**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»

# ОТЧЁТ

По лабораторной работе №5

По курсу: «Операционные системы»

Тема: «Буферизованный и небуферизованный ввод-вывод»

**Студент:** Керимов А. Ш.

**Группа:** ИУ7-64Б

**Преподаватель:** Рязанова Н. Ю.

Москва

2020

**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.

# Программа 1

Листинг 1: testCIO.c

```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
        FILE *fs1 = fdopen(fd, "r");
        char buff1[20];
        setvbuf(fs1, buff1, _IOFBF, 20);

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

        // read a char & 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;
}
```


Рис. 1: Демонстрация работы программы `testCIO.c`

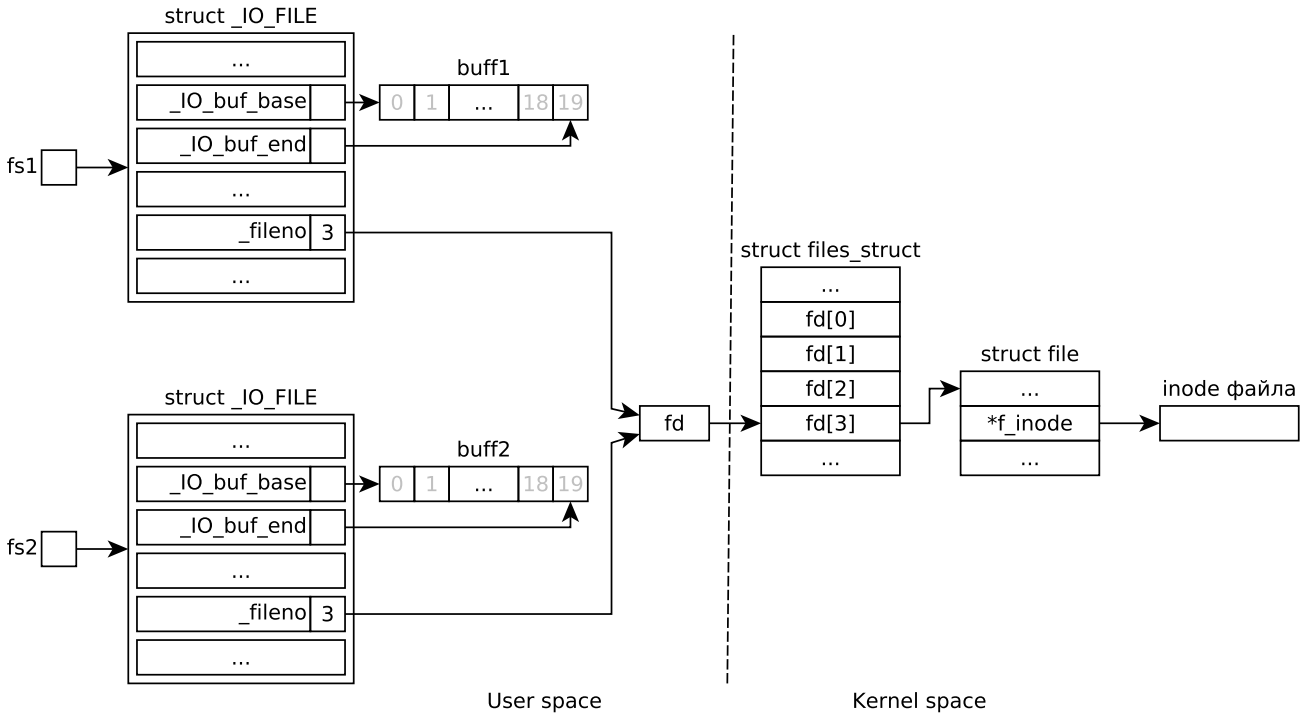


Рис. 2: Связь дескрипторов в `testCIO.c`

Системный вызов `open()` создаёт новый файловый дескриптор для открытого только на чтение (`O_RDONLY`) файла `alphabet.txt`, запись в системной таблице открытых файлов. Эта запись регистрирует смещение в файле и флаги состояния файла. Новый файловый дескриптор является наименьшим дескриптором, который ещё не открыт процессом (`3`). Функция `fdopen()` связывает потоки `fs1` и `fs2` с дескриптором `fd`. Функция `setvbuf()` устанавливает блочную буферизации (`_IOFBF`) размером 20 байт для каждого из потоков `fs1` и `fs2`.

Размер файла `alphabet.txt` составляет 27 байтов (26 букв английского алфавита и символ новой строки), в буфер потока `fs1` поместятся первые 20 байтов (`"Abcdefghijklmopqrst"`), в буфер потока `fs2` — оставшиеся 7 байтов (`"uvwxyz\n"`).

В цикле чередуется вывод из потоков `fs1` и `fs2`. Поскольку в буфере второго потока меньше символов, после последнего символа `'\n'` выведутся оставшиеся в буфере первого потока символы. Результат — строка `"Aubvcwdxeyfzg\nhijklmnopqrst"`.

# Программа 2

Листинг 2: testKernelIO.c

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

int main()
{
        char c;
        // 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
        for (ssize_t flag1 = 1, flag2 = 1; flag1 && flag2;) {
                if ((flag1 = read(fd1, &c, 1)) == 1) {
                        write(1, &c, 1);
                        if ((flag2 = read(fd2, &c, 1)) == 1) {
                                write(1, &c, 1);
```

```
                        }
                }
        }

        return 0;
}
```

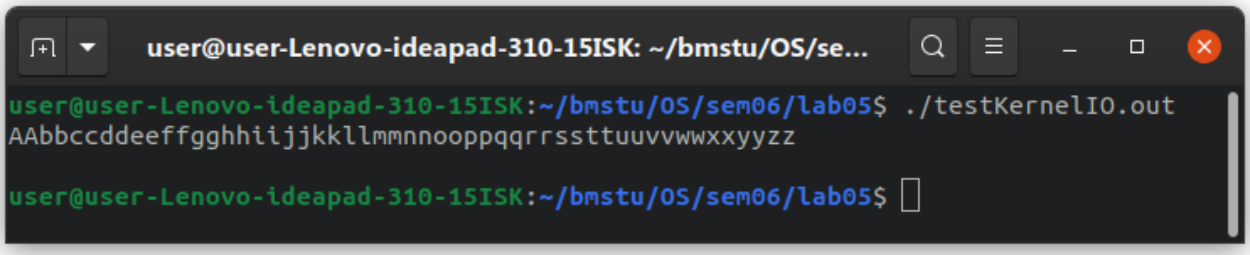


Рис. 3: Демонстрация работы программы `testKernelIO.c`

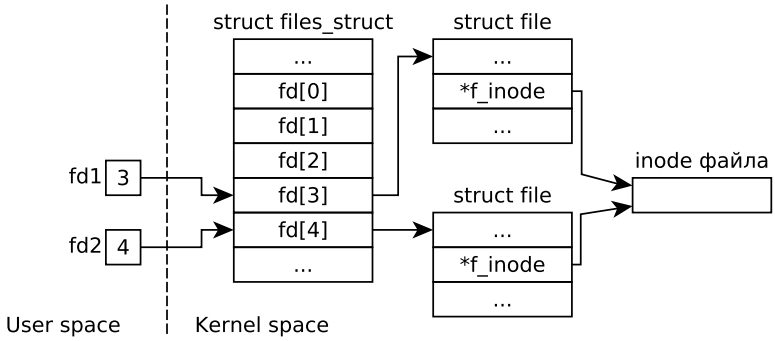


Рис. 4: Связь дескрипторов в `testKernelIO.c`

Файл `alphabet.txt` открывается системным вызовом `open()` только на чтение (`O_RDONLY`) дважды, в системной таблице открытых файлов создаются две новых записи, дескрипторы помещаются в переменные `fd1` и `fd2` (хоть и описывают один файл, являются различными). Смещения в файловых дескрипторах независимы, поэтому в цикле на экран выводится каждый символ из файла дважды. Результат — строка `"AAbbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz\n\n"`.

# Программа 3

Листинг 3: testCO.c

```
#include <stdio.h>

int main()
{
        FILE *f1 = fopen("alphabet_out.txt", "w");
        FILE *f2 = fopen("alphabet_out.txt", "w");

        for (char c = 'a'; c <= 'z'; ++c) {
                if (c % 2) {
                        fprintf(f1, "%c", c);
                } else {
```

```
                        fprintf(f2, "%c", c);
                }
        }

        fclose(f1);
        fclose(f2);

        return 0;
}
```

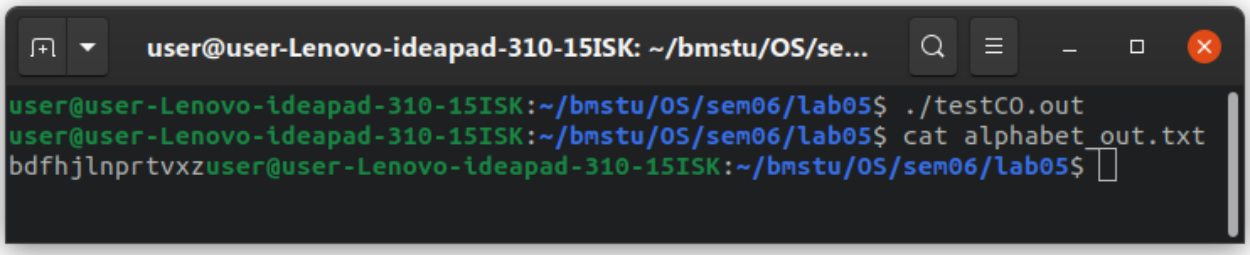


Рис. 5: Демонстрация работы программы `testCO.c`

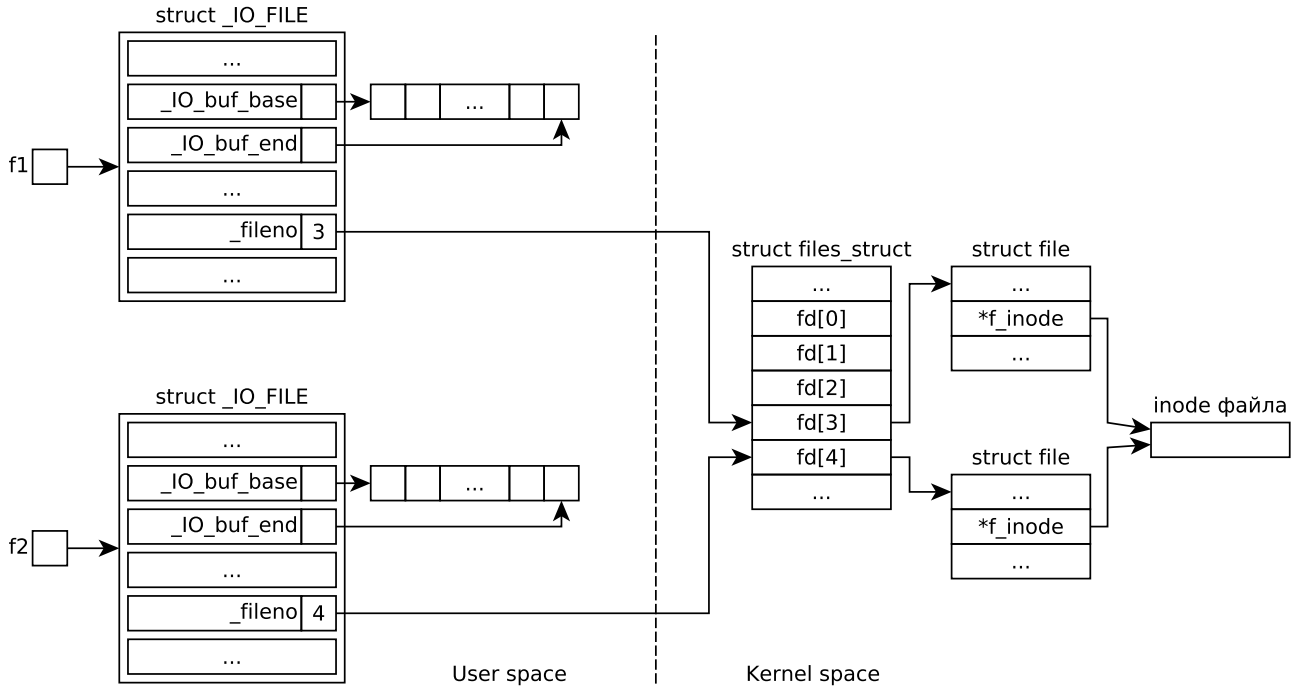


Рис. 6: Связь дескрипторов в `testCO.c`

Файл `alphabet_out.txt` открывается функцией `fopen()` на запись дважды, в два различных потока `fs1` и `fs2`.

В цикле символы, имеющие нечётный код в таблице `ASCII`, записываются в буфер потока `fs1`, чётные — в буфер потока `fs2`. После закрытия потока `fs1`, его буфер записывается в файл. Смещения в потоках независимы, поэтому после закрытия потока `fs2`, его буфер записывается в файл опять с начала. Результат — строка `"bdfhjlnprtvxz"`.

## Структура FILE

Листинг 4: FILE.h

```
/* The tag name of this struct is _IO_FILE to preserve historic
   C++ mangled names for functions taking FILE* arguments.
   That name should not be used in new code.  */
struct _IO_FILE
{
        int _flags;             /* High-order word is _IO_MAGIC; rest is flags. */

        /* The following pointers correspond to the C++ streambuf protocol. */
        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;
        int _flags2;
        __off_t _old_offset; /* This used to be _offset but it's too small.  */

        /* 1+column number of pbase(); 0 is unknown. */
        unsigned short _cur_column;
        signed char _vtable_offset;
        char _shortbuf[1];

        _IO_lock_t *_lock;
#ifdef _IO_USE_OLD_IO_FILE
};

struct _IO_FILE_complete
{
        struct _IO_FILE _file;
#endif
        __off64_t _offset;
        /* Wide character stream stuff.  */
        struct _IO_codecvt *_codecvt;
        struct _IO_wide_data *_wide_data;
        struct _IO_FILE *_freeres_list;
        void *_freeres_buf;
        size_t __pad5;
        int _mode;
        /* Make sure we don't get into trouble again.  */
        char _unused2[15 * sizeof (int) - 4 * sizeof (void *) - sizeof (size_t)];
};

/* The opaque type of streams.  This is the definition used elsewhere.  */
typedef struct _IO_FILE FILE;
```

# Заключение

Открытые файлы, для которых используется ввод/вывод потоков, могут буферизоваться. Открытие одного и того же файла каждый раз создаёт новый файловый дескриптор и новую запись в системной таблице открытых файлов. Эта запись регистрирует смещение в файле и флаги состояния файла. Поэтому у различных дескрипторов открытого файла смещения не зависят друг от друга.

Чтобы избежать потери данных или получения неверного результата при буферизованном выводе в файл, необходимо учитывать то, что файл может быть открыт несколько раз, а также помнить о своевременном выполнении `fclose()` и `fflush()`.