Сергей Юрьевич Шилов

Системное программирование для современных платформ

Системное программирование для современных платформ

3. Сигналы

Системное программирование для современных платформ

Сигналы:

- асинхронное программное прерывание
- взаимодействие в канале управления
- event-base programming
- маскируемые и немаскируемые
- пользовательские и системные обработчики

Системное программирование для современных платформ

Mnemonic	Code	Terminal	Dump	Default action	Description	Type
SUGHUP	1			Terminate	Закрытие терминала	Event
SIGINT	2	Ctrl-C		Terminate	Завершение процесса с терминала	Mgmnt
SIGQUIT	3	Ctrl-\	+	Terminate	Закрытие процесса с терминала	Mgmnt
SIGILL	4		+	Terminate	Invalid opcode exeption	Exeption
SIGTRAP	5		+	Terminate	Breakpoint	Debug
SIGABRT	6		+	Terminate	abort() syscall	Mgmnt
SIGFPE	8		+	Terminate	FPE exeption	Exeption
SIGKILL	9			Terminate	Безусловное завершение процесса	Mgmnt
SIGBUS	10		+	Terminate	Memory access exeption	Exeption
SIGSEGV	11		+	Terminate	Segmentation exeption	Exeption
SIGSYS	12		+	Terminate	Bad syscall exeption	Exeption
SIGPIPE	13			Terminate	Запись в некорректный канал/сокет	Event
SIGALRM	14			Terminate	Истечение времени таймера	Event
SIGTERM	15			Terminate	Завершение процесса	Mgmnt

Системное программирование для современных платформ

Mnemonic	Code	Terminal	Dump	Default action	Description	Type
SIGUSR1	16			Terminate	Пользовательский сигнал	User
SIGUSR2	17			Terminate	Пользовательский сигнал	User
SIGCHLD	18			Ignore	Завершение дочернего процесса	Event
SIGSTP	20	Ctrl-Z		Stoping	Останов процесса с терминала	Mgmnt
SIGURG	21			Ignore	Срочные данные TCP-сокета	Event
SIGPOLL	22			Terminate	poll() syscall	Event
SIGSTOP	23			Stop	Останов процесса	Mgmnt
SIGCONT	25			Continue	Продолжение исполнения процесса	Mgmnt
SIGTTIN	26			Stop	Чтение с управляющего терминала	Event
SIGTTOUT	27			Stop	Запись на управляющий терминал	Event
SIGVTALRM	28			Terminate	Истечение времени таймера	Event
SIGPROF	29			Terminate	Истечение таймера профайлера	Debug
SIGSCPU	30		+	Terminate	Превышен лимит времени CPU	Exeption
SIGXFSZ	31		+	Terminate	Превышен допустимый размер файла	Exeption

Системное программирование для современных платформ

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
#define raise(sig) kill(getpid(), sig);
```

Системный вызов **kill** используется для отправки сигнала к.-л. процессу или группе процессов. Если значение **pid** является положительным, сигнал **sig** посылается процессу с идентификатором **pid**. Если sig равен 0, то никакой сигнал не посылается, а только выполняется проверка на ошибку.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ:
В случае успеха, возвращается ноль. При ошибке, возвращается **-1** и устанавливается значение **errno**
ОШИБКИ:
EINVAL - Задан неправильный сигнал.
ESRCH - Идентификатор процесса **pid** или группа процесса не существуют. Заметим, что существующий процесс может быть зомби - процессом, который уже находится в состоянии завершения, но пока в котором пока ещё не выполнился **wait()**.
EPERM - екущий процесс не имеет прав на посылку сигнала к любому из указанных процессов. Процессы, которые имеют права на посылку сигнала процессу с номером **pid** должны иметь привилегии суперпользователя или, реальный или эффективный идентификатор пользователя процесса, посылающего сигнал, должен быть таким же как реальный или эффективный идентификатор пользователя процесса, принимающего сигнал. В случае, когда посылающий и принимающий процессы относятся к одной сессии, становится доступным сигнал **SIGCONT**.

ЗАМЕЧАНИЯ: Невозможно послать сигнал процессу с номером **1**, т.е. процессу **init**, для которого не устанавливается обработчик сигналов. Так сделано, чтобы быть уверенным, что в случае какой-либо нештатной ситуации, работа системы не будет завершена аварийно.

Системное программирование для современных платформ

kill [**-s** СИГНАЛ | -СИГНАЛ] **PID**...
kill -l [СИГНАЛ]...

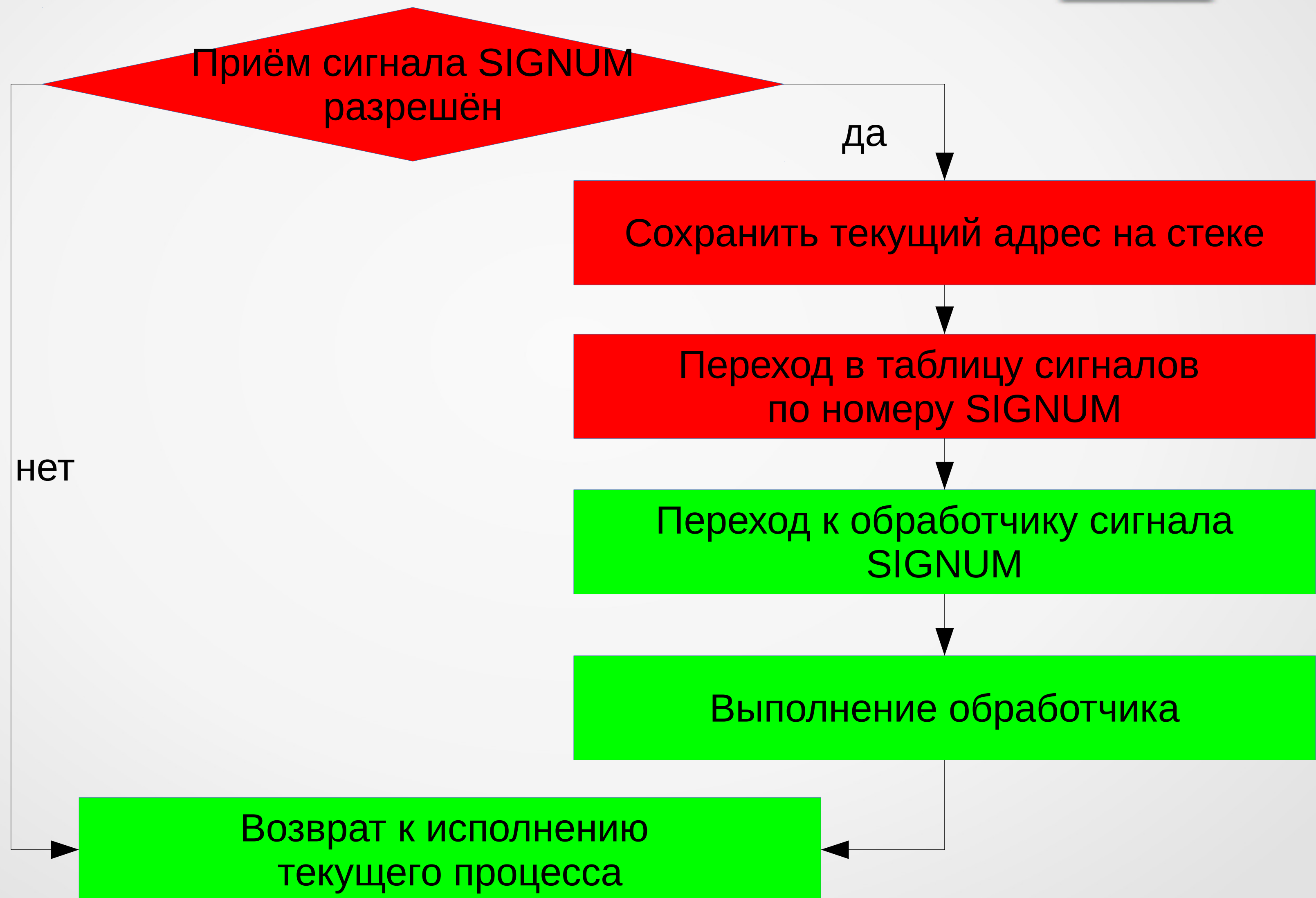
Посылает сигнал процессу или выводит список допустимых сигналов. Аргументы, обязательные для полных вариантов опций, являются обязательными также и для кратких вариантов.

-s, --signal=СИГНАЛ, -СИГНАЛ
имя или номер посылаемого сигнала
-l, --list
вывести имена сигналов или вывести имя сигнала, соответствующее номеру, и наоборот

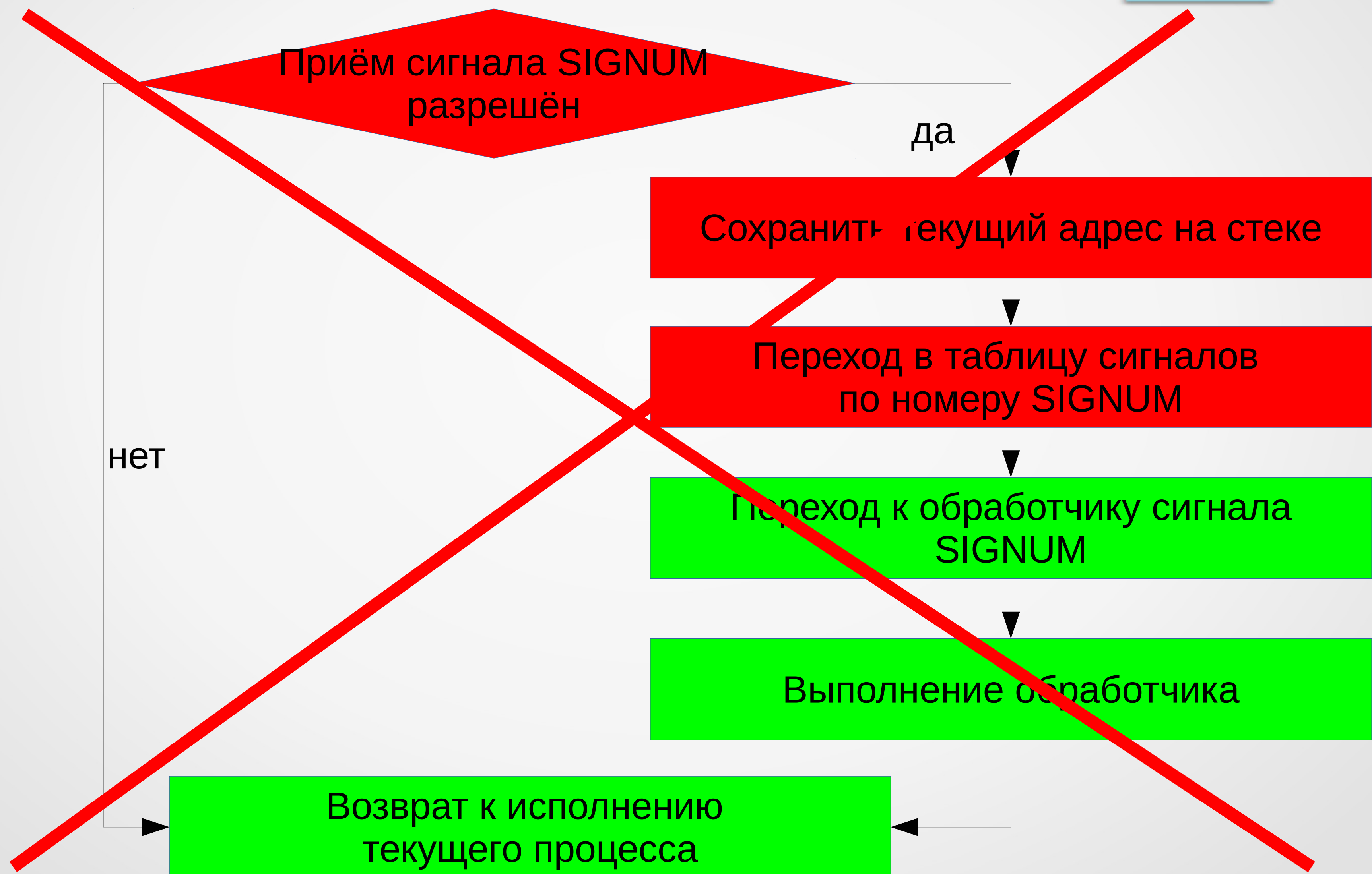
СИГНАЛ может указываться в виде имени (например, `**HUP**') или номера (например, `**1**').

PID - числовой идентификатор процесса. Если число отрицательное, оно определяет группу процесса.

Системное программирование для современных платформ



Системное программирование для современных платформ



Системное программирование для современных платформ

Системное программирование для современных платформ

Системное программирование для современных платформ

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

signal() устанавливает новый обработчик сигнала с номером **signum** в соответствии с параметром **handler**, который может быть функцией пользователя, **SIG_IGN** или **SIG_DFL.** При получении процессом сигнала с номером **signum** происходит следующее: если устанавливаемое значение обработчика равно **SIG_IGN**, то сигнал игнорируется; если оно равно **SIG_DFL**, то выполняется стандартный обработчик сигнала. Если обработчик установлен в функцию **handler**, то сначала устанавливает значение обработчика в **SIG_DFL** или выполняется зависимая от реализации блокировка сигнала, а затем вызывается функция **handler** с параметром **signum.** Использование функции-обработчика сигнала называется "перехватом". Сигналы **SIGKILL** и **SIGSTOP** не могут быть "перехвачены" или игнорированы.

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ: Функция **signal()** возвращает предыдущее значение обработчика сигнала или **SIG_ERR** при ошибке.

ОСОБЕННОСТИ: **BSD** не перезагружает обработчик, а блокирует новые сигналы на время вызова обработчика. Библиотека **glibc2** следует поведению **BSD.** В системе **libc5** включение **<bsd/signal.h>** вместо **<signal.h>**, приводит к переопределению **signal** в **__bsd_signal**, и эта функция начинает работать, как в **BSD**.

ТАКЖЕ:

```
#define SIG_DFL          void (*)(int)0
#define SIG_DFL          void (*)(int)1
```

Системное программирование для современных платформ

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

void sig_handler (int signum){
    signal (signum, sig_handler);
    printf ("Catch signal %i\n", signum)
}

int main(int argc, char **argv){
    signal (SIGTERM, sig_handler);
    signal (SIGUSR1, SIG_IGN);
    signal (SIGUSR2, SIG_DFL);
    while (1) pause ();
}
```

Системное программирование для современных платформ

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

sigprocmask используется для того, чтобы изменить список блокированных в данный момент сигналов. Работа этой функции зависит от значения параметра **cmd** следующим образом:

SIG_BLOCK	Набор блокируемых сигналов - объединение текущего набора и аргумента **set**
SIG_UNBLOCK	Сигналы, устанавливаемое значение битов которых равно **set**, удаляются из списка блокируемых сигналов. Допускается разблокировать незаблокированные сигналы
SIG_SETMASK	Набор блокируемых сигналов приравнивается к аргументу **set.** Если значение поля **oldset** не равно **NULL**, то предыдущее значение маски сохраняется в **oldset**

Для переносимости, не рекомендуется заполнять структуры типа **sigset_t** «вручную». Вместо этого используются функции:

```
int sigemptyset (sigset_t* mask)
int sigaddset (sigset_t* mask, const int sig)
int sigdelset (sigset_t* mask, const int sig)
int sigfillset (sigset_t* mask)
int sigismember (const sigset_t* mask, const int sig)
```

При успешном выполнении sigemptyset, sigaddset, sigdelset, sigfillset возвращают 0, в случае ошибки -1. Возможной причиной неудачи могут быть некорректные значения параметров mask или sig. sigismemeber возвращает 1, если в маске есть сигнал sig, 0 в противном случае, -1 в случае ошибки.

Системное программирование для современных платформ

```
#include <stdio.h>
#include <signal.h>
int main (int argc, char** argv){
     sigset_t mask;
     sigemptyset (&mask);
     if (sigprocmask(0,0, &mask) == -1){
          perror ("sigprocmask - 1");
          return 1;
     } else {
          sigaddset (&mask; SIGINT);
     }
     sigdelset (&mask, SIGSEGV);
     if (sigprocmask (SIG_SETMASK, &mask, 0) == -1){
     perror ("sigprocmask – 2");
     }
}
```

Системное программирование для современных платформ

```
int sigaction(int signum, const struct sigaction *act, struct
sigaction *oldact);

struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void); }
```

ОПИСАНИЕ

Системный вызов **sigaction** используется для изменения действий процесса при получении соответствующего сигнала. Параметр **signum** задает номер сигнала и может быть равен любому номеру, кроме **SIGKILL** и **SIGSTOP**. Конкретные операционные системы могут запрещать установку дополнительных сигналов.

Если параметр **act** не равен нулю, то новое действие, связанное с сигналом **signum**, устанавливается соответственно **act**. Если **oldact** не равен нулю, то предыдущее действие записывается в **oldact**.

sa_handler задает тип действий процесса, связанный с сигналом **signum**, и может быть равен: **SIG_DFL** для выполнения стандартных действий, **SIG_IGN** для игнорирования сигнала,- или быть указателем на функцию обработки сигнала.

Системное программирование для современных платформ

ОПИСАНИЕ (ПРОДОЛЖЕНИЕ)

sa_mask задает маску сигналов, которые должны блокироваться при обработке сигнала. Также будет блокироваться и сигнал, вызвавший запуск функции, если только не были использованы флаги **SA_NODEFER** или **SA_NOMASK.**

sa_flags содержит набор флагов, которые могут влиять на поведение процесса при обработке сигнала. Он состоит из следующих флагов:

SA_NOCLDSTOP - Если **signum** равен **SIGCHLD,** то уведомление об остановке дочернего процесса не будет получено (т.е., в тех случаях, когда дочерний процесс получает сигнал **SIGSTOP, SIGTSTP, SIGTTIN** или **SIGTTOU**).
SA_ONESHOT или **SA_RESETHAND -** Восстановить поведение сигнала после одного вызова обработчика.
SA_ONSTACK - Вызвать обработчик сигнала в дополнительном стеке сигналов, предоставленном **sigaltstack(2).** Если дополнительный стек недоступен, то будет использован стек по умолчанию. Требуется поддержка со стороны ОС и/или CPU
SA_RESTART - Поведение должно соответствовать семантике сигналов BSD и позволять некоторым системным вызовам работать, в то время как идет обработка сигналов.
SA_NOMASK или **SA_NODEFER -** Не препятствовать получению сигнала при его обработке.
SA_SIGINFO - Обработчик сигнала требует 3-х аргументов, а не одного. В этом случае надо использовать параметр **sa_sigaction** вместо **sa_handler**.

Системное программирование для современных платформ

```
int sigpending(sigset_t *set);
int sigsuspend(const sigset_t *mask);
```

ОПИСАНИЕ

Системный вызов **sigpending** позволяет определить наличие ожидающих сигналов (полученных заблокированных сигналов). Маска ожидающих сигналов помещается в **set**.

Системный вызов **sigsuspend** временно изменяет значение маски блокировки сигналов процесса на указанное в **mask**, и затем приостанавливает работу процесса до получения соответствующего сигнала.

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

Функции **sigaction**, **sigprocmask** и **sigpending** возвращают **0** в случае успеха и **-1** при ошибке. Функция **sigsuspend** всегда возвращает **-1**, обычно с кодом ошибки **EINTR**.

ОШИБКИ

EINVAL - Было задано неверное значение сигнала. Эта ошибка также возникает в случае попытки задания сигналов **SIGKILL** или **SIGSTOP**, которые не могут быть "перехвачены".
EFAULT - act, **oldact**, **set**, **oldset** или **mask** указывают на адрес, не входящий в адресное пространство процесса.
EINTR - Системный вызов был прерван.

Системное программирование для современных платформ

ПРИМЕЧАНИЯ

Невозможно заблокировать сигналы **SIGKILL** или **SIGSTOP** при помощи системного вызова **sigprocmask**. Попытки это сделать будут игнорироваться. Конкретные операционные системы могут запрещать установку дополнительных сигналов.

POSIX не определяет поведение процесса после игнорирования сигнала **SIGFPE**, **SIGILL** или **SIGSEGV**, если эти сигналы не были посланы при помощи функций **kill()** или **raise()**. Деление целого числа на ноль имеет непредсказуемый результат. В некоторых архитектурах это приводит к появлению сигнала **SIGFPE**. Деление **NEGMAXINT** самого большого по модулю отрицательного числа на -1 также приводит к появлению **SIGFPE**. Игнорирование этого сигнала может привести к появлению бесконечного цикла. Поведение в случае других запрещённых математических операций зависит от CPU и/или ОС.

POSIX запрещает установку действия для сигнала **SIGCHLD** на **SIG_IGN**. Поведение **BSD** и **SYSV** в этом случае различается. Это приводит к тому, что **BSD**-программы, устанавливающие **SIGCHLD** в **SIG_IGN**, в **Linux** не работают.

Специфические черты **POSIX** определяют только **SA_NOCLDSTOP**. Использование других флагов в **sa_flags** может быть неэффективно в других системах.

Флаг **SA_RESETHAND** совместим с одноименным флагом **SVr4**, однако требуется поддержка со стороны ОС.

Флаг **SA_NODEFER** совместим с одноименным флагом **SVr4**, однако требуется поддержка со стороны ОС.

Функция **sigaction** со вторым нулевым аргументом может быть вызвана для того, чтобы получить адрес текущего обработчика прерываний. Эту функцию можно также использовать для проверки правильности этого типа сигнала в конкретной системе, вызвав ее с нулевыми вторым и третьим параметрами.

Системное программирование для современных платформ

```
#include <signal.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
/*Вариант "надежной" функции обработки сигналов*/
void (*mysignal (int signum, void (*handler)(int)))(int) {
     struct sigaction act, oldact;
     if ((signum == SIGKILL) || (signum == SIGSTOP)) return -1;
     act.sa_handler=handler;
     sigemptyset(&act.sa_mask);
     act.sa_flags=0;
     act.sa_flags |= SA_RESETHAND;
     if (signum != SIGALRM) act.sa_flags |= SA_RESTART;
     if (sigaction(signum,&act,&oldact)!=0) return (SIG_ERR);
     return (oldact.sa_handler);

/*Функция-обработчик*/
static void sig_handler (int signum) {
     mysignal (SIGINT, sig_handler);
     printf ( "Catch signal %i\n", sigmum );
}

main (argc, **argv) {
     mysignal (SIGINT, sig_handler);
     mysignal (SIGUSR1, SIG_DFL);
     mysignal (SIGUSR2, SIG_IGN);
     while (1) pause();
}
```

Системное программирование для современных платформ

```
#include <sys/types.h>
#include <wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
pid_t wait(int *status);
```

ОПИСАНИЕ

Системный вызов **waitpid()** блокирует выполнение текущего процесса до тех пор, пока либо не завершится порожденный им процесс, определяемый значением параметра **pid**, либо пока текущий процесс не получит сигнал, для которого установлена реакция по умолчанию "завершить процесс" или реакция обработки пользовательской функцией. Если порожденный процесс, заданный параметром pid, к моменту системного вызова находится в состоянии "закончил исполнение", системный вызов завершается немедленно без блокирования текущего процесса.

Параметр **pid** определяет порожденный процесс, завершения которого дожидается процесс-родитель, следующим образом:
> 0 ожидаем завершения процесса с идентификатором **pid**.
= 0, то ожидаем завершения любого порожденного процесса в группе, к которой принадлежит процесс-родитель.
= -1, то ожидаем завершения любого порожденного процесса.
< 0, **!= -1**, то ожидаем завершения любого порожденного процесса из группы, идентификатор которой равен абсолютному значению параметра **pid**.

Системное программирование для современных платформ

ОПИСАНИЕ (ПРОДОЛЖЕНИЕ)

Значение **options** создается путем логического сложения нескольких следующих констант:

WNOHANG
означает немедленное возвращение управления, если ни один дочерний процесс не завершил выполнение.
WUNTRACED
означает возврат управления и для остановленных (но не отслеживаемых) дочерних процессов, о статусе которых еще не было сообщено. Статус для отслеживаемых остановленных процессов также обеспечивается без этой опции.

Если системный вызов обнаружил завершившийся порожденный процесс, из числа специфицированных параметром **pid**, то этот процесс удаляется из вычислительной системы, а по адресу, указанному в параметре **status**, заносится информация о статусе его завершения. Параметр **status** может быть задан равным **NULL**, если эта информация не имеет нас значения.

При обнаружении завершившегося процесса системный вызов возвращает его идентификатор. Если вызов был сделан с установленной опцией **WNOHANG**, порожденный процесс, специфицированный параметром **pid** существует, но еще не завершился, то системный вызов возвращает значение **0**. Во всех остальных случаях он возвращает отрицательное значение. Возврат из вызова, связанный с возникновением обработанного пользователем сигнала, может быть в этом случае идентифицирован по значению системной переменной **errno == EINTR**, и вызов может быть сделан снова.

Системный вызов **wait** является синонимом для системного вызова **waitpid** с значениями параметров **pid = -1**, **options** = 0.

Системное программирование для современных платформ

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

void sighandler(int sig) {
   printf("In signal handler for signal  %d\n",  sig);
   /* wait() это основное для подтверждения SIGCHLD */
   wait(0);}

int main(void) {
   int i;
   /* Установить обработчик сигнала к SIGCHLD  */
   sigset(SIGCHLD,  &sighandler);
      if (!fork()) {
         _exit(0); /* Потомок */
      }
      sleep(60);
}
```

Системное программирование для современных платформ

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

ОПИСАНИЕ

Системный вызов **alarm** выполняет в вызвавший его процесс доставку сигнала **SIGALRM** через **seconds** секунд. Если **seconds** равно нулю, то никаких новых тревожных сигналов в очередь поставлено не будет.

Если текущий процесс получит сигнал, для которого установлена реакция по умолчанию "завершить процесс" или реакция обработки пользовательской функцией, любые предыдущие установки **alarm** отменяются.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

alarm возвращает количество секунд, оставшихся до момента доставки сигнала, установленного предыдущим вызовом alarm или ноль, если в очереди нет тревожных сигналов.

ПРИМЕЧАНИЯ

Вызовы **alarm** и **setitimer** совместно используют один и тот же таймер; они будут конфликтовать друг с другом. **sleep()** также может быть реализован, используя **SIGALRM**; так что смешанное использование вызовов **alarm()** и **sleep()** - это плохая идея.

Постановка сигнала в очередь может вызывать задержу выполнения вызвавшего процесса на некоторое время.

Системное программирование для современных платформ

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

void wakeup() {};
uint sleep (uint time){
     struct sigaction act;
     act.sa_habdler = wakeup;
     act.sa_flags |= SA_RESTART;
     sigemptyset (&act.sa_mask);
     if (sigaction(SIGALARM, &act, 0)!=0) {
          perror («sigaction»);
          return 1;
     }
     (void) alarm(time);
     (void) pause();
     return 0;
}
```

Системное программирование для современных платформ

Спасибо за внимание!

:-)

Системное программирование для современных платформ

Системное программирование для современных платформ

Системное программирование для современных платформ

Резервные слайды