

Системное программирование

Лекция 4

Сигналы

Сигналы

Сигнал – уведомление о произошедшем событии.

- Стандарт POSIX определяет ряд стандартных сигналов, каждый из которых уведомляет о событии конкретного типа.
- Сигнал может быть послан процессу со стороны ОС, другого процесса или послан сам себе.
- Сигналы прерывают выполнение программы. Если сигнал не уничтожит процесс, то выполнение будет продолжено с того же места, где оно было прервано сигналом (т.е., словно сигнала не было).
- Сигнал может быть доставлен процессу в любой момент времени* -> программа должна быть готова к тому, что сигнал придет неожиданно.

* Сигнал, отправляемый процессу, сохраняется в ядре ОС. Проверка наличия сигналов с последующей доставкой сигнала процессу происходит при переключении между ядром ОС и программой, на что может происходить по 3 причинам – выполнение системных вызовов, обработка аппаратных прерываний/исключений и переключение между процессами по требованию планировщика. Поскольку последние 2 причины от процесса не зависят, можно считать, что сигнал может прийти в любой момент времени

Диспозиция сигнала

Диспозиция сигнала – действие, выполняемое при получении сигнала. Существует 5 диспозиций:

- Уничтожить процесс (диспозиция по умолчанию для SIGKILL, SIGTERM, SIGQUIT);
- Уничтожить процесс с созданием дампа (снимка) памяти (SIGSEGV, SIGQUIT, SIGABRT);
- Остановить процесс (SIGSTOP, SIGTSTP, SGTIN);
- Выполнить заданную функцию-обработчик сигнала;
- Ничего не делать (игнорировать сигнал).

Диспозицию сигналов **SIGKILL** и **SIGSTOP** изменить нельзя. SIGKILL всегда уничтожает процесс, SIGSTOP всегда останавливает процесс.

Сигнал **SIGCONT** по умолчанию игнорируется, но всегда «будит» остановленный процесс.

Вызов kill (пример 1)

Послать сигнал процессу можно вызовом `kill()` :

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

Параметры:

`pid` – PID целевого процесса*;

`sig` – номер сигнала (константы `SIG*` – `SIGKILL`, `SIGTERM`,...)

Если `sig==0`, то никакой сигнал не посылается, но проверяется существование процесса и производятся проверки прав на возможность отправки сигналов этому процессу.

* при `pid>0`, для остальных случаев см. *man 2 kill*
См. также: функция `raise()`, команда оболочки *kill*

Распространенные сигналы

- SIGKILL – уничтожить процесс;
- SIGSTOP – остановить процесс;
- SIGCONT – продолжить выполнение процесса (см. *man fg*);
- SIGCHLD – изменение состояния дочернего процесса;
- SIGINT – прерывание с терминала (Ctrl-C);
- SIGTSTP – остановка процесса с терминала (Ctrl-Z);
- SIGQUIT – завершение процесса с терминала (Ctrl-\\);
- SIGUSR1, SIGUSR2 – пользовательские сигналы.

Общий список: *man 7 signal*

Блокировка сигналов

Блокировка сигнала – откладывание получения сигнала.

Блокируемые сигналы определяются **маской сигналов**. Сигналы, указанные в маске, блокируются.

Маска сигналов сохраняется при `fork()`.

Существует коренное отличие между блокировкой и игнорированием сигнала.

- Игнорируемый сигнал доставляется процессу, но процесс на него не реагирует.
- Блокируемый сигнал сохраняется в ядре до тех пор, пока не снята блокировка. После снятия блокировки сигнал при первой возможности доставляется процессу и обрабатывается согласно диспозиции.

Если во время блокировки один и тот же сигнал был послан N раз, то процессу будет доставлен только 1 сигнал.

SIGKILL и SIGSTOP не могут быть заблокированы.

Маска сигналов

Для хранения маски сигналов служит тип `sigset_t`. Для работы с ним существует ряд функций (*man 3 sigsetops*):

```
/*инициализировать пустую маску*/  
int sigemptyset(sigset_t* set);  
  
/*инициализировать заполненную маску*/  
int sigfillset(sigset_t* set);  
  
/*добавить сигнал в маску*/  
int sigaddset(sigset_t* set, int signum);  
  
/*удалить сигнал из маски*/  
int sigdelset(sigset_t* set, int signum);  
  
/*проверить, что сигнал входит в маску*/  
int sigismember(const sigset_t* set, int signum);
```

Вызов sigprocmask

Изменение маски сигналов процесса производится вызовом `sigprocmask()`.

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

Параметры:

- `how` – тип операции (константы `SIG_BLOCK`, `SIG_UNBLOCK`, `SIG_SETMASK`);
- `set` – указатель на новую набор сигналов [опционален];
- `oldset` – указатель на буфер для старой маски [опционален].

Обработчик сигнала

Обработчик сигнала – функция, вызываемая при получении сигнала.

- обработчик может быть вызван в любой момент времени, в т.ч. в середине функции `malloc`, в момент обработки исключения или даже в момент выполнения другого обработчика, *если не были приняты доп. меры*.
- обработчики сигнала сохраняются при `fork()`, но сбрасываются при `execve()`.

Вызов sigaction (пример 2)

Для изменения диспозиции сигнала и установки обработчика используется вызов `sigaction()`.

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

Параметры:

- `signum` – номер сигнала;
- `act` – указатель на структуру, определяющую новую диспозицию;
- `oldact` – указатель на буфер для старой диспозиции [опционален].

Структура sigaction

```
struct sigaction {  
    void (*sa_handler)(int); //обработчик 1 вида  
    void (*sa_sigaction)(int, siginfo_t*, void*); // 2 вида  
    sigset_t sa_mask; //маска сигналов для обработчика  
    int sa_flags; //флаги  
    /*...*/  
};
```

Функция-обработчик передается в поле `sa_handler` или `sa_sigaction`.

Для игнорирования сигнала в `sa_handler` следует передать константу `SIG_IGN`.

Для сброса диспозиции в `sa_handler` следует передать константу `SIG_DFL`.

Обработчик из `sa_sigaction` берется, если в `sa_flags` указан флаг `SA_SIGINFO`.

Примечание: данные поля могут быть реализованы как макросы, что может создать неудобства в IDE.

Структура sigaction

```
struct sigaction {  
    void (*sa_handler)(int); //обработчик 1 вида  
    void (*sa_sigaction)(int, siginfo_t*, void*); // 2 вида  
    sigset_t sa_mask; //маска сигналов для обработчика  
    int sa_flags; //флаги  
    /*...*/  
};
```

В поле `sa_mask` передается маска сигналов, которая *добавляется* к общей маске сигналов на время обработки сигналов. По умолчанию, к этой маске добавляется обрабатываемый сигнал.

В поле `sa_flags` содержит набор флагов (`SA_SIGINFO` - использовать расширенный обработчик, `SA_NOCLDWAIT` - предотвратить создание зомби, `SA_NODEFER` – не блокировать сигнал на время обработки).

Правила написания обработчиков (пример 3)

«Безопасными» действиями в пределах обработчика сигнала являются:

- Изменение локальных переменных;
- Вызов «безопасных» (*man 7 signal-safety*) или реентерабельных функций;
- Чтение и запись глобальной переменной типа `volatile sig_atomic_t`.

Остальные действия требуют дополнительных мер для обеспечения корректности результата обработчика.

Функция является **реентерабельной** (reentrant), если последовательность <прерывание работы функции – выполнение этой же функции – возобновление работы функции> не влияет на корректность результата.

Любая чистая функция (функция, результат которой зависит только от значений ее аргументов) является реентерабельной.

Примером потокобезопасной нреентерабельной функции является `malloc()`.

Сигнал SIGCHLD. Предотвращение появления зомби

Сигнал SIGCHLD генерируется ядром ОС при изменении состояния дочернего процесса (завершен, остановлен, продолжен).

Для предотвращения появления зомби можно:

1. Установить диспозицию сигнала SIGCHLD в SIG_IGN.
2. Установить флаг SA_NOCLDWAIT в `sigaction.sa_flags` при установке обработчика SIGCHLD.

В любом случае, ОС не станет сохранять коды завершения, поскольку процесс явно указал, что его не интересуют изменения состояния его «детей».

Вызовы `wait()` и `waitpid()` будут блокировать вызывающий поток до тех пор, пока не закончат выполнение все дочерние процессы, после чего завершатся с ошибкой ECHILD.

Синхронизация по сигналу (пример 4)

Приостановить выполнение процесса до получения и обработки незаблокированного сигнала можно вызовом `sigsuspend()`.

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

Параметр:

`mask` – маска сигналов, которая применяется на время ожидания.

Вызов всегда возвращает -1, т.к. формально системный вызов прерывается сигналом. Если `errno==EINTR`, то вызов сработал «успешно».

Примечание: `sigsuspend()` не дает гарантии, что будет получен ровно 1 сигнал: если во время текущей обработки сигнала будет доставлен еще один незаблокированный сигнал, то его обработчик также выполнится до возврата из `sigsuspend()`.

Синхронизация по сигналу

Для получения заблокированного сигнала без вызова обработчика используется вызов `sigwaitinfo()`:

```
int sigwaitinfo(const sigset_t* set, siginfo_t* info);  
int sigtimedwait(const sigset_t* set, siginfo_t* info,  
                 const struct timespec* timeout);
```

Параметры:

`set` – набор ожидаемых сигналов;

`info` – буфер для дополнительной информации о сигнале [опционален].

`timeout` – время ожидания.

Вызов возвращает номер полученного сигнала.

Ожидаемые сигналы должны быть заблокированы заранее.

Сигналы реального времени

Сигналы реального времени являются одним из расширений стандарта POSIX.

- Вместе с сигналом реального времени можно передать маленькую порцию данных: целое число или указатель. Данная особенность превращает сигналы реального времени в средство межпроцессного взаимодействия.
- Если во время блокировки сигнала реального времени данный сигнал был послан N раз, то после разблокировки будут доставлены все N сигналов с сохранением порядка.

Номера сигналов реального времени находятся в интервале [SIGRTMIN, SIGRTMAX].

Примечание: сигналы реального времени отсутствуют в MacOS, т.к. являются необязательным расширением POSIX

Вызов sigqueue

Послать сигнал реальному времени можно функцией `sigqueue()`.

```
union sigval {  
    int    sival_int;  
    void*  sival_ptr;  
};
```

```
int sigqueue(pid_t pid, int sig, const union sigval value);
```

Параметры:

<code>pid</code>	– PID целевого процесса;
<code>sig</code>	– номер сигнала;
<code>value</code>	– значение, передаваемое вместе с сигналом

Обработка сигналов реального времени (пример 5)

Обработку сигналов реального времени можно производить обычным обработчиком.

Если необходимо получить значение, передаваемое сигналом, придется использовать обработчик второго типа.

```
void handler(int sig, siginfo_t* info, void*ctx){ ... }

sigaction action {};  
action.sa_sigaction = handler;  
action.sa_flags = SA_SIGINFO;  
sigaction(SIGRTMAX, &action, NULL);
```

Структура siginfo_t

```
struct siginfo_t {  
    sigval_t si_value;    /* Передаваемое значение */  
    int      si_signo;    /* Номер сигнала */  
    pid_t    si_pid;      /* PID отправителя*/  
    uid_t    si_uid;      /* UID (User ID) отправителя */  
    int      si_code;     /* Код события (причина возникновения сигнала) */  
    int      si_addr;     /* Адрес, вызвавший аппаратное исключение */  
    int      si_status;    /* Код выхода или номер сигнала (для SIGCHLD) */  
    /*...*/  
};
```

Примечание: данные поля могут быть реализованы как макросы, что может создать неудобства в IDE.

Прерывание системных вызовов сигналами (пример 6)


Если сигнал приходит во время длительного системного вызова, то системный вызов может быть прерван. В этом случае системный вызов завершится с ошибкой **EINTR**.

Некоторые системные вызовы могут быть автоматически продолжены после обработки сигнала, если при установке обработчика передать флаг **SA_RESTART** (см. *man 7 signal*).

```
int fd = open("pipe", O_RDONLY);
```

```
struct sigaction s{};  
s.sa_handler = handler;
```

```
struct sigaction s{};  
s.sa_handler = handler;  
s.sa_flags = SA_RESTART;
```

```
sigaction(SIGUSR1, &s, NULL);  
char*buffer = malloc(1024);  
 auto rsize = read(fd, buffer, 1024);  
printf("%ld", rsize);
```

Что выведет: -1 или число ≤ 1024

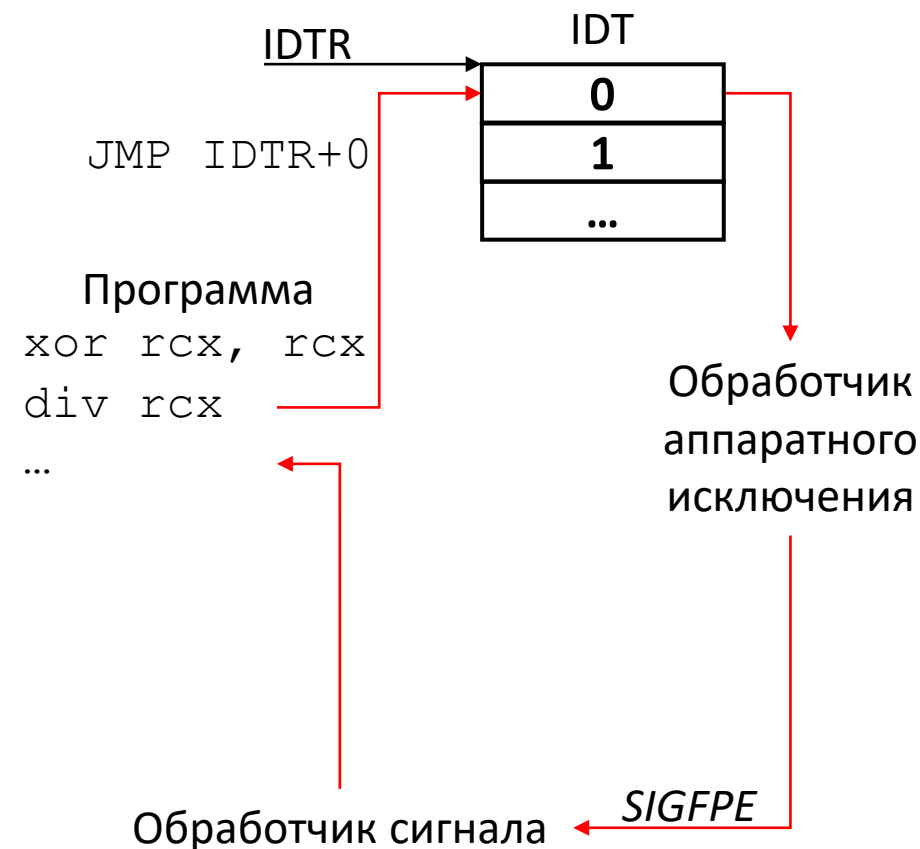
Что выведет: 1024

Сигналы и аппаратные исключения

Аппаратные исключения, возникающие при работе программы, превращаются в сигналы:

- SIGFPE – математические ошибки;
- SIGSEGV, SIGBUS – ошибки доступа к памяти;
- SIGTRAP – точки останова;
- SIGILL – невалидная инструкция.

Эти сигналы потенциально могут быть обработаны – например, стандартные библиотеки высокоуровневых языков программирования могут перехватывать SIGFPE и превращать его в обычное исключение



Сигналы и аппаратные исключения (пример 7)

```
void handler(int sig, siginfo_t* info, void* ctx)
```

```
struct siginfo_t {  
    /*...*/  
    int    si_code; /* Код события */  
    int    si_addr; /* Адрес исключения*/  
};
```

```
struct ucontext_t{  
    /*...*/  
    mcontext_t uc_mcontext; /* Состояние ЦП при  
                             получении сигнала*/  
};
```

Если сигнал вызван аппаратным исключением, то информация об исключении записывается в поля `si_code` и `si_addr` структуры `siginfo_t`, указатель на которую передается 2-ым аргументом.

Третий аргумент является указателем на структуру `ucontext_t`, описывающую состояние программы в момент получения сигнала. В частности, поле `uc_mcontext` хранит состояние ЦП – значения его регистров, в т.ч. программного счетчика (RIP/EIP). Конкретный состав полей структуры `mcontext_t` зависит от архитектуры ЦП и ОС.

Вышеуказанные значения анализируются обработчиком, устанавливаемым средой выполнения, при создании объекта исключения.

Функция abort (пример 8)

Функция `abort()` используется для аварийного завершения процесса.

```
void abort();
```

За кадром функция посылает текущему процессу сигнал SIGABRT.

Если для сигнала установлен обработчик, он выполняется.

Если процесс «переживает» сигнал, то диспозиция сигнала сбрасывается, и сигнал посылается еще раз – процесс уничтожается с созданием дампа памяти.

Вызов alarm (пример 9)

Вызов `alarm()` позволяет запланировать отправку сигнала SIGALRM через определенный промежуток времени.

```
unsigned int alarm(unsigned int seconds);
```

Параметры:

`seconds` – время, через которое необходимо отправить сигнал.

Вызов возвращает количество секунд, которое оставалось до следующего «звонка», если таковой был установлен. При вызове `alarm()` предыдущий «звонок» отменяется.

Для создания периодических действий можно установить обработчик SIGALRM, в конце которого вновь вызывается `alarm()`.

Вызов `alarm()` можно использовать в комбинации с вызовом `sigsuspend()` для организации ожидания, ограниченного по времени (для `sigwaitinfo()` проще использовать параметр `timeout`).