Сигналы, группы, сеансы

Сигналы в Unix указывают ядру, что надо прервать нормальное планирование процесса и завершить/остановить его, или, вместо продолжения выполнения процесса с места остановки, выполнить функцию – обработчик сигнала, и лишь затем продолжить выполнение основного кода.

Сигналы, предназначенные процессу, создаются (отправляются) в нескольких ситуациях: при аппаратных сбоях, при срабатывании особого таймера, при обработке спецсимволов (Ctrl C, Ctrl Z) драйвером управляющего терминала, с помощью системного вызова kill(). В зависимости от причины, отправляются сигналы разных типов. Тип сигнала обозначается целым числом (номером). В Linux сигналы нумеруются от 1 до 64. Сигнал может быть отправлен отдельной нити процесса, процессу в целом или группе процессов.

Для каждого номера сигнала в процессе (нити) предусмотрено некоторое действие (действие по умолчанию, игнорирование или вызов пользовательской функции). Доставка сигнала заключается в выполнении этого действия.

Между отправкой сигнала и его доставкой проходит некоторое непредсказуемое время, поскольку, обычно, сигналы обрабатываются (доставляются) при выходе из системных вызовов или в тот момент, когда планировщик назначает процесс на выполнение. Исключением является доставка сигнала SIGKILL остановленным процессам. Для большинства сигналов можно явно приостановить доставку с помощью установки маски блокирования доставки `sigprocmask()`, и, в случае необходимости, удалить сигнал без обработки с помощью `sigwait()`.

Сигналы SIGKILL и SIGSTOP не могут быть заблокированы или проигнорированы и на них нельзя установить свой обработчик.

Действия по умолчанию:

- SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU – остановка процесса
- SIGCONT – запуск остановленного процесса
- SIGCHLD – игнорируется
- SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGFPE, SIGSEGV – сохранение дампа памяти и завершение (Linux)
- остальные – завершение процесса

Ссылка: ПРАВИЛА ИГРЫ В СИГНАЛЫ UNIX

Сигнал, маска, обработчик

В упрощенном виде структуру в ядре, обеспечивающую доставку сигналов процессу, можно представить как таблицу:

Номер	Есть Сигнал?	Маска	Обработчик
1	1	1	SIG_DFL
2	0	0	SIG_IGN
3	0	0	sig_func()
...	...	...	...

- Есть сигнал? – битовое поле, указывающее наличие недоставленного сигнала
- Маска – битовое поле, указывающее временный запрет на доставку сигнала
- Обработчик – указатель на действие, выполняемое при доставке сигнала. Может принимать значения: SIG_DFL – действие по умолчанию, SIG_IGN – игнорирование сигнала или указатель на функцию – обработчика сигнала.

Традиционно, при отправке процессу нескольких однотипных обычных сигналов, обработчик будет вызван лишь раз. Начиная с POSIX 1003.1, кроме обычных сигналов, поддерживаются сигналы реального времени, для которых создаётся очередь недоставленных сигналов, которая кроме номера сигнала, содержит значение (целое или адрес), которое уникально для каждого экземпляра сигнала.

Примеры использования сигналов

SIGKILL, SIGTERM, SIGINT, SIGHUP – завершение процесса. SIGKILL – не может быть проигнорирован, остальные могут. SIGTERM – оповещение служб о завершении работы ОС, SIGINT – завершение программы по нажатию Ctrl C, SIGHUP – оповещение программ, запущенных через модемное соединение, об обрыве связи (в настоящее время практически не используется).

SIGILL, SIGFPE, SIGBUS, SIGSEGV – аппаратный сбой. SIGILL – недопустимая инструкция CPU, SIGFPE – ошибка вычислений с плавающей точкой (деление на ноль), SIGBUS – физический сбой памяти, SIGSEGV – попытка доступа к несуществующим (защищенным) адресам памяти.

SIGSTOP, SIGCONT – приостановка и продолжение выполнения процесса

SIGPIPE – попытка записи в канал или сокет, у которого нет читателя

SIGCHLD – оповещение о завершении дочернего процесса.

Список сигналов

Особый сигнал с номером 0 фактически не доставляется, а используется в вызове kill() для проверки возможности доставки сигнала определённому процессу.

В Linux используется 64 сигнала. Список можно посмотреть в терминале командой `kill -l`

Posix.1-1990

Источник – `man 7 signal`

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

Отправка сигнала

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

Для отправка сигнала необходимо, чтобы uid или euid текущего процесса был равен 0 или совпадал с uid процесса получателя.

Получатель сигнала зависит от величины и знака параметра `pid`:

- pid > 0 – сигнал отправляется конкретному процессу
- pid == 0 – сигнал отправляется всем членам группы
- pid == -1 – сигнал отправляется всем процессам кроме init (в Linux'е еще кроме себя)
- pid < -1 – сигнал отправляется группе с номером -pid

Если signum == 0 – сигнал не посылается, но делается проверка прав на посылку сигнала и формируются код ответа и errno.

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

Аналогично kill(), но выполняется по правилам сигналов реального времени. Сигнал и связанная с ним дополнительная информация (целое или адрес) value помещаются в очередь сигналов (FIFO). Таким образом процессу можно отправить несколько однотипных сигналов с разными значениями value.

```
raise(int signum);
```

Отправка сигнала текущему процессу. Эквивалент kill(getpid(),signum);

```
abort();
```

Убирает из маски сигналов сигнала SIGABRT и отправляет его текущему процессу . Если сигнал перехватывается или игнорируется, то после возвращения из обработчика abort() завершает программу. Выхода из функции abort() не предусмотрено. Единственный способ продолжить выполнение – не выходить из обработчика сигнала.

```
alarm(time);
```

Отправка сигнала SIGALRM себе через time секунд. Возвращает 0 если ранее alarm не был установлен или число секунд остававшихся до срабатывания предыдущего alarma. Таймер управляющий alarm'ами один. Соответственно установка нового alarm'а отменяет старый. Параметр time==0 позволяет получить оставшееся до alarm'а время без установки нового.

Исторический способ обработки сигнала – signal

Установка реакции на сигнал через функцию signal() не до конца стандартизована и сохраняется для совместимости с историческими версиями Unix. Не рекомендуется к использованию. В стандарте POSIX signal() заменен на вызов sigaction(), сохраняющий для совместимости эмуляцию поведения signal().

```
signal(SIGINT,sighandler);
```

sighandler – адрес функции обработчика void sighandler(int) или один из двух макросов: **SIG_DFL** (обработчик по умолчанию) или **SIG_IGN** (игнорирование сигнала).

signal(...) возвращает предыдущее значение обработчика или SIG_ERR в случае ошибки.

В Linux и SysV при вызове sighandler обработчик сбрасывается в SIG_DFL и возможна доставка нового сигнала во время работы sighandler. Такое поведение заставляет первой строкой в sighandler восстанавливать себя в качестве обработчика сигнала, и, даже в этом случае, не гарантирует от вызова обработчика по умолчанию. В BSD системах сброс не происходит, доставка новых сигнала блокируется до выхода из sighandler.

Пример кода:

```
#include <signal.h>
void sighandler(int signum) {
   signal(signum,sighandler);
   ...
}

main() {
      signal(SIGINT,sighandler);
      signal(SIUSR1,SIG_IGN);
      signal(SIUSR2,SIG_DFL);
}
```

Реакция на сигнал – sigaction

Параметры для установки обработчика сигнала через sigaction()

```
struct sigaction {
   void (*sa_handler)(int);                      // Обработчик сигнала старого стиля
   void (*sa_sigaction)(int, siginfo_t *, void *); // Обработчик сигнала нового стиля
                                                 // Обработчик выбирается на основе флага SA_SIGINFO
                                                 // в поле sa_flags
   sigset_t sa_mask; // Маска блокируемых сигналов
   int sa_flags; // Набор флагов
        //  SA_RESETHAND – сброс обработчика на SIG_DFL после выхода из назначенного обработчика.
        //  SA_RESTART – восстановление прерванных системных вызовов после выхода из обработчика.
        //  SA_SIGINFO – вызов sa_sigaction вместо sa_handler

  void (*sa_restorer)(void); // Устаревшее поле
}
```

Данные, передаваемые в обработчик сигнала sa_sigaction()

```
siginfo_t {
  int      si_signo;  // Номер сигнала
  int      si_code;   // Способ отправки сигнала или уточняющее значение
                      // SI_USER сигнал отправлен через вызов kill()
                      // SI_QUEUE сигнал отправлен через вызов sigqueue()
                      // FPE_FLTDIV – уточнение для сигнала SIGFPE – деление на ноль
                      // ILL_ILLOPC – уточнение для сигнала SIGILL – недопустимый опкод
                      // ...
  pid_t    si_pid;    // PID процесса отправителя (дочернего процесса при SIGCHLD)
  uid_t    si_uid;    // UID процесса отправителя
  int      si_status; // Статус завершения дочернего процесса при SIGCHLD
  sigval_t si_value;  // Значение, переданое через параметр value при вызове sigqueue()
  void *   si_addr;   // Адрес в памяти
                      // SIGILL, SIGFPE – адрес сбойной инструкции
                      // SIGSEGV, SIGBUS – адрес сбойной памяти
}
```

В Linux структура `siginfo_t` содержит больше полей, но они служат для совместимости и не заполняются.

Код

```
#include <signal.h>
void myhandler(int sig) {
   ...
}
void myaction(int signum, siginfo_t * siginfo, void *code)  {
...
}

main() {
struct sigaction act,oldact;
    act.sa_sigaction=myaction;
    act.sa_flags=SA_SIGINFO;
    sigaction(signum, &act, &oldact);
}
```

Блокирование сигналов

Все функции блокирования сигналов работают с маской сигналов типа sigset_t. В Linux это 64 бита, т.е. два слова в 32х разрядной архитектуре или одно в 64х разрядной. Выставленный бит в маске сигналов означает, что доставка сигнала с соответствующим номером будет заблокирована. Сигналы SIGKILL и SIGSTOP заблокировать нельзя.

Манипулирование маской

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset); Изменить маску сигналов
```

Параметр how определяет операцию над текущей маской. Значения SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK.

Проверка наличия заблокированных сигналов

```
int sigpending(sigset_t *set); // Получить список недоставленных из-за блокировки сигналов.
int sigsuspend(const sigset_t *mask);  //Установить новую маску и "уснуть" до получения и обработки разрешенного в ней сигнала.
```

Очистка заблокированных сигналов

sigwait ожидает блокировки какого-либо из сигналов, указанных в маске, удаляет этот сигнал и возвращает его номер в параметр sig. Если реализация сигналов предусматривает очередь сигналов, то удаляется только один элемент из очереди.

```
int sigwait(const sigset_t *set, int *sig);
```

Управляющий терминал, сеанс, группы

Управляющий терминал, сеанс, группы

Для организации диалоговой работы пользователей в Unix вводится понятие терминальной сессии. С точки зрения пользователя – это процесс работы с текстовым терминалом с момента ввода имени и пароля и до выхода из системы командой logout (exit, нажатие ^D в пустой строке). Во время терминальной сессии может быть запущено несколько программ, которые будут параллельно выполнятся в фоновом режиме и между которыми можно переключаться в диалоговом режиме. После завершения терминальной сессии возможно принудительное завершение всех запущенных в ней фоновых процессов.

С точки зрения ядра – терминальная сессия – это группа процессов, имеющих один идентификатор сеанса **sid**. С идентификатором **sid** связан драйвер управляющего терминала, доступный всем членам сеанса как файл символьного устройства **/dev/tty**. Для каждого сеанса существует свой **/dev/tty**. Управляющий терминал взаимодействует с процессами сеанса с помощью отправки сигналов.

В рамках одного сеанса могут существовать несколько групп процессов. С каждым процессом связан идентификатор группы **pgid**. Одна из групп в сеансе может быть зарегистрирована в драйвере управляющего терминала как группа фоновых процессов. Процессы могут переходить из группы в группу самостоятельно или переводить из группы в группу другие процессы сеанса. Перейти в группу другого сеанса нельзя, но можно создать свой собственный сеанс из одного процесса со своей группой в этом сеансе. Вернуться в предыдущий сеанс уже не получится.

Группа процессов

Группа процессов – инструмент для доставки сигнала нескольким процессам, а также способ арбитража при доступе к терминалу. Идентификатор группы **pgid** равен **pid** создавшего её процесса – лидера группы. Процесс может переходить из группы в группу внутри одного сеанса.

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid); // включить процесс pid в группу pgid.
                                            // pid=0 означает текущий процесс,
                                            // pgid=0 означает pgid=pid текущего процесса
                                            // pid=pgid=0 – создание новой группы с pgid=pid текущего процесса
                                            //                       и переход в эту группу
pid_t getpgid(pid_t pid); // получить номер группы процесса pid.
                          // pid=0 – текущий процесс
int setpgrp(void); // создание группы, эквивалент setpgid(0.0);
pid_t getpgrp(void); // запрос текущей группы, эквивалент getpgid(0);
```

Сеанс

Сеанс – средство для контроля путем посылки сигналов над несколькими группами процессов со стороны терминального драйвера. Как правило соответствует диалоговой пользовательской сессии. Идентификатор сеанса **sid** равняется идентификатору **pid**, создавшего его процесса – лидера сеанса. Одновременно с сеансом создаётся новая группа с **pgid** равным **pid** лидера сеанса. Поскольку переход группы из сеанса в сеанс невозможен, то создающий сеанс процесс не может быть лидером группы.

```
#include <unistd.h>
pid_t setsid(void); //Создание  новой группы и нового сеанса. Текущий процесс не должен быть лидером группы.
pid_t getsid(pid_t pid); //Возвращает номер сеанса для указанного процесса
```

Создание собственного сеанса рекомендуется начать с fork, чтобы гарантировать, что процесс не является лидером группы.

```
if( fork() ) exit(0);
setsid();
```

Фоновая группа сеанса

Процессы в фоновой группе выполняются до тех пор, пока не попытаются осуществить чтение или запись через файловый дескриптор управляющего терминала. В этот момент они получают сигнал SIGTTIN или SIGTTOU соответственно. Действие по умолчанию для данного сигнала – приостановка выполнения процесса.

Назначение фоновой группы:

```
#include <unistd.h>

 pid_t tcgetpgrp(int fd); // получить pgid фоновой группы, связанной с управляющим терминалом,
                     // на который ссылается файловый дескриптор fd
 int tcsetpgrp(int fd, pid_t pgrp); // назначить pgid фоновой группы терминалу,
                    // на который ссылается файловый дескриптор fd
```

Управляющий терминал

Некоторые сочетания клавиш позволяют посылать сигналы процессам сеанса:

- **^C** – SIGINT – завершение работы
- **^Z** – SIGTSTP – приостановка выполнения. `bash` отслеживает остановку дочерних процессов и вносит их в списки своих фоновых процессов. Остановленный процесс может быть продолжен командой `fg n`, где **n** – порядковый нрмер процесса в списке фоновых процессов

Открыть управляющий терминал сеанса

```
#include <stdio.h>

char name[L_ctermid];
int fd;
ctermid(name); // если name=NULL, то используется внутренний буфер
           // ctermid возвращает указатель на буфер
           // L_ctermid – библиотечная константа
fd = open(name, O_RDWR, 0);
```