



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»
«Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №5
По теме: "Буферизованный и не буферизованный ввод-вывод"

Студент:

Барышникова Ю.Г.

Группа:

ИУ7-63Б.

Преподаватель:

Рязанова Н.Ю.

Первая программа. Один поток

```
1 ...
2 #define BUFF_SIZE 20
3 ...
4
5 int main()
6 {
7     int fd = open(FILE_NAME, O_RDONLY);
8
9     FILE *fs1 = fdopen(fd, "r");
10    char buff1[BUFF_SIZE];
11    setvbuf(fs1, buff1, _IOFBF, BUFF_SIZE);
12
13    FILE *fs2 = fdopen(fd, "r");
14    char buff2[BUFF_SIZE];
15    setvbuf(fs2, buff2, _IOFBF, BUFF_SIZE);
16
17    int flag1 = 1, flag2 = 2;
18
19    while (flag1 == 1 || flag2 == 1)
20    {
21        char c;
22        flag1 = fscanf(fs1, "%c", &c);
23        if (flag1 == 1)
24        {
25            fprintf(stdout, GREEN "%c", c);
26        }
27        flag2 = fscanf(fs2, "%c", &c);
28        if (flag2 == 1)
29        {
30            fprintf(stdout, BLUE "%c", c);
31        }
32    }
33
34    return OK;
35 }
```

Функция **fdopen** связывает поток с существующим дескриптором файла. Режим **mode** потока должен быть совместим с режимом дескриптора файла. Указатель файловой позиции в новом потоке принимает значение, равное значению позиции в дескрипторе, а указатели ошибок и конца файла равны нулю. Дескриптор файла не копируется и будет закрыт, когда поток, созданный **fdopen**, закрывается.

Функция **setvbuf** может быть использована над открытым потоком для изменения типа буферизации. Параметр **mode** должен быть одним из трех следующих макросов:

_IONBF - отключить буферизацию, информация незамедлительно оказывается на термине (или в файле назначения);

_IOLBF - строчная буферизация, все символы сохраняются в буфере до перевода строки;

_IOFBF - блочная буферизация, сохраняется **size** символов в буфер за раз.

Аргумент **buf** должен указывать на буфер (кроме случаев, когда буферизация отключается) размером, как минимум, равным **size** байтам; и этот буфер будет использоваться вместо текущего.

При операции I/O над файлом производится вызов **malloc** и выделяется буфер.

Если поток ссылается на терминал (как это делает **stdout**), то поток буферизируется построчно.

Стандартный поток ошибок **stderr** по умолчанию не буферизируется.

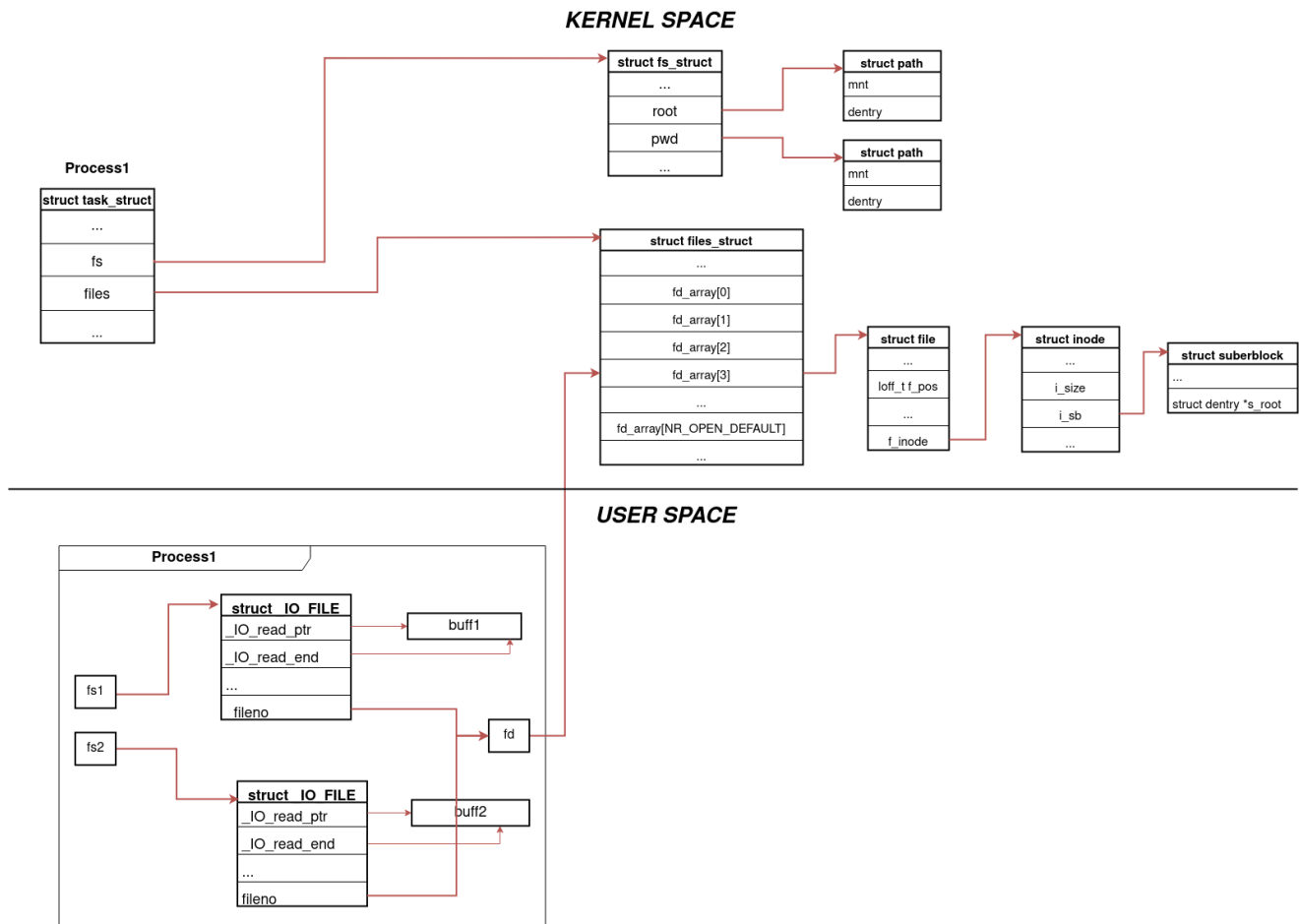


Рис. 1: Связь между структурами

Данная программа считывает информацию из файла 'alphabet.txt', который содержит строку символов 'Abcdefghijklmnopqrstuvwxyz'. И при помощи двух буферов посимвольно выводит считанные символы в стандартный поток вывода stdout.

В начале функции main `open()` создает 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.

Далее `fdopen()` создает указатели `fs1` и `fs2` на структуру FILE. В данных структурах поле `_fileno` будет содержать дескриптор `fd`, который вернула функция `fdopen()`, в данном случае 3.

Функция `setvbuf()` изменяет тип буферизации для `fs1` и `fs2` на полную буферизацию, а также явно задает размер буфера 20 байт.

Далее при первом вызове `fscanf()` буфер `fs1` заполнится полностью, т.е. первыми 20 символами. Значение `f_pos` в структуре `struct _file` открытого файла увеличится на 20.

Далее при первом вызове `fscanf()` для `fs2` в `buff2` считаются оставшиеся 6 символов, начиная с `f_pos` (т.к. `fs1` и `fs2` ссылаются на один и тот же дескриптор `fd`).

Далее в цикле поочередно выводятся символы из `buff1` и `buff2`. Т.к. в `buff2` записались оставшиеся 6 символов, после 6 итерации цикла будут выводиться символы только из `buff1`.

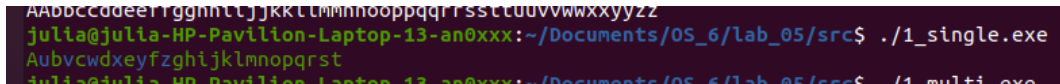


Рис. 2: Результат работы 1-й программы

Первая программа. Два потока.

```
1  ...
2  #define BUFF_SIZE 20
3  ...
4
5  int fd;
6
7  void *thread_fn(void *arg)
8  {
9      FILE *fs2 = fdopen(fd, "r");
10     char buff2[BUFF_SIZE];
11     setvbuf(fs2, buff2, _IOFBF, BUFF_SIZE);
12
13     char c;
14     while (fscanf(fs2, "%c", &c)==1)
15     {
16         fprintf(stdout, GREEN "%c", c);
17         sleep(0.5);
18     }
19 }
20
21 int main()
22 {
23     pthread_t tid;
24
25     fd = open(FILE_NAME, O_RDONLY);
26
27     int err = pthread_create(&tid, NULL, thread_fn, 0);
28     if (err)
29     {
30         printf("unable to create thread");
31         return ERROR_THREAD_CREATE;
32     }
33     FILE *fs1 = fdopen(fd, "r");
34     char buff1[BUFF_SIZE];
35     setvbuf(fs1, buff1, _IOFBF, BUFF_SIZE);
```

```

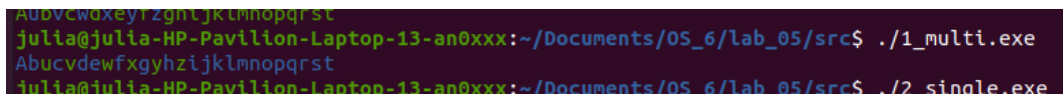
36
37 char c;
38 while (fscanf(fs1, "%c", &c)==1)
39 {
40     fprintf(stdout, BLUE "%c", c);
41     sleep(0.5);
42 }
43 pthread_join(tid, NULL);
44 printf("\n");
45 return OK;
46 }

```

При первом вызове `fscanf()` в основном потоке буфер `fs1` заполнится полностью, т.е. первыми 20 символами. Значение `f_pos` в структуре `struct _file` открытого файла увеличится на 20. Далее произойдет вызов `sleep()`, чтобы дополнительный поток успел вызвать `fscanf`.

При первом вызове `fscanf()` в дополнительном потоке для `fs2` в `buff2` считаются оставшиеся 6 символов, начиная с `f_pos`.

Далее в цикле каждый поток выводит символы из `buff1` и `buff2`. Т.к. в `buff2` записались оставшиеся 6 символов, после 6 итерации цикла будут выводиться символы только из `buff1`.



```

Abucvdxeyfzghijklmnopqrst
julia@julia-HP-Pavilion-Laptop-13-an0xxx:~/Documents/OS_6/lab_05/src$ ./1_multi.exe
Abucvdxeyfzghijklmnopqrst
julia@julia-HP-Pavilion-Laptop-13-an0xxx:~/Documents/OS_6/lab_05/src$ ./2_single.exe

```

Рис. 3: Результат работы 1-й программы при двух потоках

Вторая программа. Один поток.

```

1 ...
2
3 int main()
4 {
5     char c;
6
7     int fd1 = open(FILE_NAME, O_RDONLY);
8     int fd2 = open(FILE_NAME, O_RDONLY);
9
10    while (read(fd1, &c, 1) && read(fd2, &c, 1))
11    {
12        write(1, &c, 1);
13        write(1, &c, 1);
14    }
15
16    write(1, "\n", 1);
17    return OK;
18 }

```

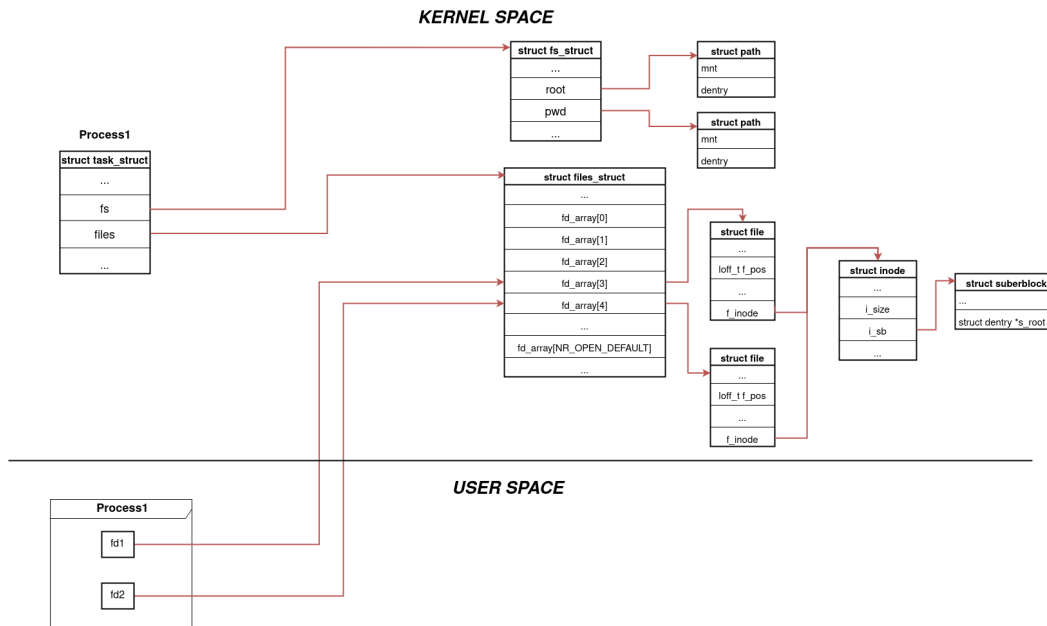


Рис. 4: Связь между структурами

Создается два файловых дескриптора при помощи функции `open()`. При этом создается две структуры `struct _file`, описывающие файл.

Далее в цикле поочередно считываются символы из файла и выводятся на экран. Т.к. созданы две структуры `struct _file`, у каждой структуры будет свой `f_pos`, поэтому смещения в файловых дескрипторах `fd1` и `fd2` будут независимы, и на экран будут дважды выводиться символы одного и того же файла.

```

AAbccddeeffgghhiijjkkllmmnnoppqrrssttuuvvwxxyyzz
julia@julia-HP-Pavilion-Laptop-13-an0xxx:~/Documents/OS_6/lab_05/src$ ./2_single.exe
AAbccddeeffgghhiijjkkllmmnnoppqrrssttuuvvwxxyyzz

```

Рис. 5: Результат работы второй программы

Вторая программа. Два потока.

```

1  ...
2
3  void read_file(int fd)
4  {
5      char c;
6      while (read(fd, &c, 1))
7      {
8          write(1, &c, 1);
9      }
10 }
11
12 void *thr_fn(void *arg)
13 {
14     int fd = open(FILE_NAME, O_RDONLY);

```

```

15     read_file(fd);
16 }
17
18 int main()
19 {
20     pthread_t tid;
21     int fd = open(FILE_NAME, O_RDONLY);
22     int err = pthread_create(&tid, NULL, thr_fn, 0);
23     if (err)
24     {
25         return ERROR_THREAD_CREATE;
26     }
27     read_file(fd);
28     pthread_join(tid, NULL);
29     return OK;
30 }

```

В программе также, как и при реализации с одним потоком, создается два файловых дескриптора для открытого файла, записи в системной таблице открытых файлов. У каждой записи будет свое смещение `f_pos`. Т.к. главный поток ждет окончания дочернего, то гарантируется вывод всего алфавита дважды. Порядок, в котором будут выводиться символы алфавита, неизвестен, т.к. вывод производится параллельно.

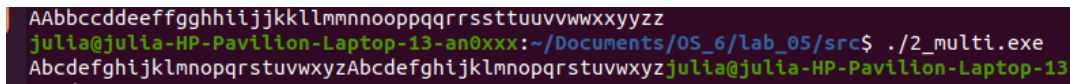


Рис. 6: Результат работы второй программы при двух потоках

Третья программа. Один поток.

```

1  ...
2
3  void Info()
4  {
5      struct stat statbuf;
6
7      stat(FILE_NAME, &statbuf);
8      printf("inode: %ld\n", statbuf.st_ino);
9      printf("st_size: %ld\n", statbuf.st_size);
10     printf("st_blksize: %ld\n\n", statbuf.st_blksize);
11 }
12
13 int main()
14 {
15     FILE *f1 = fopen(FILE_NAME, "w");
16     Info();
17
18     FILE *f2 = fopen(FILE_NAME, "w");
19     Info();

```

```

20
21 char c = 'a';
22 while (c <= 'z')
23 {
24     if (c % 2)
25         fprintf(f1, "%c", c);
26     else
27         fprintf(f2, "%c", c);
28     c++;
29 }
30
31 fclose(f1);
32 Info();
33
34 fclose(f2);
35 Info();
36
37 return OK;
38 }

```

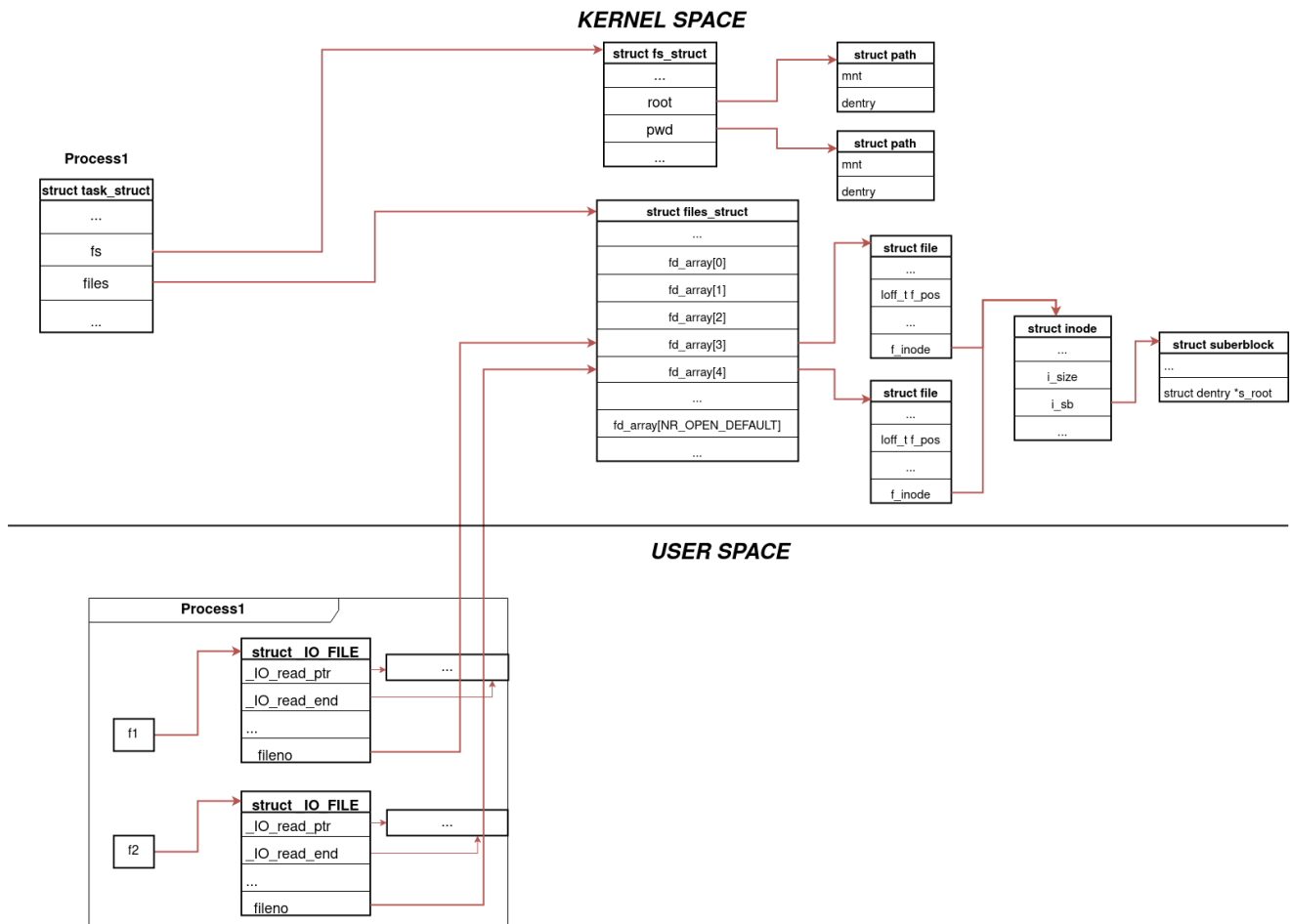


Рис. 7: Связь между структурами

В данной программе файл 'task3.txt' открывается 2 раза для записи. Выполняется ввод через стандартную библиотеку C (stdio.h) с помощью функции fprintf() - буферизованный ввод/вывод.

Буфер создается без нашего явного вмешательства. Сначала информация записывается в буфер, из буфера информация переписывается в файл в результате 3-ех действий:

1. буфер полон;
2. принудительная запись fflush();
3. если вызван fclose().

В нашей программе символы, имеющие нечетный код в таблице ASCII записываются в буфер, который находится в дескрипторе f1, в f2 соответственно записываются четные.

Таким образом в буфере, который содержится в f1 будут символы: 'acej...', а в f2 'bdfh...'. В нашем случае информация из фубера запишется в файл при вызове fclose().

Т.к. f_pos независимы у каждого дескриптора файла, то при закрытии файла запись будет производиться начиная с начала файла в обоих случаях.

Таким образом информация, которая будет записана в файл, после первого вызова fclose() будет потеряна в результате второго вызова fclose() рис. 9.

```
julia@julia-HP-Pavilion-Laptop-13-an0xxx:~/Documents/OS_6/lab_05/src$ ./3_single.exe
inode: 1447045
Общий размер в байтах: 0
Размер блока ввода-вывода: 4096

inode: 1447045
Общий размер в байтах: 0
Размер блока ввода-вывода: 4096

inode: 1447045
Общий размер в байтах: 13
Размер блока ввода-вывода: 4096

inode: 1447045
Общий размер в байтах: 13
Размер блока ввода-вывода: 4096

julia@julia-HP-Pavilion-Laptop-13-an0xxx:~/Documents/OS_6/lab_05/src$ cat task3.txt
bdfhjlnprtvxzjulia@julia-HP-Pavilion-Laptop-13-an0xxx:~/Documents/OS_6/lab_05/src$ ./
inode: 1447122
```

Рис. 8: Результат работы третьей программы

Третья программа. Два потока.

```
1 ...
2
3 void Info()
4 {
5     struct stat statbuf;
6
7     stat(FILE_NAME, &statbuf);
8     printf("inode: %ld\n", statbuf.st_ino);
9     printf("st_size: %ld\n", statbuf.st_size);
10    printf("st_blksize: %ld\n\n", statbuf.st_blksize);
11 }
```

```

12
13 void WriteToFile(char c)
14 {
15     FILE *f = fopen(FILE_NAME, "w");
16     Info();
17
18     while (c <= 'z')
19     {
20         fprintf(f, "%c", c);
21         c += 2;
22     }
23
24     fclose(f);
25     Info();
26 }
27
28 void *thr_fn(void *arg)
29 {
30     WriteToFile('b');
31 }
32
33 int main()
34 {
35     pthread_t tid;
36
37     int err = pthread_create(&tid, NULL, thr_fn, NULL);
38     if (err)
39     {
40         return ERROR_THREAD_CREATE;
41     }
42
43     WriteToFile('a');
44     pthread_join(tid, NULL);
45
46     return OK;
47 }

```

В данной программе создается поток. Главный поток записывает в файл символы, начиная с 'a', в то время, как созданный нами поток записывает символы, начиная с 'b'.

Так же как и в приведенной выше программе с одним потоком происходит потеря данных.

Данные будут записаны из того буфера (который содержится в дескрипторе), для которого будет вызван `fclose()` последним, потому что он перезапишет данные с начала файла.

```
julia@julia-HP-Pavilion-Laptop-13-an0xxx:~/Documents/OS_6/lab_05/src$ cat task3.txt
bdfhjlnprtvxzjulia@julia-HP-Pavilion-Laptop-13-an0xxx:~/Documents/OS_6/lab_05/src$ ./3_multi.exe
inode: 1447123
Общий размер в байтах: 0
Размер блока ввода-вывода: 4096

inode: 1447123
Общий размер в байтах: 13
Размер блока ввода-вывода: 4096

inode: 1447123
Общий размер в байтах: 0
Размер блока ввода-вывода: 4096

inode: 1447123
Общий размер в байтах: 13
Размер блока ввода-вывода: 4096

julia@julia-HP-Pavilion-Laptop-13-an0xxx:~/Documents/OS_6/lab_05/src$ cat task3.txt
bdfhjlnprtvxzjulia@julia-HP-Pavilion-Laptop-13-an0xxx:~/Documents/OS_6/lab_05/src$
```

Рис. 9: Результат работы третьей программы

Вывод

В ходе выполнения данной лабораторной работы были проанализированы три программы в двух вариантах реализации, в результате анализа этих программ были выявлены следующие особенности:

1. При работе с файлом через стандартную библиотеку C (stdio.h) буферизованного ввода/вывода, сначала информация записывается в буфер, из буфера информация переписывается в файл в результате 3-х действий:
 - (a) буфер полон;
 - (b) принудительная запись fflush();
 - (c) если вызван fclose().Таким образом, после записи данных в файл может оказаться, что реально они там отсутствуют, поскольку были записаны в буфер и еще не переписаны в файл. Чтобы избежать подобной ситуации необходимо своевременно очищать буфер с помощью fflush и fclose.
2. Как в однопоточной так и в многопоточной реализациях первой программы происходит аналогичная ситуация. При чтении одного и того же файла с использованием fscanf часть информации не будет прочитана, поскольку поток, который первым получил квант первым вызовом scanf заполнит буфер и сместит fpos. Из-за этого другой поток начнет читать файл не с начала, а с позиции fpos.
3. При записи информации в конец файла в многопоточной и однопоточной реализациях, данные, записанные после первого вызова fclose были потеряны после следующего вызова fclose, поскольку fpos в другом файловом дескрипторе указывал на начало файла, а не на позицию после записанных данных. Для решения этой проблемы следует использовать неделимый системный вызов O_APPEND. Если этот режим установлен, то перед каждым вызовом write смещение в файле устанавливается в конец файла, как если бы выполнялся FSEEK.

Дополнение

В файле 'cat /usr/include/x86_64-linux-gnu/bits/libio.h' находится описание структуры _IO_FILE

```
1 struct _IO_FILE
2 {
3     int _flags; /* High-order word is _IO_MAGIC; rest is flags. */
4 #define _IO_file_flags _flags
5
6     /* The following pointers correspond to the C++ streambuf protocol. */
7     /* Note: Tk uses the _IO_read_ptr and _IO_read_end fields directly. */
8     char *_IO_read_ptr; /* Current read pointer */
9     char *_IO_read_end; /* End of get area. */
10    char *_IO_read_base; /* Start of putback+get area. */
11    char *_IO_write_base; /* Start of put area. */
12    char *_IO_write_ptr; /* Current put pointer. */
13    char *_IO_write_end; /* End of put area. */
14    char *_IO_buf_base; /* Start of reserve area. */
15    char *_IO_buf_end; /* End of reserve area. */
16    /* The following fields are used to support backing up and undo. */
17    char *_IO_save_base; /* Pointer to start of non-current get area. */
18    char *_IO_backup_base; /* Pointer to first valid character of backup
19        area */
20    char *_IO_save_end; /* Pointer to end of non-current get area. */
21
22    struct _IO_marker *_markers;
23
24    struct _IO_FILE *_chain;
25
26    int _fileno;
27 #if 0
28     int _blksize;
29 #else
30     int _flags2;
31 #endif
32
33     _IO_off_t _old_offset; /* This used to be _offset but it's too small.
34        */
35
36 #define __HAVE_COLUMN /* temporary */
37     /* 1+column number of pbase(); 0 is unknown. */
38     unsigned short _cur_column;
39     signed char _vtable_offset;
40     char _shortbuf[1];
41
42     /* char* _save_gptr; char* _save_egptr; */
43
44     _IO_lock_t *_lock;
45 #ifndef _IO_USE_OLD_IO_FILE
46 };
47 #endif
```