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»

Тема Буферизованный и небуферизованный ввод–вывод

Студент Кононенко С.С.

Группа ИУ7-63Б

Оценка (баллы) _____

Преподаватели Рязанова Н.Ю.

Москва — 2021 г.

Структура FILE

В данной лабораторной работе я использовал в качестве операционной системы Ubuntu 20.04 с версией ядра 5.8.0-44-generic.

Структура FILE описана в файле /usr/include/x86_64-linux-gnu/bits/types/FILE.h:

```
#ifndef __FILE_defined
#define __FILE_defined 1

struct _IO_FILE;

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

#endif
```

Структура _IO_FILE описана в файле /usr/include/x86_64-linux-gnu/bits/types/struct_FILE.h:

```
/* Copyright (C) 1991-2020 Free Software Foundation, Inc.
   This file is part of the GNU C Library.

   The GNU C Library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   The GNU C Library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
   Lesser General Public License for more details.

   You should have received a copy of the GNU Lesser General Public
   License along with the GNU C Library; if not, see
   <https://www.gnu.org/licenses/>. */

#ifndef __struct_FILE_defined
#define __struct_FILE_defined 1

/* Caution: The contents of this file are not part of the official
   stdio.h API. However, much of it is part of the official *binary*
   interface, and therefore cannot be changed. */

#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
32
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
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) \
105         : *(unsigned char *)(_fp)->_IO_read_ptr++)
106
107 #define __putc_unlocked_body(_ch, _fp) \
108     (__glibc_unlikely((_fp)->_IO_write_ptr >= (_fp)->_IO_write_end) \
109         ? __overflow(_fp, (unsigned char)(_ch)) \
110         : (unsigned char)(*(_fp)->_IO_write_ptr++ = (_ch)))
111
112 #define _IO_EOF_SEEN 0x0010
113 #define __feof_unlocked_body(_fp) (((_fp)->_flags & _IO_EOF_SEEN) != 0)
114
115 #define _IO_ERR_SEEN 0x0020
116 #define __ferror_unlocked_body(_fp) (((_fp)->_flags & _IO_ERR_SEEN) != 0)
117
118 #define _IO_USER_LOCK 0x8000
119 /* Many more flag bits are defined internally. */
120
121 #endif
```

Первая программа

Код (один поток):

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

#define BUF_SIZE 20

int main()
{
    int fd = open("alphabet.txt", O_RDONLY);

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

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

    int flag1 = 1, flag2 = 2;
    char c;
    while (flag1 == 1 || flag2 == 1)
    {
        flag1 = fscanf(fs1, "%c", &c);
        if (flag1 == 1)
        {
            fprintf(stdout, "%c", c);
        }
        flag2 = fscanf(fs2, "%c", &c);
        if (flag2 == 1)
        {
            fprintf(stdout, "%c", c);
        }
    }

    fprintf(stdout, "\n");

    return 0;
}
```

Код (два потока):

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

```
5
6  #define BUF_SIZE 20
7
8  void *thread_run(void *fs)
9  {
10     int flag = 1;
11
12     char c;
13     while (flag == 1)
14     {
15         flag = fscanf(fs, "%c", &c);
16         if (flag == 1)
17         {
18             fprintf(stdout, "thread2: %c\n", c);
19         }
20     }
21
22     return NULL;
23 }
24
25 int main()
26 {
27     int fd = open("alphabet.txt", O_RDONLY);
28     pthread_t td;
29
30     FILE *fs1 = fdopen(fd, "r");
31     char buff1[BUF_SIZE];
32     setvbuf(fs1, buff1, _IOFBF, BUF_SIZE);
33
34     FILE *fs2 = fdopen(fd, "r");
35     char buff2[BUF_SIZE];
36     setvbuf(fs2, buff2, _IOFBF, BUF_SIZE);
37
38     pthread_create(&td, NULL, thread_run, fs2);
39     usleep(1);
40
41     int flag = 1;
42     char c;
43     while (flag == 1)
44     {
45         flag = fscanf(fs1, "%c", &c);
46         if (flag == 1)
47         {
48             fprintf(stdout, "thread1: %c\n", c);
49         }
50     }
51
52     pthread_join(td, NULL);
```

```
53
54      return 0;
55  }
```

Результат работы:

```
hackfeed@hackfeed:~/oslabs/lab_05/src$ ./prog1.exe
aubvcwdxeyfzghijklmnopqrst
```

Рисунок 1 – Программа с одним потоком

```
hackfeed@hackfeed:~/oslabs/lab_05/src$ ./prog1t.exe
thread2: a
thread1: u
thread1: v
thread1: w
thread1: x
thread1: y
thread1: z
thread2: b
thread2: c
thread2: d
thread2: e
thread2: f
thread2: g
thread2: h
thread2: i
thread2: j
thread2: k
thread2: l
thread2: m
thread2: n
thread2: o
thread2: p
thread2: q
thread2: r
thread2: s
thread2: t
```

Рисунок 2 – Программа с двумя потоками

Связь структур:

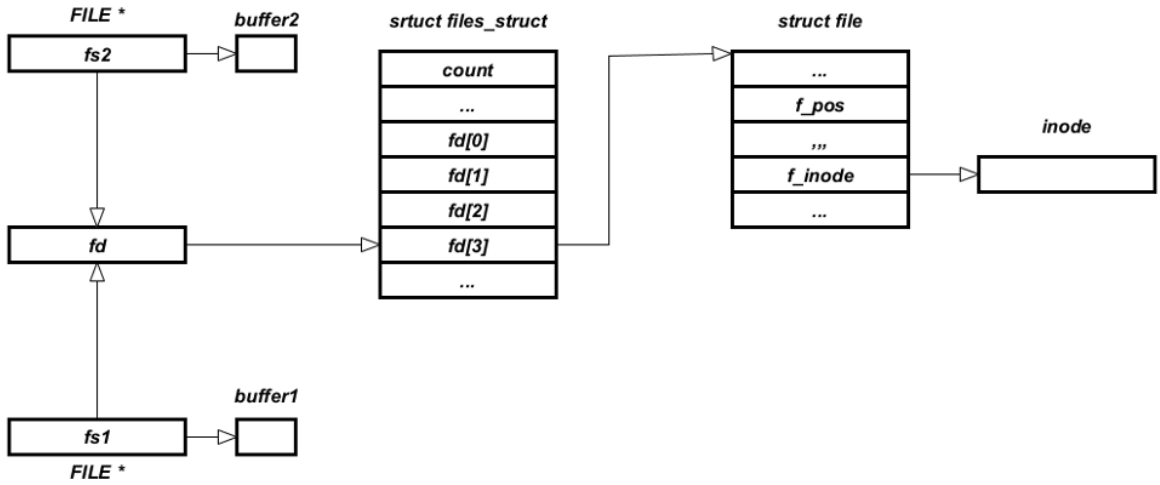

Рисунок 3 – Связь структур

Анализ: изначально в данной программе с помощью системного вызова `open()` создается файловый дескриптор `fd` для файла `alphabet.txt` с правами доступа на чтение (`O_RDONLY`), этому файловому дескриптору присваивается значение 3 (потому что 0, 1, 2 заняты `stdin`, `stdout` и `stderr` соответственно). В это же время у `struct files_struct` (можно найти в `struct task_struct` для текущего процесса (структура ядра), поле `files`) поле `fd[3]` начинает указывать на `struct file`, связанный с `struct inode`, соответствующий файлу с именем `alphabet.txt`.

Далее, с помощью вызова функции стандартной библиотеки `fdopen()`, создаются 2 структуры `FILE` (`fs1`, `fs2`), поле `_fileno` становится равным значению 3.

Затем с помощью вызова функции `setvbuf` создаются буферы для обоих структур `FILE`, в качестве аргумента функция получает буфер, размер буфера, а также стратегию буферизации (по строчкам (то есть до ближайшего символа конца строки) или полностью заполняя буфер).

Следующий интересующий нас вызов — вызов функции стандартной библиотеки `fscanf()`. Первый раз он вызывается для `fs1` указателя на структуру `FILE`. При вызове `fscanf()` буфер `fs1` заполняется полностью, так как был выбран режим `_IOFBF` (полная буферизация). В структуре `struct file`, соответствующей дескриптору `fd`, поле `f_pos` увеличивается на 20 (из файла считалось 20 символов во время вызова `fscanf()`, чтобы заполнить буфер структуры `fs1`).

Когда `fscanf()` вызывается для `fs2` (учитывая, что структура `fs1` считала все символы до 't' и `fs2` имеет один файловый дескриптор со структурой `fs1`) буферизуются все символы после 't'.

Далее при следующих вызовах `fscanf()` для символов, символы будут браться из буфера до тех пор, пока он не станет пустым. Когда символы в буфере кончатся, структуры возвращают `EOF`, так как весь файл был прочитан (был дочитан структурой `fs2`).

Вторая программа

Код (один поток):

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

int main()
{
    int fd1 = open("alphabet.txt", O_RDONLY);
    int fd2 = open("alphabet.txt", O_RDONLY);
    int rc1 = 1, rc2 = 1;

    char c;
    while (rc1 == 1 || rc2 == 1)
    {
        rc1 = read(fd1, &c, 1);
        if (rc1 == 1)
        {
            write(1, &c, 1);
        }
        rc2 = read(fd2, &c, 1);
        if (rc2 == 1)
        {
            write(1, &c, 1);
        }
    }

    fprintf(stdout, "\n");

    return 0;
}
```

Код (два потока):

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

void *thread_run(void *fs)
{
    int fd = *(int *)fs;
    int flag = 1;

    char c;
    while (flag == 1)
```

```
13      {
14          flag = read(fd, &c, 1);
15          if (flag == 1)
16          {
17              write(1, &c, 1);
18          }
19      }
20
21      return NULL;
22  }
23
24  int main()
25  {
26      int fd1 = open("alphabet.txt", O_RDONLY);
27      int fd2 = open("alphabet.txt", O_RDONLY);
28      int rc = 1;
29
30      pthread_t td;
31      pthread_create(&td, NULL, thread_run, &fd2);
32      usleep(1);
33
34      char c;
35      while (rc == 1)
36      {
37          rc = read(fd1, &c, 1);
38          if (rc == 1)
39          {
40              write(1, &c, 1);
41          }
42      }
43
44      pthread_join(td, NULL);
45      fprintf(stdout, "\n");
46
47      return 0;
48  }
```

Результат работы:

```
hackfeed@hackfeed:~/oslabs/lab_05/src$ ./prog2.exe
aabbccddeeffgghhiijjkkllmmnnooppqqrrsttuuvvwwxxyyzz
```

Рисунок 4 – Программа с одним потоком

```
hackfeed@hackfeed:~/oslabs/lab_05/src$ ./prog2t.exe
aabcbdcedfeghfigjklmhniojpqkrlsmtnuovwxpyqzrstuvwxyz
```

Рисунок 5 – Программа с двумя потоками

Связь структур:

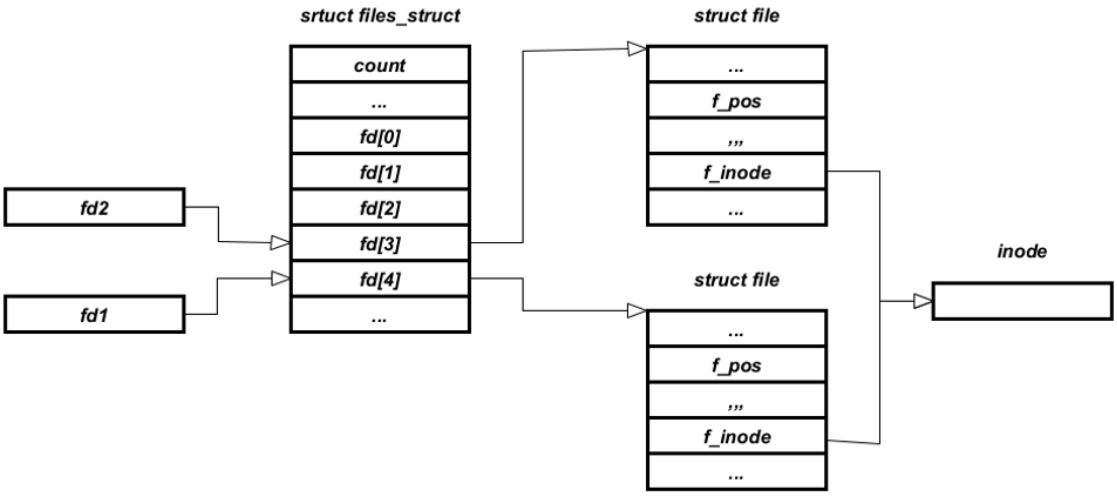

Рисунок 6 – Связь структур

Анализ: в программе работа с файлом происходит напрямую через файловые дескрипторы, но так как дескрипторы создаются дважды, дескрипторы разные), то в программе имеются 2 различные `struct file`, имеющие одинаковые поля `struct inode *f_inode`. Так как структуры разные, то посимвольная печать просто выведет содержание файла дважды, причем в однопоточной реализации вывод будет вида 'aabbcc...', а в случае с многопоточной реализацией вывод второго потока начнется чуть позже (так как на создание потока требуется время), поэтому содержимое файла будет полностью дважды, но выводы потоков перемешаются (что можно увидеть в результате работы для многопоточной реализации).

Третья программа

Код (один поток):

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

int main()
{
    FILE *fd1 = fopen("out.txt", "w");
    FILE *fd2 = fopen("out.txt", "w");

    for (char c = 'a'; c <= 'z'; ++c)
    {
        if (c % 2 == 0)
        {
            fprintf(fd1, "%c", c);
        }
        else
        {
            fprintf(fd2, "%c", c);
        }
    }

    fclose(fd1);
    fclose(fd2);

    return 0;
}
```

Код (два потока):

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/stat.h>

void *thread_run(void *fs)
{
    struct stat info;

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

```
16      stat("outt.txt", &info);
17      fprintf(stdout, "fclose outt.txt in second thread: inode is %ld, filesize is %ld\n",
            info.st_ino, info.st_size);
18
19      return NULL;
20  }
21
22  int main()
23  {
24      struct stat info;
25
26      FILE *fd1 = fopen("outt.txt", "w");
27      stat("outt.txt", &info);
28      fprintf(stdout, "first fopen outt.txt in main thread: inode is %ld, filesize is
            %ld\n", info.st_ino, info.st_size);
29
30      FILE *fd2 = fopen("outt.txt", "w");
31      stat("outt.txt", &info);
32      fprintf(stdout, "second fopen outt.txt in main thread: inode is %ld, filesize is
            %ld\n", info.st_ino, info.st_size);
33
34      pthread_t td;
35      pthread_create(&td, NULL, thread_run, fd2);
36
37      for (char c = 'b'; c <= 'z'; c += 2)
38      {
39          fprintf(fd1, "%c", c);
40      }
41      fclose(fd1);
42      stat("outt.txt", &info);
43      fprintf(stdout, "fclose outt.txt in main thread: inode is %ld, filesize is %ld\n",
            info.st_ino, info.st_size);
44      pthread_join(td, NULL);
45
46      return 0;
47  }
```

Результат работы:

```
hackfeed@hackfeed:~/oslabs/lab_05/src$ ./prog3.exe
hackfeed@hackfeed:~/oslabs/lab_05/src$ cat out.txt
acegikmoqsuwyhackfeed@hackfeed:~/oslabs/lab_05/src$
```

Рисунок 7 – Программа с одним потоком

```
hackfeed@hackfeed:~/oslabs/lab_05/src$ ./prog3t.exe
first fopen outt.txt in main thread: inode is 75, filesize is 0
second fopen outt.txt in main thread: inode is 75, filesize is 0
fclose outt.txt in main thread: inode is 75, filesize is 13
fclose outt.txt in second thread: inode is 75, filesize is 13
hackfeed@hackfeed:~/oslabs/lab_05/src$ cat outt.txt
acegikmoqsuwyhackfeed@hackfeed:~/oslabs/lab_05/src$
```

Рисунок 8 – Программа с двумя потоками

Связь структур:

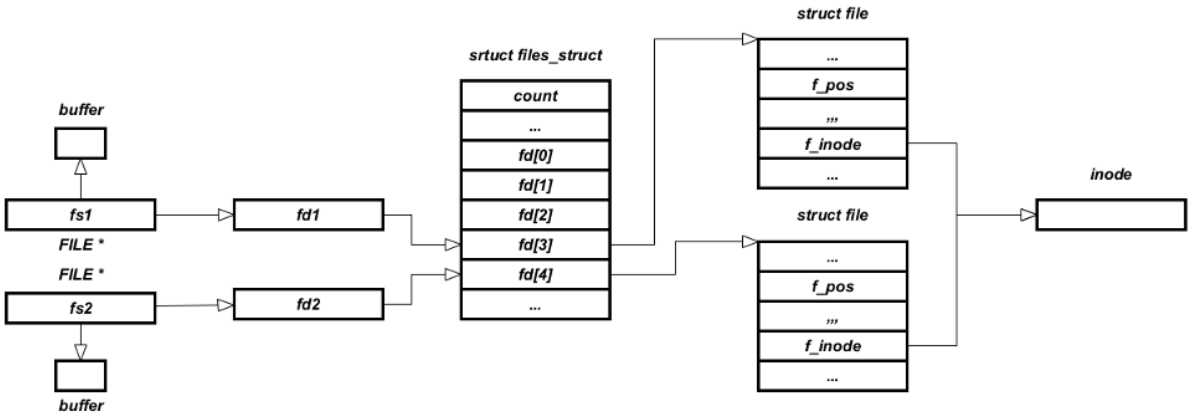

Рисунок 9 – Связь структур

Анализ: в программе работа с файлом происходит с помощью функций стандартной библиотеки и структур `FILE *`. С помощью функции `fopen()` файл `alphabet.txt` открывается дважды на запись (`mode = 'w'`).

Буфер создается при первой операции записи, размер буфера = 4096 байт (4 Кб, размер 1 страницы памяти).

В конце работы программы делается вызов функции стандартной библиотеки `fclose()`. Во время этого вызова в программе содержимое буфера переносится в файл (в общем случае это может происходить по одной из 3 причин: буфер полон, вызван `fflush()`, вызван `fclose()`). Так как `fclose(fd2)` вызывается позже, содержимое файла перепишется содержимым буфера структуры `fd2`.