|  |  |
| --- | --- |
| **Gerb-BMSTU_01** | **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»\_\_\_\_\_\_

**Лабораторная работа № 5**

**По дисциплине: “Операционные системы”**

|  |  |
| --- | --- |
| **Тема: “ Буферизованный и не буферизованный ввод-вывод”**  **Студент:** Мередова Айджахан  **Группа:** ИУ7-66Б  **Преподаватель:** Рязанова Н.Ю. |  |

Москва

2021

**Задача: анализ особенностей работы функций ввода-вывода в UNIX/LINUX.**

**Программа 1.**

**Листинг программы:**

#include <stdio.h>

#include <fcntl.h>

int main()

{

// have kernel open connection to file alphabet.txt

int fd = open("alphabet.txt",O\_RDONLY);

// create two a C I/O buffered streams using the above connection

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);

// read a char & write it alternatingly from fs1 and fs2

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);

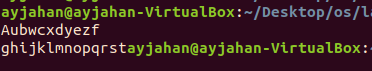
}

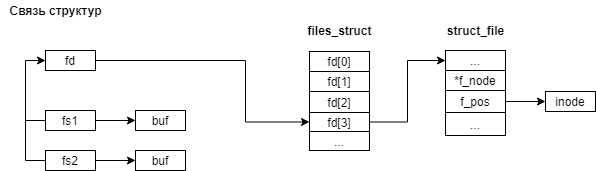
}

return 0;

}

**Результат работы:**



****

**Анализ:**

С помощью системного вызова **open()** создается дескриптор файла. Файл открывается только на чтение, так как передается флаг O\_RDONLY.

При успешном завершении системного вызова появляется новый дескриптор открытого файла alphabet.txt и запись в системной таблице открытых файлов.

Функция fdopen() создает два объекта типа FILE, которые связываются с открытым файлом, на который ссылается файловый дескриптор fd.

Структура **struct files\_struct** определяет дескрипторы файлов, открытых процессом.

struct files\_struct

{ atomic\_t count; /\* счетчик ссылок на данную структуру \*/

spinlock\_t file\_lock; /\* блокировка для защиты данной структуры \*/

int max\_fds; /\* максимальное количество файловых объектов \*/

int max\_fdset; /\* максимальное количество файловых дескрипторов \*/

int next\_fd; /\* номер следующего файлового дескриптора \*/

struct file \*\*fd; /\* массив всех файловых объектов \*/

fd\_set \*close on exec; /\* файловые дескрипторы, которые должны закрываться при вызове exec() \*/

fd\_set \*open\_fds; /\* указатель на дескрипторы открытых файлов \*/

fd\_set close\_on\_exec init; /\* первоначальные файлы для закрытия при вызове exec() \*/

fd\_set open\_fds\_init; /\* первоначальный набор файловых дескрипторов \*/ struct file \*fd\_array[NR\_OPEN\_DEFAULT]; /\* массив файловых объектов \*/

};

Структура **struct file** определяет дескриптор открытого файла в системе.

struct file {

union {

struct llist\_node fu\_llist;

struct rcu\_head fu\_rcuhead;

} f\_u;

struct path f\_path;

struct inode \*f\_inode; /\* cached value \*/

const struct file\_operations \*f\_op;

/\*

\* Protects f\_ep\_links, f\_flags.

\* Must not be taken from IRQ context.

\*/

spinlock\_t f\_lock;

enum rw\_hint f\_write\_hint;

atomic\_long\_t f\_count;

unsigned int f\_flags;

fmode\_t f\_mode;

struct mutex f\_pos\_lock;

loff\_t f\_pos;

struct fown\_struct f\_owner;

const struct cred \*f\_cred;

struct file\_ra\_state f\_ra;

u64 f\_version;

#ifdef CONFIG\_SECURITY

void \*f\_security;

#endif

/\* needed for tty driver, and maybe others \*/

void \*private\_data;

#ifdef CONFIG\_EPOLL

/\* Used by fs/eventpoll.c to link all the hooks to this file \*/

struct list\_head f\_ep\_links;

struct list\_head f\_tfile\_llink;

#endif /\* #ifdef CONFIG\_EPOLL \*/

struct address\_space \*f\_mapping;

errseq\_t f\_wb\_err;

} \_\_randomize\_layout

При создании буфера размер для данного потока выбирается системой. Чтобы его изменить используется системный вызов **setvbuf()**. В данной программе системный вызов **setvbuf()** изменяет тип буферизации для каждого объекта FILE на полную буферизацию, а также явно задает размер буфера 20 байт.

При первом вызове **fscanf()** буфер структуры FILE заполняется до тех пор, пока он не будет заполнен полностью, либо пока не будет достигнут конец файла. Так как буфер имеет размер 20 байт, а файл содержит 26 байт данных (26 букв латинского алфавита), то после первого вызова **fscanf()** в буфере первой структуры FILE будут находиться первые 20 байт файла (то есть буквы от A до t). И так как оба объекта FILE связаны с одним и тем же файловым дескриптором, то позиция в файле будет определяться для обоих файловых потоков ввода полем **f\_pos** стурктуры **struct file,** на которую ссылается указанный дескриптор файла. Поэтому после второго вызова **fscanf()** в буфере второй структуры FILE окажутся последние 6 байт файла (то есть от символа u по z).

**Программа 2**

**Листинг кода:**

#include <fcntl.h>

int main(){

char c;

// have kernel open for two connection to our file

int fd1 = open("alphabet.txt",O\_RDONLY);

int fd2 = open("alphabet.txt",O\_RDONLY);

int break\_flag = 1;

// read a char & write it alternatingly from connections fs1 & fd2

while(break\_flag){

if (read(fd1,&c,1)!= 1)

break\_flag = 0;

write(1,&c,1);

if (read(fd2,&c,1)!= 1)

break\_flag = 0;

write(1,&c,1);

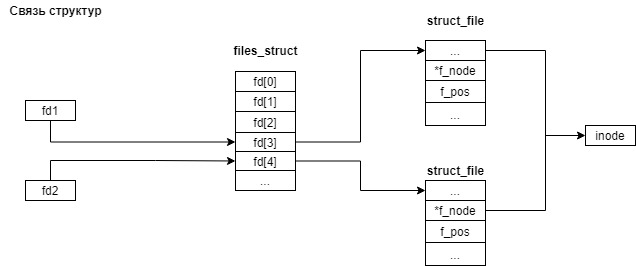
}

return 0;

}

**Результат работы:**





**Анализ:**

В данной программе файл открывается дважды для чтения с помощью системного вызова **open().** Создаются два различных структуры **struct file,** которые описывают открытый файл, которые связаны с одним и тем же физическим файлом. Текущие позиции в файле для каждой структуры будут изменяться независимо друг от друга. Поэтому чтение с использованием одной структуры не затягивает текущую позицию в другой структуре, и каждый символ из физического файла будет продублирован.

**Программа 3**

**Листинг кода:**

#include <stdio.h>

int main()

{

FILE \*f1;

FILE \*f2;

f1 = fopen("alphabet.txt","w");

f2 = fopen("alphabet.txt","w");

int flag;

for (char c = 'a'; c <= 'z'; c++)

{

if (!flag)

fprintf(f1, "%c", c);

if (!flag)

fprintf(f2, "%c", c);

flag = !flag;

}

fclose(f1);

fclose(f2);

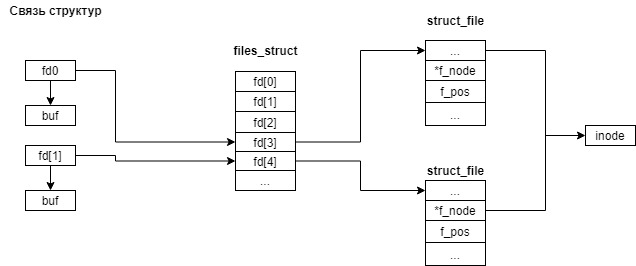
return 0;

}

**Результат работы:**

Содежимое файла alphabet.txt:





**Анализ:**

В данной программе с помощью функции **fopen()** два раза открывается на запись файл alphabet.txt, то есть создаются два разных структур **struct file,** в которых поля **f\_pos** при вызове функций ввода/вывода меняются независимо (аналогично предыдущему примеру).

В результате поочередной записи букв латинского алфавита в первый буфер будут записаны нечетные символы, а во второй буфер – четные.

Запись из буфера в файл происходит автоматически в следующих случаях:

* при заполнении буфера;
* по завершении процесаа;
* при вызове функций **fclose()** или **fflush()**

Сначала закрывается первый поток, поэтому изначально в файл alphabet.txt осуществляется запись буфера первого потока (acegikmoqsuwy), при этом данные записываются с начала файла, затем происходит закрытие второго потока и запись его буфера в файл alphabet.txt. При этом, так как оба объекта FILE связаны с разными структурами **struct file**, то значения их полей **f\_pos** изменяются независимо, и запись второго буфера будет также произведена с начала файла, то есть данные, записанные в файл из буфера первого потока, будут перезаписаны.

**ВЫВОД:**

При буферизованном вводе/выводе необходимо учитывать факт записи/чтения данных из буфера, т.к. неправильные действия с данными, записываемыми(или считываемыми) в (из) файл(а), могут привести к неправильной последовательности данных (программа 1)или даже к их потере (программа 3).

Также необходимо учитывать, что при одновременном открытии одного и того же файла создается дескриптор открытого файла. Каждый дескриптор **struct file** имеет поле **f\_pos,**  указатель на позицию чтения или записи в логическом файле.