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 дисциплине «Операционные системы»

Тема Буферизованный и не буферизованный ввод-вывод.

Студент Костев Д.И.

Группа ИУ7-61Б

Оценка (баллы) _____

Преподаватель Рязанова Н. Ю.

Москва — 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.

Программа №1

Код однопоточной версии

```c
#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

```
ldk@ldk-Ubuntu:~$ ./app
Aubvcwdxeyfzghijklmnopqrts
```

Код многопоточной версии

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

    char c;
    int flag = fscanf(fs, "%c", &c);
    while (flag == 1) {
        fprintf(stdout, "%c", c);
        flag = fscanf(fs, "%c", &c);
    }
}
int main() {
    int fd = open("alphabet.txt", O_RDONLY);

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

Результат работы

```
ldk@ldk-Ubuntu:~$ ./app
Abcdefghuvwxyzijklmnopqrst
```

Объяснение результатов

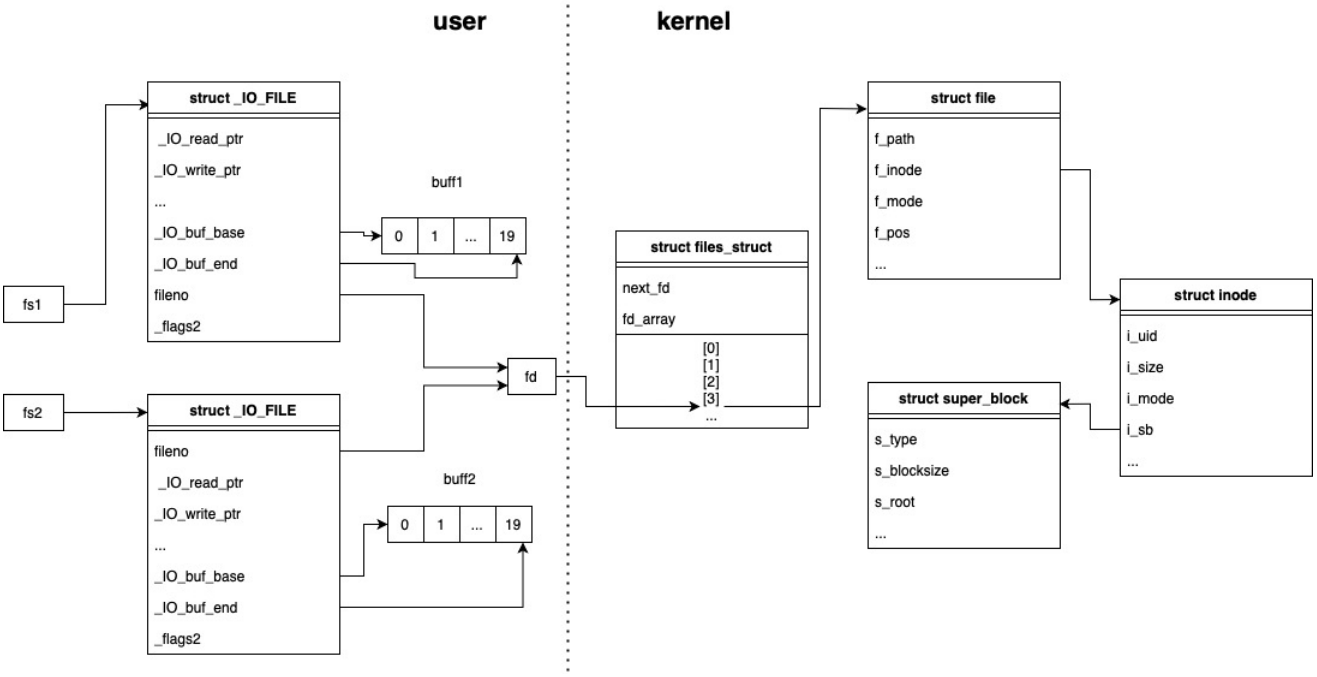

Рис. 1: Связь структур

Системный вызов open() создаёт дескриптор открытого файла (описанного структурой struct file) и возвращает индекс в массиве fd_array структуры files_struct.

Вызов fdopen() создает указатели на структуры типа _IO_FILE (fs1 и fs2), которые ссылаются на дескриптор открытого файла, созданный ранее.

С помощью setbuf для структур _IO_FILE (fs1, fs2) задаются буферы и тип буферизации _IOFBF (Fully buffered).

При первом вызове fscanf(fs1, ...) буфер структуры fs1 полностью заполнится, то есть будут прочитаны первые 20 символов, а поле f_pos структуры struct file станет равным 20. Затем при вызове fscanf(fs2, ...) оставшиеся символы считаются в буфер структуры fs2, f_pos структуры struct file станет равным 20 + количество оставшихся символов. При последующих вызовах fscanf(fs1, ...) и fscanf(fs2, ...) работа будет производиться с буферами, поэтому в результате будет получен приведённый результат:

- в однопоточной версии вызовы fscanf(fs1, ...) и fscanf(fs2, ...) происходят поочерёдно, поэтому символы в результате соответствуют поочерёдному 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

Код однопоточной версии

```c
#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

```
ldk@ldk-Ubuntu:~$ ./app
AAbbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz
```

Код многопоточной версии

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <pthread.h>
void thread()
{
    char c;
    int fd = open("alphabet.txt", O_RDONLY);
    while (read(fd, &c, 1) == 1)
    {
        write(1, &c, 1);
    }
}
int main()
{
    pthread_t tid[2];
    for (int i = 0; i < 2; i++) {
        if (pthread_create(&tid[i], NULL, thread, NULL)) {
            printf("Error: can't create thread\n");
            return -1; }
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    return 0;
}
```

Результат работы

```
ldk@ldk-Ubuntu:~$ ./app
AAbcdefghijklmnopqrstuvwxyzbcdefghijklmnopqrstuvwxyz
```

Объяснение результатов

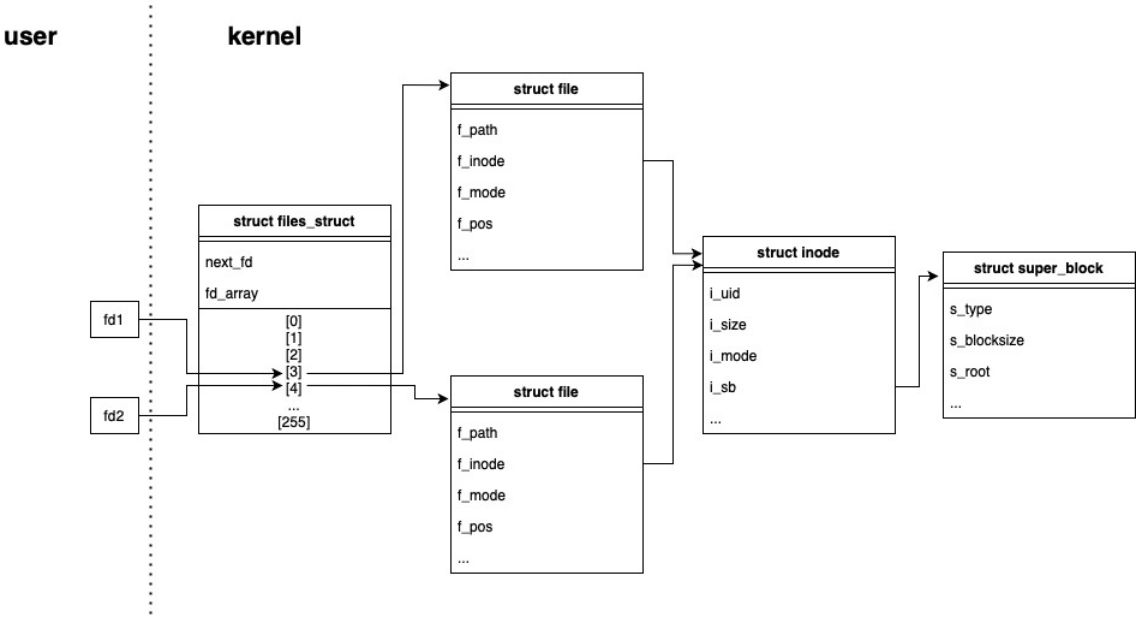

Рис. 2: Связь структур

Каждый системный вызов open() создаёт файловый дескриптор открытого файла (описанного структурами struct file) и возвращает индекс в массиве fd_array структуры files_struct. Поскольку open() вызывается два раза, 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

Код

```c
#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);
```

```
10
11      printf("FOPEN " FMT_STR, fid, statbuf.st_ino, statbuf.st_size);
12      for (char c = 'a'; c <= 'z'; c++) {
13          if ((c % 2) == (fid == 1))
14              fprintf(fs, "%c", c);
15      }
16      fclose(fs);
17      stat("out.txt", &statbuf);
18      printf("FCLOSE" FMT_STR, fid, statbuf.st_ino, statbuf.st_size);
19  }
20  int main() {
21      pthread_t tid[2]; int fid[2] = {0, 1};
22      for (int i = 0; i < 2; i++) {
23          if (pthread_create(&tid[i], NULL, thread, fid[i])) {
24              printf("Error: can't create thread\n");
25              return -1; }
26      }
27      pthread_join(tid[0], NULL);
28      pthread_join(tid[1], NULL); return 0;
29  }
```

Результат работы

```
1  ldk@ldk-Ubuntu:~$ ./app
2  FOPEN  FS0: inode = 41305511, size = 0
3  FOPEN  FS1: inode = 41305511, size = 0
4  FCLOSE FS1: inode = 41305511, size = 13
5  FCLOSE FS0: inode = 41305511, size = 13
6
7  ldk@ldk-Ubuntu:~$ cat out.txt
8  acegikmoqsuwy
9
10 ldk@ldk-Ubuntu:~$ ./app
11 FOPEN  FS1: inode = 41305511, size = 0
12 FOPEN  FS0: inode = 41305511, size = 0
13 FCLOSE FS1: inode = 41305511, size = 13
14 FCLOSE FS0: inode = 41305511, size = 13
15
16 ldk@ldk-Ubuntu:~$ cat out.txt
17 bdfhjlnprtvxz
```

Объяснение результатов

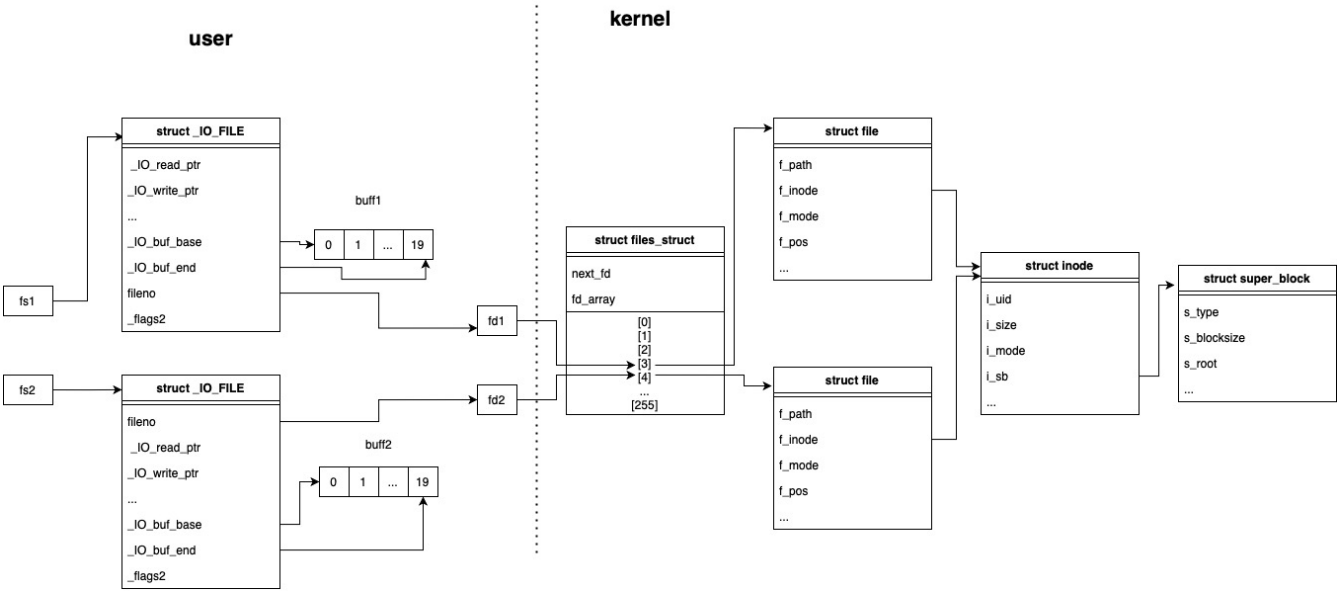

Рис. 3: Связь структур

Вызов fopen() создаёт файловый дескриптор открытого файла (описанного структурой struct file) и возвращает индекс в массиве fd_array структуры files_struct. Кроме того, fopen создаёт указатель на структуру типа _IO_FILE, которая ссылается на дескриптор открытого файла, созданный ранее. Таким образом, создаётся две структуры открытого файла, каждая из которых имеет собственное поле f_pos.

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:

```c
#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;
}
```

struct _IO_FILE

```
1  struct _IO_FILE
2  {
3    int _flags;               /* High-order word is _IO_MAGIC; rest is
       flags. */
4
5    /* The following pointers correspond to the C++ streambuf protocol. */
6    char *_IO_read_ptr;       /* Current read pointer */
7    char *_IO_read_end;       /* End of get area. */
8    char *_IO_read_base;       /* Start of putback+get area. */
9    char *_IO_write_base;       /* Start of put area. */
10   char *_IO_write_ptr;       /* Current put pointer. */
11   char *_IO_write_end;       /* End of put area. */
12   char *_IO_buf_base;       /* Start of reserve area. */
13   char *_IO_buf_end;       /* End of reserve area. */
14
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   int _flags2;
26   __off_t _old_offset; /* This used to be _offset but it's too small.  */
27
28   /* 1+column number of pbase(); 0 is unknown. */
29   unsigned short _cur_column;
30   signed char _vtable_offset;
31   char _shortbuf[1];
32
33   _IO_lock_t *_lock;
34 #ifdef _IO_USE_OLD_IO_FILE
35 };
```