Семинары 14-15. Сигналы

Аппаратные прерывания (interrupt), исключения (exception), программные прерывания (trap, software interrupt). Их обработка

Концепция сигналов в UNIX тесно связана с понятиями аппаратного прерывания, исключения и программного прерывания, которые будут подробно рассмотрены на лекциях. Здесь лишь краткое изложение.

После выдачи запроса ввода-вывода у процессора существует два способа узнать о том, что обработка запроса устройством завершена. Первый способ заключается в регулярной проверке процессором ***бита занятости*** в ***регистре состояния*** контроллера соответствующего устройства (polling). Второй способ заключается в использовании прерываний. При втором способе процессор имеет специальный вход, на который устройства ввода-вывода, используя контроллер прерываний или непосредственно, выставляют сигнал запроса прерывания (interrupt request) при завершении операции ввода-вывода. При наличии такого сигнала процессор после выполнения текущей команды не выполняет следующую, а, сохранив состояние ряда регистров и, возможно, загрузив в часть регистров новые значения, переходит к выполнению команд, расположенных по некоторым фиксированным адресам. После окончания обработки прерывания можно восстановить состояние процессора и продолжить его работу с команды, выполнение которой было отложено.

Аналогичный механизм часто используется при обработке исключительных ситуаций (exception), возникающих при выполнении команды процессором (неправильный адрес в команде, защита памяти, деление на ноль и т. д.). В этом случае процессор не завершает выполнение команды, а поступает, как и при прерывании, сохраняя своё состояние до момента начала её выполнения.

Этим же механизмом часто пользуются и для реализации так называемых программных прерываний (software interrupt, trap), применяемых, например, для переключения процессора из режима пользователя в режим ядра внутри системных вызовов. Для выполнения действий, аналогичных действиям по обработке прерывания, процессор в этом случае должен выполнить специальную команду.

Необходимо чётко представлять себе разницу между этими тремя понятиями.

Как правило, обработку аппаратных прерываний от устройств ввода-вывода и программных прерываний производит сама операционная система, не доверяя работу с системными ресурсами процессам пользователя. Обработка же части исключительных ситуаций вполне может быть возложена на пользовательский процесс через механизм сигналов. Этот же механизм может быть задействован и для оповещения пользовательских процессов о некоторых событиях, происходящих в вычислительной системе и связанных с работой других процессов

Понятие сигнала. Способы возникновения сигналов и виды их обработки

С точки зрения пользователя получение процессом сигнала выглядит как возникновение прерывания. Процесс прекращает регулярное исполнение, и управление передаётся механизму обработки сигнала. По окончании обработки сигнала процесс может возобновить регулярное исполнение. Типы сигналов (их принято задавать номерами, как правило, в диапазоне от 1 до 31 включительно или специальными символьными обозначениями) и способы их возникновения в системе жёстко регламентированы.

Процесс может получить сигнал от:

1. hardware (например, от UPS при потере питания во внешней сети, от процессора и блока управления памятью при возникновении исключительных ситуаций);
2. другого процесса, выполнившего системный вызов передачи сигнала;
3. операционной системы (при наступлении некоторых событий);
4. терминала (при нажатии определённой комбинации клавиш);
5. системы управления заданиями (при выполнении команды **kill** – мы рассмотрим её позже).

Передачу сигналов процессу в случаях его генерации источниками 2, 3 и 5, т. е., в конечном счете, каким-либо другим процессом, можно рассматривать как реализацию в UNIX сигнальных средств связи, о которых рассказывалось в лекциях.

Существует три варианта реакции процесса на сигнал:

1. Принудительно проигнорировать сигнал.
2. Произвести обработку по умолчанию: проигнорировать, остановить процесс (перевести в состояние ожидания до получения другого специального сигнала), либо завершить работу с образованием core файла или без него.
3. Выполнить обработку сигнала, специфицированную пользователем.

Изменить реакцию процесса на сигнал можно с помощью специальных системных вызовов, которые мы рассмотрим позже. Реакция на некоторые сигналы не допускает изменения, и они могут быть обработаны только по умолчанию. Так, например, сигнал с номером 9 – **SIGKILL** обрабатывается только по умолчанию и всегда приводит к завершению процесса.

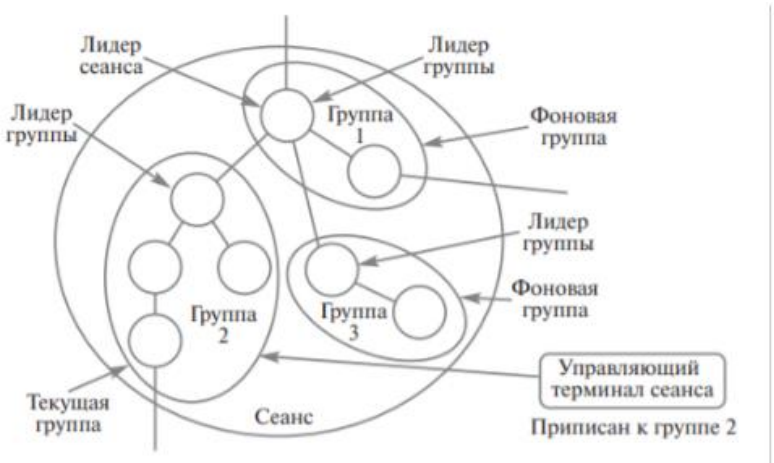
Необходимо отметить, что игнорирование сигналов по умолчанию и принудительное игнорирование сигналов — это принципиально разные реакции. Так, например, при завершении процесса-ребёнка процесс-родитель получает сигнал **SIGCHLD**, который по умолчанию процессом-родителем игнорируется. При этом завершившийся процесс остается в состоянии «завершил исполнение» (т.е. становится зомби-процессом) до тех пор, пока родитель не поинтересуется причинами его завершения или сам не завершит работу. При принудительном игнорировании этого сигнала операционная система понимает, что программа написана грамотным пользователем, и процесс-родитель не собирается интересоваться причинами завершения процесса-ребёнка. Поэтому завершившийся процесс-ребёнок не остаётся в состоянии «завершил исполнение», а немедленно покидает вычислительную систему.

Важным вопросом при программировании с использованием сигналов является вопрос о сохранении реакции на них при порождении нового процесса или замене его пользовательского контекста. При системном вызове ***fork()*** все установленные реакции на сигналы наследуются порождённым процессом. При системном вызове ***exec()*** сохраняются реакции только для тех сигналов, которые игнорировались или обрабатывались по умолчанию. Получение любого сигнала, который до вызова ***exec()*** обрабатывался пользователем, приведёт к завершению процесса.

Прежде, чем продолжить дальнейший разговор о сигналах, нам придётся подробнее остановиться на иерархии процессов в операционной системе.

Понятия группы процессов, сеанса, лидера группы, лидера сеанса, управляющего терминала сеанса

Мы уже говорили, что все процессы в системе связаны родственными отношениями и образуют генеалогическое дерево или лес из таких деревьев, где в качестве узлов деревьев выступают сами процессы, а связями служат отношения родитель-ребёнок. Все эти деревья принято разделять на *группы процессов*, или семьи.



Группа процессов включает в себя один или более процессов и существует, пока в группе присутствует хотя бы один процесс. Каждый процесс обязательно включён в какую-нибудь группу. При рождении нового процесса он попадает в ту же группу процессов, в которой находится его родитель. Процессы могут мигрировать из группы в группу по своему желанию или по желанию другого процесса, как правило – процесса-родителя, (в зависимости от версии UNIX). Многие системные вызовы могут быть применены не к одному конкретному процессу, а ко всем процессам в некоторой группе. Поэтому то, как именно следует объединять процессы в группы, зависит от того, как предполагается их использовать. Чуть позже мы поговорим об использовании групп процессов для передачи сигналов.

В свою очередь, группы процессов объединяются в *сеансы*, образуя, с родственной точки зрения, некие кланы семей. Понятие сеанса изначально было введено в UNIX для логического объединения групп процессов, созданных в результате каждого входа и последующей работы пользователя в системе. С каждым сеансом, поэтому, может быть связан в системе терминал, называемый *управляющим терминалом сеанса*, через который обычно и общаются процессы сеанса с пользователем. Сеанс не может иметь более одного управляющего терминала, и один терминал не может быть управляющим для нескольких сеансов. В то же время могут существовать сеансы, вообще не имеющие управляющего терминала.

Каждая группа процессов в системе получает собственный уникальный номер — pgid (Process Group IDentifier). Узнать этот номер можно с помощью системного вызова *getpgid()*. Используя его, процесс может узнать номер группы для себя самого или для процесса из своего сеанса. Для перевода процесса в другую группу процессов (возможно, с одновременным её созданием) применяется системный вызов *setpgid()*. Перевести в другую группу процесс может либо самого себя (и то не во всякую и не всегда), либо свой процесс-ребёнок, который не выполнял системный вызов *exec()*, т. е. не запускал на выполнение другую программу. При определённых значениях параметров системного вызова создаётся новая группа процессов с идентификатором, совпадающим с идентификатором переводимого процесса, состоящая первоначально только из одного этого процесса. Новая группа может быть создана только таким способом, поэтому идентификаторы групп в системе уникальны. Переход в другую группу без создания новой группы возможен лишь в пределах одного сеанса.

Процесс, идентификатор которого совпадает с идентификатором его группы, называется *лидером группы*. Одно из ограничений на переход из одной группы в другую состоит в том, что лидер группы не может покинуть свою группу.

Каждый сеанс в системе также имеет собственный номер — sid (Session IDentifier). Для того чтобы узнать его, можно воспользоваться системным вызовом *getsid()*. В разных версиях UNIX на него накладываются различные ограничения. В Linux такие ограничения отсутствуют.

Использование системного вызова *setsid()* приводит к созданию новой группы, состоящей только из процесса, который его выполнил (он становится лидером новой группы), и нового сеанса, идентификатор которого совпадает с идентификатором процесса, сделавшего вызов. Такой процесс называется *лидером сеанса*. Этот системный вызов может применять только процесс, не являющийся лидером группы.

Если сеанс имеет управляющий терминал, то этот терминал обязательно приписывается к некоторой группе процессов, входящей в сеанс. Такая группа процессов называется *текущей группой процессов* для данного сеанса. Все процессы, входящие в текущую группу процессов, могут совершать операции ввода-вывода, используя управляющий терминал. Все остальные группы процессов сеанса называются *фоновыми группами*, а процессы, входящие в них – *фоновыми процессами*. При попытке ввода-вывода фонового процесса через управляющий терминал, этот процесс обычно блокируется, либо прекращает свою работу. Передавать управляющий терминал от одной группы процессов к другой может только лидер сеанса. Заметим, что для сеансов, не имеющих управляющего терминала, все процессы являются фоновыми.

При завершении работы процесса – лидера сеанса все процессы из текущей группы сеанса получают сигнал **SIGHUP**, который при стандартной обработке приведёт к их завершению. Таким образом, после завершения лидера сеанса в нормальной ситуации работу продолжат только фоновые процессы. Процессы, входящие в текущую группу сеанса, могут получать сигналы, инициируемые нажатием определенных клавиш на терминале – **SIGINT** при нажатии клавиш <CTRL> и <C>, и **SIGQUIT** при нажатии клавиш <CTRL> и <4>. Стандартная реакция на эти сигналы – завершение процесса (с образованием core файла для сигнала **SIGQUIT**).

Необходимо ввести ещё одно понятие, связанное с процессом, – эффективный идентификатор пользователя. В материалах первого семинара говорилось о том, что каждый пользователь в системе имеет собственный идентификатор – **UID**. Каждый процесс, запущенный пользователем, задействует этот **UID** для определения своих полномочий. Однако иногда, если у исполняемого файла были выставлены соответствующие атрибуты, процесс может выдать себя за процесс, запущенный другим пользователем, являющимся хозяином исполняемого файла. Идентификатор пользователя, от имени которого процесс пользуется полномочиями, и является эффективным идентификатором пользователя для процесса – **EUID**. За исключением выше оговорённого случая, эффективный идентификатор пользователя совпадает с идентификатором пользователя, создавшего процесс.

Системный вызов kill() и команда kill

Из всех перечисленных ранее источников сигнала пользователю доступны только два – команда kill и посылка сигнала процессу с помощью системного вызова *kill()*. Команда kill обычно используется в следующей форме:

```
$ kill [-номер] pid
```

Здесь ***pid*** – это идентификатор процесса, которому посылается сигнал, а ***номер*** – номер сигнала, который посылается процессу. Послать сигнал (если у вас нет полномочий суперпользователя) можно только процессу, у которого эффективный идентификатор пользователя совпадает с идентификатором пользователя, посылающего сигнал. Если параметр ***-номер*** отсутствует, то посылается сигнал **SIGTERM**, обычно имеющий номер **15**, и реакция на него по умолчанию – завершить работу процесса, который получил сигнал.

При использовании команды kill пользователь (если он – не системный администратор), может послать сигнал только процессам, работающим с его правами, то есть имеющим EUID, совпадающий с UID пользователя.

При использовании системного вызова *kill()* послать сигнал (не имея полномочий суперпользователя) можно только процессу или процессам, у которых эффективный идентификатор пользователя совпадает с эффективным идентификатором пользователя процесса, посылающего сигнал.

Системный вызов signal(). Установка собственного обработчика сигнала

Одним из способов изменения поведения процесса при получении сигнала в операционной системе UNIX является использование системного вызова *signal()*.

Этот системный вызов имеет два параметра: один из них задает номер сигнала, реакцию процесса на который требуется изменить, а второй определяет, как именно мы собираемся её менять. Для первого варианта реакции процесса на сигнал – его принудительного игнорирования – применяется специальное значение этого параметра — **SIG_IGN**. Например, если требуется игнорировать сигнал **SIGINT**, начиная с некоторого места работы программы, в этом месте программы вы должны употребить конструкцию

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

Для второго варианта реакции процесса на сигнал – восстановления его обработки по умолчанию – применяется специальное значение этого параметра — **SIG_DFL**. Для третьего варианта реакции процесса на сигнал — пользовательской обработки сигнала — в качестве значения параметра подставляется указатель на пользовательскую функцию обработки сигнала, которая должна иметь прототип вида

```
void *handler(int);
```

Ниже приведён пример скелета конструкции для пользовательской обработки сигнала **SIGHUP**:

```
void *my_handler(int nsig) {

  <обработка сигнала>

}



int main() {

  ...

  (void)signal(SIGHUP, my_handler);

  ...

}
```

В качестве значения параметра в пользовательскую функцию обработки сигнала (в нашем скелете – параметр ***nsig***) передаётся номер возникшего сигнала, так что одна и та же функция может быть использована для обработки нескольких сигналов.

Давайте рассмотрим программу, расположенную в файле **ex2.c**. Эта программа не делает ничего полезного, кроме переустановки реакции на нажатие клавиш <ctrl> и <c> на игнорирование возникающего сигнала и своего бесконечного зацикливания. Откомпилируйте и запустите эту программу, убедитесь, что на нажатие клавиш <ctrl> и <c> она не реагирует, а реакция на нажатие клавиш <ctrl> и <4> осталась прежней.

Давайте рассмотрим теперь программу, расположенную в файле **ex3.c**. Эта программа отличается от программы **ex2.c** тем, что в ней введена обработка сигнала **SIGINT** пользовательской функцией. Откомпилируйте и запустите эту программу, проверьте её реакцию на нажатие клавиш <ctrl> и <c> и на нажатие клавиш <ctrl> и <4>.

До сих пор в примерах мы игнорировали значение, возвращаемое системным вызовом *signal()*. На самом деле этот системный вызов возвращает указатель на предыдущий обработчик сигнала, что позволяет восстанавливать переопределённую реакцию на сигнал. Рассмотрите пример программы **ex4.c**, возвращающей первоначальную реакцию на сигнал **SIGINT** после 5 пользовательских обработок сигнала, откомпилируйте её и запустите на исполнение.

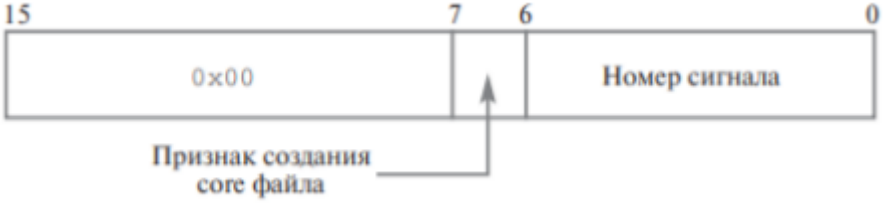
Завершение порождённого процесса. Системный вызов waitpid(). Сигнал SIGCHLD

В материалах семинаров 3–4 при изучении завершения процесса говорилось о том, что если процесс-ребёнок завершает свою работу прежде процесса-родителя, и процесс-родитель явно не указал, что он не заинтересован в получении информации о статусе завершения процесса-ребёнка, то завершившийся процесс не исчезает из системы окончательно, а остается в состоянии **«закончил исполнение»** (зомби-процесс) либо до завершения процесса-родителя, либо до того момента, когда родитель соблаговолит получить эту информацию.

Для получения такой информации процесс-родитель может воспользоваться системным вызовом *waitpid()* или его упрощённой формой *wait()*. Системный вызов *waitpid()* позволяет процессу-родителю синхронно получить данные о статусе завершившегося процесса-ребёнка либо блокируя процесс-родитель до завершения процесса-ребёнка, либо без блокировки при его периодическом вызове с опцией **WNOHANG**.

Эти данные занимают 16 бит и в рамках нашего курса могут быть расшифрованы следующим образом:
1. Если процесс завершился при помощи явного или неявного вызова функции *exit()*, то данные выглядят так (старший бит находится слева)

2. Если процесс был завершён сигналом, то данные выглядят так (старший бит находится слева)



Каждый процесс-ребёнок при завершении работы посылает своему процессу-родителю специальный сигнал **SIGCHLD**, на который у всех процессов по умолчанию установлена реакция «игнорировать сигнал». Наличие такого сигнала совместно с системным вызовом *waitpid()* позволяет организовать асинхронный сбор информации о статусе завершившихся порождённых процессов процессом-родителем.

Используя системный вызов *signal()*, вы можете явно установить игнорирование этого сигнала (**SIG_IGN**), тем самым проинформировав систему, что нас не интересует, каким образом завершатся порождённые процессы. В этом случае зомби-процессов возникать не будет, но и применение системных вызовов *wait()* и *waitpid()* будет запрещено.

Для закрепления материала рассмотрим пример программы **ex5.c** с асинхронным получением информации о статусе завершения порождённого процесса.

В этой программе родитель порождает два процесса. Один из них завершается с кодом 200, а второй зацикливается. Перед порождением процессов родитель устанавливает обработчик прерывания для сигнала **SIGCHLD**, а после их порождения уходит в бесконечный цикл. В обработчике прерывания вызывается *waitpid()* для любого порождённого процесса. Так как в обработчик мы попадаем, когда какой-либо из процессов завершился, системный вызов не блокируется, и мы можем получить информацию об идентификаторе завершившегося процесса и причине его завершения. Второй порождённый процесс завершайте с помощью команды *kill* с каким-либо номером сигнала. Родительский процесс также будет необходимо завершать командой *kill*.

Понятие о надёжности сигналов. POSIX-функции для работы с сигналами

Основным недостатком системного вызова *signal()* является его низкая надёжность.

Во многих вариантах операционной системы UNIX установленная при его помощи обработка сигнала пользовательской функцией выполняется только один раз, после чего автоматически восстанавливается реакция на сигнал по умолчанию. Для постоянной пользовательской обработки сигнала необходимо каждый раз заново устанавливать реакцию на сигнал прямо внутри функции-обработчика.

В системных вызовах и пользовательских программах могут существовать критические участки, на которых процессу недопустимо отвлекаться на обработку сигналов. Мы можем выставить на этих участках реакцию «игнорировать сигнал» с последующим восстановлением предыдущей реакции, но если сигнал всё-таки возникнет на критическом участке, то информация о его возникновении будет безвозвратно потеряна.

Наконец, последний недостаток связан с невозможностью определения количества сигналов одного и того же типа, поступивших процессу, пока он находился в состоянии **готовность**. Сигналы одного типа в очередь не ставятся! Процесс может узнать о том, что сигнал или сигналы определённого типа были ему переданы, но не может определить их количество.

Эту черту можно проиллюстрировать, слегка изменив программу с асинхронным получением информации о статусе завершившихся процессов, рассмотренную вами ранее. Пусть в новой программе **ex6.c** процесс-родитель порождает в цикле 10 новых процессов, каждый из которых после некоторой задержки завершается со своим собственным кодом, после чего уходит в бесконечный цикл. Сколько сообщений о статусе завершившихся детей мы ожидаем получить? Десять! А сколько получим? It depends... Откомпилируйте, прогоните и посчитайте.

Последующие версии System Y и BSD пытались устранить эти недостатки собственными средствами. Единый способ более надёжной обработки сигналов появился с введением POSIX-стандарта на системные вызовы UNIX. Набор функций и системных вызовов для работы с сигналами был существенно расширен и построен таким образом, что позволял временно блокировать обработку определённых сигналов, не допуская их потери. Однако проблема, связанная с определением количества пришедших сигналов одного типа, по-прежнему остаётся актуальной. (Надо отметить, что подобная проблема существует на аппаратном уровне и для внешних прерываний. Процессор зачастую не может определить, какое количество внешних прерываний с одним номером возникло, пока он выполнял очередную команду.)

Рассмотрение POSIX-сигналов выходит за рамки нашего курса. Желающие могут самостоятельно просмотреть описания функций и системных
вызовов *sigemptyset()*, *sigfillset()*, *sigaddset()*, *sigdelset()*, *sigismember()*, *sigaction()*, *sigprocmask()*, *sigpending()*, *sigsuspend()* в UNIX Manual.

Сигналы SIGUSR1 и SIGUSR2. Использование сигналов для передачи информации между процессами

В операционной системе UNIX существует два сигнала, источниками которых могут служить только системный вызов *kill()* или команда kill – это сигналы **SIGUSR1** и **SIGUSR2**. Обычно их применяют для передачи информации о происшедшем событии от одного пользовательского процесса другому в качестве сигнального средства связи. В материалах семинара 5 речь шла о том, что pipe является однонаправленным каналом связи, и что для организации связи через один pipe в двух направлениях необходимо задействовать механизмы взаимной синхронизации процессов.

Представим себе, что в системе нет других средств связи между процессами, кроме сигналов, а от одного процесса к другому нужно передать целое число. Это можно сделать побитово, используя сигналы **SIGUSR1** и **SIGUSR2**.

При реализации нитей исполнения в операционной системе Linux
сигналы **SIGUSR1** и **SIGUSR2** используются для организации синхронизации между процессами, представляющими нити исполнения, и процессом-координатором в служебных целях. Поэтому пользовательские программы, применяющие в своей работе нити исполнения, не могут задействовать сигналы **SIGUSR1** и **SIGUSR2**.