Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет: “Информационные технологии и прикладная математика”
Кафедра: 806 “Вычислительная математика и программирование”

Лабораторная работа №2
по курсу “Операционные системы”

Студент	Полей-Добронравова А.В.
Группа	М8О-207Б-18
Вариант	7
Преподаватель	Миронов Е.С.
Дата	10.03.2020
Оценка	

Москва, 2020

1. Постановка задачи

На вход программе подается название 2-ух файлов. Необходимо отсортировать оба файла (каждый в отдельном процессе) произвольной сортировкой (на усмотрение студента). Содержимое обоих файлов вывести в стандартный поток вывода родительским процессом.

Цель работы

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 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.

2. Решение задачи

Используемые системные вызовы:

- **ssize_t write(int fd, const void* buf, size_t count)** - пишет в файл, на который ссылается файловый дескриптор fd, count байт из буфера buf.
- **ssize_t read(int fd, void* buf, size_t count)** - считывает из файла, на который ссылается файловый дескриптор fd, count байт в буфер, начинающийся по адресу buf. Для файлов, поддерживающих смещения, операция чтения начнется с текущего смещения, потом сдвинется на count байт. Если текущее смещение за концом файла, считывание не произойдет, и **read()** вернет **0**. Если count = 0, **read()** вернет **0**.

- **pid_t fork(void)** - Создаёт новый процесс копированием вызывающего процесса. Новый процесс называется *дочерним*, вызывающий процесс - *родительский*. Находятся в разных пространствах памяти, но имеют одинаковое содержимое сразу после вызова.
- **int close(int fd)** - закрывает файловый дескриптор.
- **int pipe(int* fd[2])** - неименованный канал между процессами. Неименованный канал создается вызовом pipe, который заносит в массив int [2] два дескриптора открытых файлов. fd[0] – открыт на чтение, fd[1] – на запись (вспомните STDIN == 0, STDOUT == 1). Канал уничтожается, когда будут закрыты все файловые дескрипторы ссылающиеся на него.

  В рамках одного процесса pipe смысла не имеет, передать информацию о нем в произвольный процесс нельзя (имени нет, а номера файловых дескрипторов в каждом процессе свои). Единственный способ использовать pipe – унаследовать дескрипторы при вызове fork (и последующем exec). После вызова fork канал окажется открытым на чтение и запись в родительском и дочернем процессе. Т.е. теперь на него будут ссылаться 4 дескриптора. Теперь надо определиться с направлением передачи данных – если надо передавать данные от родителя к потомку, то родитель закрывает дескриптор на чтение, а потомок - дескриптор на запись.

  Оставлять оба дескриптора незакрытыми плохо по двум причинам:

1. Родитель после записи не может узнать считал ли дочерний процесс данные, а если считал то сколько. Соответственно, если родитель попробует читать из pipe, то, вполне вероятно, он считает часть собственных данных, которые станут недоступными для потомка.
2. Если один из процессов завершился или закрыл свои дескрипторы, то второй этого не заметит, так как pipe на его стороне по-прежнему открыт на чтение и на запись.

Если надо организовать двунаправленную передачу данных, то можно создать два pipe.

Файлы сортируются по лексикографическому сравнению строк из файла длиной 5 символов, строки заносятся в массив, сортировка пузырьком.

Демонстрация работы программы:

```
MacBook-Pro-Amelia:Documents amelia$ gcc -o f lab2.c
MacBook-Pro-Amelia:Documents amelia$ ./f lab2os.txt lab2os1.txt
in parent process
 child process 1
heloo
please i want be very cute
so fo gkgr
sooooooooooooooooooooooooooooooooooo
no if lfg
a a aaaaaaaa aaa
f
aa

 child process 2
heloo
please i want be very cute
so fo gkgr
sooooooooooooooooooooooooooooooooooo
no if lfg
a a aaaaaaaa aaa
f
aa

 ended child process 1: 0

 begining of file1
want te
sose i ry cuooooooooooooooooooooooooooooooooooo
nokgr
sheloofg
a be veaaaaaa aaa if l fo g aaa

pleaf
aa
```

a
 end of file1

 ended child process 2: 0

 begining of file2
want te
sose i ry cuoooooooooooooooooooooooooooooooooooo
nokgr
sheloofg
a be veaaaaaa aaa if l fo g aaa

pleaf
aa

a
 end of file2

4. Листинг программы

```
#include <stdio.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <signal.h>

int Sort(int* pipe_fd, const char* argv) // сортировка вставкой
{
   int fp, n = 100;
   if ((fp=open(argv, O_RDONLY)) < 0) {
      printf("Cannot open file.\n");
      printf("%d \n", fp);
      exit(-1);
      return -1;
   }
   char mass[n][6];
   char newElement[6], s[6], ss[6];
   int i = -1;
   close(pipe_fd[0]);
   while(read(fp,&newElement,(sizeof(char)* 5)) > 0) {
      if (i >= n) {
         printf("above limit \n");
         break;
```

```
        }
        i++;
        for (int f = 0; f < 6; f++) {
            mass[i][f] = newElement[f];
            }
    }
    for (int k = 0; k <= i; k++)
    {
        for (int f = 0; f < 6; f++) {
            s[f] = mass[k][f];
        }
        printf("%s", s);
    }
    printf(" \n");
    for (int j = 0; j <= i; j++)
    {
        for (int k = (i - 1); k > j; k--) // для всех элементов после i-ого
        {
            for (int f = 0; f < 6; f++) {
                s[f] = mass[k][f];
                ss[f] = mass[k - 1][f];
            }
           /* printf(" \n");
            printf("%s \n", s);
            printf("%s \n", ss);
            printf(" \n");
            */
            if (strcmp(ss,s) < 0) // если текущий элемент меньше предыдущего
            {
                strcpy(newElement,ss); // меняем их местами
                for (int f = 0; f < 6; f++) {
                    mass[k - 1][f] = mass[k][f];
                }
                for (int f = 0; f < 6; f++) {
                    mass[k][f] = newElement[f];
                }
            }
        }
    }
    printf(" \n");
    close(fp);
    close(pipe_fd[0]);
    for(int j = 0; j <= i; j++) {
        write(pipe_fd[1],mass[j],sizeof(char) * 6);
    }

    close(pipe_fd[1]);
    return 0;
}

int main(int argc, const char * argv[]) {
    pid_t pid1, pid2;
    char s[6],ss[6];
    int status1, status2;
```

```
int pipe_fd1[2];
int pipe_fd2[2];
if (pipe(pipe_fd1) == -1) {
    perror("pipe");
}
if (pipe(pipe_fd2) == -1) {
    perror("pipe");
}
pid1 = fork();
if (pid1 == 0) { // child process 1
    printf(" child process 1 \n");
    exit(Sort(pipe_fd1, argv[1]));
} else if (pid1 < 0) {
    perror("fork");
} else {
    printf("in parent process \n");
    pid2 = fork();
    if (pid2 == 0) { // child process 2
        printf(" child process 2 \n");
        exit(Sort(pipe_fd2, argv[2]));
    }
    else if (pid2 < 0) {
        perror("fork");
    }
    if (waitpid(pid1, &status1, 0) == -1) {
        perror("waitpid");
    }
    else {
        printf("\n ended child process 1: %d \n", status1);
        printf("\n begining of file1 \n");
        close(pipe_fd1[1]);
        while (read(pipe_fd1[0], &s, sizeof(char) * 5) > 0)
        {
            for (int f = 0; f < 6; f++) {
                printf("%c", s[f]);
            }
        }
        printf("\n end of file1 \n");
        close(pipe_fd1[0]);
    }
    if (waitpid(pid2, &status2, 0) == -1) {
        perror("waitpid");
    }
    else {
        printf("\n ended child process 2: %d \n", status2);
        printf("\n begining of file2 \n");
        close(pipe_fd2[1]);
        while (read(pipe_fd2[0], &ss, sizeof(char) * 5) > 0)
        {
            for (int f = 0; f < 6; f++) {
                printf("%c", ss[f]);
            }
        }
        printf("\n end of file2 \n");
```

```
            close(pipe_fd2[0]);
        }
    }
    return 0;
}
```

5. 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.