4. Работа с файлами.

Файлы представляют собой области памяти на внешнем носителе (как правило магнитном диске), предназначенные для:

- хранения данных, превосходящих по объему память компьютера (меньше, разумеется, тоже можно);
- долговременного хранения информации (она сохраняется при выключении машины).

В *UNIX* и в *MS DOS* файлы не имеют предопределенной структуры и представляют собой просто линейные **массивы байт.** Если вы хотите задать некоторую структуру хранимой информации - вы должны позаботиться об этом в своей программе **сами.**

Файлы отличаются от обычных массивов тем, что

- они могут изменять свой размер;
- обращение к элементам этих массивов производится не при помощи операции индексации [], а при помощи специальных системных вызовов и функций;
- доступ к элементам файла происходит в так называемой "позиции чтения/записи", которая автоматически продвигается при операциях чтения/записи, т.е. файл просматривается последовательно. Есть, правда, функции для произвольного изменения этой позиции.

Файлы имеют **имена** и организованы в иерархическую древовидную структуру из **каталогов** и простых файлов. Об этом и о системе именования файлов прочитайте в документации по *UNIX*.

4.1.

Для работы с каким-либо файлом наша программа должна **открыть** этот файл - установить связь между именем файла и некоторой переменной в программе. При открытии файла в ядре операционной системы выделяется "связующая" структура *file* "**открытый файл**", содержащая:
f_offset: указатель позиции чтения/записи, который в дальнейшем мы будем обозначать как **RWptr.** Это *long*-число, равное расстоянию в байтах от начала файла до позиции чтения/записи;
f_flag: режимы открытия файла: чтение, запись, чтение и запись, некоторые дополнительные флаги;
f_inode: расположение файла на диске (в *UNIX* - в виде ссылки на *I-узел* файла*);
и кое-что еще.

У каждого процесса имеется таблица открытых им файлов - это массив ссылок на упомянутые "связующие" структуры**. При открытии файла в этой таблице ищется

```
- длина файла                  long   di_size;
- номер владельца файла        int    di_uid;
- коды доступа и тип файла     ushort di_mode;
- время создания и последней модификации
              time_t di_ctime, di_mtime;
- начало таблицы блоков файла  char   di_addr[...];
- количество имен файла        short  di_nlink;
и.т.п.
```

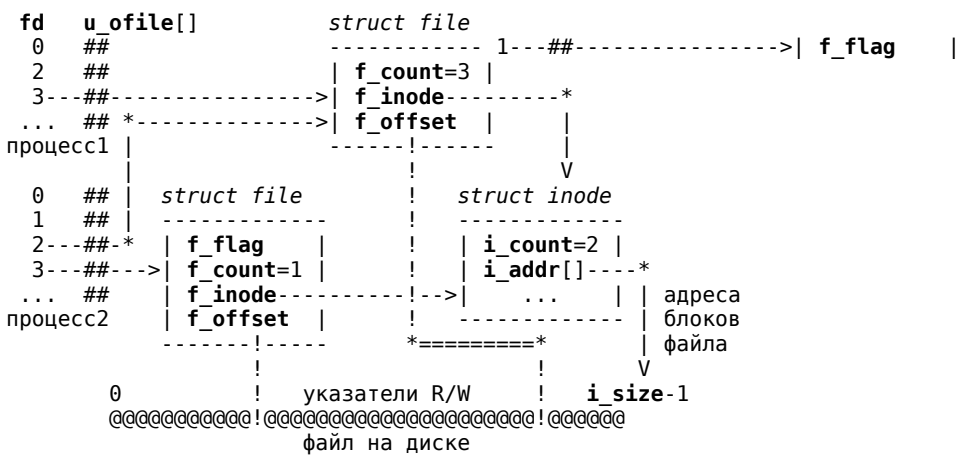
Содержимое некоторых полей этого паспорта можно узнать вызовом *stat*(). Все I-узлы собраны в единую область в начале файловой системы - так называемый *I-файл*. Все Iузлы пронумерованы, начиная с номера 1. Корневой каталог (файл с именем "/") как правило имеет I-узел номер 2.

** - У каждого процесса в *UNIX* также есть свой "паспорт". Часть этого паспорта находится в таблице процессов в ядре ОС, а часть - "приклеена" к самому процессу, однако не доступна из программы непосредственно. Эта вторая часть паспорта носит название "*u-area*" или структура *user*. В нее, в частности, входят таблица открытых процессом файлов, свободная ячейка, в нее заносится ссылка на структуру "открытый

файл" в ядре, и ИНДЕКС этой ячейки выдается в вашу программу в виде целого числа – так называемого "**дескриптора файла**".

При **закрытии** файла связная структура в ядре уничтожается, ячейка в таблице считается свободной, т.е. связь программы и файла разрывается.

Дескрипторы являются **локальными** для каждой программы. Т.е. если две программы открыли один и тот же файл – дескрипторы этого файла в каждой из них не обязательно совпадут (хотя и могут). Обратно: одинаковые дескрипторы (номера) в разных программах не обязательно обозначают один и тот же файл. Следует учесть и еще одну вещь: несколько или один процессов могут открыть один и тот же файл одновременно **несколько** раз. При этом будет создано несколько "связующих" структур (по одной для каждого открытия); каждая из них будет иметь СВОЙ указатель чтения/записи. Возможна и ситуация, когда несколько дескрипторов ссылаются к одной структуре – смотри ниже описание вызова *dup2*.




```
/* открыть файл */
int fd = open(char имя_файла[], int как_открыть);
        ...  /* какие-то операции с файлом */
close(fd);  /* закрыть */
```

Параметр **как_открыть:**

```
#include <fcntl.h>
O_RDONLY  - только для чтения.
O_WRONLY  - только для записи.
O_RDWR    - для чтения и записи.
O_APPEND  - иногда используется вместе с
открытием для записи, "добавление" в файл:
    O_WRONLY|O_APPEND, O_RDWR|O_APPEND
```

Если файл еще не существовал, то его нельзя открыть: *open* вернет значение (-1),

```
struct file *u_ofile[NOFILE];
```

ссылка на I-узел текущего каталога

```
struct inode *u_cdir;
```

а также ссылка на часть паспорта в таблице процессов

```
struct proc *u_procp;
```

сигнализирующее об ошибке. В этом случае файл надо создать:

```
int fd = creat(char имя_файла[], int коды_доступа);
```

Дескриптор **fd** будет открыт для записи в этот новый пустой файл. Если же файл уже существовал, *creat* опустошает его, т.е. уничтожает его прежнее содержимое и делает его длину равной 0L байт. **Коды_доступа** задают права пользователей на доступ к файлу. Это число задает битовую шкалу из 9и бит, соответствующих строке

```
биты:   876 543 210
        rwx rwx rwx
r - можно читать файл
w - можно записывать в файл
x - можно выполнять программу из этого файла
```

Первая группа – эта права владельца файла, вторая – членов его группы, третяя – всех прочих. Эти коды для владельца файла имеют еще и мнемонические имена (используемые в вызове *stat*):

```
#include <sys/stat.h>  /* Там определено: */
#define S_IREAD          0400
#define S_IWRITE         0200
#define S_IEXEC          0100
```

Подробности - в руководствах по системе *UNIX*. Отметим в частности, что *open*() может вернуть код ошибки **fd** < 0 не только в случае, когда файл не существует (**errno**==*ENOENT*), но и в случае, когда вам не разрешен соответствующий доступ к этому файлу (**errno**==*EACCES*; про переменную кода ошибки **errno** см. в главе "Взаимодействие с *UNIX*").

Вызов *creat* - это просто разновидность вызова *open* в форме

```
fd = open( имя_файла,
           O_WRONLY|O_TRUNC|O_CREAT, коды_доступа);
```

O_TRUNC
: означает, что если файл уже существует, то он должен быть опустошен при открытии. Коды доступа и владелец не изменяются.

O_CREAT
: означает, что файл должен быть создан, если его не было (без этого флага файл не создастся, а *open* вернет **fd** < 0). Этот флаг требует задания третьего аргумента

коды_доступа[***].
: Если файл уже существует - этот флаг не имеет никакого эффекта, но зато вступает в действие *O_TRUNC*.

Существует также флаг

O_EXCL
: который может использоваться совместно с *O_CREAT*. Он делает следующее: если файл уже существует, *open* вернет код ошибки (**errno**==*EEXIST*). Если файл не

*** - Заметим, что на самом деле коды доступа у нового файла будут равны

di_mode = (**коды_доступа** & ~**u_cmask**) | *IFREG*;
: (для каталога вместо *IFREG* будет *IFDIR*), где маска **u_cmask** задается системным вызовом

umask(**u_cmask**);
: (вызов выдает прежнее значение маски) и в дальнейшем наследуется всеми потомками данного процесса (она хранится в *u-area* процесса). Эта маска позволяет запретить доступ к определенным операциям для **всех** создаваемых нами файлов, несмотря на явно заданные коды доступа, например

umask(0077); /* ???------ */
: делает значащими только первые 3 бита кодов доступа (для владельца файла). Остальные биты будут равны нулю.

Все это относится и к созданию каталогов вызовом *mkdir*.

существовал - срабатывает *O_CREAT* и файл создается.
: Это позволяет предохранить уже существующие файлы от уничтожения.

Файл удаляется при помощи

```
int unlink(char имя_файла[]);
```

У каждой программы по умолчанию открыты три первых дескриптора, обычно связанные

```
0 - с клавиатурой (для чтения)
1 - с дисплеем    (выдача результатов)
2 - с дисплеем    (выдача сообщений об ошибках)
```

Если при вызове *close*(**fd**) дескриптор **fd** не соответствует открытому файлу (не был открыт) - ничего не происходит.

Часто используется такая метафора: если представлять себе файлы как книжки (только чтение) и блокноты (чтение и запись), стоящие на полке, то **открытие** файла это выбор блокнота по заглавию на его обложке и открытие обложки (на первой странице). Теперь можно читать записи, дописывать, вычеркивать и править записи в середине, листать книжку! Страницы можно сопоставить **блокам** файла (см. ниже), а "полку" с книжками - каталогу.

4.2.

Напишите программу, которая копирует содержимое одного файла в другой (новый) файл. При этом используйте системные вызовы чтения и записи *read* и *write*. Эти сисвызовы пересылают массивы байт из памяти в файл и наоборот. Но любую переменную можно рассматривать как массив байт, если забыть о структуре данных в переменной!

Читайте и записывайте файлы большими кусками, кратными 512 байтам. Это уменьшит число обращений к диску. Схема:

```
char buffer[512]; int n; int fd_inp, fd_outp;
        ...
while((n = read (fd_inp,  buffer, sizeof buffer)) > 0)
          write(fd_outp, buffer, n);
```

Приведем несколько примеров использования *write*:

```
char c = 'a';
int  i = 13, j = 15;
char s[20] = "foobar";
char p[]   = "FOOBAR";
struct { int x, y; } a = { 666, 999 };
/* создаем файл с доступом    rw-r--r-- */
int fd = creat("aFile", 0644);
write(fd, &c, 1);
write(fd, &i, sizeof i);  write(fd, &j, sizeof(int));
write(fd, s,  strlen(s)); write(fd, &a, sizeof a);
write(fd, p,  sizeof(p) - 1);
close(fd);
```

Обратите внимание на такие моменты:

- При использовании *write*() и *read*() надо передавать АДРЕС данного, которое мы хотим записать в файл (места, куда мы хотим прочитать данные из файла).
- Операции *read* и *write* возвращают число действительно прочитанных/записанных байт (при записи оно может быть меньше указанного нами, если на диске не хватает места; при чтении – если от позиции чтения до конца файла содержится меньше информации, чем мы затребовали).
- Операции *read*/*write* продвигают указатель чтения/записи

  ```
  RWptr += прочитанное_или_записанное_число_байт;
  ```

  При открытии файла указатель стоит на начале файла: **RWptr**=0. При записи файл если надо автоматически увеличивает свой размер. При чтении – если мы достигнем конца файла, то *read* будет возвращать "прочитано 0 байт" (т.е. при чтении указатель чтения не может стать больше размера файла).
- Аргумент **сколькоБайт** имеет тип *unsigned*, а не просто *int*:

  ```
  int n = read (int fd, char *адрес, unsigned сколькоБайт);
  int n = write(int fd, char *адрес, unsigned сколькоБайт);
  ```

  Приведем **упрощенные** схемы логики этих сисвызовов, когда они работают с обычным дисковым файлом (в *UNIX* **устройства** тоже выглядят для программ как файлы, но иногда с особыми свойствами):

4.2.1. m = write(fd, addr, n);

```
если( ФАЙЛ[fd] не открыт на запись) то вернуть (-1);
если(n == 0) то вернуть 0;
если( ФАЙЛ[fd] открыт на запись с флагом O_APPEND ) то
  RWptr = длина_файла; /* т.е. встать на конец файла */
если( RWptr > длина_файла ) то
  заполнить нулями байты файла в интервале
  ФАЙЛ[fd][ длина_файла..RWptr-1 ] = '\0';
скопировать байты из памяти процесса в файл
  ФАЙЛ[fd][ RWptr..RWptr+n-1 ] = addr[ 0..n-1 ];
  отводя на диске новые блоки, если надо
RWptr += n;
если( RWptr > длина_файла ) то
      длина_файла = RWptr;
вернуть n;
```

4.2.2. m = read(fd, addr, n);

```
если( ФАЙЛ[fd] не открыт на чтение) то вернуть (-1);
если( RWptr >= длина_файла ) то вернуть 0;
m = MIN( n, длина_файла - RWptr );
скопировать байты из файла в память процесса
  addr[ 0..m-1 ] = ФАЙЛ[fd][ RWptr..RWptr+m-1 ];
RWptr += m;
вернуть m;
```

4.3.

Найдите ошибки в фрагменте программы:

```
#define STDOUT 1  /* дескриптор стандартного вывода */
int i;
static char s[20] = "hi\n";
char c = '\n';
struct a{ int x,y; char ss[5]; } po;
scanf( "%d%d%d%s%s", i, po.x, po.y, s, po.ss);
write( STDOUT, s, strlen(s));
write( STDOUT, c, 1 );       /* записать 1 байт */
```

Ответ: в функции *scanf* перед аргументом **i** должна стоять операция "адрес", то есть *&***i**. Аналогично про *&***po.x** и *&***po.y**. Заметим, что **s** – это **массив**, т.е. **s** и так есть адрес, поэтому перед **s** операция *&* не нужна; аналогично про **po.ss** – здесь *&* не требуется.

В системном вызове *write* второй аргумент должен быть **адресом** данного, которое мы хотим записать в файл. Поэтому мы должны были написать *&***c** (во втором вызове *write*).

Ошибка в *scanf* – указание **значения** переменной вместо ее **адреса** – является довольно распространенной и **не может** быть обнаружена компилятором (даже при использовании прототипа функции *scanf*(char ***fmt**, ...), так как *scanf* – функция с переменным числом аргументов заранее **не определенных** типов). Приходится полагаться исключительно на собственную внимательность!

4.4.

Как по дескриптору файла узнать, открыт он на чтение, запись, чтение и запись одновременно? Вот два варианта решения:

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/param.h> /* там определено NOFILE */
#include <errno.h>
char *typeOfOpen(fd){
  int flags;
  if((flags=fcntl (fd, F_GETFL, NULL)) < 0 )
    return NULL;  /* fd вероятно не открыт */
  flags &= O_RDONLY | O_WRONLY | O_RDWR;
  switch(flags){
  case O_RDONLY:  return "r";
  case O_WRONLY:  return "w";
  case O_RDWR:    return "r+w";
  default:        return NULL;
  }
}
char *type2OfOpen(fd){
  extern errno; /* см. главу "системные вызовы" */
  int r=1, w=1;
  errno = 0; read(fd, NULL, 0);
  if( errno == EBADF ) r = 0;
  errno = 0; write(fd, NULL, 0);
  if( errno == EBADF ) w = 0;
  return (w && r) ? "r+w" :
          w       ? "w"   :
          r       ? "r"   :
                    "closed";
}
main(){
  int i; char *s, *p;
  for(i=0; i < NOFILE; i++ ){
     s = typeOfOpen(i); p = type2OfOpen(i);
    printf("%d:%s %s\n", i, s? s: "closed", p);
  }
}
```

Константа *NOFILE* означает максимальное число одновременно открытых файлов для одного процесса (это размер таблицы открытых процессом файлов, таблицы дескрипторов). Изучите описание системного вызова *fcntl* (file control).

4.5.

Напишите функцию **rename**() для переименования файла. Указание: используйте системные вызовы *link*() и *unlink*(). Ответ:

```
rename( from, to )
  char *from,     /* старое имя */
       *to;       /* новое имя  */
{
    unlink( to );   /* удалить файл to    */
    if( link( from, to ) < 0 ) /* связать */
        return (-1);
    unlink( from ); /* стереть старое имя */
    return 0;       /* OK */
}
```

Вызов

link(**существующее_имя, новое_имя**);

создает файлу альтернативное имя - в *UNIX* файл может иметь несколько имен: так каждый каталог имеет какое-то имя в родительском каталоге, а также имя "." в себе самом.

Каталог же, содержащий подкаталоги, имеет некоторое имя в своем родительском каталоге, имя "." в себе самом, и по одному имени ".." в каждом из своих подкаталогов.

Этот вызов будет неудачен, если файл **новое_имя** уже существует; а также если мы попытаемся создать альтернативное имя в **другой** файловой системе. Вызов

unlink(**имя_файла**)

удаляет имя файла. Если файл больше не имеет имен - он уничтожается. Здесь есть одна тонкость: рассмотрим фрагмент

```
int fd;
close(creat("/tmp/xyz", 0644)); /*Создать пустой файл*/
fd = open("/tmp/xyz", O_RDWR);
unlink("/tmp/xyz");
        ...
close(fd);
```

Первый оператор создает пустой файл. Затем мы открываем файл и уничтожаем его единственное имя. Но поскольку есть программа, открывшая этот файл, он не удаляется немедленно! Программа далее работает с **безымянным** файлом при помощи дескриптора **fd**. Как только файл закрывается - он будет уничтожен системой (как не имеющий имен). Такой трюк используется для создания временных рабочих файлов.

Файл можно удалить из каталога только в том случае, если данный **каталог** имеет для вас код доступа "запись". Коды доступа самого **файла** при удалении **не играют роли.**

В современных версиях *UNIX* есть системный вызов *rename*, который делает то же самое, что и написанная нами одноименная функция.

4.6.

Существование альтернативных имен у файла позволяет нам решить некоторые проблемы, которые могут возникнуть при использовании чужой программы, от которой нет исходного текста (которую нельзя поправить). Пусть программа выдает некоторую информацию в файл **zz.out** (и это имя жестко зафиксировано в ней, и не задается через аргументы программы):

```
/* Эта программа компилируется в a.out */
main(){
    int fd = creat("zz.out", 0644);
    write(fd, "It's me\n", 8);
}
```

Мы же хотим получить вывод на терминал, а не в файл. Очевидно, мы должны сделать файл **zz.out** синонимом устройства */dev/tty* (см. конец этой главы). Это можно сделать командой *ln*:

```
$ rm zz.out ; ln /dev/tty zz.out
$ a.out
$ rm zz.out
```

или программно:

```
/* Эта программа компилируется в start */
/* и вызывается  вместо a.out          */
#include <stdio.h>
main(){
   unlink("zz.out");
   link("/dev/tty", "zz.out");
     if( !fork()){ execl("a.out", NULL); }
     else wait(NULL);
```

```
        unlink("zz.out");
    }
```

(про *fork*, *exec*, *wait* смотри в главе про *UNIX*).

Еще один пример: программа *a.out* желает запустить программу **/usr/bin/vi** (смотри про функцию *system*() сноску через несколько страниц):

```
    main(){
        ... system("/usr/bin/vi xx.c"); ...
    }
```

На вашей же машине редактор *vi* помещен в **/usr/local/bin/vi**. Тогда вы просто создаете альтернативное имя этому редактору:

```
    $ ln /usr/local/bin/vi /usr/bin/vi
```

Помните, что альтернативное имя файлу можно создать лишь в **той же** файловой системе, где содержится исходное имя. В семействе *BSD*[****] это ограничение можно обойти, создав "символьную ссылку" вызовом

```
    symlink(link_to_filename,link_file_name_to_be_created);
```

Символьная ссылка - это файл, содержащий имя другого файла (или каталога). Система не производит автоматический подсчет числа таких ссылок, поэтому возможны "висячие" ссылки - указывающие на уже удаленный файл. Прочесть содержимое файла-ссылки можно системным вызовом

```
    char linkbuf[ MAXPATHLEN + 1]; /* куда поместить ответ */
    int len = readlink(pathname, linkbuf, sizeof linkbuf);
    linkbuf[len] = '\0';
```

Системный вызов *stat* автоматически разыменовывает символьные ссылки и выдает информацию про указуемый файл. Системный вызов *lstat* (аналог *stat* за исключением названия) выдает информацию про саму ссылку (тип файла *S_IFLNK*). Коды доступа к ссылке не имеют никакого значения для системы, существенны только коды доступа самого указуемого файла.

Еще раз: символьные ссылки удобны для указания файлов и каталогов на другом диске. Пусть у вас не помещается на диск каталог /opt/wawa. Вы можете разместить каталог wawa на диске USR: /usr/wawa. После чего создать символьную ссылку из /opt:

```
        ln -s /usr/wawa /opt/wawa
```

чтобы программы видели этот каталог под его прежним именем /opt/wawa.

Еще раз:

hard link
: - то, что создается системным вызовом *link*, имеет тот же I-node (индексный узел, паспорт), что и исходный файл. Это просто альтернативное имя файла, учитываемое в поле **di_nlink** в I-node.

symbolic link
: - создается вызовом *symlink*. Это отдельный самостоятельный файл, с собственным I-node. Правда, коды доступа к этому файлу не играют никакой роли; значимы только коды доступа указуемого файла.

4.7.

Напишите программу, которая находит в файле символ *@* и выдает файл с этого места дважды. Указание: для запоминания позиции в файле используйте вызов *lseek*() позиционирование указателя чтения/записи:

```
    long offset, lseek();
        ...
    /* Узнать текущую позицию чтения/записи:
     * сдвиг на 0 от текущей позиции. lseek вернет новую
     * позицию указателя (в байтах от начала файла). */
    offset = lseek(fd, 0L, 1);  /* ftell(fp) */
```

А для возврата в эту точку:

```
        lseek(fd, offset, 0);    /* fseek(fp, offset, 0) */
```

По поводу *lseek* надо помнить такие вещи:

- *lseek*(**fd, offset, whence**) устанавливает указатель чтения/записи на расстояние **offset** байт при **whence:**

```
0    от начала файла     RWptr  = offset;
1    от текущей позиции  RWptr += offset;
2    от конца файла      RWptr  = длина_файла + offset;
```

Эти значения **whence** можно обозначать именами:

```
#include <stdio.h>
  0    это   SEEK_SET
  1    это   SEEK_CUR
  2    это   SEEK_END
```

- Установка указателя чтения/записи - это **виртуальная** операция, т.е. реального подвода магнитных головок и вообще обращения к диску она не вызывает. Реальное движение головок к нужному месту диска произойдет только при операциях чтения/записи *read*()/*write*(). Поэтому *lseek*() - **дешевая** операция.
- *lseek*() возвращает новую позицию указателя чтения/записи **RWptr** относительно **начала файла** (long смещение в байтах). Помните, что если вы используете это значение, то вы должны предварительно описать *lseek* как функцию, возвращающую длинное целое: *long lseek*();
- Аргумент **offset** должен иметь тип *long* (не ошибитесь!).
- Если поставить указатель за конец файла (это допустимо!), то операция записи *write*() сначала заполнит байтом '\0' все пространство от конца файла до позиции указателя; операция *read*() при попытке чтения из-за конца файла вернет "прочитано 0 байт". Попытка поставить указатель перед началом файла вызовет ошибку.
- Вызов *lseek*() неприменим к pipe и FIFO-файлам, поэтому попытка сдвинуться на 0 байт выдаст ошибку:

```
/* это стандартная функция */
int isapipe(int fd){
     extern errno;
     return (lseek(fd, 0L, SEEK_CUR) < 0 && errno == ESPIPE);
}
```

выдает "истину", если **fd** - дескриптор "трубы"(pipe).

4.8.

Каков будет эффект следующей программы?

```
int fd = creat("aFile", 0644); /* creat создает файл
    открытый на запись, с доступом rw-r--r-- */
write(fd, "begin", 5 );
lseek(fd, 1024L * 1000, 0);
write(fd, "end", 3 );
close(fd);
```

Напомним, что при записи в файл, его длина **автоматически** увеличивается, когда мы записываем информацию за прежним концом файла. Это вызывает отведение места на диске для хранения новых данных (порциями, называемыми *блоками* - размером от 1/2 до 8 Кб в разных версиях). Таким образом, размер файла ограничен только наличием свободных блоков на диске.

В нашем примере получится файл длиной 1024003 байта. Будет ли он занимать на диске 1001 блок (по 1 Кб)?

В системе *UNIX* - нет! Вот кое-что про механику выделения блоков:

- Блоки располагаются на диске не обязательно подряд - у каждого файла есть специальным образом организованная таблица адресов его блоков.
- Последний блок файла может быть занят не целиком (если длина файла не кратна размеру блока), тем не менее число блоков у файла всегда **целое** (кроме семейства *BSD*, где блок может делиться на фрагменты, принадлежащие разным файлам). Операционная система в каждый момент времени знает длину файла с точностью до одного байта и не позволяет нам "заглядывать" в остаток блока, пока при своем "росте" файл не займет эти байты.
- Блок на диске физически выделяется лишь **после операции записи** в этот блок.

В нашем примере: при создании файла его размер 0, и ему выделено 0 блоков. При первой записи файлу будет выделен один блок (логический блок номер 0 для файла) и в его начало запишется "begin". Длина файла станет равна 5 (остаток блока - 1019 байт - не используется и файлу логически не принадлежит!). Затем *lseek* поставит указатель записи далеко за конец файла и *write* запишет в 1000-ый блок слово "end". 1000-ый блок будет выделен на диске. В этот момент у файла "возникнут" и все промежуточные блоки 1..999. Однако они будут только "числиться за файлом", но на диске отведены **не будут** (в таблице блоков файла это обозначается адресом 0)! При чтении из них будут читаться байты '\0'. Это так называемая "**дырка**" в файле. Файл имеет размер 1024003 байта, но на диске занимает всего 2 блока (на самом деле чуть больше, т.к. часть таблицы

блоков файла тоже находится в специальных блоках файла). Блок из "дырки" станет реальным, если в него что-нибудь записать.

Будьте готовы к тому, что "размер файла" (который, кстати, можно узнать системным вызовом *stat*) - это в *UNIX* не то же самое, что "место, занимаемое файлом на диске".

4.9.

Найдите ошибки:

```
FILE *fp;
    ...
fp = open( "файл", "r" ); /* открыть */
close(fp);                /* закрыть */
```

Ответ: используется системный вызов *open*() вместо функции *fopen*(); а также *close* вместо *fclose*, а их форматы (и результат) различаются! Следует четко различать две существующие в Си модели обмена с файлами: через системные вызовы: *open*, *creat*, *close*, *read*, *write*, *lseek*; и через библиотеку буферизованного обмена *stdio*: *fopen*, *fclose*, *fread*, *fwrite*, *fseek*, *getchar*, *putchar*, *printf*, и.т.д. В первой из них обращение к файлу происходит по целому **fd** - дескриптору файла, а во втором - по указателю *FILE* ***fp** - указателю на файл. Это параллельные механизмы (по своим возможностям), хотя второй является просто надстройкой над первым. Тем не менее, лучше их не смешивать.

* *I-узел* (I-node, индексный узел) - своеобразный "паспорт", который есть у каждого файла (в том числе и каталога). В нем содержатся:

** *BSD* - семейство *UNIX*-ов из University of California, Berkley. *Berkley Software Distribution*.