

МГТУ им. БАУМАНА

ЛАБОРАТОРНАЯ РАБОТА №5

По курсу: "ОПЕРАЦИОННЫЕ СИСТЕМЫ"

Буферизованный и не буферизованный ввод-вывод

Работу выполнил: студент группы ИУ7-63Б

Наместник Анастасия

Преподаватели: Рязанова Н. Ю.

Москва, 2021

0.1 Задание

В лабораторной работе анализируется результат выполнения трех программ. Программы демонстрируют открытие одного и того же файла несколько раз. Реализация открытия файла в одной программе несколько раз выбрана для простоты. Такая ситуация возможна в системе, когда один и тот же файл несколько раз открывают разные процессы. Но для получения ситуаций аналогичных тем, которые демонстрируют приведенные программы надо было бы синхронизировать работу процессов. При выполнении асинхронных процессов такая ситуация вероятна и ее надо учитывать, чтобы избежать потери данных или получения неверного результата при выводе в файл.

0.2 Программа 1

Код программы:

```
1 //testC1O.c
2 #include <stdio.h>
3 #include <fcntl.h>
4
5 /*
6  On my machine, a buffer size of 20 bytes
7  translated into a 12-character buffer.
8  Apparently 8 bytes were used up by the
9  stdio library for bookkeeping.
10 */
11
12 int main()
13 {
14     // have kernel open connection to file alphabet.txt
15     int fd = open("alphabet.txt", O_RDONLY);
16
17     // create two a C I/O buffered streams using the above
18     connection
19     FILE *fs1 = fdopen(fd, "r");
20     char buff1[20];
21     setvbuf(fs1, buff1, _IOFBF, 20);
```

```

22 FILE *fs2 = fdopen(fd, "r");
23 char buff2[20];
24 setvbuf(fs2, buff2, _IOFBF, 20);
25
26 // read a char & write it alternately from fs1 and fs2
27 int flag1 = 1, flag2 = 2;
28 while(flag1 == 1 || flag2 == 1)
29 {
30     char c;
31     flag1 = fscanf(fs1, "%c", &c);
32     if (flag1 == 1)
33     {
34         fprintf(stdout, "%c", c);
35     }
36
37     flag2 = fscanf(fs2, "%c", &c);
38     if (flag2 == 1)
39     {
40         fprintf(stdout, "%c", c);
41     }
42 }
43 return 0;
44 }

```

Программа использует файл **alphabet.txt**, содержащий символы: `Abcdefghijklmnopqrstuvwxyz`.
В результате выполнения программы в стандартный поток вывода `stdout` запишется следующая последовательность символов: `Aubvcwdxeyfzghijklmnopqrst`.

0.3 Анализ первой программы

В стандартную библиотеку C `stdio.h` включен заголовочный файл, полный путь которого `glibc/libio/bits/types/FILE.h`, содержащий объявление структуры `FILE`:

```

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 которой содержится в файле *glibc/libio/bits/types/struct_FILE.h*:

```

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

```

```

27         too small. */
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 };

```

Структура FILE - сущность языка C и его стандартной библиотеки stdio.h. При работе со структурой FILE автоматически создается буфер, и программист работает с более высокоуровневой абстракцией. Объект типа FILE содержит информацию о файле и обращается к соответствующему файловому дескриптору Unix.

Размер буфера по дефолту в stdio.h:

```

1 /* Default buffer size. */
2 #define BUFSIZ 8192

```

Связь структур представлена на рисунке 0.1

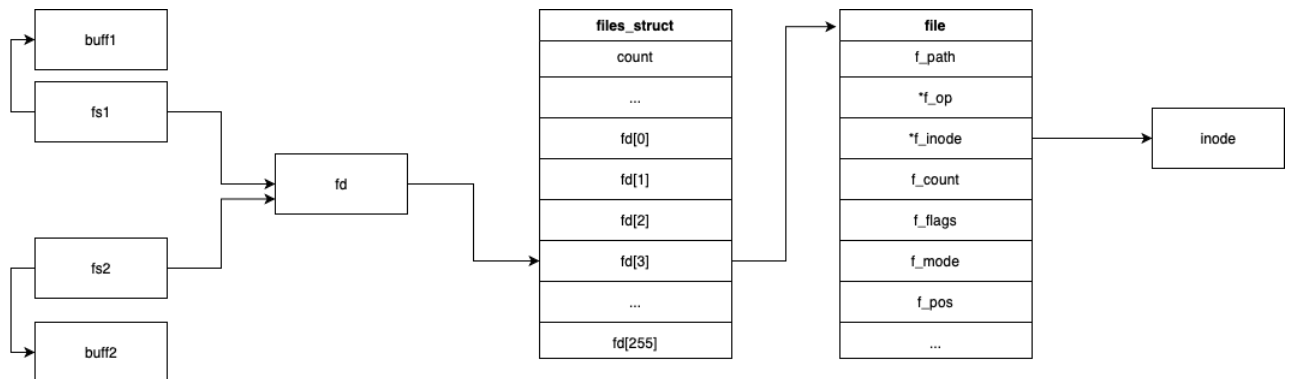


Рис 0.1: Связь структур для первой программы

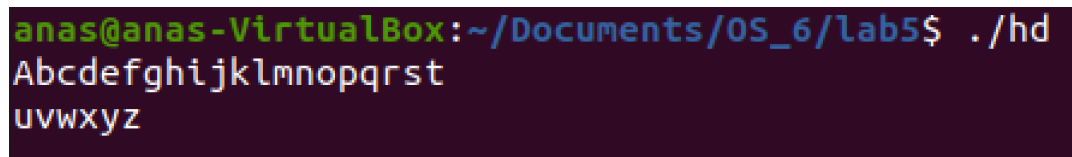
В функции main() вызывается функция open(), вызывающая, в свою очередь, системный вызов (system call) open() для чтения файла alphabet.txt, содержащего последовательность символов латинского алфавита: Abcdefghijklmnopqrstuvwxyz. Возвращаемое значение - целое неотрицательное число (int) - индекс в

массиве `fd_array[NP_OPEN_DEFAULT]` структур типа `file`, определенном в таблице открытых файлов процесса - `struct files_struct`. Полученное число является **файловым дескриптором** файла `alphabet.txt`.

Функция `fdopen()` может использоваться для инициализации структуры `FILE` файловым дескриптором. С помощью функции `fd_open()`, принимающей `fd` создаются два указателя `fs1` и `fs2` типа `struct FILE`, ссылающиеся на структуру типа `file` в массиве `fd_array[NP_OPEN_DEFAULT]` с индексом, равным `fd`.

Иными словами, `fs1` и `fs2` ссылаются на файловый дескриптор, полученный с помощью `open()`, так как `fd_open()` передается файловый дескриптор, созданный с помощью `open()`.

С помощью функции `setvbuf()` для `fs1` и `fs2` создается два буфера на 20 байт, тип буферизации - полная буферизация. Содержимое буферов `buff1` и `buff2` после вызова `fscanf()` для `fs1` и затем для `fs2` представлено на рисунке 0.2.

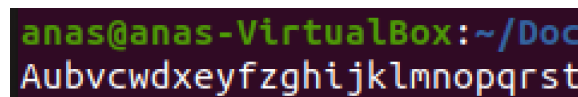


```
anas@anas-VirtualBox:~/Documents/OS_6/lab5$ ./hd
Abcdefghijklmnopqrst
uvwxyz
```

Рис 0.2: `buff1` и `buff2` после вызова `fscanf()`

Такой результат объясняется тем, что сначала полностью заполняется `buff1` первыми 20 символами - `Abcdefghijklmnopqrst`, а затем остаток данных файла `alphabet.txt` записывается в `buff2` - `uvwxyz`. Это происходит благодаря тому, что указатель `f_pos` структуры `file`, описывающей файл, имеющий дескриптор `fd`, увеличивается на 20 после выполнения первой операции чтения `fscanf(fs1, "%c &c)`. Так как обе структуры `FILE` `fs1` и `fs2` ссылаются на одну и ту же запись в таблице открытых файлов процесса, при вызове `fscanf(fs2, "%c &c)` произойдет обращение к тому же указателю `f_pos`.

Далее в цикле содержимое буферов `buff1` и `buff2` попеременно выводится в стандартный поток вывода `stdout`, что можно видеть на рисунке 0.3.



```
anas@anas-VirtualBox:~/Doc
Aubvcwdxeyfzghijklmnopqrst
```

Рис 0.3: Результат работы первой программы

0.4 Программа 2 (2 процесса)

Код программы (главный процесс):

```
1 //testKernelIO.c
2 #include <fcntl.h>
3
4 int main()
5 {
6     char c;
7     // have kernel open two connection to file alphabet.txt
8     int fd1 = open("alphabet.txt", O_RDONLY);
9     int fd2 = open("alphabet.txt", O_RDONLY);
10    // read a char & write it alternatingly from connections
11    // fs1 & fd2
12    int flag = 1;
13
14    while( flag )
15    {
16        if ( read(fd1, &c, 1) == 1 )
17            write(1, &c, 1);
18        else
19            flag = 0;
20
21        if ( read(fd2, &c, 1) == 1 )
22            write(1, &c, 1);
23        else
24            flag = 0;
25    }
26    return 0;
27 }
```

Код программы (процесс-предок и процесс-потомок):

```
1 //testKernelIO.c
2 #include <stdio.h>
3 #include <fcntl.h>
4 #include <unistd.h>
5
6 int main()
7 {
```

```

8   char c;
9   // have kernel open two connection to file alphabet.txt
10  int fd1;
11  int fd2 = open("alphabet.txt",O_RDONLY);
12  // read a char & write it alternatingly from connections
   fs1 & fd2
13  int flag = 1;
14  int pid;
15
16  if ((pid = fork()) == -1)
17  {
18      perror("fork Error\n");
19      return 0;
20  }
21
22  if (pid == 0)
23      fd1 = open("alphabet.txt",O_RDONLY);
24
25  while(flag)
26  {
27      if (pid == 0) //child process
28      {
29          if (read(fd1,&c,1) == 1) //-1 if fails to read
30              write(1,&c,1);
31          else
32              flag = 0;
33          printf("Child process = %d\n", getpid());
34      }
35
36      else
37      {
38          if (read(fd2,&c,1) == 1)
39              write(1,&c,1);
40          else
41              flag = 0;
42          printf("Parent process = %d\n", getpid());
43      }
44  }
45
46  return 0;

```


0.5 Анализ второй программы (2 процесса)

0.5.1 Главный процесс

Файл `alphabet.txt` открывается 2 раза с помощью 2-х вызовов `open()` с режимом доступа на чтение - `O_RDONLY`. Следовательно, в таблице открытых файлов процесса будет находиться 2 дескриптора, с которыми будут связаны две записи в системной таблице открытых файлов, указатель `*f_inode` которых будет указывать на один `inode`, так как дескрипторы `fd1` и `fd2` описывают один файл.

Связь структур представлена на рисунке 0.4.

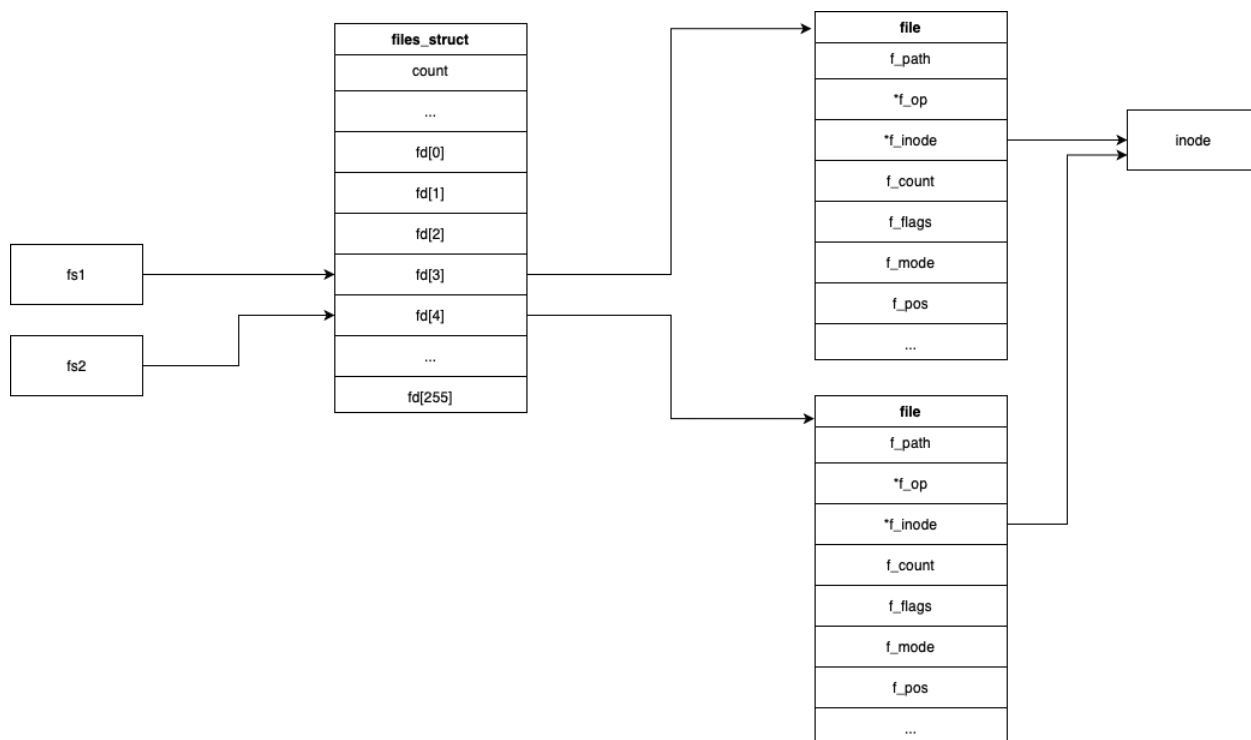


Рис 0.4: Связь структур для второй программы (главный процесс)

Результат работы второй программы без использования процесса-потомка представлен на рисунке 0.5.

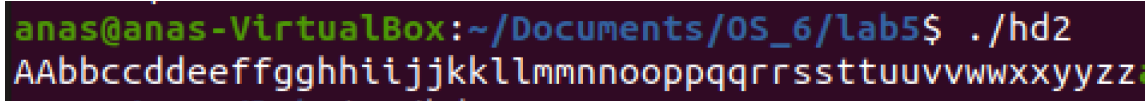
A terminal window with a dark background. The prompt is 'anas@anas-VirtualBox:~/Documents/OS_6/lab5\$'. The command './hd2' has been executed, and the output is 'AAbbccddeeffgghhiijjkkllmmnnoopppqqrrssttuuvvwwxxyyzz'.

Рис 0.5: Результат работы второй программы (главный процесс)

Символы записываются в стандартный поток вывода `stdout` попеременно, дублируясь, так как для дескрипторов `fd1` и `fd2` имеется две разных структуры `file`, и, соответственно, указатель позиции на чтение/запись в файле `*f_pos` будет независимым для них.

0.5.2 Процесс-предок и процесс-потомок

Чтобы создать процесс-потомок, используется вызов `fork()`, после которого процесс-предок считывает из одного открытого файла (с использованием `fd1`), а процесс-потомок - из другого открытого файла (с использованием `fd2`). При этом дескриптор `fd2` был создан процессом-потомком, благодаря чему создается еще одна структура `struct files_struct` - таблица открытых файлов процесса-потомка.

Связь структур для процесса-предка и процесса-потомка представлена на рисунке 0.6.

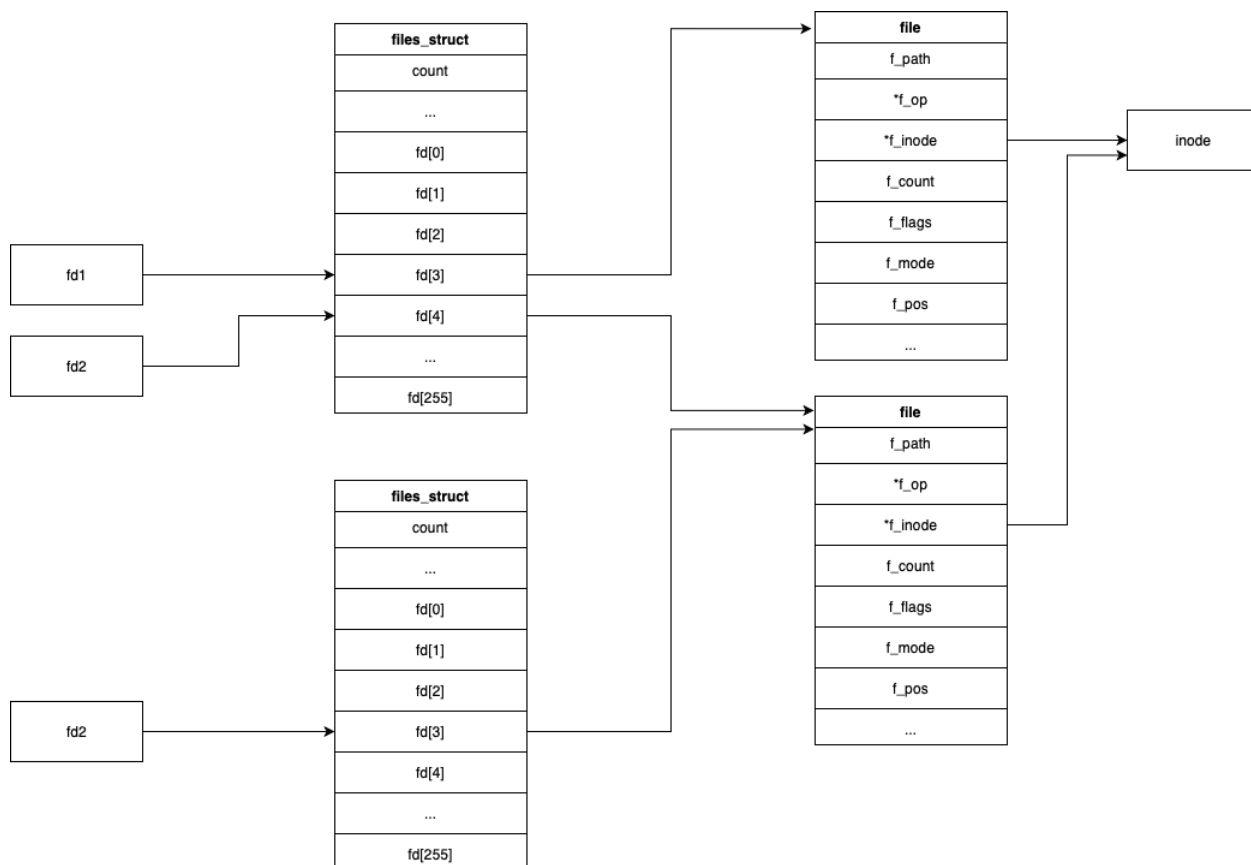


Рис 0.6: Связь структур для второй программы (процесс-предок и процесс-потомок)

Результат работы второй программы с использованием процесса-потомка представлен на рисунке 0.7.

```

anas@anas-VirtualBox:~/Documents/OS_6/lab5$ ./hd2
Abcdefghijklmnopqrstuvwxyz
anas@anas-VirtualBox:~/Documents/OS_6/lab5$ Abcdefghijklmnopqrstuvwxyz
  
```

Рис 0.7: Результат работы второй программы (процесс-предок и процесс-потомок)

На рисунках 08-0.10 представлен результат программы с более подробным анализом вывода за счет вывода на экран идентификаторов процессов.

```
anas@anas-VirtualBox:~/Documents/OS_6/lab5$ ./hd2
AParent process = 3773
bParent process = 3773
cParent process = 3773
dParent process = 3773
eParent process = 3773
fParent process = 3773
gParent process = 3773
hParent process = 3773
iParent process = 3773
jParent process = 3773
kParent process = 3773
lParent process = 3773
mParent process = 3773
nParent process = 3773
oParent process = 3773
pParent process = 3773
qParent process = 3773
rParent process = 3773
sParent process = 3773
tParent process = 3773
uParent process = 3773
vParent process = 3773
wParent process = 3773
```

Рис 0.8: Анализ вывода с 2-умя процессами (1 часть)

```
xParent process = 3773
yParent process = 3773
zParent process = 3773
Parent process = 3773
process = 3773
anas@anas-VirtualBox:~/Documents/OS_6/lab5$ AChild process = 3774
bChild process = 3774
cChild process = 3774
dChild process = 3774
eChild process = 3774
fChild process = 3774
gChild process = 3774
hChild process = 3774
iChild process = 3774
jChild process = 3774
kChild process = 3774
lChild process = 3774
mChild process = 3774
nChild process = 3774
oChild process = 3774
pChild process = 3774
qChild process = 3774
rChild process = 3774
sChild process = 3774
tChild process = 3774
uChild process = 3774
vChild process = 3774
wChild process = 3774
xChild process = 3774
yChild process = 3774
zChild process = 3774
Child process = 3774
process = 3774
anas@anas-VirtualBox:~/Documents/OS_6/lab5$
```

Рис 0.9: Анализ вывода с 2-умя процессами (2 часть)

```
tChild process = 3774
uChild process = 3774
vChild process = 3774
wChild process = 3774
xChild process = 3774
yChild process = 3774
zChild process = 3774
Child process = 3774
process = 3774
anas@anas-VirtualBox:~/Documents/OS_6/lab5$
```

Рис 0.10: Анализ вывода с 2-умя процессами (3 часть)

0.6 Программа 2 (2 потока)

Код программы:

```
1 //testKernelIO.c
2 #include <stdio.h>
3 #include <pthread.h>
4 #include <fcntl.h>
5 #include <unistd.h> //System calls
6 #include <stdlib.h>
7 #define ERROR_CREATE_THREAD -11
8 #define ERROR_JOIN_THREAD -12
9
10 typedef struct file_descr
11 {
12     int fd;
13 } file_descr_t;
14
15 void *read_from_file(void *args)
16 {
17     file_descr_t *file_descriptor = (file_descr_t*) args;
18     char c;
19
20     while(read(file_descriptor->fd, &c, 1) == 1)
21         write(1, &c, 1);
22
23     write(1, " ", 1);
24 }
25
26 int main()
27 {
28     char c;
29     int status, status_addr;
30     // have kernel open two connection to file alphabet.txt
31     int fd1 = open("alphabet.txt", O_RDONLY);
32     int fd2 = open("alphabet.txt", O_RDONLY);
33
34     pthread_t thread;
35
36     // read a char & write it alternatingly from connections
```

```

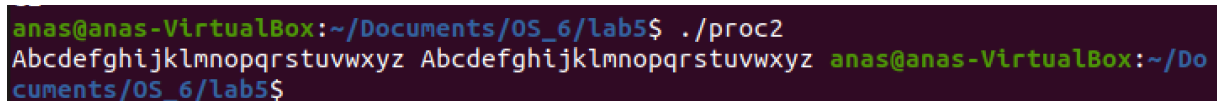
37     fs1 & fd2
status = pthread_create(&thread, NULL, read_from_file, &
38     fd1);
39     if (status != 0)
40     {
41         printf("Error: couldn't create a thread. Status = %d\n",
, status);
42         exit(ERROR_CREATE_THREAD);
43     }
44     read_from_file(&fd2);
45
46     status = pthread_join(thread, (void**)&status_addr);
47     if (status != 0)
48     {
49         printf("Error: couldn't join thread. Status = %d\n",
, status);
50         exit(ERROR_JOIN_THREAD);
51     }
52
53     return 0;
54 }

```

0.7 Анализ второй программы (2 потока)

С помощью функции *pthread_create()*, объявленной в заголовочном файле *pthread.h*, создается поток **thread**, который будет читать из одного открытого файла, в то время как главный поток читает из другого открытого файла. Таким образом, реализовано параллельное чтение из двух открытых файлов, имеющих дескрипторы *fd1* и *fd2*.

Результат работы второй программы с использованием 2-х потоков представлен на рисунке 0.11.



```

anas@anas-VirtualBox:~/Documents/OS_6/lab5$ ./proc2
Abcdefghijklmnopqrstuvwxyz Abcdefghijklmnopqrstuvwxyz anas@anas-VirtualBox:~/Do
cuments/OS_6/lab5$

```

Рис 0.11: Результат работы второй программы (2 потока)

Связь структур в случае с использованием 2-ух потоков представлена на рисунке 0.4.

0.8 Программа 3

Код программы:

```
1 #include <stdio.h>
2 #include <fcntl.h>
3 #include <pthread.h>
4 #include <sys/stat.h>
5 #include <time.h>
6 #include <stdlib.h> //exit
7 #define ALPHABET "Abcdefghijklmnopqrstuvwxyz"
8 #define ALPHABET_SIZE 26
9 #define FILENAME "proc3_alphabet.txt"
10 #define ERROR_CREATE_THREAD -11
11 #define ERROR_JOIN_THREAD -12
12
13 void print_stat(char *fd, char *operation, struct stat
    statbuf)
14 {
15     printf("%s %s:\ninode: %d\nsize: %d\nl/O_block_size: %d\
    nLast modification of file data: %s\n",
16         fd, operation, (int)statbuf.st_ino,
17         (int)statbuf.st_size, (int)statbuf.st_blksize, ctime(&
    statbuf.st_mtime));
18 }
19
20
21 typedef struct file_descr
22 {
23     FILE *fd;
24 } file_descr_t;
25
26 void *read_from_file(void *args)
27 {
28     file_descr_t *file_descriptor = (file_descr_t*) args;
29     char c;
30
```



```

31     for (int i = 0; i < ALPHABET_SIZE; i+=2)
32         fprintf(file_descriptor->fd, "%c", ALPHABET[i]); //
           acegikmoqsuwy
33     }
34
35     int main()
36     {
37         struct stat statbuf;
38         pthread_t thread;
39         int status, status_addr;
40
41         FILE *fd1 = fopen(FILENAME, "w");
42         stat(FILENAME, &statbuf);
43         print_stat("fd1", "fopen", statbuf);
44
45         FILE *fd2 = fopen(FILENAME, "w");
46         stat(FILENAME, &statbuf);
47         print_stat("fd2", "fopen", statbuf);
48
49         status = pthread_create(&thread, NULL, read_from_file, &
           fd1);
50         if (status != 0)
51         {
52             printf("Error: couldn't create a thread. Status = %d\n",
           status);
53             exit(ERROR_CREATE_THREAD);
54         }
55
56         for (int i = 1; i < ALPHABET_SIZE; i+=2)
57             fprintf(fd2, "%c", ALPHABET[i]);
58
59         status = pthread_join(thread, (void**)&status_addr);
60         if (status != 0)
61         {
62             printf("Error: couldn't join thread. Status = %d\n",
           status);
63             exit(ERROR_JOIN_THREAD);
64         }
65
66         fclose(fd1);

```

```

67 stat(FILENAME, &statbuf);
68 print_stat("fd1", "fclose", statbuf);
69
70
71 fclose(fd2);
72 stat(FILENAME, &statbuf);
73 print_stat("fd2", "fclose", statbuf);
74
75 return 0;
76 }

```

0.9 Анализ третьей программы

Один и то же файл открывается 2 раза на запись. Для этого с помощью функции *fopen()* объявляются 2 указателя на структуру типа FILE - fd1 и fd2. В обоих случаях функциях *fopen()* принимает имя файла - **"proc3_alphabet.txt"** и режим доступа к файлу - **"w"**. Затем в цикле с помощью функции *fprintf()* в файл записываются символы латинского алфавита, при этом созданные файловые дескрипторы используются попеременно - один за другим, что реализовано с помощью создания потока. Один поток читает из одного открытого файла, второй поток - из другого открытого файла. Так как по умолчанию используется полная буферизация, информация запишется из буфера в файл в 3 случаях:

1. Буфер заполнился.
2. Была вызвана функция *fclose()*.
3. Была вызвана функция *fflush()*.

Таким образом, в таблице открытых файлов процесса будет 2 записи, соответствующие 2-ум созданным файловым дескрипторам, которые, в свою очередь, будут связаны с 2-умя структурами типа *file*, ссылающимися на один *inode*.

Связь структур представлена на рисунке 0.11.

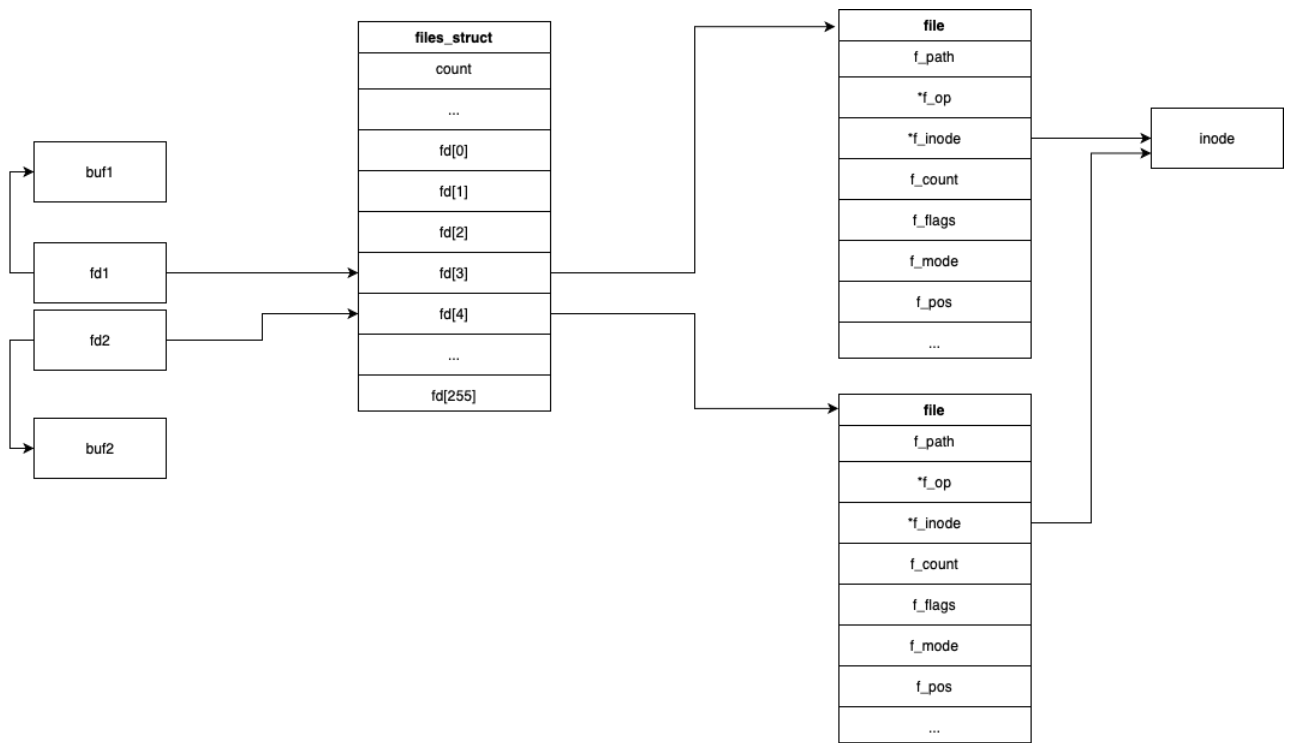


Рис 0.11: Связь структур для третьей программы

Результат работы третьей программы представлен на рисунке 0.12.

```
1 bdfhjlnprtvxz
```

Рис 0.12: Результат работы третьей программы

Такой вывод объясняется порядком вызова функции `fclose()`. Функция `fclose()` отделяет указанный поток от связанного с ним файла. Если поток использовался для вывода данных, то все данные, содержащиеся в буфере, сначала записаны в файл.

Так как сначала `fclose()` была вызвана для `fd1`, символы в буфере для `fd1` были записаны в файл. Затем `fclose()` была вызвана для `fd2`, и, так как указатель смещения в файле `*f_pos` разный для `fd1` и `fd2`, символы из

буфера для fd2 запишутся в файл, начиная с 0-го смещения. Тем самым, произойдет утеря данных.

Результат работы третьей программы, если поменять местами вызовы fclose(), представлен на рисунке 0.13.

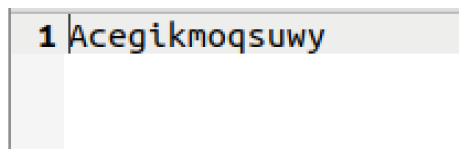


Рис 0.13: Результат работы третьей программы с другим порядком fclose()

0.9.1 Структура stat

```

1 struct stat {
2     dev_t      st_dev;           /* ID of device
3     containing file */
4     ino_t      st_ino;           /* Inode number */
5     mode_t     st_mode;          /* File type and
6     mode */
7     nlink_t    st_nlink;         /* Number of hard
8     links */
9     uid_t      st_uid;           /* User ID of
10    owner */
11    gid_t      st_gid;           /* Group ID of
12    owner */
13    dev_t      st_rdev;          /* Device ID (if
14    special file) */
15    off_t      st_size;          /* Total size , in
16    bytes */
17    blksize_t   st_blksize;       /* Block size for
18    filesystem I/O */
19    blkcnt_t    st_blocks;        /* Number of 512B
20    blocks allocated */
21
22    /* Since Linux 2.6, the kernel supports
23    nanosecond
24    precision for the following timestamp
25    fields.

```

```

15      For the details before Linux 2.6, see
16      NOTES. */
17      struct timespec st_atim; /* Time of last
18      access */
19      struct timespec st_mtim; /* Time of last
20      modification */
21      struct timespec st_ctim; /* Time of last
22      status change */
23
24      #define st_atime st_atim.tv_sec      /* Backward
25      compatibility */
26      #define st_mtime st_mtim.tv_sec
27      #define st_ctime st_ctim.tv_sec
28      };

```

Источник: Linux manual page.

```

1 stat(const char *restrict pathname,
2      struct stat *restrict statbuf);

```

stat возвращает информацию о файле *pathname* и заполняет буфер *statbuf*.

Результат работы третьей программы с использованием структуры *stat* представлен на рисунке 0.14.

```
anas@anas-VirtualBox:~/Documents/OS_6/lab5$ ./hd3
fd1_fopen:
inode: 939477
size: 0
I/O_block_size: 4096
Last modification of file data: Wed Apr 28 12:59:08 2021

fd2_fopen:
inode: 939477
size: 0
I/O_block_size: 4096
Last modification of file data: Wed Apr 28 12:59:08 2021

fd1_fclose:
inode: 939477
size: 13
I/O_block_size: 4096
Last modification of file data: Wed Apr 28 12:59:08 2021

fd2_fclose:
inode: 939477
size: 13
I/O_block_size: 4096
Last modification of file data: Wed Apr 28 12:59:08 2021
```

Рис 0.14: Результат работы третьей программы с использованием структуры stat