Взаимодействие процессов. Сигналы

- Сигналы. Основные понятия
- Старая модель работы с сигналами
  - Установка диспозиции сигнала
  - Ожидание сигналов
  - Отправка сигналов
- Новая модель работы с сигналами
  - Наборы сигналов
  - Блокировка сигналов
  - Новые средства управления сигналами
  - Отправка сигналов с информацией

Linux man pages: alphabetic list of all pages
https://man7.org/linux/man-pages/dir_all_alphabetic.html

Очень много технических подробностей !!!

Сигналы. Основные понятия

```
student@linux-VirtualBox:~/Work/Work2020/Lesson12/Text/Present$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
student@linux-VirtualBox:~/Work/Work2020/Lesson12/Text/Present$
```

Сигнал — это оповещение процесса о том, что произошло некое событие.

Список всех сигналов: kill -l

Отправка сигнала: kill -s SIGUSR1 pid

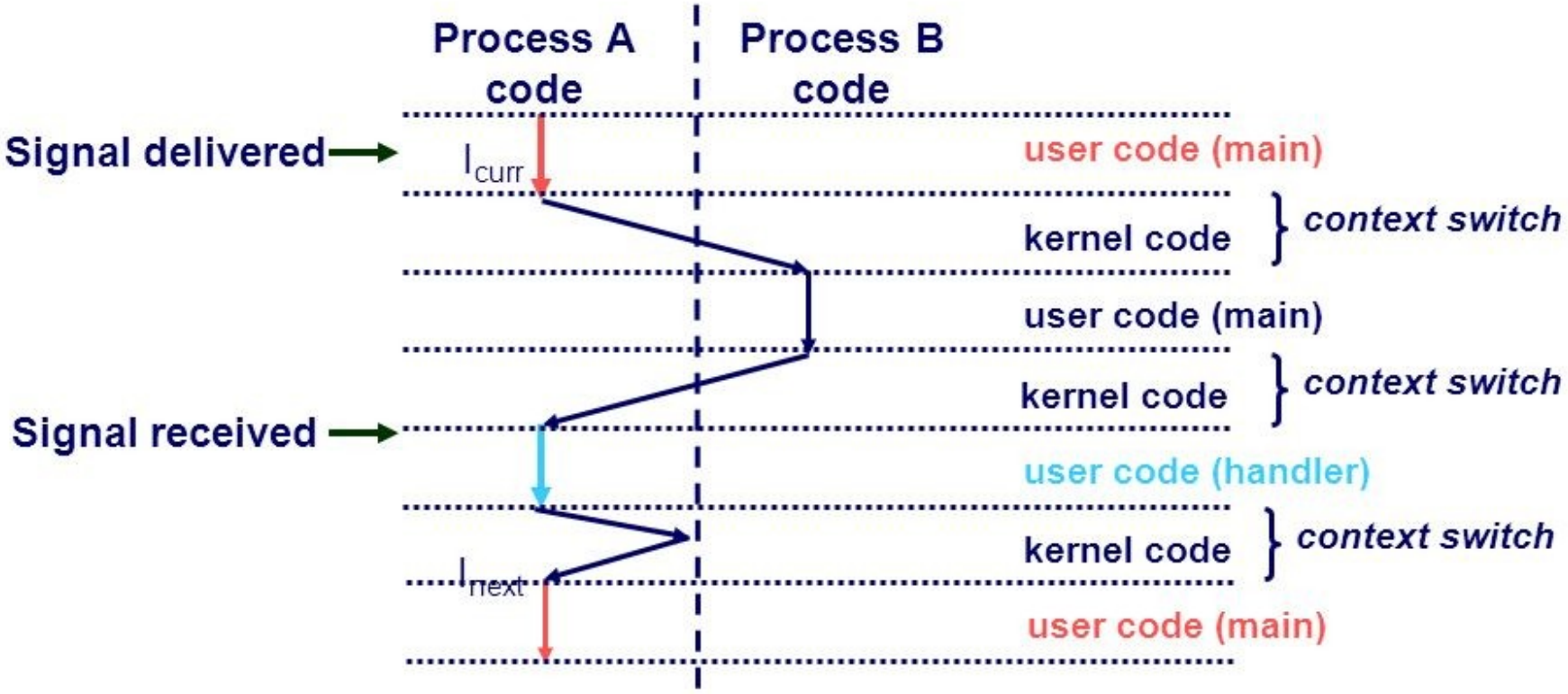
Все сигналы определены в заголовочном файле **<signal.h>**.

У каждого сигнала есть символическое имя, начинающееся с префикса **SIG**. Это определения препроцессора, ставящие в соответствие именам положительные целые числа.

Номера сигналов начинаются с единицы и непрерывно возрастают. Сигнала со значением **0** нет — это специальное значение, называемое нулевым сигналом (null signal). У него нет отдельного имени.

Сигналы иногда описываются как программные прерывания в том смысле, что они останавливают нормальное выполнение программы.

В большинстве случаев невозможно предсказать, когда именно будет доставлен тот или иной сигнал.



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. После того, как сигнал доставлен, ядро соответствующим образом *обрабатывает* (*handle*) его. Процесс не может изменить смысл сигналов SIGKILL и SIGSTOP.

При обработке сигнала, ядро может выполнить одно из трех действий:

- *Игнорировать сигнал*. Не предпринимаются никакие действия.
- *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.
- *Выполнить действие по умолчанию*. Это действие зависит от того, какой сигнал отправляется. Действием по умолчанию часто бывает либо завершение процесса, либо игнорирование сигнала.

Старая модель работы с сигналами

Установка диспозиции сигнала:

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

sighandler_t signal(int signo,
                    sighandler_t handler);
```

Прототип функции - обработчика:

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

Специальные значения 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 возвращает предыдущую диспозицию сигнала. В случае ошибки она возвращает значение SIG_ERR. Эта функция не устанавливает переменную errno.

Ожидание сигнала

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

Вызов функции приостанавливает выполнение вызывающего процесса до получения любого сигнала.

Если сигнал приводит к нормальному завершению процесса или игнорируется процессом, то в результате вызова pause будет просто выполнено соответствующее действие.

Если сигнал перехватывается, то после вызова обработчика сигнала вызов pause вернет значение (-1) и поместит в переменную errno значение EINTR.

Функция alarm устанавливает 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.

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

Сопоставление номеров сигналов и строк

Один способ — извлечь строку из статически определенного массива строк, содержащих имена поддерживающихся системой сигналов, проиндексированный по номерам сигналов:

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

Другой способ:

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

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

Следующий способ:

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

Вызов функции возвращает указатель на описание сигнала, указанного в аргументе 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.

Отправка сигнала

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

Аргументы:

- pid — идентификатор процесса, которому отправляется сигнал,
- signo — идентификатор отправляемого сигнала;

Если signo = 0, то такой вызов не отправляет сигнал, а выполняет проверку может ли процесс необходимыми разрешениями для отправки сигнала определенному процессу / процессам, и существуют ли эти процессы.

Обычный пользователь Linux может отправить сигнал только процессу, владельцем которого он является.

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 вызова (хотя бы один сигнал был отправлен) функция kill() возвращает 0.

Если ни одного сигнала не было отправлено, то 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.

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.

Новая модель работы с сигналами

Набор сигналов

```
#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);
```

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).

Управление сигнальной маской

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

Поведение sigprocmask() зависит от значения аргумента how.

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.

Возможные значения аргумента how:

- SIG_SETMASK Сигнальная маска для вызывающего процесса меняется на set.
- SIG_BLOCK Сигналы из set добавляются к сигнальной маске вызывающего процесса. (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.

Извлечение ожидающих сигналов

Заблокированный сигнал не доставляется процессу. Такие сигналы называются ожидающими. Когда ожидающий сигнал разблокируется, ядро передает его процессу для обработки.

Получение набора 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

Временное изменение сигнальной маски, и последующее ождание, 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.

Расширенное управление сигналами

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

Вызов sigaction() изменяет диспозицию поведение сигнала signo (не работает с сигналами SIGKILL и SIGSTOP). Если значение act != NULL, то системный вызов изменяет текущее поведение сигнала в соответствии со значением параметра act. Если значение oldact != NULL, то вызов сохраняет предыдущую (или текущую, если значение аргумента act == NULL) диспозицию 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); /* обработчики */
    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 */
}
```

Если в поле sa_flags:

- не установлен флаг SA_SIGINFO — функцию обработки сигналов определяет поле sa_handler
  Прототип функции:

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

- установлен флаг SA_SIGINFO — функцию обработки сигналов определяет поле sa_sigaction

Прототип функции:

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

Первый параметра — номер сигнала;
Второй — структура siginfo_t;
Третий — структуру типа ucontext_t (приведенную к указателю void). Два последние параметра предоставляют большое количество информации для обработчика сигналов;

Поле sa_mask содержит набор сигналов, которые система должна блокировать на время выполнения обработчика сигналов. Сигнал, который в данный момент обрабатывается, автоматически добавляется в маску, если только в поле sa_flags не содержится флаг SA_NODEFER.

Невозможно заблокировать сигналы SIGKILL или SIGSTOP (вызов функции игнорирует любое из этих значений, даже если оно добавляется в sa_mask).

Поле sa_flags — битовая маска, включающая ноль, один или несколько флагов, объединенных битовой операцией ИЛИ (|), изменяющих способ обработки сигнала, указанного в signo.

Рассмотрим некоторые из оставшихся значения поля sa_flags:

- SA_NOCLDWAIT — если signo == SIGCHLD, то данный флаг включает автоматическое удаление потомков: они не превращаются в зомби при завершении и предку не нужно делать для них системный вызов wait().
- SA_RESTART — включает автоматический перезапуск системных вызовов, прерванных сигналами.
- SA_RESETHAND — 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 функция 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 возврата или сигнал */
```

```
    clock_t si_utime;
     /* затраченное пользовательское время */
    clock_t si_stime;
     /* затраченное 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;    /* адрес в памяти, вызвавший
                         сбой */
    int si_band;            /* событие полосы */
    int si_fd;              /* дескриптор файла */
};
```

Полное описание полей — в справочной системе.
Рассмотрим только некоторые.

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

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.

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.

Некоторые значения, используемые для всех сигналов:
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().

См. в методичке значения, применяемые с сигналом SIGCHLD.

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.

Для отправки процессу сигналов с такой информацией используется функция sigqueue():

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

В случае успешного завершения вызова 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 переменная errno принимает одно из следующих значений:

EINVAL — значение аргумента signo соответствует недопустимому сигналу.

EPERM — у вызывающего процесса отсутствуют разрешения на отправку сигналов любым из запрошенных процессов.

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

Сигналы, в принципе, могут использоваться для организации взаимодействия процессов, но, программирование на сигналах сложно и громоздко.

Этому есть следующие причины:

- Асинхронная природа сигналов означает, что появляется вероятность столкновения с проблемами, характерными для параллельных программ (напр., с требованиями реентерабельности, состоянием гонки, с проблемой корректной обработки глобальных переменных из обработчиков сигналов);

- Стандартные сигналы не ставятся в очередь. Даже для сигналов реального времени существуют пределы по количеству сигналов, которые могут быть поставлены в очередь. Это значит, что во избежание потери информации процесс, получающий сигналы, должен иметь метод информирования отправителя о том, что он (получатель) готов к приему следующего сигнала. Самый простой способ решения этой проблемы — отправка получателем сигнала отправителю.

Еще одна проблема заключается в том, что сигналы могут передавать только ограниченный объем информации: номер сигнала и — в случае с сигналами реального времени — слово (целое число или указатель) дополнительных данных.