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»

Дисциплина: «Операционные системы»

Лабораторная работа №5

Тема работы: «Буферизованный и небуферизованный ввод-вывод»

Студент: Левушкин И. К.
Группа: ИУ7-62Б
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.

1 программа

```c
//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;
}
```

Результат работы:

Листинг 1: Результат testCIO.out

```
1  Aubvcwdxeyfzghijklmnopqrst
```

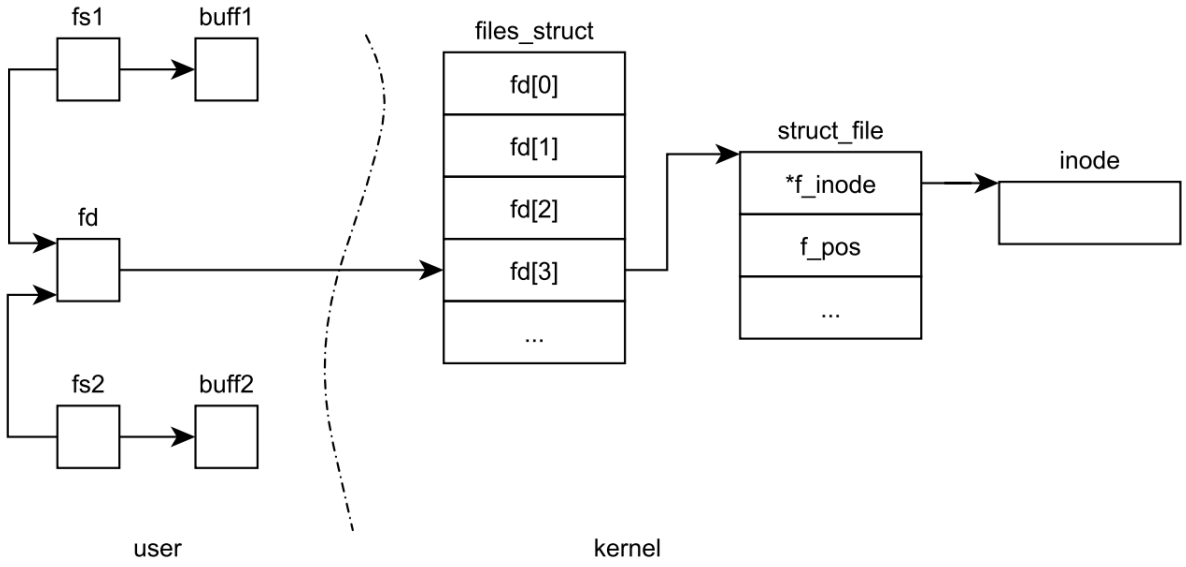

Рис. 1: Связь дескрипторов в программе 1

С помощью системного вызова *open()* создается дискриптор файла, при этом файл открывается только на чтение (флаг O_RDONLY), указатель устанавливается на начало файла. Если системный вызов завершается успешно, возвращенный файловый дескриптор является наименьшим, который еще не открыт процессом. В результате этого вызова появляется новый открытый файл, не разделяемый никакими процессами, и запись в системной таблице открытых файлов.

Затем с помощью функции *fdopen()* создаются два потока данных *fs1* и *fs2*, которые связываются с файлом, описанным дескриптором *fd*. Функция *setvbuf()* изменяет тип буферизации на блочную (полную) размером в 20 байт.

В цикле осуществляется чтение из потоков (функция *fscanf()*) и вывод в *stdout* (функция *fprintf()*). Флаги *flag1* и *flag2* изменят свое значение 3 с 1 на -1 тогда, когда число прочитанных символов станет равно нулю. Вследствие буферизации в *buff1* фактически помещается строка *Abcdefghijklmnopqrst* – первые 20 символов, а в *buff2* – *uvwxyz* – оставшаяся часть (при первом чтении указатель в файле сместился на 20 символов). Отсюда получаем соответствующий результат (листинг 1).

2 программа

```c
//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);

  char c;

  // read a char & write it alternatingly from connections fs1 & fd2
  int flag1 = 1, flag2 = 2;
  while(flag1 == 1 || flag2 == 1)
  {
    flag1 = read(fd1,&c,1);
    if (flag1 == 1)
    {
        write(1,&c,1);
    }
    flag2 = read(fd2,&c,1);
    if (flag2 == 1)
    {
        write(1,&c,1);
    }
  }

  return 0;
}
```

Результат работы программы:

Листинг 2: Результат testKernelIO.out

```
AAbbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz
```

Здесь создаются два файловых дескриптора и две разные записи в системной таблице открытых файлов. Файловые дескрипторы независимы друг от друга, поэтому положения указателей в файле также независимы.

В цикле с помощью системных вызовов *read()* и *write()* посимвольно происходит чтение из файла и запись в стандартный поток вывода соответственно. Поэтому получаем строку с дублирующимися буквами (листинг 2).

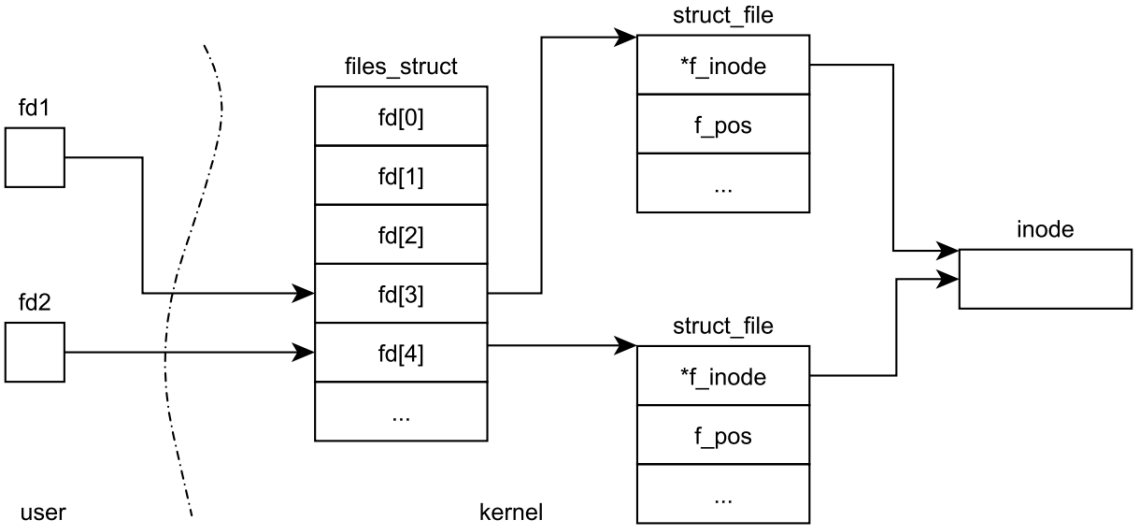

Рис. 2: Связь дескрипторов в программе 2

3 программа

```c
#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);
  }
  fclose(fd[0]);
  fclose(fd[1]);
  return 0;
}
```

Результат работы программы:

Листинг 3: Результат testFopen.out

```
1  bdfhjlnprtvxz
```

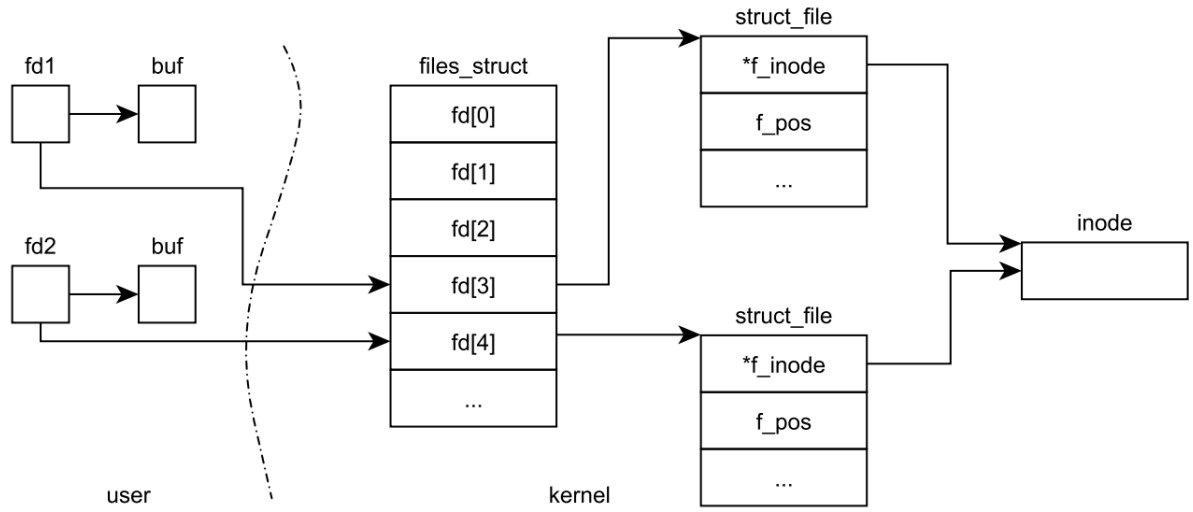

Рис. 3: Связь дескрипторов в программе 3

С помощью функций *fopen()* открываются два потока на запись с начала файла. Они имеют два разных файловых дескриптора, а значит и независимые позиции в файле.

В цикле с помощью *fprintf()* в потоки поочередно записываются буквы от *a* до *z* (нечетные – в первый поток, четные – во второй). Как было отмечено ранее, функция *fprintf()* обеспечивает буферизацию. Запись в файл происходит

либо при полном заполнении буфера, либо при вызове функций *fclose()* или *fflush()*.

Таким образом, когда в программе происходит вызов функции *fclose(fd1)*, в файл записываются данные первого потока, однако далее после вызова функции *fclose(fd2)* содержимое файла удаляется (ранее был установлен режим записи) и записывается информация из буфера второго потока.

Структура FILE

Листинг 4: Структура FILE

```
1  // "bits/types/FILE.h"
2
3  #ifndef __FILE_defined
4  #define __FILE_defined 1
5
6  struct _IO_FILE;
7
8  /*
9    The opaque type of streams. This is the definition used elsewhere.
10 */
11
12 typedef struct _IO_FILE FILE;
13
14 #endif
15
16 ...
17
18 // "bits/libio.h"
19
20 struct _IO_FILE
21 {
22   int _flags;                /* High-order word is _IO_MAGIC; rest is
       flags. */
23   /* The following pointers correspond to the C++ streambuf protocol. */
24   char *_IO_read_ptr;        /* Current read pointer */
25   char *_IO_read_end;        /* End of get area. */
26   char *_IO_read_base;        /* Start of putback+get area. */
27   char *_IO_write_base;        /* Start of put area. */
28   char *_IO_write_ptr;        /* Current put pointer. */
29   char *_IO_write_end;        /* End of put area. */
30   char *_IO_buf_base;        /* Start of reserve area. */
31   char *_IO_buf_end;        /* End of reserve area. */
32   /* The following fields are used to support backing up and undo. */
33   char *_IO_save_base; /* Pointer to start of non-current get area. */
34   char *_IO_backup_base;  /* Pointer to first valid character of backup
       area */
35   char *_IO_save_end; /* Pointer to end of non-current get area. */
36   struct _IO_marker *_markers;
37   struct _IO_FILE *_chain;
38   int _fileno;
39   int _flags2;
```

```
40      __off_t _old_offset; /* This used to be _offset but it's too small.  */
41      /* 1+column number of pbase(); 0 is unknown. */
42      unsigned short _cur_column;
43      signed char _vtable_offset;
44      char _shortbuf[1];
45      _IO_lock_t *_lock;
46      #ifdef _IO_USE_OLD_IO_FILE
47    };
```

Вывод

Из проделанной работы можно сделать следующие выводы:

1. При буферизованном вводе (выводе) необходимо учитывать, что запись (чтение) происходит именно из буфера. Поэтому неправильные действия могут привести к неверной обработке данных или даже к потере информации, что и было продемонстрировано в 1 и 3 программах. В 3 программе запись в файл осуществлялась с помощью функции fclose(), в результате чего при повторном ее вызове для второго потока содержимое файла, записанное первым потоком, удалялось и записывалась информация из буфера второго потока. В следствие чего терялась ранее записанная информация.

2. При небуферизованном вводе (выводе) стоит учесть, что при одновременном открытии одного и того же файла создается столько дескрипторов и записей в таблице открытых файлов, сколько раз был открыт файл. Каждый дескриптор *struct_file* имеет поле *f_pos*, указывающее на позицию чтения (записи) в файле, независящую от других дескрипторов. Это и позволяет процессам обрабатывать файл независимо друг от друга. Таким образом, чтение из файла будет происходить нормально, но при попытке одновременно писать в один файл при небуферизованном вводе (выводе) будет получаться неправильный результат, смешанный из того, что хотели напечатать. Это и было продемонстрировано в 2 программе:

   хотели напечатать: *Abcdefghijklmnopqrstuvwxyz**Abcdefghijklmnopqrstuvwxyz*,

   получили: *AAbbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz*.

3. Поэтому необходимо четко понимать разницу между функциями из стандартной библиотеки C, обеспечивающих буферизованный ввод-вывод и системными вызовами с небуферизованным вводом-выводом.