

НИЗКОУРОВНЕВЫЙ ВВОД-ВЫВОД

1. Обзор механизмов ввода-вывода	1
2. Файловые дескрипторы	1
3. Открытие файла: системный вызов <code>open()</code>	2
4. Закрытие файла: системный вызов <code>_close()</code>	4
5. Чтение файла: системный вызов <code>_read()</code>	6
6. Запись в файл: системный вызов <code>_write()</code>	9
7. Произвольный доступ: системный вызов <code>_lseek()</code>	10

1. Обзор механизмов ввода-вывода

В языке C для осуществления файлового ввода-вывода используются механизмы стандартной библиотеки языка, объявленные в заголовочном файле `stdio.h`. Как вы вскоре узнаете консольный ввод-вывод — это не более чем частный случай файлового ввода-вывода. В C++ для ввода-вывода чаще всего используются потоковые типы данных. Однако все эти механизмы являются всего лишь надстройками над низкоуровневыми механизмами ввода-вывода ядра операционной системы.

С точки зрения модели КИС (Клиент-Интерфейс-Сервер), сервером стандартных механизмов ввода вывода языка C (`printf`, `scanf`, `FILE*`, `fprintf`, `fputc` и т. д.) является библиотека языка. А сервером низкоуровневого ввода-вывода в ОС Linux является само ядро операционной системы.

Пользовательские программы взаимодействуют с ядром операционной системы посредством специальных механизмов, называемых *системными вызовами* (system calls, syscalls). Внешне системные вызовы реализованы в виде обычных функций языка C, однако каждый раз вызывая такую функцию, мы обращаемся непосредственно к ядру операционной системы. Список всех системных вызовов Linux можно найти в файле `/usr/include/asm/unistd.h`. Мы рассмотрим основные системные вызовы, осуществляющие ввод-вывод: `_open()`, `_close()`, `_read()`, `_write()`, `_lseek()` и некоторые другие.

2. Файловые дескрипторы

В языке C при осуществлении ввода-вывода мы используем указатель `FILE*`. Даже функция `printf()` в итоге сводится к вызову `vfprintf(stdout,...)`, разновидности функции `fprintf()`; константа `stdout` имеет тип `struct _IO_FILE*`, синонимом которого является тип `FILE*`. Это я к тому, что консольный ввод-вывод — это файловый ввод-вывод. Стандартный поток ввода, стандартный поток вывода и поток ошибок (как в C, так и в C++) — это файлы. В Linux все, куда можно что-то записать или откуда можно что-то прочитать представлено (или может быть представлено) в виде файла. Экран, клавиатура, аппаратные и виртуальные устройства, каналы, сокет — все это файлы. Это очень удобно, поскольку ко

всему можно применять одни и те же механизмы ввода-вывода. Владение механизмами низкоуровневого ввода-вывода дает **свободу перемещения данных** в Linux. Работа с локальными файловыми системами, межсетевое взаимодействие, работа с аппаратными устройствами, — все это осуществляется в Linux посредством низкоуровневого ввода-вывода.

Вы уже знаете, что при запуске программы в системе создается новый. У каждого процесса (кроме `init`) есть свой родительский процесс (`parent process` или просто `parent`), для которого новоиспеченный процесс является дочерним (`child process`, `child`). Каждый процесс получает копию окружения (`environment`) родительского процесса. Оказывается, кроме окружения дочерний процесс получает в качестве багажа еще и копию *таблицы файловых дескрипторов*.

Файловый дескриптор (`file descriptor`) — это целое число (`int`), соответствующее открытому файлу. Дескриптор, соответствующий реально открытому файлу всегда больше или равен нулю. Копия таблицы дескрипторов (читай: таблицы открытых файлов внутри процесса) скрыта в ядре. Мы не можем получить прямой доступ к этой таблице, как при работе с окружением через `environ`. Можно, конечно, кое-что "вытянуть" через дерево `/proc`, но нам это не надо. Программист должен лишь понимать, что каждый процесс имеет свою копию таблицы дескрипторов. В пределах одного процесса все дескрипторы уникальны (даже если они соответствуют одному и тому же файлу или устройству). В разных процессах дескрипторы могут совпадать или не совпадать — это не имеет никакого значения, поскольку у каждого процесса свой собственный набор открытых файлов.

Возникает вопрос: сколько файлов может открыть процесс? В каждой системе есть свой лимит, зависящий от конфигурации.

Например, в командной оболочке `bash`, открыты три файла: стандартный ввод (дескриптор 0), стандартный вывод (дескриптор 1) и стандартный поток ошибок (дескриптор 2). Когда под оболочкой запускается программа, в системе создается новый процесс, который является для этой оболочки дочерним процессом, следовательно, получает копию таблицы дескрипторов своего родителя (то есть все открытые файлы родительского процесса). Таким образом, программа может осуществлять консольный ввод-вывод через эти дескрипторы.

Таблица дескрипторов, помимо всего прочего, содержит информацию о *текущей позиции* чтения-записи для каждого дескриптора. При открытии файла позиция чтения-записи устанавливается в ноль. Каждый прочитанный или записанный байт увеличивает на единицу указатель текущей позиции.

3. Открытие файла: системный вызов `open()`

Чтобы получить возможность прочитать что-то из файла или записать что-то в файл, его нужно открыть. Это делает системный вызов `_open()`. Этот системный вызов не имеет постоянного списка аргументов (за счет использования механизма `va_arg`); в связи с этим существуют две "разновидности" `_open()`. Не только в C++ есть перегрузка функций ;-). Если интересно, то о механизме `va_arg` можно прочитать на `man`-странице `stdarg` (`man 3 stdarg`) или в книге Б. Кернигана и Д. Ритчи "Язык программирования Си". Ниже приведены адаптированные прототипы системного вызова `_open()`.

```
int _open (const char * filename, int flags, mode_t mode)
int _open (const char * filename, int flags)
```

Системный вызов `_open()` объявлен в заголовочном файле `fcntl.h`. Ниже приведен общий адаптированный прототип `_open()`.

```
int _open (const char * filename, int flags, ...)
```

Начнем по порядку. Первый аргумент - имя файла в файловой системе в обычной форме: полный путь к файлу (если файл не находится в текущем каталоге) или сокращенное имя (если файл в текущем каталоге).

Второй аргумент - это режим открытия файла, представляющий собой один или несколько флагов открытия, объединенных оператором побитового ИЛИ. Список доступных флагов приведен в Таблице 1.

Таблица 1 - Флаги режима открытия файла

Флаг	Описание
O_RDONLY	Только чтение (0)
O_WRONLY	Только запись (1)
O_RDWR	Чтение и запись (2)
O_CREAT	Создать файл, если не существует
O_TRUNC	Стереть файл, если существует
O_APPEND	Дописывать в конец
O_EXCL	Выдать ошибку, если файл существует при использовании O_CREAT
O_DSYNC	Принудительная синхронизация записи
O_RSYNC	Принудительная синхронизация перед чтением
O_SYNC	Принудительная полная синхронизация записи
O_NONBLOCK	Открыть файл в неблокируемом режиме, если это возможно
O_NDELAY	То же, что и O_NONBLOCK
O_NOCTTY	Если открываемый файл - терминальное устройство, не делать его управляющим терминалом процесса
O_NOFOLLOW	Выдать ошибку, если открываемый файл является символической ссылкой
O_DIRECTORY	Выдать ошибку, если открываемый файл не является каталогом
O_DIRECT	Попытаться минимизировать кэширование чтения/записи файла
O_ASYNC	Генерировать сигнал, когда появляется возможность чтения или записи в файл
O_LARGEFILE	Разрешить большие файлы (размер которых не может быть представлен в 31 бите (для систем с поддержкой LFS))

Наиболее часто используют только первые семь флагов. Если вы хотите, например, открыть файл в режиме чтения и записи, и при этом автоматически создать файл, если такового не существует, то второй аргумент `_open()` будет выглядеть примерно так: **O_RDWR|O_CREAT**. Константы-флаги открытия объявлены в заголовочном файле

bits/fcntl.h, однако не стоит включать этот файл в свои программы, поскольку он уже включен в файл fcntl.h.

Третий аргумент используется в том случае, если `_open()` создаёт новый файл. В этом случае файлу нужно задать права доступа (режим), с которыми он появится в файловой системе. Права доступа задаются перечислением флагов, объединенных побитовым ИЛИ. Вместо флагов можно использовать число (как правило восьмеричное), однако первый способ нагляднее и предпочтительнее. Список флагов приведен в Таблице 2.

Таблица 2 - Флаги общего режима

Флаг	Восьмеричное представление	RWX-представление
S_IRWXU	00700	rwX --- ---
S_IRUSR	00400	r-- --- ---
S_IREAD	00400	r-- --- ---
S_IWUSR	00200	-w- --- ---
S_IWRITE	00200	-w- --- ---
S_IXUSR	00100	--x --- ---
S_IEXEC	00100	--x --- ---
S_IRWXG	00070	--- rwx ---
S_IRGRP	00040	--- r-- ---
S_IWGRP	00020	--- -w- ---
S_IXGRP	00010	--- --x ---
S_IRWXO	00007	--- --- rwx
S_IROTH	00004	--- --- r--
S_IWOTH	00002	--- --- -w-
S_IXOTH	00001	--- --- --x

Чтобы, например, созданный файл был доступен в режиме "чтение-запись" пользователем и группой и "только чтение" остальными пользователями, - в третьем аргументе `_open()` надо указать примерно следующее: **S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH** или **0664**. Флаги режима доступа реально объявлены в заголовочном файле `bits/stat.h`, но он не предназначен для включения в пользовательские программы, и вместо него мы должны включать файл `sys/stat.h`. Тип `mode_t` объявлен в заголовочном файле `sys/types.h`.

Если файл был успешно открыт, `_open()` возвращает файловый дескриптор, по которому мы будем обращаться к файлу. Если произошла ошибка, то `_open()` возвращает -1.

4. Заккрытие файла: системный вызов `_close()`

Системный вызов `_close()` закрывает файл. Вообще говоря, по завершении процесса все открытые файлы (кроме файлов с дескрипторами 0, 1 и 2) автоматически закрываются. Тем не менее, это не освобождает нас от самостоятельного вызова `close()`, когда файл нужно закрыть. К тому же, если файлы не закрывать самостоятельно, то соответствующие

дескрипторы не освобождаются, что может привести к превышению лимита открытых файлов. Простой пример: приложение может быть настроено так, чтобы каждую минуту открывать и перечитывать свой файл конфигурации для проверки обновлений. Если каждый раз файл не будет закрываться, то в конкретной системе, например, приложение может "накрыться медным тазом" примерно через 17 часов. Автоматически!

Кроме того, файловая система Linux поддерживает механизм буферизации. Это означает, что данные, которые якобы записываются, реально записываются на носитель (синхронизируются) только через какое-то время, когда система сочтет это правильным и оптимальным. Это повышает производительность системы и даже продлевает ресурс жестких дисков. Системный вызов `_close()` не форсирует запись данных на диск, однако дает больше гарантий того, что данные останутся в целости и сохранности.

Системный вызов `_close()` объявлен в файле `unistd.h`. Ниже приведен его адаптированный прототип.

```
int _close (int fd)
```

Очевидно, что единственный аргумент — это файловый дескриптор. Возвращаемое значение - ноль в случае успеха, и -1 - в случае ошибки. Довольно часто `_close()` вызывают без проверки возвращаемого значения. Это не очень грубая ошибка, но, тем не менее, иногда закрытие файла бывает неудачным (в случае неправильного дескриптора, в случае прерывания функции по сигналу или в случае ошибки ввода-вывода, например). В любом случае, если программа сообщит пользователю, что файл невозможно закрыть, это хорошо.

Теперь можно написать простую программу, использующую системные вызовы `_open()` и `_close()`. Мы еще не умеем читать из файлов и писать в файлы, поэтому напишем программу, которая создает файл с именем, переданным в качестве аргумента (`argv[1]`) и с правами доступа 0600 (чтение и запись для пользователя). Ниже приведен исходный код программы.

```
#include <fcntl.h>          /* open() and O_XXX flags */
#include <sys/stat.h>        /* S_IXXX flags */
#include <sys/types.h>       /* mode_t */
#include <unistd.h>          /* close() */
#include <stdlib.h>
#include <stdio.h>

int main (int argc, char ** argv)
{
    int fd;
    mode_t mode = S_IRUSR | S_IWUSR;
    int flags = O_WRONLY | O_CREAT | O_EXCL;
    if (argc < 2)
    {
        fprintf (stderr, "openclose: Too few arguments\n");
        fprintf (stderr, "Usage: openclose <filename>\n");
        exit (1);
    }

    fd = _open (argv[1], flags, mode);
    if (fd < 0)
    {
        fprintf (stderr, "openclose: Cannot open file '%s'\n",
                 argv[1]);
        exit (1);
    }
}
```

```

    }

    if (_close (fd) != 0)
    {
        fprintf (stderr, "Cannot close file (descriptor=%d)\n", fd);
        exit (1);
    }
    exit (0);
}

```

Обратите внимание, если запустить программу дважды с одним и тем же аргументом, то на второй раз `_open()` выдаст ошибку. В этом виноват флаг `O_EXCL` (см. Таблицу 4 Приложения 2), который "дает добро" только на создание еще не существующих файлов. Наглядности ради, флаги открытия и флаги режима мы занесли в отдельные переменные, однако можно было бы сделать так:

```

fd = _open (argv[1], O_WRONLY | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR);
Или так:
fd = _open (argv[1], O_WRONLY | O_CREAT | O_EXCL, 0600);

```

5. Чтение файла: системный вызов `_read()`

Системный вызов `_read()`, объявленный в файле `unistd.h`, позволяет читать данные из файла. В отличие от библиотечных функций файлового ввода-вывода, которые предоставляют возможность интерпретации считываемых данных. Можно, например, записать в файл следующее содержимое:

```
2021
```

Теперь, используя библиотечные механизмы, можно читать файл по-разному:

```

fscanf (filep, "%s", buffer);
fscanf (filep, "%d", number);

```

Системный вызов `_read()` читает данные в "сыром" виде, то есть как последовательность байт, без какой-либо интерпретации. Ниже представлен адаптированный прототип `_read()`.

```
ssize_t _read (int fd, void * buffer, size_t count);
```

Первый аргумент — это файловый дескриптор. Здесь больше сказать нечего. Второй аргумент — это указатель на область памяти, куда будут помещаться данные. Третий аргумент - количество байт, которые функция `_read()` будет **пытаться** прочитать из файла. Возвращаемое значение - количество прочитанных байт, если чтение состоялось и -1, если произошла ошибка. Хочу заметить, что если `_read()` возвращает значение меньше `count`, то это не символизирует об ошибке.

Хочу сказать несколько слов о типах. Тип `size_t` в Linux используется для хранения размеров блоков памяти. Какой тип реально скрывается за `size_t`, зависит от архитектуры; как правило это `unsigned long int` или `unsigned int`. Тип `ssize_t` (Signed SIZE Type) - это тот же `size_t`, только знаковый. Используется, например, в тех случаях, когда нужно сообщить об ошибке, вернув отрицательный размер блока памяти. Системный вызов `_read()` именно так и поступает.

Теперь напишем программу, которая просто читает файл и выводит его содержимое на экран. Имя файла будет передаваться в качестве аргумента (`argv[1]`). Ниже приведен исходный код этой программы.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>

int main (int argc, char ** argv)
{
    int fd;
    ssize_t ret;
    char ch;
    if (argc < 2)
    {
        fprintf (stderr, "Too few arguments\n");
        exit (1);
    }

    fd = _open (argv[1], O_RDONLY);
    if (fd < 0)
    {
        fprintf (stderr, "Cannot open file\n");
        exit (1);
    }

    while ((ret = _read (fd, &ch, 1)) > 0)
    {
        putchar (ch);
    }

    if (ret < 0)
    {
        fprintf (stderr, "myread: Cannot read file\n");
        exit (1);
    }
    close (fd);
    exit (0);
}
```

В этом примере используется укороченная версия `_open()`, так как файл открывается только для чтения. В качестве буфера (второй аргумент `_read()`) мы передаём адрес переменной типа `char`. По этому адресу будут считываться данные из файла (по одному байту за раз) и передаваться на стандартный вывод. Цикл чтения файла заканчивается, когда `_read()` возвращает нуль (нечего больше читать) или `-1` (ошибка). Системный вызов `_close()` закрывает файл.

Как можно заметить, в нашем примере системный вызов `read()` вызывается ровно столько раз, сколько байт содержится в файле. Иногда это действительно нужно; но не здесь. Чтение-запись посимвольным методом (как в нашем примере) значительно замедляет процесс ввода-вывода за счет многократных обращений к системным вызовам. По этой же причине возрастает вероятность возникновения ошибки. Если нет действительной необходимости, файлы нужно читать блоками. Ниже приведен исходный код программы,

которая делает то же самое, что и предыдущий пример, но с использованием блочного чтения файла. Размер блока установлен в 64 байта.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>

#define BUFFER_SIZE 64

int main (int argc, char ** argv)
{
    int fd;
    ssize_t read_bytes;
    char buffer[BUFFER_SIZE+1];
    if (argc < 2)
    {
        fprintf(stderr, "Too few arguments\n");
        exit(1);
    }

    fd = _open(argv[1], O_RDONLY);
    if (fd < 0)
    {
        fprintf(stderr, "Cannot open file\n");
        exit(2);
    }

    while ((read_bytes = _read(fd, buffer, BUFFER_SIZE)) > 0)
    {
        buffer[read_bytes] = 0; // Null-terminator for C-string
        fputs(buffer, stdout);
    }

    if (read_bytes < 0)
    {
        fprintf(stderr, "myread: Cannot read file\n");
        exit(3);
    }
    close(fd);
    exit(0);
}
```

Теперь можно примерно оценить и сравнить скорость работы двух примеров. Для этого надо выбрать в системе достаточно большой файл (видеофильм например) и посмотреть на то, как быстро читаются эти файлы:

```
$ time ./myread /boot/vmlinuz > /dev/null
real    0m1.443s
user    0m0.383s
sys     0m1.039s
```

```
$ time ./myread1 /boot/vmlinuz > /dev/null
real    0m0.055s
user    0m0.010s
sys     0m0.023s
```


6. Запись в файл: системный вызов `_write()`

Для записи данных в файл используется системный вызов `_write()`. Ниже представлен его прототип.

```
ssize_t _write (int fd, const void * buffer, size_t count)
```

Как видите, прототип `_write()` отличается от `_read()` только спецификатором `const` во втором аргументе. В принципе `write()` выполняет процедуру, обратную `_read()`: записывает `count` байтов из буфера `buffer` в файл с дескриптором `fd`, возвращая количество записанных байтов или `-1` в случае ошибки. За основу возьмем программу из предыдущего раздела.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>          /* _read(), _write(), _close() */
#include <fcntl.h>          /* _open(), O_RDONLY */
#include <sys/stat.h>       /* S_IRUSR */
#include <sys/types.h>      /* mode_t */

#define BUFFER_SIZE 64

int main(int argc, char ** argv)
{
    int fd;
    ssize_t read_bytes, written_bytes;
    char buffer[BUFFER_SIZE];
    if(argc < 2)
    {
        fprintf(stderr, "Too few arguments\n");
        exit(1);
    }

    fd = _open(argv[1], O_RDONLY);
    if(fd < 0)
    {
        fprintf(stderr, "Cannot open file\n");
        exit(2);
    }

    while((read_bytes = _read(fd, buffer, BUFFER_SIZE)) > 0)
    {
        /* 1 == stdout */
        written_bytes = _write(1, buffer, read_bytes);
        if(written_bytes != read_bytes)
        {
            fprintf(stderr, "Cannot write\n");
            exit(3);
        }
    }

    if(read_bytes < 0)
    {
        fprintf(stderr, "myread: Cannot read file\n");
        exit(4);
    }
    close(fd);
    exit(0);
}
```

В этом примере нам уже не надо изощряться в попытках вставить нуль-терминатор в строку для записи, поскольку системный вызов `_write()` не запишет большее количество байт, чем мы ему указали. В данном случае для демонстрации `_write()` мы просто записывали данные в файл с дескриптором 1, то есть в стандартный вывод. Но прежде, чем переходить к чтению следующего раздела, попробуйте самостоятельно записать что-нибудь (при помощи `_write()`, естественно) в обычный файл. Когда будете открывать файл для записи, обратите пожалуйста внимание на флаги `O_TRUNC`, `O_CREAT` и `O_APPEND`. Подумайте, все ли флаги сочетаются между собой по смыслу.

7. Произвольный доступ: системный вызов `_lseek()`

Как уже говорилось, с каждым открытым файлом связано число, указывающее на текущую позицию чтения-записи. При открытии файла позиция равна нулю. Каждый вызов `_read()` или `_write()` увеличивает текущую позицию на значение, равное числу прочитанных или записанных байт. Благодаря этому механизму, каждый повторный вызов `_read()` читает **следующие** данные, и каждый повторный `_write()` записывает данные **в продолжение** предыдущих, а не затирает старые. Такой механизм последовательного доступа очень удобен, однако иногда требуется получить произвольный доступ к содержимому файла, чтобы, например, прочитать или записать файл заново.

Для изменения текущей позиции чтения-записи используется системный вызов `_lseek()`. Ниже представлен его прототип.

```
off_t _lseek (int fd, off_t offset, int against)
```

Первый аргумент, как всегда, - файловый дескриптор. Второй аргумент - смещение, как положительное (вперед), так и отрицательное (назад). Третий аргумент обычно передается в виде одной из трех констант `SEEK_SET`, `SEEK_CUR` и `SEEK_END`, которые показывают, от какого места отсчитывается смещение. `SEEK_SET` - означает начало файла, `SEEK_CUR` - текущая позиция, `SEEK_END` - конец файла. Рассмотрим следующие вызовы:

```
_lseek (fd, 0, SEEK_SET);  
_lseek (fd, 20, SEEK_CUR);  
_lseek (fd, -10, SEEK_END);
```

Первый вызов устанавливает текущую позицию в начало файла. Второй вызов смещает позицию вперед на 20 байт. В третьем случае текущая позиция перемещается на 10 байт назад относительно конца файла.

В случае удачного завершения, `_lseek()` возвращает значение установленной "новой" позиции относительно начала файла. В случае ошибки возвращается -1.

Наиболее подходящим примером работы `_lseek()` оказалась идея создания программы рисования символами. Программа оказалась не слишком простой, однако если вы сможете разобраться в ней, то можете считать, что успешно овладели азами низкоуровневого ввода-вывода Linux. Ниже представлен исходный код этой программы.

```
#include <stdlib.h>  
#include <stdio.h>  
#include <fcntl.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <string.h> /* memset() */  
#include "iostream"  
#include <conio.h>  
#include <io.h>  
  
#define N_ROWS    15                                /* Image height */
```

```

#define N_COLS    40                /* Image width */
#define FG_CHAR   'O'              /* Foreground character */
#define IMG_FN    "a:\\image.txt"  /* Image filename */
#define N_MIN(A,B) ((A)<(B)?(A):(B))
#define N_MAX(A,B) ((A)>(B)?(A):(B))

static char buffer[N_COLS];

void init_draw(int fd)
{
    size_t bytes_written = 0;
    memset(buffer, ' ', N_COLS);
    buffer[N_COLS] = '\n';
    while (bytes_written < (N_ROWS * (N_COLS + 1)))
        bytes_written += _write(fd, buffer, N_COLS + 1);
}

void draw_point(int fd, int x, int y)
{
    char ch = FG_CHAR;
    _lseek(fd, y * (N_COLS + 1) + x, SEEK_SET);
    _write(fd, &ch, 1);
}

void draw_hline(int fd, int y, int x1, int x2)
{
    size_t bytes_write = abs(x2 - x1) + 1;
    memset(buffer, FG_CHAR, bytes_write);
    _lseek(fd, y * (N_COLS + 1) + N_MIN(x1, x2), SEEK_SET);
    _write(fd, buffer, bytes_write);
}

void draw_vline(int fd, int x, int y1, int y2)
{
    int i = N_MIN(y1, y2);
    while (i <= N_MAX(y2, y1)) draw_point(fd, x, i++);
}

int main(void)
{
    int a, b, c, i = 0;
    char ch;
    int fd = _open(IMG_FN, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) {
        fprintf(stderr, "Cannot open file\n");
        exit(1);
    }
    init_draw(fd);
    const char * icode[] = { "v 1 1 11", "v 11 7 11", "v 14 5 11",
        "v 18 6 11", "v 21 5 10", "v 25 5 10", "v 29 5 6", "v 33 5 6",
        "v 29 10 11", "v 33 10 11", "h 11 1 8", "h 5 16 17",
        "h 11 22 24", "p 11 5 0", "p 15 6 0", "p 26 11 0", "p 30 7 0",
        "p 32 7 0", "p 31 8 0", "p 30 9 0", "p 32 9 0", NULL };
    while (icode[i] != NULL) {
        scanf_s(icode[i], "%c %d %d %d", &ch, &a, &b, &c);
        switch (ch) {
            case 'v': draw_vline(fd, a, b, c); break;
            case 'h': draw_hline(fd, a, b, c); break;
            case 'p': draw_point(fd, a, b); break;
        }
        i++;
    }
}

```

```

        default: abort();
    }
    i++;
}
close(fd);
exit(0);
}

```

Теперь разберемся, как работает эта программа. Изначально "полотно" заполняется пробелами. Функция `init_draw()` построчно записывает в файл пробелы, чтобы получился "холст", размером `N_ROWS` на `N_COLS`. Массив строк `icode` в функции `main()` — это набор команд рисования. Команда начинается с одной из трех литер: 'v' - нарисовать вертикальную линию, 'h' - нарисовать горизонтальную линию, 'r' - нарисовать точку. После каждой такой литеры следуют три числа. В случае вертикальной линии первое число - фиксированная координата X, а два других числа — это начальная и конечная координаты Y. В случае горизонтальной линии фиксируется координата Y (первое число). Два остальных числа - начальная координата X и конечная координата X. При рисовании точки используются только два первых числа: координата X и координата Y. Итак, функция `draw_vline()` рисует вертикальную линию, функция `draw_hline()` рисует горизонтальную линию, а `draw_point()` рисует точку.

Функция `init_draw()` пишет в файл `N_ROWS` строк, каждая из которых содержит `N_COLS` пробелов, заканчивающихся переводом строки. Это процедура подготовки "холста".

Функция `draw_point()` вычисляет позицию (исходя из значений координат), перемещает туда текущую позицию ввода-вывода файла, и записывает в эту позицию символ (`FG_CHAR`), которым мы рисуем "картину".

Функция `draw_hline()` заполняет часть строки символами `FG_CHAR`. Так получается горизонтальная линия. Функция `draw_vline()` работает иначе. Чтобы записать вертикальную линию, нужно записывать по одному символу и каждый раз "перескакивать" на следующую строку. Эта функция работает медленнее, чем `draw_hline()`, но иначе мы не можем.

Полученное изображение записывается в файл `image`. Будьте внимательны: чтобы разгрузить исходный код, из программы исключены многие проверки (`_read()`, `_write()`, `_close()`, диапазон координат и проч.). Попробуйте включить эти проверки самостоятельно.