ОПЕРАЦИОННЫЕ СИСТЕМЫ И СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лекция 06 — Сигналы в UNIX

Преподаватель: Поденок Леонид Петрович
prep@lsi.bas-net.by
БГУИР 505а 5 к. +375 17 293 8039
ОИПИ, 116/118 +375 17 320 7402

БГУИР 2023

Оглавление

Сигналы в UNIX

С точки зрения пользователя получение процессом сигнала выглядит как возникновение прерывания.

Процесс прекращает свое регулярное исполнение, и управление передается механизму обработки сигнала. По окончании обработки сигнала процесс может возобновить регулярное исполнение.

Типы сигналов (их принято задавать номерами, как правило, в диапазоне от 1 до 31 включительно или специальными символьными обозначениями) и способы их возникновения в системе жестко регламентированы.

Процесс может получить сигнал от:

1) hardware (при возникновении исключительной ситуации);
2) другого процесса, выполнившего системный вызов передачи сигнала;
3) операционной системы (при наступлении некоторых событий);
4) терминала (при нажатии определенной комбинации клавиш);
5) системы управления заданиями при выполнении команды **`kill`**.

Передачу сигналов процессу в случаях его генерации источниками 2, 3 и 5, т.е. в конечном счете каким-либо другим процессом, можно рассматривать как реализацию в UNIX *сигнальных средств связи*.

Существует три варианта реакции процесса на сигнал:

1) принудительно проигнорировать сигнал;

2) произвести обработку по умолчанию (проигнорировать, остановить процесс (перевести в состояние ожидания до получения другого специального сигнала), либо завершить работу с образование core файла или без него);

3) выполнить обработку сигнала, указанную пользователем.

Изменение реакции процесса на сигнал

Изменить реакцию процесса на сигнал можно с помощью специальных системных вызовов.

Реакция на некоторые сигналы не допускает изменения и они могут быть обработаны только по умолчанию. Так, например, сигнал с номером 9 (**`SIGKILL`**) обрабатывается только по умолчанию и всегда приводит к завершению процесса.

Сохранение реакции на сигнал

Важным вопросом при программировании с использованием сигналов является вопрос о сохранении реакции на них при рождении нового процесса или замене его пользовательского контекста.

При системном вызове **`fork(2)`** все установленные реакции на сигналы наследуется порожденным процессом.

При системном вызове **`exec(2)`** сохраняются реакции только для тех сигналов, которые игнорировались или обрабатывались по умолчанию. Получение любого сигнала, который до вызова **`exec(2)`** обрабатывался пользователем, приведет к завершению процесса.

Обработка сигналов

«Сигнал» — это программное прерывание, доставляемое процессу.

Операционная система использует сигналы, чтобы сообщать об исключительных ситуациях исполняющейся программе.

Определено множество типов сигналов, каждый для определенного типа события.

Некоторые виды событий делают нецелесообразным или невозможным выполнение программы в обычном режиме, и соответствующие сигналы обычно прерывают выполнение программы.

Другие виды сигналов, сообщающих о безобидных событиях, по умолчанию игнорируются.

Если приложению необходимо ожидать событие, которое вызывает сигналы, ему следует определить функцию-обработчик и указать операционной системе, что она должна запускаться при поступлении сигнала определенного типа.

Также, один процесс может послать сигнал другому процессу — это позволяет родительскому процессу прервать дочерний процесс или позволить сообщаться двум процессам и координироваться.

Основные понятия сигналов

Некоторые примеры сигналов

Некоторые из событий, которые могут вызывать (или «генерировать», или «поднимать») сигнал:

- Ошибка программы, такая как деление на ноль или выдача адреса за пределами допустимого диапазона;

- Запрос пользователя на прерывание или завершение программы. Большинство сред исполнения настроены так, чтобы пользователь мог приостановить программу, набрав «Ctrl-z», или завершить ее нажатием «Ctrl-c». Какая бы последовательность клавиш не использовалась, операционная система посылает соответствующий сигнал для прерывания процесса.

- Завершение дочернего процесса.

- Истечение таймера или срабатывание будильника.

- Вызов **`kill`** или **`raise`** из того же процесса.

- Вызов **`kill`** или **`raise`** из другого процесса. Сигналы — это ограниченные, но полезные формы межпроцессного взаимодействия.

- Попытка выполнить операцию ввода-вывода, которая не может быть выполнена. Например, попытка чтения из канала, у которого нет записывающего (15[1] Pipes and FIFOs), а также чтение или запись в терминал в определенных ситуациях (28 Job Control).

Каждый из этих типов событий (за исключением явных вызовов **`kill`** и **`raise`**) генерирует свой особый вид сигнала.

Различные типы сигналов перечислены и подробно описаны в (24.2 Standard Signals).

---

1) Пункт info

Концепции генерации сигналов

События, которые генерируют сигналы, делятся на три основные категории:
- ошибки (errors);
- внешние события (external events);
- явные запросы (explicit requests).

Ошибка означает, что программа сделала что-то недопустимое и не может продолжить выполнение.

Не все виды ошибок генерируют сигналы — на самом деле, большинство из них ничего не генерирует. Например, открытие несуществующего файла является ошибкой, но не вызывает сигнала. Вместо этого функция/вызов **open** возвращает -1.

Как правило, об ошибках, которые обязательно связаны с определенными библиотечными функциями, сообщается путем возврата значения, указывающего на ошибку.

Ошибки, которые вызывают сигналы, могут возникать в любом месте программы, а не только при вызовах библиотеки. К ним относятся деление на ноль и недопустимые адреса памяти.

Внешнее событие обычно связано с вводом-выводом (I/O) или другими процессами. К ним относятся поступление ввода, истечение таймера и завершение дочернего процесса.

Явный запрос означает использование библиотечной функции, такой как **`kill`**, специально предназначенной для генерации сигнала.

Сигналы могут генерироваться «синхронно» или «асинхронно».

Синхронный сигнал относится к определенному действию в программе и доставляется (если не заблокирован) во время этого действия.

Большинство ошибок генерируют сигналы синхронно, как и явные запросы процесса на генерацию сигнала для того же процесса. На некоторых машинах определенные виды аппаратных ошибок (обычно исключения с плавающей запятой) не выдаются полностью синхронно, тем не менее, они могут появиться через несколько инструкций.

Асинхронные сигналы генерируются событиями, не зависящими от процесса, который их получает. Эти сигналы поступают в непредсказуемое время во время выполнения. Внешние события генерируют сигналы асинхронно, как и явные запросы, которые вызываются в отношении какого-либо другого процесса.

Определенный тип сигнала обычно либо синхронный, либо обычно асинхронный. Например, сигналы об ошибках обычно синхронны, потому что ошибки генерируют сигналы синхронно. Но любой тип сигнала может быть сгенерирован синхронно или асинхронно с помощью явного запроса.

Как доставляются сигналы

Когда сигнал генерируется, он становится «ожидающим, незавершенным». Обычно он остается незавершенным в течение короткого периода времени, а затем «доставляется» тому процессу, которому был послан. Однако, если такой сигнал в данный момент времени «заблокирован», он может оставаться в состоянии ожидания неопределенно долго - до тех пор, пока такие сигналы не будут «разблокированы». После разблокировки он будет доставлен немедленно (24.7 Blocking Signals).

Как только сигнал доставляется, сразу или после долгой задержки, выполняется «указанное действие» для этого сигнала. Для определенных сигналов, таких как «**SIGKILL**» и «**SIGSTOP**», действие фиксировано, но для большинства сигналов у программы есть выбор, что делать при получении данного вида сигнала. Это:

- игнорировать сигнал;
- указать «функцию-обработчик»;
- принять «действие по умолчанию».

Программа указывает свой выбор с помощью таких функций, как **`signal()`** или **`sigaction()`** (24.3 Specifying Signal Actions). Иногда мы говорим, что обработчик «ловит» сигнал. Во время работы обработчика сигнал данного типа обычно блокируется.

Если заданное действие для определенного типа сигнала заключается в его игнорировании, то любой такой сигнал, который генерируется, немедленно отбрасывается. Это происходит, даже если сигнал в это время заблокирован. Сигнал, отброшенный таким образом, никогда не будет доставлен, даже если программа впоследствии укажет другое действие для этого типа сигнала, а затем разблокирует его.

Если поступает сигнал, который программа и не обрабатывает, и не игнорирует, для сигнала выполняется «действие по умолчанию». Каждый вид сигнала имеет собственное действие по умолчанию, указанное ниже (24.2 Standard Signals).

Для большинства сигналов действие по умолчанию — завершить процесс. Для определенных типов сигналов, которые представляют «безобидные» события, действие по умолчанию — ничего не делать.

Когда сигнал завершает процесс, его родительский процесс может определить причину завершения, исследуя код состояния завершения, сообщаемый функциями **`wait()`** или **`waitpid()`**[2]. Информация, которую он может получить, включает в себя тот факт, что завершение произошло из-за сигнала, и тип задействованного сигнала. Если программа, которую вы запускаете из оболочки, завершается сигналом, оболочка обычно выводит какое-то сообщение об ошибке.

Сигналы, которые обычно представляют собой программные ошибки, обладают особым свойством — когда один из этих сигналов завершает процесс, он также записывает «файл дампа ядра», в котором записывается состояние процесса на момент завершения. Дамп ядра можно изучить с помощью отладчика и выяснить, что вызвало ошибку.

Если вы сигнал «программная ошибка (program error)» вызывается явным запросом, и он завершает процесс, файл дампа ядра создается так же, как если бы сигнал был вызван непосредственно ошибкой.

---

2) Это обсуждается более подробно в разделе «Завершение процесса (26.6 Process Completion)»

Стандартные сигналы

Стандартные сигналы — особые виды сигналов со стандартными названиями и значениями.

В этом разделе перечислены названия различных стандартных типов сигналов и описано, какие события они означают. Каждое имя сигнала — это макрос, обозначающий положительное целое число — «номер сигнала» для данного типа сигнала.

Программы никогда не должны делать никаких предположений о числовом коде для определенного вида сигнала, а вместо этого всегда обращаться к ним по именам, определенным ниже. Это связано с тем, что номер для данного вида сигнала может варьироваться от системы к системе, но значения названий стандартизированы и довольно единообразны.

Имена сигналов определены в заголовочном файле **`signal.h`**.

`Макрос: int NSIG`

Значение этой символьной константы — общее количество определенных сигналов. Поскольку номера сигналов назначаются последовательно, «**`NSIG`**» также на единицу больше, чем наибольший определенный номер сигнала.

Типы сигналов

- сигналы программных ошибок;
- сигналы завершения;
- аварийные сигналы (Alarm Signals);
- асинхронные сигналы ввода/вывода;
- сигналы управления заданиями;
- сигналы ошибок обслуживания;
- прочие сигналы.

Сигналы программных ошибок

Сигналы ошибок программы используются для сообщения о серьезных ошибках программы. Действие по умолчанию для всех этих сигналов — завершить процесс.

Они генерируются при обнаружении серьезной программной ошибки операционной системой или самим компьютером. В общем, все эти сигналы указывают на то, что ваша программа каким-то образом серьезно нарушена, и обычно нет возможности продолжить вычисления, в которых возникла ошибка.

Некоторые программы обрабатывают сигналы программных ошибок, чтобы привести в порядок среду перед завершением работы. Например, программы, которые отключают эхо ввода терминала, чтобы снова включить эхо, должны обрабатывать сигналы программных ошибок.

Обработчик сигнала программных ошибок должен завершиться указанием действия по умолчанию для произошедшего сигнала и его повторным вызовом — это приведет к тому, что программа завершится с этим сигналом, как если бы у нее не было обработчика.

Прекращение работы — это разумное завершение в случае программной ошибки в большинстве программ. Однако некоторым программным системам, таким, например, как Lisp, которые могут загружать скомпилированные пользовательские программы, может потребоваться продолжать выполнение, даже если пользовательская программа вызывает ошибку. У этих программ есть обработчики, которые используют **`longjmp()`** для возврата управления на командный уровень.

Если заблокировать или проигнорировать эти сигналы, либо установить для них обработчики, которые будут возвращатся нормально, как будто ничего не произошло, программа сломается как только такие сигналы появятся. Конечно, если только они не будут сгенерированы с помощью **`raise()`** или **`kill()`** вместо реальной ошибки.

Когда один из таких сигналов программной ошибки завершает процесс, он также записывает «файл дампа памяти», в котором записывается состояние процесса на момент завершения.

Файл дампа ядра называется «core» и записывается в тот каталог, который в данный момент используется в процессе. (В некоторых системах можно указать имя файла для дампа ядра с помощью переменной среды **COREFILE**.) Файлы дампа ядра предназначены для того, чтобы можно было исследовать их с помощью отладчика и выяснить, что вызвало ошибку.

SIGFPE – фатальная арифметическая ошибка;
SIGILL – неверная инструкция;
SIGSEGV – нарушение доступа к памяти;
SIGBUS – доступ к недопустимому адресу (неправильное выравнивание указателя);
SIGABRT – программа обнаружила ошибку и вызвала abort();. Программа прерывается.
SIGIOT – другое название SIGABRT;
SIGTRAP – точка останова машины;
SIGEMT – ловушка эмулятора;
SIGSYS – плохой системный вызов.

SIGFPE — сообщить о фатальной арифметической ошибке

Хотя название происходит от «исключения с плавающей запятой», этот сигнал фактически охватывает все арифметические ошибки, включая деление на ноль и переполнение. Если программа хранит целочисленные данные в месте, которое затем используется в операции с плавающей запятой, это часто вызывает исключение «недопустимая операция», поскольку процессор не может распознать данные как число с плавающей запятой.

Фактические исключения с плавающей запятой — сложная тема, потому что существует много типов исключений с слегка разными значениями, и сигнал **SIGFPE** не различает их.

Стандарт IEEE для двоичной арифметики с плавающей запятой (ANSI / IEEE Std 754-2017) определяет различные исключения с плавающей запятой и требует, чтобы соответствующие компьютерные системы сообщали об их возникновении. Однако этот стандарт не определяет, как сообщается об исключениях или какие виды обработки и контроля операционная система может предложить программисту.

Системы BSD предоставляют обработчику **SIGFPE** дополнительный аргумент, который различает различные причины исключения. Чтобы получить доступ к этому аргументу, следует определить обработчик для приема двух аргументов, что означает, что для того, чтобы установить такой обработчик, необходимо привести его к типу функции с одним аргументом.

Библиотека GNU C предоставляет этот дополнительный аргумент, но это имеет смысл только в операционных системах, которые предоставляют информацию (системы BSD и системы GNU).

FPE_INTOVF_TRAP — Целочисленное переполнение (невозможно в программе на C, если вы не включили перехват переполнения в зависимости от оборудования).

FPE_INTDIV_TRAP — Целочисленное деление на ноль.

FPE_SUBRNG_TRAP — Нижний индекс (то, что программы C никогда не проверяют).

FPE_FLTOVF_TRAP — Ловушка переполнения с плавающей запятой через верхнюю границу (overflow).

FPE_FLTUND_TRAP — Ловушка переполнения с плавающей запятой через нижнюю границу (underflow) (отлов обычно не включен).

FPE_FLTDIV_TRAP — Плавающее/десятичное деление на ноль.

FPE_DECOVF_TRAP — Десятичная ловушка переполнения. (Только несколько машин имеют десятичную арифметику, и C никогда ее не использует.)

SIGILL – неверная инструкция

Название этого сигнала происходит от «illegal instruction»; обычно это означает, что ваша программа пытается выполнить мусор или привилегированную инструкцию. Поскольку компилятор C генерирует только допустимые инструкции, **SIGILL** обычно указывает, что исполняемый файл поврежден или что вы пытаетесь выполнить данные.

Некоторые распространенные способы попасть в такую ситуацию — передать недопустимый объект в случае, когда ожидался указатель на функцию, или выполнить запись за конец автоматического массива (аналогичные проблемы возникают с указателями на автоматические переменные), повредить другие данные в стеке, например, адрес возврата в кадре стека.

SIGILL также может быть сгенерирован при переполнении стека или когда в системе возникают проблемы с запуском обработчика сигнала.

SIGSEGV – нарушение доступа к памяти

Этот сигнал генерируется, когда программа пытается читать или писать за пределами выделенной для нее памяти или записывать в память, доступную только для чтения. (Фактически, сигналы возникают только тогда, когда программа выходит достаточно далеко за пределы, чтобы быть обнаруженной механизмом защиты памяти системы.) Название является аббревиатурой от «segmentation violation — нарушение сегментации».

Общие способы получения условия «SIGSEGV» включают разыменование нулевого или неинициализированного указателя или когда вы используете указатель для перехода по массиву, но не можете проверить конец массива. В разных системах зависит, генерирует ли разыменование нулевого указателя **SIGSEGV** или **SIGBUS**.

SIGBUS — разыменование неинициализированного указателя

Этот сигнал генерируется при разыменовании недопустимого указателя. Как и **SIGSEGV**, этот сигнал обычно является результатом разыменования неинициализированного указателя.

Разница между ними в том, что **SIGSEGV** указывает на недопустимый доступ к действительной памяти, а **SIGBUS** указывает на доступ к недопустимому адресу. В частности, сигналы **SIGBUS** часто возникают в результате разыменования неверно выровненного указателя, такого как обращение к целому числу из четырех слов по адресу, не делящемуся на четыре. (У каждого типа компьютера есть свои требования к выравниванию адресов.)

Название этого сигнала представляет собой сокращение от «bus error — ошибка шины».

SIGABRT

Этот сигнал указывает на ошибку, обнаруженную самой программой и сообщенную с помощью вызова **abort()**. Программа прерывается.

SIGIOT

Генерируется инструкцией **iot** PDP-11. На большинстве машин это просто другое название **SIGABRT**.

SIGTRAP

Генерируется инструкцией точки останова машины и, возможно, другими инструкциями прерывания. Этот сигнал используется отладчиками. Программа, вероятно, увидит **SIGTRAP**, только если она каким-то образом выполняет неверные инструкции.

SIGEMT

Ловушка эмулятора. Это происходит из-за некоторых нереализованных инструкций, которые могут быть эмулироваться в программном обеспечении, или из-за того, что операционная система не может их должным образом эмулировать.

SIGSYS

Плохой системный вызов; то есть инструкция по перехвату для операционной системы была выполнена, но кодовый номер для выполнения системного вызова был недопустимым.

Сигналы завершения

Сигналы завершения используются, чтобы так или иначе сказать процессу о завершении. У них разные имена, потому что они используются для немного разных целей, и программы могут обращаться с ними по-другому.

Причина обработки этих сигналов обычно заключается в том, чтобы ваша программа могла привести себя в порядок перед фактическим завершением. Например, вы можете захотеть сохранить информацию о состоянии, удалить временные файлы или восстановить предыдущие режимы терминала.

Такой обработчик должен заканчиваться указанием действия по умолчанию для пришедшего сигнала и его повторным вызовом — это приведет к тому, что программа завершится с этим сигналом, как если бы у нее не было обработчика. (4.4.2 Handlers That Terminate the Process)

Действие по умолчанию для всех этих сигналов — завершить процесс.

SIGTERM – общий сигнал, используемый для завершения программы;
SIGQUIT – прерывание программы с выдачей дампа (Ctrl-\);
SIGKILL – немедленное завершение программы;
SIGHUP – терминал пользователя отключен.

SIGTERM — завершить программу

Сигнал **SIGTERM** — это общий сигнал, используемый для завершения программы. В отличие от **SIGKILL**, этот сигнал можно блокировать, обрабатывать и игнорировать. Это нормальный способ вежливо попросить программу завершить работу.

Команда оболочки **kill** по умолчанию генерирует **SIGTERM**.

SIGINT — прервать программу

Сигнал SIGINT («program interrupt — прерывание программы») отправляется, когда пользователь вводит символ **INTR** (обычно «Ctrl-C»).

SIGQUIT — прервать программу с выдачей дампа

Сигнал **SIGQUIT** похож на **SIGINT**, за исключением того, что он управляется другой клавишей — символом **QUIT**, обычно «Ctrl - \», и создает дамп памяти при завершении процесса, как сигнал об ошибке программы. О нем можно думать, как об «обнаружении ошибки» пользователем.

Некоторые виды очистки лучше не выполнять при обработке **SIGQUIT**. Например, если программа создает временные файлы, она должна обрабатывать другие запросы на завершение, удаляя временные файлы. Но в случае **SIGQUIT** лучше их не удалять, чтобы пользователь мог изучить их вместе с дампом ядра.

SIGKILL — немедленное завершение программы

Сигнал **SIGKILL** используется для немедленного завершения программы. Его нельзя обрабатывать или игнорировать, поэтому он всегда фатален.

Заблокировать сигнал SIGKILL невозможно.

Этот сигнал обычно генерируется только явным запросом. Поскольку с ним невозможно справиться, создавать его следует только в крайнем случае, после того как сначала будут испробованы менее радикальные методы, такие как «Ctrl-C» или **SIGTERM**». Если процесс не отвечает ни на какие другие сигналы завершения, отправка ему сигнала **SIGKILL** почти всегда приведет к его завершению.

Фактически, если **SIGKILL** не может завершить процесс, это само по себе является ошибкой операционной системы, о которой вам следует сообщить.

Система будет генерировать **SIGKILL** для самого процесса в некоторых необычных условиях, когда программа не может продолжать работу (даже для запуска обработчика сигналов).

SIGHUP — разрыв соединения

Сигнал **SIGHUP** («Отбой») используется для сообщения о том, что терминал пользователя отключен, возможно, из-за разрыва сетевого или телефонного соединения (17.4.6 Control Modes).

Этот сигнал также используется для сообщения о завершении управляющего процесса на терминале заданиям, связанным с этим сеансом; это завершение эффективно отключает все процессы в сеансе от управляющего терминала (25.7.5 Termination Internals).

Аварийные сигналы (Alarm Signals)

Аварийные сигналы используются для индикации истечения таймеров. Установка сигнала тревоги для получения информации о функциях, которые вызывают отправку этих сигналов.

По умолчанию эти сигналы вызывают завершение программы.

Это значение по умолчанию редко бывает полезным, но любое другое значение по умолчанию бесполезно — для большинства способов использования этих сигналов в любом случае потребуются функции-обработчики.

SIGALRM – истечение таймера, который измеряет реальное время;
SIGVTALRM – истечение таймера процесса;
SIGPROF – истечение таймера процесса и системы.

SIGALRM – истечение таймера реального времени

Этот сигнал обычно указывает на истечение таймера, который измеряет реальное время или CLK-время. Например, он используется функцией **alarm(), ualarm(), setitimer()**.

SIGVTALRM – истечение таймера процесса

Этот сигнал обычно указывает на истечение таймера, который измеряет время ЦП, используемое текущим процессом. Название представляет собой аббревиатуру от «virtual time alarm – виртуальный будильник».

SIGPROF – истечение таймера процесса и системы

Этот сигнал обычно указывает на истечение таймера, который измеряет как время ЦП, используемое текущим процессом, так и время ЦП, затраченное системой на обслуживание текущего процесса. Такой таймер используется для реализации средств профилирования кода, отсюда и название этого сигнала.

Асинхронные сигналы ввода/вывода

Эти сигналы используются для обозначения доступности ввода и используются вместе со средствами асинхронного ввода/вывода. Чтобы конкретный файловый дескриптор мог генерировать эти сигналы (инициирование прерывания), необходимо предпринять явное действие, вызвая системный вызов **`fcntl()`** (13.19 Interrupt-Driven Input).

Действие по умолчанию для этих сигналов — игнорировать их.

SIGIO – дескриптор файла готов к вводу/выводу;

SIGURG – внеполосные данные;

SIGIO — дескриптор файла готов к вводу/выводу

Этот сигнал отправляется, когда дескриптор файла готов к вводу или выводу.

В большинстве операционных систем терминалы и сокеты — единственные типы файлов, которые могут генерировать **SIGIO**.

Другие типы файлов, включая обычные файлы, никогда не создают **SIGIO**, даже если это попросить.

В системах GNU, если успешно установлен асинхронный режим с помощью **fcntl()**, **SIGIO** всегда будет сгенерирован правильно.

SIGURG — внеполосные данные

Этот сигнал отправляется, когда на сокет поступают «срочные» или «внеполосные данные» (16.9.8 Out-of-Band Data).

SIGPOLL —

Это имя сигнала System V, более или менее похожее на SIGIO. Определен только для совместимости.

Сигналы управления заданиями

Эти сигналы используются для поддержки управления заданиями. Если система не поддерживает управление заданиями, то эти макросы определены, но сами сигналы не могут быть вызваны или обработаны.

Если нет понимания, как работает управление заданиями (28 Job Control), эти сигналы следует игнорировать.

SIGCHLD – завершение дочернего процесса;

SIGSTOP – останавить процесс;

SIGTSTP – интерактивный стоп-сигнал;

SIGCONT – продолжить выполнение;

SIGTTIN – попытка прочитать с терминала в фоновом режиме;

SIGTTOU – попытка записать на терминал в фоновом режиме;

SIGCHLD — завершение дочернего процесса

Этот сигнал отправляется родительскому процессу всякий раз, когда один из его дочерних процессов завершается или останавливается.

Действие по умолчанию для этого сигнала — его игнорировать. Если обработчик для этого сигнала устанавлен, в случае, когда есть дочерние процессы, которые завершились, но не сообщили о своем статусе через **wait()** или **waitpid()** (завершение процесса), будет ли этот обработчик применяться к таким процессам или нет, зависит от конкретной операционной системы.

SIGCLD — устаревшее название для SIGCHLD

SIGCONT — продолжить выполнение

Этот сигнал особый — он всегда заставляет процесс продолжать выполнение, если он остановлен, до того, как сигнал будет доставлен. По умолчанию больше ничего не делается.

Заблокировать сигнал SIGCONT невозможно.

Можно установить обработчик, но **SIGCONT** всегда заставит процесс продолжать выполнение.

У большинства программ нет причин обрабатывать **SIGCONT** — они просто возобновляют исполнение, даже не подозревая, что их когда-либо останавливали.

Можно использовать обработчик для «SIGCONT», чтобы сделать программу какой-то особенной, когда она была остановлена и продолжена, например, для повторной печати приглашения, когда она приостанавливается во время ожидания ввода.

SIGSTOP — останавить процесс

Сигнал SIGSTOP останавливает процесс.
Его нельзя обработать, игнорировать или заблокировать.

SIGTTIN — попытка прочитать с терминала в фоновом режиме

Процесс не может считывать данные с пользовательского терминала, пока он выполняется в качестве фонового задания. Когда какой-либо процесс в фоновом задании пытается прочитать с терминала, всем процессам в задании отправляется сигнал **SIGTTIN**.
Действие по умолчанию для этого сигнала — остановить процесс.

SIGTTOU — попытка записать на терминал в фоновом режиме

Это похоже на **SIGTTIN**, но генерируется, когда процесс в фоновом задании пытается записать в терминал или установить свои режимы.
Действие по умолчанию — остановить процесс.
SIGTTOU генерируется только для попытки записи в терминал, если установлен режим вывода **TOSTOP (**Режимы вывода).

SIGTSTP — интерактивный стоп-сигнал

Сигнал **SIGTSTP** — это интерактивный стоп-сигнал.

В отличие от SIGSTOP, этот сигнал можно как обработать, так и проигнорировать.

Программа должна обрабатывать этот сигнал, если есть особая необходимость оставить файлы или системные таблицы в безопасном состоянии при остановке процесса.

Например, программы, которые отключают эхо на вводе, должны обрабатывать SIGTSTP, чтобы они могли снова включить эхо перед остановкой.

Этот сигнал генерируется, когда пользователь вводит символ **SUSP** (обычно «Ctrl-Z»).

Когда процесс остановлен, никакие другие сигналы кроме сигналов **SIGKILL** и, что очевидно, **SIGCONT** не могут быть доставлены ему до тех пор, пока он не будет продолжен.

В этом случае сигналы помечаются как ожидающие и не доставляются пока процесс не будет продолжен.

Сигнал **SIGKILL** всегда вызывает завершение процесса и не может быть заблокирован, обработан или проигнорирован.

Можно игнорировать **SIGCONT**, но это всегда приводит к продолжению процесса, если тот остановлен. Отправка сигнала **SIGCONT** процессу приводит к тому, что все ожидающие сигналы остановки для этого процесса отбрасываются.

Аналогичным образом, любые сигналы **SIGCONT**, ожидающие обработки, отбрасываются, когда процесс получает сигнал остановки.

Когда процесс в группе осиротевших процессов (Orphaned Process Groups) получает сигнал **SIGTSTP**, **SIGTTIN** или **SIGTTOU** и не обрабатывает его, процесс не останавливается.

Остановка процесса, вероятно, не будет очень полезной, поскольку нет программы оболочки, которая заметит его остановку и позволит пользователю ее продолжить. Что произойдет вместо этого, зависит от используемой вами операционной системы. Некоторые системы могут ничего не делать, другие могут вместо этого передать другой сигнал, например **SIGKILL** или **SIGHUP**.

В системах GNU/Hurd процесс завершается с **SIGKILL** — это позволяет избежать проблемы множества остановленных, потерянных процессов, лежащих в системе.

Сигналы ошибок обслуживания

Сигналы ошибок обслуживания используются для сообщения об ошибках операционной системы. Эти сигналы используются для сообщения о различных ошибках, вызванных работой выполняемой программы. Они не обязательно указывают на ошибку программирования в программе, но на ошибку, которая препятствует выполнению вызова к операционной системе.

Действие по умолчанию для всех из них — завершить процесс.

SIGPIPE – разрушенный канал/конвейер (pipe);
SIGXCPU – превышено ограничение по времени ЦП;
SIGXFSZ – превышен предел размера файла;
SIGLOST – потеря ресурса.

SIGPIPE — разрушенный канал/конвейер (pipe)

При использовании каналов или FIFO, приложение должно быть спроектировано так, чтобы один процесс открывал канал для чтения, прежде чем другой начнет запись.

Если процесс чтения никогда не начинается или неожиданно завершается, запись в канал или FIFO вызывает сигнал **SIGPIPE**.

Если **SIGPIPE** блокируется, обрабатывается или игнорируется, вызов, вызывающий нарушение, завершается неудачно с ошибкой **EPIPE**.

Другая причина **SIGPIPE** состоит в попытке вывода в сокет, который не подключен.

SIGXCPU — превышено ограничение по времени ЦП

Этот сигнал генерируется, когда процесс превышает свой программный предел ресурсов по времени ЦП.

SIGXFSZ — превышен предел размера файла

Этот сигнал генерируется, когда процесс пытается расширить файл так, чтобы он превышал программный предел ресурсов процесса на размер файла.

SIGLOST — потеря ресурса

Этот сигнал генерируется, когда установлена рекомендательная блокировка для NFS-файла, а NFS-сервер перезагрузился и о блокировке «забыл».

В системах GNU/Hurd **SIGLOST** генерируется, когда неожиданно умирает какая-либо серверная программа.

Нормально игнорировать этот сигнал — какой бы вызов не был сделан к умершему серверу, будет просто возвращана ошибка.

Прочие сигналы

Эти сигналы используются для различных целей.

В общем, они никак не повлияют на поведение программы, если она явно не использует эти сигналы для чего-либо.

SIGUSR1, SIGUSR2 — пользовательские сигналы

Сигналы **SIGUSR1** и **SIGUSR2** зарезервированы для пользователя, чтобы он мог их использовать любым удобным для него способом.

Эти сигналы полезны для простого межпроцессного взаимодействия, если в программе, принимающей сигнал, для них будет написан обработчик.

Действие по умолчанию — завершить процесс.

SIGWINCH — изменение размера окна

Сигнал генерируется в некоторых системах (включая GNU) при изменении записи драйвера терминала, в которой указывается количество строк и столбцов на экране.

Действие по умолчанию — игнорировать.

Если программа работает в полноэкранном режиме, она должна обрабатывать **SIGWINCH**. Когда поступит этот сигнал, программа должна получить новый размер экрана и соответствующим образом переформатировать его отображение.

SIGINFO — запрос информации

В системах BSD 4.4 и GNU/Hurd этот сигнал отправляется всем процессам в группе процессов переднего плана управляющего терминала, когда пользователь вводит символ **STATUS**.

Если процесс является лидером группы процессов, действие по умолчанию — распечатать некоторую информацию о состоянии системы и о том, что делает процесс. В противном случае по умолчанию ничего не делать.

Сообщения о сигналах

Оболочка может печатать сообщение, описывающее сигнал, который завершил дочерний процесс. Прямой способ распечатать сообщение, описывающее сигнал, — это использовать функции **strsignal()**[3] и **psignal**[4]**()**.

Эти функции используют номер сигнала, чтобы указать, для какого тип сигнала вывести соответствующее описание.

Номер сигнала может поступить из состояния завершения дочернего процесса (завершение процесса) или может поступать от обработчика сигнала в том же процессе.

strsignal

```
#include <string.h>

char *strsignal(int sig);
extern const char *const sys_siglist[];
```

Эта функция возвращает указатель на статически выделенную строку, содержащую сообщение, описывающее сигнал **sig**.

Изменять содержимое этой строки нельзя и, поскольку она может быть переписана при последующих вызовах, необходимо сохранять копию строки, если понадобится ссылаться на нее позже.

Эта функция является расширением GNU, объявленным в заголовочном файле **string.h**.

---

3) strsignal — return string describing signal
4) psignal — print signal message

psignal

```
#include <signal.h>

void psignal(int sig, const char *sigmsg);
void psiginfo(const siginfo_t *pinfo, const char *sigmsg);

extern const char *const sys_siglist[];
```

Эта функция выводит сообщение, описывающее сигнал **sig**, в стандартный поток вывода ошибок **stderr**.

Если **psignal** вызывается с **sigmsg**, которое является либо нулевым указателем, либо пустой строкой, **psignal()** просто печатает сообщение, соответствующее **sig**, добавляя завершающий **'\n'** .

Если указан ненулевой аргумент **sigmsg**, **psignal()** добавит эту строку к своему выводу.

psignal() добавляет двоеточие и пробел, чтобы отделить **sigmsg** от строки, соответствующей **sig**.

Эта функция является функцией BSD, объявленной в заголовочном файле **signal.h**.

Также существует массив **sys_siglist**, который содержит сообщения для различных сигнальных кодов. Этот массив существует в системах BSD.

Обе ничего не возвращают.

Определение сигнальных действий

Самый простой способ изменить действие для сигнала — использовать функцию **`signal()`**.

Эта функция позволяет указать действие (например, игнорировать сигнал) или «*установить обработчик*».

Библиотека GNU C также реализует более гибкую функцию **`sigaction()`**.

Базовая обработка сигналов

Функция **`signal()`** предоставляет простой интерфейс для установки действия для определенного сигнала. Функция и связанные с ней макросы объявлены в заголовочном файле **`signal.h`**.

`sighandler_t` — тип функции-обработчика сигналов

```
#include <signal.h>

typedef void (*sighandler_t)(int); // ANSI C
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, как:

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

Имя **`sighandler_t`** является расширением GNU.

`signal()` — установить обработчик сигнала

```
#include <signal.h>

sighandler_t signal(int SIGNUM, sighandler_t ACTION)
```

Функция **`signal()`** устанавливает **`ACTION`** как действие для сигнала **`SIGNUM`**.

Первый аргумент, **`SIGNUM`**, определяет сигнал, поведение которого вы хотите контролировать, и должен быть номером сигнала.

Правильный способ указать номер сигнала — использовать одно из символических имен сигналов (см. Стандартные сигналы). Не следует использовать явное число, потому что числовой код для данного вида сигнала может варьироваться от одной ОС к другой.

Второй аргумент, **`ACTION`**, указывает действие, используемое для сигнала **`SIGNUM`**. Он может быть одним из следующих:

`SIG_DFL`
`SIG_IGN`
`HANDLER`

`SIG_DFL` определяет действие по умолчанию для конкретного сигнала.

Действия по умолчанию для различных типов сигналов указаны в разделе «Стандартные сигналы».

SIG_IGN указывает, что сигнал следует игнорировать.

Программа, как правило, не должна игнорировать сигналы, которые представляют серьезные события или обычно используются для запроса завершения.

Сигналы **SIGKILL** или **SIGSTOP** не возможно игнорировать.

Можно игнорировать сигналы об ошибках программы, такие как **SIGSEGV**, но игнорирование такой ошибки не позволит программе продолжить осмысленное выполнение.

Игнорирование пользовательских запросов, таких как **SIGINT**, **SIGQUIT** и **SIGTSTP**, является неприемлемым.

Если нежелательно, чтобы сигналы передавались во время определенной части программы, нужно их заблокировать, но не игнорировать.

HANDLER – указывает адрес функции обработчика в программе, чтобы указать запуск этого обработчика, как способ доставки сигнала.

Если устанавливается действие для сигнала в **SIG_IGN**, или если действие устанавливается в **SIG_DFL**, а действие по умолчанию — игнорировать этот сигнал, то все ожидающие сигналы этого типа отбрасываются (даже если они заблокированы).

Отказ от ожидающих сигналов означает, что они никогда не будут доставлены, даже если вы впоследствии укажете другое действие и разблокируете этот вид сигнала.

Возвращает

Функция **signal()** возвращает действие, которое ранее действовало для указанного **SIGNUM**. Можно сохранить это значение и восстановить его позже, вызвав **signal()** снова.

Если **signal()** не может выполнить запрос, то она возвращает **SIG_ERR**.

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

EINVAL

Указан неверный **SIGNUM** или была попытка игнорировать или предоставить обработчик для **SIGKILL** или **SIGSTOP**.

Примечание по совместимости:

Проблема, возникающая при работе с функцией **signal()**, заключается в том, что она имеет разную семантику в системах BSD и SVID.

Разница в том, что в системах SVID обработчик сигнала удаляется после доставки сигнала.

В системах BSD обработчик должен быть явно деинсталлирован.

В библиотеке GNU C по умолчанию используется версия BSD.

Чтобы использовать версию SVID, следует использовать функцию **sysv_signal()** или использовать макрос выбора функции **_XOPEN_SOURCE** (Feature Test Macros).

В общем, следует избегать использования этих функций из-за проблем совместимости.

Лучше использовать функцию **sigaction()**, если она доступна, поскольку ее переносимость намного надежнее.

Пример настройки обработчика для удаления временных файлов при возникновении определенных фатальных сигналов:

```
#include <signal.h>

void termination_handler(int signum) {

  struct temp_file *p; // указатель на элемент списка, содержащего tmpfile info

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

int main (void) {
  ...
  if (signal(SIGINT, termination_handler) == SIG_IGN)
      signal(SIGINT, SIG_IGN);
  if (signal(SIGHUP, termination_handler) == SIG_IGN)
      signal(SIGHUP, SIG_IGN);
  if (signal(SIGTERM, termination_handler) == SIG_IGN)
      signal(SIGTERM, SIG_IGN);
  ...
}
```

Следует обратить внимание, что если данный сигнал ранее был настроен на игнорирование, код избегает изменения этой настройки. Это связано с тем, что оболочки, не управляющие заданиями, часто игнорируют определенные сигналы при запуске дочерних элементов, и важно, чтобы потомки вели себя аналогично.

В этом примере не обрабатывается **SIGQUIT** или сигналы программных ошибок, потому что они предназначены для предоставления информации для отладки (-> дамп ядра), а временные файлы могут содержать полезную информацию.

`sysv_signal` — установить обработчик сигнала по версии SVID

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t sysv_signal(int signum, sighandler_t ACTION);
```

`sysv_signal()` реализует поведение стандартной функции **`signal()`**, как это имеет место в системах SVID.

Отличие от систем BSD – обработчик деинсталлируется после доставки сигнала.

Примечание по совместимости:
Как сказано выше для функции **`signal()`**, по возможности следует избегать этой функции.
Предпочтительный метод – **`sigaction()`**.

`sighandler_t SIG_ERR`

Значение этого макроса используется как возвращаемое значение от **`signal()`** для указания ошибки.