Концепция сигналов Linux

Сигнал — это асинхронное уведомление процесса о каком либо событии.

Сигнал может посылать ядро и процесс.

Отличительная особенность сигнала в том, что посылать/принимать сигналы можно только в пределах одного компьютера и процессам, связанным родственными отношениями (предку, потомку, группе, должны иметь один и тот же идентификатор пользователя).

Типы сигналов

Есть 2 типа сигналов:

• сигналы стандарта POSIX (1-31, не разделяемые во времени);

• сигналы реального времени (32 (**SIGRTMIN)** – 63(**SIGRTMAX**), упорядоченные во времени)

Жизненный цикл сигнала

Посылка сигнала ➡ Ожидание обработки ➡ Обработка сигнала

Вначале сигнал создается – посылается процессом или генерируется ядром. Затем сигнал ожидает доставки в процесс-приемник. Принято считать, что период ожидания сигнала равен промежутку времени между созданием сигнала и действием, которое выполняется по этому сигналу. В конце жизненного цикла сигнала происходит его перехват (прием) процессом и выполнение связанных с сигналом действий.

Процесс узнает, что ему послали сигнал, когда он переходит из состояния выполнения ядра в состояние выполнения программы.

Список основных сигналов POSIX

Имя сигнала	Номер	Назначение сигнала	Реакция по умолчанию
SIGHUP	1	Разрыв связи, лидер завершил работу	Завершить
SIGINT	2	Прервать процесс, Ctrl-C	Завершить
SIGILL	4	Некорректная команда	Завершить (дамп)
SIGABRT	6	Аварийное завершение	Завершить (дамп)
SIGFPE	8	Неверная операция (переполнение, деление на 0)	Завершить (дамп)
SIGKILL	9	Безусловно завершить процесс	Завершить
SIGUSR1	10	Сигнал пользователя	Завершить
SIGSEGV	11	Нарушение защиты памяти	Завершить (дамп)
SIGUSR2	12	Сигнал пользователя	Завершить
SIGPIPE	13	Запись в канал не открытый для чтения	Завершить
SIGALRM	14	Сигнал от программного таймера	Завершить
SIGTERM	15	Условное завершение	Завершить
SIGCHLD	17	Завершение или остановка потомка	Игнорировать
SIGCONT	18	Возобновить процесс после остановки	Игнорировать
SIGSTOP	19	Безусловная приостановка процесса	TASK_STOPPED
SIGTSTP	20	Терминальная остановка , Ctrl-Z	TASK_STOPPED
SIGXCPU	24	Превышено время работы процессора	Завершить (дамп)
SIGXFSZ	25	Превышен размер файла	Завершить (дамп)
SIGVTALRM	26	Сигнал от виртуального таймера	Завершить
SIGPROF	27	Закончилось время профилирующего таймера	Завершить

Сигналы реального времени

1. Для работы с сигналами реального времени применяются те же самые системные вызовы, что и для работы с обычными сигналами.
2. Одинаковые сигналы реального времени не сливаются.
3. Упорядоченная доставка сигналов, сигналы с меньшими номерами доставляются перед сигналами с большими номерами.

Посылка сигнала

int kill(pid_t *pid*, int *sig*);

- Если значение *pid* является положительным, сигнал *sig* посылается процессу с идентификатором *pid*.
- Если *pid* равен 0, то *sig* посылается каждому процессу, который входит в группу текущего процесса.
- Если *pid* равен -1, то *sig* посылается каждому процессу, за исключением процесса с номером 1 (init).
- Если *pid* меньше чем -1, то *sig* посылается каждому процессу, который входит в группу процесса *-pid*.
- Если *sig* равен 0, то никакой сигнал не посылается, а только выполняется проверка на существования процесса или группы.

int raise(int sig); текущему процессу

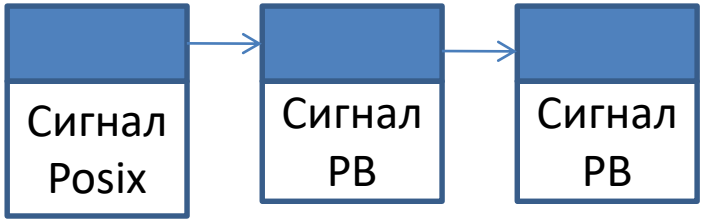
int pthread_kill(pthread_t thread, int sig); отдельному потку

Сигналы могут быть сгенерированы и обрабатываться синхронно или асинхронно.

Хранение сигналов

Очередь сигналов

```
struct sigpending {
    struct list_head list;
    sigset_t signal;
};
typedef struct {
    unsigned long sig[_NSIG_WORDS];
} sigset_t;
```



Для различных платформ _NSIG_WORDS варьируется от 1 до 4

каждый разряд массива соответствует одному сигналу.

Функции работы с sigset_t

Очистка

int sigemptyset(sigset_t *set);

Заполнение единицами

int sigfillset(sigset_t *set);

Добавление сигнала

int sigaddset(sigset_t *set, int signum);

Удаление сигнала

int sigdelset(sigset_t *set, int signum);

Проверка присутствия сигнала

int sigismember(const sigset_t *set, int signum);

Обработка сигнала

```
struct  sigaction {
            union {
              __sighandler_t   _sa_handler;    обработчик сигналов POSIX
              void (*_sa_sigaction)(int, struct siginfo *, void *); обработчик сигналов
реального времени
            } _u;
            sigset_t   sa_mask;   маска сигналов (набор заблокированных сигналов)
            unsigned long   sa_flags;  флаги
            void (*sa_restorer)(void); не используется
};
```

sa_handler=
- SIG_DFL (0) реакция по умолчанию
- SIG_IGN (1) игнорировать сигнал
- адрес обработчика сигнала

sa_flags=
- SA_ONESHOT Восстановить стандартную реакцию сигнала после одного вызова обработчика
- SA_NOMASK Не препятствовать получению сигнала при его обработке
- SA_ONSTACK Вызвать обработчик сигнала в дополнительном стеке сигналов

Установка обработчика

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

Требования к обработчику сигнала

Существует 2 класса сигналов: *ненадежные (unreliable)* и *надежные (reliable)*.

Ненадежные сигналы это те, для которых надо перенастраивать обработчик, что может привести к неготовности процесса в нужный момент принять сигнал.

Обработчики сигналов должны удовлетворять требованию реентерабельности, то есть, он должен допускать свой повторный вызов, когда процесс уже находится в обработчике. Требование реентерабельности сводится к тому, чтобы функция не использовала никаких глобальных ресурсов, не позаботившись о синхронизации доступа к этим ресурсам.

Если обработчик использует какую-то переменную в качестве флага поступления сигнала, она должна иметь специальный тип sig_atomic_t. Linux гарантирует, что операция присваивания значения такой переменной займет ровно один такт и не будет прервана.

Реентерабельные функции, которые могут быть вызваны из обработчика сигнала

accept	*fdatasync*	*link*	*rename*	*sigprocmask*
access	*fork*	*listen*	*rmdir*	*sigqueue*
alarm	*fpathconf*	*lseek*	*select*	*sigset*
bind	*fstat*	*lstat*	*send*	*sigsuspend*
chdir	*fsync*	*mkdir*	*sendmsg*	*sleep*
chmod	*ftruncate*	*mkfifo*	*setsid*	*socket*
chown	*getegid*	*open*	*setsockopt*	*socketpair*
close	*geteuid*	*pathconf*	*setuid*	*time*
connect	*getgid*	*pause*	*shutdown*	*times*
creat	*getgroups*	*pipe*	*sigaction*	*umask*
dup	*getpeername*	*poll*	*sigaddset*	*uname*
dup2	*getpgrp*	*pselect*	*sigdelset*	*unlink*
execle	*getpid*	*raise*	*sigemptyset*	*utime*
execve	*getppid*	*read*	*sigfillset*	*wait*
_Exit и *_exit*	*getsockname*	*readlink*	*sigismember*	*waitpid*
fchmod	*getsockopt*	*recv*	*signal*	*write*
fchown	*getuid*	*recvfrom*	*sigpause*	
fcntl	*kill*	*recvmsg*	*sigpending*	

Простая установка обработчика

signal(signum,function);

Функция signal эквивалентна применению sigaction с флагами SA_ONESHOT и SA_NOMASK.

Обработчик сигнала должен быть функцией, объявляемой по такому прототипу:

void *имя_обработчика*(int signum);

- функция не блокирует получение других сигналов пока выполняется текущий обработчик, он будет прерван и начнет выполняться новый обработчик
- после первого получения сигнала (для которого мы установили свой обработчик), его обработчик будет сброшен на SIG_DFL

Обработчик нескольких сигналов

```
void signal_handler(int sig) {
switch(sig) {
case SIGHUP:
//Обработка сигнала SIGHUP
break;
……….
case SIGTERM:
//Обработка сигнала SIGTERM
break; }
 }
```

Маскирование сигналов

Установить сигнальную маску

int sigprocmask(int *how*, const sigset_t **set*, sigset_t **oldset*);

how=
- SIG_BLOCK объединение текущей маски и аргумента *set*
- SIG_UNBLOCK пересечение текущей маски и аргумента *set*
- SIG_SETMASK замена текущей маски на аргумент *set*

Получить заблокированные сигналы

int sigpending(sigset_t **set*);

Устанавливает маску и переводит процесс в состояние ожидания, после обработки сигнала маска восстанавливается

int sigsuspend(const sigset_t **mask*);

Блокировка SIGCHLD в обработчике

```
void chld_handler(int signum) {
sigset_t set;
if (sigemptyset(&set)) { return; } // обнуляем маску
// добавляем сигнал SIGCHLD в маску
if (sigaddset(&set, SIGCHLD)) { return; }
// блокируем SIGCHLD
if (sigprocmask(SIG_BLOCK, &set, NULL)) { return; }

/* обработка сигнала*/
// разблокируем SIGCHLD
if (sigprocmask(SIG_UNBLOCK, &set, NULL)) { return; }
return; }
```

Ожидание сигнала

int pause(void);

После вызова функции вызывающий процесс приостанавливается до тех пор, пока не получит сигнал. Данный сигнал либо остановит процесс, либо заставит его вызвать функцию обработки этого сигнала.

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

Первым параметром функции sigwait() является указатель на набор сигналов, получения которых будет ждать функция. Во втором параметре sigwait() вернет номер того сигнала, который возобновил работу программы.

Данная функция позволяет приостановить выполнении процесса до получения нужного сигнала (или одного из набора сигналов). Перед тем как вызывать sigwait(), набор ожидаемых сигналов следует заблокировать с помощью функции sigprocmask(), иначе, при получении сигнала, вместо выхода из sigwait() будет вызван соответствующий обработчик.

Алгоритм обработки сигнала

алгоритм psig {

выбрать номер сигнала из дескриптора процесса;

очистить поле с номером сигнала;

если (установлен признак игнорировать сигнал данного типа) возвратить управление;

если (пользователь указал функцию, которую нужно выполнить по получении сигнала) {

из пространства процесса выбрать пользовательский виртуальный адрес функции обработки сигнала;

внести изменения в пользовательский контекст: искусственно создать в стеке задачи запись, имитирующую обращение к функции обработки сигнала;

внести изменения в системный контекст: записать адрес функции обработки сигнала в поле счетчика команд, принадлежащее сохраненному регистровому контексту задачи;

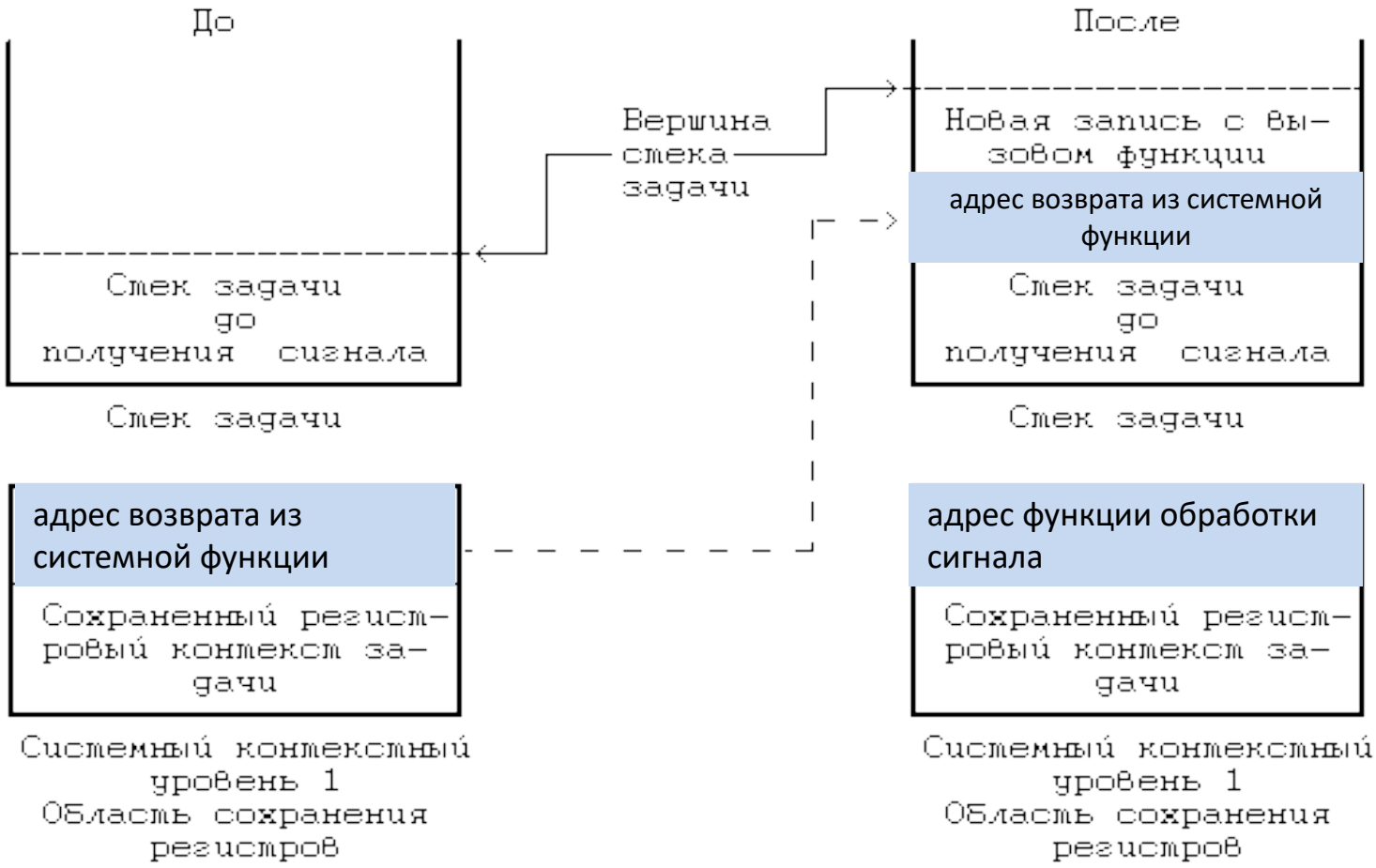
возвратить управление;

}

немедленно запустить алгоритм exit;

}

Стек задачи и область сохранения структур ядра до и после получения сигнала



Сигналы, передающиеся через fork и exec

В потомке после fork:

- установленные обработчики остаются на месте
- заблокированные сигналы остаются заблокированными
- ожидающие в родителе сигналы сбрасываются
- маска блокированных сигналов наследуется

В потомке после exec:

- Все обработчики сбрасываются в состояние с действием по умолчанию

Пример обработки сигнала

```
#include <signal.h>
void Fsig ( )   /* перехватывающая функция*/
{
 kill (0, SIGINT);  /* посылка сигнала SIGINT предку и всем потомкам*/
}
main ( )
{
    int i;
    sigset_t mask;
    signal ( SIGALRM, Fsig); /* установка реакции на сигнал SIGALRM*/
    for (i=1; i<5; i++) {
         if (fork()==0) {              /* порождение потомков*/
         printf("запущен потомок с идентификатором %d\n", getpid() );
         pause();     /* потомки ожидают сигнал SIGINT и, получив его, завершают работу*/
         }
    }
/* продолжение предка*/
 sigemptyset(&mask); /* очистка маски*/
 sigaddset(&mask,SIGINT); /* маска для сигнала SIGINT*/
  sigprocmask (SIG_BLOCK, &mask, 0); /* маскирование сигнала SIGINT   для предка*/
  alarm(10);    /*установка программного таймера на посылку сигнала   SIGALRM предку через 10 с */
  pause();       /* предок  ждет сигнала SIGALRM*/
  printf ("нормальное окончание работы предка \n");
}
```

Задание к лабораторной работе 5

Напишите программу, которая реагирует на ошибки при выполнении операции деления и неверном использовании указателя (деление на ноль, нарушение защиты памяти). При обнаружении ошибки программа должна передать управление функции, которая выведет сообщение и завершит работу программы с кодом ошибки (1 или 2). Тип ошибки, который должна зафиксировать программа, задается как параметр при ее запуске.