



Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ _____ «Информатика и системы управления»

КАФЕДРА _____ «Программное обеспечение ЭВМ и информационные технологии»

Лабораторная работа № 5

Дисциплина Операционные системы

Тема Буферизованный и не буферизованный ввод-вывод

Студент Ильясов И. М.

Группа ИУ7-63Б

Оценка (баллы) _____

Преподаватель Рязанова Н. Ю.

Москва, 2020 г.

Задание.

В лабораторной работе анализируется результат выполнения трех программ. Программы демонстрируют открытие одного и того же файла несколько раз. Реализация открытия файла в одной программе несколько раз выбрана для простоты. Такая ситуация возможна в системе, когда один и тот же файл несколько раз открывают разные процессы. Но для получения ситуаций аналогичных тем, которые демонстрируют приведенные программы надо было бы синхронизировать работу процессов. При выполнении асинхронных процессов такая ситуация вероятна и ее надо учитывать, чтобы избежать потери данных или получения неверного результата при выводе в файл.

Структура FILE:

```
struct _IO_FILE {
    int _flags;      /* High-order word is _IO_MAGIC; rest is flags. */
    #define _IO_file_flags _flags

    /* The following pointers correspond to the C++ streambuf protocol. */
    /* Note: Tk uses the _IO_read_ptr and _IO_read_end fields directly. */
    char* _IO_read_ptr; /* Current read pointer */
    char* _IO_read_end; /* End of get area. */
    char* _IO_read_base; /* Start of putback+get area. */
    char* _IO_write_base; /* Start of put area. */
    char* _IO_write_ptr; /* Current put pointer. */
    char* _IO_write_end; /* End of put area. */
    char* _IO_buf_base; /* Start of reserve area. */
    char* _IO_buf_end; /* End of reserve area. */
    /* The following fields are used to support backing up and undo. */
    char* _IO_save_base; /* Pointer to start of non-current get area. */
    char* _IO_backup_base; /* Pointer to first valid character of backup area */
    char* _IO_save_end; /* Pointer to end of non-current get area. */

    struct _IO_marker *_markers;

    struct _IO_FILE *_chain;

    int _fileno;
    #if 0
    int _blksize;
    #else
    int _flags2;
    #endif
    _IO_off_t _old_offset; /* This used to be _offset but it's too small. */

    #define __HAVE_COLUMN /* temporary */
    /* 1+column number of pbase[]; 0 is unknown. */
    unsigned short _cur_column;
    signed char _vtable_offset;
    char _shortbuf[1];

    /* char* _save_gptr; char* _save_egptr; */

    _IO_lock_t *_lock;
    #ifdef _IO_USE_OLD_IO_FILE
};
```

Программа №1:

```
//testCIO.c
#include <stdio.h>
#include <fcntl.h>

/*
On my machine, a buffer size of 20 bytes
translated into a 12-character buffer.
Apparently 8 bytes were used up by the
stdio library for bookkeeping.
*/

int main()
{
    // have kernel open connection to file alphabet.txt
    int fd = open("alphabet.txt", O_RDONLY);

    // create two a C I/O buffered streams using the above connection
    // associates a stream with the existing file descriptor
    FILE *fs1 = fdopen(fd, "r");
    char buff1[20];
    setvbuf(fs1, buff1, _IOFBF, 20); // set fully buffering

    FILE *fs2 = fdopen(fd, "r");
    char buff2[20];
    setvbuf(fs2, buff2, _IOFBF, 20);

    // read a char and write it alternatingly from fs1 and fs2
    int flag1 = 1, flag2 = 2;
    while (flag1 == 1 || flag2 == 1)
    {
        char c;
        flag1 = fscanf(fs1, "%c", &c);

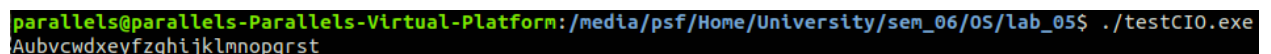
        if (flag1 == 1)
        {
            fprintf(stdout, "%c", c);
        }

        flag2 = fscanf(fs2, "%c", &c);

        if (flag2 == 1)
        {
            fprintf(stdout, "%c", c);
        }
    }

    return 0;
}
```

Ниже на рисунке №1 приведен скриншот **результата выполнения** первой программы.



```
parallels@parallels-Parallels-Virtual-Platform: /media/psf/Home/University/sem_06/05/lab_05$ ./testCIO.exe
Aubvcwdxeyfzghijklmnopqrst
```

Рисунок 1. Результат выполнения первой программы.

Анализ полученного результата:

С помощью системного вызова `open()` создается дескриптор файла, файл открывается только на чтение (передается флаг `O_RDONLY`), указатель устанавливается на начало файла. Если системный вызов завершается успешно, в результате этого вызова

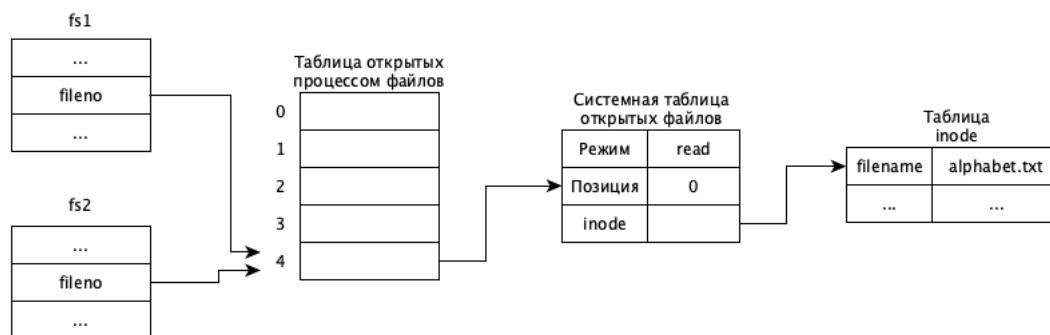
появляется новый дескриптор открытого файла alphabet.txt, не разделяемый никакими процессами, и запись в системной таблице открытых файлов.

Далее функция `fdopen()` связывает два объекта типа `FILE` с существующим дескриптором файла. При создании буфера размер для потока выбирается системой. Для его изменения используется системный вызов `setvbuf()`. Здесь системный вызов `setvbuf()` изменяет тип буферизации на полную размером в 20 байт.

При первом вызове `fscanf()` буфер структуры `FILE` либо заполняется полностью, либо пока не будет достигнут конец файла. Так как размер буфера равен 20 байт, а в файле хранится информация размером 26 байт (26 букв латинского алфавита по одному байту), то после первого вызова `fscanf()` в буфере первой структуры `FILE` будут находиться первые 20 байт файла (Abcdefghijklmnopqrst), а в буфере второй – uvwxуз.

Затем в стандартный поток вывода `stdout` будет поочередно выводиться по одному символу из каждого буфера. Когда второй буфер опустеет, из первого буфера будут выводиться оставшиеся символы. Получим результат, приведенный выше на скриншоте.

Связь структур:



Программа №2:

```
//testKernelIO.c
#include <fcntl.h>
#include <unistd.h>

int main()
{
    // have kernel open two connection to file alphabet.txt
    int fd1 = open("alphabet.txt", O_RDONLY);
    int fd2 = open("alphabet.txt", O_RDONLY);

    // read a char & write it alternatingly from connections fs1 & fd2
    while(1)
    {
        char c;
        if (read(fd1, &c, 1) != 1) break;
        write(1, &c, 1);
        if (read(fd2, &c, 1) != 1) break;
        write(1, &c, 1);
    }
    write(1, "\n", 1);

    return 0;
}
```

Ниже на рисунке №2 приведен скриншот **результата выполнения** второй программы.

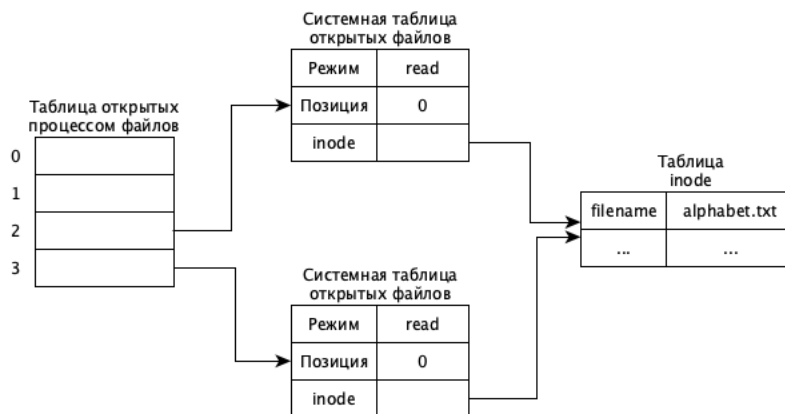
```
parallels@parallels-Parallels-Virtual-Platform:/media/psf/Home/University/sem_06/05/lab_05$ ./testKernelIO.exe
AAbbccddeeffgghhijjkkllmmnnoppqqrrssttuuvvwwxxyyzz
```

Рисунок 2. Результат выполнения второй программы.

Анализ полученного результата:

В отличие от первой программы, в данной программе файл дважды открывается для чтения с помощью системного вызова `open()`, создаются два файловых дескриптора, две разные записи в системной таблице открытых файлов, которые связаны с одним и тем же физическим файлом. Файловые дескрипторы независимы друг от друга, поэтому положения указателей в файле, связанных с данным файловым дескриптором, будут независимы. В результате прохода цикла, где с помощью системных вызовов `read`, `write` в `stdout` выводится по одному символу из файла, увидим строку с дублирующимися буквами.

Связь структур:



Программа №3:

```
#include <stdio.h>
int main() {

    FILE* fd[2];
    fd[0] = fopen("testFOpen_output.txt", "w");
    fd[1] = fopen("testFOpen_output.txt", "w");

    int curr = 0;

    for(char c = 'a'; c <= 'z'; c++, curr = ((curr != 0) ? 0 : 1))
    {
        fprintf(fd[curr], "%c", c);
    }
    fprintf(fd[curr], "\n");
    fclose(fd[0]);
    fclose(fd[1]);
    return 0;
}
```

Ниже на рисунке №3 приведен скриншот **результата выполнения** второй программы.

```
parallels@parallels-Parallels-Virtual-Platform: /media/psf/Home/University/sem_06/OS/lab_05$ cat testFOpen_output.txt  
bdfhjlnprtvxz
```

Рисунок 3. Результат выполнения третьей программы.

Анализ полученного результата:

В данной программе с помощью функции `fork()` открываем два потока на запись с начала файла `alphabet.txt`. Они имеют два разных файловых дескриптора и следовательно независимые позиции в файле. Затем в цикле с помощью `fprintf()` в потоки поочередно записываются буквы от `a` до `z`, при этом нечетные в первый поток, четные во второй. Следует помнить о том, что функция `fprintf()` обеспечивает буферизацию. Запись непосредственно в сам файл происходит в трех случаях: либо при полном заполнении буфера, либо при вызове функций `fclose()` и `fflush()`. Функция `fclose()` отделяет указанный поток от связанного с ним файла или набора функций. Если поток использовался для вывода данных, то все данные, содержащиеся в буфере, сначала записываются с помощью `fflush()`. Функция `fflush()` принудительно записывает все буферизированные данные в устройство вывода данных. При этом поток остается открытым. Так как оба потока открыты в режиме перезаписи в файл, то после выполнения второго `fclose()` данные в файле, записанные с помощью первого потока, будут переписаны и мы увидим там буквы, стоящие на нечетных местах.

Связь структур:



Вывод:

При буферизированном вводе-выводе необходимо учитывать факт записи в буфер и чтения данных из буфера, так как неправильные действия с данными могут привести к потере данных или к неверной последовательности данных. Необходимо также учитывать то, что при одновременном открытии одного и того же файла создается дескриптор открытого файла. Каждый дескриптор файла имеет указатель на позицию чтения или записи в логическом файле.