ОПЕРАЦИОННЫЕ СИСТЕМЫ И СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лекция 19 — Асинхронный ввод-вывод

Преподаватель: Поденок Леонид Петрович, 505а-5

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

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

Кафедра ЭВМ, 2023

2023.05.31

Оглавление

I/O в программном обеспечении

В среде программного обеспечения существует большое количество видов ввода и вывода:
- блокирующий (наивный);
- неблокирующий;
- мультиплексированный;
- асинхронный.

Блокирующий метод

Любая пользовательская программа запускается внутри процесса, а код выполняется в контексте потока.

Предположим, мы пишем программу, которой нужно читать информацию из файла.

С блокирующим вводом-выводом мы просим ОС «усыпить» читающий поток и «разбудить» его после того, как данные из файла будут доступны для чтения.

То есть блокирующий ввод-вывод называется так, потому что поток, который его использует, блокируется и переходит в режим ожидания, пока ввод-вывод не будет завершён.

Неблокирующий метод

Проблема блокирующего метода заключается в том, что поток будет спать, пока ввод-вывод не завершится – поток не сможет выполнять никаких других задач, кроме ожидания завершения ввода-вывода.

Иногда программе больше и не надо ничего делать.

В противном же случае во время ожидания ввода-вывода было бы полезно выполнять другие задачи.

Один из способов осуществить это — использовать ***неблокирующий*** ввод-вывод.

Идея заключается в том, что когда программа делает вызов на чтение файла, ОС не будет блокировать поток, а просто вернёт программе либо готовые данные, либо информацию о том, что ввод-вывод ещё не завершен. В этом случае поток не будет заблокирован, но программе придётся позже проверять, завершён ли ввод-вывод.

Это означает, что можно по-разному реагировать в зависимости от того, завершён ли ввод-вывод и выполнять другие задачи.

Когда же программе снова понадобится ввод-вывод, она сможет повторно попробовать прочесть содержимое файла, и если ввод-вывод завершён, то получит содержимое файла. В противном случае ПО снова получит сообщение о том, что операция ещё не завершена и сможет заняться другими задачами.

Мультиплексированный метод

Проблема с неблокирующим вводом-выводом в том, что с ним не удобно работать, если задачи, которые выполняет программа, ожидая ввода-вывода, сами из себя представляют ввод-вывод.

Допустим, программа просит ОС прочитать содержимое из файла A, после чего выполняет какие-нибудь сложные вычисления, потом проверяет, завершилось ли чтение файла A:

- если да, то просто продолжает ту работу, для которой нужно было содержимое файла;
- иначе снова выполняет некоторое количество сложных вычислений и так далее.

Допустим, программе не нужно выполнять сложные вычисления, а нужно просто прочесть файл A и одновременно файл B. В этом случае пока программа ожидает завершения обработки файла A, она делает неблокирующий вызов чтения содержимого файла B.

Во время ожидания программе больше нечего делать, потому она входит в бесконечный цикл опроса, проверяя, готово ли A и готово ли B. Это либо нагрузит процессор проверками состояния неблокирующих вызовов, либо придётся вручную добавить какое-то произвольное время, которое поток будет «спать», а значит, программа заметит, что ввод-вывод готов, позже, что отрицательно скажется на пропускной способности ПО.

Во избежание этого можно использовать ***мультиплексированный ввод-вывод***. Он тоже блокирует поток на операциях ввода-вывода, но вместо того, чтобы производить блокировку по очереди, можно запланировать все операции ввода-вывода, которые нужно сделать, чтобы продолжить выполнение, и блокировать их все.

Операционная система разбудит поток, когда какая-нибудь из операций завершится.

В некоторых реализациях мультиплексированного ввода-вывода можно даже точно указать, чтобы поток был разбужен только тогда, когда будет завершён заданный набор операций ввода-вывода, например, когда будут готовы файлы A и C, или файлы B и D.

Например, программа делает неблокирующий вызов чтения файла A, потом неблокирующий вызов чтения файла B, и наконец говорит операционной системе — усыпи мой поток, и разбуди его, когда A и B будут оба готовы, или когда один из них будет готов.

Асинхронный метод

Проблема мультиплексированного ввода-вывода в том, что поток всё-таки спит, пока ввод-вывод не будет готов для обработки.

Для многих программ это подходит, поскольку у них нет других задач, пока они ждут завершения операций ввода-вывода.

Но иногда у них есть и другие задачи[1].

Например, ПО вычисляет цифры числа π и одновременно обрабатывает данные из нескольких файлов. Хотелось бы запланировать все операции чтения файлов, а пока программа ждёт их завершения, вычислять цифры числа π. Когда какой-нибудь из файлов будет прочитан, ПО выполнит обработку и продолжит вычислять цифры числа π дальше, пока ещё один файл не будет прочитан.

Чтобы это работало, нужно, чтобы вычисление цифр числа π могло быть прервано вводом-выводом, когда он завершается.

Это можно сделать с помощью обратных вызовов, связанных с событиями.

Вызов на чтение принимает функцию обратного вызова и немедленно возвращается.

Как только ввод-вывод завершается, операционная система остановит ваш поток и выполнит обратный вызов.

Когда обратный вызов завершится, система возобновит работу вашего потока.

---

1) mining, SETI ...

Многопоточность или однопоточность?

Все вышеописанные способы ввода-вывода работают в рамках одного потока — главного потока приложения.

На самом деле, для выполнения ввода-вывода не требуется отдельный поток, поскольку, вся периферия выполняет ввод-вывод асинхронно.

Поэтому возможно делать блокирующий, неблокирующий, мультиплексированный и асинхронный ввод-вывод в однопоточной модели.

Т.е. одновременный ввод-вывод с помощью вышеуказанных методов может работать и без поддержки многопоточности.

Тем не менее, обработка результатов операций ввода-вывода может быть и многопоточной.

Это позволяет программе делать одновременные вычисления поверх одновременного ввода-вывода. Так что, ничто не мешает совмещать многопоточность и вышеперечисленные механизмы ввода-вывода.

На самом деле, есть пятый популярный метод ввода-вывода, который требует многопоточности. Его часто путают с неблокирующим или асинхронным, поскольку он похож по интерфейсу на эти два.

Этот метод работает просто — используется блокирующий ввод-вывод, но каждый блокирующий вызов делается в отдельном потоке. В зависимости от реализации, вызов либо использует функцию обратного вызова, либо использует модель опроса.

Формы ввода-вывода и примеры функций POSIX[2]

	Блокирующий	неблокирующий	асинхронный
API	`write, read`	`write, read + poll/select`	`aio_write, aio_read`

Все формы асинхронного ввода-вывода могут инициировать потенциальные конфликты ресурсов и привести к связанным с ними сбоям.

Для предотвращения этого требуется тщательное программирование (часто с использованием взаимного исключения, семафоров и т. д.).

Существует несколько широких классов API реализующих асинхронный ввод-вывод.

Форма API, предоставляемого приложению, не обязательно соответствует механизму, фактически предоставляемому операционной системой — возможны эмуляции.

Более того, одно приложение может использовать более одного метода, в зависимости от его потребностей и желаний программистов. Многие операционные системы предоставляют более одного из этих механизмов, возможно, что некоторые могут предоставлять их все.

2) https://en.wikipedia.org/wiki/Asynchronous_I/O

Процессы

В многозадачной операционной системе обработка может быть распределена между различными процессами, которые выполняются независимо, имеют свою собственную память и обрабатывают свои собственные потоки ввода-вывода — эти потоки обычно соединяются в конвейеры.

Однако, процессы довольно дороги в создании и обслуживании, поэтому это решение работает хорошо только в том случае, если набор процессов небольшой и относительно стабильный.

Также предполагается, что помимо обработки ввода-вывода отдельные процессы в рамках приложения работают независимо. Если же им нужно общаться более тесно, их координация может стать трудной. Расширением этого подхода является программирование потока данных, которое позволяет создавать более сложные сети, чем просто цепочки, поддерживаемые конвейерами.

Опрос (Polling)

Варианты:

а) получить ошибку, если невозможно выполнить сразу и повторить вызов позже;

б) получить сообщение, когда ввод-вывод можно сделать без блокировки и выполнить вызов.

Опрос предоставляет собой неблокирующий синхронный API, который можно использовать для реализации некоторого асинхронного API.

Доступно в традиционных Unix и Windows.

Его основная проблема заключается в том, что он тратит время ЦП на многократные опросы, если процессу, инициировавшему ввод/вывод больше нечего делать — это сокращает время, доступное для других процессов.

Недостатки

Поскольку приложение для опроса по существу является однопоточным, оно может быть не в состоянии полностью использовать параллелизм ввода-вывода, на который способно оборудование.

Циклы select(poll)

Метод доступен в BSD Unix и почти во всех остальных ОС со стеком протоколов TCP/IP, который либо прямо использует, либо моделирует реализацию BSD. Это вариант опроса.

Цикл **select** использует системный вызов **`select()`** для перехода в спящий режим до тех пор, пока:

- в дескрипторе файла не возникнет условие, например, станут доступны для чтения данные;
- не истечет время ожидания;
- не будет получен сигнал (например, когда дочерний процесс завершается).

Возвращаемые параметры вызова **`select()`**, анализируются в цикле и определяется, какой дескриптор файла был изменен, и выполняется соответствующий код.

Часто для простоты использования цикл **select** реализуется как цикл событий, возможно, с использованием функций обратного вызова — такой подход особенно хорошо годится для программирования, управляемого событиями.

Недостатки

Этот метод надежен и относительно эффективен, однако он зависит от парадигмы Unix, согласно которой «все является файлом» — любой блокирующий ввод-вывод, не связанный с указанным файловым дескриптором, заблокирует процесс.

Цикл **select** основан на предположении, что можно задействовать весь ввод/вывод в одном вызове **`select()`**, и это может создавать проблемы при использовании библиотек, которые выполняют собственный ввод-вывод.

Дополнительная потенциальная проблема заключается в том, что операции **select** и ввода/вывода не атомарны, поэтому результат **`select()`** может фактически оказаться фальшивым:

- если два процесса читают из одного файлового дескриптора (возможно, из-за плохого дизайна), **`select()`** может указать на доступность чтения данных, которые к моменту выполнения чтения могут исчезнуть, что приведет к блокировке;

- если два процесса записывают в один файловый дескриптор (не так уж редко), **`select()`** может указать на немедленную возможность записи, но запись может все еще блокироваться, потому что буфер был заполнен другим процессом в промежуточный период, или из-за слишком длинной записи для доступного буфера или другими способами, не подходящими для получателя.

Цикл **select** не достигает максимальной эффективности, возможной, например, с помощью метода очередей завершения, потому что семантика вызова **`select()`** предполагает настройку допустимого набора событий для каждого вызова и это требует некоторого количества времени.

Данный факт создает небольшие накладные расходы для пользовательских приложений, которые могут открывать один файловый дескриптор для оконной системы и несколько для открытых файлов, но становится серьезной проблемой по мере роста числа потенциальных источников событий.

Некоторые системы Unix предоставляют системные вызовы с лучшим масштабированием, например, **`epoll()`** в Linux (который заполняет массив выбора возврата только теми источниками событий, в которых произошло событие), **`kqueue()`** – во FreeBSD, порты событий (**`/dev/poll`**) в Solaris.

В SVR3 Unix есть системный вызов **`poll()`**. Возможно, это лучшее имя, чем **`select()`**, но это по сути одно и то же.

Системы SVR4 Unix (и, следовательно, POSIX) предоставляют оба вызова.

Сигналы (прерывания)

Доступно в BSD и POSIX Unix.

Ввод-вывод выполняется асинхронно, и по его завершении генерируется сигнал (прерывание).

Недостатки

Как и в низкоуровневом программировании на уровне ядра, возможности, доступные для безопасного использования в обработчике сигналов, ограничены, а основной поток процесса может быть прерван практически в любой точке, что может привести к несогласованным структурам данных, которые видны в обработчике сигнала.

Сигнальный подход, хотя и относительно прост для реализации, однако привносит в прикладную программу нежелательный багаж, связанный с написанием системы обработки прерываний.

Худшая характеристика сигнального подхода состоит в том, что каждый блокирующий (синхронный) системный вызов может быть потенциально прерван — обычно программист должен включать код повтора при каждом вызове.

Функции обратного вызова (Callback functions)

Доступен в классической Mac OS, VMS и Windows.

Обладает многими характеристиками сигнального метода, поскольку в основе своей это то же самое, хотя и редко признается таковым.

Разница в том, что каждый запрос ввода-вывода обычно может иметь свою собственную функцию завершения, тогда как сигнальная система имеет единственный обратный вызов.

Этот метод используется некоторыми сетевыми фреймворками из-за простоты реализации, тем не менее, метод может привести к появлению вложенного и беспорядочного кода, получившего название «ад обратных вызовов» (Node.js).

Недостатки

Потенциальная проблема использования обратных вызовов заключается в том, что может неуправляемо расти стек, поскольку очень часто после завершения одного ввода-вывода планируется другой.

Если операция должно быть немедленно удовлетворена, предыдущий обратный вызов не «откручивается» из стека, а «закрывается» следующим вызовом.

Системы, предотвращающие такое поведение (например, «промежуточное» планирование новых работ), усложняют работу и снижают производительность.

На практике, однако, это обычно не проблема, потому что новая операция ввода/вывода обычно возвращается, как только запускается новый ввод/вывод, что позволяет стеку "раскручиваться".

Проблему также можно предотвратить, избегая дальнейших обратных вызовов с помощью очереди до тех пор, пока не вернется первый обратный вызов.

Облегченные процессы или потоки (Light-weight processes or threads)

Облегченные процессы (LWP) или потоки, доступные в более современных Unix, не обеспечивают изоляции данных, что затрудняет их координацию.

Это отсутствие изоляции создает свои собственные проблемы, обычно требующие механизмов синхронизации, предоставляемых ядром, и поточно-ориентированных библиотек.

Каждый LWP или поток сам по себе использует традиционный блокирующий синхронный ввод-вывод. Однако, необходимый отдельный стек для каждого потока может препятствовать крупномасштабным реализациям, использующим очень большое количество потоков (сервера).

Очереди завершения/порты (Completion queues/ports)

Доступно в Microsoft Windows, Solaris, AmigaOS, DNIX и Linux (с использованием **io_uring**[3], доступно в версии 5.1 и выше).

Запросы ввода-вывода отправляются асинхронно, но уведомления о завершении предоставляются через механизм очереди синхронизации в порядке их выполнения.

Обычно ассоциируется с конечным автоматом (программирование, управляемое событиями), что может создать трудности программирования процессов, не использующих асинхронный ввод-вывод или использующих одну из других форм.

Не требует дополнительных специальных механизмов синхронизации или поточно-ориентированных библиотек.

---

3) ftp://...@lsi/ОсиСП-2023/info/io_uring.pdf – Efficient IO with io_uring.

Реализации

Подавляющее большинство универсального вычислительного оборудования полностью полагается на два метода реализации асинхронного ввода-вывода — опрос и прерывания.

Обычно оба метода используются вместе, баланс во многом зависит от конструкции оборудования и его требуемых характеристик производительности.

DMA – DMA сам по себе не является еще одним независимым методом — это просто средство, с помощью которого можно выполнять больше работы за один опрос или прерывание.

Опрос – возможны чистые системы опроса, небольшие микроконтроллеры часто строятся именно таким образом.

Если максимальная производительность необходима только для нескольких задач из набора, опрос также годится, несмотря на то, что это реализуется за счет любых других задач, поскольку могут быть достаточно велики накладные расходы на получение прерываний — для обслуживания прерывания требуется время на сохранения хотя бы части состояния процессора, а также время, необходимое для возобновления прерванной задачи.

Тем не менее, большинство универсальных вычислительных систем в значительной степени полагаются на прерывания.

Прерывания – возможна чистая система прерываний, хотя обычно требуется некоторый компонент опроса, поскольку очень часто несколько потенциальных источников прерываний используют общую линию сигнала прерывания, и в этом случае опрос используется в драйвере устройства для разрешения проблемы. Это время разрешения также способствует снижению производительности системы прерываний.

Гибридные реализации

Также возможны гибридные подходы, в которых прерывание может инициировать начало некоторого пакета асинхронного ввода-вывода, а опрос используется внутри самого пакета.

Этот метод распространен в драйверах высокоскоростных устройств, таких как сетевые или дисковые, где время, теряемое при возврате к прерванной задаче больше, чем время до следующего требуемого обслуживания.

Современное оборудование ввода-вывода в значительной степени полагается на DMA и большие буферы данных, что позволяет компенсировать относительно плохо работающую систему прерываний. Обычно оно использует опрос внутри циклов драйвера и может демонстрировать огромную пропускную способность, поскольку в идеале опросы всегда успешны или, самое большее, повторяются небольшое количество раз.

Используя только эти два инструмента (опрос и прерывания), синтезируются все формы асинхронного ввода-вывода. Например, в виртуальной машине Java (JVM), асинхронный в/в может быть синтезирован, даже если среда, в которой работает JVM его не поддерживает.

Техническое противоречие в методе опроса

- каждый цикл ЦП, который представляет собой опрос, тратится впустую и теряется из-за накладных расходов, а не для выполнения желаемой задачи.

- каждый цикл ЦП, не являющийся опросом, означает увеличение задержки реакции на ожидающий ввод-вывод.

Достичь приемлемого баланса между этими двумя противоборствующими силами сложно.

Именно поэтому были изобретены системы аппаратных прерываний.

Максимальной эффективности можно достичь, если свести к минимуму объем работы, который должен быть выполнен при получении прерывания, чтобы пробудить соответствующее приложение.

Прерывания очень хорошо отображаются на сигналы, функции обратного вызова и очереди завершения, поэтому такие системы могут быть очень эффективными.

aio — введение в асинхронный ввод-вывод POSIX

Интерфейс асинхронного ввода-вывода POSIX (AIO) позволяет приложениям запускать одну или несколько операций ввода-вывода, которые выполняются асинхронно (т. е., в фоновом режиме). Приложение может выбрать каким образом оно будет уведомлено о завершении операции ввода-вывода: с помощью сигнала, созданием новой нити или вообще не получать уведомления.

Интерфейс POSIX AIO состоит из следующих функций:

aio_read(3) Ставит запрос на чтение в очередь. Это асинхронный аналог read(2).

aio_write(3) Ставит запрос на запись в очередь. Это асинхронный аналог write(2).

aio_fsync(3) Ставит запрос синхронизации операций ввода-вывода над файловым дескриптором. Это асинхронный аналог **fsync(2)** и **fdatasync(2)**.

aio_error(3) Возвращает информацию о состоянии поставленного в очередь запроса ввода-вывода.

aio_return(3) Возвращает информацию о выполненном запросе ввода-вывода.

aio_suspend(3) Приостанавливает вызывающего до тех пор, пока не выполнится один или более указанных запросов ввода-вывода.

aio_cancel(3) Пытается отменить ожидающие выполнения запросы ввода-вывода над заданным файловым дескриптором.

lio_listio(3) Ставит в очередь сразу несколько запросов ввода-вывода за один вызов функции.

Параметры, которые управляют операцией ввода-вывода, задаются в структуре **aiocb** («блок управления асинхронным вводом-выводом») .

Аргумент данного типа передаётся во все функции, перечисленные ранее.

Данная структура имеет следующий вид:

```
#include <aiocb.h>

struct aiocb {
    // Порядок данных полей определяется реализацией

    int             aio_fildes;     // файловый дескриптор
    off_t           aio_offset;     // файловое смещение
    volatile void  *aio_buf;        // расположение буфера
    size_t          aio_nbytes;     // длина передачи данных
    int             aio_reqprio;    // приоритет запроса
    struct sigevent aio_sigevent;   // метод уведомления
    int             aio_lio_opcode; // выполняемая операция; только в lio_listio()

    // Различные поля, используемые в реализациях, не показаны
};

// Коды операций для 'aio_lio_opcode'
enum {
    LIO_READ,
    LIO_WRITE,
    LIO_NOP
};
```

Поля этой структуры имеют следующее назначение:

aio_fildes Файловый дескриптор, над которым будут выполняться операции ввода-вывода.

aio_offset Файловое смещение, начиная с которого будут выполняться операции ввода-вывода.

aio_buf Буфер, используемый для пересылки данных при операции чтения или записи.

aio_nbytes Размер буфера, на который указывает **aio_buf**.

aio_reqprio В этом поле задаётся значение, которое вычитается из приоритета реального времени вызывающей нити, чтобы определить приоритет выполнения данного запроса ввода-вывода (**pthread_setschedparam(3)**). Указываемое значение должно быть в диапазоне от 0 и до значения, возвращаемого **sysconf(_SC_AIO_PRIO_DELTA_MAX)**. Данное поле игнорируется при операциях синхронизации файла.

aio_sigevent В этом поле задаётся структура **sigevent** — структура для уведомления из асинхронных процедур, которая указывает как вызывающему должно быть сообщено о завершении анонимной операции ввода-вывода.

Возможные значения для **aio_sigevent.sigev_notify**:

SIGEV_NONE;

SIGEV_SIGNAL;

SIGEV_THREAD.

aio_lio_opcode Задаёт тип операции, которая будет выполнена; используется только в **lio_listio(3)**.

В дополнении к стандартным функциям, перечисленным ранее, в библиотеке GNU C есть следующее расширение программного интерфейса POSIX AIO:

aio_init(3) Позволяет изменить настройки поведения реализации glibc для POSIX AIO.

Ошибки, связанные с **`struct aiocb`**

EINVAL — значение поля **`aio_reqprio`** структуры **`aiocb`** меньше 0 или больше, чем значение ограничения, возвращаемое вызовом **`sysconf(_SC_AIO_PRIO_DELTA_MAX)`**.

Желательно обнулять буфер блока управления перед использованием. Буфер блока управления и буфер, который задаётся в aio_buf, не должны изменяться во время выполнения операции ввода-вывода. Данные буферы должны оставаться рабочими до завершения операции ввода-вывода.

Одновременное выполнение операций чтения или записи через совместно используемую структуру **`aiocb`** приводит к непредсказуемым результатам.

Имеющаяся реализация Linux POSIX AIO предоставляется glibc в пользовательском пространстве. Она имеет ряд ограничений, наиболее существенные из которых — затраты на сопровождение нескольких нитей при операциях ввода-вывода и плохое масштабирование.

sigevent — структура для уведомления из асинхронных процедур

Структура **sigevent** используется в различных программных интерфейсах для описания способа, которым нужно уведомлять процесс о событии (например, окончание асинхронного запроса, истечение таймера или поступление сообщения).

Определение, приведённое ниже, приблизительно — некоторые поля в структуре **sigevent** могут быть определены как часть объединения.

Программы должны использовать только те поля, которые применимы к значению, заданном в **sigev_notify** — способ выполнения уведомления.

```
#include <signal.h>

union sigval {          // Данные, передаваемые с уведомлением
    int     sival_int;  // целое
    void   *sival_ptr;  // указатель
};

struct sigevent {
    int          sigev_notify; // метод уведомления
    int          sigev_signo;  // сигнал уведомления
    union sigval sigev_value;  // данные, передаваемые с уведомлением

    // функция, используемая для потока уведомления (SIGEV_THREAD)
    void    (*sigev_notify_function)(union sigval);
    void     *sigev_notify_attributes; // атриб. для потока уведомления (SIGEV_THREAD)
    pid_t     sigev_notify_thread_id;  // ID потока для уведомления (SIGEV_THREAD_ID)
};
```

В поле **sigev_notify** задаётся как выполняется уведомление. Значением поля может быть:

SIGEV_NONE — «Пустое» уведомление: ничего не делать при возникновении события.

SIGEV_SIGNAL — Уведомить процесс, отправив сигнал, указанный в **sigev_signo**.

Если сигнал пойман обработчиком сигнала, который зарегистрирован с помощью **sigaction(2)** с флагом **SA_SIGINFO**, то следующим полям в структуре **siginfo_t**, передаваемой во втором аргументе обработчика, назначаются значения:

si_code — в этом поле задаётся значение, которое зависит от программного интерфейса, доставляющего уведомление.

si_signo — в этом поле указывается номер сигнала (т. е., тоже значение, что и в **sigev_signo**).

si_value — в этом поле содержится значение, указанное в **sigev_value**.

В зависимости от программного интерфейса остальным полям в структуре **siginfo_t** также могут быть присвоены значения.

SIGEV_THREAD — Уведомить процесс с помощью вызова **sigev_notify_function** «как если бы» это была бы начальная функция новой нити (среди возможностей реализации здесь может быть:

- каждое уведомление таймера приводит к созданию новой нити, или создаётся нить для получения всех уведомлений). Функция вызывается с единственным аргументом **sigev_value**. Если **sigev_notify_attributes** не равно **NULL**, то значение должно указывать на структуру **pthread_attr_t**, в которой определены атрибуты новой нити (**pthread_attr_init(3)**).

SIGEV_THREAD_ID – (linux) в настоящее время используется только таймерами POSIX (timer_create(2)).

aio_read() — асинхронное чтение

```
#include <aio.h>

int aio_read(struct aiocb *aiocbp);
```

Компонуется при указании параметра -lrt.

Функция **aio_read()** ставит в очередь запрос ввода-вывода, описанный в буфере, на который указывает **aiocbp**.

Эта функция является асинхронным аналогом вызова **read(2)**.

Полям **aio_fildes**, **aio_buf** и **aio_nbytes** структуры, на которую указывает **aiocbp**, соответствуют (в указанном порядке) аргументам функции **read(fd, buf, count)**.

Чтение данных выполняется начиная с абсолютного положения в файле **aiocbp->aio_offset**, независимо от смещения в файле.

После вызова значение смещения в файле не определено.

Прилагательное «асинхронный» означает, что вызов возвращает управление сразу после установки запроса в очередь — при завершении вызова чтение может уже выполнится, а может и нет.

Для проверки выполнения чтения следует использовать **aio_error(3)**.

Состояние возврата завершённой операции ввода-вывода можно получить с помощью **aio_return(3)**.

Асинхронное уведомление о выполнении ввода-вывода можно получить, установив соответствующим образом **aiocbp->aio_sigevent**.

Если определён макрос **_POSIX_PRIORITIZED_IO** и данный файл его поддерживает, асинхронная операция устанавливается в очередь с приоритетом вызывающего процесса минус **aiocbp->aio_reqprio**.

Поле **aiocbp->aio_lio_opcode** игнорируется.

Если смещение превышает максимум, данные из обычного файла не читаются.

Возвращаемое значение

При успешном выполнении возвращается **0**.

При ошибке запрос не устанавливается в очередь, возвращается **-1**, **errno** присваивается соответствующее значение.

Если ошибка обнаруживается не сразу, то о ней будет сообщено посредством **aio_return(3)** (возвращается состояние -1) и **aio_error(3)** (состояние ошибки — всё, что было бы в **errno**, например **EBADF**).

Ошибки

EAGAIN — не хватает ресурсов.

EBADF — значение **aio_fildes** не является допустимым файловым дескриптором для открытия на чтение.

EINVAL — одно или несколько значений у **aio_offset**, **aio_reqprio** или **aio_nbytes** неверны.

ENOSYS — функция **aio_read( )** не реализована.

EOVERFLOW — файл является обычным файлом, мы начинаем читать его до конца файла и хотим получить не менее одного байта, но начальная позиция находится за максимальным значением смещения этого файла.

Примечания

Желательно обнулять буфер блока управления перед использованием.

Блок управления не должен изменяться во время выполнения операции чтения.

Нельзя читать или писать в буферные области во время выполнения операций, иначе результат непредсказуем.

Используемые области памяти должны оставаться корректными (valid).

Одновременное выполнение операций ввода-вывода через совместно используемую структуру `aiocb` приводит к непредсказуемым результатам.

aio_write() — асинхронная запись

```
#include <aio.h>

int aio_write(struct aiocb *aiocbp);  // Компонуется при указании параметра -lrt.
```

Функция **aio_write()** ставит в очередь запрос ввода-вывода, описанный в буфере, на который указывает **aiocbp**.

Эта функция является асинхронным аналогом вызова **write(2)**.

Полям **aio_fildes**, **aio_buf** и **aio_nbytes** структуры, на которую указывает **aiocbp**, соответствуют (в указанном порядке) аргументам функции **write(fd, buf, count)**.

Если не установлен флаг **O_APPEND**, то данные записываются начиная от абсолютного положения в файле **aiocbp->aio_offset**, независимо от смещения в файле.

Если флаг **O_APPEND** установлен, то данные записываются в конец файла в том порядке, в котором запускались вызовы **aio_write()**.

После вызова значение смещения файла не определено.

Прилагательное «асинхронный» означает, что вызов возвращает управление сразу после установки запроса в очередь — при завершении вызова запись может уже выполнится, а может и нет.

Для проверки выполнения операции следует использовать **aio_error(3)**.

Состояние возврата завершённой операции ввода-вывода можно получить с помощью **aio_return(3)**.

Асинхронное уведомление о выполнении ввода-вывода можно получить, установив соответствующим образом **aiocbp->aio_sigevent**.

Если определён макрос **_POSIX_PRIORITIZED_IO** и данный файл его поддерживает, асинхронная операция устанавливается в очередь с приоритетом вызывающего процесса минус **aiocbp->aio_reqprio**.

Поле **aiocbp->aio_lio_opcode** игнорируется.

Если смещение превышает максимум, запись в обычный файл не производится.

Возвращаемое значение

При успешном выполнении возвращается **0**.

При ошибке запрос не устанавливается в очередь, возвращается **-1**, **errno** присваивается соответствующее значение.

Если ошибка обнаруживается не сразу, то о ней будет сообщено посредством **aio_return(3)** (возвращается состояние -1) и **aio_error(3)** (состояние ошибки — всё, что было бы в **errno**, например **EBADF**).

Ошибки

EAGAIN Не хватает ресурсов.

EBADF Значение **aio_fildes** не является правильным файловым дескриптором, открытым для записи.

EFBIG Файл является обычным файлом, мы хотим записать не менее одного байта, но начальная позиция равна максимальному значению смещения этого файла или превышает его.

EINVAL Одно или несколько значений у **aio_offset**, **aio_reqprio** или **aio_nbytes**, неверны.

ENOSYS Функция **aio_write()** не реализована.

Рекомендуется обнулять управляющие блоки перед использованием. Управляющие блоки не должны изменяться пока выполняются операции ввода-вывода.

Нельзя читать или писать в буферные области во время выполнения операций, иначе результат непредсказуем.

Используемые области памяти должны оставаться корректными (valid).

Одновременное выполнение операций ввода-вывода через совместно используемую структуру `aiocb` приводит к непредсказуемым результатам.

lio_listio() — запускает список запросов ввода-вывода на выполнение

```
#include <aio.h>

int lio_listio(int mode,                          // LIO_WAIT, LIO_NOWAIT
               struct aiocb *const aiocb_list[],
               int nitems,
               struct sigevent *sevp);
```

Компонуется при указании параметра -lrt.

Функция **lio_listio()** запускает на выполнение список операций ввода-вывода, описанных в массиве **aiocb_list**.

Значение операции **mode** может быть одним из следующих:

LIO_WAIT — вызов не завершается до тех пор, пока не будут выполнены все операции. Аргумент **sevp** при этом игнорируется.

LIO_NOWAIT — операции ввода-вывода ставятся в очередь на обработку и вызов сразу завершается. После выполнения всех операций ввода-вывода посылается асинхронное уведомление, задаваемое в аргументе **sevp**. Если значение **sevp** равно **NULL**, то асинхронные уведомления не посылаются.

Аргумент **aiocb_list** представляет собой массив указателей на структуры **aiocb**, в которых описаны операции ввода-вывода.

Эти операции выполняются в произвольном порядке.

В аргументе **nitems** указывается размер массива **aiocb_list**.

Указатели со значением **NULL** в **aiocb_list** игнорируются.

В каждом управляющем блоке в **aiocb_list** в поле **aio_lio_opcode** задаётся выполняемая операция ввода вывода:

LIO_READ — выполнить операцию чтения. Операция ставится в очередь как вызов **aio_read(3)** с указанным управляющим блоком.

LIO_WRITE — выполнить операцию записи. Операция ставится в очередь как вызов **aio_write(3)** с указанным управляющим блоком.

LIO_NOP — игнорировать управляющий блок.

Остальные поля в каждом управляющем блоке имеют то же назначение, что и для **aio_read(3)** и **aio_write(3)**.

Поля **aio_sigevent** в каждом управляющем блоке могут использоваться для указания уведомлений по отдельным операциям ввода-вывода (**sigevent(7)**).

Возвращаемое значение

При значении **mode** равным **LIO_NOWAIT**, если все операции ввода-вывода были поставлены в очередь, функция **lio_listio()** возвращает **0**.

При значении **mode** равным **LIO_WAIT**, если все операции ввода-вывода были выполнены без ошибок, функция **lio_listio()** возвращает **0**.

В противном случае возвращается **-1** и в **errno** содержится код ошибки.

В возвращаемом **lio_listio()** состоянии предоставляется информация только о самом вызове, а не об отдельных операция ввода-вывода. Одна или несколько операций ввода-вывода могут завершиться с ошибкой, но это не повлияет на выполнение остальных операций.

Состояние отдельный операций ввода-вывода в **aiocb_list** можно определить с помощью **aio_error(3)**.

После завершения операции её результат можно получить с помощью **aio_return(3)**.

Отдельные операции ввода-вывода могут завершиться с ошибкой по причинам, описанным в **aio_read(3)** и **aio_write(3)**.

Ошибки

Функция **lio_listio()** может завершиться с ошибкой по следующим причинам:

EAGAIN — не хватает ресурсов.

EAGAIN — количество операций ввода-вывода, указанное в **nitems**, превысило ограничение **AIO_MAX**.

EINTR — значение **mode** равно **LIO_WAIT** и был получен сигнал до завершения всех операций ввода-вывода, в том числе один из асинхронных сигналов о завершении ввода-вывода (**SA_RESTART**).

EINVAL — неправильное значение **mode**, или значение **nitems** превышает ограничение **AIO_LISTIO_MAX**.

EIO — одна или более операций, указанных в **aiocb_list**, завершились с ошибкой.

Приложение может проверить состояние каждой операции с помощью **aio_return(3)**.

Если вызов **lio_listio()** завершился с ошибкой **EAGAIN**, **EINTR** или **EIO**, то некоторые операции из **aiocb_list** могли всё же начаться.

Если вызов **lio_listio()** завершился по другой причине, то ни одна из операций ввода-вывода не началась.

Рекомендуется обнулять управляющие блоки перед использованием.

Управляющие блоки не должны изменяться пока выполняются операции ввода-вывода.

Нельзя читать или писать в буферные области во время выполнения операций, иначе результат непредсказуем.

Используемые области памяти должны оставаться корректными (valid).

Одновременное выполнение операций ввода-вывода через совместно используемую структуру `aiocb` приводит к непредсказуемым результатам.

aio_error() — возвращает состояние ошибки операции асинхронного в/в

```
#include <aio.h>

int aio_error(const struct aiocb *aiocbp); // Компонуется при указании параметра -lrt.
```

Функция **aio_error()** возвращает состояние ошибки запроса асинхронного ввода-вывода для указанного блока управления **aiocbp**.

Возвращаемое значение

Функция возвращает одно из следующих значений:

EINPROGRESS, если запрос ещё не выполнен.

ECANCELED, если запрос отменён.

0, если запрос выполнен без ошибок.

Если асинхронная операция ввода-вывода завершилась с ошибкой, то возвращается положительное число.

Это то же значение, которое сохраняется в переменной **errno** при синхронном вызове **read(2)**, **write(2)**, **fsync(2)** или **fdatasync(2)**.

Ошибки

EINVAL — значение **aiocbp** не указывает на блок управления запросом асинхронного ввода-вывода, значение результата которого ещё не получено.

ENOSYS — функция **aio_error()** не реализована.

aio_return() — возвращает состояние операции асинхронного в/в

```
#include <aio.h>

ssize_t aio_return(struct aiocb *aiocbp); // Компонуется при указании параметра -lrt.
```

Функция **aio_return()** возвращает окончательное значение завершения запроса асинхронного ввода-вывода, задаваемого указателем на контрольный блок **aiocbp**.

Эта функция должна вызываться один раз для любого запроса в случае, если aio_error(3) возвращает результат, отличный от EINPROGRESS.

Возвращаемое значение

Если операция асинхронного ввода-вывода завершена, данная функция возвращает значение, которое может быть возвращено в случае запроса синхронного вызова **read(2)**, **write(2)**, **fsync(2)** или **fdatasync(2)**.

При ошибке возвращается **-1** и **errno** изменяется соответствующим образом.

Если асинхронная операция ввода-вывода ещё не выполнена, то возвращаемое значение и действие **aio_return()** не определены.

Ошибки

EINVAL — **aiocbp** не указывает на контрольный блок запроса асинхронного ввода-вывода, значение результата для которого ещё не получено.

ENOSYS — функция **aio_return()** не реализована.

io_cancel() — отменяет ожидающий асинхронный запрос ввода-вывода

```
#include <aio.h>

int aio_cancel(int fd, struct aiocb *aiocbp);
```

Компонуется при указании параметра -lrt.

Функция **aio_cancel()** пытается отменить ожидающие асинхронные запросы ввода-вывода для файлового дескриптора **fd**.

Если значение **aiocbp** равно **NULL**, то отменяются все запросы.

В противном случае, отменяется только запрос, описанный в управляющем блоке, на который указывает значение **aiocbp**.

При отмене запроса посылаются обычные асинхронные уведомления (**aio(7)** и **sigevent(7)**).

Запрос возвращает состояние (**aio_return(3)**) равное **-1**, а состояние ошибки (**aio_error(3)**) устанавливается в **ECANCELED**. Управляющие блоки запросов, которые не могут быть отменены, не изменяются.

Если запрос не может быть отменён, то он завершается как обычно после выполнения операции ввода-вывода (в этом случае **aio_error(3)** вернёт состояние **EINPROGRESSS**).

В случае, если значение **aiocbp** не равно **NULL** и **fd** отличается от файлового дескриптора, для которого создавалась асинхронная операция, то поведение непредсказуемо.

Список операций, которые можно отменять, зависит от реализации.

Возвращаемое значение

Функция **`aio_cancel()`** возвращает одно из следующих значений:

AIO_CANCELED — все запросы успешно отменены.

AIO_NOTCANCELED — как минимум, один указанный запрос не отменён, так как он находится в состоянии выполнения. В этом случае можно проверить состояние каждого запроса с помощью **`aio_error(3)`**.

AIO_ALLDONE — все запросы выполнены ещё до вызова.

`-1` — произошла ошибка. Значение ошибки можно узнать из **`errno`**.

Ошибки

EBADF — значение **`fd`** не является правильным файловым дескриптором.

ENOSYS — функция **`aio_cancel()`** не реализована.

aio_suspend() — ожидает завершения операции в/в или истечения срока

```
#include <aio.h>

int aio_suspend(const struct aiocb * const aiocb_list[],
                int nitems,
                const struct timespec *timeout);
```

Компонуется при указании параметра -lrt.

Функция **aio_suspend( )** приостанавливает выполнение вызывающего потока до тех пор, пока не случиться одно из следующих событий:

- завершится один или более асинхронных запросов ввода-вывода из списка **aiocb_list**;
- будет получен сигнал;
- если значение **timeout** не **NULL** и истечёт указанный срок

```
struct timespec {
    time_t tv_sec;      /* секунды */
    long   tv_nsec;     /* наносекунды [0 .. 999999999] */
};
```

В аргументе **nitems** задаётся количество элементов в **aiocb_list**.

Каждый элемент в списке, который задаётся в **aiocb_list**, должен быть равен **NULL** (игнорируется), или указывать на управляющий блок, который создаётся при создании операции ввода-вывода с помощью **aio_read(3)**, **aio_write(3)** или **lio_listio(3)**.

Если поддерживается **CLOCK_MONOTONIC**, то данный таймер используется для измерения периода ожидания (**clock_gettime(3)**).

Возвращаемое значение

Если функция завершается после выполнения запроса ввода-вывода из списка **aiocb_list**, то возвращается **0**.

В противном случае возвращается **-1**, а значением **errno** определяется тип ошибки.

Ошибки

EAGAIN — истёк период ожидания в вызове до завершения какой-либо операции.

EINTR — вызов завершён по сигналу, возможно по сигналу, который ожидался как завершение операции (**signal(7)**/**SA_RESTART**).

ENOSYS — функция **aio_suspend()** не реализована.

Можно выполнить опрос со значением **timeout**, не равным **NULL**, указав нулевой временной интервал.

Если одна или несколько асинхронных операций ввода-вывода, указанных в **aiocb_list**, уже завершилась на момент вызова **aio_suspend()**, то вызов сразу завершает работу.

Чтобы после успешного возврата из **aio_suspend()** определить, какие операции ввода-вывода завершились, следует использовать **aio_error(3)** для проверки списка структур **aiocb**, на который указывает **aiocb_list**.

Проблемы

Реализация **aio_suspend()** в glibc не является безопасной для асинхронных сигналов, что нарушает требования POSIX.1.

aio_fsync() — асинхронная файловая синхронизация

```
#include <aio.h>

int aio_fsync(int op, struct aiocb *aiocbp); // Компонуется при указании -lrt.
```

Функция **aio_fsync()** выполняет синхронизацию всех ожидающих выполнения асинхронных операций ввода-вывода, связанных с файловым дескриптором **aiocbp->aio_fildes**.

Если значение **op** равно **O_SYNC**, то все операции ввода-вывода в очереди будут выполнены, как если бы сработал вызов **fsync(2)**.

Если **op** равно **O_DSYNC**, то данная функция является асинхронным аналогом **fdatasync(2)**.

Это только запрос — функция не ожидает завершения выполнения ввода-вывода.

Кроме **aio_fildes**, в структуре **aiocbp** используется только поле **aio_sigevent**, в котором указывается желаемый тип асинхронного уведомления по завершению. Остальные игнорируются.

Возвращаемое значение

При успешном выполнении (запрос синхронизации добавлен в очередь) данная функция возвращает **0**. При ошибке возвращается **-1**, и соответствующим образом устанавливается **errno**.

Ошибки

EAGAIN — не хватает ресурсов.

EBADF — значение **aio_fildes** не является правильным файловым дескриптором, открытым для записи.

EINVAL — синхронизированный ввод-вывод не поддерживается для этого файла, или значение op не равно **O_SYNC** или **O_DSYNC**.

ENOSYS — функция **aio_fsync()** не реализована.

Пример

Программа открывает все файлы, указанные в параметрах командной строки и ставит в очередь запрос на полученные файловые дескрипторы с помощью **aio_read(3)**.

Затем программа входит в цикл, в котором периодически следит за всеми выполняемыми операциями ввода-вывода с помощью **aio_error(3)**.

Для каждого запроса ввода-вывода настроено получение уведомления посредством сигнала. После завершения всех запросов ввода-вывода, программа возвращает их состояние с помощью **aio_return(3)**.

Сигнал **SIGQUIT** (генерируемый нажатием **control-**) заставляет программу отменить все невыполненные запросы с помощью **aio_cancel(3)**.

Результат работы программы, которая ставит в очередь два запроса для стандартного ввода, и они отрабатываются двумя введёнными строками «abc» и «x».

```
$ ./prog /dev/stdin /dev/stdin
открыт /dev/stdin в дескрипторе 3
открыт /dev/stdin в дескрипторе 4
aio_error():
    запрос 0 (дескриптор 3): выполняется
    запрос 1 (дескриптор 4): выполняется
abc
Получен сигнал завершения ввода-вывода
aio_error():
    запрос 0 (дескриптор 3): ввод-вывод завершён
    запрос 1 (дескриптор 4): выполняется
aio_error():
    запрос 1 (дескриптор 4): выполняется
x
Получен сигнал завершения ввода-вывода
aio_error():
    запрос 1 (дескриптор 4): ввод-вывод завершён
Завершены все запросы ввода-вывода
aio_return():
    запрос 0 (дескриптор 3): 4
    запрос 1 (дескриптор 4): 2
```

```c
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <aio.h>
#include <signal.h>

#define BUF_SIZE 20      // размер буферов для операций чтения
#define err_exit(msg) do { perror(msg); exit(EXIT_FAILURE); } while (0)
#define err_msg(msg)  do { perror(msg); } while (0)

// определяемая приложением структура для слежения за запросами ввода-вывода
struct io_request {
    int           reqNum;
    int           status;
    struct aiocb *aiocbp;
};

static volatile sig_atomic_t gotSIGQUIT = 0; // флаг получения сигнала SIGQUIT, при
                                             // получении которого мы пытаемся
                                             // отменить все невыполненные запросы
                                             // ввода-вывода
// обработчик сигнала SIGQUIT
static void
quit_handler(int sig) {
    gotSIGQUIT = 1;
}
```

```
#define IO_SIGNAL SIGUSR1   // сигнал, уведомляющий о завершении ввода-вывода

// обработчик завершения ввода-вывода (sigaction())
static void
aioSigHandler(int sig, siginfo_t *si, void *ucontext) {

    if (si->si_code == SI_ASYNCIO) {
        write(STDOUT_FILENO, "Получен сигнал завершения ввода-вывода\n", 31);

        // соответствующая структура io_request в обработчике доступна как
        // struct io_request *io_req = si->si_value.sival_ptr;
        // а файловый дескриптор доступен через io_req->aiocbp->aio_fildes
    }
}

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

    struct io_request *io_req_list;
    struct aiocb      *aiocb_list;
    struct sigaction   sa;
    int    s, j;
    int req_cnt;   // общее количество устанавливаемых в очередь запросов ввода-вывода
    int open_reqs; // количество выполняющихся запросов ввода-вывода

    if (argc < 2) {
        fprintf(stderr, "Использование: %s <имя_файла> <имя_файла>...\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    req_cnt = argc - 1;
```

```
    // выделяем место под массивы

    io_req_list = calloc(req_cnt, sizeof(struct io_request));
    if (io_req_list == NULL)
        err_exit("calloc");

    aiocb_list = calloc(req_cnt, sizeof(struct aiocb));
    if (aiocb_list == NULL)
        err_exit("calloc");

    // устанавливаем обработчик сигнала SIGQUIT
    sa.sa_flags = SA_RESTART; // перезапуск прерванных сигналом вызовов и операций в/в
    sigemptyset(&sa.sa_mask); //

    sa.sa_handler = quit_handler;
    if (sigaction(SIGQUIT, &sa, NULL) == -1)
        err_exit("sigaction");

    // устанавливаем обработчик сигнала завершения ввода-вывода
    sa.sa_flags = SA_RESTART | SA_SIGINFO; // sa_sigaction() вместо sa_handler()
    sa.sa_sigaction = aioSigHandler;
    if (sigaction(IO_SIGNAL, &sa, NULL) == -1)
        err_exit("sigaction");
```

```
    // открываем каждый файл, заданный в командной строке и ставим в
    // очередь запрос на чтение полученного файлового дескриптора
    for (j = 0; j < req_cnt; j++) {
        io_req_list[j].reqNum = j;
        io_req_list[j].status = EINPROGRESS; // Запрос ещё не выполнен
        io_req_list[j].aiocbp = &aiocb_list[j];

        io_req_list[j].aiocbp->aio_fildes = open(argv[j + 1], O_RDONLY);
        if (io_req_list[j].aiocbp->aio_fildes == -1)
            err_exit("open");
        printf("opened %s on descriptor %d\n",
               argv[j + 1],
               io_req_list[j].aiocbp->aio_fildes);

        io_req_list[j].aiocbp->aio_buf = malloc(BUF_SIZE);
        if (io_req_list[j].aiocbp->aio_buf == NULL)
            err_exit("malloc");

        io_req_list[j].aiocbp->aio_nbytes  = BUF_SIZE;
        io_req_list[j].aiocbp->aio_reqprio = 0;
        io_req_list[j].aiocbp->aio_offset  = 0;
        io_req_list[j].aiocbp->aio_sigevent.sigev_notify = SIGEV_SIGNAL;
        io_req_list[j].aiocbp->aio_sigevent.sigev_signo  = IO_SIGNAL;    // SIGUSR1
        io_req_list[j].aiocbp->aio_sigevent.sigev_value.sival_ptr = &io_req_list[j];

        s = aio_read(io_req_list[j].aiocbp);
        if (s == -1)
            err_exit("aio_read");
    }
```

```
    open_reqs = req_cnt;

    // цикл отслеживания состояния запросов ввода-вывода
    while (open_reqs > 0) {
        sleep(3);       // задержка между проверками

        if (gotSIGQUIT) {
            // при получении SIGQUIT пытаемся отменить каждый невыполненный запрос
            // ввода-вывода и показываем состояние, возвращаемое при отмене запроса
            printf("получен SIGQUIT; отмена запросов ввода-вывода: \n");

            for (j = 0; j < req_cnt; j++) {
                if (io_req_list[j].status == EINPROGRESS) {
                    printf("    запрос %d в дескриптор %d:", j,
                            io_req_list[j].aiocbp->aio_fildes);
                    s = aio_cancel(io_req_list[j].aiocbp->aio_fildes,
                                   io_req_list[j].aiocbp);
                    if (s == AIO_CANCELED)
                        printf("ввод-вывод отменён\n");
                    else if (s == AIO_NOTCANCELED)
                        printf("ввод-вывод не отменён\n");
                    else if (s == AIO_ALLDONE)
                        printf("I/O all done\n");
                    else
                        err_msg("aio_cancel");
                }
            }
            gotSIGQUIT = 0;
        }
```

```
        // проверяем состояние каждого запроса в/в, которые ещё не завершились
        printf("aio_error():\n");
        for (j = 0; j < req_cnt; j++) {
            if (io_req_list[j].status == EINPROGRESS) {
                printf("    запрос %d (дескриптор %d): ",
                        j,
                        io_req_list[j].aiocbp->aio_fildes);
                io_req_list[j].status = aio_error(io_req_list[j].aiocbp);

                switch (io_req_list[j].status) {
                case 0:
                    printf("ввод-вывод завершён\n");
                    break;
                case EINPROGRESS:
                    printf("выполняется\n");
                    break;
                case ECANCELED:
                    printf("отменён\n");
                    break;
                default:
                    err_msg("aio_error");
                    break;
                }

                if (io_req_list[j].status != EINPROGRESS)
                    open_reqs--;
            }
        }
    }
    printf("Завершены все запросы ввода-вывода\n");
```

```
    // проверяем возвращаемое состояние всех запросов ввода-вывода

    printf("aio_return():\n");
    for (j = 0; j < req_cnt; j++) {
        ssize_t s;

        s = aio_return(io_req_list[j].aiocbp);
        printf("    запрос %d (дескриптор %d): %zd\n",
                j,
                io_req_list[j].aiocbp->aio_fildes,
                s);
    }

    exit(EXIT_SUCCESS);
}
```

fsync(), fdatasync() — синхронизирует состояние файла в памяти с состоянием на устройстве хранения

```
#include <unistd.h>

int fsync(int fd);      // _BSD_SOURCE || _XOPEN_SOURCE || _POSIX_C_SOURCE >= 200112L
int fdatasync(int fd);  // _POSIX_C_SOURCE >= 199309L || _XOPEN_SOURCE >= 500
```

Вызов **fsync()** пересылает («сбрасывает») все изменённые в памяти (in-core) данные (т. е., изменённые страницы буферного кэша) файла, на который указывает файловый дескриптор **fd**, на дисковое устройство (или другое устройство постоянного хранения). Т.о. вся изменённая информация может быть получена даже после падения системы или внезапной перезагрузки.

При этом выполняется непосредственная запись или сброс дискового кэша (если он есть).

Помимо сброса самих данных, также сбрасывается информация о метаданных, связанная с файлом (inode(7)).

Вызов блокируется до тех пор, пока устройство не сообщит, что пересылка завершена.

fsync() необязательно сбрасывает на диск элемент каталога, содержащий файл – для этого нужно явно выполнить **fsync()** для файлового дескриптора каталога.

Вызов **fdatasync()** подобен **fsync()**, но не записывает изменившиеся метаданные, если эти метаданные не нужны для последующего получения данных. Например, изменения **st_atime** или **st_mtime** (время последнего доступа и последнего изменения, соответственно) не нужно записывать, так как они не требуются для чтения самих данных. С другой стороны, при изменении размера файла (**st_size**, изменяется, например, **ftruncate(2)**) запись метаданных будет нужна.

Целью создания **fdatasync()** является сокращение обменов с диском для приложений, которым не нужна синхронизация метаданных с диском.

В случае успеха данные системные вызовы возвращают ноль.

Если произошла ошибка возвращается -1 и **`errno`** устанавливается должным образом.

Ошибки

EBADF — **`fd`** не является правильным открытым файловым дескриптором.

EIO — во время синхронизации произошла ошибка. Эта ошибка может относится к данным, записанным в какой-то другой файловый дескриптор того же файла.

ENOSPC — при синхронизации закончилось дисковое пространство.

EROFS, **EINVAL** — значение **`fd`** связано со специальным файлом (например, канал, FIFO или сокет), который не поддерживает синхронизацию.

ENOSPC, **EDQUOT** — значение **`fd`** указывает на файл в NFS или другой файловой системе, которая не выделяет место во время системного вызова **`write(2)`**, и какая-то предыдущая операция записи завершилась ошибкой из-за нехватки места.