# Лабораторная работа 8. Сигналы в Linux

# Краткие теоретические сведения

### Сигналы

Сигналы - асинхронное уведомление процесса о каком-либо событии.

Это один из основных и простейших способов взаимодействия между процессами.

**Жизненный цикл сигнала:**

1. **сигнал порождается** (*отсылается* или *генерируется*).
2. **ядро сохраняет сигнал** до тех пор, пока не сможет его доставить.
3. как только появляется такая возможность, **ядро обрабатывает сигнал** требуемым образом.

Когда сигнал послан процессу, операционная система прерывает выполнение процесса, при этом, если процесс установил собственный *обработчик сигнала*, операционная система запускает этот обработчик, передав ему информацию о сигнале. Если процесс не установил обработчик, то выполняется обработчик по умолчанию.

Названия сигналов «SIG…» являются числовыми константами со значениями, определяемыми в заголовочном файле signal.h.

Утилита kill позволяет задавать сигнал как числом, так и символьным обозначением.

Спецификация сигналов включена в стандарты POSIX.

**Посылка сигналов**

* из терминала, нажатием специальных клавиш или комбинаций (*например*, нажатие Ctrl-C генерирует SIGINT, Ctrl-\ SIGQUIT, а Ctrl-Z SIGTSTP);
* ядром системы;
* одним процессом другому (или самому себе).

Ядро генерирует и отправляет процессу сигнал в ответ на ряд событий, которые могут быть вызваны самим процессом, другим процессом, прерыванием или какими-либо внешними событиями.

**Случаи отправки сигналов ядром системы:**

* при возникновении аппаратных исключений (недопустимых инструкций, нарушениях при обращении в память, системных сбоях и т. п.);
* ошибочных системных вызовах;
* для информирования о событиях ввода-вывода;

Обычно это те сигналы, которые не поддаются обработке и приводят к немедленному завершению процесса.

Способы отправки сигналов процессом:

* на программном уровне – системным вызовом kill(),
* из терминала командной оболочки shell – утилитой /bin/kill.

Сигналы не могут быть посланы завершившемуся процессу, находящемуся в состоянии «зомби».

*Процесс-получатель* может отреагировать на сигнал одним из следующих трех способов:

* Принятие (выполнение) сигнала.
* Игнорирование сигнала.
* Перехватывание и обработка сигнала.

В зависимости от требований процесса ядро может выполнить одно из трех действий:

* Игнорировать сигнал.
* Перехватить сигнал и обработать его.
* Выполнение действия, задаваемого по умолчанию.

Игнорировать сигнал.

Никаких действий не предпринимается.

Есть два сигнала, которые не могут быть проигнорированы: SIGKILL и SIGSTOP. Дело в том, что системный администратор должен иметь возможность останавливать или завершать (убивать) процессы. Если бы процесс был способен проигнорировать SIGKILL (стать «бессмертным») или SIGSTOP (стать «неудержимым»), то системный администратор лишился бы такой возможности.

Перехватить сигнал и обработать его.

Ядро приостанавливает исполнение текущего кода в процессе и переключается на функцию, которая была зарегистрирована ранее. Затем процесс выполняет эту функцию. Когда он вернется после ее выполнения, процесс вновь перейдет к выполнению той работы, которую прервал в момент получения сигнала. Особенно часто приходится отлавливать сигналы SIGINT и SIGTERM. Процессы отлавливают SIGINT, реагируя на действие пользователя, сгенерировавшего символ прерывания.

Например, терминал может перехватить этот сигнал и вернуться в основное окно с приглашением. Процессы отлавливают SIGTERM для выполнения необходимой очистки, например для отсоединения от сети или для удаления временных файлов — до завершения.

Сигналы SIGKILL и SIGSTOP нельзя перехватить.

Выполнение действия, задаваемого по умолчанию.

Это действие зависит от того, какой именно сигнал был отправлен. Зачастую стандартное действие — это завершение процесса. Например, именно так обрабатывается сигнал SIGKILL.

Тем не менее многие сигналы предоставляются для строго определенной цели и интересуют программиста лишь в конкретной ситуации. Поэтому по умолчанию подобные сигналы игнорируются, так как большинство программ в них «не заинтересованы». Ниже мы рассмотрим различные сигналы и стандартные действия, выполняемые по умолчанию при их получении.

## Командные утилиты для работы с сигналами

Синтаксическая семантика команды kill

$ kill -s SIGNAL PID

* SIGNAL — это символическая константа посылаемого сигнала,
* PID — идентификатор процесса-получателя сигнала.

### Консольное упражнение 1

Результаты выполнения отразите в скриншотах.

$ ps

…

$ yes > /dev/null &

…

$ kill -s SIGINT 1258527

$ ps

…

### Консольное упражнение 2

$ ps

…

$ yes > /dev/null &

…

$ kill -s SIGSEGV 1258584

$ ps

…

Выполните упражнение и объясните значение сигналов.

**Системный вызов** kill() объявлен в заголовочном файле signal.h

int kill (pid\_t PID, int SIGNAL);

* PID — идентификатор процесса-получателя сигнала,
* SIGNAL — это символическая константа посылаемого сигнала.

### Упражнение 1

Пример использования системного вызова *kill().*

**Программа kill1.c**

#include <signal.h>

#include <stdio.h>

#include <stdlib.h>

int main (int argc, char \*\* argv)

{

pid\_t dpid;

if (argc < 2) {

fprintf (stderr, "Too few arguments\n");

return 1;

}

dpid = atoi (argv[1]);

if (kill (dpid, SIGKILL) == -1) {

fprintf (stderr, "Cannot send signal\n");

return 1;

}

return 0;

}

**Пример kill2.c**

#include <signal.h>

#include <stdio.h>

#include <stdlib.h>

int **main** (void)

{

pid\_t dpid = getpid ();

if (kill (dpid, SIGABRT) == -1) {

fprintf (stderr, "Cannot send signal\n");

return 1;

}

return 0;

}

Выполните упражнение и объясните принцип работы программы.

## Обработка сигнала

Системный вызов sigaction() позволяет задавать поведение процесса по отношению к конкретным сигналам.

Объявлен в заголовочном файле signal.h

int sigaction (int SIGNAL, const struct sigaction \* ACTION,

struct sigaction \* OLDACTION);

Данный системный вызов устанавливает политику реагирования процесса на сигнал SIGNAL.

### Упражнение 2

Программа sigaction1.c

#include <signal.h>

#include <stdio.h>

void sig\_handler (int snum) { fprintf (stderr, "signal...\n"); }

int **main** (void)

{

struct sigaction act;

sigemptyset (&act.sa\_mask);

act.sa\_handler = &sig\_handler;

act.sa\_flags = 0;

if (sigaction (SIGINT, &act, NULL) == -1) {

fprintf (stderr, "sigaction() error\n");

return 1;

}

while (1);

return 0;

}

Скомпилируйте программу и просто запустите ее, затем нажимайте комбинацию клавиш <Ctrl>+<C> до тех пор, пока не надоест. А когда надоест, попробуйте **корректно** «убить» соответствующий процесс.

## Сигналы и многозадачность

Итак, мы выяснили, что при получении сигнала процесс немедленно оставляет все свои "дела" и приступает к обработке этого сигнала.

Недостатки использования сигналов при взаимодействии:

* сигналы очень ресурсоемки.
* сигналы слабо информативны.

**Ресурсоемкость**. Отправка сигнала требует выполнения системного вызова, а его доставка — *прерывания процесса-получателя* и *интенсивных операций со стеком* процесса для вызова функции обработки и продолжения его нормального выполнения.

**Информативность.** Число сигналов ограничено.

Поэтому в межпроцессном взаимодействии сигналы используются для обеспечения **событийной парадигмы** программирования.

Если при помощи sigaction() сигнал перехватывается, то вызывается функция-обработчик.

### Упражнение 3

Сначала реализуем "потомка"

**Программа-потомок kinsfolk-child2.c**

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <unistd.h>

int **main** (int argc, char \*\* argv)

{

int year;

if (argc < 2) {

fprintf (stderr, "child: too few arguments\n");

return 2;

}

year = atoi (argv[1]);

if (year <= 0)

return 2;

if ( ((year%4 == 0) && (year%100 != 0)) || (year%400 == 0) )

kill (getppid (), SIGUSR1);

else

kill (getppid (), SIGUSR2);

return 0;

}

Теперь реализуем "родителя"

**Программа-родитель kinsfolk-parent2.c**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <wait.h>

#include <signal.h>

/\* 0 — no signal, 1 — SIGUSR1, 2 — SIGUSR2 \*/

sig\_atomic\_t sig\_status = 0;

void handle\_usr1 (int s\_num) { sig\_status = 1; }

void handle\_usr2 (int s\_num) { sig\_status = 2; }

int main (int argc, char \*\* argv)

{

struct sigaction act\_usr1, act\_usr2;

sigemptyset (&act\_usr1.sa\_mask); sigemptyset (&act\_usr2.sa\_mask);

act\_usr1.sa\_flags = 0; act\_usr2.sa\_flags = 0;

act\_usr1.sa\_handler = &handle\_usr1; act\_usr2.sa\_handler = &handle\_usr2;

if (sigaction (SIGUSR1, &act\_usr1, NULL) == -1) {

fprintf (stderr, "sigaction (act\_usr1) error\n");

return 1;

}

if (sigaction (SIGUSR2, &act\_usr2, NULL) == -1) {

fprintf (stderr, "sigaction (act\_usr2) error\n");

return 1;

}

if (argc < 2) {

fprintf (stderr, "Too few arguments\n");

return 1;

}

if (!fork()) {

execl ("./kinsfolk-child2", "Child", argv[1], NULL);

fprintf (stderr, "execl() error\n");

return 1;

}

while (1) {

if (sig\_status == 1) {

printf ("%s: leap year\n", argv[1]);

return 0;

}

if (sig\_status == 2) {

printf ("%s: not leap year\n", argv[1]);

return 0;

}

}

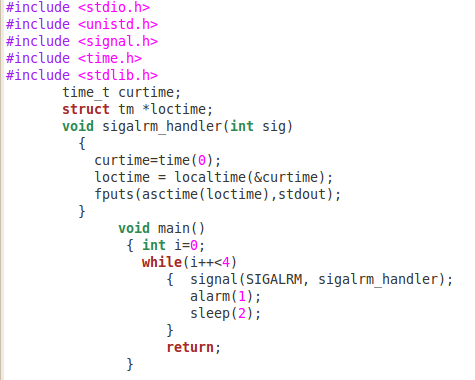
return 0;

}

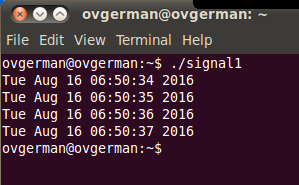
Скомпилируйте файлы, запустите программу. Объясните принцип ее работы.

### Упражнение 4

Вывод времени каждую секунду:

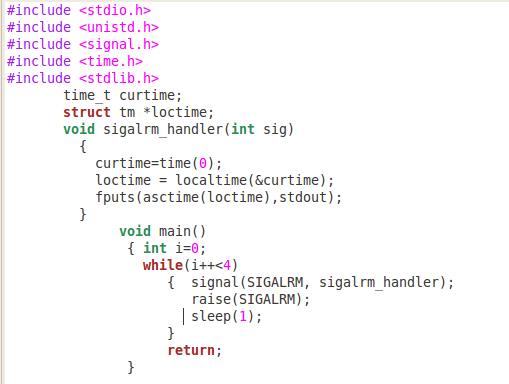


Результат работы программы демонстрирует следующий скриншот:



### Упражнение 5

Укажем более предпочтительный вариант программы (более универсальный, подходящий под различные типы сигналов).



Здесь использована функция raise(). Эта функция «зажигает сигнал». Отметим, что функция signal просто устанавливает связь с обработчиком сигнала, но не активизирует его (легко проверить, *закомментировав* вызов raise, *проведите эксперимент*, объясните поведение программы).

**Дополнительная информация к упражнению 2**

Чтобы завершить программу sigaction1, просто откройте другой терминал и представленному примеру "убейте" соответствующий процесс при помощи команды kill:

$ ps -e | grep sigaction1

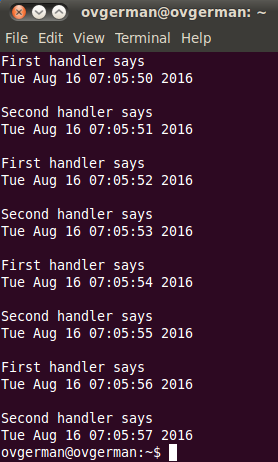
4883 pts/1 00:44:27 sigaction1

$ kill 4883

## Задания

**Задание 0**. Выполните упражнения 1-5.

**Задание 1**. Создайте два обработчика сигнала SIGALRM.

Здесь должны быть объявлены два обработчика для сигнала типа SIGALRM. Логика работы обоих обработчиков одна и та же. Скриншот дан.

**Задание** 2. Измените последнюю программу так, чтобы она выдавала не время и дату, а счет в игре двух человек. Первый обработчик играет за первого человека (условно говоря). Второй обработчик играет за второго человека. Первый обработчик добавляет случайное число к общей переменной, второй – отнимает. Играют до 10 раз. Если в конце игры на счету будет число больше нуля – то выиграл первый игрок, иначе выиграл второй игрок. Случайное число разыгрывается таким образом:

Для получения случайных чисел можно использовать устройство-генератор случайных чисел Linux или библиотечные функции. Последняя возможность иллюстрируется следующей программой:

#include "stdafx.h"

#include <stdio.h>

#include <time.h>

#include <stdlib.h>

#include <conio.h>

int \_tmain(int argc, \_TCHAR\* argv[])

{

int player [5] = {0,0,0,0,0};

srand(time(NULL)); //инициализация случайного генератора

for (int i = 0; i <5 ; i++ ){

player[i] =rand() % 100;

}

for (int i = 0; i <5 ;i++ ){

printf ("Player[%d] - %d\n",i, player[i]);

}

getch();

return 0;

}

Функция srand устанавливает начальное состояние генератора случайных чисел. Собственно, случайные числа в диапазоне от 1 до 100 получаются в команде player[i] =rand () % 100;

Функция возвращает случайное число от 1 до 100. Просто rand() возвращает достаточно большое случайное число, ограниченно системной константой.

Функция random\_ возвращает целое случайное число в диапазоне от min до max. Устройство для получения случайных чисел открывается в команде

dev\_random = open (“/dev/random”, O\_RDONLY);

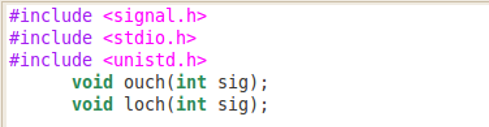
Считывание случайного числа выполняется в команде

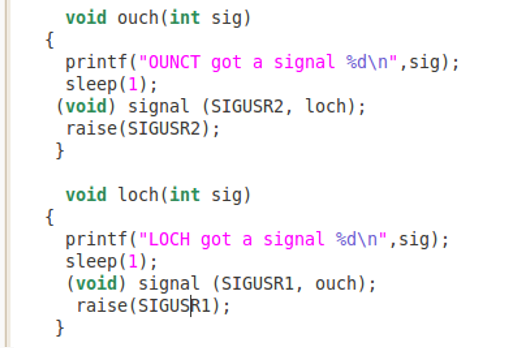
bytes\_read = read (dev\_random, newbyte, sizeof (random\_value));

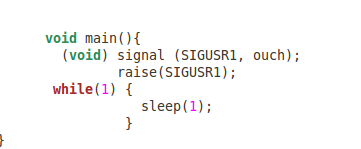
Случайное число помещается в переменную newbyte. Обычно случайные числа, генерируемые машиной, довольно большие. Чтобы уложить их в нужный диапазон, используем преобразование: newbyte % (max-min). Операция % соответствует взятию остатка от деления. Остаток от деления не может превосходить делимое, т.е. max-min.

Теперь познакомимся с сигналами пользователя – SIGUSR1 и SIGUSR2.

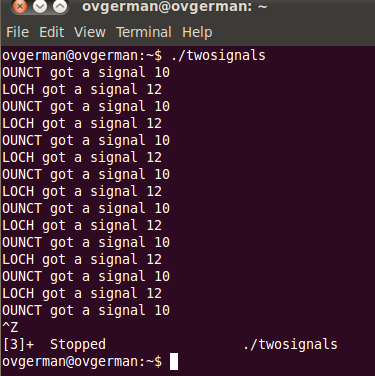
Рассмотрим следующий пример







Отметим, что команда (void) signal (…) не посылает сигнал, а связывает сигнал с обработчиком. Сигнал зажигается по команде raise(…). В нашем примере использованы два сигнала пользователя и два обработчика.

Работу их легко понять из порождаемого скриншота.

Происходит обмен сигналами между функциями с поочередным выводом на экран.

(цикл закрыть сигналом килл)

**Задание 3**. Представим, что есть клиент банка, который справляется о состоянии своего счета. Клиент посылает запрос (сигнал), банк возвращает ответ (сумма). Допустим, что сумма постепенно растет с каждым вызовом по формуле

X= X+0.1\*X

Написать соответствующий код.

Указание – использовать предыдущую программу.

**Дополнение к заданию 3.** Перестройте программу так чтобы сигналы передавались между отдельными процессами (программами), а не функциями одной программы.

**Дополнительное задание для отличников.** Изменить программу клиент-банк, так, чтобы сумма в банке росла по сигналам от источника, который не зависит от клиента, а клиент обращался за значением суммы на счете также независимо от процесса роста его счета. Подумайте!!!

Перестройте приложения так чтобы сигналы передавали отдельные программы, а не отдельные функции одной программы. Делайте клиент-серверные приложения.

## Контрольные вопросы

1. Что такое сигнал?
2. Что может отправлять сигналы в системе? Что может являться отправителем сигнала?
3. Что является идентификатором сигнала?
4. Укажите способы реакции процессом-получателем на полученный сигнал.
5. Чем отличаются команда kill и системный вызов kill()?
6. Что такое команда kill? Объясните ее синтаксическую семантику.
7. Что такое системный вызов kill()? Объясните ее синтаксическую семантику.
8. Что такое обработчик сигнала?
9. Для обеспечения какой парадигмы программирования используются сигналы?

# Дополнительная информация

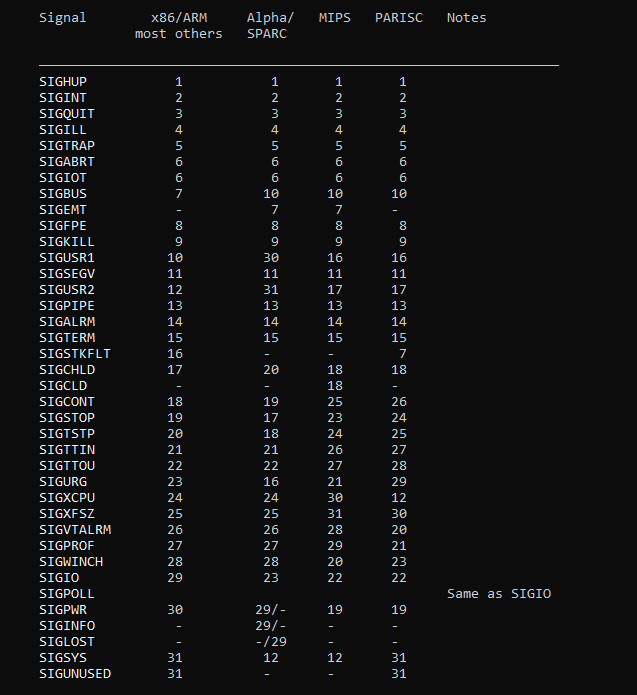
Основные причины отправки сигнала:

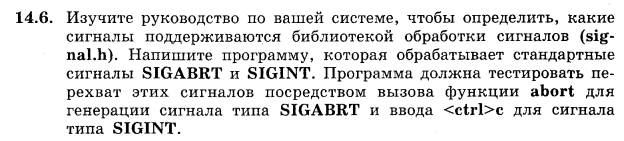
|  |  |
| --- | --- |
| Особые ситуации | Например, процесс вызывает деление на ноль. |
| Терминальные прерывания | Нажатие клавиш терминала, <Ctrl>+<C> или <Ctrl>+<\> |
| Другие процессы | Процесс отправляет сигнал другому процессу или группе процессов системный вызова kill(2). Элементарная форма межпроцессного взаимодействия. |
| Управление заданиями | Командные интерпретаторы, поддерживающие систему управления заданиями, используют сигналы для манипулирования фоновым и текущими задачами. |
| Квоты | Процесс превышает выделенную ему квоту вычислительных ресурсов или ресурсов файловой системы. |
| Уведомления | Процесс может запросить уведомление о наступлении тех или иных событий, например, готовности устройства и т.д. |
| Алармы | Если процесс установил таймер, ему будет отправлен сигнал, когда значение таймера станет равным нулю. |

POSIX, который стандартизировал обработку сигналов.

Man signal ???

Man 7 signal





# Литературные и интернет-источники

1. Робачевский А. М. Операционная система UNIX®. - СПб.: 2002. - 528 ил.

Стр. 236 – 252. (16)

1. Иванов Н. Н. И20 Программирование в Linux. Самоучитель. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2012. — 400 с.: ил.

Стр. 220 – 270. (50)

1. Харви Дейтел, Пол Дейтел. Как программировать на Си

Стр. 613 – 615. (2)

1. Linux. Системное программирование. 2-е изд. — СПб.: Питер, 2014. — 448 с.: ил.

Стр. 365 – 393. (28)