Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №2 по курсу**

**«Операционные системы»**

**Управление процессами в ОС**

Студент: Базаргармаев Нима Дондокович

Группа: М80 – 301Б-18

Вариант: 2

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2020

**Содержание**

1. Постановка задачи
2. Общие сведения о программе
3. Общий метод и алгоритм решения
4. Основные файлы программы
5. Тестирование
6. Демонстрация работы программы
7. Вывод

**Постановка задачи**.

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.

**Задание согласно 2 варианту:**

Рекурсивное вычисление чисел Фибоначчи, где каждый отдельный уровень рекурсии вычисляется в отдельном процессе.

**Общие сведения о программе**

Программа компилируется из одного файла fib.c. В данном файле используются заголовочные файлы sys/types.h, sys/wait.h, unistd.h, errno.h, stdlib.h, string.h. В программе используются следующие системные вызовы:

1. **read/write** – предназначены для осуществления потоковых операций ввода (чтения) и вывода (записи) информации над каналами связи, описываемыми файловыми дескрипторами, т.е. для pipe, файлов и для потокового ввода.
2. **pipe** – для создания однонаправленного канала, через который могут общаться два процесса. При нормальном завершении вызова в первый элемент массива(аргумент pipe) – fd[0] – будет занесен файловый дескриптор, соответствующий выходному потоку данных pip’а и позволяющий выполнять только операцию чтения, а во второй элемент массива – fd[1] – будет занесен файловый дескриптор, соответствующий входному потоку данных и позволяющий выполнять только операцию записи. Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибок.
3. **fork** – системный вызов для порождения нового процесса. Процесс, который инициировал системный вызов fork, принято называть родительским процессом (parent process). Вновь порожденный процесс принято называть процессом-ребенком (child process). Процесс-ребенок является почти полной копией родительского процесса. У порожденного процесса по сравнению с родительским изменяются значения следующих параметров: PID, PPID. При однократном системном вызове возврат из него может произойти дважды: один раз в родительском процессе, а второй раз в порожденном процессе. Если создание нового процесса произошло успешно, то в порожденном процессе системный вызов вернет значение 0, а в родительском процессе – положительное значение, равное идентификатору процесса-ребенка. Если создать новый процесс не удалось, то системный вызов вернет в инициировавший его процесс отрицательное значение.

**Общий метод и алгоритм решения**.

Для реализации поставленной задачи необходимо:

1. Используя системный вызов pipe создать канал, по которому будут обмениваться данными два процесса.
2. Используя системный вызов fork создать дочерний процесс.
3. В родительском процессе отправлять числа в первый pipe с помощью системного вызова write.
4. Пока родительский процесс не записал данные в канал. Дочерний процесс ждет. Как только родительский процесс записал данные в канал, дочерний процесс считывает их, производит вычисления и возвращает результат родительскому процессу с помощью второго pipe. Родительский процесс ожидает, пока придут данные во второй pipe, а затем этот процесс их считает с помощью системного вызова read.
5. Родительский процесс возвращает результат вычислений.

**Основные файлы программы**.

**Файл fib.c**

#include <sys/types.h> // библиотека с типами данных size\_t

#include <sys/wait.h> // библиотека с waitpid()

#include <unistd.h> // библиотека с системными вызовами fork(), pipe(), read(), write(), close()

#include <errno.h> // бибоиотека с perror()

#include <stdlib.h>

#include <string.h>

int fd\_1[2]; // массив для pipe(), дескриптор 0 - чтение, 1 - запись

int Fibonacci(int n) {

int result, n\_1, n\_2 = 0;

int status = 0; // переменная для ожидания завершения процесса

ssize\_t wr = 0; // переменная для проверки записи в поток

if (n == 0) {

return 0;

} else if (n < 2) {

return n;

}

pid\_t pid, ppid; // id процесса-родителя и ребенка

pid = fork();

if (pid < 0) {

perror("Can't create fork\n");

exit(-1);

} else if (pid == 0) { // процесс-ребенок

n\_1 = Fibonacci(n - 1);

close(fd\_1[0]);

wr = write(fd\_1[1], &n\_1, sizeof(n\_1));

if (wr < 0) {

perror("Can't write in pipe\n");

}

close(fd\_1[1]);

exit(0);

} else { // процесс-родитель

n\_2 = Fibonacci(n - 2);

}

// ожидаем завершения процессов и сохраняем все в буффер

if (waitpid(pid, &status, 0) < 0) {

perror("Error in waitpid\n");

}

if (read(fd\_1[0], &n\_1, sizeof(n\_1)) < 0) {

perror("Can't read from pipe\n");

}

result = n\_1 + n\_2;

return result;

}

void reverse(char s[]) {

int i, j;

char c;

for (i = 0, j = strlen(s)-1; i<j; i++, j--) {

c = s[i];

s[i] = s[j];

s[j] = c;

}

}

void itoa(int n, char s[]) {

int i, sign;

i = 0;

do { // генерируем цифры в обратном порядке

s[i] = n % 10 + '0'; // берем следующую цифру

++i;

} while ((n /= 10) > 0); // удаляем

s[i] = '\0';

reverse(s);

}

int main() {

int n, num\_len = 0; // номер числа в последовательности и его длина

char read\_sym[10]; // массив, содержащий считываемое число

write(1, "Enter the number: ", sizeof("Enter the number: ")); // 1 - это стандартный поток вывода

if (pipe(fd\_1) < 0) {

perror("Can't create pipe\n");

exit(-1);

}

num\_len = read(2, &read\_sym, 10); // 2 - стандартный поток ввода

read\_sym[num\_len - 1] = '\0';

n = atoi(read\_sym); // преобразуем считанную строку в int

if (n <= 0) {

write(1, "Number must be > 0\n", sizeof("Number must be > 0\n"));

} else {

n = Fibonacci(n);

char result[10];

for (int i = 0; i < 10; ++i) {

result[i] = ' ';

}

itoa(n, result);

write(1, result, sizeof(result));

write(1, "\n", sizeof("\n"));

}

return 0;

}

**Тестирование.**

См. следующий пункт.

**Демонстрацияработыпрограммы.**

user@DESKTOP-4RFVSK2:~/os/2lab$ ./a.out

Enter the number: 1

1

user@DESKTOP-4RFVSK2:~/os/2lab$ ./a.out

Enter the number: 0

Number must be > 0

user@DESKTOP-4RFVSK2:~/os/2lab$ ./a.out

Enter the number: 2

1

user@DESKTOP-4RFVSK2:~/os/2lab$ ./a.out

Enter the number: 3

2

user@DESKTOP-4RFVSK2:~/os/2lab$ ./a.out

Enter the number: 4

3

user@DESKTOP-4RFVSK2:~/os/2lab$ ./a.out

Enter the number: 5

5

user@DESKTOP-4RFVSK2:~/os/2lab$ ./a.out

Enter the number: 7

13

user@DESKTOP-4RFVSK2:~/os/2lab$ ./a.out

Enter the number: 8

21

**Вывод.**

Pipe() не очень удобен в более усложненных задачах. Механизм pipe пригоден лишь для родственных процессов, имеющих общего прародителя, инициировавшего системный вызов pipe() и для общения с другими процессами использоваться pipe()не может.

Так же pipe() имеет однонаправленную связь, т.е. процесс-родитель пишет информацию в pipe, а затем процесс-ребенок ее оттуда считывает, но уже процесс-ребенок не сможет записать данные в этот pipe.

При громоздких программах очень сложно уследить все особенности и нюансы работы системных вызовов read() и write() при работе c pip’ом, а, следовательно, возможны ошибки и проблемы с работоспособностью программы.

Системный вызов pipe() явно не лучшее решение для обмена данными между процессами, он требует определенных доработок и упрощенного использования для студентов.