



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 небуферизованный ввод-вывод»

Студент ИУ7-61Б  
(Группа)

\_\_\_\_\_  
(Подпись, дата)

Мицевич М. Д.  
(И. О. Фамилия)

Преподаватель

\_\_\_\_\_  
(Подпись, дата)

Рязанова Н. Ю.  
(И. О. Фамилия)

2022 г.

## Задание

В лабораторной работе анализируется результат выполнения трех программ. Программы демонстрируют открытие одного и того же файла несколько раз. Реализация, когда файл открывается в одной программе несколько раз выбрана для простоты. Однако, как правило, такая ситуация возможна в системе, когда один и тот же файл несколько раз открывают разные процессы или потоки одного процесса. При выполнении асинхронных процессов такая ситуация является вероятной и ее надо учитывать, чтобы избежать потери данных, получения неверного результата при выводе данных в файл или чтения данных не в той последовательности, в какой предполагалось, и в результате при обработке этих данных получения неверного результата. Каждую из приведенных программ надо выполнить в многопоточном варианте: в программах создается дополнительный поток, а работа с открываемым файлом выполняется в потоках.

Проанализировать работу приведенных программ и объяснить результаты их работы.

### `struct _IO_FILE`

```
struct _IO_FILE
{
int _flags;                /* High-order word is _IO_MAGIC; rest
    is flags. */

/* The following pointers correspond to the C++ streambuf
    protocol. */
char *_IO_read_ptr;        /* Current read pointer */
char *_IO_read_end;        /* End of get area. */
char *_IO_read_base;       /* Start of putback+get area. */
char *_IO_write_base;      /* Start of put area. */
char *_IO_write_ptr;       /* Current put pointer. */
char *_IO_write_end;       /* End of put area. */
char *_IO_buf_base;        /* Start of reserve area. */
char *_IO_buf_end;         /* End of reserve area. */

/* The following fields are used to support backing up and undo.
    */
```

```

char *_IO_save_base; /* Pointer to start of non-current get area.
    */
char *_IO_backup_base; /* Pointer to first valid character of
    backup area */
char *_IO_save_end; /* Pointer to end of non-current get area. */

struct _IO_marker *_markers;

struct _IO_FILE *_chain;

int _fileno;
int _flags2;
__off_t _old_offset; /* This used to be _offset but it's too
    small. */

/* 1+column number of pbase(); 0 is unknown. */
unsigned short _cur_column;
signed char _vtable_offset;
char _shortbuf[1];

_IO_lock_t *_lock;
#ifdef _IO_USE_OLD_IO_FILE
};

```

## Программа №1

Листинг 1 – Однопоточная версия

```

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

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

    FILE *fs1 = fdopen(fd, "r");
    char buff1[20];
    setvbuf(fs1, buff1, _IOFBF, 20);

    FILE *fs2 = fdopen(fd, "r");
    char buff2[20];
    setvbuf(fs2, buff2, _IOFBF, 20);

```

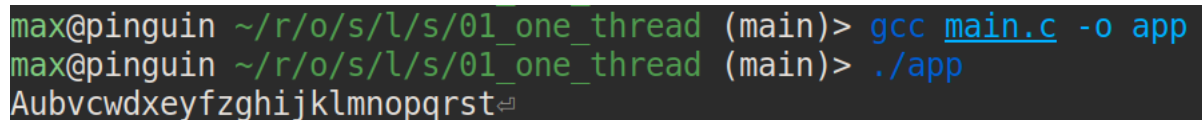
```

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



```

max@penguin ~/r/o/s/l/s/01_one_thread (main)> gcc main.c -o app
max@penguin ~/r/o/s/l/s/01_one_thread (main)> ./app
Aubvcwdxeyfzghijklmnopqrst

```

Рисунок 1 – Однопоточная версия 1-ой программы

## Листинг 2 – Многопоточная версия

```

#include <stdio.h>
#include <fcntl.h>
#include <pthread.h>
void thread(int args[2])
{
    FILE *fs = fdopen(args[0], "r");
    char buff[20];
    setvbuf(fs, buff, _IOFBF, 20);

    char c;
    int flag = fscanf(fs, "%c", &c);
    while (flag == 1) {
        fprintf(stdout, "thread_%d:_%c\n", args[1], c);
        flag = fscanf(fs, "%c", &c);
    }
}

```

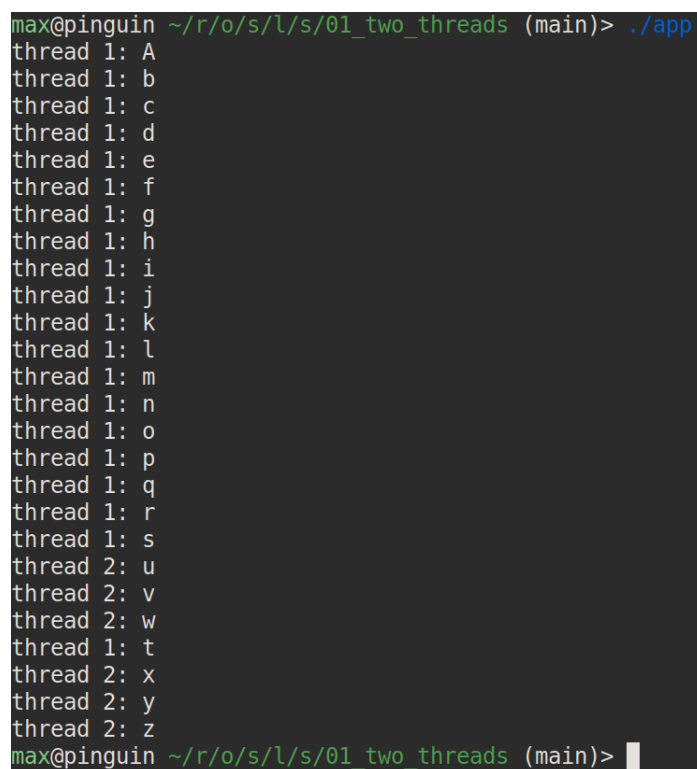
```

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

    pthread_t tid[2];
    int data1[2] = {fd, 1};
    int data2[2] = {fd, 2};
    for (int i = 0; i < 2; i++) {
        int *data = i == 0 ? data1 : data2;
        if (pthread_create(&tid[i], NULL, thread, data)) {
            printf("Error: can't create thread\n");
            return -1; }
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    return 0;
}

```

## Результат работы



```

max@penguin ~/r/o/s/l/s/01_two_threads (main)> ./app
thread 1: A
thread 1: b
thread 1: c
thread 1: d
thread 1: e
thread 1: f
thread 1: g
thread 1: h
thread 1: i
thread 1: j
thread 1: k
thread 1: l
thread 1: m
thread 1: n
thread 1: o
thread 1: p
thread 1: q
thread 1: r
thread 1: s
thread 2: u
thread 2: v
thread 2: w
thread 1: t
thread 2: x
thread 2: y
thread 2: z
max@penguin ~/r/o/s/l/s/01_two_threads (main)>

```

Рисунок 2 – Многопоточная версия 1-ой программы

## Объяснение результатов

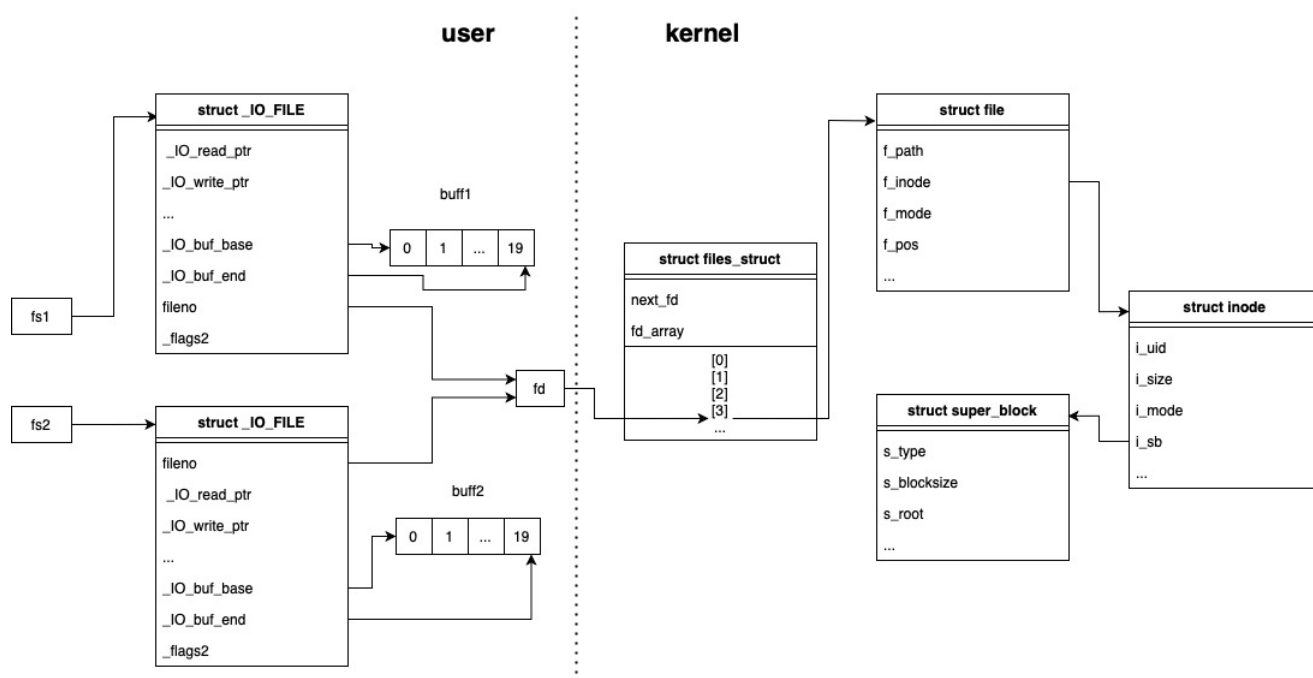


Рисунок 3 – Связь структур

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.

- в однопоточной версии вызовы `fscanf(fs1, ...)` и `fscanf(fs2, ...)` происходят

поочерёдно, поэтому символы в результате соответствуют поочерёднему чтению из первого и второго буферов;

- в многопоточной версии порядок вызовов `fscanf(fs, ...)` двух потоков не определён, поэтому символы в результате соответствуют чтению из буферов в произвольном порядке.

## Программа 2

Листинг 3 – Однопоточная версия

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main()
{
    char c;
    int fd1 = open("alphabet.txt", O_RDONLY);
    int fd2 = open("alphabet.txt", O_RDONLY);
    int flag = 1;
    while(flag)
    {
        if (read(fd1, &c, 1) == 1) {
            write(1, &c, 1);
            if (read(fd2, &c, 1) == 1) {
                write(1, &c, 1);
            } else {
                flag = 0;
            }
        } else {
            flag = 0;
        }
    }
    return 0;
}
```

## Результат работы

```
max@penguin ~/r/o/s/l/s/02_one_thread (main)> gcc main.c -o app
max@penguin ~/r/o/s/l/s/02_one_thread (main)> ./app
AAbbccddeeffgghhiijjkkllmmnnooppqrrssttuuvvwxxyyzz↵
max@penguin ~/r/o/s/l/s/02_one_thread (main)> █
```

Рисунок 4 – Однопоточная версия 2-ой программы

Листинг 4 – Многопоточная версия

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main()
{
    char c;
    int fd1 = open("alphabet.txt", O_RDONLY);
    int fd2 = open("alphabet.txt", O_RDONLY);
    int flag = 1;
    while(flag)
    {
        if (read(fd1, &c, 1) == 1) {
            write(1, &c, 1);
            if (read(fd2, &c, 1) == 1) {
                write(1, &c, 1);
            } else {
                flag = 0;
            }
        } else {
            flag = 0;
        }
    }
    return 0;
}
```



## Результат работы

```
max@penguin ~/r/o/s/l/s/02_two_threads (main)> ./app
AbcdefghijklmAnbocpdqerfsgthuivjwklmznoqrstuvwxyz↵
max@penguin ~/r/o/s/l/s/02_two_threads (main)> █
```

Рисунок 5 – Многопоточная версия 2-ой программы

## Объяснение результатов

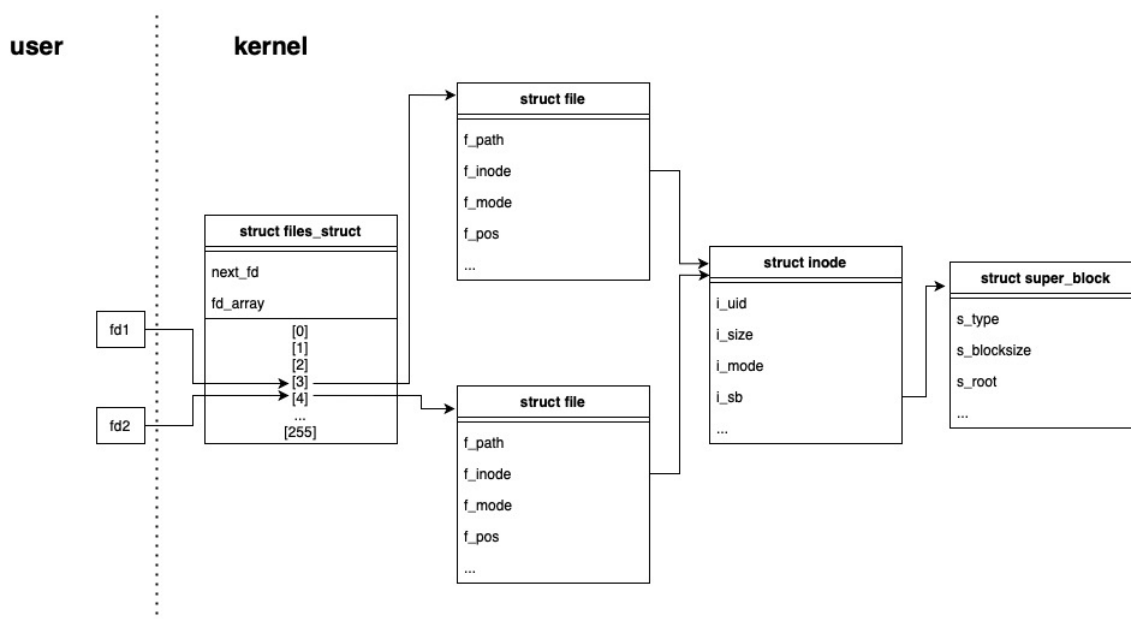


Рисунок 6 – Связь структур

Системный вызов `open()` создаёт 2 файловых дескриптора открытого файла (описанного структурами `struct file`) и возвращает индекс в массиве `fd_array` структуры `files_struct`. Таким образом, создаётся две структуры открытого файла, каждая из которых имеет собственное поле `f_pos`. Именно поэтому в результате будет получен приведённый результат:

- в однопоточной версии вызовы `read` происходят поочерёдно, поэтому символы в результате соответствуют двум поочерёдным независимым чтениям из файла;
- в многопоточной версии порядок вызовов `read` двух потоков не определён, поэтому символы в результате соответствуют двум независимым чтениям в произвольном порядке.

## Программа 3

### Листинг 5 – Программа 3

```
#include <stdio.h>
#include <sys/stat.h>
#include <pthread.h>
#define FMT_STR "FS%d: inode=%ld, size=%ld\n"
void *thread(int data) {
    int fid = (int)data;
    struct stat statbuf;
    FILE *fs = fopen("out.txt", "w");
    stat("out.txt", &statbuf);

    printf("FOPEN" FMT_STR, fid, statbuf.st_ino, statbuf.st_size);

    for (char c = 'a'; c <= 'z'; c++) {
        if ((c % 2) == (fid == 1))
            fprintf(fs, "%c", c);
    }
    fclose(fs);
    stat("out.txt", &statbuf);
    printf("FCLOSE" FMT_STR, fid, statbuf.st_ino, statbuf.st_size);
}

int main() {
    pthread_t tid[2]; int fid[2] = {0, 1};
    for (int i = 0; i < 2; i++) {
        if (pthread_create(&tid[i], NULL, thread, fid[i])) {
            printf("Error: can't create thread\n");
            return -1; }
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL); return 0;
}
```

## Результат работы

```
max@pinguin ~/r/o/s/l/s/03_threads (main)> ./app
FOPEN FS0: inode = 1587937, size = 0
FOPEN FS1: inode = 1587937, size = 0
FCLOSE FS0: inode = 1587937, size = 13
FCLOSE FS1: inode = 1587937, size = 13
max@pinguin ~/r/o/s/l/s/03_threads (main)> cat out.txt
acegikmoqsuwy
max@pinguin ~/r/o/s/l/s/03_threads (main)> █
```

Рисунок 7 – 3-я программа

## Объяснение результатов

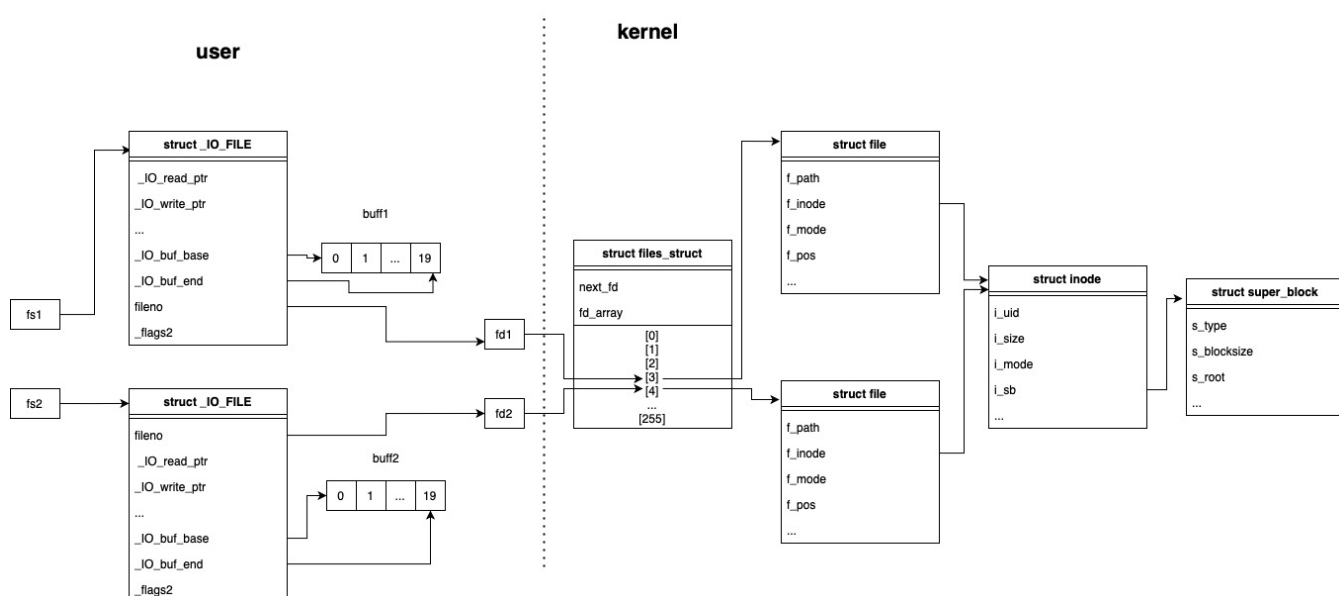


Рисунок 8 – Связь структур

С помощью `open()` открываются два потока на запись, которые имеют разные файловые дескрипторы. Нечётные буквы алфавита записываются в первый поток, чётные - во второй. Так как функция `open()` выполняет ввод-вывод с буферизацией, окончательная запись в файл осуществляется либо при полном заполнении буфера, либо при вызове функций `fclose()` или функции `fflush()`.

Так как запись производится с помощью функции `fprintf`, использующий буферизованный ввод-вывод, то запись в двух потоках будет осуществляться в буфер. Существуют 3 условия записи из буфера в файл:

- переполнение буфера;

- `fflush`;
- закрытие файла.

Так как буфер не переполняется и не вызывается `fflush`, то запись в файл в приведённой программе осуществится лишь при вызове `fclose`. Так как потоки работают с одним и тем же файлом, причём ссылаются на разные структуры `struct file` (то есть поле `f_pos` для каждого потока своё), то результат записи будет зависеть от того, какой поток вызвал `fclose` позже. При этом данные, записанные в файл ранее, будут утеряны.

## Решение проблемы

Причина: каждый дескриптор открытого файла имеет своё поле `f_pos`.

**Решение №1:** необходимо использовать режим `O_APPEND`. В данном режиме перемещение позиции в конец файла и добавление символа происходят атомарно, поэтому данные не будут утеряны.

**Решение №2:** необходимо использовать мьютекс для перемещения позиции в конец файла и записи символа. Код:

Листинг 6 – Решение с мьютексом

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

#define FMT_STR "FS%d: inode=%ld, size=%ld\n"
pthread_mutex_t lock;
void *thread(int data) {
    int fid = (int)data;
    struct stat statbuf;
    FILE *fs = fopen("out.txt", "w");
    stat("out.txt", &statbuf);

    printf("FOPEN" FMT_STR, fid, statbuf.st_ino, statbuf.st_size);

    for (char c = 'a'; c <= 'z'; c++) {
        if ((c % 2) == (fid == 1)) {
            pthread_mutex_lock(&lock);
            lseek(fileno(fs), NULL, SEEK_END);
```

```

        fprintf(fs, "%c", c);
        pthread_mutex_unlock(&lock);
    }
}
fclose(fs);
stat("out.txt", &statbuf);
printf("FCLOSE" FMT_STR, fid, statbuf.st_ino, statbuf.st_size
    );
}
int main() {
    if (pthread_mutex_init(&lock, NULL) != 0)
    {
        printf("\n_mutex_init_failed\n");
        return 1;
    }
    pthread_t tid[2]; int fid[2] = {0, 1};
    for (int i = 0; i < 2; i++) {
        if (pthread_create(&tid[i], NULL, thread, fid[i])) {
            printf("Error: can't create thread\n");
            return -1; }
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL); return 0;
}

```