26.

Дочерний процесс при создании принимает имя файла. При работе дочерний процесс получает числа от родительского процесса и пишет их в файл. Родительский процесс создает n дочерних процессов и передает им поочередно числа из последовательности от 1...m.

Операционная система: Unix.

**Целью лабораторной работы является:**

* Приобретение практических навыков в:
* Управление процессами в ОС
* 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 решения 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.

1. **Решение задачи**

Используемые системные вызовы:

* **ssize\_t write(int***fd***, const void \****buf***, size\_t***count***) -** пишет до *count* байт из буфера, на который указывает *buf*, в файле, на который ссылается файловый дескриптор *fd*.
* **ssize\_t read(int***fd***, void \****buf***, size\_t***count***) -** пытается прочитать *count* байт из файлового дескриптора *fd* в буфер, начинающийся по адресу *buf*. Для файлов, поддерживающих смещения, операция чтения начинается с текущего файлового смещения, и файловое смещение увеличивается на количество прочитанных байт. Если текущее файловое смещение находится за концом файла, то ничего не читается и **read**() возвращает ноль. Если значение *count* равно 0, то **read**() *может* обнаружить ошибки, описанные далее. При отсутствии ошибок, или если **read**() не выполняет проверки, то **read**() с *count* равным 0 возвращает 0 и ничего не меняет.
* **pid\_t fork(void) -** создаёт новый процесс посредством копирования вызывающего процесса. Новый процесс считается *дочерним* процессом. Вызывающий процесс считается *родительским* процессом. Дочерний и родительский процессы находятся в отдельных пространствах памяти. Сразу после **fork**() эти пространства имеют одинаковое содержимое.
* **int close(int***fd***) -** закрывает файловый дескриптор.
* **int pipe2(int***pipefd***[2], int***flags***) -** создаёт однонаправленный канал данных, который можно использовать для взаимодействия между процессами. Массив *pipefd* используется для возврата двух файловых описателей, указывающих на концы канала. *pipefd[0]* указывает на конец канала для чтения. *pipefd[1]* указывает на конец канала для записи. Данные, записанные в конец канала, буферизируются ядром до тех пор, пока не будут прочитаны из конца канала для чтения.

Каждая из функций - родитель и ребёнок - принимают идентификаторы потоков, на чтение и на запись. Каждый из них передаётся так, что "чтение с читабельного конца потока 1 и запись в писательный конец потока 2" и "чтение с читабельного конца потока 2 и запись в писательный конец потока1".

Функция pipe() создаёт собственный поток обмена данными. При этом в передаваемый ей массив записываются числовые идентификаторы (дескрипторы) двух "концов" потока: один на чтение, другой на запись. Поток (pipe) работает по принципу "положенное первым - первым будет считано".

int pid = fork(); Начиная с этого момента, процессов становится два. У каждого своя память. в процессе-родителе pid хранит идентификатор ребёнка. в ребёнке в этой же переменной лежит 0. Далее в каждом случае надо закрыть "лишние" концы потоков. поскольку сама программа теперь существует в двух экземплярах, то фактически у каждого потока появляются вторые дескрипторы. И если произошла хоть какая-то ошибка, то общая переменная ошибок != 0.

**ТЕСТЫ ПРОГРАММЫ:**

FORK: Success

==========

Enter name file

Enter n

Enter m

Enter number

End Enter

Enter number

End Enter

Child

Enter number

End Enter

Child

chappybunny@chappybunny:~/OS$ cat out.txt

0 1 2 3 4 5 6 7

0 1 2 3 4 5 6 7

0 1 2 3 4 5 6 7

chappybunny@chappybunny:~/OS$

1. **Руководство по использованию программы**

Компиляция и запуск программного кода в *Ubuntu* :

gcc os2.c && echo "out.txt 3 8"|./a.out && cat out.txt

1. **Листинг программы**

#include <unistd.h>

#include <stdio.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <fcntl.h>

void parentProces(int\* pipe\_fd, int m, char \*fname) {

int d;

close(pipe\_fd[0]);

printf("Enter number\n");

for (int i = 0; i < m; ++i)

write(pipe\_fd[1], &i, sizeof(i));

close(pipe\_fd[1]);

}

void childProces(int\* pipe\_fd, char \*fname) {

int d;

int fd;

printf("Child\n");

close(pipe\_fd[1]);

fd = open(fname, O\_CREAT | O\_APPEND | O\_WRONLY, S\_IWUSR | S\_IRUSR); // Открыть на дозапись, если нет создать с правами без sudo

dup2(fd, 1); // Перенаправить вывод в файл fd

while(read(pipe\_fd[0], &d, sizeof(d)) > 0) {

printf("%d ", d);

}

printf("\n");

close(fd);

}

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

int pipe\_fd[2];

pid\_t pid;

char name\_file[20];

int count\_process;

int m;

int err = 0;

err = -1;

perror("FORK");

printf("\n==========\nEnter name file\n");

scanf ("%s", name\_file);

printf("Enter n\n");

scanf ("%d", &count\_process);

printf("Enter m\n");

scanf ("%d", &m);

int i = 0;

for (i; i < count\_process; ++i) {

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

perror("PIPE");

err = -2;

}

pid = fork();

if(pid == -1) {

perror("FORK");

err = -1;

}

else if (pid == 0) {

childProces(pipe\_fd, name\_file);

break;

} else

parentProces(pipe\_fd, m, name\_file);

printf("End Enter\n");

}

return err;

}

1. **Вывод**

Современные программы редко работают в одном процессе или потоке. Довольно частая ситуация: нам необходимо запустить какую-то программу из нашей. Также многие программы создают дочерние процессы не для запуска другой программы, а для выполнения параллельной задачи. Например, так поступают простые сетевые серверы — при подсоединении клиента, сервер создаёт свою копию (дочерний процесс), которая обслуживает клиентское соединение и завершается по его закрытии. Родительский же процесс продолжает ожидать новых соединений.

При работе с потоками и процессами необходимо быть весьма осторожным, так как возможно допустить различные ошибки, которые затем будет сложно отловить и исправить.