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
По курсу «Операционные системы»

Тема Буферизованный и не буферизованный ввод-вывод

Студент Неклепаева А. Н.

Группа ИУ7-63б

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.

Первая программа:

```
1  //testCIO.c
2  #include <stdio.h>
3  #include <fcntl.h>
4
5  /*
6  On my machine, a buffer size of 20 bytes
7  translated into a 12-character buffer.
8  Apparently 8 bytes were used up by the
9  stdio library for bookkeeping.
10  */
11
12 int main()
13 {
14   // have kernel open connection to file alphabet.txt
15   int fd = open("alphabet.txt",O_RDONLY);
16
17   // create two a C I/O buffered streams using the above connection
18   FILE *fs1 = fdopen(fd,"r");
19   char buff1[20];
20   setvbuf(fs1,buff1,_IOFBF,20);
21
22   FILE *fs2 = fdopen(fd,"r");
23   char buff2[20];
24   setvbuf(fs2,buff2,_IOFBF,20);
25
26   // read a char & write it alternatingly from fs1 and fs2
27   int flag1 = 1, flag2 = 2;
28   while(flag1 == 1 || flag2 == 1)
29   {
```

```
30          char c;
31          flag1 = fscanf(fs1,"%c",&c);
32          if (flag1 == 1)
33          {
34              fprintf(stdout,"%c",c);
35          }
36          flag2 = fscanf(fs2,"%c",&c);
37          if (flag2 == 1)
38          {
39              fprintf(stdout,"%c",c);
40          }
41      }
42      return 0;
43  }
```

```
anastasia@anastasia-Swift-SF314-54G:~/bmstu/sem_6/os/lab_05/1$ ./a
Aubvcwdxeyfzg
hijklmnopqrstanastasia@anastasia-Swift-SF314-54G:~/bmstu/sem_6/os/lab_05/1$
```

Рис. 1: Результат выполнения первой программы

Анализ полученного результата:

Работа с содержимым файла происходит через целочисленный файловый дескриптор, который представляет из себя номер строки в таблице ссылок на открытые файлы процесса. При помощи системного вызова open() создается файловый дескриптор fd, файл открывается на чтение, указатель на текущую позицию в файле устанавливается на начало файла. Если системный вызов завершается успешно, возвращенный файловый дескриптор является самым маленьким дескриптором, который еще не открыт процессом. Возвращается -1 в случае ошибки. Функция fdopen связывает два потока на чтение с существующим файловым дескриптором fd. Функция setvbuf задает блочную буферизацию с размером буфера 20 байт. _IOFBF - полная буферизация, то есть данные будут буферизироваться, пока буфер не заполниться полностью. В цикле данные считываются из двух потоков fs1 и fs2 в стандартный поток вывода stdout. Так как открытые файлы, для которых используется ввод/вывод потоков, буферизуются и размер буфера 20 байт, то в поток fs1 будут считаны первые 20 символов и указатель на текущую позицию в файле будет смещён на 20. В поток fs2 будут считаны оставшиеся 6 символов и символ конца строки. Осуществляется поочередное чтение из двух потоков, через 7 итераций все данные из второго буфера будут

считаны и будут выводиться только символы из первого буфера (символы hijklmnopqrst).

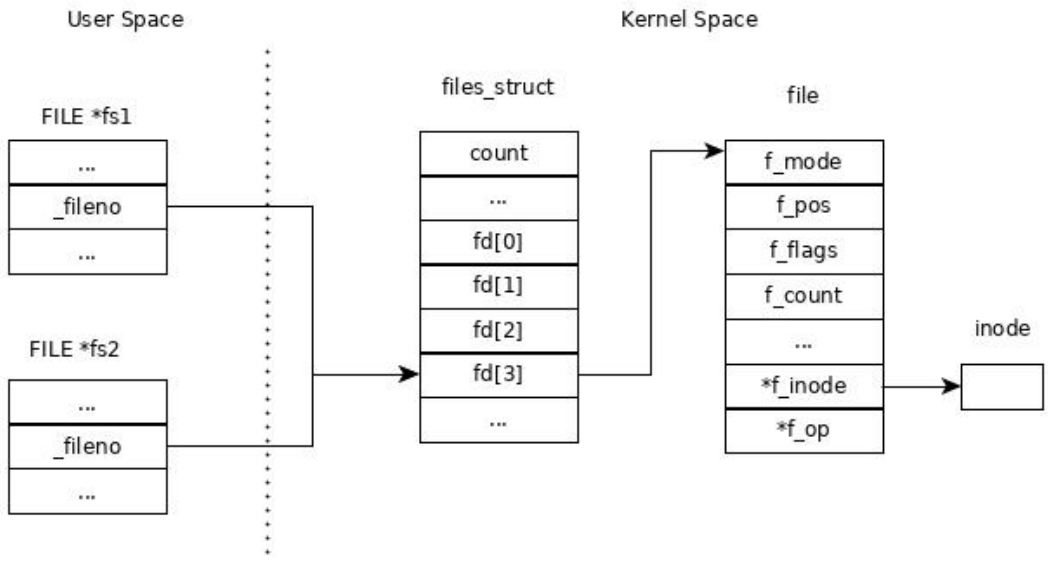

Рис. 2: Рисунок, демонстрирующий созданные дескрипторы и связь между ними

Вторая программа:

```
1  //testKernelIO.c
2  #include <fcntl.h>
3  #include <unistd.h>
4
5  int main()
6  {
7      char c;
8      // have kernel open two connection to file alphabet.txt
9      int fd1 = open("alphabet.txt",O_RDONLY);
10     int fd2 = open("alphabet.txt",O_RDONLY);
11     // read a char & write it alternatingly from connections fs1 &
           fd2
12     int fl = 1;
13
14     while(fl)
```

```
15      {
16          fl = 0;
17          if (read(fd1,&c,1))
18          {
19              write(1,&c,1);
20              if (read(fd2,&c,1))
21              {
22                  write(1,&c,1);
23                  fl = 1;
24              }
25          }
26      }
27      return 0;
28  }
```

```
anastasia@anastasia-Swift-SF314-54G:~/bmstu/sem_6/os/lab_05/2$ ./a
AAbbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz
```

Рис. 3: Результат выполнения второй программы

Анализ полученного результата:

В данной программе с помощью системного вызова open() создаются два файловых дескриптора одного и того же файла, открытого на чтение, то есть создаются две разные записи в системной таблице открытых файлов. Каждая запись будет иметь свою текущую позицию в файле, то есть положения указателей в файле независимы друг от друга. Системные вызовы read(), write() предоставляют небуферизованный ввод-вывод. Ввод-вывод является небуферизованным в том смысле, что каждый вызов read, write делает системный вызов в ядро ОС. В цикле при помощи системных вызовов read() и write() считывается символ из файла и этот символ записывается два раза в stdout, так как указатели в файле независимы друг от друга.

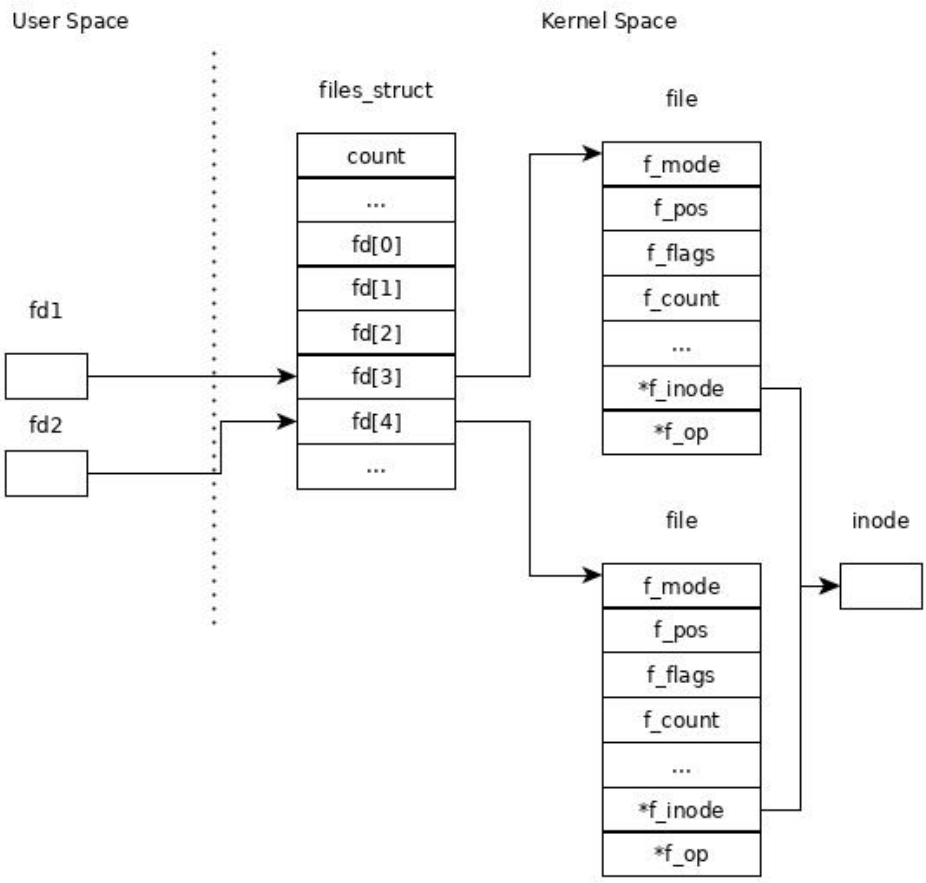

Рис. 4: Рисунок, демонстрирующий созданные дескрипторы и связь между ними

Третья программа:

```
#include <fcntl.h>
#include <stdio.h>

int main()
{
    FILE *fs1 = fopen("q.txt", "w");
    FILE *fs2 = fopen("q.txt", "w");

    for(char c = 'a'; c <= 'z'; c++)
    {
        if (c%2)
```

```
12          {
13              fprintf(fs1, &c, 1);
14          }
15          else
16          {
17              fprintf(fs2, &c, 1);
18          }
19      }
20      fclose(fs1);
21      fclose(fs2);
22      return 0;
23  }
```

```
anastasia@anastasia-Swift-SF314-54G:~/bmstu/sem_6/os/lab_05/3$ ./a
anastasia@anastasia-Swift-SF314-54G:~/bmstu/sem_6/os/lab_05/3$ cat q.txt
bdfhjlnprtvxzanastasia@anastasia-Swift-SF314-54G:~/bmstu/sem_6/os/lab_05/3$
```

Рис. 5: Результат выполнения третьей программы

Анализ полученного результата:

С помощью fopen() открываются два потока на запись, которые имеют разные файловые дескрипторы. Нечётные буквы алфавита записываются в первый поток fs1, чётные - во второй fs2. Так как функция fopen() выполняет ввод-вывод с буферизацией, окончательная запись в файл осуществляется либо при полном заполнении буфера, либо при вызове функций fclose() или функции fflush(). Если поток использовался для вывода данных, то после вызова fclose() все данные, содержащиеся в буфере, записываются в файл с помощью fflush().Так как поток fs1 использовался для вывода данных, после вызова fclose(fs1) все данные записываются из буфера в файл. При этом поток остается открытым. Так как используются два различных файловых дескриптора, то их указатели на текущую позицию в файле независимы, поэтому при вызове fclose(fs2) данные из второго буфера записываются в файл с начальной позиции, таким образом затирая данные записанные ранее из первого буфера. В итоге в файл будут записаны чётные буквы алфавита - bdfhjlnprtvxz.

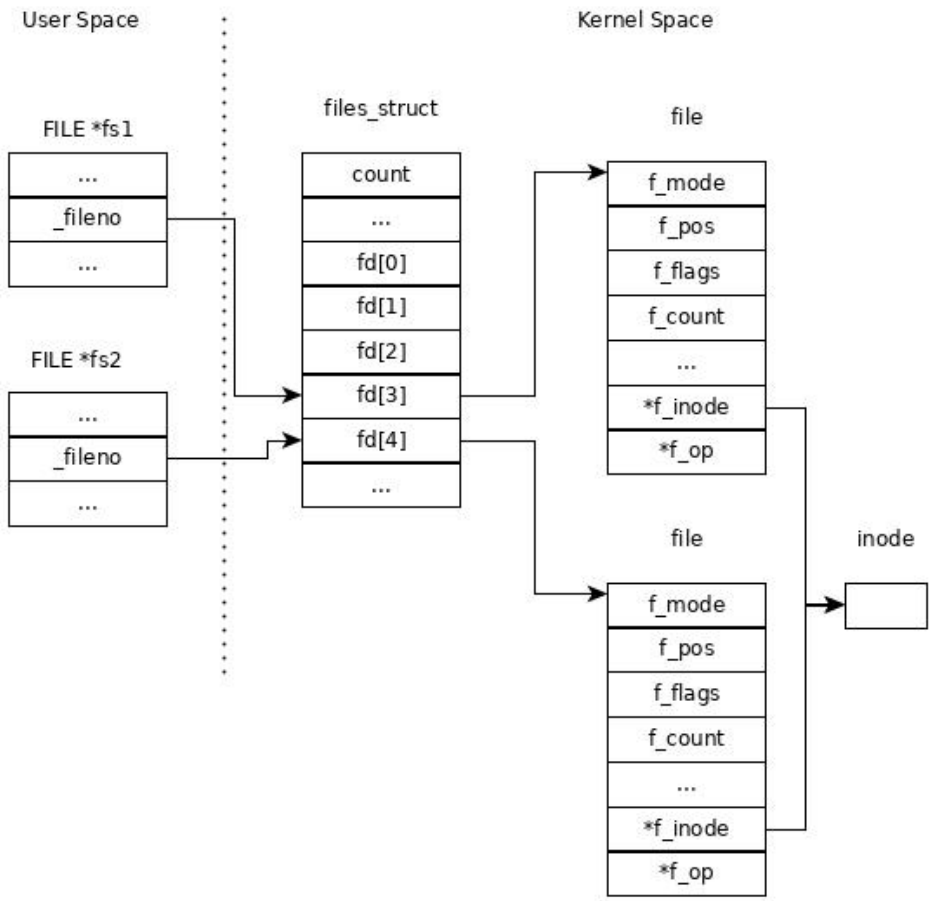

Рис. 6: Рисунок, демонстрирующий созданные дескрипторы и связь между ними

Структура FILE

```
1 struct _IO_FILE {
2   int _flags;  /* High-order word is _IO_MAGIC; rest is flags. */
3 #define _IO_file_flags _flags
4
5   /* The following pointers correspond to the C++ streambuf
       protocol. */
6   /* Note: Tk uses the _IO_read_ptr and _IO_read_end fields
       directly. */
7   char* _IO_read_ptr; /* Current read pointer */
```

```
8   char* _IO_read_end; /* End of get area. */
9   char* _IO_read_base; /* Start of putback+get area. */
10  char* _IO_write_base; /* Start of put area. */
11  char* _IO_write_ptr; /* Current put pointer. */
12  char* _IO_write_end; /* End of put area. */
13  char* _IO_buf_base; /* Start of reserve area. */
14  char* _IO_buf_end; /* End of reserve area. */
15  /* The following fields are used to support backing up and undo.
        */
16  char *_IO_save_base; /* Pointer to start of non-current get area.
        */
17  char *_IO_backup_base; /* Pointer to first valid character of
        backup area */
18  char *_IO_save_end; /* Pointer to end of non-current get area. */
19
20  struct _IO_marker *_markers;
21
22  struct _IO_FILE *_chain;
23
24  int _fileno;
25 #if 0
26  int _blksize;
27 #else
28  int _flags2;
29 #endif
30  _IO_off_t _old_offset; /* This used to be _offset but it's too
        small. */
31
32 #define __HAVE_COLUMN /* temporary */
33  /* 1+column number of pbase(); 0 is unknown. */
34  unsigned short _cur_column;
35  signed char _vtable_offset;
36  char _shortbuf[1];
37
38  /* char* _save_gptr; char* _save_egptr; */
39
40  _IO_lock_t *_lock;
41 #ifdef _IO_USE_OLD_IO_FILE
42 };
43
44 struct _IO_FILE_complete
45 {
46  struct _IO_FILE _file;
```

```
47 #endif
48 #if defined _G_IO_IO_FILE_VERSION && _G_IO_IO_FILE_VERSION ==
       0x20001
49   _IO_off64_t _offset;
50 # if defined _LIBC || defined _GLIBCPP_USE_WCHAR_T
51   /* Wide character stream stuff. */
52   struct _IO_codecvt *_codecvt;
53   struct _IO_wide_data *_wide_data;
54   struct _IO_FILE *_freeres_list;
55   void *_freeres_buf;
56 # else
57   void *__pad1;
58   void *__pad2;
59   void *__pad3;
60   void *__pad4;
61 # endif
62   size_t __pad5;
63   int _mode;
64   /* Make sure we don't get into trouble again. */
65   char _unused2[15 * sizeof (int) - 4 * sizeof (void *) - sizeof (size_t)];
66 #endif
67 };
68
69 #ifndef __cplusplus
70 typedef struct _IO_FILE _IO_FILE;
71 #endif
72 typedef struct _IO_FILE FILE;
```