Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

«Московский государственный технический университет имени Н.Э. Баумана»
(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»

Студент группы ИУ7-63Б: М. В. Заколесник
(Фамилия И.О.)
Преподователь: Н. Ю. Рязанова
(Фамилия И.О.)

Москва, 2019

1 CIO.c

Текст программы

```c
//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
  // associates a stream with the existing file descriptor
  FILE *fs1 = fdopen(fd,"r");
  char buff1[20];
  setvbuf(fs1,buff1,_IOFBF,20); // set fully buffering

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

  // read a char and 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;
}
```

Вывод

```
aubvcwdxeyfzg
hijklmnopqrst
```

Анализ

Анализ работы программы и результатов: в самом начале программы создается дескриптор файла “alphabet.txt” в таблице файловых дескрипторов процесса и запись в таблице открытых файлов. В записи этой таблицы хранится текущее смещение указателя в файле, которое используется во всех операциях чтения и записи в файл, а также режим открытия файла (O_RDONLY, O_WRONLY или O_RDWR). При выполнении операции чтения-записи система выполняет неявный сдвиг указателя. Далее создаются два объекта типа FILE которые ссылаются на созданный файловый дескриптор и при помощи функцией $setvbuf(\dots)$ мы задаем размер буфера каждого объекта равный 20 символам. При первом вызове $fscanf(fs1, "\%c\,\&c)$, буфер первого файлового дескриптора предварительно заполняется первыми 20 символами - abcdefghijklmnopqrst, а при втором $fscanf(fs2, "\%c\,\&c)$ буфер второго файлового дескриптора заполняется оставшейся частью – uvwxyz (т. к. обе структуры FILE ссылаются на одну и ту же запись в таблице открытых файлов, а при первом чтении указатель в файле уже сместился на 20 позиций). Далее в цикле происходит поочередная печать по одному символу, который уже берется не из файла, а из соответствующего буфера файлового дескриптора. Поэтому мы можем наблюдать такие результаты.

2 testKernelIO.c

Текст программы

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

Вывод

```
1  aabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz
```

Анализ

В данной программе мы создали два дескриптора файла “alphabet.txt” и две различные записи в таблице открытых файлов системы. В этом случае, каждая запись имеет своё независимое смещение в файле. Поэтому при поочередной печати символа в цикле мы имеем, что каждый указатель смещается независимо друг от друга.

3 FOpen.c

```
1  #include <stdio.h>
2  int main() {
3
4    FILE* fd[2];
5    fd[0] = fopen("FOpen_output.txt","w");
6    fd[1] = fopen("FOpen_output.txt","w");
7
8    int curr = 0;
9
10   for(char c = 'a'; c <= 'z'; c++, curr = ((curr != 0) ? 0 : 1))
11   {
12       fprintf(fd[curr], "%c", c);
13   }
14   fclose(fd[0]);
15   fclose(fd[1]);
16   return 0;
17 }
```

Вывод

```
1  bdfhjlnprtvxz
```

Анализ

при вызове функции fopen() создаются 2 файловых дескриптора и 2 записи в таблице открытых файлов. Вспомним, что при вызове fprintf(...), запись производится в буфер.И только тогда, когда буфер будет заполнен полностью или если, будут вызваны функции fclose(...), fflush(...) данные будут записаны в файл. По этой причине, когда в программе вызвалась функция fclose(fs1), в файл записались все буквы английского алфавита, которые были в буфере файлового дескриптора fs1 (acegikmoqsuwy), тогда как после вызова функции fclose(fs2), содержимое файла удалилось (т.к. мы открыли файл для записи с режимом “w”), и записалась информация из буфера fs2.

4 Структура FILE

Листинг 1: Структура FILE для OS X.

```
typedef struct __sFILE {
  unsigned char *_p;  /* current position in (some) buffer */
  int _r;    /* read space left for getc() */
  int _w;    /* write space left for putc() */
  short _flags;   /* flags, below; this FILE is free if 0 */
  short _file;    /* fileno, if Unix descriptor, else -1 */
  struct __sbuf _bf; /* the buffer (at least 1 byte, if !NULL) */
  int _lbfsize; /* 0 or -_bf._size, for inline putc */

  /* operations */
  void  *_cookie; /* cookie passed to io functions */
  int (* _Nullable _close)(void *);
  int (* _Nullable _read) (void *, char *, int);
  fpos_t  (* _Nullable _seek) (void *, fpos_t, int);
  int (* _Nullable _write)(void *, const char *, int);

  /* separate buffer for long sequences of ungetc() */
  struct __sbuf _ub; /* ungetc buffer */
  struct __sFILEX *_extra; /* additions to FILE to not break ABI */
  int _ur;    /* saved _r when _r is counting ungetc data */

  /* tricks to meet minimum requirements even when malloc() fails */
  unsigned char _ubuf[3]; /* guarantee an ungetc() buffer */
  unsigned char _nbuf[1]; /* guarantee a getc() buffer */

  /* separate buffer for fgetln() when line crosses buffer boundary */
  struct __sbuf _lb; /* buffer for fgetln() */

  /* Unix stdio files get aligned to block boundaries on fseek() */
  int _blksize; /* stat.st_blksize (may be != _bf._size) */
  fpos_t  _offset;  /* current lseek offset (see WARNING) */
} FILE;
```

5 Вывод

Исходя из вышеприведенных рассуждений, можно сделать несколько выводов.

1. Предпочтительней использовать функцию $fopen()$, т.к. $fopen()$ выполняет ввод-вывод с буферизацией, что может оказаться значительно быстрее, чем с использованием $open()$, $FILE*$ дает возможность использовать $fscanf()$ и другие функции $stdio.h$.

2. Следует помнить о буферизации и вовремя использовать $fclose()$ для записи в файл.

3. С острожностью использовать $fflush()$, т.к. она оставляет поток открытым.

4. Необходимо следить за режимом, с котором открывается поток.

5. Созданный новый дескриптор открытого файла изначально не разделяется с любым другим процессом, но разделение может возникнуть через $fork()$.

6. Функции $fscanf, fprintf, fopen, fclose$ являются обертками высшего уровня над системными вызовами $open, close, read, write$.