Понятие виртуальной файловой системы

На семинарах 16–17 рассказывалось про устройство физической файловой системы s5fs. Существуют и другие физические файловые системы, имеющие архитектуру, отличную от архитектуры s5fs (иные способы отображения файла на пространство физического носителя, иное построение директорий и т. д.). Современные версии UNIX-подобных операционных систем умеют работать с разнообразными файловыми системами, различающимися своей организацией. Такая возможность достигается с помощью разбиения каждой файловой системы на зависимую и независимую от конкретной реализации части, подобно тому, как в лекционной теме 12, посвященной вопросам ввода-вывода, мы отделяли аппаратно-зависимые части для каждого устройства – драйверы – от общей базовой подсистемы ввода-вывода. Независимые части всех файловых систем одинаковы и представляют для всех остальных элементов ядра абстрактную файловую систему, которую принято называть виртуальной файловой системой. Зависимые части для различных файловых систем могут встраиваться в ядро на этапе компиляции либо добавляться к нему динамически по мере необходимости, без перекомпиляции системы (как в системах с микроядерной архитектурой).

Рассмотрим схематично устройство виртуальной файловой системы. В файловой системе s5fs данные о физическом расположении и атрибутах каждого открытого файла представлялись в операционной системе структурой данных в таблице индексных узлов открытых файлов (см. предыдущий семинар), содержащей информацию из индексного узла файла во вторичной памяти. В виртуальной файловой системе, в отличие от s5fs, каждый файл характеризуется не индексным узлом inode, а некоторым виртуальным узлом vnode. Соответственно, вместо таблицы индексных узлов открытых файлов в операционной системе появляется таблица виртуальных узлов открытых файлов. При открытии файла в операционной системе для него заполняется (если, конечно, не был заполнен раньше) элемент таблицы виртуальных узлов открытых файлов, в котором хранятся, как минимум, тип файла, счетчик числа открытий файла, указатель на реальные физические данные файла и, обязательно, указатель на таблицу системных вызовов, совершающих операции над файлом, – таблицу операций. Реальные физические данные файла (равно как и способ расположения файла на диске и т. п.) и системные вызовы, реально выполняющие операции над файлом, уже не являются элементами виртуальной файловой системы. Они относятся к одной из зависимых частей файловой системы, так как определяются ее конкретной реализацией.

При выполнении операций над файлами по таблице операций, чей адрес содержится в vnode, определяется системный вызов, который будет на самом деле выполнен над реальными физическими данными файла, чей адрес также находится в vnode. В случае с s5fs данные, на которые ссылается vnode, – это как раз данные индексного узла, рассмотренные на семинарах 15–16 и в лекционной теме 11. Заметим,

что таблица операций является общей для всех файлов, принадлежащих одной и той же файловой системе.

Блочные, символьные устройства. Понятие драйвера. Блочные, символьные драйверы, драйверы низкого уровня. Файловый интерфейс

Обремененные знаниями об устройстве современных файловых систем в UNIX, мы можем, наконец, заняться вопросами реализации подсистемы ввода-вывода.

В лекционной теме 12 (раздел «Структура системы ввода-вывода») речь шла о том, что все устройства ввода-вывода можно разделить на относительно небольшое число типов, в зависимости от набора операций, которые могут ими выполняться. Такое деление позволяет организовать «слоистую» структуру подсистемы ввода-вывода, вынеся все аппаратно-зависимые части в драйверы устройств, с которыми взаимодействует базовая подсистема ввода-вывода, осуществляющая стратегическое управление всеми устройствами.

В операционной системе UNIX принята упрощенная классификация устройств (см. лекционную тему 12, раздел «Систематизация внешних устройств и интерфейс между базовой подсистемой ввода-вывода и драйверами»): все устройства разделяются по способу передачи данных на символьные и блочные. Символьные устройства осуществляют передачу данных байт за байтом, в то время как блочные устройства передают блок байт как единое целое. Типичным примером символьного устройства является клавиатура, примером блочного устройства – жесткий диск. Непосредственное взаимодействие операционной системы с устройствами ввода-вывода обеспечивают их драйверы. Существует пять основных случаев, когда ядро обращается к драйверам.

1. Автоконфигурация. Происходит в процессе инициализации операционной системы, когда ядро определяет наличие доступных устройств.
2. Ввод-вывод. Обработка запроса ввода-вывода.
3. Обработка прерываний. Ядро вызывает специальные функции драйвера для обработки прерывания, поступившего от устройства, в том числе, возможно, для планирования очередности запросов к нему.
4. Специальные запросы. Например, изменение параметров драйвера или устройства.
5. Повторная инициализация устройства или остановка операционной системы.

Так же как устройства подразделяются на символьные и блочные, драйверы тоже существуют символьные и блочные. Особенностью блочных устройств является возможность организации на них файловой системы, поэтому блочные драйверы обычно используются файловой системой UNIX. При обращении к блочному устройству, не содержащему файловой системы, применяются специальные драйверы

низкого уровня, как правило, представляющие собой интерфейс между ядром операционной системы и блочным драйвером устройства.

Для каждого из этих трех типов драйверов были выделены основные функции, которые базовая подсистема ввода-вывода может совершать над устройствами и драйверами: инициализация устройства или драйвера, временное завершение работы устройства, чтение, запись, обработка прерывания, опрос устройства и т. д. (об этих операциях уже говорилось в лекционной теме 12, раздел «Систематизация внешних устройств и интерфейс между базовой подсистемой ввода-вывода и драйверами»). Эти функции были систематизированы и представляют собой интерфейс между драйверами и базовой подсистемой ввода-вывода.

Каждый драйвер определенного типа в операционной системе UNIX получает собственный номер, который по сути дела является индексом в массиве специальных структур данных операционной системы – *коммутаторе устройств* соответствующего типа. Этот индекс принято также называть *старшим номером устройства*, хотя на самом деле он относится не к устройству, а к драйверу. Несмотря на наличие трех типов драйверов, в операционной системе используется всего два коммутатора: для блочных и символьных драйверов. Драйверы низкого уровня распределяются между ними по преобладающему типу интерфейса (к какому типу ближе – в такой массив и заносятся). Каждый элемент коммутатора устройств обязательно содержит адреса (точки входа в драйвер), соответствующие стандартному набору функций интерфейса, которые и вызываются операционной системой для выполнения тех или иных действий над устройством и/или драйвером.

Помимо старшего номера устройства существует еще и *младший номер устройства*, который передается драйверу в качестве параметра и смысл которого определяется самим драйвером. Например, это может быть номер раздела на жестком диске (partition), доступ к которому должен обеспечить драйвер (надо отметить, что в операционной системе UNIX различные разделы физического носителя информации рассматриваются как различные устройства). В некоторых случаях младший номер устройства может не использоваться, но для единообразия он должен присутствовать. Таким образом, пара драйвер–устройство всегда однозначно определяется в операционной системе заданием пары номеров (старшего и младшего номеров устройства) и типа драйвера (символьный или блочный).

Для связи приложений с драйверами устройств операционная система UNIX использует *файловый интерфейс*. В числе типов файлов на предыдущем семинаре упоминались специальные файлы устройств. Так вот, каждой тройке тип–драйвер–устройство в файловой системе соответствует специальный файл устройства, который не занимает на диске никаких логических блоков, кроме индексного узла. В качестве атрибутов этого файла помимо обычных атрибутов используются соответствующие старший и младший номера устройства и тип драйвера (тип драйвера определяется по типу файла: ибо есть специальные файлы символьных устройств и специальные файлы блочных устройств, а номера устройств занимают

место длины файла, скажем, для регулярных файлов). Когда открывается специальный файл устройства, операционная система, в числе прочих действий, заносит в соответствующий элемент таблицы открытых виртуальных узлов указатель на набор функций интерфейса из соответствующего элемента коммутатора устройств. Теперь при попытке чтения из файла устройства или записи в файл устройства виртуальная файловая система будет транслировать запросы на выполнение этих операций в соответствующие вызовы нужного драйвера. Обычно принято специальные файлы устройств сосредотачивать в директории `/dev`.

Мы не будем останавливаться на практическом применении файлового интерфейса для работы с устройствами ввода-вывода, поскольку это выходит за пределы нашего курса, а вместо этого приступим к изложению концепции сигналов в UNIX, тесно связанных с понятиями аппаратного прерывания, исключения и программного прерывания.

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 на выполнение операции ввода-вывода некоторым устройством 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 механизма прерываний. При втором способе процессор имеет специальный вход, на который устройства ввода-вывода непосредственно или через контроллер прерываний, 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 на выполнение 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 невозможности выполнении команды процессором (неправильный адрес в команде, защита памяти, возникло деление на ноль и т.д.). В этом случае процессор не заканчивает выполнение команды, а поступает, как и при прерывании, сохраняя свое состояние до момента начала ее выполнения.

Этим же механизмом часто пользуются и для реализации так называемых программных прерываний (software interrupt, trap), используемых, 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 четко 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 свое регулярное исполнение, и управление передается 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 жестко регламентированы.

Хотя любой сигнал в конечном смете доставляется процессу операционной системой, первоначальные источники происхождения сигналов могут быть различны. 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, например, при попытке записи в pipe, из которого некому читать).
4. Пользователя за терминалом (при нажатии определенной комбинации клавиш).
5. Системы управления заданиями (при выполнении команды `kill` - мы рассмотрим ее 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 мы говорили на лекции, посвященной кооперации процессов.

Существует три варианта реакции процесса на сигнал:

1. Произвести обработку по умолчанию (проигнорировать сигнал, приостановить процесс до получения другого специального сигнала, продолжить работу ранее остановленного процесса), либо завершить работу процесса с образование core файла или без оного).
2. Принудительно проигнорировать сигнал.
3. Выполнить обработку сигнала, специфицированную пользовательской функцией.

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.

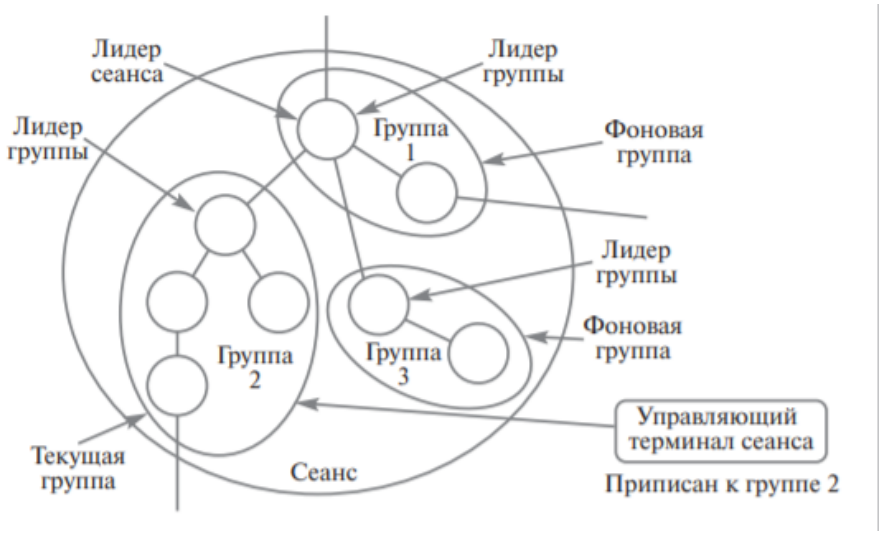
Необходимо отметить, что игнорирование сигналов по умолчанию и принудительное игнорирование сигналов — это принципиально разные реакции. Так, например, при завершении процесса-ребенка процесс-родитель получает сигнал `SIGCHLD`, который по умолчанию процессом-родителем игнорируется. При этом завершившийся 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-ребенка. Поэтому завершившийся процесс-ребенок не остается в состоянии «завершил исполнение», а немедленно покидает вычислительную систему.

Важным вопросом при программировании с использованием сигналов является вопрос о сохранении реакции на них при рождении нового процесса или замене его пользовательского контекста. При системном вызове `fork()` все установленные реакции на сигналы наследуется порожденным 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 пользовательской функцией, приведет к завершению процесса.

Прежде, чем продолжить дальнейший разговор о сигналах, нам придется 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, образуя генеалогическое дерево или лес из таких деревьев, где в качестве узлов деревьев выступают сами процессы, а связями служат отношения родитель-ребенок. Все эти деревья принято разделять на *группы процессов*, так сказать, их семьи.



Группа процессов включает в себя один или более процессов и существует, пока в группе присутствует хотя бы один процесс. Каждый процесс обязательно включен 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 собственному желанию или по желанию другого процесса, как правило – процесса-родителя, (в зависимости от версии UNIX) с определенными ограничениями. Многие системные вызовы могут быть применены не к одному конкретному процессу, а ко всем процессам в некоторой группе. Поэтому то, как именно вы будете объединять процессы в группы, зависит от того, как вы собираетесь 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 (логина) 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 свой собственный уникальный номер — pgid (Process Group IDentifier), узнать который процесс может с помощью специального системного вызова. Для перевода процесса в другую группу процессов, возможно с одновременным ее созданием, применяется еще один системный вызов. Перевести в другую группу процесс может либо самого себя (и то не во всякую и не всегда), либо свой процесс-ребенок, который не выполнял системный вызов `exec()`, т.е. не запускал на выполнение другую программу. При определенных значениях параметров системного вызова создается 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 только 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 свой собственный номер — sid (Session IDentifier). При определенных условиях процесс может использовать специальный системный вызов, который приводит к созданию новой группы, состоящий 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 он 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 приведет к их завершению. Таким образом, после завершения лидера сеанса в нормальной ситуации работать 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 процесса на эти сигналы — завершение процесса (в одном случае без core файла, в другом – с core файлом). Точно также процессы текущей группы получат другой сигнал

SIGTSTP при нажатии на клавиатуре клавиш <ctrl> и <z>, стандартная реакция на который — остановка процесса. Важно понимать разницу между остановкой процесса и его завершением. При остановке процесс в дальнейшем может быть возобновлен с места остановки, он продолжает удерживать закрепленные за ним ресурсы. У завершенного процесса нет никаких закрепленных ресурсов.

Нам понадобится еще одно понятие, связанное с процессом, — эффективный идентификатор пользователя. В начале семестра мы говорили, что каждый пользователь в системе имеет свой собственный идентификатор — UID. Каждый процесс, запущенный пользователем, использует этот UID для определения своих полномочий. Однако иногда, если у исполняемого файла был выставлен соответствующий атрибут (флаг), процесс может прикинуться процессом, запущенным пользователем, являющимся хозяином исполняемого файла. Идентификатор пользователя, от имени которого процесс пользуется полномочиями, и является эффективным идентификатором пользователя для процесса — EUID. За исключением выше оговоренного 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 (если у вас нет 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()`.

О возвращаемом значении системного вызова `signal()` мы поговорим позже.

Этот системный вызов имеет два параметра: один из них задает номер сигнала, реакцию процесса на который мы хотим изменить, а второй определяет, как именно мы собираемся ее менять. Для второго 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 мы должны употребить конструкцию

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

Для первого 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 приведен 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`) передается номер возникшего сигнала, так что одна и та же функция может быть использована для различной обработки нескольких сигналов.

Давайте рассмотрим пример 17-2.c, который не делает ничего полезного, кроме переустановки реакции на нажатие клавиш <ctrl>+<c>. Завершить её можно при помощи <ctrl>+<4>.

***Задача 1 (1 балл)**:*

*Модифицируйте программу из примера **17-2.c** так, чтобы она перестала реагировать и на нажатие клавиш <ctrl> и <4>. Откомпилируйте и запустите её, убедитесь в отсутствии её реакций на внешние раздражители. Снимать программу придётся теперь с другого терминала командой kill. Узнать идентификаторы своих работающих процессов можно с помощью команды ps -a.*

Далее рассмотрим пример 17-3.c. Он отличается тем, что в нем введена обработка сигнала пользовательской функцией.

***Задача 2 (2 балла)**:*

*Модифицируйте программу из предыдущего примера **17-3.c** так, чтобы она печатала сообщение и об нажатии клавиш <ctrl> и <4>. На разные комбинации клавиш должны печататься разные сообщения. Используйте одну и ту же функцию для обработки сигналов **SIGINT** и **SIGQUIT**. Откомпилируйте и запустите её, убедитесь в правильной работе. Снимать программу также придётся с другого терминала командой kill.*

Восстановление предыдущей реакции на сигнал.

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 переопределенную реакцию на сигнал. Рассмотрите пример программы, возвращающей первоначальную реакцию на сигнал `SIGINT` после 5 пользовательских обработок сигнала, — 17-4.c, откомпилируйте ее и запустите на исполнение.

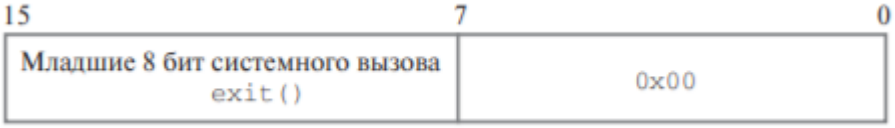
Завершение порождённого процесса. Системный вызов waitpid(). Сигнал SIGCHLD

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.

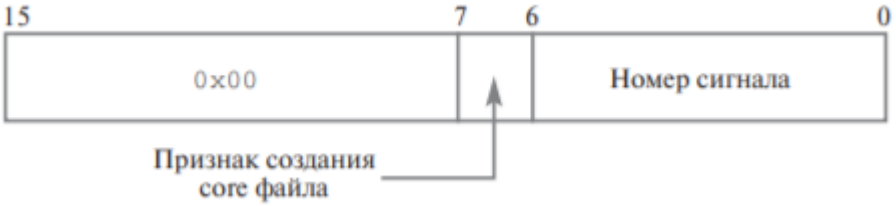
Для получения такой информации процесс-родитель может воспользоваться системным вызовом `waitpid()` или его упрощённой формой `wait()`. Системный вызов

`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 14-5.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 14-6.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.

Задачи на семинар

***Задача 1 (1 балл)**:*

См. выше.

***Задача 2 (2 балла)**:*

См. выше.

***Задача 3 (5 баллов)**:*

*К счастью, для сигнала **SIGCHLD** последняя проблема с надёжностью может быть легко разрешена. Модифицируйте обработку этого сигнала в программе **17-6.c**, так, чтобы процесс-родитель все-таки сообщал о статусе всех завершившихся детей. Изменять в программе можно только обработчик сигнала, используя только сведения текущего семинара, без всяких POSIX сигналов!*