Системное программное обеспечение вычислительных машин (СПОВМ)

Лекция 07 — Сигналы в UNIX

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, 2021

2022.03.03

Оглавление

sigaction() – расширенная обработка сигналов

Функция **sigaction()** имеет тот же основной эффект, что и **signal()** – указывает, как должен обрабатываться сигнал процессом. Однако **sigaction()** предлагает больший контроль за счет большей сложности. В частности, **sigaction()** позволяет указать дополнительные флаги для управления, когда генерируется сигнал и как вызывается обработчик.

Для определения всей информации о том, как обрабатывать конкретный сигнал, в функции **sigaction()** используются структуры типа **struct sigaction**.

Функция **sigaction()** объявляется следующим образом:

```
#include <signal.h>

int sigaction(int sig,
              const struct sigaction *restrict act,
                    struct sigaction *restrict oldact)
```

Аргумент **act** используется для установки нового действия для сигнала **sig**, а аргумент **oldact** используется для возврата информации о действии, ранее связанном с этим сигналом.

Параметр **oldact** имеет ту же цель, что и возвращаемое значение функции **signal()** – можно проверить, какое старое действие действовало для сигнала, и, если необходимо, восстановить его позже.

Либо **act**, либо **oldact** могут быть нулевым указателем (**NULL**). Если нулевым указателем является **oldact**, это просто подавляет возврат информации о старом действии.

Если нулевым указателем является **act**, действие, связанное с сигналом **sig**, не изменяется.

Такое поведение позволяет узнать, как обрабатывается сигнал, без изменения этой обработки.

Функция `sigaction()` возвращает:

0 в случае успеха и -1 в случае неудачи.

Для этой функции определены следующие условия ошибки **`errno`**:

`EINVAL` — аргумент **`sig`** недействителен, или имеет место попытка перехватить или игнорировать **`SIGKILL`** либо **`SIGSTOP`**.

Структура **`sigaction`** определяется примерно так:

```
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
};
```

Член **`sa_restorer`** не предназначен для использования приложением. (POSIX не определяет **`sa_restorer`**). Некоторые дополнительные сведения о назначении этого поля можно найти в **`sigreturn(2)`**.

Эта структура содержит как минимум члены **`sa_handler`**, **`sa_mask`** и **`sa_flags`**.

sa_handler

```
sighandler_t sa_handler
```

sighandler_t – тип функции-обработчика сигналов

```
#include <signal.h>

typedef void (*sighandler_t)(int); //
sighandler_t signal(int signum, sighandler_t handler);
```

Обработчики сигналов принимают один целочисленный аргумент, определяющий номер сигнала, и имеют тип возвращаемого значения **void**. Поэтому функции-обработчики должны выглядеть так:

```
void handler_function(int signum) { ... }
```

Член **sa_handler** используется так же, как второй аргумент **handler** функции **signal()** – он определяет действие, которое должно быть связано с сигналом, и может быть **SIG_DFL** для действия по умолчанию, **SIG_IGN** для игнорирования этого сигнала или указателем на функцию обработки сигнала. Эта функция получает номер сигнала в качестве единственного аргумента.

sa_mask

```
sigset_t sa_mask
```

Член определяет набор сигналов, которые должны быть заблокированы во время работы обработчика. **Доставленный сигнал автоматически блокируется по умолчанию перед запуском его обработчика независимо от значения параметра sa_mask.**

Если необходимо, чтобы этот сигнал не блокировался в его обработчике, следует в обработчике написать код для его разблокировки.

sigset_t – тип набора сигналов POSIX. Для работы с объектами данного типа используются следующие фукнкции:

```
#include <signal.h>                                    // _POSIX_C_SOURCE

int sigemptyset(sigset_t *set);                        // инициализация набора сигналов
int sigfillset(sigset_t *set);                         // инициализация набора сигналов
int sigaddset(sigset_t *set, int signum);              // добавить сигнал в набор
int sigdelset(sigset_t *set, int signum);              // удалить сигнал из набора
int sigismember(const sigset_t *set, int signum); // проверить присутсвие сигнала
```

sigemptyset() инициализирует набор сигналов **set**, пустым значением, то есть все сигналы исключены из набора.

sigfillset() инициализирует **set** максимальным значением – все сигналы входят в набор.

sigaddset()/sigdelset() добавляет/удаляет сигнал **signum** из **set**.

sigismember() проверяет, является ли **signum** членом набора **set**.

Объекты типа **sigset_t** должны быть инициалированы с помощью **sigemptyset()** или **sigfillset()** до передачи в функции **sigaddset()**, **sigdelset()**, **sigismember()**, или другие дополнительные функции (**sigisemptyset()**, **sigandset()** и **sigorset()**). Если этого не делать, то результат не определён.

Возвращают

При успешном выполнении функции **sigemptyset()**, **sigfillset()**, **sigaddset()** и **sigdelset()** возвращают 0 и -1 при ошибке.

Функция **sigismember()** возвращает:

1, если **signum** является членом набора **set**;

0, если **signum** не является членом;

-1 при ошибке.

При ошибке эти функции изменяют значение **errno** соответствующим образом.

Ошибки

EINVAL — в **signum** задан неправильный сигнал.

sa_flags

```
int sa_flags
```

Член **sa_flags** структуры **sigaction** определяет различные флаги, которые могут влиять на поведение сигнала.

Значение **sa_flags** интерпретируется как битовая маска. Таким образом, флаги, которые необходимо установить, следует указать вместе по ИЛИ и сохранить результат в члене **sa_flags** структуры **sigaction**.

Каждому номеру сигнала соответствует свой набор флагов — каждый вызов **sigaction()** влияет на один конкретный номер сигнала, и указанные в вызове флаги применяются только к этому конкретному сигналу.

В большинстве случаев хорошим значением для этого поля является **SA_RESTART**.

В библиотеке GNU C при установке обработчика с помощью **signal()** все флаги устанавливаются в ноль, за исключением **SA_RESTART**, значение которого зависит от настроек, которые сделаны с помощью **siginterrupt()** (POSIX.1 считает ее устаревшей и рекомендует использовать **SA_RESTART**).

Флаги определены как макросы в заголовочном файле **signal.h**.

SA_NOCLDSTOP

Если **signum** — **SIGCHLD**, не получать уведомления при остановке дочерних процессов (т.е. когда они получают один из **SIGSTOP**, **SIGTSTP**, **SIGTTIN** или **SIGTTOU**) или возобновлении (т.е. когда они получают **SIGCONT**). Флаг имеет значение только при установке обработчика для **SIGCHLD**. По умолчанию **SIGCHLD** доставляется как для завершенных, так и для остановленных дочерних элементов.

SA_NOCLDWAIT

Если **signum** — **SIGCHLD**, не превращать потомков в зомби, когда они завершат исполнение.

Этот флаг имеет смысл только при установке обработчика для **SIGCHLD** или при установке реакции на этот сигнал в **SIG_DFL**.

Если флаг **SA_NOCLDWAIT** выставлен при установке обработчика сигнала **SIGCHLD**, POSIX.1 не указывает, должен ли генерироваться сигнал **SIGCHLD** при завершении дочернего процесса. В Linux в этом случае сигнал **SIGCHLD** генерируется. В некоторых других реализациях это не так.

SA_NODEFER

Не препятствовать получению сигнала из собственного обработчика сигналов. Этот флаг имеет значение только при установке обработчика сигнала. **SA_NOMASK** — устаревший нестандартный синоним этого флага.

SA_ONSTACK

Вызов обработчика сигнала в альтернативном стеке сигналов, предоставленном по запросу функции **sigaltstack(2)**. Если альтернативный стек недоступен, будет использоваться стек по умолчанию. Флаг имеет значение только при установке обработчика сигнала.

Альтернативный стек сигналов используется при обработке сигнала **SIGSEGV**, который возникает при нехватке свободного места в обычном стеке процесса — в этом случае обработчик сигнала **SIGSEGV** не может использовать стек процесса. Если требуется обработка данного сигнала, нужно использовать альтернативный стек сигналов, используя функцию **sigaltstack(2)**.

SA_RESETHAND (SA_ONESHOT)

Восстанавливает действие сигнала по умолчанию при входе в обработчик сигнала.

Этот флаг имеет значение только при установке обработчика сигнала.

SA_RESTART

Обеспечивает поведение, совместимое с семантикой сигналов BSD, путем перезапуска определенных системных вызовов, если во время их выполнения пришел сигнал.

Флаг контролирует, что происходит, когда сигнал доставляется во время определенных примитивов (таких как **open()**, **read()** или **write()**), и обработчик сигнала возвращает управление нормально.

Есть две альтернативы — библиотечная функция может быть возобновлена (resume), либо будет возвращена ошибка с кодом **EINTR**.

Выбор контролируется флагом **SA_RESTART** для конкретного типа доставляемого сигнала.

Если флаг установлен, возврат из обработчика возобновляет библиотечную функцию.

Если флаг снят, возврат из обработчика приводит к отказу функции.

Этот флаг имеет значение только при установке обработчика сигнала.

SA_RESTORER

Не предназначен для использования в приложениях.

SA_SIGINFO

Обработчик сигнала принимает три аргумента, а не один. В этом случае вместо **sa_handler** следует установить **sa_sigaction**. Флаг имеет значение только при установке обработчика.

sa_sigaction – обработчик с тремя параметрами для флага SA_SIGINFO

```
void func(int signo);      // "стандартный" обработчик sa_handler

void func(int signo,       // более продвинутый обработчик sa_sigaction
          siginfo_t *info, //
          void *context);  //
```

Имеет два дополнительных аргумента, которые передаются функции-обработчику сигнала.

Второй аргумент должен указывать на объект типа **siginfo_t**, объясняющий причину, по которой был сгенерирован сигнал.

Третий аргумент может быть приведен к указателю на объект типа **ucontext_t** для ссылки на контекст принимающего потока, который был прерван при доставке сигнала.

В этом случае приложение должно использовать элемент **sa_sigaction** для описания функции перехвата сигнала, и приложение не должно изменять элемент **sa_handler**.

Элемент **si_signo** содержит сгенерированный системой номер сигнала.

Элемент **si_errno** может содержать определяемую реализацией дополнительную информацию об ошибке. Если **si_errno** не равен нулю, он содержит номер ошибки, определяющий условие, вызвавшее генерацию сигнала.

Элемент **si_code** содержит код, идентифицирующий причину сигнала, как описано в Разделе 2.4.3, Действия с сигналом.

Аргумент siginfo_t для SA_SIGINFO обработчика

```
siginfo_t {
    int      si_signo;     // Номер сигнала
    int      si_errno;     // Значение errno
    int      si_code;      // Код сигнала
    int      si_trapno;    // Номер ловушки, которую вызвал аппаратный сигнал
    pid_t    si_pid;       // ID процесса, пославшего сигнал
    uid_t    si_uid;       // ID реального пользователя процесса, пославшего сигнал
    int      si_status;    // Выходное значение или номер сигнала
    clock_t  si_utime;     // Использованное пользовательское время
    clock_t  si_stime;     // Использованное системное время
    sigval_t si_value;     // Значение сигнала
    int      si_int;       // sigqueue(3)
    void    *si_ptr;       // sigqueue(3)
    int      si_overrun;   // Счётчик переполнения таймера
    int      si_timerid;   // ID таймера; таймеры POSIX.1b
    void    *si_addr;      // Адрес памяти, приводящий к ошибке
    long     si_band;      // Внутреннее событие
    int      si_fd;        // Файловый дескриптор
    short    si_addr_lsb;  // Наименее значимый бит адреса
    void    *si_lower;     // Нижняя граница при нарушении адреса
    void    *si_upper;     // Верхняя граница при нарушении адреса
    int      si_pkey;      // Ключа защиты в PTE, который привёл к ошибке
    void    *si_call_addr; // Адрес инструкции системного вызова
    int      si_syscall;   // Количество попыток системного вызова
    unsigned int si_arch;  // Архитектура пытавшегося системного вызова
}
```

Поля **si_signo**, **si_errno** и **si_code** определены для всех сигналов.

В Linux **si_errno** обычно не используется.

Оставшаяся часть структуры может представлять собой объединение, поэтому нужно читать только те поля, которые имеют смысл для заданного сигнала:

- Для сигналов, посылаемых **kill(3)** и **sigqueue(3)**, заполняются **si_pid** и **si_uid**. Также для сигналов, посылаемых **sigqueue(3)**, заполняются **si_int** и **si_ptr** значениями, задаваемыми отправителем сигнала.

- Для сигналов, посылаемых таймерами POSIX.1b (начиная с Linux 2.6), заполняются **si_overrun** и **si_timerid**. Поле **si_timerid** является внутренним идентификатором, который используется ядром для различения таймеров, однако, это не идентификатор таймера, возвращаемого функцией **timer_create(2)**. Поле **si_overrun** отражает счётчик превышения таймера — эту же информацию можно получить с помощью вызова **timer_getoverrun(2)**.

- Для сигналов, посылаемых уведомлением очереди сообщений (см. описание SIGEV_SIGNAL в **mq_notify(3)**), заполняются:

si_int/**si_ptr** значением **sigev_value**, предоставляемым **mq_notify(3)**;

si_pid — значением идентификатора процесса, отправившего сообщение;

si_uid — значением реального идентификатора пользователя, отправившего сообщение.

- Для **SIGCHLD** заполняются ... (не используем, вместо этого пользуемся **wait( )**/**waitpid( )**)

- При **SIGILL**, **SIGFPE**, **SIGSEGV**, **SIGBUS** и **SIGTRAP** заполняется **si_addr** адресом ошибки.

- Для **SIGIO**/**SIGPOLL** (синонимы в Linux) заполняются **si_band** и **si_fd**.

Поле **si_code**

В поле **si_code** аргумента **siginfo_t**, передаваемого обработчику сигналов **SA_SIGINFO** содержится значение (не маска битов), определяющее причину отправки сигнала.

Например, при событии **ptrace(2)** в **si_code** будет содержаться **SIGTRAP** и событие **ptrace** в старшем байте: **(SIGTRAP | PTRACE_EVENT_foo << 8)**.

Для обычного сигнала ниже приведены значения, которые могут быть в **si_code** и причина возникновения сигнала:

SI_USER	послан **kill(2)**
SI_KERNEL	послан ядром
SI_QUEUE	послан **sigqueue(3)**
SI_TIMER	таймер POSIX истёк.
SI_MESGQ	изменилось состояние очереди сообщений POSIX (**mq_notify(3)**)
SI_ASYNCIO	AIO завершён.
SI_SIGIO	SIGIO/SIGPOLL заполняют si_code как описано выше.
SI_TKILL	**tkill(2)** или **tgkill(2)**

Возвращаемое значение

При успешном выполнении **sigaction()** возвращается 0;
при ошибке возвращается -1, а в **errno** содержится код ошибки.

Ошибки (см. man sigaction(2))

Замечания

Потомок, созданный с помощью fork(2), наследует реакцию на сигналы от своего родителя.
При execve(2) реакция на сигналы устанавливается в значение по умолчанию.
Реакция на игнорируемые сигналы не изменяется.

Блокирование сигналов для обработчика

Когда вызывается обработчик сигнала, желательно, чтобы он мог закончить свою работу, не прерываясь другим сигналом. С момента, как обработчик начинает выполнение, до момента его завершения, необходимо блокировать все сигналы, которые могут помешать его работе или испортить его данные.

Когда для сигнала вызывается функция-обработчик, этот сигнал в дополнение к любым другим сигналам, которые находятся в маске сигналов процесса, на время работы обработчика автоматически блокируется. Например, если установлен обработчик для «SIGTSTP», то прибытие этого сигнала заставляет дальнейшие сигналы «SIGTSTP» ожидать во время выполнения обработчика.

Однако другие виды сигналов по умолчанию не блокируются, поэтому они могут поступать во время выполнения обработчика.

Надежный способ заблокировать другие виды сигналов во время выполнения обработчика — использовать элемент «**sa_mask**» структуры «**sigaction**».

Это более надежно, чем явное блокирование других сигналов в коде обработчика – если сигналы блокируются в обработчике явно, невозможно избежать хотя бы короткого интервала в начале этого обработчика, когда они еще не заблокированы.

Удалить сигналы из текущей маски процесса, используя этот механизм, невозможно. Однако внутри обработчика можно вызывать **sigprocmask()**, чтобы блокировать или разблокировать сигналы по своему усмотрению. В любом случае, когда обработчик возвращается, система восстанавливает маску, которая была до входа в обработчик. Если какие-либо сигналы, разблокированные этим восстановлением, ожидают обработки, процесс немедленно получит эти сигналы, прежде чем вернуться к коду, который был прерван.

Пример блокирования

```
#include <signal.h>
#include <stddef.h>

void catch_stop( );           // перехватчик сигнала SIGSTP

void install_handler(void) {

    struct sigaction setup_action;
    sigset_t         block_mask;   // объявление набора сигналов
    sigemptyset(&block_mask);      // и его инициализация как пустого

    // блокируем другие сигналы, генерируемые терминалом на время обработки
    sigaddset(&block_mask, SIGINT);
    sigaddset(&block_mask, SIGQUIT);

    // заполняем объект типа struct sigaction
    setup_action.sa_handler = catch_stop; // "классический" обработчик
    setup_action.sa_mask    = block_mask; // копируем маску блокировки сигналов
    setup_action.sa_flags   = 0;          // флаги не нужны

    sigaction(SIGTSTP,          // сигнал
              &setup_action,    // обработчик
              NULL);            // старый обработчик не нужен
}
```

Взаимодействие функций signal() и sigaction()

В одной программе можно использовать обе функции **signal()** и **sigaction()**, но следует быть осторожным, потому что они могут взаимодействовать несколько странным образом.

Функция **sigaction()** определяет больше информации, чем функция **signal()**, поэтому возвращаемое значение **signal()** не может выражать весь диапазон возможностей **sigaction()**. Следовательно, если используется **signal()** для сохранения и последующего восстановления действия, возможно, не удастся восстановить должным образом обработчик, который был установлен с помощью **sigaction()**.

Чтобы избежать возникновения проблем, для сохранения и восстановления обработчика всегда следует использовать **sigaction()**, если ваша программа вообще использует **sigaction()**.

Поскольку **sigaction()** является более общим, он может должным образом сохранять и восстанавливать любое действие, независимо от того, было ли оно установлено изначально с помощью **signal()** или **sigaction()**.

В некоторых системах, если действие установлено с помощью **signal()**, а затем оно проверяется с помощью **sigaction()**, полученный адрес обработчика может отличаться от того, который был указан с помощью **signal()**. Адрес может даже не подходить для использования в качестве аргумента действия с **signal()**.

Данная проблема никогда не возникает в системах GNU. Тем не менее, лучше всего использовать тот или иной механизм последовательно в рамках одной программы.

Примечание о переносимости:

Основная функция **signal()** является стандартной функцией C, а **sigaction()** — частью стандарта POSIX.1. Если есть необходимость переносимости в системы, отличные от POSIX, следует вместо **sigaction()** использовать функцию **signal()**.

Пример функции `sigaction()`

В разделе «Базовая обработка сигналов» приведен пример создания простого обработчика сигналов завершения с использованием функции **`signal()`**.

Эквивалентный пример с использованием функции **`sigaction()`** представлен ниже:

```
#include <signal.h>

void termination_handler(int signum) {

  struct temp_file *p;

  for (p = temp_file_list; p; p = p->next)
    unlink(p->name);
}

int main (void) {
  ...
  struct sigaction new_action, old_action;

  // Укажем новое действие как обработчик
  new_action.sa_handler = termination_handler;
  sigemptyset(&new_action.sa_mask);       // инициализация набора сигналов
  new_action.sa_flags = 0;                // чистим флаги (простой обработчик)
```

```
  sigaction(SIGINT, NULL, &old_action);   // состояние текущего обработчика
  if (old_action.sa_handler != SIG_IGN)
      sigaction(SIGINT, &new_action, NULL); // установка нового если не SIG_IGN
  sigaction(SIGHUP, NULL, &old_action);
  if (old_action.sa_handler != SIG_IGN)
      sigaction(SIGHUP, &new_action, NULL);
  sigaction (SIGTERM, NULL, &old_action);
  if (old_action.sa_handler != SIG_IGN)
      sigaction(SIGTERM, &new_action, NULL);
  ...
}
```

Программа загружает структуру **new_action** с желаемыми параметрами и передает ее в вызов **sigaction()**.

Как и в примере с использованием **signal()**, код избегает обработки сигналов, ранее настроенных на игнорирование.

Используя функцию **sigaction()**, которая позволяет проверять текущее действие, не указывая новое, можно избежать изменения обработчика сигнала даже на мгновение.

Другой пример.
Извлечение информации о текущем действии для **SIGINT**, не изменяя это действие.

```
struct sigaction query_action;

if (sigaction(SIGINT, NULL, &query_action) < 0) { // прерывание с клавиатуры
  /* ‘sigaction’ возвратит -1 в случае ошибки */
} else if (query_action.sa_handler == SIG_DFL) {
  /* ‘SIGINT’ обрабатывается по умолчанию, фатальным для программы образом */
} else if (query_action.sa_handler == SIG_IGN) {
  /* ‘SIGINT’ игнорируется.  */
} else {
  /* Установлен свой обработчик */
}
```

Первоначальные сигнальные действия

Когда создается новый процесс (**fork( )**), он наследует обработку сигналов от своего родительского процесса. Однако, когда загружается новый образ процесса с помощью функции **exec( )**, обработка любых сигналов, для которых были определены собственные обработчики, возвращается к обработке по умолчанию **SIG_DFL**.

Это имеет явный смысл — функции-обработчики из старой программы специфичны для этой программы и не присутствуют в адресном пространстве нового образа программы.

Собственно, новая программа может установить свои собственные обработчики.

Когда программа запускается оболочкой, оболочка обычно устанавливает начальные действия для дочернего процесса на **SIG_DFL** или **SIG_IGN**, в зависимости от ситуации.

Перед установкой собственных обработчиков сигналов рекомендуется убедиться, что оболочка не настроила начальное действие SIG_IGN.

Ниже пример того, как установить обработчик для **SIGHUP**, но не в том случае, если **SIGHUP** в настоящее время игнорируется:

```
...
struct sigaction temp;

sigaction(SIGHUP, NULL, &temp);    // узнаем реакцию на сигнал

if (temp.sa_handler != SIG_IGN) {  // Если не устанвлено игнорирование
  temp.sa_handler = handle_sighup; // прописываем свой обработчик в struct sigaction
  sigemptyset(&temp.sa_mask);      // инициализируем набор сигналов
  sigaction(SIGHUP, &temp, NULL);  // и регистрируем обработчик
}
```

Определение обработчиков сигналов

Рассмотрим, как написать функцию-обработчик сигнала, которая может быть установлена с помощью функций **`signal()`** или **`sigaction()`**.

Обработчик сигнала — это просто функция, которую компилируется вместе с остальной частью программы.

Вместо прямого вызова функции используется **`signal()`** или **`sigaction()`**, чтобы сообщить операционной системе, что она должна вызывать ее при поступлении сигнала.

Эта процедура известна как «установка» или «регистрация» обработчика.

В функциях-обработчиках сигналов можно использовать две основные стратегии:

1) можно заставить функцию-обработчик отметить, что сигнал прибыл, изменив некоторые глобальные структуры данных, после чего выполнить возврат в обычном режиме.

2) можно заставить функцию-обработчик завершить программу или передать управление в точку, где она сможет восстановиться из ситуации, вызвавшей сигнал.

Следует проявлять особую осторожность при написании функций-обработчиков, потому что они могут вызываться асинхронно. То есть обработчик может быть вызван в любой момент программы непредсказуемо. Если два сигнала поступают в течение очень короткого интервала, один обработчик может работать внутри другого.

Итак, что должен делать обработчик и чего следует избегать.

Обработчики сигналов, которые возвращают управление

Обработчики, которые возвращают управление, обычно используются для таких сигналов, как **SIGALRM**, а также для сигналов ввода-вывода и межпроцессного взаимодействия.

Обработчик для **SIGINT** может также нормально возвратить управление после установки флага, который сообщает программе о завершении в удобное для нее время.

Нормальный возврат из обработчика сигнала программной ошибки небезопасен, поскольку поведение программы при возврате из функции-обработчика после программной ошибки будет не определено.

Обработчики, которые обычно возвращают управление, должны изменить некоторую глобальную переменную, чтобы был какой-либо эффект. Обычно это переменная, которая периодически проверяется программой во время нормальной работы. Ее тип данных должен быть **sig_atomic_t** по причинам, описанным в разделе «Atomic Data Access — Доступ к атомарным данным».

Ниже простой пример такой программы. Он выполняет тело цикла до тех пор, пока не заметит, что поступил сигнал **SIGALRM**. Этот метод полезен, потому что он позволяет выполнять итерации, до того момента, пока не поступит сигнал выхода из цикла.

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

volatile sig_atomic_t keep_going = 1; // Флаг управляет завершением основного цикла

// Обработчик сигнала просто сбрасывает флаг, снова себя включает и выходит
void catch_alarm(int sig) {
  keep_going = 0;
  signal(sig, catch_alarm);
}

void do_stuff(void) {
  puts("Делаем что-то в ожидании сигнала будильника ...");
}

int main (void) {

  signal(SIGALRM, catch_alarm); // Устанавливаем обработчик сигналов SIGALRM

  alarm(2);     // Устанавливаем будильник, чтобы он сработал через некоторое время

  // Время от времени проверяем флаг, чтобы узнать, когда нужно выйти
  while (keep_going) {
    do_stuff();
  }
  return EXIT_SUCCESS;
}
```

Обработчики, завершающие процесс

Функции-обработчики, завершающие программу, обычно используются для упорядоченной очистки или восстановления после сигналов ошибок программы и интерактивных прерываний.

Самый простой способ для обработчика завершить процесс — это поднять тот же сигнал, которым с самого начала был запущен обработчик. Это можно сделать следующим образом:

```
volatile sig_atomic_t fatal_error_in_progress = 0;

void fatal_error_signal(int sig) {

  // Поскольку этот обработчик устанавливается для более чем одного вида сигналов,
  // он все равно может быть вызван рекурсивно каким-либо другим сигналом.
  // Чтобы отслеживать это, можно использовать статическую переменную.
  if (fatal_error_in_progress) {
     raise(sig);
  }
  fatal_error_in_progress = 1;

  // Теперь выполним действия по очистке: сбросим режимы терминала, покиляем
  // процессы-потомки, удалим залоченные файлы
  ...
  // Теперь поднимем сигнал. Мы повторно активируем обработку сигнала по умолчанию,
  // которая должна завершить процесс. Мы могли бы просто вызвать exit() или abort(), но
  // повторное поднятие сигнала правильно устанавливает статус возврата  из из процесса.
  signal(sig, SIG_DFL);
  raise(sig);
}
```

Доступ к атомарным данным и обработка сигналов

Независимо от того, являются ли данные в приложении атомарными, или это просто текст, следует быть осторожным в отношении того факта, что доступ к одному элементу данных не обязательно является «*атомарным*» — это означает, что для чтения или записи одного объекта может потребоваться более одной инструкции. В таких случаях обработчик сигнала может быть вызван в середине чтения или записи объекта.

Есть три способа справиться с этой проблемой.

- можно использовать типы данных, доступ к которым всегда выполнется атомарно;
- можно тщательно позаботиться о том, чтобы ничего плохого не произошло, если доступ был прерван;
- можно заблокировать кругом все сигналы для любого доступа, который лучше не прерывать (Блокирование сигналов).

Ниже приведен пример, который показывает, что может произойти, если обработчик сигнала запускается во время изменения переменной.

Прерывание чтения переменной также может привести к парадоксальным результатам, но в примере показана только запись.

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

volatile struct two_words {
  int a;
  int b;
} memory;

void handler(int signum) {
   printf ("%d,%d\n", memory.a, memory.b);
   alarm(1);
}

int main (void) {

   static struct two_words zeros = { 0, 0 }, ones = { 1, 1 };
   signal(SIGALRM, handler);
   memory = zeros;
   alarm(1);
   while (1) {
       memory = zeros;
       memory = ones;
   }
}
```

Эта программа заполняет «память» нулями, единицами, нулями, единицами, чередующимися бесконечно.

Между тем, один раз в секунду обработчик сигнала будильника распечатывает текущее содержимое. Вызов **`printf()`** в обработчике безопасен в этой программе, потому что он определенно не вызывается вне обработчика, когда возникает сигнал.

Понятно, что эта программа может печатать пару нулей или пару единиц. Но это еще не все! На большинстве машин требуется несколько инструкций для сохранения нового значения в памяти, и значение сохраняется по одному слову за раз. Если сигнал доставляется между этими инструкциями, обработчик может обнаружить, что **`memory.a`** равен нулю, а **`memory.b`** равен единице (или наоборот).

На некоторых машинах можно сохранить новое значение в памяти с помощью только одной инструкции, которую нельзя прерывать. На этих машинах обработчик всегда будет печатать два нуля или две единицы.

Атомарные типы

Чтобы избежать неопределенности в отношении прерывания доступа к переменной, можно использовать определенный тип данных, для которого доступ всегда атомарный:

`sig_atomic_t`

Чтение и запись этого типа данных гарантированно происходит как бы с помощью одной инструкции, поэтому обработчик не может запуститься «в середине» доступа.

Тип **`sig_atomic_t`** всегда является целочисленным типом данных, но его тип и количество битов, которые он содержит, может варьироваться от машины к машине.

На практике можно допустить, что **`int`** является атомарным.

Также можно допустить, что типы указателей тоже являются атомарными.

Оба эти допущения верны на всех машинах, которые поддерживает библиотека GNU C, и на всех известных системах POSIX.

kill() — послать сигнал процессу

```
#include <sys/types.h>          // _POSIX_C_SOURCE
#include <signal.h>

int kill(pid_t pid, int sig);
```

Системный вызов **kill()** может быть использован для посылки какого-либо сигнала какому-либо процессу или группе процессов.

Если значение **pid** является положительным, то сигнал **sig** посылается процессу с идентификатором **pid**.

Если значение **pid** равно 0, то **sig** посылается каждому процессу, который входит в группу вызывающего процесса.

Если значение **pid** равно -1, то **sig** посылается каждому процессу, которым вызывающий процесс имеет право отправлять сигналы, за исключением процесса с номером 1 (init).

Если значение **pid** меньше **-1**, то **sig** посылается каждому процессу, который входит в группу процессов, чей ID равен **-pid**.

Если значение **sig** равно 0, то никакой сигнал не посылается, но выполняется проверка существования и права — это можно использовать для проверки существования процесса или группы процессов с заданным ID и допустимости отправки сигнала вызывающим.

Чтобы процесс мог посылать сигнал, он должен быть привилегированным, либо реальный или эффективный идентификатор пользователя посылающего процесса должен быть равен реальному или сохранённому идентификатору пользователя процесса, которому отправляется сигнал.

Для сигнала **SIGCONT** посылающий и получающий процессы должны принадлежать одному сеансу.

Возвращаемое значение

При успешном выполнении (по крайней мере, был послан один сигнал) возвращается 0.
В случае ошибки возвращается -1, а **`errno`** устанавливается в соответствующее значение.

Ошибки

EINVAL — Указан некорректный сигнал.
EPERM — Вызывающий процесс не имеет достаточно прав для отправки сигнала ни одному из группы процессов-получателей.
ESRCH — Процесс-получатель или группа процессов не существует. Надо заметить, что существующий процесс может быть в состоянии зомби — процесс, завершивший выполнение, но которого ещё не дождались с помощью **`wait(2)`**.

Замечания
Процессу **`init`** с идентификатором 1 можно послать только те сигналы, для которых он явно установил обработчики сигналов.
Так сделано, чтобы быть уверенным, что в случае какой-либо нештатной ситуации работа системы не будет завершена аварийно.
Детальное поведение **`kill()`** для разных реализаций обычно описано в руководстве (man) для реализации.

sigqueue() - вставляет сигнал и данные в очередь процесса

```
#include <signal.h>                          // _POSIX_C_SOURCE >= 199309L

int sigqueue(pid_t pid,
             int sig,
             const union sigval value);
```

Вызов **sigqueue()** отправляет сигнал, указанный в **sig**, процессу с идентификатором **PID**, определённом в **pid**.

Для отправки сигнала требуются определённые права, такие же как для **kill(2)**.

Как и в случае с **kill(2)** для проверки того, что заданный PID вообще существует, может использоваться пустой сигнал (0).

Аргумент **value** используется для указания сопутствующего элемента данных (либо целого, либо указателя), отправляемых сигналу, и имеет следующий тип:

```
union sigval {
    int   sival_int;
    void *sival_ptr;
};
```

Если у процесса, принимающего сигнал, установлен обработчик посредством **sigaction(2)** с флагом **SA_SIGINFO**, то он может получить эти данные через поле **si_value** структуры **siginfo_t**, передаваемой как второй аргумент для обработчика.

Кроме этого, значение поля **si_code** этой структуры будет установлено в **SI_QUEUE**.

При успешном выполнении **sigqueue()** возвращается 0, что означает, что сигнал попал в очередь принимающего процесса. При ошибке возвращается -1 и в **errno** содержится код ошибки.

Реализация в Linux

В Linux **sigqueue()** реализована через системный вызов **rt_sigqueueinfo(2)** — низкоуровневый интерфейс для отправки сигнала с данными процессу или нити.

Приёмник сигнала может получить сопутствующие данные, установив обработчик сигнала с помощью **sigaction(2)** с флагом **SA_SIGINFO**.

```
int rt_sigqueueinfo(pid_t tgid,       // группа нитей (процесс)
                    int sig,          // сигнал
                    siginfo_t *info); //
```

Данный системный вызов отличается от **sigqueue()** третьим аргументом — структурой **siginfo_t**, которая будет предоставляться обработчику сигнала принимающего процесса или возвращаться вызовом **sigtimedwait(2)** из принимающего процесса.

В обёрточной функции glibc **sigqueue()** этот аргумент, **info**, инициализируется следующим образом:

```
uinfo.si_signo = sig;      // аргумент, передаваемый в sigqueue()
uinfo.si_code  = SI_QUEUE;
uinfo.si_pid   = getpid(); // ID процесса отправителя
uinfo.si_uid   = getuid(); // реальный UID отправителя
uinfo.si_value = val;      // аргумент, передаваемый в sigqueue()
```