

Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №2
по курсу «Операционные системы»
III Семестр

Вариант 17

Студент:	Короткевич Л. В.
Группа:	М80-208Б-19
Преподаватель:	Миронов Е.С
Оценка:	
Дата:	

1. 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 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

Группа вариантов 5:

Родительский процесс создает два дочерних процесса. Первой строкой пользователь в консоль родительского процесса вводит имя файла, которое будет использовано для открытия File с таким именем на запись для child1. Аналогично для второй строки и процесса child2. Родительский и дочерний процесс должны быть представлены разными программами.

Вариант 17:

Правило фильтрации: строки длины больше 10 символов отправляются в pipe2, иначе в pipe1. Дочерние процессы удаляют все гласные из строк.

2. Метод решения

Используемые системные вызовы для выполнения работы:

int pipe(int filedes[3]);

pipe создает пару файловых дескрипторов, указывающих на запись inode именowanego канала, и помещает их в массив, на который указывает filedes. filedes[0] предназначен для чтения, а filedes[1] предназначен для записи, filedes[2] – информации об ошибках.

void exit(int status);

Функция exit() приводит к обычному завершению программы, и величина status & 0377 (least significant byte of status) возвращается процессу-родителю.

pid_t fork(void);

fork создает процесс-потомок, который отличается от родительского только значениями PID (идентификатор процесса) и PPID (идентификатор родительского процесса), а также тем фактом, что счетчики использования ресурсов установлены в 0. Блокировки файлов и сигналы, ожидающие обработки, не наследуются.

int close(int fd);

close закрывает файловый дескриптор, который после этого не ссылается ни на один и файл и может быть использован повторно. Все блокировки, находящиеся на соответствующем файле, снимаются (независимо от того, был ли использован для установки блокировки именно этот файловый дескриптор).

int open(const char *pathname, int flags, mode_t mode);

Вызов open() используется, чтобы преобразовать путь к файлу в описатель файла. Если системный вызов завершается успешно, возвращенный файловый описатель является наименьшим описателем, который еще не открыт процессом. Новый описатель файла будет оставаться открытым при выполнении функции exes(2). Указатель устанавливается в начале файла.

int dup2(int oldfd, int newfd);
int dup(int oldfd);

dup и dup2 создают копию файлового дескриптора oldfd.

pid_t waitpid(pid_t pid, int *status, int options);

Функция waitpid приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс, указанный в параметре pid, не завершит выполнение, или пока не появится сигнал, который либо завершает текущий процесс либо требует вызвать функцию-обработчик.

Предисловие: считывать названия файлов для записи первого, второго потомков, как мне кажется, будет удобней, лаконичней через аргументы при запуске программы.

```
if (argc < 3)
{
    printf("Usage: ./main filename1 filename2\n");
    exit(-1);
}
```

```
char filename1[MAX_FNAME_LENGTH], filename2[MAX_FNAME_LENGTH];
printf("Name of the 1st child-output file: %s\n", argv[1]);
printf("Name of the 2nd child-output file: %s\n", argv[2]);
strcpy(filename1, argv[1]);
strcpy(filename2, argv[2]);
```

Основная же задача предельно проста: создать в основном процессе два потомка, перенаправить их вывод в соответствующие файлы; считывать, в свою очередь, потомки будут из родительского пайпа. Коль скоро у нас два ребенка, создадим массив “два на два” для хранения файловых дескрипторов.

```
int fd[2][2];
for (int i = 0; i < 2; ++i)
{
    if (pipe(fd[i]) < 0)
```

```

    {
        perror("Pipe error");
        exit(1);
    }
}

```

Теперь необходимо создать первый и второй процесс-потомок поочередно. Для этого я создал функцию `pid_t createchild(int *myfd, int *exfd, filename)`.

```

pid_t createchild(int *myfd, int *exfd, char *filename) {
...

```

По мере создания потомков, инициализации переменных `pid_t pid`, необходимо проверить, все ли хорошо.

```

pid_t pid = fork();
if(pid < 0) {
    perror("Fork err");
    exit(1);
}

```

Наконец, нужно закрыть лишние каналы у дочернего проц-а.

```

if(pid == 0) {
    close(exfd[0]);
    close(exfd[1]);
    close(myfd[WR]);
}

```

После чего нехитрыми манипуляциями перенаправить `stdout` первого потомка в `filename1`, прежде создав его.

```

int file_out = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0777);
if (file_out < 0)
{
    perror("File err");
    exit(1);
}

int new_out = dup2(file_out, STDOUT_FILENO);
close(file_out);
if (new_out < 0)
{
    perror("Duping child stdout err");
    exit(1);
}

```

И, конечно же, связать `stdin` с каналом родителя.

```

int new_in = dup2(myfd[RD], STDIN_FILENO);
close(myfd[RD]);
if (new_in < 0)

```

```

{
    perror("Duping child1 stdin err");
    exit(1);
}

```

В конце концов, запустим процесс и вернем PID.

```

if (execlp("./child", "child", NULL) < 0)
{
    perror("Execl err");
    exit(1);
}

```

```

return pid;

```

Мы успешно создали процесс-потомок, который элементарным образом будет считывать строки, что ему подадут, с помощью обычных функций ввода; с помощью же стандартных функций вывода он будет выводить результирующие строки (полный листинг “детей” можно наблюдать двумя разделами ниже).

Аналогичным образом, перед тем как считать входные данные, закроем у родительского процесса ненужные каналы.

```

close(fd[0][RD]);
close(fd[1][RD]);

```

Просто и лаконично считаем входные данные, строки, а затем отправим их первому или второму процессу-потомку в зависимости от их длины (строк).

```

printf("Enter strings to process: \n");
char msg[MAX_STR_LENGTH];
while (fgets(msg, MAX_STR_LENGTH, stdin))
{
    if (strlen(msg) <= 10)
    {
        if (write(fd[0][WR], msg, strlen(msg)) < 0)
        {
            perror("Write err");
            exit(1);
        }
    }
    else
    {
        if (write(fd[1][WR], msg, strlen(msg)) < 0)
        {
            perror("Write err");
            exit(1);
        }
    }
}

```

Финальный штрихкод: закрытие оставшихся каналов, дожидание окончания работы процессов-потомков, проверка успешного завершения их работы.

```
close(fd[0][WR]);
close(fd[1][WR]);

int statusChild1, statusChild2;
waitpid(pid1, &statusChild1, 0);
if (WIFEXITED(statusChild1))
{
    printf("Child 1 exited, returned %d\n", WEXITSTATUS(statusChild1));
}
else
{
    fprintf(stderr, "Something is wrong with child process\n");
}
waitpid(pid2, &statusChild2, 0);
if (WIFEXITED(statusChild1))
{
    printf("Child 2 exited, returned %d\n", WEXITSTATUS(statusChild2));
}
else
{
    fprintf(stderr, "Something is wrong with 2nd child process\n");
}
```

3. Тестирование

```
[leo@pc simplified]$ gcc main.c -o main
[leo@pc simplified]$ gcc child.c -o child
[leo@pc simplified]$ cat test01.txt
"The unexamined life is not worth living" – Socrates
https://www.google.com/search?
q=philosophy+phrases&oq=philosophy+phrases&aqs=chrome..69i57j0l7.14207j0j9&sourceid=chrome&ie=UTF-8
123
...biba
boba...
"If God did not exist, it would be necessary to invent Him" – Voltaire
[leo@pc simplified]$ ./main out1 out2 <test01.txt
Name of the 1st child-output file: out1
Name of the 2nd child-output file: out2
Enter strings to process:
Child 1 exited, returned 0
Child 2 exited, returned 0
[leo@pc simplified]$ cat out1
Child 558364: Started!
Received line: 123
Processed line: 123
Received line: ...biba
Processed line: ...bb
Received line: boba...
Processed line: bb...
```

```

Child 558364: I'm Done!
[leo@pc simplified]$ cat out2
Child 558365: Started!
Received line: "The unexamined life is not worth living" – Socrates
Processed line: "Th nxmnd lf s nt wrth lvng" – Scrts
Received line: https://www.google.com/search?
q=philosophy+phrases&oq=philosophy+phrases&aqs=chrome..69i57j0l7.14207j0j9&sourceid=chr
ome&ie=UTF-8
Processed line: https://www.ggl.cm/srch?
q=phlsph+phrss&q=phlsph+phrss&q=chr..6957j0l7.14207j0j9&srcd=chr&=TF-8
Received line: "If God did not exist, it would be necessary to invent Him" – Voltaire
Processed line: "f Gd dd nt xst, t wld b ncscr t nvnt Hm" – Vltr
Child 558365: I'm Done!
[leo@pc simplified]$ cat test02.txt
a
b
AxAxAx322
15102001leonidvitalyevich
!@#%&*(*)_+alabama
[leo@pc simplified]$ ./main out1 out2 <test02.txt
Name of the child-output file: out1
Name of the 2nd child-output file: out2
Enter strings to process:
Child 1 exited, returned 0
Child 2 exited, returned 0
[leo@pc simplified]$ cat out1
Child 558402: Started!
Received line: a
Processed line:
Received line: b
Processed line: b
Received line: AxAxAx322
Processed line: xxx322
Child 558402: I'm Done!
[leo@pc simplified]$ cat out2
Child 558403: Started!
Received line: 15102001leonidvitalyevich
Processed line: 15102001lndvtlvch
Received line: !@#%&*(*)_+alabama
Processed line: !@#%&*(*)_+lbn
Child 558403: I'm Done!
[leo@pc simplified]$ ./main out1 out2 <test03.txt
Name of the child-output file: out1
Name of the 2nd child-output file: out2
Enter strings to process:
Child 1 exited, returned 0
Child 2 exited, returned 0
[leo@pc simplified]$ cat out1
Child 558427: Started!
Child 558427: I'm Done!
[leo@pc simplified]$ cat out2
Child 558428: Started!

```

Received line: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaB
Processed line: B
Received line: Baaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Processed line: B
Received line: 1a2e3u4i5o6e
Processed line: 123456
Received line: U lukomoria dub zeleniy
Processed line: lkmr db zln
Child 558428: I'm Done!

4. Листинг программы

main.c:

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

#define MAX_FNAME_LENGTH 100
#define MAX_STR_LENGTH 300

#define RD 0 // Read end of pipe
#define WR 1 // Write end of pipe

pid_t createchild(int *myfd, int *exfd, char *filename) {
    pid_t pid = fork();
    if(pid < 0) {
        perror("Fork err");
        exit(1);
    }
    if(pid == 0) {
        close(exfd[0]);
        close(exfd[1]);
        close(myfd[WR]);
        int file_out = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0777);
        if (file_out < 0)
        {
            perror("File err");
            exit(1);
        }

        int new_out = dup2(file_out, STDOUT_FILENO);
        close(file_out);
        if (new_out < 0)
        {
            perror("Duping child stdout err");
            exit(1);
        }

        int new_in = dup2(myfd[RD], STDIN_FILENO);
        close(myfd[RD]);
        if (new_in < 0)
        {
            perror("Duping child stdin err");
            exit(1);
        }

        if (execlp("./child", "child", NULL) < 0)
        {
            perror("Execl err");
            exit(1);
        }
    }
    return pid;
}
```



```

int main(int argc, char *argv[])
{
    if (argc < 3)
    {
        printf("Usage: ./main filename1 filename2\n");
        exit(-1);
    }

    char filename1[MAX_FNAME_LENGTH], filename2[MAX_FNAME_LENGTH];
    printf("Name of the 1st child-output file: %s\n", argv[1]);
    printf("Name of the 2nd child-output file: %s\n", argv[2]);
    strcpy(filename1, argv[1]);
    strcpy(filename2, argv[2]);

    int fd[2][2];
    for (int i = 0; i < 2; ++i)
    {
        if (pipe(fd[i]) < 0)
        {
            perror("Pipe error");
            exit(1);
        }
    }

    pid_t pid1, pid2;
    pid1 = createchild(fd[0], fd[1], filename1);
    pid2 = createchild(fd[1], fd[0], filename2);

    close(fd[0][RD]);
    close(fd[1][RD]);

    printf("Enter strings to process: \n");
    char msg[MAX_STR_LENGTH];
    while (fgets(msg, MAX_STR_LENGTH, stdin))
    {
        if (strlen(msg) <= 10)
        {
            if (write(fd[0][WR], msg, strlen(msg)) < 0)
            {
                perror("Write err");
                exit(1);
            }
        }
        else
        {
            if (write(fd[1][WR], msg, strlen(msg)) < 0)
            {
                perror("Write err");
                exit(1);
            }
        }
    }

    close(fd[0][WR]);
    close(fd[1][WR]);

    int statusChild1, statusChild2;
    waitpid(pid1, &statusChild1, 0);
    if (WIFEXITED(statusChild1))
    {
        printf("Child 1 exited, returned %d\n", WEXITSTATUS(statusChild1));
    }
    else
    {
        fprintf(stderr, "Something is wrong with 1st child process\n");
    }
    waitpid(pid2, &statusChild2, 0);
    if (WIFEXITED(statusChild1))
    {

```

```

    printf("Child 2 exited, returned %d\n", WEXITSTATUS(statusChild2));
}
else
{
    fprintf(stderr, "Something is wrong with 2nd child process\n");
}

return 0;
}

```

child.c:

```

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <ctype.h>

#define INBUFSIZE 300 // Buffer size

int isVowel(char t)
{
    t = tolower(t);
    if (t == 'a' || t == 'e' || t == 'i' || t == 'o' || t == 'u' || t == 'y')
        return 1;
    return 0;
}

int main(void)
{
    char buff[INBUFSIZE], res[INBUFSIZE];

    pid_t mypid = getpid();
    printf("Child %d: Started!\n", mypid);

    while (fgets(buff, INBUFSIZE, stdin))
    {
        printf("Received line: %s", buff);

        int d = 0;
        for (int i = 0; i < strlen(buff); ++i)
        {
            if (!isVowel(buff[i]))
            {
                res[d++] = buff[i];
            }
        }
        res[d] = '\0';

        printf("Processed line: %s", res);
    }
    printf("Child %d: I'm Done!\n", mypid);

    return 0;
}

```

Вывод

По мере выполнения данной лабораторной работы я закрепил знания о работе с файловыми дескрипторами, улучшил навыки создания новых процессов внутри основной программы, познал суть их коммуникации по неименованным каналам. Также я научился замещать образ процессов с помощью функций семейства `exec` и разобрался с тем, как ожидать завершения работы процессов-потомков.

Многопроцессорность — сила: благодаря ей ОС выполняет разделение памяти и прочих ресурсов между ними и, как следствие:

- а) внезапно упавший процесс не уронит остальные;
- б) если в процессе начал выполняться чужеродный код (например, из-за RCE уязвимости), то он не получит доступ к содержимому памяти в других процессах.

Многопроцессорность сегодня можно увидеть, например, в браузерах, когда отдельные вкладки выполняются в разных процессах, и упавшая вкладка (из-за `js` или из-за кривого плагина) тянет за собой не весь браузер, а только себя или еще пару вкладок.

Еще важно заметить, что многопроцессорные программы часто потребляют большое количество памяти, что не есть хорошо. IPC (межпроцессное взаимодействие) - довольно сложный с большими накладными расходами процесс.