1 Сигналы

Сигнал можно рассматривать как программное прерывание нормальной работы процесса. Ядро использует сигналы чтобы сообщить процессу об исключительных ситуациях, которые могут возникнуть в его работе. Некоторые сигналы сообщают об ошибках, таких как недопустимое обращение к памяти, другие сигналы сообщают об асинхронных событиях, таких как потеря связи с терминалом.

В операционной системе определено большое количество типов сигналов, каждый для своего типа события. Некоторые события делают нежелательным или невозможным продолжение нормальной работы процесса, такие сигналы по умолчанию завершают работу процесса. Другие сигналы, сообщающие о «безобидных» событиях, по умолчанию игнорируются.

Процесс может послать сигнал другому процессу. Это позволяет, например, родительскому процессу завершить выполнение сыновнего процесса. В другой ситуации два процесса могут таким образом синхронизовать своё выполнение.

Обработка сигналов — это область, где работа операционных систем-наследниц System V (сейчас наиболее распространены SCO Unix и Solaris) наиболее сильно отличается от работы систем семейства BSD. В дальнейшем изложении все системы первого типа будут обобщённо названы System V, а системы второго типа — BSD. Хотя при разработке Linux за образец была взята System V, обработка сигналов по умолчанию ведётся как в BSD. Все примеры проверялись на Linux, то есть в них используется семантика сигналов BSD.

1.1 Генерация сигналов

События, которые генерируют сигналы, делятся на три основные группы: ошибки, внешние события и явные запросы.

Ошибка означает, что программа сделала что-то недопустимое и не может продолжить своё выполнение. Не все типы ошибок генерируют сигналы (на самом деле, большинство ошибок не генерирует). Например, попытка открытия несуществующего файла — ошибка, но она не генерирует сигнал, вместо этого системный вызов `open` возвращает `-1`. Вообще, ошибки, которые связаны с библиотечными функциями и системными вызовами, сообщаются программе с помощью специальных возвращаемых значений. Ошибки, которые могут встретиться в любом месте программы, а не только в библиотечных функциях и системных вызовах генерируют сигналы. К таким ошибкам относятся деление на 0 и неверное обращение к памяти.

Внешнее событие обычно относится к операциям ввода/вывода или другим процессам. Например, получение данных при асинхронных операциях, срабатывание таймера, завершение сыновнего процесса.

Явный запрос предполагает использование функций, таких как `kill`, задача которых — сгенерировать сигнал.

Сигналы могут генерироваться *синхронно* или *асинхронно*. Синхронный сигнал относится к некоторому действию в программе и доставляется (если только не заблокирован) во время этого действия. Большинство ошибок генерируют синхронные сигналы. Посылка процессом сигнала самому себе также синхронна.

Асинхронные сигналы генерируются событиями, неподконтрольными процессу, который их получает. Такие сигналы могут приходить в процесс в непредсказуемые моменты времени. Посылка сигнала одним процессом другому процессу также асинхронна.

Каждый тип сигнала как правило синхронен или асинхронен. Например, сигналы, соответствующие ошибкам выполнения программы, синхронны. Но любой сигнал может быть синхронным или асинхронным, когда посылается явно.

Системный вызов `kill` позволяет послать сигнал процессу или группе процессов.

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

Если `pid` больше нуля, заданный сигнал `sig` посылается процессу с заданным идентификатором процесса.

Если `pid` равен 0, сигнал `sig` посылается всем процессам в группе процессов, к которой относится данный процесс.

Если `pid` меньше -1, сигнал `sig` посылается всем процессам в группе процессов с идентификатором `-pid`.

Наконец, если `pid` равен -1, сигнал `sig` посылается всем процессам с тем же эффективным идентификатором пользователя, что у текущего процесса.

Если номер сигнала равен 0, сигнал не посылается, но все возможные ошибки проверяются. Таким образом можно проверить, например, существование процесса с заданным `pid`.

Процесс может послать сигнал самому себе с помощью вызова вида `kill(getpid(), sig)`. Если этот сигнал не блокируется процессом, `kill` доставит хотя бы один сигнал из ожидающих доставки текущему процессу (это может быть другой сигнал, ожидающий доставки, вместо `sig`) перед возвратом из системного вызова `kill`.

В случае успеха системный вызов возвращает 0. Если сигнал посылается группе процессов, вызов `kill` заканчивается успешно, если сигнал был послан хотя бы одному процессу в группе. Нет способа узнать, какой из процессов получил сигнал, или получили ли его они все. В случае неудачи системный вызов возвращает значение `-1`, а переменная `errno` устанавливается в код ошибки, который может быть одним из следующих:

`EINVAL` Аргумент `sig` является недопустимым или неподдерживаемым номером сигналом.

`EPERM` Недостаточно прав послать сигнал процессу или группе процессов.

`ESCRH` Аргумент `pid` не является правильным номером процесса или группы процессов. Эта ошибка возвращается и в случае, когда заданному номеру соответствует процесс-зомби.

Действие функции `raise`, определённой следующим образом:

```
int raise(int sig);
```

эквивалентно вызову `kill(getpid(), sig)`.

1.2 Получение сигналов

После генерации сигнала он становится *ожидающим доставки*. Обычно он таковым остаётся в течение непродолжительного времени, после чего доставляется процессу-адресату. Однако если данный тип сигнала в текущий момент времени заблокирован процессом, сигнал может оставаться ожидающим доставки неограниченно долгое время до тех пор, пока сигналы этого типа не будут разблокированы. После разблокирования сигнал будет доставлен немедленно.

Доставка сигнала означает выполнение определённого действия. Для некоторых сигналов (`SIGKILL` и `SIGSTOP`) это действие фиксировано, но для большинства сигналов у программы есть выбор: игнорировать сигнал, обрабатывать сигнал или выполнять действие по

умолчанию для данного сигнала. Программа может задавать реакцию на сигнал с помощью вызовов `signal` или `sigaction`. Во время работы обработчика сигнала этот тип сигналов заблокирован.

Если для некоторого типа сигнала установлено игнорирование сигнала, любой сигнал такого типа будет сброшен сразу же после генерации, даже если этот тип сигналов в тот момент был заблокирован. Такой сигнал никогда не будет доставлен, даже если программа затем установит обработчик сигнала и разблокирует его.

Если процесс получает сигнал, который ни обрабатывается, ни игнорируется, выполняется действие по умолчанию. Каждый тип сигналов имеет своё действие по умолчанию. Для большинства сигналов действием по умолчанию является завершение процесса. Для некоторых сигналов, обозначающих «безобидные» события, действие по умолчанию состоит в том, чтобы ничего не делать. Обратите внимание, что такое действие по умолчанию не является игнорированием сигнала.

Когда сигнал приводит к завершению работы процесса, родитель процесса может определить причину завершения, проверив статус завершения процесса, возвращаемый функциями семейства `wait`. Сигналы, по умолчанию завершающие процесс, вызывают функцию ядра, аналогичную `_exit`, то есть не вызываются обработчики завершения, зарегистрированные по `atexit`, не записываются буферы дескрипторов потока `FILE`.

Сигналы, обозначающие ошибки выполнения процесса, имеют специальное свойство: когда процесс завершается по одному из таких сигналов, ядро записывает на диск дамп памяти (core dump), который хранит состояние процесса в момент остановки. Этот дамп можно просматривать с помощью отладчика, чтобы определить причину ошибки.

Дамп памяти имеет по умолчанию имя `core` и создаётся в текущем каталоге. Ядро не записывает дамп памяти в случае, когда выполняется хотя бы одно из следующих условий.

- Эффективный идентификатор пользователя или группы пользователей процесса не совпадает с реальным идентификатором пользователя или группы пользователей.
- У процесса отсутствуют права записи в текущий каталог процесса, если файл `core` не существует.
- У процесса отсутствуют права записи в файл `core` в текущем каталоге, если такой файл существует.
- Пользователь запретил создание файла дампа памяти с помощью команды `ulimit -c 0`.

Если сигнал, обозначающий ошибку выполнения процесса, посылается явным запросом и завершает процесс, ядро точно так же сохраняет дамп памяти, как будто бы сигнал был вызван непосредственно ошибкой.

1.3 Стандартные сигналы

В этом разделе перечислены имена стандартных типов сигналов с описанием, какое событие они обозначают. Каждое имя сигнала — это макрос, соответствующий положительному числу — номеру сигнала. Программа не должна делать никаких предположений о номерах сигналов и всегда ссылаться на сигналы, используя символические константы. Номера сигналов могут меняться от системы к системе, а имена сигналов стандартизованы.

Имена сигналов определены в заголовочном файле <signal.h>. Макрос NSIG даёт общее количество сигналов, определённых в системе. Поскольку сигналы нумеруются последовательно, его значение на 1 больше максимального номера сигнала.

Чтобы напечатать строку, описывающую сигнал (например, для SIGSEGV — "Segmentation fault"), можно использовать функцию strsignal, описанную следующим образом:

```
char *strsignal(int signum);
```

Здесь signum — номер сигнала.

Далее приведены таблицы сигналов, разбитых по группам. В столбце «реакция» определена реакция на данный сигнал по умолчанию. Реакция может описываться комбинацией букв, каждая из которых означает следующее:

- C Записать дамп памяти (core dump).
- T Завершить процесс.
- B Сигнал не может быть обработан, заблокирован или проигнорирован.
- S Остановить процесс.
- N Ничего не делать.
- R Продолжить процесс после остановки.

Имя	Реакция	Описание
		Программные ошибки
SIGFPE	CT	Фатальная арифметическая ошибка. Для целочисленных операций это может быть деление на 0, умножение минимального целого числа на -1. Для вещественных чисел существует много возможных ошибок. Например, переполнение, антипереполнение, деление на 0 и т. д.
SIGILL	CT	Недопустимая инструкция. Процесс пытался выполнить код, не соответствующий никакой инструкции процессора, или привилегированную инструкцию процессора. Обычно это означает, что процесс пытался выполнить какие-то данные. Это может произойти из-за переполнения массива, размещённого в стеке, или из-за неинициализированного указателя на функцию.
SIGSEGV	CT	Попытка чтения или записи по адресу, на который не отображается память, попытка записи в память, открытую только для чтения, попытка выполнения невыполняемой памяти. Чтение или запись по адресу 0 (NULL) может вызывать эту ошибку.
SIGBUS	CT	Неверное обращение к памяти, например, обращение по невыровненному адресу. Обращение по адресу 0 также может вызывать этот сигнал. Типы сигналов SIGSEGV и SIGBUS близки по смыслу, и точное деление между ними зависит от операционной системы и типа процессора.
SIGABRT	CT	Ошибка, выявленная самой программой, которая в этом случае вызвала функцию abort().
SIGTRAP	CT	Сигнал трассировки. Генерируется специальной инструкцией трассировки процессора и, возможно, другими инструкциями. Этот сигнал используется отладчиками.
		Завершение процесса

Продолжение на следующей странице

Имя	Реакция	Описание
SIGTERM	T	Завершить выполнение процесса. В отличие от SIGKILL этот сигнал может быть заблокирован, проигнорирован или обработан процессом. Посылка этого сигнала — стандартный способ «вежливо» попросить программу завершиться. Команда kill командного интерпретатора посылает по умолчанию этот сигнал.
SIGINT	T	Завершение процесса. Сигнал обычно посылается процессу, когда пользователь нажимает клавиши Ctrl-C (символ INTR).
SIGQUIT	CT	Аварийное завершение процесса. Сигнал обычно посылается процессу при нажатии клавиш Ctrl-\ на клавиатуре (символ QUIT). По умолчанию этот сигнал вызывает завершение процесса с дампом памяти. Его можно рассматривать как ошибку выполнения программы, распознанную пользователем.
SIGKILL	BT	Немедленное завершение процесса. Этот сигнал не может быть обработан, заблокирован или проигнорирован и, следовательно, всегда фатален. Сигнал SIGKILL нужно рассматривать как последнее средство завершить работу процесса, после того, как процесс не завершился по SIGTERM или SIGINT. Если сигнал SIGKILL не завершил процесс, это — ошибка ядра операционной системы.
SIGHUP	T	Сигнал посылается процессу, когда отсоединяется управляющий терминал данного процесса (например, разорванное сетевое соединение). Кроме того, сигнал посылается всем процессам в сессии, когда завершается лидер сессии. Завершение процесса-лидера сессии означает отсоединение всех процессов в сессии от управляющего терминала.
Срабатывание таймеров		
SIGALRM	T	Сработал таймер, измеряющий реальные (календарные) интервалы времени.
SIGVTALRM	T	Сработал таймер, измеряющий виртуальные интервалы времени (то есть время работы процесса в режиме пользователя).
SIGPROF	T	Сработал таймер, измеряющий время процессора, потраченное на данный процесс и в режиме пользователя, и в режиме ядра. Такой таймер используется для профилирования программы, отсюда и название.
Работа с процессами и управление заданиями (job control)		

Продолжение на следующей странице

Имя	Реакция	Описание
SIGCHLD	N	Этот сигнал посылается родительскому процессу, когда один из его сыновних процессов завершается или останавливается. Обработчик по умолчанию этого сигнала ничего не делает. Если пользовательский обработчик сигнала устанавливается в то время, когда есть сыновние процессы-зомби, будет ли этот обработчик вызван для процессов-зомби, зависит от конкретной операционной системы (Linux — нет). Для систем BSD, если сигнал SIGCHLD явно установлен как игнорируемый процессом, система не создаёт процессов-зомби, а сразу уничтожает их по завершению. Однако стандарт POSIX запрещает процессам явно игнорировать сигнал SIGCHLD. Программы, написанные для BSD, в этом случае не будут работать в других операционных системах. Новейший стандарт Unix98 снова разрешает игнорирование сигнала SIGCHLD в стиле BSD.
SIGSTOP	BS	Сигнал останавливает выполнение процесса. Он не может быть обработан, проигнорирован или заблокирован.
SIGCONT	R	Сигнал вызывает продолжение работы процесса, если он был остановлен. Этот сигнал не может быть заблокирован. Для него можно определить обработчик, но перед вызовом обработчика процесс всё равно будет продолжен. Для большинства программ не имеет смысла обрабатывать SIGCONT. Такие программы просто продолжают свою работу, не заметив, что они были приостановлены.
SIGTSTP	S	Интерактивный сигнал остановки выполнения процесса. В отличие от SIGSTOP этот сигнал может обрабатываться, блокироваться или игнорироваться. Сигнал генерируется, когда пользователь нажимает на клавиатуре Ctrl-Z (символ SUSP). Процесс должен обрабатывать этот сигнал, если требуется оставлять системные данные или файлы в целостном состоянии при остановке. Например, программа, которая отключает канонический режим ввода символов, может его снова включить при остановке.
SIGTTIN	S	Процесс не может считывать данные с терминала, когда он запущен как фоновое задание. Когда какой-либо процесс в фоновом задании пытается это сделать, все процессы в задании получают сигнал SIGTTIN. По умолчанию этот сигнал вызывает остановку выполнения процесса.
SIGTTOU	S	Аналогично SIGTTIN, но этот сигнал генерируется, когда процесс из фонового задания пытается записать на терминал или установить его режимы работы. По умолчанию запись на терминал для фоновых процессов разрешена, чтобы запретить её должен быть установлен режим TOSTOP терминала.

Ошибки операций ввода/вывода

Продолжение на следующей странице

Имя	Реакция	Описание
SIGPIPE	T	Попытка записи в канал (анонимный или именованный), у которого закрыт выходной конец. Если этот сигнал блокируется, игнорируется или обрабатывается, операция, вызвавшая ошибку, завершается с кодом ошибки EPIPE.
		Прочие сигналы
SIGUSR1	T	
SIGUSR2	T	Сигналы SIGUSR1 и SIGUSR2 предназначены для использования в прикладных программах произвольным образом.

Когда процесс остановлен, он не может получать сигналы, кроме SIGKILL и SIGCONT. Все посланные процессу сигналы будут сделаны ожидающими доставки. Как только процесс продолжит работу, сигналы будут ему доставлены. Когда процесс получает сигнал SIGCONT, все сигналы остановки, ожидающие доставки, будут сброшены. Аналогично, когда процесс получает сигнал остановки, все сигналы SIGCONT, ожидающие доставки, будут сброшены.

1.4 Работа с множествами сигналов

Аргументами многих функций, работающих с сигналами, могут быть множества сигналов. Например, sigprocmask позволяет изменить множество блокируемых сигналов. Для удобства работы с множествами сигналов стандартная библиотека предоставляет типы и функции. Они определены в заголовочном файле <signal.h>.

Тип sigset_t должен использоваться для хранения множества сигналов. Программа не должна предполагать, что этот тип эквивалентен некоторому целому типу, как изначально было в системах BSD.

```
int sigemptyset(sigset_t *pset);
int sigfillset(sigset_t *pset);
int sigaddset(sigset_t *pset, int signum);
int sigdelset(sigset_t *pset, int signum);
int sigismember(const sigset_t *pset, int signum);
```

Функция sigemptyset очищает множество сигналов, на которое указывает pset. Пустое множество не содержит ни одного сигнала.

Функция sigfillset полностью заполняет множество сигналов, на которое указывает pset. В получившееся множество включены все сигналы.

Функция sigaddset добавляет сигнал signum в множество сигналов, на которое указывает pset.

Функция sigdelset удаляет из множества сигналов, на которое указывает pset, сигнал signum.

Функция sigismember проверяет, присутствует ли в множестве сигналов, на которое указывает pset, сигнал с номером signum.

1.5 Установка обработчика сигнала

1.5.1 Функция signal

Простейшая функция, с помощью которой можно изменить обработку сигнала, это функция signal, описанная следующим образом:

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

Если ввести специальный тип для указателя на функцию-обработчик сигнала, определение функции упростится:

```
typedef void (*sighandler_t)(int signum);
sighandler_t signal(int signum, sighandler_t handler);
```

Первый аргумент `signum` задает номер сигнала, обработку которого нужно изменить. Вместо номера сигнала предпочтительнее использовать символическое имя сигнала, как определено выше.

Второй аргумент `handler` определяет, как будет обрабатываться сигнал. Он может принимать следующие значения:

`SIG_DFL` устанавливает обработку сигнала на стандартную обработку по умолчанию (см. таблицы выше).

`SIG_IGN` задает, что сигнал должен игнорироваться. Программа не должна игнорировать сигналы, которые обозначают серьёзные программные ошибки или используются для завершения процесса. Если процесс игнорирует сигнал `SIGSEGV` и другие аналогичные сигналы, его поведение после ошибки неопределено (например, он может зациклиться на месте ошибки). Игнорировать запросы пользователя, такие как `SIGINT` и пр. — недружественно по отношению к пользователю.

Третья возможность — это задать функцию обработки сигнала. Эта функция будет вызвана, когда процесс получит сигнал.

Если обработка сигнала устанавливается в `SIG_IGN`, или когда обработка сигнала устанавливается в `SIG_DFL`, а обработка по умолчанию игнорирует сигнал, все сигналы этого типа, ожидающие доставки, будут сброшены, даже если они заблокированы. Такие сигналы никогда не будут доставлены, даже если впоследствии обработчик сигнала будет переустановлен, и сигнал будет разблокирован.

Функция `signal` возвращает предыдущий обработчик сигнала. Это значение может использоваться для того, чтобы восстановить старый обработчик, если это необходимо.

Функция `signal` присутствует в стандарте ANSI C, тем не менее её использование не рекомендуется. Исторически существовало два подхода к обработке сигналов: подход, реализованный в System V, и подход BSD, различия между которыми приведены в таблице ниже. Поэтому рекомендуется использовать более универсальную функцию `sigaction`, описанную ниже.

Свойство	System V	BSD
Блокировка сигнала	Текущий обрабатываемый сигнал не блокируется на время выполнения обработчика.	Текущий обрабатываемый сигнал блокируется на время выполнения обработчика.
Сброс обработчика	Обработчик переустанавливается на обработчик по умолчанию.	Обработчик не переустанавливается на обработчик по умолчанию.
Системные вызовы	Прерываются с кодом ошибки `EINTR`.	Перезапускаются.

1.5.2 Пример программы

В следующем примере программа завершит работу, когда три раза будет нажата комбинация Ctrl-C. Предполагается, что функция `signal` поддерживает семантику BSD, то есть

программа будет работать без изменений на операционных системах семейства BSD и на Linux.

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

int cnt = 0;

void sigint_handler(int signo)
{
  printf("Ctrl-C pressed\n");
  if (++cnt == 3) {
    signal(SIGINT, SIG_DFL);
    raise(SIGINT);
  }
}

int main(void)
{
  signal(SIGINT, sigint_handler);
  while (1) {
    printf("Some string to print\n");
  }
  return 0;
}
```

Обратите внимание, как происходит выход из программы в случае, когда три раза подряд нажата комбинация Ctrl-C. Обработчик сигнала `SIGINT` устанавливается в стандартное значение по умолчанию (то есть выход из программы), затем сигнал `SIGINT` посылается самому себе с помощью функции `raise`. Это гарантирует, что родительскому процессу будет сообщена истинная причина завершения программы: получение сигнала `SIGINT`. Если бы программа завершала свою работу по вызову `exit(0)`, родительский процесс получил бы информацию, что наш процесс завершился нормально с кодом завершения `0`. Какой из двух вариантов завершения программы: посылка сигнала самой себе еще раз или выход по `exit` — предпочтительнее, зависит от конкретной ситуации.

Приведённая выше программа имеет серьёзный дефект, связанный с тем, что сигнал может поступить в программу и начать обрабатываться в любой момент времени. Если сигнал поступит, например, в середине работы функции `printf`, выполнение функции `printf` будет приостановлено, и начнется выполнение обработчика сигнала, который вновь вызовет `printf`. Получается, что функция будет вызвана вновь из середины самой себя. Далеко не все функции стандартной библиотеки будут корректно работать в этой ситуации (на самом деле, большинство не будет). Функция, которая может безопасно вызываться из обработчика сигнала, называется *асинхронно-безопасной*.

Чтобы устранить этот дефект можно поступить двумя способами: во-первых, на время выполнения функции `printf` можно заблокировать сигнал `SIGINT`. Блокирование сигналов будет рассмотрено ниже. Во-вторых, можно переписать программу так, чтобы обработчик сигнала просто устанавливал некоторую глобальную переменную, которую будет проверять основная программа. Основная программа будет печатать сообщение о нажатой клавише.

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

```c
#include <signal.h>

int cnt = 0;
volatile int flag = 0;

void sigint_handler(int signo)
{
  flag = 1;
  if (++cnt == 3) {
    signal(SIGINT, SIG_DFL);
    raise(SIGINT);
  }
}

int main(void)
{
  signal(SIGINT, sigint_handler);
  while (1) {
    if (flag) {
      printf("Ctrl-C␣pressed\n");
      flag = 0;
    } else {
      printf("Some␣string␣to␣print\n");
    }
  }
  return 0;
}
```

Ключевое слово **volatile** в определении переменной `flag` говорит компилятору, что значение переменной может быть изменено асинхронно, и поэтому компилятор не должен пытаться оптимизировать обращения к этой переменной, например, сохраняя её на регистрах процессора.

Новый вариант программы будет работать, как ожидается, при выводе в файл или на консоль, но всё равно не будет работать, когда вывод происходит в окно эмулятора терминала `xterm`. В этом случае проблема уже не в самой программе, а в том, как взаимодействуют X-сервер, эмулятор терминала `xterm` и программа. Сигнал `SIGINT` посылает программе не ядро операционной системы, а программа `xterm`. Перед тем, как послать сигнал, `xterm` сбрасывает считанные, но еще не выведенные на экран символы выходного потока. По всей видимости, на это свойство `xterm` из программы никак повлиять нельзя. Поэтому, если программа получила сигнал `SIGTERM` в момент интенсивной записи на экран, нельзя гарантировать, что запись будет целостна, то есть не будут пропущены отдельные символы. Единственный способ обойти эту проблему — отключить режим канонического ввода с терминала.

1.5.3 Функция `sigaction`

Системный вызов `sigaction` позволяет установить обработчик сигнала. Функция описана следующим образом:

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

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

Структура sigaction описана следующим образом:

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

Функция sigaction устанавливает обработчик для сигнала signum, который задается в аргументе act, если этот аргумент не равен NULL. Если аргумент oldact не равен NULL, информация о предыдущем обработчике возвращается в структуру, на которую указывает oldact.

Поле sa_restorer сохранено для совместимости со старым программным обеспечением и не должно использоваться. Поле sa_sigaction позволяет задать обработчик сигнала, которому будет передаваться больше информации, чем обычно. Описание работы расширенного обработчика сигналов выходит за рамки данного документа.

Поле sa_handler задаёт обработчик сигнала. Он может быть равен SIG_IGN или SIG_DFL, что означает игнорирование сигнала или установку обработчика по умолчанию для сигнала.

Поле sa_mask задаёт множество сигналов, которые будут заблокированы на время работы обработчика сигнала.

Поле sa_flags позволяет определить режим, в котором будет обрабатываться сигнал. Значение этого поля получается объединением значений флагов, перечисленных ниже.

SA_NOCLDSTOP		Может задаваться только для сигнала SIGCHLD. В этом случае процесс не будет получать сигнал SIGCHLD, когда какой-либо из его сыновних процессов будет остановлен, то есть получит сигнал SIGSTOP, SIGTSTP, SIGTTIN или SIGTTOU.
SA_ONESHOT	или	Устанавливает обработку сигнала по умолчанию перед тем, как
SA_RESETHAND		будет вызван обработчик сигнала (семантика System V функции signal).
SA_RESTART		Устанавливает перезапуск системных вызовов после возврата из обработчика сигнала (семантика BSD функции signal).
SA_NOMASK	или	Не блокирует получение сигнала в обработчике этого сигнала
SA_NODEFER		(семантика System V функции signal).
SA_SIGINFO		Если установлен этот флаг, функция обработки сигнала будет получать 3 аргумента вместо одного. В этом случае адрес функции-обработчика сигнала должен содержаться в sa_sigaction, а не в sa_handler. Обсуждение расширенной обработки сигналов выходит за пределы данного документа.

Из всех этих флагов стандарт POSIX определяет только SA_NOCLDSTOP. Остальные флаги являются расширениями стандарта. В системах BSD и Linux поддерживаются все флаги.

1.5.4 Пример программы

Следующий пример делает то же самое, что и предыдущий пример, но использует функцию `sigaction`.

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

int cnt = 0;
volatile int flag = 0;

struct sigaction orig_sigint_handler;

void sigint_handler(int signo)
{
  flag = 1;
  if (++cnt == 3) {
    sigaction(SIGINT, &orig_sigint_handler);
    raise(SIGINT);
  }
}

int main(void)
{
  struct sigaction sa;

  sa.sa_handler = sigint_handler;
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = SA_RESTART;
  sigaction(SIGINT, &sa, &orig_sigint_handler);

  while (1) {
    if (flag) {
      printf("Ctrl-C␣pressed\n");
      flag = 0;
    } else {
      printf("Some␣string␣to␣print\n");
    }
  }
  return 0;
}
```

1.6 Сигналы и системные вызовы

Процесс может получить и обработать сигнал в тот момент, когда процесс ждёт или выполняет какую-либо операцию ввода-вывода, например `read` или `open`. Если обработчик сигнала возвращает управление процессу (не завершает его), возникает вопрос, что должна сделать система с идущей операцией ввода/вывода?

Если функция передачи данных, такая как `read` или `write`, прервана сигналом после того, как была считана или записана часть данных, ядро завершит выполнение соответствующего системного вызова и вернёт в качестве результата то количество байт, которое было

обработано к моменту прихода сигнала, обозначая частичный успех операции. Это не касается атомарных операций, таких как чтение из канала или запись в канал, чтение/запись очереди сообщений и пр.

Если сигнал пришёл в момент, когда операция ввода/вывода еще не начала собственно обмен данными, например, когда системный вызов `read` ждёт ввода очередного символа с терминала, поведение системы отличается в зависимости от того, реализует ли она семантику System V, стандартизованную впоследствии POSIX, или семантику BSD.

Стандарт POSIX определяет, что в этом случае системный вызов должен завершиться с ошибкой `EINTR`. Этот подход даёт гибкость, но обычно достаточно неудобен для программиста. Приложение в этом случае должно проверять код ошибки `EINTR` после каждого системного вызова, который может вернуть эту ошибку, и перезапускать системный вызов при необходимости. Если программист забудет это сделать, программа будет работать неправильно.

Семантика BSD предполагает, что прерванный системный вызов будет автоматически перезапущен. В этом случае системный вызов никогда не вернёт ошибку `EINTR`.

Функция `sigaction` позволяет выбрать, какая семантика предпочтительнее для обработчика данного сигнала. Если указан флаг `SA_RESTART`, прерванный системный вызов будет автоматически перезапущен, а если флаг не указан, системный вызов завершится с ошибкой `EINTR`.

1.7 Блокирование сигналов

Временная блокировка сигнала с помощью `sigprocmask` позволяет предотвратить прерывание нормальной работы процесса в критической секции кода. Если сигнал поступит процессу в это время, он будет доставлен позднее, когда процесс его разблокирует.

Временная блокировка может быть полезна, когда и обработчик сигнала, и основная программа работают с некоторой разделяемой структурой данных (в примере выше — это функция `printf`, которая работает с дескриптором потока `stdout`). Если работа с этой структурой не атомарна, обработчик сигнала может быть запущен, когда структура находится в нецелостном состоянии, что может приводить к самым неприятным последствиям. Чтобы предотвратить прерывание программы в момент модификации разделяемой структуры, критическая секция кода должна быть защищена командами блокирования опасного сигнала.

Кроме того, сигнал должен блокироваться, когда программа должна выполнить некоторое действие только тогда, когда сигнал не пришёл. В противном случае программа будет содержать временну́ю ошибку (timing error). Пример такой программы будет разобран ниже в разделе, посвящённом ожиданию прихода сигнала.

Множество сигналов, которые в текущий момент заблокированы процессом, называется маской сигналов процесса. Каждый процесс имеет свою маску сигналов. Когда вызовом `fork()` создаётся новый процесс, он наследует маску сигналов родительского процесса.

Функция `sigprocmask` позволяет изменить маску сигналов процесса.

```
int sigprocmask(int mode,
                const sigset_t *pset,
                sigset_t *poldset);
```

Аргумент `mode` задаёт, какая операция будет выполнена с маской сигналов. Он должен быть равен одному из следующих значений:

SIG_BLOCK	Множество сигналов, на которое указывает pset, добавляется к маске сигналов процесса.
SIG_UNBLOCK	Множество сигналов, на которое указывает pset, удаляется из маски сигналов процесса. Допускается удалить из маски неблокируемый сигнал.
SIG_SETMASK	Маска сигналов копируется из множества сигналов, на которое указывает pset.

Если указатель poldset не равен NULL, то по этому адресу копируется старое значение маски сигналов процесса. Если нужно только узнать текущую маску процесса, но не изменять её, аргумент pset можно задать равным NULL.

Следующий пример блокирует сигнал SIGINT на время выполнения функции printf.

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

int cnt = 0;

void sigint_handler(int signo)
{
  printf("Ctrl-C␣pressed\n");
  if (++cnt == 3) {
    signal(SIGINT, SIG_DFL);
    raise(SIGINT);
  }
}

int main(void)
{
  sigset_t blockset;

  sigemptyset(&blockset);
  sigaddset(&blockset, SIGINT);
  signal(SIGINT, sigint_handler);
  while (1) {
    sigprocmask(SIG_BLOCK, &blockset, 0);
    printf("Some␣string␣to␣print\n");
    sigprocmask(SIG_UNBLOCK, &blockset, 0);
  }
  return 0;
}
```

1.8 Ожидание прихода сигналов

Простейший способ приостановить выполнение процесса до поступления сигнала заключается в использовании функции pause.

```
#include <unistd.h>
int pause(void);
```

Функция pause приостанавливает выполнение процесса до поступления сигнала, который не блокируется и не игнорируется. Если сигнал обрабатывается по умолчанию, он дол-

жен вызывать завершение работы процесса. Если поступление сигнала запускает функцию обработки сигнала, которая возвращает управление в процесс, функция `pause` возвращается с кодом завершения -1 и кодом ошибки `EINTR` даже в случае, когда включена семантика перезапускаемых системных вызовов.

Однако кажущаяся простота этой функции может приводить к серьёзным ошибкам. Рассмотрим программу, которая при поступлении сигнала `SIGINT` печатает сообщение. Возможный вариант этой программы представлен ниже:

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

void sigint_handler(int signo)
{
}

int main(void)
{
  signal(SIGINT, sigint_handler);
  while (1) {
    pause();
    printf("Ctrl-C␣pressed\n");
  }
  return 0;
}
```

Эта программа содержит *временну́ю ошибку* (timing error). Предположим, что сигнал поступит в момент, когда программа выполняет оператор `printf`. В этом случае, когда управление вернётся на вызов `pause`, сигнал будет уже обработан, и функция `pause` «повиснет» на неопределённое время. Сигнал был потерян.

Программа может быть модифицирована так, что обработчик сигнала устанавливает некоторую переменную, которая потом проверяется в основном цикле.

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

volatile int flag;
int cnt;

void sigint_handler(int signo)
{
  flag = 1;
}

int main(void)
{
  signal(SIGINT, sigint_handler);
  while (1) {
    if (!flag) pause();
    flag = 0;
    printf("Ctrl-C␣pressed\n");
```

```
  }
  return 0;
}
```

Модифицированная программа на самом деле содержит ту же самую ошибку, только в более завуалированной форме. Предположим, что сигнал поступил в тот момент, когда значение переменной `flag` уже было проверено, но функция `pause` ещё не была вызвана. В этом случае сигнал будет потерян, и программа опять зависнет на неопределённое время. Что самое неприятное, такая ошибка может проявлять себя крайне редко и практически невоспроизводима.

Корректный способ ждать прихода сигнала реализуется с использованием функции `sigsuspend`.

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

Функция заменяет маску сигналов процесса маской, на которую указывает аргумент `pset`, затем приостанавливает выполнение процесса до поступления сигнала, который либо вызывает завершение процесса, либо обрабатывается процессом.

Если сигнал обрабатывается процессом, и обработчик возвращает управление в процесс, функция `sigsuspend` также возвращается, при этом восстанавливается маска сигналов процесса, которая была на момент вызова функции.

Программа, корректно ожидающая поступления сигнала `SIGINT`, приведена ниже.

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

volatile int flag;
int cnt;

void sigint_handler(int signo)
{
  flag = 1;
}

int main(void)
{
  sigset_t mask, oldmask;

  sigemptyset(&mask);
  sigaddset(&mask, SIGINT);
  signal(SIGINT, sigint_handler);

  while (1) {
    sigprocmask(SIG_BLOCK, &mask, &oldmask);
    while (!flag)
      sigsuspend(&oldmask);
    flag = 0;
    sigprocmask(SIG_UNBLOCK, &mask, NULL);
    printf("Ctrl-C␣pressed\n");
  }
  return 0;
```

```
}
```

Обратите внимание, что значение переменной `flag` сбрасывается в 0 при заблокированном сигнале, так как в противном случае сигнал может поступить в процесс в момент, когда сигнал уже разблокирован, но переменная `flag` ещё не сброшена. Сигнал в этом случае будет потерян.

1.9 Слияние сигналов

Рассмотрим следующий пример:

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

void sigint_handler(int signo)
{
  printf("Ctrl-C␣pressed\n");
}

int main(void)
{
  sigset_t mask;

  sigemptyset(&mask);
  sigaddset(&mask, SIGINT);
  signal(SIGINT, sigint_handler);

  while (1) {
    sigprocmask(SIG_BLOCK, &mask, NULL);
    sleep(2);
    sigprocmask(SIG_UNBLOCK, &mask, NULL);
  }
  return 0;
}
```

Вызов `sleep` здесь заменяет некоторую операцию, требующую значительного времени.

Сколько бы раз в процесс не поступил сигнал `SIGINT` в то время, пока он заблокирован, обработчик сигнала будет вызван после разблокировки не более одного раза. Ядро не поддерживает очередь сигналов, ожидающих доставки, а хранит для каждого типа сигнала единственный бит, означающий наличие недоставленного сигнала соответствующего типа. Эти биты все вместе образуют маску сигналов процесса, ожидающих доставки. Поэтому сигналы не могут использоваться там, где необходимо считать количество их поступлений. В этом случае необходимо использовать другой механизм межпроцессного взаимодействия.

В современных системах введён новый тип сигналов — сигналы реального времени, для которых ядро поддерживает очередь ожидающих доставки сигналов, но традиционные сигналы, рассмотренные в этом документе к ним не относятся.

1.10 Пример программы

Рассмотрим следующую программу. Два процесса обмениваются друг с другом сообщениями в стиле пинг-понг, то есть первый посылает второму число 1, на что второй отвечает первому числом 2, и так далее. Для обмена данными используется единственный канал, а процессы синхронизуются посылкой друг другу сигнала SIGUSR1. Главный процесс породит два подпроцесса, которые и будут обмениваться данными, а он сам будет просто ожидать завершения обоих процессов.

Первая проблема, которая возникает, как передать каждому процессу идентификатор другого процесса. Мы обойдем ее поместив оба процесса в одну группу процессов. Каждый процесс будет посылать сигнал SIGUSR1 всей группе процессов.

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

volatile int flag;
void handler(int signo)
{
  flag = 1;
}

void do_work(int pgid, int sig, int *p)
{
  sigset_t bs, os, b2;
  int      val;

  sigemptyset(&bs);
  sigaddset(&bs, SIGUSR1);
  sigaddset(&bs, SIGUSR2);
  sigemptyset(&b2);
  sigaddset(&b2, SIGINT);
  sigaddset(&b2, SIGTERM);
  sigprocmask(0, NULL, &os);
  while (1) {
    /* ожидаем прихода либо SIGUSR1, либо SIGUSR2 */
    /* так как один из них игнорируется, мы всегда получим нужный */
    sigprocmask(SIG_BLOCK, &bs, NULL);
    while (!flag)
      sigsuspend(&os);
    flag = 0;
    sigprocmask(SIG_UNBLOCK, &bs, NULL);
    read(p[0], &val, sizeof(val));
    /* мы не хотим, чтобы нас прервали на середине печати */
    sigprocmask(SIG_BLOCK, &b2, NULL);
    printf("Process␣%d␣got␣value␣%d\n", getpid(), val);
    fflush(stdout);
    sigprocmask(SIG_UNBLOCK, &b2, NULL);
    val++;
```

```
    write(p[1], &val, sizeof(val));
    kill(-pgid, sig);
  }
}

int main(void)
{
  int pid1, pid2;
  int pgid;
  int p[2];
  int val = 0;

  pgid = getpid();
  signal(SIGUSR1, handler);
  signal(SIGUSR2, SIG_IGN);
  if (pipe(p) < 0) { perror("pipe"); exit(1); }
  if ((pid1 = fork()) < 0) { perror("fork"); exit(1); }
  if (!pid1) {
    setpgid(0, pgid);
    do_work(pgid, SIGUSR2, p);
    _exit(0);
  }
  setpgid(pid1, pgid);
  signal(SIGUSR1, SIG_IGN);
  signal(SIGUSR2, handler);
  if ((pid2 = fork()) < 0) { perror("fork"); exit(1); }
  if (!pid2) {
    setpgid(0, pgid);
    do_work(pgid, SIGUSR1, p);
    _exit(0);
  }
  setpgid(pid2, pgid);
  write(p[1], &val, sizeof(val));
  close(p[0]); close(p[1]);
  signal(SIGUSR2, SIG_IGN);
  kill(-pgid, SIGUSR1);
  sleep(1);
  signal(SIGTERM, SIG_IGN);
  kill(-pgid, SIGTERM);
  wait(0); wait(0);

  return 0;
}
```

Обратите внимание, как основная программа последовательно переустанавливает обработчики сигналов `SIGUSR1` и `SIGUSR2`. Это необходимо делать, чтобы избежать временны́х ошибок. В противном случае главный процесс мог бы послать сигнал `SIGUSR1` первому процессу, который бы еще не успел установить обработчик, и сигнал был бы либо потерян, либо проигнорирован, что в любом случае привело бы к зависанию программы. Вызов `setpgid` делается и в отце, и в каждом из сыновних процессов опять-таки чтобы избежать временны́х ошибок.