Лабораторное занятие №10 «Процессы. Сигналы»

На данном занятии следует познакомится с сигналами — традиционным механизмом общения ядра и пользователя, который может быть использован для организации простейшего межпроцессного взаимодействия.

Основные задания

Задание №1

Напишите программу, в которой определите пользовательскую реакцию процесса на сигналы SIGINT и SIGTERM (вывод информации о захвате соответствующего сигнала), восстановите поведение по умолчанию для сигнала SIGPROF и игнорируйте сигнал SIGHUP. Для задержки выполнения процесса используйте функцию pause() в бесконечном цикле. Убедитесь в работоспособности программы. Реализуйте две версии программы: с помощью signal и с помощью sigaction.

Задание №2

Напишите программу, в которой создаются два процесса (родительский и дочерний). Эти процессы должны поочередно посылать сообщения в стандартный поток вывода. Они синхронизируют свою работу, посылая друг другу сигнал SIGUSR1 при помощи вызова kill.

Задание №3

С помощью функций alarm и pause напишите программу, которая при помощи командной строки получает интервал времени (в минутах) и текстовое сообщение, завершает основной процесс и через заданное время выводит в стандартный поток вывода заданный текст.

Задание №4

Напишите программу, в которой создаются два процесса (родительский и дочерний). Сначала родительский процесс заданное количество раз посылает дочернему процессу сигнал SIGUSR1 с дополнительной информацией (номер вызова). Дочерний процесс обрабатывает сигнал и выводит номер, текстовое представление сигнала и дополнительную информацию в стандартный поток вывода. Затем основной процесс посылает дочернему сигнал SIGTERM, завершающие его работу, ожидает завершение дочернего процесса и завершает работу сам.

Рекомендации по выполнению

Сигналы — это программные прерывания, предназначенные для обработки асинхронных событий. Такие события могут происходить либо в результате действий пользователя (например, ввод пользователем комбинации Ctrl+C), либо в результате некоторых действий в программе или ядре ОС. Сигналы можно рассматривать как простую форму *взаимодействия между процессами* (*interprocess communication*, *IPC*) — один процесс также может отправлять сигналы другому процессу.

Сигналы проходят определенный жизненный цикл. Сначала сигнал *отправляется* (*send,* или *генерируется — generate*). После этого ядро *сохраняет* (*store*) сигнал до тех пор, пока не появится возможность доставить его по назначению. Затем, после того, как сигнал

доставлен, ядро соответствующим образом *обрабатывает* (*handle*) его. При этом, ядро может выполнить одно из трех действий:

- *Игнорировать сигнал*
  Не предпринимаются никакие действия. Существуют два сигнала, которые не могут быть проигнорированы: SIGKILL и SIGST0P.
- *Захватить и обработать сигнал*
  Ядро приостанавливает исполнение текущего процесса и переходит к выполнению ранее зарегистрированной функции. После этого процесс возвращается туда, где находился на момент захвата сигнала. Чаще всего захватываются сигналы SIGINT и SIGTERM. Сигналы SIGKILL и SIGST0P захватить невозможно.
- *Выполнить действие по умолчанию*
  Это действие зависит от того, какой сигнал отправляется. Действием по умолчанию часто бывает либо завершение процесса, либо игнорирование сигнала.

Раньше, когда сигнал доставлялся, у функции, которая обрабатывала сигнал, не было никакой информации о том, что произошло, она знала только, что был создан определенный сигнал. Сегодня ядро умеет предоставлять программистам, заинтересованным в этом, обширный контекст, и сигналы даже могут передавать пользовательские данные — так работают современные более продвинутые механизмы взаимодействия между процессорами.

Идентификаторы сигналов

У каждого сигнала есть символическое имя, начинающееся с префикса SIG. Например, SIGINT — это сигнал, который отправляется, когда пользователь нажимает клавишное сочетание Ctrl+C, SIGABRT — это сигнал, отправляемый, когда процесс вызывает функцию abort(), а SIGKILL — сигнал, отправляемый, когда процесс принудительно завершается.

Все сигналы определены в заголовочном файле <signal.h>. Это по-сути определения препроцессора, ставящие в соответствие именам положительные целые числа. Номера сигналов начинаются с единицы (обычно единице соответствует SIGHUP) и непрерывно возрастают. Вывести весь список сигналов, поддерживаемых в вашей системе, можно при помощи команды kill -l. Сигналов достаточно много (в разных системах их может быть разное количество), но в большинстве программ регулярно используется лишь некоторое количество из них. Сигнала со значением 0 нет — это специальное значение, называемое *нулевым сигналом* (*null signal*). У него нет отдельного имени; некоторые системные вызовы применяют значение 0 в специальных случаях.

Простое управление сигналами

Самый простой и самый старый интерфейс для управления сигналами — это функция signal():

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signo, sighandler_t handler);
```

Если вызов функции signal() закончился успешно, то удаляется текущее действие, предпринимаемое при получении сигнала signo, и вместо этого сигнал обрабатывается обработчиком, указанным с помощью аргумента handler. В качестве значения аргумента signo можно указать как символическое имя сигнала, та и его целочисленный эквивалент. Так как процесс не может захватывать сигналы SIGKILL, и SIGST0P, то установка

обработчика для любого из этих сигналов не имеет смысла.

Функция handler возвращает значение void и принимает один аргумент — целочисленное значение, представляющее собой идентификатор обрабатываемого сигнала (например, SIGUSR1). Таким образом, с помощью одной функции-обработчика можно обрабатывать несколько сигналов. Функция-обработчик должна иметь такой вид:

```
void my_handler(int signo);
```

В *Linux* для этого есть специальный тип sighandler_t.

Когда ядро генерирует сигнал для процесса, который зарегистрировал обработчик сигнала, оно приостанавливает выполнение обычного потока инструкций программы и вызывает обработчик сигнала. Обработчику передается значение сигнала, содержащееся в аргументе signo, первоначально предоставленном вызову signal().

Кроме того, вызов функции signal() можно использовать для того, чтобы заставить ядро игнорировать определенный сигнал для текущего процесса или восстановить повеление по умолчанию для этого сигнала. Это делается при помощи специальных значений параметра handler:

SIG_DFL

Восстановить поведение по умолчанию для сигнала, указанного при помощи аргумента signo.

SIG_IGN

Игнорировать сигнал, указанный при помощи параметра signo.

Функция signal() возвращает предыдущее поведение сигнала, которое может принимать вид указателя на обработчик сигнала, SIG_DFL или SIG_IGN. В случае ошибки она возвращает значение SIG_ERR. Эта функция не устанавливает переменную errno.

Ожидание сигнала

Функция alarm позволяет установить таймер, по истечении периода времени которого будет сгенерирован сигнал SIGALRM. Если этот сигнал не игнорируется и не перехватывается приложением, он вызывает завершение процесса.

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

Функция возвращает либо 0, либо количество секунд до истечения периода времени, установленного ранее

Аргумент seconds определяет время (количество секунд), через которое должен быть сгенерирован сигнал SIGALRM.

Вызов alarm не приостанавливает выполнение процесса, как вызов sleep, вместо этого сразу же происходит возврат из вызова alarm, и продолжается нормальное выполнение процесса до тех пор, пока не будет получен сигнал SIGALRM.

Вызовы alarm не накапливаются. Если вызвать alarm дважды, то второй вызов отменяет предыдущий. Таким образом, если функция alarm вызывается до истечения таймера, установленного ранее, то она возвращает количество оставшихся секунд, а ранее установленный интервал времени заменяется новым.

Выключить таймер можно при помощи вызова с нулевым параметром: alarm(0) - работающий таймер останавливается, а функция возвращает количество секунд, оставшихся до истечения периода времени таймера.

Действие сигнала SIGALRM по умолчанию заключается в завершении процесса, но большинство приложений перехватывают его. Если в результате получения этого сигнала

приложение должно завершить работу, оно может выполнить все необходимые заключительные операции перед выходом. Если предполагается перехват сигнала SIGALRM, то необходимо установить обработчик сигнала до того, как будет вызвана функции alarm. Если функция alarm будет вызвана первой и при этом успеет сгенерировать сигнал до установки обработчика сигнала, то процесс завершится.

Функция pause приостанавливает вызывающий процесс до тех пор, пока не будет перехвачен какой-либо сигнал.

```
#include <unistd.h>
int pause(void);
```

Вызов pause приостанавливает выполнение вызывающего процесса (так что процесс при этом не занимает процессорного времени) до получения любого сигнала. Если сигнал вызывает нормальное завершение процесса или игнорируется процессом, то в результате вызова pause будет просто выполнено соответствующее действие (завершение работы или игнорирование сигнала). Если же сигнал перехватывается, то после вызова соответствующего обработчика сигнала вызов pause вернет значение (-1) и поместит в в переменную errno значение EINTR.

Пример

В приведенном ниже примере мы регистрируем один обработчик для сигналов SIGTERM и SIGINT. Кроме того, восстанавливаем поведение по умолчанию для сигнала SIGPR0F (прекращение процесса) и игнорируем SIGHUP (который также завершает процесс):

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

/* обработчик для SIGINT */
static void signal_handler(int signo) {
	if (signo == SIGINT)
		printf("Захвачен сигнал SIGINT!\n");
	else if (signo == SIGTERM)
		printf("Захвачен сигнал SIGTERM!\n");
	else {
		/* это никогда не должно случаться */
		fprintf(stderr, "Неожиданный сигнал!\n");
		exit(EXIT_FAILURE);
	}
	exit(EXIT_SUCCESS);
}

int main (void) {
	/*
	*  Регистрируем signal_handler нак наш обработчик сигнала
	*  для SIGINT. */
	if (signal(SIGINT, signal_handler) == SIG_ERR) {
		fprintf(stderr, "Невозможно обработать SIGINT!\n");
		exit(EXIT_FAILURE);
```

```
        }

        /*
        *  Регистрируем signal_handler как наш обработчик сигнала
        *  для SIGTERM. */
        if (signal(SIGTERM, signal_handler) == SIG_ERR) {
                fprintf(stderr, "Невозможно обработать SIGTERM!\n");
                exit(EXIT_FAILURE);
        }

        /* Восстановление поведения по умолчание для сигнала SIGPROF. */
        if (signal(SIGPROF, SIG_DFL) == SIG_ERR)  {
                fprintf(stderr, "Невозможно сбросить SIGPROF!\n").
                exit(EXIT_FAILURE);
        }

        /* Игнорировать SIGHUP. */
        if (signal(SIGHUP, SIG_IGN) == SIG_ERR) {
                fprintf(stderr, "Невозможно игнорировать SlGHUP!\n");
                exit(EXIT_FAILURE);
        }

        for (;;)
                pause();

        return 0;
}
```

Сопоставление номеров сигналов и строк

Иногда бывает необходимо преобразовать номер сигнала в его строковое. Есть несколько способов сделать это. Один из них - извлечь строку из статически определенного списка:

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

sys_siglist — это массив строк, содержащих имена поддерживающихся системой сигналов, проиндексированный по номерам сигналов.

Кроме того, можно использовать определенный в *BSD* интерфейс psignal(). Этот интерфейс часто поддерживается в *Linux*:

```
#include <signal.h>
void psignal(int signo, const char *msg);
```

Вызов psignal() выводит в поток stderr строку, указанную при помощи аргумента msg, за которой следуют двоеточие, пробел и имя сигнала, соответствующего значению аргумента signo. Если значение signo недопустимо, то об этом будет говориться в сообщении.

Можно еще использовать интерфейс strsignal(). Он не является стандартным, но многие версии *Linux* поддерживают его:

```
#define _GNU_SOURCE
#include <string.h>
char *strsignal(int signo);
```

Вызов strsignal() возвращает указатель на описание сигнала, указанного при помощи signo. Если аргумент signo получил недопустимое значение, то об этом будет сказано в возвращаемом описании. Возвращаемая строка действительна только до следующего вызова strsignal(), поэтому данную функцию небезопасно использовать с потоками.

Таким образом, рекомендуется использовать массив sys_siglist. С его помощью можно написать такой обработчик сигналов:

```
static void signal_handler(int signo) {
        printf("Захвачен сигнал %s\n", sys_siglist[signo]);
}
```

Отправка сигнала

Для отправки согнала можно воспользоваться традиционным средством — системным вызовом kill(). Именно этот вызов лежит в основе утилиты kill, отправляющей сигнал от одного процесса другому:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int signo);
```

Первый аргумент (pid) является идентификатором процесса, которому отправляется сигнал, определяемый вторым аргументов (signo).

При вызове, когда значение pid больше 0, функция kill() отправляет сигнал signo процессу, указанному при помощи идентификатора pid.

Если аргумент pid равен 0, то сигнал signo отправляется всем процессам в группе процессов вызывающего процесса.

Если аргумент pid равен -1, то signo отправляется всем процессам, для которых у вызывающего процесса есть разрешение на отправку сигнала, за исключением самого себя и процесса init.

Если аргумент pid меньше -1, то signo отправляется группе процессов -pid.

В случае успешного завершения функция kill() возвращает 0. Вызов считается успешным, если хотя бы один сигнал был отправлен. В случае сбоя (ни одного сигнала не отправлено) вызов возвращает значение -1 и присваивает переменной errno один из следующих кодов:

EINVAL

Значение signo представляет недопустимый сигнал.

EPERM

У вызывающего процесса недостаточно полномочий для отправки сигнала какому-либо из запрошенных процессов.

ESRCH

Процесс или группа процессов, указанная при помощи pid, не существует или в случае процесса? является зомби.

Следует отметить, что обычный пользователь *Linux* может отправить сигнал только процессу, владельцем которого он является.

Если значение signo равно 0, то такой вызов не отправляет сигнал, а выполняет

проверку ошибок. Таким образом, позволяет протестировать, обладает ли процесс необходимыми разрешениями для отправки сигнала определенному процессу или процессам, и существуют ли эти процессы.

Пример отправки сигнала SIGHUP процессу с идентификатором процесса 1722:

```
int ret;
ret = kill(1722, SIGHUP);
if (ret) perror("kill");
```

Пример проверки, есть ли разрешение на отправку сигнала процессу с идентификатором 1722:

```
int ret;
ret = kill(1722, 0);
if (ret)
        : /* у нас нет разрешения */
else
        : /* у нас есть разрешение */
```

Отправка сигнала себе

При помощи функции raise() — процесс может отправить сигнал самому себе:

```
#include <signal.h>
int raise(int signo);
```

Этот вызов raise(signo); эквивалентен такому вызову kill(getpid( ), signo);

Функция возвращает значение 0 в случае успешного вызова и ненулевое значение в случае ошибки. Она не устанавливает переменную errno.

Отправка сигнала всей группе процессов

Существует функция, упрощающая отправку сигнала всем процессам из одной группы:

```
#include <signal.h>
int killpg (int pgrp, int signo);
```

Этот вызов killpg(pgrp, signo); эквивалентен следующему вызову: kill(-pgrp, signo);

Если значение pgrp равно 0, то killpg() отправит сигнал signo каждому процессу в группе вызывающего процесса.

В случае успешного завершения функция killpg() возвращает 0. В случае ошибки этот будет возвращено значение -1 и переменной errno будет присваивает одно из следующих значений:

EINVAL

Значение signo представляет недопустимый сигнал.

EPERM

У вызывающего процесса отсутствуют разрешения на отправку сигнала какому-либо

из запрошенных процессов.

ESRCH
Группа процессов, указанная при помощи аргумента pgrp, не существует.

Набор сигналов

Существуют функции, работающие не с одним сигналом, а с целым набором:

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
```

sigemptyset() инициализирует набор сигналов set, помечая его как пустой (в набор не включено ни одного сигнала).

sigfillset() инициализирует набор сигналов set, помечая его как полный (в набор включены все сигналы).

Обе функции в случае успешного завершения возвращают значение 0. Перед тем, как использовать набор сигналов sigset_t set его необходимо инициализировать при помощи одной из этих двух функций.

sigaddset() добавляет сигнал signo в набор сигналов, указанный при помощи аргумента set, а sigdelset() удаляет signo из набора сигналов set. Обе функции возвращают 0 в случае успешного завершения и -1 в случае ошибки. Кроме того, в случае ошибки переменной errno присваивается код ошибки EINVAL, указывающий, что signo содержит недопустимый идентификатор сигнала.

sigismember() возвращает 1, если signo находится в наборе сигналов, указанном при помощи аргумента set, и 0, если это не так. В случае ошибки возвращается значение и -1. При этом переменной errno присваивается значение EINVAL, указывающее на недопустимое значение signo.

Блокировка сигналов

При выполнения программы возможны такие ситуации, когда нежелательно прерывать ее выполнение даже обработкой сигналов. Такие части программы обычно называют *критическими областями* (*critical region*) и защищают их, временно приостанавливая доставку сигналов. При этом считается, что такие сигналы заблокированы. Все сигналы, сгенерированные во время блокировки, не обрабатываются до тех пор, пока не разблокируются. Процесс может заблокировать любое количество сигналов; набор сигналов, заблокированных процессом, называется его *сигнальной маской* (*signal mask*).

В *Linux* существует функция для управления сигнальной маской процесса:

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

Поведение sigprocmask() зависит от значения аргумента how, который может содержать один из следующих флагов:

SIG_SETMASK
Сигнальная маска для вызывающего процесса меняется на set.

SIG_BLOCK

Сигналы из set добавляются к сигнальной маске вызывающего процесса. Таким образом, сигнальная маска меняется на объединение (двоичное *ИЛИ*) текущей маски и set.
SIG_UNBLOCK

Сигналы из set удаляются из сигнальной маски вызывающего процесса. Таким образом, сигнальная маска меняется на пересечение (двоичное *И*) текущей маски и отрицания (двоичное *НЕ*) набора set. Невозможно разблокировать сигнал, который не был заблокирован.

Если значение oldset не равно NULL, то функция помещает предыдущую сигнальную маску в oldset.

Если значение set равно NULL, то функция игнорирует how и не меняет сигнальную маску, а помещает ее в аргумент oldset. Таким образом, передача нулевого значения в качестве set — это способ извлечь текущую сигнальную маску.

В случае успешного завершения вызов функции возвращает значение 0. В случае неудачи вызов возвращает -1 и присваивает переменной errno либо код EINVAL, указывая на недопустимое значение how, либо код EFAULT, указывая, что set или oldset содержит недопустимый указатель.

Блокировка сигналов SIGKILL и SIGSTOP не допускается. Функция sigprocmask() просто игнорирует любые попытки добавить любой из этих сигналов в сигнальную маску.

Извлечение ожидающих сигналов

Если ядро генерирует заблокированный сигнал, он не доставляется процессу. Такие сигналы называются *ожидающими*. Когда ожидающий сигнал разблокируется, ядро передает его процессу для обработки.

Существует функция, позволяющая получить такой набор ожидающих сигналов:

```
#include <signal.h>
int sigpending(sigset_t *set);
```

Успешный вызов функции sigpending() помещает набор ожидающих сигналов в аргумент set и возвращает значение 0. В случае ошибки вызов возвращает -1 и присваивает переменной errno значение EFAULT, указывая, что set содержит недопустимый указатель.

Ожидание набора сигналов

Существует функция, позволяющая процессу временно менять свою сигнальную маску, а затем ждать, пока не будет сгенерирован сигнал, который либо завершит этот процесс, либо будет этим процессом обработан:

```
#include <signal.h>
int sigsuspend(const sigset_t *set);
```

Если сигнал завершает процесс, то sigsuspend() не возвращает результата. Если сигнал генерируется и обрабатывается, то sigsuspend() возвращает -1 и, после того как обработчик сигнала возвратит результат, присваивает переменной errno значение EINTR. Если set содержит недопустимый указатель, то переменной errno присваивается код EFAULT.

Обычно функция sigsuspend() применяется для извлечения сигналов, которые могли прибыть и оказаться заблокированными во время выполнения критической области программы. Сначала процесс при помощи sigprocmask() блокирует набор сигналов, сохраняя старую маску в oldset. После выхода из критической области процесс вызывает sigsuspend(), предоставляя oldset в качестве значения set.

Расширенное управление сигналами

В качестве альтернативы простой функции signal(), в *POSIX* определен системный вызов sigaction(), предоставляющий гораздо больше возможностей по управлению сигналами. Среди прочего, его можно применять для блокировки определенных сигналов во время выполнения обработчика, а также для получения большого количества данных о системе и состоянии процесса на момент, когда был сгенерирован сигнал:

```
#include <signal.h>
int sigaction(int signo, const struct sigaction *act, struct sigaction *oldact);
```

Вызов sigaction() изменяет поведение сигнала, указанного при помощи аргумента signo. Этот аргумент может принимать любые значения, за исключением тех, которые ассоциируются с SIGKILL и SIGSTOP. Если значение act не равно NULL, то системный вызов изменяет текущее поведение сигнала в соответствии со значением параметра act. Если значение oldact не равно NULL, то вызов сохраняет предыдущее (или текущее, если аргумент act равен NULL) поведение указанного сигнала в структуре oldact.

Одноименная структура sigaction позволяет организовать тонкое управление сигналами. В заголовочном файле <sys/signal.h>, который подключается через <signal.h>, эта структура определяется так:

```
struct sigaction {
        void (*sa_handler)(int);            /* обработчик сигнала или действие */
        void (*sa_sigaction)(int, siginfo_t *, void *);
        sigset_t sa_mask;                   /* блокируемые сигналы */
        int sa_flags;                       /* флаги */
        void (*sa_restorer)(void);          /* поле для внутренних нужд */
}
```

В поле sa_handler определяется действие, которое должно выполняться при получении сигнала. Как и для вызова signal(), данное поле может содержать флаг SIG_DFL, обозначающий действие по умолчанию, флаг SIG_IGN, приказывающий ядру игнорировать сигнал для процесса, или указатель на функцию обработки сигнала. У функции тот же прототип, что и у обработчика сигналов, устанавливаемого signal():

```
void my_handler(int signo);
```

Однако, если в поле sa_flags установлен флаг SA_SIGINFO, то тогда поле sa_sigaction, а не sa_handler определяет используемую функцию обработки сигналов с другим прототипом:

```
void my_handler(int signo, siginfo_t *si, void *ucontext);
```

Данная функция получает в качестве первого параметра номер сигнала, в качестве второго — структуру siginfo_t и в качестве третьего — структуру ucontext_t (приведенную к указателю void). Она не возвращает значения. Структура siginfo_t предоставляет большое количество информации для обработчика сигналов; мы рассмотрим ее чуть позже.

В поле sa_mask находится набор сигналов, которые система должна блокировать на время выполнения обработчика сигналов. Сигнал, который в данный момент обрабатывается, также блокируется, если только в sa_flags не содержится флаг SA_NODEFER. Как обычно, невозможно заблокировать сигналы SIGKILL или SIGSTOP; вызов просто игнорирует любое

из этих значений, даже если оно добавляется в sa_mask.

Поле sa_flags — это битовая маска, включающая ноль, один или несколько флагов, объединенных битовой операцией *ИЛИ* (|), изменяющих способ обработки сигнала, указанного в signo. Флаги SA_SIGINFO и SA_NODEFER уже обсуждались. Рассмотрим некоторые из оставшихся значения поля sa_flags (полный список лучше посмотреть в справочной системе):

SA_NOCLDSTOP

Если значение signo равно SIGCHLD, то данный флаг заставляет систему не отправлять сигнал, когда дочерний процесс останавливается или возобновляется.

SA_NOCLDWAIT

Если значение signo равно SIGCHLD, то данный флаг включает *автоматическое удаление потомков* (*automatic child reaping*): потомки не превращаются в зомби при завершении, при этом предку не нужно делать для них системный вызов wait().

SA_RESTART

Данный флаг включает автоматический перезапуск системных вызовов, прерванных сигналами.

SA_RESETHAND

Данный флаг включает «одноразовый» режим. Для указанного сигнала автоматически восстанавливается поведение по умолчанию, как только обработчик сигнала закончит работу.

Успешный вызов функции sigaction() возвращает значение 0. В случае ошибки он возвращает -1 и присваивает переменной errno один из следующих кодов ошибки:

EFAULT

Аргумент act или oldact содержит недопустимый указатель.

EINVAL

Аргумент signo содержит недопустимый сигнал, SIGKILL или SIGSTOP.

Структура siginfo_t

Структура siginfo_t также определяется в <sys/signal.h>:

```
typedef struct siginfo_t {
        int si_signo; /* номер сигнала */
        int si_errno; /* значение errno */
        int si_code; /* код сигнала */
        pid_t si_pid; /* PID отправляющего процесса */
        uid_t si_uid; /* действительный UID отправляющего процесса */
        int si_status; /* значение выхода или сигнал */
         clock_t si_utime; /* потребленное пользовательское время */
        clock_t si_stime; /* потребленное системное время */
        sigval_t si_value;   /* значение полезной нагрузки сигнала */
        int si_int;               /* сигнал POSIX.lb */
        void *si_ptr;             /* сигнал POSIX.lb */
        void *si_addr:            /* местоположение в памяти, вызвавшее сбой */
        int si_band;              /* событие полосы */
        int si_fd;                /* дескриптор файла */
};
```

Эта структура предоставляет разнообразную информацию обработчику сигналов, если вместо sa_sighandler используется sa_sigaction. В этой структуре содержится большое количество данных, включая информацию о процессе, отправившем сигнал, и о

причине сигнала. Рассмотрим описание полей:

si_signo

Номер сигнала.

si_errno

Если значение этого поля не равно нулю, то это код ошибки, связанный с сигналом. Данное поле применяется для всех сигналов.

si_code

Объяснение, почему и откуда процесс получил данный сигнал. Это поле будет рассмотрено в следующем разделе. Оно используется для всех сигналов.

si_pid

Для сигнала SIGCHLD это идентификатор *PID* процесса, который завершился.

si_uid

Для сигнала SIGCHLD это *UID* владельца процесса, который завершился.

si_status

Для сигнала SIGCHLD это статус выхода процесса, который завершился.

si_utime

Для сигнала SIGCHLD это пользовательское время, потребленное процессом, который завершился.

si_stime

Для сигнала SIGCHLD это системное время, потребленное процессом, который завершился.

si_value

Объединение si_int и si_ptr.

si_int

Для сигналов, отправленных с помощью вызова sigqueue(), это переданная дополнительная информация, указанная в виде целочисленного значения.

si_ptr

Для сигналов, отправленных с помощью вызова sigqueue(), это переданная полезная дополнительная информация, указанная в виде указателя void.

Поля si_value, si_int и si_ptr процесс может применять для передачи произвольных данных другому процессу. Таким образом, их можно использовать для отправки либо простого целочисленного значения, либо указателя на структуру данных (следует обратить внимание на то, что указатель не имеет особого смысла, если процессы не используют совместно одно адресное пространство).

Стандарт *POSIX* гарантирует, что только первые три поля используются для всех сигналов. К остальным полям следует обращаться только при обработке соответствующего сигнала.

Поле si_code

Поле si_code содержит причину сигнала. Для сигналов, отправленных пользователем, оно указывает на то, как сигнал был отправлен. Для сигналов, отправленных ядром, поле указывает, почему был отправлен сигнал. В качестве значения поле может содержит большое количество символьных констант, полный список которых лучше посмотреть в справочной системе. Рассмотрим только некоторые из них.

Следующие значения si_code допустимы для любого сигнала. Они сообщают, как/почему сигнал был отправлен:

SI_ASYNCIO

Сигнал был отправлен из-за завершения асинхронного ввода-вывода.

SI_KERNEL

Сигнал был сренерирован ядром.

SI_QUEUE
: Сигнал был отправлен sigqueue().

SI_TIMER
: Сигнал был отправлен из-за завершения таймера *POSIX*.

SI_USER
: Сигнал был отправлен вызовом kill() или raise().

Для сигнала SIGCHLD следующие значения показывают, какие действия были сделаны потомком, вследствие чего предку был отправлен сигнал:

CLD_CONTINUED
: Потомок был остановлен, но затем возобновлен.

CLD_DUMPED
: Ненормальное завершение потомка.

CLD_EXITED
: Нормальное завершение потомка благодаря вызову exit().

CLD_KILLED
: Потомок был убит.

CLD_STOPPED
: Потомок остановился.

CLD_TRAPPED
: Потомок попал в ловушку.

Отправка сигнала с дополнительной информацией

Обработчики сигналов, зарегистрированные с флагом SA_SIGINFO, получают параметр типа siginfo_t. Эта структура включает поле с именем si_value, которое может содержать необязательную дополнительную информацию, передаваемую от создателя сигнала его получателю. Функция sigqueue(), определенная в *POSIX*, позволяет процессу отправлять сигналы с такой информацией:

```
#include <signal.h>
int sigqueue(pid_t pid, int signo, const union sigval value);
```

Вызов sigqueue() работает аналогично вызову kill(). В случае успешного завершения сигнал, указанный при помощи аргумента signo, ставится в очередь к процессу или группе процессов, идентифицирующейся значением pid, а функция возвращает 0. Дополнительная информация сигнала передается с помощью параметра value, представляющего собой объединение (union) целочисленного значения и указателя void:

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

В случае ошибки вызов возвращает -1 и присваивает переменной errno одно из следующих значений:

EINVAL
: Значение аргумента signo соответствует недопустимому сигналу.

EPERM
: У вызывающего процесса отсутствуют разрешения на отправку сигналов любым из

запрошенных процессов. Разрешения, необходимые для отправки сигнала, аналогичны тем, которые требует функция kill().

ESRCH

Процесс или группа процессов, указанная при помощи аргумента pid, не существует или, в случае процесса, является зомби.

Как и в случае вызова kill(), для тестирования можно отправлять при помощи параметра signo нулевой сигнал (значение 0).

В этом примере процессу с pid 1765 отправляется сигнал SIGUSR1 с дополнительной информацией — целочисленным значения 101:

```
sigval value;
int ret;

value.sival_int = 101;

ret = sigqueue(1765, SIGUSR1, value);
if (ret) {
	fprintf(stderr, "Error while invoking sigqueue\n");
}
```

Если процесс с идентификатором 1765 обрабатывает сигнал SIGUSR1 обработчиком SA_SIGINFO, то для него значение signo равно SIGUSR1, si->si_int — 101, а si->si_code — SI_QUEUE.

В заключение следует отметить, что сигналы — это старый механизм общения ядра и пользователя. Этот механизм может служить простейшей формой взаимодействия между процессами. Следует быть осторожным при написании обработчиков сигналов, которые должны быть реентабельны. Для этого их следует делать максимально простыми и использовать только реентабельные функции и атомарные типы. Кроме того, в современных программах для управления сигналами рекомендуется применять вызовы sigaction() и sigqueue() вместо signal() и kill().