Сигналы

Андрей Боровский

We know when to kiss
And we know when to kill
If we can`t have it all
Then nobody will
Garbage. The World is not Enough

Главное отличие сигналов от других средств взаимодействия между процессами заключается в том, что их обработка программой обычно происходит сразу же после поступления сигнала (или не происходит вообще), независимо от того, что программа делает в данный момент. Сигнал прерывает нормальный порядок выполнения инструкций в программе и передает управление специальной функции – обработчику сигнала. Если обработка сигнала не приводит к завершению процесса, то по выходе из функции-обработчика выполнение процесса возобновляется с той точки, в которой оно было прервано. У программ также есть возможность приостановить обработку поступающих сигналов временно, на период выполнения какой-либо важной операции. В традиционной терминологии приостановка получения определенных сигналов называется *блокированием*. Если для поступившего сигнала было установлено блокирование, сигнал будет передан программе, как только она разблокирует данный тип сигналов. Этим блокирование отличается от *игнорирования* сигнала, при котором сигналы соответствующего типа никогда не передаются программе. Следует помнить, что не все сигналы могут быть проигнорированы. Например, при получении программой сигнала принудительного завершения SIGKILL система ничего не сообщает программе, а просто прекращает ее работу. Таким образом, преимущество сигналов перед другими средствами взаимодействия с программой заключается в том, что посылать программе сигналы можно в любой момент ее работы, не дожидаясь наступления каких-то особых условий. Источником сигналов может быть как сама операционная система, так и другие программы пользователя. Если вам показалось, что сигналы похожи на прерывания, то вы совершенно правы. Для реализации сигналов действительно используются программные прерывания.

Нужно ли обрабатывать сигналы в вашей программе? Большинство программ не делают этого. В случае программирования для графических оболочек многие функции сигналов берут на себя механизмы сообщений графической оболочки. Тем не менее, есть целый ряд программ (например, демоны и консольные многопоточные приложения), в которых обработка сигналов необходима.

Большинству сигналов системы присвоена конкретная роль и, хотя у программиста существует возможность использовать сигналы для передачи произвольной информации, не соответствующей их стандартному назначению, делать этого не рекомендуется. Собственно говоря, с помощью сигналов можно передать не так уж и много информации – только номер сигнала (хотя на платформе x86, например, можно было бы организовать и передачу дополнительных параметров). Скудость данных, передаваемых сигналами, не удивительна, если учесть, что по умолчанию большинство сигналов просто завершают работу программы. При этом в некоторых случаях на диске сохраняется образ памяти выгруженной программы (знаменитый файл core dump). Соответственно и программа-источник сигнала обычно не ждет никакого ответа от программы-приемника. Номерам сигналов сопоставлены константы, определенные в файле <signal.h>. Имена всех этих констант начинаются с

префикса SIG, за которыми следует сокращенное название сигнала. Стандарт POSIX определяет две группы сигналов – «классические» сигналы Unix и сигналы реального времени. В отличие от классических сигналов сигналы реального времени всегда буферизуются, так что программа получит все посланные ей сигналы. В этой статье мы рассмотрим только классические сигналы Unix, каковых в Linux насчитывается 31. Этим сигналам назначены номера с 1 до 31 (номер 0, так называемый null-сигнал имеет особый смысл). Полный список сигналов можно получить из заголовочного файла signal.h. Мы рассмотрим несколько наиболее интересных сигналов.

Сигнал SIGHUP (номер 1) изначально был предназначен для того, чтобы информировать программу о потере связи с управляющим терминалом (терминалы часто подключались к системе с помощью модемов, так что название сигнала происходит от hung up – повесить трубку). Сигнал SIGHUP посылается приложению так же и в том случае, если процесс-лидер сессии завершил свою работу. Многие программы-демоны, у которых нет лидера сессии, так же обрабатывают этот сигнал. В ответ на получение SIGHUP демон обычно перезапускается (или просто повторно читает файл конфигурации). По умолчанию программа, получившая этот сигнал, завершается.

Сигнал SIGINT (номер 2) обычно посылается процессу, если пользователь терминала дал команду прервать процесс (обычно эта команда – сочетание клавиш Ctrl-C) .

Сигнал SIGABRT (номер 6) посылается программе в результате вызова функции abort(3). В результате программа завершается с сохранением на диске образа памяти.

Сигнал SIGKILL (номер 9) завершает работу программы. Программа не может ни обработать, ни игнорировать этот сигнал.

Сигнал SIGSEGV (номер 11) посылается процессу, который пытается обратиться к не принадлежащей ему области памяти. Если обработчик сигнала не установлен, программа завершается с сохранением на диске образа памяти.

Сигнал SIGTERM (номер 15) вызывает «вежливое» завершение программы. Получив этот сигнал, программа может выполнить необходимые перед завершением операции (например, высвободить занятые ресурсы). Получение SIGTERM свидетельствует не об ошибке в программе, а о желании ОС или пользователя завершить ее.

Сигнал SIGCHLD (номер 17) посылается процессу в том случае, если его дочерний процесс завершился или был приостановлен. Родительский процесс также получит этот сигнал, если он установил режим отслеживания сигналов дочернего процесса и дочерний процесс получил какой-либо сигнал. По умолчанию сигнал SIGCHLD игнорируется.

Сигнал SIGCONT (номер 18) возобновляет выполнение процесса, остановленного сигналом SIGSTOP.

Сигнал SIGSTOP (номер 19) приостанавливает выполнение процесса. Как и SIGKILL, этот сигнал не возможно перехватить или игнорировать.

Сигнал SIGTSTP (номер 20) приостанавливает процесс по команде пользователя (обычно эта команда – сочетание клавиш Ctrl-Z).

Сигнал SIGIO/SIGPOLL (в Linux обе константы обозначают один сигнал – номер 29) сообщает процессу, что на одном из дескрипторов, открытых асинхронно, появились данные. По умолчанию этот сигнал, как ни странно, завершает работу программы.

В стандартной системе Unix определены два сигнала, SIGUSR1 (в Linux – номер 10) и SIGUSR2 (номер 12), предназначенные для передачи произвольной

информации, но использование этих сигналов не приветствуется. Одной из причин негативного отношения программистов Unix к пользовательским сигналам является то, что сигналы, вообще говоря, представляют собой ограниченный ресурс, совместное использование которого может вызвать конфликты (например, если программист задействовал эти сигналы в своей программе и при этом использует стороннюю библиотеку, в которой эти сигналы также задействованы).

Если вы не знали, то вам, возможно, будет интересно узнать, что обработка сигналов является частью стандарта языка Си и, как таковая, поддерживается даже на платформе Microsoft Windows. Однако, стандартный интерфейс сигналов Си, основанный на функции signal(), довольно неуклюж (недостатки интерфейса сигналов Си подробно описаны в книге [2]), так что мы воспользуемся более совершенным вариантом интерфейса, основанным на функции sigaction(2). Для демонстрации работы обработки сигналов мы напишем небольшую программу (файл sigdemo.c на компакт-диске).

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

void term_handler(int i)
{
  printf ("Terminating\n");
  exit(EXIT_SUCCESS);
}

int main(int argc, char ** argv) {
  struct sigaction sa;
  sigset_t newset;
  sigemptyset(&newset);
  sigaddset(&newset, SIGHUP);
  sigprocmask(SIG_BLOCK, &newset, 0);
  sa.sa_handler = term_handler;
  sigaction(SIGTERM, &sa, 0);
  printf("My pid is %i\n", getpid());
  printf("Waiting...\n");
  while(1) sleep(1);
  return EXIT_FAILURE;
}
```

Наша программа делает две вещи: обрабатывает сигнал SIGTERM (при получении этого сигнала программа выводит диагностическое сообщение и завершает свою работу) и блокирует сигнал SIGHUP, так что этот сигнал не может завершить ее работу. В тексте программы мы первым делом определяем функцию-обработчик сигнала SIGTERM term_handler(). Функции-обработчики сигналов – это обычные функции Си, они имеют доступ ко всем глобально видимым переменным и функциям. Однако, поскольку мы не знаем, в какой момент выполнения программы будет вызвана функция-обработчик, мы должны проявлять особую осторожность при обращении к глобальным структурам данных из этой функции. Для функций, обрабатывающих потоки, существует и еще одно важное требование – реентерабильность. Поскольку обработчик сигнала может быть вызван в любой точке выполнения программы (а при не кототорых условиях во время обработки одного сигнала может быть вызван другой обработчик сигнала) в обработчиках додлжны использоваться функции, которые удовлетворяют требованию реентерабельности, то есть, могут быть вызваны в то время, когда они уже вызваны где-то в другой точке программы. Фактически, требование реентерабельности сводится к тому, чтобы функция не использовала никаких глобальных ресурсов, не позаботившись о синхронизации доступа к этим ресурсам. Некоторые функции ввода-вывода, в том числе, функция printf(), которую мы (и не только мы) используем в примерах обработчиков сигналов, реентерабельными не являются. Это значит, что выводу одной функции printf()

может помешать вывод другой функции. В приложении приводится список реентерабельных функций, которые безопасно вызвать из обработчиков сигналов.

Единственным параметром нашего варианта функции-обработчика сигнала (в Unix-системах существует и другой вариант) является переменная типа int, в которой передается номер сигнала, вызвавшего обработчик. Нам этот номер не нужен, поскольку мы *знаем*, что только один сигнал, - SIGTERM, может вызвать нашу функцию, однако, в принципе, ничто не мешает нам использовать одну функцию для обработки нескольких разных сигналов, и тогда параметр функции-обработчика будет иметь для нас смысл. Функция-обработчик не возвращает никакого значения, что вполне логично, так как она вызывается не нашей программой, а неким системным компонентом. Особый интерес представляет завершение программы из обработчика сигнала. Назначение обработчика сигналу SIGTERM означает, что умалчиваемое действие сигнала, – завершение программы, не будет выполняться автоматически, и нам необходимо (если, конечно, мы хотим, чтобы этот сигнал завершал программу) позаботиться об этом явным образом. Если вы закомментируете вызов exit() в нашем примере, то увидите, что программа не будет завершать по получении сигнала SIGTERM. В принципе, вы можете придать сигналу SIGTERM совершенно иной смысл, например, оповещать программу о наступлении времени вашей любимой телепередачи (или о выходе нового номера журнала Linux Format), однако назначать стандартным сигналам нестандартные действия категорически не рекомендуется. Обработчик SIGTERM предназначен для того, чтобы, по требованию системы или пользователя, программа могла быстро и элегантно закончить текущую задачу и завершить свое выполнение. Именно этим обработчик и должен заниматься.

Перейдем теперь к тексту главной функции программы. Установка и удаление обработчиков сигналов осуществляются функцией sigaction(2). Первым параметром этой функции является номер сигнала, а в качестве второго и третьего параметров следует передать указатели на структуру sigaction. Эта структура содержит данные об операции, выполняемой над обработчиком сигнала. Второй параметр sigaction() служит для передачи новых значений для обработки сигнала, а третий – возвращает ранее установленные значения. В таблице 1 приводится краткое описание полей структуры sigaction.

Таблица 1. Поля структуры sigaction.

Поле	**Значение**
sa_handler	Указатель на функцию обработчик сигнала или константа.
sa_mask	Маска сигналов, блокируемых на время вызова обработчика.
sa_flags	Дополнительные флаги.

Поле sa_handler должно содержать либо адрес функции-обработчика, либо специальную константу, указывающую, что нужно делать с сигналом. Константа SIG_IGN указывает, что сигнал следует игнорировать, а константа SIG_DFL – что нужно восстановить обработку сигнала, заданную системой по умолчанию. Поле sa_mask позволяет заблокировать некоторое множество сигналов на время выполнения обработчика данного сигнала. Делается это для того, чтобы обработка других сигналов не могла прервать обработку данного (это может быть необходимо, особенно, если один обработчик обрабатывает несколько разных сигналов). Параметр sa_flags позволяет задать ряд флагов для выполнения более тонкой настройки обработчика сигналов. Например, флаг SA_RESETHAND указывает, что после завершения обработки сигнала заданным обработчиком должен быть восстановлен обработчик, заданный по умолчанию, так что все последующие сигналы будут обрабатываться умалчиваемым обработчиком.

В результате вызова функции sigaction() мы устанавливаем обработчик сигнала SIGTERM. Затем наша программа распечатывает значение PID (это значение понадобится нам для вызова команды kill) и входит в бесконечный цикл, из

которого она может быть выведена одним из сигналов. Следует отметить, что функция sleep() возвращает управление (возобновляет выполнение программы раньше срока) если обработчик какого-либо сигнала возвращает управление в этот момент. Иначе говоря, любой обрабатываемый сигнал прерывает выполнение sleep(). Впрочем, в нашем примере с бесконечным циклом это не помогло бы программе завершиться. Сигнал SIGTERM приведет к тому, что программа выдаст диагностическое сообщение и завершит работу, а сигналы SIGINT и SIGABRT – к тому, что программа завершится без всякого сообщения. Скомпилируйте и запустите программу в окне терминала. В другом окне скомандуйте

```
kill <PID>
```

где PID – идентификатор процесса программы. Вы увидите, что перед тем как завершиться программа выдает диагностическое сообщение, тогда как при завершении с помощью Ctrl-C никакого сообщения не выводится.

Рассмотрим теперь блокировку сигналов. Поскольку игнорирование сигнала устанавливается функцией sigaction(), можно было бы ожидать, что и блокировка устанавливается этой же функцией, но это не так. Так как зачастую программисту приходится блокировать несколько сигналов сразу, для блокировки существует специальная функция sigprocmask(2), которая оперирует наборами сигналов (signal sets). Разделение интерфейса между несколькими функциями вызвано еще и требованиями многопоточности. Параметры, устанавливаемые sigaction(), действительны для всей программы в целом, тогда как блокировку сигналов потоки осуществляют независимо друг от друга. Наборы сигналов хранятся в переменных специального типа - sigset_t, а операции над ними осуществляются с помощью специальных функций. Функция sigemptyset() инициализирует набор сигналов пустыми значениями, а функция sigfillset() устанавливает все возможные значения в наборе. Используемая нами функция sigaddset() добавляет значение сигнала в набор, а функция sigdelset() удаляет сигнал из набора. После того как набор сигналов сформирован, мы передаем его функции sigprocmask(), которая выполняет блокирование и разблокирование сигналов. Первым параметром этой функции должна быть одна из констант, определяющих операцию над заданными сигналами. Константа SIG_BLOCK указывает, что сигналы из нового набора должны быть добавлены к списку уже заблокированных сигналов. Константа SIG_SETMASK указывает, что новый набор блокируемых сигналов должен заменить уже существующий (при этом заблокированные ранее сигналы будут разблокированы, если они не заблокированы в новом наборе), а константа SIG_UNBLOCK указывает на необходимость разблокировать сигналы, переданные в наборе. В нашей программе мы блокируем сигнал SIGHUP и вы можете видеть, что программа не обрабатывает этот сигнал. Послать нашей программе сигнал SIGHUP вы можете с помощью консольной команды

```
kill –s 1 <PID>
```

где PID – идентификатор процесса.

Сигналы прерывают нормальный порядок выполнения программы и могут завершить работу программы, не способной завершиться иным образом. Но иногда бывает так, что программе просто нечего делать до тех пор, пока она не получит какой-либо сигнал. Иначе говоря, программу нужно заставить ждать появления сигнала, по возможности не нагружая процессор. Такая ситуация может возникнуть, например, в многопоточном программировании, когда нужно синхронизировать завершение нескольких потоков. Ожидание сигнала можно реализовать с помощью цикла, проверяющего значение флажка, который может сбросить обработчик сигнала. В некоторых случаях (таких как рассмотренный выше пример) можно реализовать ожидание и с помощью бесконечного цикла. Очевидно, однако, что эти методы не эффективны и не элегантны. В POSIX-системах существует специальная функция sigwait(3), которая «усыпляет» процесс до тех пор, пока процессу не будет передан один из заданного набора сигналов.

Модифицируем нашу программу так, чтобы вместо бесконечного цикла она входила в цикл ожидания сигнала SIGHUP (файл swdemo.c на компакт-диске):

```
sigprocmask(SIG_BLOCK, &newset, 0);
while(!sigwait(&newset, &sig))
  printf("SIGHUP recieved\n");
```

Первым параметром функции sigwait() является указатель на набор сигналов, получения которых будет ждать функция. Во втором параметре sigwait() вернет номер того сигнала, который возобновил работу программы (эта информация может быть полезна, если установлено несколько ожидаемых сигналов). Перед тем как вызывать sigwait(), набор ожидаемых сигналов следует заблокировать с помощью функции sigprocmask(), иначе, при получении сигнала, вместо выхода из sigwait() будет вызван соответствующий обработчик. Сигнал, который возобновил работу программы после вызова sigwait(), уже не может быть перехвачен назначенным ему обработчиком. В нашем примере мы «усыпляем» программу до тех пор, пока она не получит сигнал SIGHUP, распечатываем соответствующее сообщение и снова усыпляем (функция sigwait() возвращает 0, если ее вызов прошел успешно). В то время, когда программа приостановлена в ожидании некоторых сигналов, обработчики всех не заблокированных и не игнорируемых сигналов выполняются обычным образом.

Функцию sigwait() можно использовать и для исследования сигналов. На компакт-диске вы найдете программку siglog.c, которая распечатывает информацию о каждом поступившем сигнале (естественно, исследуются только те сигналы, которые могут быть заблокированы). Рассмотрим здесь фрагмент этой программы:

```
sigset_t sset;
int sig;

...

sigfillset(&sset);
sigdelset(&sset, SIGTERM);
sigprocmask(SIG_SETMASK, &sset, 0);
while(!sigwait(&sset, &sig))
  printf("Signal %i - %s\n", sig, sys_siglist[sig]);
```

С помощью вызовов sigfillset() и sigdelset() мы создаем набор из всех сигналов, за исключением сигнала SIGTERM (этот сигнал понадобится нам для того, чтобы мы могли завершить работу программы). Далее мы блокируем сигналы набора sset и вызываем для них функцию sigwait(). Функция вернет управление при получении любого сигнала, кроме SIGTERM (для которого назначен отдельный обработчик). Получив новый сигнал, мы распечатываем информацию о нем. Массив char * sys_siglist[] определен в стандартной библиотеке glibc. Этот массив содержит наименования сигналов на «человеческом» языке (эти наименования можно использовать при выводе диагностических и отладочных сообщений). Наименования расположены так, чтобы их индексы в массиве соответствовали номерам сигналов. Те же данные возвращает и функция strsignal(), единственным параметром которой является номер сигнала.

На протяжении всей этой статьи мы занимались обработкой сигналов, но не их генерацией. Поскольку основным источником сигналов является операционная система, нам и в «реальной жизни» чаще приходится заниматься именно обработкой. Однако, в заключение статьи следует рассмотреть и функции генерации сигналов. Для генерации сигналов в Unix предусмотрены две функции – kill(2) и raise(3). Первая функция предназначена для передачи сигналов любым процессам, к которым владелец данного процесса имеет доступ, а с помощью второй функции процесс может передать сигнал самому себе. Как это обычно принято в мире Unix, семантика вызова функции kill() совпадает с семантикой одноименной команды ОС. У функции kill() два аргумента –PID процесса-

приемника и номер передаваемого сигнала. С помощью функции kill() как и с помощью одноименной команды можно передавать сообщения не только конкретному процессу, но и группе процессов. Таблица 2 демонстрирует поведение функции kill() в зависимости от значения PID:

PID > 1	Сигнал посылается процессу с соответствующим PID.
PID == 0	Сигнал посылается всем процессам из той же группы что и процесс-источник.
PID < 0	Сигнал посылается всем процессам, чей идентификатор группы равен абсолютному значению PID.
PID == 1	Сигнал посылается всем процессам системы.

Вызов

```
raise(sig);
```

эквивалентен вызову

```
kill(getpid(), sig);
```

Так же как и для других примитивов IPC, для сигналов действует система прав доступа, основанная на правах доступа владельцев процессов. Процесс-приемник получит сигнал только в том случае, если у процесса-источника есть соответствующие права. С помощью функции kill() можно проверить, существует ли в системе процесс с заданным PID, не посылая процессу никаких сигналов. Для этого предназначен псевдо-сигнал с номером 0. Если соответствующего процесса не существует, функция kill() вернет значение 1, соответствующее об ошибке. В любом случае, сигнал не будет отправлен. Читателей, полюбивших обработку сигналов, я могу обрадовать тем, что мы рассмотрели далеко не все функции, связанные с сигналами. При изучении документации вас ждет еще много полезного и приятного, мы же закончим на этом наше знакомство с сигналами.

Приложение. Список реентерабельных функций

accept()	access()	aio_error()	aio_return()
aio_suspend()	alarm()	bind()	cfgetispeed()
cfgetospeed()	cfsetispeed()	cfsetospeed()	chdir()
chmod()	chown()	clock_gettime()	close()
connect()	creat()	dup()	dup2()
execle()	execve()	_Exit()	_exit()
fchmod()	fchown()	fcntl()	fdatasync()
fork()	fpathconf()	fstat()	fsync()
ftruncate()	getegid()	geteuid()	getgid()
getgroups()	getpeername()	getpgrp()	getpid()
getppid()	getsockname()	getsockopt()	getuid()
kill()	link()	listen()	lseek()
lstat()	mkdir()	mkfifo()	open()
pathconf()	pause()	pipe()	poll()
posix_trace_event()	pselect()	raise()	read()
readlink()	recv()	recvfrom()	recvmsg()
rename()	sendto()	setgid()	setpgid()
setsid()	setsockopt()	setuid()	shutdown()

sigaction()	sigaddset()	sigdelset()	sigemptyset()
sigfillset()	sigismember()	signal()	sigpause()
sigpending()	sigprocmask()	sigqueue()	sigset()
sigsuspend()	sleep()	socket()	socketpair()
stat()	symlink()	sysconf()	tcdrain()
tcflow()	tcflush()	tcgetattr()	tcgetpgrp()
tcsendbreak()	tcsetattr()	tcsetpgrp()	time()
timer_getoverrun()	timer_gettime()	timer_settime()	times()
umask()	uname()	unlink()	utime()
wait()	waitpid()	write()	

Литература:

1. D. P. Bovet, M. Cesati, **Understanding the Linux Kernel, 3rd Edition,** O'Reilly, 2005
2. W. R. Stevens, S. A. Rago, **Advanced Programming in the UNIX® Environment: Second Edition,** Addison Wesley Professional, 2005