UNIX

Лекция 6

СИГНАЛЫ

Прерывания и особые ситуации

Прерывания. Внешние устройства ввода-вывода, системные часы и т.п. асинхронно прерывают работу ЦП. По получении сигнала прерывания ядро операционной системы сохраняет свой текущий контекст и обрабатывает прерывание. Далее прерванный контекст восстановится. Устройствам обычно приписываются приоритеты в соответствии с очередностью обработки прерываний.

Особые ситуации связаны с возникновением незапланированных событий, вызванных процессом, таких как недопустимая адресация, задание привилегированных команд, деление на нуль и т.д.

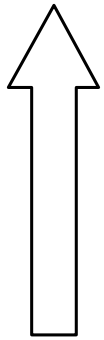
- Они отличаются от прерываний, которые вызываются событиями, внешними по отношению к процессу.
- Особые ситуации возникают во время выполнения команды, и система, обработав особую ситуацию, пытается перезапустить команду; в отличие от этого считается, что прерывания возникают между выполнением двух команд, при этом система после обработки прерывания продолжает выполнение процесса уже начиная со следующей команды.

Для обработки прерываний и особых ситуаций в системе UNIX используется один и тот же механизм.

Уровни прерывания процессора

- Ядро иногда обязано предупреждать возникновение прерываний во время критических действий, могущих в случае прерывания запортить информацию. Обычно имеется ряд привилегированных команд, устанавливающих уровень прерывания процессора в т.н. *слове состояния процессора*.
- Установка уровня прерывания на определенное значение отсекает прерывания этого и более низких уровней, разрешая обработку только прерываний с более высоким приоритетом.

- Машинные сбои
- Системные часы
- Диск
- Сетевое оборудование
- Терминалы
- Программные прерывания



Высокий приоритет

Низкий приоритет

Сигналы

- Сигнал - это программное средство, с помощью которого может быть прервано функционирование процесса.
- Сигналы сообщают процессам о возникновении асинхронных событий. Механизм сигналов позволяет процессам реагировать на различные события, которые могут происходить в ходе работы процесса внутри него самого или во внешней среде.
- Сигналы инициируются событиями и посылаются в процесс для уведомления его о том, что произошло нечто неординарное, требующее определенного действия.
- Событие может вызываться процессом, пользователем или ядром UNIX. Например, если процесс попытается выполнить математическую операцию “деление на нуль” ядро пошлет такому процессу сигнал, который прервет его.
- Родительский процесс и порожденные процессы могут посылать друг другу сигналы для синхронизации.
- Сигналы являются программной версией аппаратных прерываний.

Описание сигналов

- Сигналы описаны в файле <signal.h>. Количество и семантика сигналов зависят от версии ОС UNIX. В версии System V сигналы имеют номера от 1 до 19:

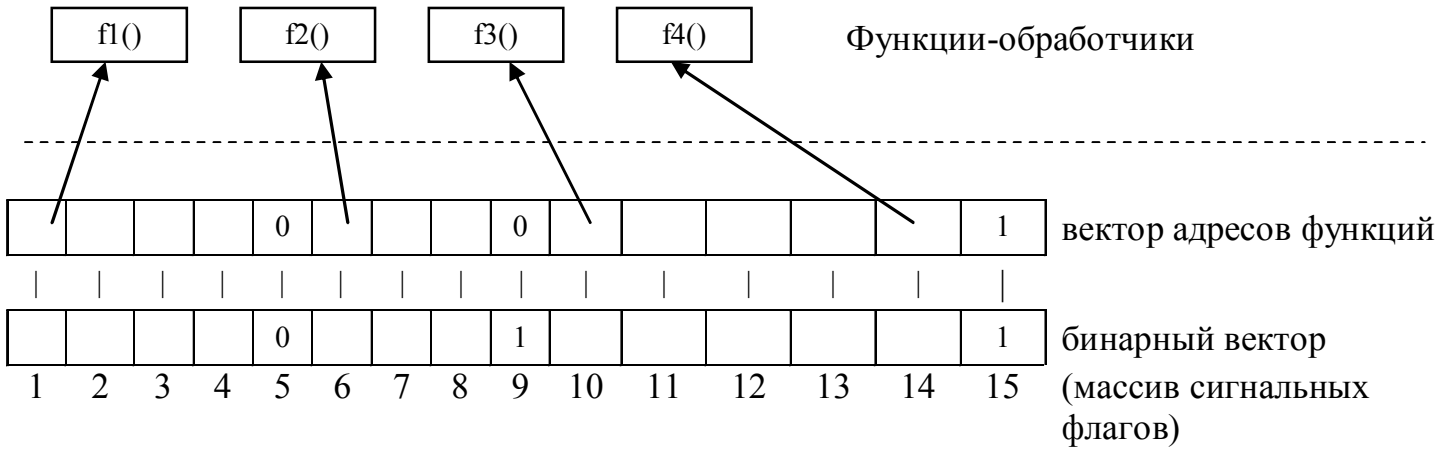
```
#define NSIG                20
#define SIGHUP      1       /* разрыв связи */
#define SIGINT      2       /* прерывание */
#define SIGQUIT     3       /* аварийный выход */
#define SIGILL      4       /* неверная машинная инструкция */
#define SIGTRAP     5       /* прерывание-ловушка */
#define SIGIOT      6       /* прерывание ввода-вывода */
#define SIGEMT      7       /* программное прерывание EMT */
#define SIGFPE      8       /* авария при выполнении операции с плавающей точкой */
#define SIGKILL     9       /* уничтожение процесса */
#define SIGBUS      10      /* ошибка шины */
#define SIGSEGV     11      /* нарушение сегментации */
#define SIGSYS      12      /* ошибка выполнения системного вызова */
#define SIGPIPE     13      /* запись в канал есть, чтения нет */
#define SIGALRM     14      /* прерывание от таймера */
#define SIGTERM     15      /* программный сигнал завершения от kill */
#define SIGUSR1     16      /* определяется пользователем */
#define SIGUSR2     17      /* определяется пользователем */
#define SIGCLD      18      /* процесс-потомок завершился */
#define SIGPWR      19      /* авария питания */
#define SIG_DFL     (int(*)())0   /* все установки «по умолчанию» */
#define SIG_IGN     (int(*)())1   /* игнорировать этот сигнал */
```

Причины возникновения сигналов

В System V возникновение сигналов можно классифицировать следующим образом:

- введение пользователем управляющего символа с терминала всем процессам, ассоциированным с данным терминалом (SIGINT, SIGQUIT, SIGHUP);
- возникновение аварийной ситуации при функционировании пользовательского процесса (SIGILL, SIGTRAP, SIGFPE, SIGBUS, SIGSEGV, SIGSYS, SIGPIPE);
- возникновение непредусмотренного или не поддающегося идентификации события (SIGTERM, SIGCLD, SIGPWR);
- возникновение некоторого заранее описанного события (SIGALRM).

Посылка сигналов производится процессами друг другу с помощью функции ***kill*** или ядром.



Обработка сигналов

- Ядро обрабатывает сигналы в контексте того процесса, который получает их.
- Существуют три способа обработки сигналов:
  1. реакция по умолчанию,
  2. игнорирование сигнала,
  3. выполнение особой (пользовательской) функции по его получении.
- Реакцией по умолчанию - обычно вызов функции *exit*().
- Обрабатывая сигнал, ядро определяет тип сигнала и очищает (гасит) разряд в записи таблицы процессов, соответствующий данному типу сигнала и установленный в момент получения сигнала процессом. Таким образом, когда процесс получает любой неигнорируемый им сигнал (за исключением SIGILL и SIGTRAP), UNIX автоматически восстанавливает реакцию «по умолчанию» на всякое последующее получение этого сигнала.

Замечание 1: Если необходима многократная обработка одного и того же сигнала, процесс должен каждый раз осуществлять системный вызов ***signal*** для установления требуемой реакции на данный сигнал.

Замечание 2: Процесс не в состоянии узнать, сколько однотипных сигналов им было получено. В том случае, если процесс не успевает обработать все поступившие сигналы, происходит потеря информации.

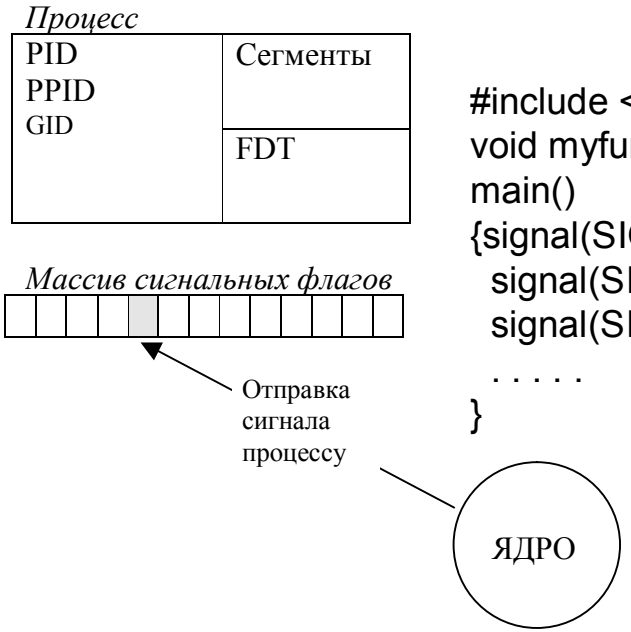
Функции

signal (int sign, void(*func)(int))

sign номер сигнала,
*void(*func)(int)* адрес функции обработки сигнала,
void(*)0 - указание на использование реакции по умолчанию,
void(*)1 - игнорирование сигнала.

int kill(pid, sig)

- pid>0: ядро посылает сигнал процессу с идентификатором pid.
- pid=0: сигнал посылается всем процессам, входящим в одну группу с процессом, вызвавшим функцию *kill*.
- pid=-1, сигнал посылается всем процессам, у которых реальный код идентификации пользователя совпадает с тем, под которым исполняется процесс, вызвавший функцию *kill*.
- pid<0, pid!=-1: сигнал посылается всем процессам, входящим в группу с номером, равным |pid|.

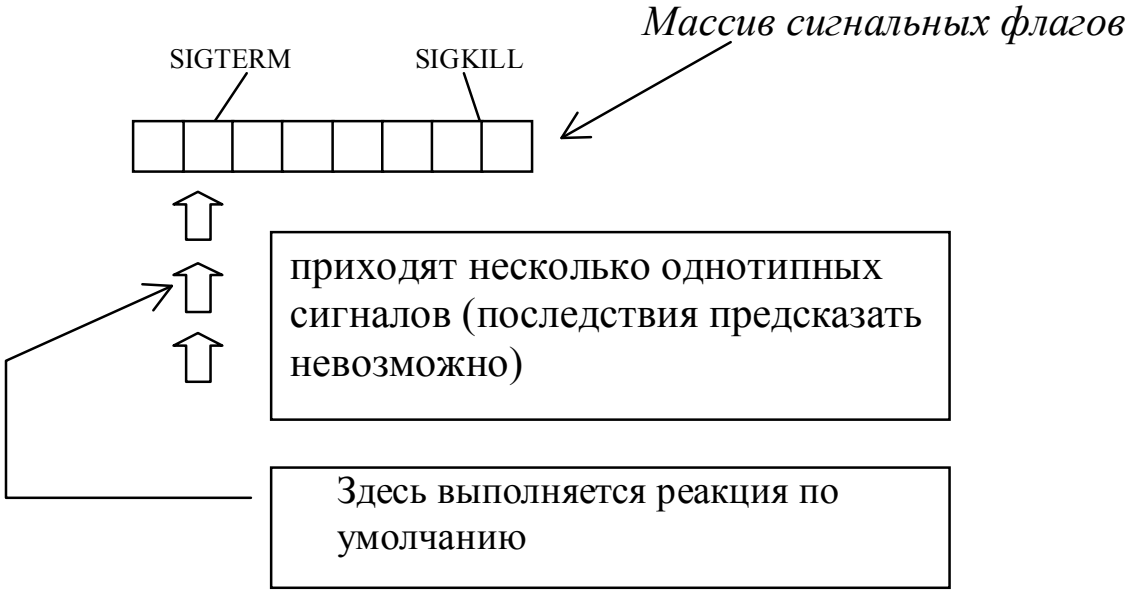



```
#include <signal.h>
void myfunc (int sign);
main()
{signal(SIGTERM, myfunc);  // Сигнал, который завершает процесс
  signal(SIGUSR1, SIG_IGN);  // Игнорировать, вместо 1 пишем SIG_IGN
  signal(SIGKILL, SIG_DFL);  // По умолчанию, вместо 0 пишем SIG_DFL
  . . . . .
}
```

Особенности сигналов

- Сигналы могут теряться;
- При обработке сигналов необходимо восстанавливать диспозицию

```
#include <signal.h>
void myfunc (int sign)
{    signal(sign, myfunc); // чтобы далее не было реакции
                           // по умолчанию, восстанавливаем обработчик
     …
}
```



Пример

- П1 порождает П2.
- Далее П1 выполняет некоторые действия, затем инициирует выполнение действий П2, который затем вновь инициирует П1

Схема П1=>П2=>П1.

Не совсем корректная программа

```
#include <unistd.h>
#include <signal.h>
#include <fcntl.h>
void f(int n)
{     signal(n,f);
      printf("\n** process %d has got a signal\n",getpid());
}
main()
{     int pid;
      signal(SIGUSR1,f);
      switch(pid=fork())
      {        case 0:     printf("\nchild %d: wait\n",getpid());
                           pause();
                           printf("\nCHILD %d works...\n",getpid());
                           kill(getppid(),SIGUSR1);
                           break;
               default:    printf("\nPARENT %d works-1\n",getpid());
                           kill(pid,SIGUSR1);
                           pause();
                           printf("\nPARENT %d works-2\n",getpid());
      }
}
```

Более правильная программа

```
#include <unistd.h>
#include <signal.h>
#include <fcntl.h>
int lock = 1;
void f(int n)
{       signal(n,f);
        printf("\n** process %d has got a signal\n",getpid());
        lock = 0;
}
main()
{       int pid;
        signal(SIGUSR1,f);
        switch(pid=fork())
        {           case 0:     printf("\nchild %d: wait\n",getpid());
                                while(lock);
                                printf("\nCHILD %d works...\n",getpid());
                                kill(getppid(),SIGUSR1);
                                break;
                    default:    printf("\nPARENT %d works-1\n",getpid());
                                kill(pid,SIGUSR1);
                                pause();
                                printf("\nPARENT %d works-2\n",getpid());
        }
}
```