



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»

ОТЧЕТ

по Лабораторной работе №5

по курсу «Операционные системы»

на тему: «Буферизованный и не буферизованный ввод-вывод»

Студент ИУ7и-65Б
(Группа)

(Подпись, дата)

Каримзай А- Х.
(И. О. Фамилия)

Преподаватель

(Подпись, дата)

Рязанова Н. Ю.
(И. О. Фамилия)

2022 г.

1 Структура FILE

В данной лабораторной работе использовалась версия Linux 5.13.0-41-generic

Описание структуры FILE приведено в Листинге 1.1

Листинг 1.1 – Файл /usr/include/bits/types/FILE.h

```
1 #ifndef __FILE_defined
2 #define __FILE_defined 1
3
4 struct _IO_FILE;
5
6 /* The opaque type of streams. This is the definition used elsewhere. */
7 typedef struct _IO_FILE FILE;
8
9 #endif
```

Описание структуры _IO_FILE приведено в Листингах 1.2 – 1.4

Листинг 1.2 – Файл /usr/include/bits/types/struct_FILE.h, Часть 1

```
1 /* Copyright (C) 1991-2022 Free Software Foundation, Inc.
2    This file is part of the GNU C Library.
3
4    The GNU C Library is free software; you can redistribute it and/or
5    modify it under the terms of the GNU Lesser General Public
6    License as published by the Free Software Foundation; either
7    version 2.1 of the License, or (at your option) any later version.
8
9    The GNU C Library is distributed in the hope that it will be useful,
10   but WITHOUT ANY WARRANTY; without even the implied warranty of
11   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
12   Lesser General Public License for more details.
13
14   You should have received a copy of the GNU Lesser General Public
15   License along with the GNU C Library; if not, see
16   <https://www.gnu.org/licenses/>. */
17
18 #ifndef __struct_FILE_defined
19 #define __struct_FILE_defined 1
20
21 /* Caution: The contents of this file are not part of the official
22    stdio.h API. However, much of it is part of the official *binary*
23    interface, and therefore cannot be changed. */
24
25 #if defined _IO_USE_OLD_IO_FILE && !defined _LIBC
26 # error "_IO_USE_OLD_IO_FILE should only be defined when building libc itself"
27 #endif
28
29 #if defined _IO_lock_t_defined && !defined _LIBC
30 # error "_IO_lock_t_defined should only be defined when building libc itself"
31 #endif
```

Листинг 1.3 – Файл /usr/include/bits/types/struct_FILE.h, Часть 2

```

33 #include <bits/types.h>
34
35 struct _IO_FILE;
36 struct _IO_marker;
37 struct _IO_codecvt;
38 struct _IO_wide_data;
39
40 /* During the build of glibc itself, _IO_lock_t will already have been
41    defined by internal headers.  */
42 #ifndef _IO_lock_t_defined
43 typedef void _IO_lock_t;
44 #endif
45
46 /* The tag name of this struct is _IO_FILE to preserve historic
47    C++ mangled names for functions taking FILE* arguments.
48    That name should not be used in new code.  */
49 struct _IO_FILE
50 {
51     int _flags;          /* High-order word is _IO_MAGIC; rest is flags.  */
52
53     /* The following pointers correspond to the C++ streambuf protocol.  */
54     char *_IO_read_ptr;  /* Current read pointer */
55     char *_IO_read_end;  /* End of get area.  */
56     char *_IO_read_base; /* Start of putback+get area.  */
57     char *_IO_write_base; /* Start of put area.  */
58     char *_IO_write_ptr; /* Current put pointer.  */
59     char *_IO_write_end; /* End of put area.  */
60     char *_IO_buf_base;  /* Start of reserve area.  */
61     char *_IO_buf_end;   /* End of reserve area.  */
62
63     /* The following fields are used to support backing up and undo.  */
64     char *_IO_save_base; /* Pointer to start of non-current get area.  */
65     char *_IO_backup_base; /* Pointer to first valid character of backup area */
66     char *_IO_save_end; /* Pointer to end of non-current get area.  */
67
68     struct _IO_marker *_markers;
69
70     struct _IO_FILE *_chain;
71
72     int _fileno;
73     int _flags2;
74     __off_t _old_offset; /* This used to be _offset but it's too small.  */
75
76     /* 1+column number of pbase(); 0 is unknown.  */
77     unsigned short _cur_column;
78     signed char _vtable_offset;
79     char _shortbuf[1];
80
81     _IO_lock_t *_lock;
82 #ifdef _IO_USE_OLD_IO_FILE
83 };
84
85 struct _IO_FILE_complete
86 {
87     struct _IO_FILE _file;
88 #endif

```

Листинг 1.4 – Файл /usr/include/bits/types/struct_FILE.h, Часть 2

```

89  __off64_t _offset;
90  /* Wide character stream stuff. */
91  struct _IO_codecvt *_codecvt;
92  struct _IO_wide_data *_wide_data;
93  struct _IO_FILE *_freeres_list;
94  void *_freeres_buf;
95  size_t __pad5;
96  int _mode;
97  /* Make sure we don't get into trouble again. */
98  char _unused2[15 * sizeof (int) - 4 * sizeof (void *) - sizeof (size_t)];
99 };
100
101 /* These macros are used by bits/stdio.h and internal headers. */
102 #define __getc_unlocked_body(_fp) \
103     (__glibc_unlikely ((_fp)->_IO_read_ptr >= (_fp)->_IO_read_end) \
104      ? __uflow (_fp) : *(unsigned char *) (_fp)->_IO_read_ptr++)
105
106 #define __putc_unlocked_body(_ch, _fp) \
107     (__glibc_unlikely ((_fp)->_IO_write_ptr >= (_fp)->_IO_write_end) \
108      ? __overflow (_fp, (unsigned char) (_ch)) \
109      : (unsigned char) (*( _fp)->_IO_write_ptr++ = (_ch)))
110
111 #define _IO_EOF_SEEN 0x0010
112 #define __feof_unlocked_body(_fp) (((_fp)->_flags & _IO_EOF_SEEN) != 0)
113
114 #define _IO_ERR_SEEN 0x0020
115 #define __ferror_unlocked_body(_fp) (((_fp)->_flags & _IO_ERR_SEEN) != 0)
116
117 #define _IO_USER_LOCK 0x8000
118 /* Many more flag bits are defined internally. */
119
120 #endif

```

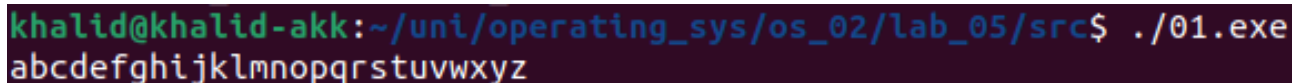
2 Реализация программ

2.1 Первая программа

2.1.1 Коды программ

Листинг 2.1 – Открытие одного и того же файла несколько раз для чтения в одном процессе

```
1  #include <stdio.h>
2  #include <fcntl.h>
3
4  #define BUF_N 20
5
6  int main(void)
7  {
8      int fd = open("alphs.txt", O_RDONLY);
9
10     FILE *fs1 = fdopen(fd, "r");
11     FILE *fs2 = fdopen(fd, "r");
12     char buff1[BUF_N], buff2[BUF_N];
13     setvbuf(fs1, buff1, _IOFBF, BUF_N);
14     setvbuf(fs2, buff2, _IOFBF, BUF_N);
15
16     int flags[2] = { 1, 1 };
17     while (flags[0] == 1 || flags[1] == 1)
18     {
19         char c[2];
20         flags[0] = fscanf(fs1, "%c", &c[0]);
21         flags[1] = fscanf(fs1, "%c", &c[1]);
22         if (flags[0] == 1) fprintf(stdout, "%c", c[0]);
23         if (flags[1] == 1) fprintf(stdout, "%c", c[1]);
24     }
25     fprintf(stdout, "\n");
26     return 0;
27 }
```

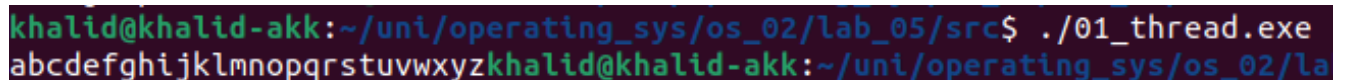


```
khalid@khalid-akk:~/uni/operating_sys/os_02/lab_05/src$ ./01.exe
abcdefghijklmnopqrstuvwxyz
```

Рисунок 2.1 – Программа с одним процессом

Листинг 2.2 – Открытие одного и того же файла для чтения в нескольких потоках

```
1  #include <stdio.h>
2  #include <fcntl.h>
3  #include <pthread.h>
4
5  #define BUF_N 20
6  #define THD_N 2
7
8  void *open_read(void *args)
9  {
10     int fd = *(int *)args;
11     FILE *fs = fdopen(fd, "r");
12     char buff[BUF_N];
13     setvbuf(fs, buff, _IOFBF, BUF_N);
14
15     char c;
16     while (fscanf(fs, "%c", &c) == 1)
17         fprintf(stdout, "%c", c);
18 }
19
20 int main(void)
21 {
22     int fd = open("alphs.txt", O_RDONLY);
23
24     pthread_t tid[THD_N];
25     for (size_t i = 0; i < THD_N; i++)
26         pthread_create(&tid[i], NULL, open_read, (void *)&fd);
27
28     for (size_t i = 0; i < THD_N; i++)
29         pthread_join(tid[i], NULL);
30     return 0;
31 }
```



```
khalid@khalid-akk:~/uni/operating_sys/os_02/lab_05/src$ ./01_thread.exe
abcdefghijklmnopqrstuvwxyzkhalid@khalid-akk:~/uni/operating_sys/os_02/la
```

Рисунок 2.2 – Открытие одного и того же файла для чтения в нескольких потоках

2.1.2 Схема взаимодействия структур

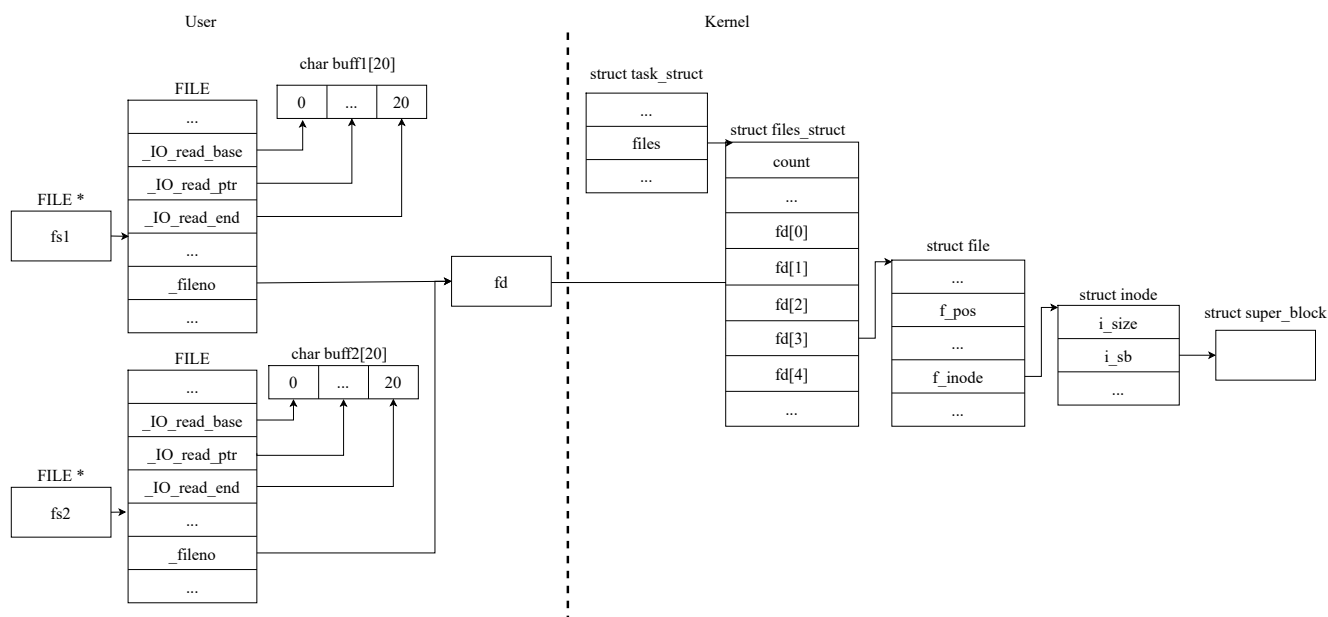


Рисунок 2.3 – Схема взаимодействия структур

Выводы

- функция `open()` создает новый файловый дескриптор файла (открытого только на чтение) "data.txt", запись в системной таблице открыт файлов. Эта запись регистрирует смещение в файле и флаги состояния файла;
- функция `fdopen()` создает указатели на структуру `FILE`. Поле `_fileno` содержит дескриптор, который вернула функция `fopen()`;
- функция `setvbuf()` явно задает размер буфера в 20 байт и меняет тип буферизации (для `fs1` и `fs2`) на полную;
- при первом вызове функции `fscanf()` в цикле (для `fs1`), `buff1` будет заполнен полностью – первыми 20 символами (буквами алфавита). `f_pos` в структуре `struct_file` открытого файла увеличится на 20;
- при втором вызове `fscanf()` в цикле (для `fs2`) буфер `buff2` будет заполнен оставшимися 6 символами (начиная с `f_pos`);
- в цикле поочередно выводятся символы из `buff1` и `buff2`;

- все это справедливо и для многопоточной реализации: в потоке, который первый получит квант, первый вызов `fscanf()` заполнит буфер и увеличит `f_pos` в структуре `struct_file` открытого файла, а оставшиеся 6 символов будут записаны в буфер, находящийся в другом потоке.

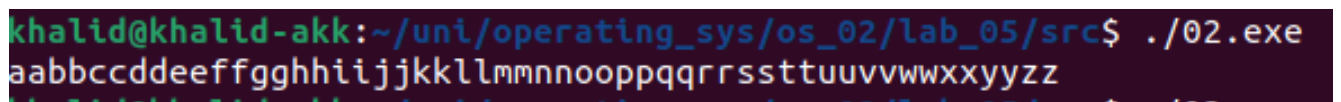
2.2 Вторая программа

2.2.1 Коды программ

Листинг 2.3 – Открытие одного и того же файла несколько раз для чтения в одном процессе

```

1  #include <fcntl.h>
2  #include <unistd.h>
3
4  int main(void)
5  {
6      int fd1 = open("alphs.txt", O_RDONLY);
7      int fd2 = open("alphs.txt", O_RDONLY);
8
9      int flags[2] = { 1, 1 };
10     while (flags[0] == 1 && flags[1] == 1)
11     {
12         char c;
13         flags[0] = read(fd1, &c, 1);
14         if (flags[0] == 1) write(1, &c, 1);
15         flags[1] = read(fd2, &c, 1);
16         if (flags[1] == 1) write(1, &c, 1);
17     }
18     write(1, "\n", 1);
19     return 0;
20 }
```



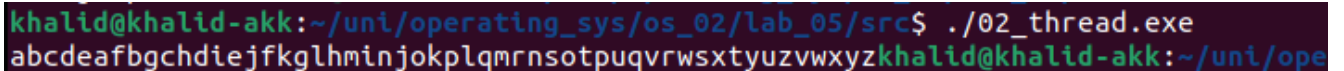
```

khalid@khalid-akk:~/uni/operating_sys/os_02/lab_05/src$ ./02.exe
aabbccddeeffgghhiijjkkllmmnnoooppqrrssttuuvvwwxxyyzz
```

Рисунок 2.4 – Открытие одного и того же файла несколько раз для чтения в одном процессе

Листинг 2.4 – Открытие одного и того же файла несколько раз для чтения в двух потоках

```
1  #include <fcntl.h>
2  #include <unistd.h>
3  #include <pthread.h>
4
5  #define BUF_N 20
6  #define THD_N 2
7
8  void *open_read(void *args)
9  {
10     int fd = open((char *)args, O_RDONLY);
11
12     char c;
13     while (read(fd, &c, 1) == 1)
14         write(1, &c, 1);
15 }
16
17 int main(void)
18 {
19     char *file_name = "alphs.txt";
20
21     pthread_t tid[THD_N];
22     for (size_t i = 0; i < THD_N; i++)
23         pthread_create(&tid[i], NULL, open_read, (void *)file_name);
24
25     for (size_t i = 0; i < THD_N; i++)
26         pthread_join(tid[i], NULL);
27     return 0;
28 }
```



```
khalid@khalid-akk:~/uni/operating_sys/os_02/lab_05/src$ ./02_thread.exe
abcdefghijklmnopqrstuvwxyzkhalid@khalid-akk:~/uni/ope
```

Рисунок 2.5 – Открытие одного и того же файла несколько раз для чтения в двух потоках

2.2.2 Схема взаимодействия структур

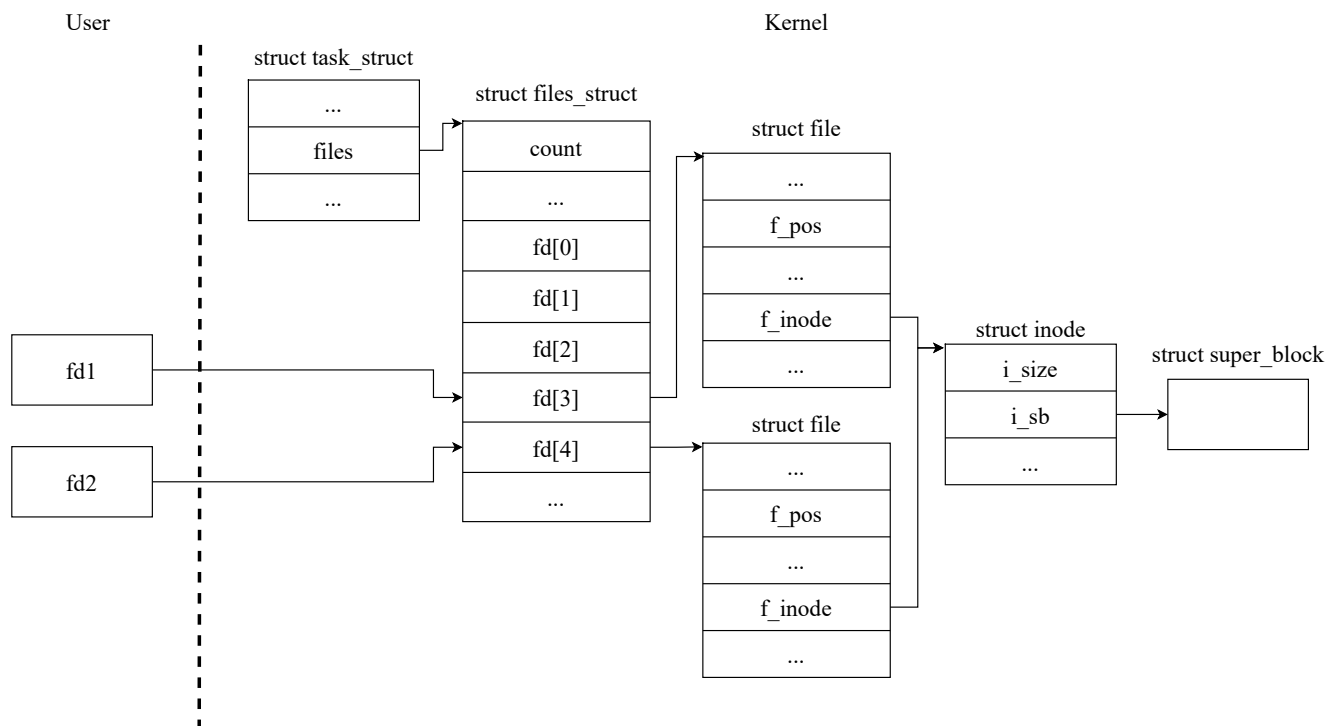


Рисунок 2.6 – Схема взаимодействия структур

Вывод

- функция `open()` создает файловые дескрипторы, два раза для одного и того же файла, поэтому в программе существует две различные `struct file`, но ссылающиеся на один и тот же `struct inode`;
- из-за того что структуры `struct file` разные, посимвольная печать просто дважды выведет содержимое файла в формате «AAbbccs...» (в случае однопроцессной реализации);
- в случае многопоточной реализации, вывод второго потока начнется позже (нужно время, для создание этого потока) и символы перемешаются (см. рис. 2.5).

2.3 Третья программа

2.3.1 Схема взаимодействия структур

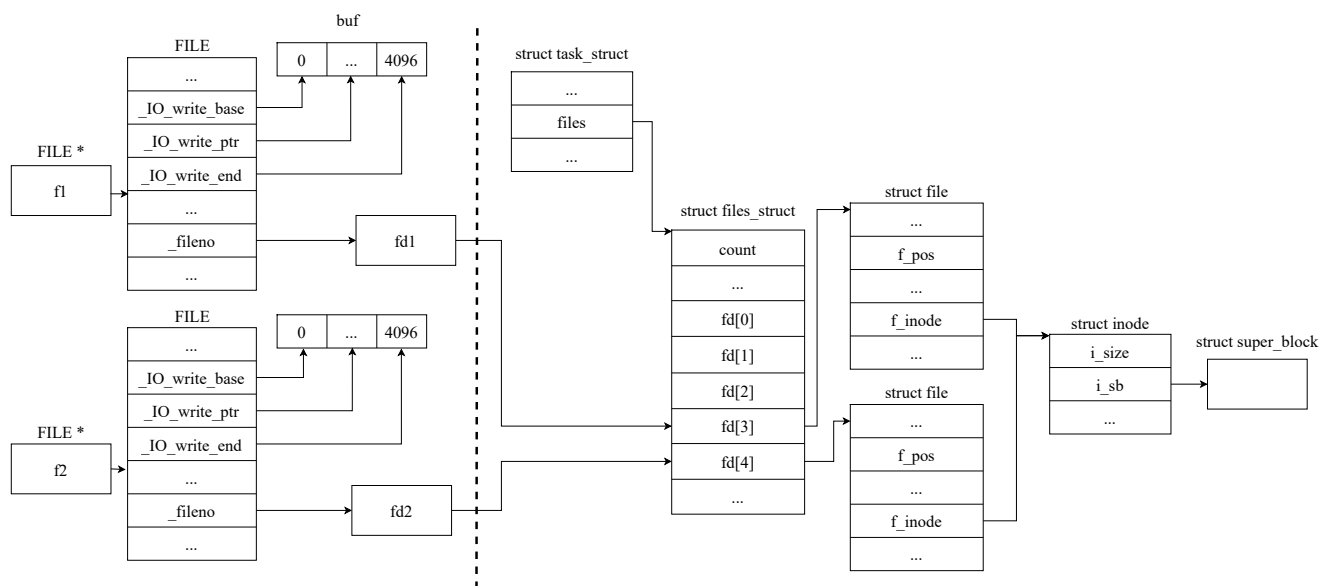


Рисунок 2.7 – Схема взаимодействия структур

2.3.2 Коды программ

Листинг 2.5 – Открытие одного и того же файла несколько раз для записи в одном процессе

```
1  #include <fcntl.h>
2  #include <unistd.h>
3  #include <pthread.h>
4
5  int main(void)
6  {
7      char *file_name = "temp.txt";
8
9      int fd1 = open(file_name, O_RDWR);
10     int fd2 = open(file_name, O_RDWR);
11
12     for (char c = 'a'; c <= 'z'; c++)
13         write(c % 2 ? fd1 : fd2, &c, 1);
14
15     close(fd1);
16     close(fd2);
17     return 0;
18 }
```

```
khalid@khalid-akk:~/uni/operating_sys/os_02/lab_05/src$ ./03.exe && cat temp.txt
bdfhjlnprtvxzkhalid@khalid-akk:~/uni/operating_sys/os_02/lab_05/src$
```

Рисунок 2.8 – Открытие одного и того же файла несколько раз для записи в одном процессе

Листинг 2.6 – Открытие одного и того же файла несколько раз для записи в двух потоках

```
1  #include <fcntl.h>
2  #include <unistd.h>
3  #include <pthread.h>
4
5  #define BUF_N 20
6  #define THD_N 2
7
8  struct thread_data {
9      int thd_n;
10     char *file_name;
11 };
12
13 void *open_read(void *args)
14 {
15     struct thread_data *tdata = (struct thread_data *)args;
16     int fd = open(tdata->file_name, O_RDWR);
17
18     for (char c = 'a' + tdata->thd_n; c <= 'z'; c += (tdata->thd_n + 1))
19         write(fd, &c, 1);
20 }
21
22 int main(void)
23 {
24     char *file_name = "temp.txt";
25
26     pthread_t tid[THD_N];
27     for (size_t i = 0; i < THD_N; i++)
28     {
29         struct thread_data tdata = { i, file_name };
30         pthread_create(&tid[i], NULL, open_read, (void *)&tdata);
31     }
32
33     for (size_t i = 0; i < THD_N; i++)
34         pthread_join(tid[i], NULL);
35     return 0;
36 }
```

```
khalid@khalid-akk:~/uni/operating_sys/os_02/lab_05/src$ ./03_thread.exe && cat temp.txt
bdfhjlnprtvxzkhalid@khalid-akk:~/uni/operating_sys/os_02/lab_05/src$
```

Рисунок 2.9 – Открытие одного и того же файла несколько раз для записи в двух потоках

Вывод

- файл открывается на запись два раза, с помощью функции `fopen()`;
- функция `fprintf()` предоставляет буферизованный вывод - буфер создается без нашего вмешательства;
- изначально информация пишется в буфер, а из буфера в файл если произошло одно из событий:

буффер полон;

вызвана функция `fclose()`;

вызвана функция `fflush()`;

- в случае нашей программы, информация в файл запишется в результате вызова функция `fclose()`;
- из-за того `f_pos` независимы для каждого дескриптора файла, запись в файл будет производиться с самого начала;
- таким образом, информация записанная при первом вызове `fclose()` будет потеряна в результате второго вызова `fclose()`.
- в многопоточной реализации результат аналогичен - с помощью `pthread_join` мы ожидаем вызова `fclose()` для `f2` в отдельном потоке и далее вызываем `fclose()` для `f1`.

Для исключения проблемы потери данных нужно открывать файл в режиме добавления - `O_APPEND`. Тогда операция записи в файл становится атомарной и перед каждым вызовом `write`, смещение в файле устанавливается в конец файла.