Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
Московский Государственный Технический Университет
имени Н.Э. Баумана

Лабораторная работа № 2.

По курсу «Операционные системы».
Тема: «Анализ особенностей работы функций ввода-вывода в UNIX/Linux».

Выполнил: Петухов И.С.
Группа: ИУ7-61
Проверил: Рязанова Н.Ю

Москва, 2016.

Задание: лабораторная работа – анализ особенностей работы функций ввода-вывода в UNIX/Linux

Ввод-вывод с использование системного вызова open() и ввод-вывод с использованием библиотечной функции fopen() библиотеки stdio.h

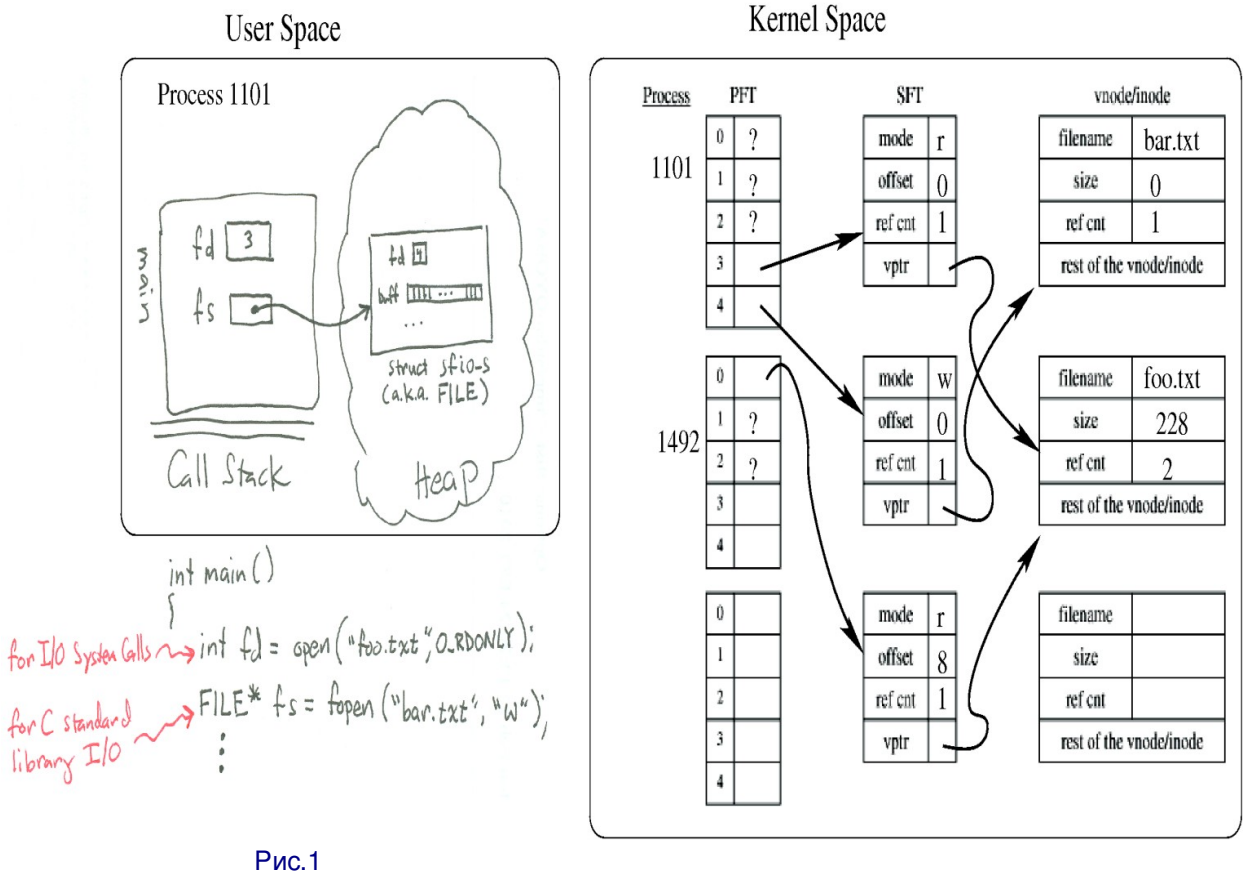

Рис.1

- Когда мы выполняем ввод.вывод через стандартную библиотеку C (stdio.h) (т.е. используя fscanf, fprintf, fopen, fclose), вызовы на открытие файла, его закрытие, чтение и запись (to open, close, read and write) не завершаются немедленно, так как это просто библиотечные функции, которые делают системные вызовы. Другими словами стандартная библиотека ввода-вывода C является просто верхним уровнем для выполнения системных вызовов.

Рассмотрим ситуацию, показанную на рис.1. Здесь процесс выполняет системный вызов open("foo.txt",O.RDONLY) для чтения файла foo.txt и вызов функции fopen("bar.txt","w") из стандартной библиотеки для записи в файл bar.txt.
Обратите внимание на то, что на рисунке показан другой процесс. Что Вы можете сказать об этой ситуации, если другой процесс создан, например, shell?

```
bash$ ./prog1 < foo.txt > bar.txt 2> bar.txt
```

Необходимо помнить, что стандартный ввод-вывод буферизуется!

Ввод/вывод потоком берет данные как поток отдельных символов. Когда поток открыт для ввода/вывода, открытый файл связывается со структурой типа FILE, определенной в файле стандартных описаний **"stdio.h"**. Указатель на структуру FILE возвращается при открытии файла. Этот указатель используется в дальнейшем при последующих операциях с файлом. Ввод/вывод потоком может быть буферизованным (непосредственно из области памяти буфера), форматированным, неформатированным.
Функции **fclose, fopen, fprintf, fscanf, fgetc, fputc, fgets, fputs, fcloseall, getc, gets, putc, puts, getchar** работают с форматированными данными.
Функции **fread, fwrite** работают с неформатированными данными.
Фугкции **scanf, printf, getchar, putchar** работают со стандартными потоками **stdin, stdout.**

Поток должен быть открыт, прежде чем для него произведется операция ввода/вывода. Исключение составляют следующие потоки:

stdin - стандартный ввод;
stdout - стандартный вывод;
stderr - стандартные ошибки;
stdaux - стандартный порт;
stdprn - стандартная печать.

Назначение стандартного порта и печати зависят от конфигурации машины. Обычно эти потоки указывают на вспомогательный порт и принтер.
Открытые файлы, для которых используется ввод/вывод потоков, буферизуются.
Не буферизуются стандартные потоки.
Буфера, размещенные в системе, не доступны пользователю.

Задание

Проанализировать работу приведенных программ и объяснить результаты их работы.

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

/*
On my machine, a buffer size of 20 bytes
translated into a 12-character buffer.
Apparently 8 bytes were used up by the
stdio library for bookkeeping.
 */

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;
}
```

```
//testKernelIO.c
#include <fcntl.h>

int main()
```

```
{
  // have kernel open two connection to file alphabet.txt
  int fd1 = open("alphabet.txt",O_RDONLY);
  int fd2 = open("alphabet.txt",O_RDONLY);


  // read a char & write it alternatingly from connections fs1 & fd2
  while(1)
  {
    char c;
    if (read(fd1,&c,1) != 1) break;
    write(1,&c,1);
    if (read(fd2,&c,1) != 1) break;
    write(1,&c,1);
  }

  return 0;
}
```

```
Файл alphabet.txt

Abcdefghijklmnopqrstuvwxyz
```

Дополнительно написать программу, которая открывает один и тот же файл два раза с использованием библиотечной функции fopen(). Для этого объявляются два файловых дескриптора. В цикле записать в файл буквы латинского алфавита поочередно передавая функции fprintf() то первый дескриптор, то – второй.
Результат прокомментировать.

Замечание 1:
Можно выделить 4 причины для использования fopen() а не open():

1. fopen() выполняет ввод-вывод с буферизацией, что может оказаться значительно быстрее, чем с использованием open();
2. fopen() делает перевод конца строки, если только файл не открыт в двоичном режиме , который может быть очень полезен, если ваша программа иногда переносится в среду, отличную от Unix;
3. FILE * дает возможность использовать fscanf() и другие функции stdio.h;
4. Ваш код однажды может быть использован на другой платформе, которая предполагает использование только ANSI C, которая не поддерживает функцию open().

Однако, при более детальном разборе можно сказать, что конец строки чаще мешает, чем помогает, а детальный разбор fscanf() часто заставляет заменять функцию, на более полезные.
Большинство платформ поддерживают open(). И, наконец, буферизация. В случае, если операции чтения и записи в файл выполняются последовательно, буферизация представляется действительно полезной и обеспечивающей высокую скорость выполнения.
Но это может привести к некоторым проблемам. Например, Вы предполагаете, что данные записаны в файл, но данные там еще отсутствуют. Необходимо помнить о своевременном выполнении fclose() и fflush().
Если Вы выполняете fseek() буферизация перестает быть полезной.
Все это справедливо, если постоянно работать с сокетами и при этом выполнять неблокирующий ввод-вывод. Но нельзя отрицать полезность FILE* для выполнения сложных разборов текста.
Замечание 2:

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

```
     FILE *fopen(const char *filename, const char *mode);
Открытие файла с режиме добавления (в качестве первого символа в
     аргумент режима – “a” ) приводит к тому, что все последующие операции
записи в файл работать с  current end-of-file, даже если вмешиваются вызовы
FSEEK(3C). Если два независимых процесса откроют один и  тот же файл для
добавления данных, каждый процесс может свободно писать  в файл без опасения
нарушить вывод сохраненный другим процессом. Информация  будет записана в
файл в том порядке, в котором процессы записывали ее в файл.

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname,int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

Вызов open () создает новый файловый дескриптор для открытого файла, запись в общесистемной таблице открытых файлов. Эта запись регистрирует смещение в файле и флаги состояния файла (модифицируемые с помощью the fcntl (2) операции F_SETFL). Дескриптор файла является ссылкой на одну из этих записей; эта ссылка не влияет, если путь впоследствии удален или изменен ссылаться на другой файл. Новый дескриптор открытого файла изначально не разделяется с любым другим процессом, но разделение может возникнуть через ***fork****(* (2).

O_APPEND
Файл открывается в режиме добавления - O_APPEND . Перед каждым вызовом *write* (2), смещение в файле устанавливается в конец файла, как если бы выполнялся вызов LSEEK (2). O_APPEND может привести к повреждению файлов на файловых системах NFS, если несколько процессов добавляют данные в файл одновременно. Это потому, что NFS не поддерживает добавление в файл, так что клиент ядра должен имитировать его, что не может быть сделано без состояния гонки.

Программа testCIO.c:

```c
//testCIO.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:

aubvcwdxeyfzg

hijklmnopqrst

Объяснение:

1. С помощью системного вызова open (строка 7), мы создаем одну запись в общесистемной таблице открытых файлов (open file table - OFT). В этой записи, помимо всего прочего, указано текущее смещение, относительно начала файла. Сразу после вызова open, смещение равно нулю. Возвращается файловый дескриптор (число), соответствующий этой записи.
2. В строках 9, 13 вызывается fdopen. Он открывает поток данных по дескриптору файла, и возвращает указатель на файловую структуру _IO_FILE.
3. С помощью функции setvbuf (строки 11 и 15) , мы указываем, каким буфером пользоваться каждой файловой структуре.

4. Таким образом, созданы две файловые структуры fs1 и fs2, которые будут пользоваться разными буферами, но работать будут с одной и той же записью в OFT (соответственно, поле "смещение" в этой записи будет общим для файловых структур fs1 и fs2).
5. Во время первого вызова функции fscanf на файловой структуре fs1 (строка 21) , т.к. буфер пустой, произойдет считывание файла в буфер этой файловой структуры. Будут считаны 20 символов ("abcdefghigklmnopqrst") (т.к. мы установили буфер на 20 элементов) начиная с позиции 0 (т.к. в начальный момент в поле "смещение" записи OFT записано число 0). После этого, поле "смещение" будет равно 20.
6. Эта функция fscanf напечатает первый символ из буфера файловой структуры fs1 - 'a' .
7. Во время первого вызова функции fscanf на файловой структуре fs2 (строка 23) , т.к. буфер пустой, произойдет считывание файла в буфер этой файловой структуры. Символы будут считываться, начиная с позиции 20 (т.к. в этот момент в поле "смещение" записи OFT уже записано число 20 из-за предыдущего шага #5). 7 символов поместятся в буффер ("uvwxyz") + символ перевода строки (т.к. после 7 символа файл закончился).
8. Эта функция fscanf напечатает первый символ из буфера файловой структуры fs2 - 'u' .
9. Далее продолжая цикл while (строка 18) буду напечатаны остальные символы из буфферов файловых структур fs1 и fs2.

Программа testKernelIO.c:

```c
#include <fcntl.h>

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

  while(1)
  {
    char c;
    if (read(fd1,&c,1) != 1) break;
    write(1,&c,1);
    if (read(fd2,&c,1) != 1) break;
    write(1,&c,1);
  }

  return 0;
}
```

Результат:

aabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz

Объяснение:

1. В строка 5 и 6 вызывается два раза системный вызов open. В результате в OFT будет создано две записи, на каждую из которых можно сослаться используя возвращаемый дескриптор. Таким образом, в каждой записи OFT поле "смещение" будет равно нулю.
2. В строке 11 вызывается системный вызов read. Будет считан 1 символ, начиная со смещения, указанного в записи OFT, которой соответствует передаваемый файловый дескриптор fd1 в аргументы функции read. (смещение в этот момент будет равно 0)
3. В строке 12 напечатается этот символ – 'a'.
4. В строке 13 вызывается read для файлового дескриптора fd2, при этом смещение, указанное в OFT будет равно 0 (так как это другая запись, нежели в шаге 2 для файлового дескриптора fd1).
5. В строке 14 напечатается этот символ – 'a'.
6. Продолжая цикл while, будут напечатаны остальные символы из файла.

Выводы.

Функция fopen() возвращает указатель на структуру _IO_FILE и в своей реализации использует системный вызов open(). Функции библиотеки stdio.h (fscanf, fprintf) используют буферизацию. Это делается для увеличения производительности, так как следующие вызовы fscanf() будут считывать данные из буфера, а не из файла до тех пор, пока не считается весь буфер, затем он снова заполняется из файла.

Системные вызовы read() и write() не используют буфер при работе.