Семинар 12-13. Организация ввода-вывода в UNIX. Файлы устройств. Аппарат прерываний. Сигналы в UNIX.

Понятие виртуальной файловой системы. На предыдущих занятиях мы с вами рассматривали устройство файловой системы s5fs. Существуют и другие файловые системы, имеющие архитектуру, существенно отличающуюся 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, отличающимися своей организацией. Структура системы ввода-вывода в лекции Такая возможность достигается с помощью разбиения каждой существующей файловой системы на зависимую и независимую от конкретной реализации части, подобно тому, как на лекции, посвященной вопросам ввода-вывода, мы отделяли аппаратно-зависимые части для каждого устройства (драйвера) 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 каждый открытый файл представлялся в операционной ситеме структурой данных в таблице индексных узлов открытых файлов, 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. 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, адрес которой содержится в vnode, определяется системный вызов, который будет на самом деле выполнен над реальными физическими данными файла, адрес которых также находится в

vnode. Метод индексных узлов в лекции В случае с s5fs, данные, на которые ссылается vnode, — это как раз данные индексного узла, рассмотренные на предыдущих семинарах и на соответствующей лекции. Заметим, что таблица операций является общей для всех файлов, принадлежащих одной и той же файловой системе.

Операции над файловыми системами. Монтирование файловых систем. На предыдущих семинарах мы рассматривали только одну файловую систему, расположенную в одном разделе физического носителя. Как только мы переходим к взаимному сосуществованию нескольких файловых систем в рамках одной операционной системы, встает вопрос о логическом объединении структур этих файловых систем. При начале работы операционной системы нам изначально доступна лишь одна, так называемая корневая, файловая система. Операция монтирования в лекции Прежде, чем приступить к работе с файлом, лежащим в некоторой другой файловой системе, мы должны встроить ее в уже существующий ациклический граф файлов. Эта операция - операция над файловой системой - называется монтированием файловой системы (mount).

Для монтирования файловой системы в существующем графе должна быть найдена или создана некоторая пустая директория - точка монтирования, к которой и присоединится корень монтируемой файловой системы. При операции монтировании в ядре заводятся структуры данных, описывающие файловую систему, а в vnode для точки монтирования файловой системы помещается специальная информация.

Монтирование файловых систем обычно является прерогативой системного администатора и осуществляется командой операционной системы mount в ручном режиме, либо автоматически при старте операционной системы. Использование этой команды без параметров не требует специальных полномочий и позволяет пользователю получить информацию о всех подмонтированных файловых системах и соответствующих им физических устройствах. Для пользователя также обычно разрешается монтирование файловых систем, расположенных на гибких магнитных дисках. Для первого накопителя на гибких магнитных дисках такая команда в Linux будет выглядеть следующим образом:

mount /dev/fd0 <имя пустой директории>,

где <имя пустой директории> описывает точку монтирования, а /dev/fd0 - специальный файл устройства, соответствующего этому накопителю (о специальных файлах устройств мы подробнее поговорим в следующем разделе).

Если мы не собираемся использовать подмонтированную файловую систему в дальнейшем (например, хотим вынуть ранее подмонтированную дискету), нам необходимо выполнить операцию логического разъединения смонтированных файловых систем (umount). Для этой операции, которая тоже, как правило, является привилегией

системного администратора, используется команда umount (может выполняться в ручном режиме или автоматически при завершении работы операционной системы). Для пользователя обычно доступна команда по отмонтированию файловой системы на дискете в форме

umount <имя точки монтирования>,

где <имя точки монтирования> - это <имя пустой директории>, использованное ранее в команде mount, или в форме

umount /dev/fd0,

где /dev/fd0 - специальный файл устройства, соответствующего первому накопителю на гибких магнитных дисках. Nota beneЗаметим, что для последующей корректной работы операционной системы при удалении физического носителя информации, обязательно необходимо предварительное логическое разъединение файловых систем, если они перед этим были объединены.

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.

Структура системы ввода-вывода в лекции На лекции мы говорили о том, что все возможные устройства ввода-вывода можно разделить на относительно небольшое число типов, отличающихся по набору операций, которые могут быть ими выполнены, считая все остальные различия несущественными. Такое деление позволяет организовать подсистему ввода-вывода слоеным образом, вынеся все аппаратно-зависимые части в драйвера устройств, с которыми взаимодействует базовая подсистема ввода-вывода, осуществляющая стратегическое управление всеми устройствами.

Классификация устройств в лекции В операционной системе UNIX принята упрощенная классификация устройств: 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, типичным примером блочного устройства является жесткий диск. Непосредственное взаимодействие операционной системы с устройствами ввода-вывода обеспечивают их драйверы. Существует пять основных случаев, когда ядро обращается к драйверам:

Автоконфигурация. Происходит в процессе инициализации операционной системы, когда ядро определяет наличие доступных устройств.

Ввод-вывод. Обработка запроса ввода-вывода. Обработка прерываний. Ядро вызывает специальные функции драйвера для обработки прерывания, поступившего от устройства, в том числе, возможно, для планирования очередности запросов к нему. Специальные запросы. Например, изменение параметров драйвера или устройства.

Повторная инициализация устройства или останов операционной системы. Так же как устройства подразделяются на символьные и блочные, так и драйверы 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 мы уже говорили на лекции). Эти функции были систематизированы и представляют собой интерфейс между драйверами и базовой подсистемой ввода-вывода.

Каждый драйвер определенного типа в операционной системе UNIX получает свой 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. Драйвера 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 обязан 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 мы упоминали 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 для, скажем, регулярных файлов). Когда открывается специальный файл устройства, операционная система, в числе прочих действий, заносит в соотвествующий 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 необходимые вызовы нужного драйвера.

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. На лекциях уже вводились понятия аппаратного прерывания, исключения и программного прерывания. Кратко напомним сказанное.

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 непосредственно, или используя 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 некотором 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, 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, для чего не лишним будет в очередной раз обратиться к лекциям.

Как правило, обработку аппаратных прерываний от устройств ввода-вывода производит сама операционная система, не доверяя работу с системными ресурсами процессам пользователя. Обработка же исключительных ситуаций и некоторых программных прерываний вполне может быть возложена на пользовательский процесс через механизм сигналов.

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.

Процесс может получить сигнал от:

Hardware (при возникновении исключительной ситуации).

Другого процесса, выполнившего системный вызов передачи сигнала.

Операционной системы (при наступлении некоторых событий).

Терминала (при нажатии определенной комбинации клавиш).
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 4.

Существует три варианта реакции процесса на сигнал:
Принудительно проигнорировать сигнал.
Произвести обработку по умолчанию (проигнорировать, остановить процесс (перевести в состояние ожидания до получения другого специального сигнала), либо завершить работу с образование core файла или без оного).
Выполнить обработку сигнала, специфицированную пользователем.

Изменить реакцию процесса на сигнал можно с помощью специальных системных вызовов, которые мы рассмотрим позже. Реакция на некоторы 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.

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 пользователем, 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. Системные вызовы getpgrp(), setpgrp(), getpgid(), setpgid(), getsid(), setsid(). 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, 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 (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 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 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. 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. К сожалению, не во всех версиях UNIX присутствует данный системный вызов. Здесь мы сталкиваемся с тяжелым наследием разделения линий UNIX'ов на линию BSD и линию System V, которое будет нас преследовать почти на всем протяжении данной темы. Вместо вызова getpgid() в таких системах существует системный вызов getpgrp(), который возвращает вам номер группы только для текущего процесса.

Для перевода процесса в другую группу процессов, возможно с одновременным ее 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-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.

В некоторых разновидностях UNIX системный вызов setpgid() отсутствует, а вместо него существует системный вызов setpgrp(), который умеет только создавать новую группу процессов с идентификатором, совпадаюшим с идентификатором текущего процесса, и переводить текущий процесс в нее. (В некоторых разновидностях UNIX, где совместно сосуществуют вызовы setpgrp() и setpgid(), например в Solaris, вызов setpgrp() ведет себя иначе - он аналогичен рассматриваемому ниже вызову setsid().)

Процесс, идентификатор которого совпадает с идентификатором его группы, называется лидером группы. Одно из ограничений на применение вызовов setpgid() и setpgrp() состоит в том, что лидер группы не может перебраться в другую группу.

Каждый сеанс в системе также имеет свой собственный номер. 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, 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 получит сигналы, которые стандартно приводят к прекращению работы процесса. 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.

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 были выставлены соответствующие атрибуты, процесс может прикинуться процессом, запущенным другим пользователем. 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 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().

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 первого варианта реакции процесса на сигнал - его игнорирования - применется 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 второго варианта реакции процесса на сигнал - восстановления его обработки по умолчанию - применется специальное значение этого параметра SIG_DFL. Для третьего варианта реакции процесса на сигнал 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 склете - параметр nsig) передается номер возникшего сигнала, так что одна и та же функция может быть использована для различной обработки нескольких сигналов.

Сигналы SIGUSR1 и SIGUSR2. Использование сигналов для синхронизации процессов. 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 между процессами пользователя.

Завершение порожденного процесса. Системный вызов waitpid(). Сигнал SIGCHLD. При изучении завершения процесса, мы с вами говорили о том, что если процесс-ребенок завершает свою работу прежде процесса-родителя, и процесс-родитель явно не указал, что он не заинтересован в получении информации о статусе завершения процесса-ребенка, 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 упрощенной формой wait(). Системный вызов waitpid() позволяет процессу-родителю синхронно получить данные о статусе завершившегося процесса-ребенка, либо блокируя процесс-родитель до завершения процесса-ребенка, 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:

Каждый процесс-ребенок 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 порожденных процессов процессом-родителем.

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

Возникновение сигнала SIGPIPE при попытке записи в pipe или FIFO, который никто не собирается читать. При обсуждении работы с pip'ами и FIFO мы говорили, что для них системные вызовы read() и write() имеют определенные особенности поведения. Одной из таких особенностей является получение сигнала SIGPIPE процессом, который пытается записывать информацию в pipe или в FIFO в том случае, когда читать ее уже некому (нет ни одного процесса, который имеет соответствующий pipe или FIFO открытым для чтения). Реакция по умолчанию на этот сигнал - прекратить работу процесса. Теперь мы с вами уже можем написать корректную обработку этого сигнала пользователем для, например, элегантного прекращения работы пишущего процесса. Однако, нам для полноты картины необходимо познакомиться с особенностями поведения некоторых системных вызовов при получении сигналов процессом во время их выполнения.

По ходу нашего курса мы с вами познакомились с рядом системных вызовов, которые могут блокировать процесс во время своего выполнения. К их числу относятся системный вызов open() при открытии FIFO, системные вызовы read() и write() при работе с pip'ами и FIFO, системные вызовы msgsnd() и msgrcv() при работе с очередями сообщений, системный вызов semop() при работе с семафорами и т.д. Что произойдет с процессом, если он, выполняя один из этих системных вызовов, получит какой-либо

сигнал. Дальнейшее поведение процесса зависит от установленной для него реакции на этот сигнал.

Если реакция на полученный сигнал была "игнорировать сигнал" (независимо от того, установлена она по умолчанию или пользователем с помощью системного вызова signal()), то поведение процесса не изменится.

Если реакция на полученный сигнал установлена по умолчанию и заключается в прекращении работы процесса, то процесс перейдет в состояние закончил исполнение.

Если реакция процесса на сигнал заключается в выполнении пользовательской функции, то процесс выполнит эту функцию (если он находился в состоянии ожидание, он перейдет в состояние готовность и затем в состояние исполнение) и вернется из системного вызова с констатацией ошибочной ситуации. Отличить такой возврат от действительно ошибочной ситуации можно с помощью значения системной переменной errno, которая в этом случае примет значение EINTR.

После этого краткого обсуждения становится до конца ясно, как корректно обработать ситуацию "никто не хотел прочитать" для системного вызова write(). Чтобы пришедший сигнал SIGPIPE не завершил работу нашего процесса по умолчанию, мы должны его самостоятельно обработать (функция-обработчик при этом может быть и пустой!). Но этого мало. Поскольку нормальный ход выполнения системного вызова был нарушен сигналом, мы вернемся из него с отрицательным значением, которое свидетельствует об ошибке. Проанализировав значение системной переменной errno на предмет совпадения с значением EINTR, мы можем отличить возникновение сигнала SIGPIPE от других ошибочных ситуаций (неправильные значения параметров и т.д.) и грациозно продолжить работу программы.