6.4. Сигналы.

Процессы в *UNIX* используют много разных механизмов взаимодействия. Одним из них являются **сигналы.**

Сигналы – это **асинхронные** события. Что это значит? Сначала объясним, что такое **синхронные** события: я два раза в день подхожу к почтовому ящику и проверяю – нет ли в нем почты (событий). Во-первых, я произвожу **опрос** – "нет ли для меня события?", в программе это выглядело бы как вызов функции опроса и, может быть, ожидания события. Во-вторых, я знаю, что почта **может** ко мне прийти, поскольку я подписался на какие-то газеты. То есть я предварительно **заказывал** эти события.

Схема с синхронными событиями очень распространена. Кассир сидит у кассы и ожидает, пока к нему в окошечко не заглянет клиент. Поезд периодически проезжает мимо светофора и останавливается, если горит красный. Функция Си пассивно "спит" до тех пор, пока ее не вызовут; однако она всегда готова выполнить свою работу (обслужить клиента). Такое ожидающее заказа (события) действующее лицо называется **сервер.** После выполнения заказа сервер вновь переходит в состояние ожидания вызова. Итак, если событие **ожидается** в специальном месте и в определенные моменты времени (издается некий вызов для ОПРОСА) – это синхронные события. Канонический пример – функция *gets*, которая задержит выполнение программы, пока с клавиатуры не будет введена строка. Большинство ожиданий внутри системных вызовов – **синхронны.** Ядро ОС выступает для программ пользователей в роли сервера, выполняющего сисвызовы (хотя и не только в этой роли – ядро иногда предпринимает и активные действия: передача процессора другому процессу через определенное время (режим разделения времени), убивание процесса при ошибке, и.т.п.).

Сигналы – это асинхронные события. Они приходят неожиданно, в любой момент времени – вроде телефонного звонка. Кроме того, их не требуется заказывать – сигнал процессу может поступить совсем без повода. Аналогия из жизни такова: человек сидит и пишет письмо. Вдруг его окликают посреди фразы – он отвлекается, отвечает на вопрос, и вновь продолжает прерванное занятие. Человек **не ожидал** этого оклика (быть может, он **готов** к нему, но он не озирался по сторонам специально). Кроме того, сигнал мог поступить когда он писал 5-ое предложение, а мог – когда 34-ое. Момент времени, в который произойдет прерывание, не фиксирован.

Сигналы имеют **номера**, причем их количество ограничено – есть определенный список допустимых сигналов. Номера и мнемонические имена сигналов перечислены в includeфайле <**signal.h**> и имеют вид *SIG***нечто.** Допустимы сигналы с номерами 1..*NSIG*–1, где *NSIG* определено в этом файле. При получении сигнала мы узнаем его номер, но не узнаем никакой иной информации: ни **от кого** поступил сигнал, ни что от нас хотят. Просто "звонит телефон". Чтобы получить дополнительную информацию, наш процесс должен взять ее из другого известного места; например – прочесть заказ из некоторого файла, об имени которого все наши программы заранее "договорились". Сигналы процессу могут поступать тремя путями:

- От другого процесса, который **явно** посылает его нам вызовом

  ```
  kill(pid, sig);
  ```

  где **pid** – идентификатор (номер) процесса-получателя, а **sig** – номер сигнала. Послать сигнал можно только родственному процессу – запущенному тем же пользователем.
- От операционной системы. Система может посылать процессу ряд сигналов, сигнализирующих об ошибках, например при обращении программы по несуществующему адресу или при ошибочном номере системного вызова. Такие сигналы обычно прекращают наш процесс.
- От пользователя – с клавиатуры терминала можно нажимом некоторых клавиш послать сигналы *SIGINT* и *SIGQUIT*. Собственно, сигнал посылается **драйвером терминала** при получении им с клавиатуры определенных символов. Так можно прервать зациклившуюся или надоевшую программу.

Процесс-получатель должен как-то отреагировать на сигнал. Программа может:

- проигнорировать сигнал (не ответить на звонок);
- перехватить сигнал (снять трубку), выполнить какие-то действия, затем продолжить прерванное занятие;
- быть убитой сигналом (звонок был подкреплен броском гранаты в окно);

В большинстве случаев сигнал по умолчанию убивает процесс-получатель. Однако процесс может изменить это умолчание и задать свою реакцию явно. Это делается вызовом *signal*:

```
#include <signal.h>
void (*signal(int sig, void (*react)() )) ();
```

Параметр **react** может иметь значение:

SIG_IGN
: сигнал **sig** будет отныне игнорироваться. Некоторые сигналы (например *SIGKILL*) невозможно перехватить или проигнорировать.

SIG_DFL
: восстановить реакцию по умолчанию (обычно - смерть получателя). **имя_функции** Например

```
void fr(gotsig){ ..... }  /* обработчик */
... signal (sig, fr); ... /* задание реакции */
```

Тогда при получении сигнала **sig** будет вызвана функция *fr*, в которую в качестве аргумента **системой** будет передан номер сигнала, действительно вызвавшего ее **gotsig==sig**. Это полезно, т.к. можно задать одну и ту же функцию в качестве реакции для **нескольких** сигналов:

```
... signal (sig1, fr); signal(sig2, fr); ...
```

После возврата из функции *fr*() программа продолжится с прерванного места. **Перед** вызовом функции-обработчика реакция автоматически сбрасывается в реакцию по умолчанию *SIG_DFL*, а после выхода из обработчика снова восстанавливается в *fr*. Это значит, что во время работы функции-обработчика может прийти сигнал, который **убьет** программу.

Приведем список **некоторых** сигналов; полное описание посмотрите в документации. Колонки таблицы: *G* - может быть перехвачен; *D* - по умолчанию убивает процесс (*k*), игнорируется (*i*); *C* - образуется дамп памяти процесса: файл *core*, который затем может быть исследован отладчиком *adb*; *F* - реакция на сигнал сбрасывается; *S* - посылается обычно системой, а не явно.

сигнал	G	D	C	F	S	смысл
SIGTERM	+	k	-	+	-	завершить процесс
SIGKILL	-	k	-	+	-	убить процесс
SIGINT	+	k	-	+	-	прерывание с клавиш
SIGQUIT	+	k	+	+	-	прерывание с клавиш
SIGALRM	+	k	-	+	+	будильник
SIGILL	+	k	+	-	+	запрещенная команда
SIGBUS	+	k	+	+	+	обращение по неверному
SIGSEGV	+	k	+	+	+	адресу
SIGUSR1, USR2	+	i	-	+	-	пользовательские
SIGCLD	+	i	-	+	+	смерть потомка

- Сигнал *SIGILL* используется иногда для эмуляции команд с плавающей точкой, что происходит примерно так: при обнаружении "запрещенной" команды для отсутствующего процессора "плавающей" арифметики аппаратура дает прерывание и система посылает процессу сигнал *SIGILL*. По сигналу вызывается функция-эмулятор плавающей арифметики (подключаемая к выполняемому файлу автоматически), которая и обрабатывает требуемую команду. Это может происходить много раз, именно поэтому реакция на этот сигнал **не сбрасывается.**
- *SIGALRM* посылается в результате его заказа вызовом *alarm*() (см. ниже).
- Сигнал *SIGCLD* посылается процессу-родителю при выполнении процессом-потомком сисвызова *exit* (или при смерти вследствие получения сигнала). Обычно процессродитель при получении такого сигнала (если он его заказывал) реагирует, выполняя в обработчике сигнала вызов *wait* (см. ниже). По-умолчанию этот сигнал игнорируется.
- Реакция *SIG_IGN* **не** сбрасывается в *SIG_DFL* при приходе сигнала, т.е. сигнал игнорируется постоянно.
- Вызов *signal* возвращает старое значение реакции, которое может быть запомнено в переменную вида void (***f**)(); а потом восстановлено.
- Синхронное ожидание (сисвызов) может иногда быть прервано асинхронным событием (сигналом), но об этом ниже.

Некоторые версии *UNIX* предоставляют более развитые средства работы с сигналами. Опишем некоторые из средств, имеющихся в *BSD* (в других системах они могут быть смоделированы другими способами).

Пусть у нас в программе есть "критическая секция", во время выполнения которой приход сигналов нежелателен. Мы можем "заморозить" (заблокировать) сигнал, отложив момент его поступления до "разморозки":

```
sighold(sig);  заблокировать сигнал
```

```
   |          :
 КРИТИЧЕСКАЯ  :<---процессу послан сигнал sig,
 СЕКЦИЯ       :  но он не вызывает реакцию немедленно,
   |          :  а "висит", ожидая разрешения.
   |          :
sigrelse(sig); разблокировать
   |<---------- sig
   |    накопившиеся сигналы доходят,
   |    вызывается реакция.
```

Если во время блокировки процессу было послано **несколько** одинаковых сигналов **sig**, то при разблокировании поступит **только один.** Поступление сигналов во время блокировки просто отмечается в специальной битовой шкале в паспорте процесса (примерно так):

```
mask |= (1 << (sig - 1));
```

и при разблокировании сигнала **sig**, если соответствующий бит выставлен, то приходит один такой сигнал (система вызывает функцию реакции). То есть *sighold* заставляет приходящие сигналы "накапливаться" в специальной маске, вместо того, чтобы немедленно вызывать реакцию на них. А *sigrelse* разрешает "накопившимся" сигналам (если они есть) прийти и вызывает реакцию на них. Функция

```
sigset(sig, react);
```

аналогична функции *signal*, за исключением того, что на время работы обработчика сигнала **react**, приход сигнала **sig** блокируется; то есть перед вызовом **react** как бы делается *sighold*, а при выходе из обработчика – *sigrelse*. Это значит, что если во время работы обработчика сигнала придет такой же сигнал, то программа не будет убита, а "запомнит" пришедший сигнал, и обработчик будет вызван повторно (когда сработает *sigrelse*).

Функция

```
sigpause(sig);
```

вызывается внутри "рамки"

```
sighold(sig);
      ...
   sigpause(sig);
      ...
sigrelse(sig);
```

и вызывает задержку выполнения процесса до прихода сигнала **sig**. Функция разрешает приход сигнала **sig** (обычно на него должна быть задана реакция при помощи *sigset*), и "засыпает" до прихода сигнала **sig**.

В *UNIX* стандарта *POSIX* для управления сигналами есть вызовы *sigaction*, *sigproc- mask*, *sigpending*, *sigsuspend*. Посмотрите в документацию!

6.4.1. Напишите программу, выдающую на экран файл **/etc/termcap**. Перехватывайте сигнал *SIGINT*, при получении сигнала запрашивайте "**Продолжать**?". По ответу 'y' – продолжить выдачу; по 'n' – завершить программу; по 'r' – начать выдавать файл с начала: *lseek*(**fd**,0L,0). Не забудьте заново переустановить реакцию на *SIGINT*, поскольку после получения сигнала реакция автоматически сбрасывается.

```
#include <signal.h>
void onintr(sig){       /* sig - номер сигнала  */
  signal (sig, onintr); /* восстановить реакцию */
  ... запрос и действия ...
}
main(){ signal (SIGINT, onintr); ... }
```

Сигнал прерывания можно игнорировать. Это делается так:

```
signal (SIGINT, SIG_IGN);
```

Такую программу нельзя прервать с клавиатуры. Напомним, что реакция *SIG_IGN* сохраняется при приходе сигнала.

6.4.2. Системный вызов, находящийся в состоянии ожидания какого-то события (*read* ждущий нажатия кнопки на клавиатуре, *wait* ждущий окончания процесса-потомка, и.т.п.), может быть **прерван** сигналом. При этом сисвызов вернет значение "ошибка" (-1) и *errno* станет равно *EINTR*. Это позволяет нам писать системные вызовы с выставлением **тайма- ута:** если событие не происходит в течение заданного времени, то завершить ожидание и прервать сисвызов. Для этой цели используется вызов *alarm*(**sec**), заказывающий посылку сигнала *SIGALRM* нашей программе через целое число **sec** секунд (0 – отменяет заказ):

```
#include <signal.h>
void (*oldaction)(); int alarmed;
/* прозвонил будильник */
void onalarm(nsig){ alarmed++; }
   ...
/* установить реакцию на сигнал */
oldaction = signal (SIGALRM, onalarm);
/* заказать будильник через TIMEOUT сек. */
alarmed = 0; alarm ( TIMEOUT /* sec */ );

     sys_call(...);  /* ждет события */
  // если нас сбил сигнал, то по сигналу будет
  // еще вызвана реакция на него - onalarm

if(alarmed){
  // событие так и не произошло.
  // вызов прерван сигналом т.к. истекло время.
}else{
  alarm(0); /* отменить заказ сигнала */
  // событие произошло, сисвызов успел
  // завершиться до истечения времени.
}
signal (SIGALRM, oldaction);
```

Напишите программу, которая ожидает ввода с клавиатуры в течение 10 секунд. Если ничего не введено – печатает "**Нет ввода**", иначе – печатает "**Спасибо**". Для ввода можно использовать как вызов *read*, так и функцию *gets* (или *getchar*), поскольку функция эта все равно внутри себя издает системный вызов *read*. Исследуйте, какое значение возвращает *fgets* (*gets*) в случае прерывания ее системным вызовом.

```
/* Копирование стандартного ввода на стандартный вывод
 * с установленным тайм-аутом.
 * Это позволяет использовать программу для чтения из FIFO-файлов
 * и с клавиатуры.
 * Небольшая модификация позволяет использовать программу
 * для копирования "растущего" файла (т.е. такого, который в
 * настоящий момент еще продолжает записываться).
 * Замечание:
 *       В ДЕМОС-2.2 сигнал НЕ сбивает чтение из FIFO-файла,
 *       а получение сигнала откладывается до выхода из read()
 *       по успешному чтению информации. Пользуйтесь open()-ом
 *       с флагом O_NDELAY, чтобы получить требуемый эффект.
 *
 *      Вызов: a.out /dev/tty
 *
 * По мотивам книги М.Дансмура и Г.Дейвиса.
 */

#define WAIT_TIME 5 /* ждать 5 секунд */
#define MAX_TRYS  5 /* максимум 5 попыток */
#define BSIZE     256
#define STDIN     0 /* дескриптор стандартного ввода  */
#define STDOUT    1 /* дескриптор стандартного вывода */

#include <signal.h>
#include <errno.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
char buffer [ BSIZE ];
extern int errno;       /* код ошибки */

void timeout(nsig){ signal( SIGALRM, timeout ); }
void main(argc, argv) char **argv;{
   int fd, n, trys = 0;  struct stat stin, stout;

   if( argc != 2 ){
       fprintf(stderr, "Вызов: %s файл\n", argv[0]); exit(1);
   }
   if((fd = !strcmp(argv[1],"-")? STDIN : open(argv[1],O_RDONLY)) < 0){
       fprintf(stderr, "Не могу читать %s\n", argv[1]); exit(2);
   }
   /* Проверить, что ввод не совпадает с выводом,
    *     hardcat aFile >> aFile
    * кроме случая, когда вывод - терминал.
    * Такая проверка полезна для программ-фильтров (STDIN->STDOUT),
    * чтобы исключить порчу исходной информации */
   fstat(fd, &stin); fstat(STDOUT, &stout);
   if( !isatty(STDOUT) && stin.st_ino == stout.st_ino &&
                          stin.st_dev == stout.st_dev
   ){ fprintf(stderr,
      "\aВвод == выводу, возможно потеряна информация в %s.\n",argv[1]);
      exit(33);
```

```
        }

        signal( SIGALRM, timeout );
        while( trys < MAX_TRYS ){
                alarm( WAIT_TIME ); /* заказать сигнал через 5 сек */

                /* и ждем ввода ... */
                n = read( fd, buffer, BSIZE );

                alarm(0);       /* отменили заказ сигнала */
                      /* (хотя, возможно, он уже получен) */

                /* проверяем: почему мы слезли с вызова read() ? */
                if( n < 0 && errno == EINTR ){
                    /* Мы были сбиты сигналом SIGALRM,
                     * код ошибки EINTR - сисвызов прерван
                     * неким сигналом.
                     */
                    fprintf( stderr, "\7timed out (%d раз)\n", ++trys );
                    continue;
                }

                if( n < 0 ){
                    /* ошибка чтения */
                    fprintf( stderr, "read error.\n" ); exit(4);
                }
                if( n == 0 ){
                    /* достигнут конец файла */
                    fprintf( stderr, "Достигнут EOF.\n\n" ); exit(0);
                }
                /* копируем прочитанную информацию */
                write( STDOUT, buffer, n );
                trys = 0;
        }
        fprintf( stderr, "Все попытки провалились.\n" ); exit(5);
    }
```

Если мы хотим, чтобы сисвызов не мог прерываться сигналом, мы должны защитить его:

```
    #include <signal.h>
    void (*fsaved)();
            ...
    fsaved = signal (sig, SIG_IGN);
       sys_call(...);
    signal (sig, fsaved);
```

или так:

```
    sighold(sig);
       sys_call(...);
    sigrelse(sig);
```

Сигналами могут быть прерваны **не все** системные вызовы и не при всех обстоятельствах.

6.4.3. Напишите функцию *sleep*(**n**), задерживающую выполнение программы на **n** секунд. Воспользуйтесь системным вызовом *alarm*(**n**) (будильник) и вызовом *pause*(), который задерживает программу до получения **любого** сигнала. Предусмотрите рестарт при получении во время ожидания другого сигнала, нежели *SIGALRM*. Сохраняйте заказ *alarm*, сделанный до вызова *sleep* (*alarm* выдает число секунд, оставшееся до завершения предыдущего заказа). На самом деле есть такая СТАНДАРТНАЯ функция. Ответ:

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

    int got;   /* пришел ли сигнал */

    void onalarm(int sig)
    { printf( "Будильник\n" ); got++; } /* сигнал получен */

    void sleep(int n){
       time_t time(), start = time(NULL);
       void (*save)();
       int oldalarm, during = n;

       if( n <= 0 ) return;
       got = 0;
       save = signal(SIGALRM, onalarm);
       oldalarm = alarm(3600); /* Узнать старый заказ */
       if( oldalarm ){
           printf( "Был заказан сигнал, который придет через %d сек.\n",
                    oldalarm );
```

```
            if(oldalarm > n) oldalarm -= n;
            else { during = n = oldalarm; oldalarm = 1; }
        }
        printf( "n=%d oldalarm=%d\n", n, oldalarm );
        while( n > 0 ){
          printf( "alarm(%d)\n", n );
          alarm(n);  /* заказать SIGALRM через n секунд */

          pause();

          if(got) break;
          /* иначе мы сбиты с pause другим сигналом */
          n = during - (time(NULL) - start); /* прошло времени */
        }
        printf( "alarm(%d) при выходе\n", oldalarm );
        alarm(oldalarm);  /* alarm(0) - отмена заказа сигнала */
        signal(SIGALRM, save); /* восстановить реакцию */
    }

    void onintr(int nsig){
        printf( "Сигнал SIGINT\n"); signal(SIGINT, onintr);
    }

    void onOldAlarm(int nsig){
        printf( "Звонит старый будильник\n");
    }

    void main(){
            int time1 = 0;  /* 5, 10, 20 */
            setbuf(stdout, NULL);
            signal(SIGINT, onintr);
            signal(SIGALRM, onOldAlarm); alarm(time1);
                        sleep(10);
            if(time1) pause();
            printf("Чао!\n");
    }
```

6.4.4. Напишите "часы", выдающие текущее время каждые 3 секунды.

```
    #include <signal.h>
    #include <time.h>
    #include <stdio.h>
    void tick(nsig){
        time_t tim; char *s;
        signal (SIGALRM, tick);
        alarm(3); time(&tim);
        s = ctime(&tim);
        s[ strlen(s)-1 ] = '\0'; /* обрубить '\n' */
        fprintf(stderr, "\r%s", s);
    }
    main(){ tick(0);
            for(;;) pause();
    }
```

© Copyright А. Богатырев, 1992-95
Си в UNIX

Рейти ТОП Разработка игр