ОПЕРАЦИОННЫЕ СИСТЕМЫ И СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лекция 13 — Каналы и FIFO

Преподаватель: Поденок Леонид Петрович, 505a-5

+375 17 293 8039 (505a-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2024

2024.04.25

Оглавление

Каналы и FIFO – обзор

Канал и FIFO (именованный канал)— это механизмы межпроцессного взаимодействия: данные, записанные в канал одним процессом, могут быть прочитаны другим процессом.

Данные обрабатываются в порядке «первым поступил — первым обслужен» (FIFO).

Канал имеет конец для чтения и конец для записи. Канал должен быть открыт с обоих концов.

При чтении из канала или файла FIFO, в который никакие процессы не пишут (возможно, потому что все они закрыли файл или завершили работу), возвращается конец файла.

Запись в канал или FIFO без процесса чтения рассматривается как ошибка – она генерирует сигнал «SIGPIPE» и завершается ошибкой с кодом «EPIPE», если сигнал обрабатывается или блокируется.

Каналы и специальные файлы FIFO не поддерживают позиционирование файла – операции чтения и записи выполняются последовательно – чтение с начала файла и запись в конец.

Каналы

Каналы и FIFO (также известные как именованные каналы) обеспечивают *однонаправленный канал связи между процессами*.

, при этом оба конца канала должны быть унаследованы от одного процесса, создавшего канал.

Обычно процесс создает канал непосредственно перед тем, как создать один или несколько дочерних процессов. Затем канал используется для связи между родительским и дочерними процессами, или для связи между двумя одноуровневыми процессами.

Канал создается с помощью **`pipe(2)`**, которая создает новый канал и возвращает два файловых дескриптора, один из которых относится к концу канала для чтения, а другой — к концу для записи.

FIFO (сокращение от First In First Out) имеет имя в файловой системе, которое создается с помощью **mkfifo(3)**, и открывается с помощью **open(2)**.

Любой процесс может открыть FIFO, если права доступа к файлу позволяют это сделать. Чтобы FIFO работал, он должен быть открыт как на запись, так и на чтение.

Конец для чтения открывается с помощью флага **O_RDONLY.**

Конец для записи открывается с помощью флага **O_WRONLY**.

Примечание

Хотя FIFO имеют имя пути в файловой системе, ввод-вывод в FIFO не включает операции на базовом устройстве (если оно есть) – это специальный файл.

Процесс может открыть FIFO в неблокирующем режиме. В этом случае открытие **O_RDONLY** выполняется без ошибки, даже если другая сторона ещё не открыла его для записи. Однако, открытие **O_WRONLY** завершается ошибкой **ENXIO** (нет такого устройства или адреса), если другая сторона FIFO к этому моменту не была открыта на чтение. Существует опасность взаимоблокировки при использовании процессом FIFO для чтения и записи (для связи с самим собой).

Ввод/вывод через каналы и FIFO

Единственная разница между каналами и FIFO заключается в том, как они создаются и открываются. Как только эти задачи выполнены, ввод-вывод по каналам и FIFO имеет одинаковую семантику.

Если процесс попытается прочитать из пустого канала, **read(2)** заблокируется до тех пор, пока в канале не станут доступны данные.

Если процесс пытается записать в заполненный канал (см. ниже), то блокируется **write(2)** до тех пор, пока из канала не будет прочитано достаточно данных, чтобы запись можно было завершить.

При использовании операции **fcntl(2) F_SETFL** для включения флага состояния открытия файла **O_NONBLOCK** возможен неблокирующий ввод-вывод.

Канал связи, предоставляемый каналом, представляет собой поток байтов – понятия границ сообщения для него нет.

Если все файловые дескрипторы, относящиеся к концу канала для записи, были закрыты, то попытка **read(2)** из канала увидит конец файла (**read(2)** вернет **0**).

Если все файловые дескрипторы, относящиеся к концу канала для чтения, закрыты, то **write(2)** вызовет генерацию для вызывающего процесса сигнала **SIGPIPE**.

Если вызывающий процесс игнорирует этот сигнал, то **write(2)** завершается ошибкой **EPIPE**.

Приложение, использующее **pipe(2)** и **fork(2)**, должно использовать подходящие вызовы **close(2)** для закрытия ненужных файловых дескрипторов – это гарантирует, что при необходимости **SIGPIPE/EPIPE** и конец файла будут доставлены.

К каналу невозможно применить вызов **lseek(2)**.

Емкость канала

Канал имеет ограниченную емкость.

Если канал заполнен, то **write(2)** заблокируется или завершится ошибкой, в зависимости от того, установлен ли флаг **O_NONBLOCK**.

Различные реализации имеют разные ограничения на емкость канала. Приложения не должны полагаться на конкретную емкость – приложение должно быть спроектировано таким образом, чтобы процесс чтения потреблял данные, как только они становятся доступными, чтобы процесс записи не оставался заблокированным.

В версиях Linux до 2.6.11 емкость канала была равна размеру системной страницы (например, 4096 байт на i386).

Начиная с Linux 2.6.11, емкость канала составляет 16 страниц (т. е. 65 536 байт в системе с размером страницы 4096 байт).

Начиная с Linux 2.6.35 емкость канала по умолчанию составляет 16 страниц, но емкость можно запросить и установить с помощью операций **fcntl(2) F_GETPIPE_SZ** и **F_SETPIPE_SZ**.

Есть операция системного вызова **ioctl(2)**, которую можно применить к файловому дескриптору, ссылающемуся на любой конец канала. Она помещает количество непрочитанных байтов в канале в буфер **int**, на который указывает последний аргумент вызова:

```
ioctl(fd, FIONREAD, &nbytes);
```

Операция **FIONREAD** не указана ни в одном стандарте, но существует во многих реализациях.

Файлы в /proc

В Linux для управления количеством памяти, которое может быть использовано для каналов, существуют следующие файлы:

/proc/sys/fs/pipe-max-pages (только в Linux 2.6.34)

Это верхний предел емкости канала в страницах, которую непривилегированный пользователь (без возможности **CAP_SYS_RESOURCE**) может для канала установить.

Этот интерфейс был удален в Linux 2.6.35 в пользу **/proc/sys/fs/pipe-max-size**.

/proc/sys/fs/pipe-max-size (начиная с Linux 2.6.35)

Максимальный размер (в байтах) отдельных каналов, который может быть установлен пользователями без возможности CAP_SYS_RESOURCE.

Значение, назначенное этому файлу, может быть округлено в большую сторону, чтобы отразить фактически используемое значение. Чтобы определить округленное значение, следует прочитать содержимое этого файла после присвоения ему значения.

Значение по умолчанию для этого файла равно 1048576 (1 МиБ).

```
cat /proc/sys/fs/pipe-max-size
1048576
```

Минимальное значение, которое можно присвоить этому файлу, — размер системной страницы.

Попытки установить ограничение меньше размера страницы приводят к сбою **write(2)** с ошибкой **EINVAL**.

Начиная с Linux 4.9, значение в этом файле также действует как потолок емкости по умолчанию для нового канала или вновь открытого FIFO.

/proc/sys/fs/pipe-user-pages-hard (начиная с Linux 4.5)

Жесткое ограничение на общий размер (в страницах) всех каналов, созданных или установленных одним непривилегированным пользователем (т. е. пользователем, не имеющим ни полномочий **CAP_SYS_RESOURCE**, ни возможностей **CAP_SYS_ADMIN**). С того момента, как общее количество страниц, выделенных буферам канала для этого пользователя, достигнет этого предела, попытки создания новых каналов будут отклонены, также будут отклонены попытки увеличить емкость канала. Когда значение этого ограничения равно нулю (по умолчанию), жесткое ограничение не применяется.

```
$ cat /proc/sys/fs/pipe-user-pages-hard
0
```

/proc/sys/fs/pipe-user-pages-soft (начиная с Linux 4.5)

Мягкое ограничение на общий размер (в страницах) всех каналов, созданных или установленных одним непривилегированным пользователем (т. е. пользователем, не имеющим ни полномочий **CAP_SYS_RESOURCE**, ни возможностей **CAP_SYS_ADMIN**).

С того момента, как общее количество страниц, выделенных буферам канала для этого пользователя, достигнет этого предела, отдельные каналы, созданные пользователем, будут ограничены одной страницей, а попытки увеличить емкость канала будут отклонены. Когда значение этого ограничения равно нулю, мягкое ограничение не применяется. Значение по умолчанию для этого файла — 16384, что позволяет создать до 1024 каналов с емкостью по умолчанию.

```
$ cat /proc/sys/fs/pipe-user-pages-soft
16384
```

PIPE_BUF

POSIX.1 говорит, что все **write(2)**, которые пишут меньше байтов, чем **PIPE_BUF**, должны быть атомарными. В этом случае выходные данные записываются в канал как непрерывная последовательность.

Запись большего количества байтов, чем PIPE_BUF, может быть неатомарной – ядро может чередовать данные с данными, записанными другими процессами

POSIX.1 требует, чтобы **PIPE_BUF** был не менее 512 байт (**POSIX_PIPE_BUF**).

В Linux **PIPE_BUF** равен 4096 байтам (`limits.h`).

Точная семантика зависит от того, является ли файловый дескриптор неблокирующим (**O_NONBLOCK**), есть ли в канале несколько записывающих устройств и от **n** — количества записываемых байтов:

1) O_NONBLOCK disabled, n <= PIPE_BUF

Все **n** байтов записываются атомарно.

write(2) может блокироваться, если нет места для немедленной записи **n** байтов.

2) O_NONBLOCK enabled, n <= PIPE_BUF

Если есть место для записи **n** байтов в канал, то **write(2)** сразу записывает все **n** байтов и завершается успешно.

Если места нет, **write(2)** завершится ошибкой, а для **errno** будет установлено значение **EAGAIN**.

3) O_NONBLOCK disabled, n > PIPE_BUF

Запись неатомарна – данные, переданные в **write(2)**, могут чередоваться с данными, переданными во **write(2)** в другом процессе.

При этом **write(2)** блокируется до тех пор, пока не будет записано **n** байтов.

4) O_NONBLOCK enabled, n > PIPE_BUF

Если в канале нет места, то **`write(2)`** завершается ошибкой, а **`errno`** устанавливается в **`EAGAIN`**.

Если место есть, может быть записано от 1 до **`n`** байтов. Т. е. может произойти «частичная запись», поэтому вызывающая сторона должна проверять возвращаемое из **`write(2)`** значение, чтобы увидеть, сколько байтов было фактически записано. Причем эти байты могут чередоваться с **`write(2)`** из других процессов.

Флаги статуса открытия файла

Единственными флагами состояния открытого файла, которые можно осмысленно применить к каналу или FIFO, являются **`O_NONBLOCK`** и **`O_ASYNC`**.

`O_ASYNC` для дескриптора конца канала, открытого для чтения вызывает генерацию сигнала (по умолчанию **`SIGIO`**), как только в канале становится доступным новый ввод.

Цель доставки сигналов должна быть установлена с помощью команды **`fcntl(2) F_SETOWN`**.

В Linux **`O_ASYNC`** поддерживается для каналов и FIFO только начиная с ядра 2.6.

Примечания о переносимости

В некоторых системах (но не в Linux) каналы являются *двунаправленными*: данные могут передаваться в обоих направлениях между концами канала.

POSIX.1 требует только однонаправленных каналов.

Переносимые приложения не должны полагаться на семантику двунаправленных конвейеров.

pipe(), pipe2() — создать канал

Функция **pipe(2)** создает однонаправленный канал.

```
#include <unistd.h>

// On Alpha, IA-64, MIPS, SuperH, and SPARC/SPARC64
struct fd_pair {
    long fd[2];
};
struct fd_pair pipe();

// На других архитектурах
int pipe(int pipefd[2]); // pipefd[0] -- чтение, pipefd[1] -- запись

#define _GNU_SOURCE      // feature_test_macros(7)
#include <fcntl.h>       // Для определения констант O_*
#include <unistd.h>

int pipe2(int pipefd[2], int flags); // O_CLOEXEC, O_DIRECT, O_NONBLOCK
```

pipefd – массив, использующийся для возврата двух файловых дескрипторов, относящихся к концам канала.

pipefd[0] относится к концу канала для чтения.

pipefd[1] относится к концу канала для записи.

Данные, записываемые в конец канала для записи, буферизируются ядром до тех пор, пока они не будут прочитаны из конца канала для чтения.

pipe2()

```
#define _GNU_SOURCE             // feature_test_macros(7)
#include <fcntl.h>              // Для определения констант O_*
#include <unistd.h>

int pipe2(int pipefd[2], // [0] -- чтение, [1] -- запись
          int flags);    // O_CLOEXEC, O_DIRECT, O_NONBLOCK
```

Если **flags** равен 0, то **pipe2(2)** эквивалентен **pipe(2)**.

Следующие значения могут быть объединены побитовым ИЛИ во флагах, чтобы получить другое поведение:

O_CLOEXEC

Устанавливает флаг закрытия (**FD_CLOEXEC**) при вызове exec для двух файловых дескрипторов.

O_NONBLOCK — устанавливает флаг состояния файла **O_NONBLOCK** для описаний открытого файла, на которые ссылаются новые файловые дескрипторы.

Использование этого флага избавляет от дополнительных вызовов **fcntl(2)** для достижения того же результата.

O_DIRECT

Создает канал, осуществляющий ввод-вывод в «пакетном» режиме. Каждый **write(2)** в канал обрабатывается как отдельный пакет, а операции **read(2)** из канала будут считывать по одному пакету за раз.

Следует обратить внимание на следующие моменты:

1) записи размером больше чем **PIPE_BUF** байт будут разделены на несколько пакетов.

Константа **PIPE_BUF** определена в **<limits.h>**.

2) если **read(2)** указывает размер буфера, который меньше размера следующего пакета, то считывается запрошенное количество байтов, а лишние байты в пакете отбрасываются.

Размером буфера, достаточным для чтения максимально больших пакетов, является **PIPE_BUF**.

Пакеты нулевой длины не поддерживаются — операция **read(2)**, указывающая нулевой размер буфера, не выполняется и возвращает **0**.

Старые ядра, не поддерживающие этот флаг, укажут на это с помощью ошибки EINVAL.

Начиная с Linux 4.5, можно изменить настройку **O_DIRECT** дескриптора файла канала с помощью **fcntl(2)**.

В случае успеха **pipe()** возвращает значение 0.

В случае неудачи возвращается -1 и устанавливается **errno**:

EFAULT — **pipefd** недействителен.

EINVAL — (pipe2()) недействительное значение **flags**.

EMFILE — достигнут предел количества открытых файловых дескрипторов для процесса.

ENFILE — достигнут общесистемный лимит на общее количество открытых файлов.

ENFILE — достигнуто жесткое ограничение пользователя на память, которая может быть выделена для каналов, и вызывающий объект не имеет привилегий.

Ниже пример простой программы, которая создает канал. Эта программа использует функцию **fork( )** для создания дочернего процесса. Родительский процесс записывает данные в канал, который считывается дочерним процессом.

```
#define _POSIX_C_SOURCE 200809L // для fdopen( )
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

// Читает символы из канала и выводит их на ‘stdout’.
void read_from_pipe (int file) {
  int c;
  FILE *stream = fdopen(file, "r"); // откр. поток, связанный с файл.дескриптором
  while ((c = fgetc(stream)) != EOF) {
      putchar(c);
  }
  fclose (stream);
}

// Пишет какой -либо текст в канал
void write_to_pipe (int file) {
    FILE *stream = fdopen(file, "w");
    fprintf(stream, "hello, world!\n");
    fprintf(stream, "goodbye, world!\n");
    fclose(stream);
}
```

```
int main (void) {

  pid_t pid;
  int   mypipe[2];

  // Создаем канал
  if (pipe(mypipe)) {
      fprintf (stderr, "Pipe failed.\n");
      return EXIT_FAILURE;
  }

  // Создаем дочерний процесс
  if ((pid = fork()) == (pid_t)0) {
      // дочерний процесс
      close (mypipe[1]);          // Сначала закрываем второй конец (для записи)
      read_from_pipe(mypipe[0]); // а теперь читаем из канала
      return EXIT_SUCCESS;
  } else if (pid < (pid_t)0) {
      // fork() не удался
      fprintf (stderr, "fork() не удался.\n");
      return EXIT_FAILURE;
  } else {
      // Родительский процесс
      close (mypipe[0]);          // Сначала закрываем второй конец (для чтения)
      write_to_pipe (mypipe[1]); // а теперь пишем в канал
      return EXIT_SUCCESS;
    }
}
```

Следующая программа создает канал, а затем вызывает **fork(2)** для создания дочернего процесса, который наследует дублированный набор дескрипторов файлов, которые ссылаются на тот же самый канал.

После **fork(2)** каждый процесс закрывает дескрипторы файлов, которые ему не нужны.

Затем родитель записывает строку, содержащуюся в аргументе командной строки программы, в конвейер, а потомок считывает эту строку побайтно из канала и выводит ее на **stdout**.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[]) {

    int   pipefd[2];
    pid_t cpid;
    char  buf;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <string>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
```

```
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (cpid == 0) {       // Дочерний процесс читает из канала
        close(pipefd[1]);  // поэтому закрываем неиспользуемый конец для записи

        while (read(pipefd[0], &buf, 1) > 0) {
            write(STDOUT_FILENO, &buf, 1);
        }
        write(STDOUT_FILENO, "\n", 1);
        close(pipefd[0]);
        exit(EXIT_SUCCESS);

    } else {               // Родитель пишет в канал argv[1]
        close(pipefd[0]);  // поэтому закрывает неиспользуемый конец для чтения
        write(pipefd[1], argv[1], strlen(argv[1]));
        close(pipefd[1]);   // Читающий увидит EOF
        wait(NULL);         // ждем завершения дочернего
        exit(EXIT_SUCCESS);
}   }
```

Использование канала для связи с дочерним процессом

Обычно каналы используются для отправки данных или получения данных от программы, запущенной в качестве дочернего процесса. Один из способов сделать это — использовать комбинацию **pipe()** для создания канала, **fork()** для создания дочернего процесса, **dup2()**, чтобы заставить дочерний процесс использовать канал в качестве стандартного потока ввода или вывода, и **exec()** для выполнения новой программы.

Как альтернативу, можно использовать **popen()** и **pclose()** – функции для организации канальных (конвейерных) потоков.

Преимущество использования **popen()** и **pclose()** состоит в том, что их интерфейс намного проще и удобнее в использовании. Однако, он не предлагает такой гибкости, как это имеет место в случае прямого использования низкоуровневых функций.

```
#include <stdio.h>

FILE *popen(const char *command, const char *type);
int   pclose(FILE *stream);
```

Функция **popen()** тесно связана с функцией **system()**. Она выполняет команду оболочки **command** как дочерний процесс. Однако вместо ожидания завершения команды она создает канал для дочернего процесса и возвращает поток (файловый), соответствующий этому каналу.

Поскольку канал – однонаправленная сущность, то в **type** может быть задан только один режим – либо «**r**», либо «**w**».

Если указать в качестве аргумента **type** значение «**r**», можно получать данные из стандартного выходного потока дочернего процесса, читая из потока. Дочерний процесс наследует свой стандартный входной канал от родительского процесса.

Точно так же, если указать «**w**», можно писать в поток для отправки данных на стандартный входной поток дочернего процесса. Дочерний процесс наследует свой стандартный поток вывода от родительского процесса.

В случае ошибки **popen()** возвращает нулевой указатель. Это может произойти, если канал или поток не могут быть созданы, если дочерний процесс не может быть создан или если программа не может быть выполнена.

Кроме «**w**» и «**r**» type может содержать букву «**e**», которая указывает на необходимость установки флага закрытия файлового дескриптора при выполнении (**FD-CLOEXEC**).

Если не удалось выделить память, **errno** не изменяется.

Если завершились ошибкой **fork()** или **pipe()**, в **errno** будет код ошибки.

Если некорректно значение **type**, в **errno** будет **EINVAL**.

Функция **pclose()** используется для закрытия потока, созданного с помощью **popen()**.

Она ожидает завершения дочернего процесса и возвращает его значение состояния (код выхода), как и в случае **system()**.

По умолчанию потоки вывода **popen()** блокируемые и буферизованные.

Если **pclose()** не удаётся получить код выхода потомка, то **errno** присваивается **ECHILD**.

Ниже пример, показывающий, как использовать **popen( )** и **pclose( )** для фильтрации вывода через другую программу, в данном случае программу просмотра **less**.

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

void write_data(FILE * stream);

int main (void) {

    FILE *output = popen("less", "w");
    if (!output) {
        fprintf (stderr, "Некорректный параметр или слишком много файлов.\n");
        return EXIT_FAILURE;
    }
    write_data(output);          // выводим данные в поток
    if (pclose(output) != 0) {
        fprintf(stderr, "Не могу запустить less или еще какая-то ошибка.\n");
    }
  return EXIT_SUCCESS;
}
```

```
void write_data (FILE * stream) {

    int i;
    for (i = 0; i < 100; i++) {
        fprintf(stream, "%d\n", i);
    }
    if (ferror(stream)) {
        fprintf(stderr, "Вывод в поток не удался.\n");
        exit (EXIT_FAILURE);
    }
}
```

mkfifo() – Специальные файлы FIFO

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);

#include <fcntl.h>           // определения констант AT_*
#include <sys/stat.h>

// _POSIX_C_SOURCE >= 200809L
int mkfifoat(int dirfd, const char *pathname, mode_t mode);
```

Функция **mkfifo()** создает специальный файл FIFO с именем **pathname**.
mode – используется для установки прав доступа к файлу.

Специальный файл FIFO похож на канал, за исключением того, что он создается другим способом. Вместо того, чтобы быть анонимным каналом связи, специальный файл FIFO вводится в файловую систему с помощью вызова **mkfifo()**.

После того, как специальный файл FIFO создан, любой процесс может открыть его для чтения или записи точно так же, как и обычный файл. Однако, прежде чем можно выполнять на нем какие-либо операции ввода или вывода, файл должен быть открыт с обоих концов *одновременно*.

Открытие FIFO для чтения обычно блокируется до тех пор, пока какой-либо другой процесс не откроет тот же FIFO для записи, и наоборот.

Нормальное успешное возвращаемое значение **mkfifo()** — **0**.

В случае ошибки возвращается **-1**.

В дополнение к обычным ошибкам в имени файла для этой функции определены следующие условия ошибки **errno**:

EEXIST — указанный файл уже существует

ENOSPC — в каталоге или на файловой системе нет места

EROFS — каталог, который будет содержать файл FIFO, находится в файловой системе, доступной только для чтения.

mkfifoat() работает также как как **mkfifo()**, за исключением:

Если в **pathname** задан относительный путь, то он считается относительно каталога, на который ссылается файловый дескриптор **dirfd**, а не относительно текущего рабочего каталога вызывающего процесса, как это имеет место в **mkfifo(3)**).

Если в **pathname** задан относительный путь и **dirfd** равно специальному значению **AT_FDCWD**, то **pathname** рассматривается относительно текущего рабочего каталога вызывающего процесса (как **mkfifo(3)**).

Если в **pathname** задан абсолютный путь, то **dirfd** игнорируется.

Атомарность конвейерного ввода-вывода (Atomicity of Pipe I/O)

Если размер записываемых данных не превышает «PIPE_BUF», чтение или запись данных канала является «атомарным». Это означает, что передача данных кажется мгновенной единицей, поскольку ничто другое в системе не может наблюдать состояние, в котором она частично завершена.

Атомарный ввод-вывод может начаться не сразу (может потребоваться ожидание буферного пространства или данных), но как только он начнется, он сразу же завершится.

Чтение или запись большего объема данных может быть не атомарным. Например, выходные данные других процессов, совместно использующих дескриптор, могут перемежаться. Кроме того, как только PIPE_BUF символов будет записано, дальнейшая запись будет заблокирована до тех пор, пока некоторые символы не будут прочитаны.