****№ занятия:**** ****5.**** Лекция ****№**** ****4****

****Тема:**** ****Сигналы.****

Вопрос 1. Концепция сигналов. Типы сигналов

Концепция сигналов имеет несколько аспектов, связанных с тем, каким образом ядро посылает сигнал процессу, каким образом процесс обрабатывает сигнал и управляет реакцией на него. Посылая сигнал процессу, ядро устанавливает в единицу разряд в поле сигнала записи таблицы процессов, соответствующий типу сигнала. Если процесс находится в состоянии приостанова с приоритетом, допускающим прерывания, ядро возобновит его выполнение. На этом роль отправителя сигнала (процесса или ядра) исчерпывается. Процесс может запоминать сигналы различных типов, но не имеет возможности запоминать количество получаемых сигналов каждого типа.

Ядро проверяет получение сигнала, когда процесс собирается перейти из режима ядра в режим задачи, а также когда он переходит в состояние приостанова или выходит из этого состояния с достаточно низким приоритетом планирования. Ядро обрабатывает сигналы только тогда, когда процесс возвращается из режима ядра в режим задачи. Таким образом, сигнал не оказывает немедленного воздействия на поведение процесса, исполняемого в режиме ядра.

Если процесс исполняется в режиме задачи, а ядро тем временем обрабатывает прерывание, послужившее поводом для посылки процессу сигнала, ядро распознает и обработает сигнал по выходе из прерывания.

Таким образом, процесс не будет исполняться в режиме задачи, пока какие-то сигналы остаются необработанными.

Сигналы являются способом передачи от одного процесса другому или от ядра операционной системы какому-либо процессу уведомления о возникновении определенного события.

Сигналы можно рассматривать как простейшую форму межпроцессного взаимодействия.

Процесс может посылать сигналы другим процессам и самому себе. Основное назначение сигналов – управление процессами. В частности, с помощью сигналов пользователи, ядро могут прерывать зависшие процессы или реализовать простые средства межпроцессорного взаимодействия.

Сигналы сообщают процессам о возникновении асинхронных событий. Посылка сигналов производится процессами друг другу, с помощью функции *kill* или ядром.

Каждому типу сигналов присвоено мнемоническое имя:

*SIGALRM* – сигнал таймера. Посылается ядром. Временной интервал может быть установлен самим процессом функцией *alarm*;

*SIGABRT* – аварийное завершение процесса. Может быть инициализирован

*API abort*;

*SIGBUS* – сигнал ошибки на шине (при попытке обращения к допустимому виртуальному адресу, для которого нет физической страницы). Вызывает аварийное завершение;

*SIGCHLD –* посылается в родительский процесс при завершении дочернего;

*SIGCONT* – возобновлять остановленный процесс;

*SIGHUP* – разрыв связи с управляющим терминалом;

*SIGILL* – попытка выполнить недопустимую машинную команду;

*SIGINT* – прерывание процесса. Обычно генерируется клавишей *DELETE*

или *CTRL+C*;

*SIGKILL* – уничтожение процесса. Может генерироваться командой *kill*; *SIGPIPE* – попытка выполнить недопустимую запись в канал;

*SIGPOLL* – сигнал о возникновении одного из опрашиваемых событий;

*SIGPROF* – сигнал профилирующего таймера. Работает при взаимодействии с сигналом *SIGALARM*;

*SIGQUIT* – выход из процесса. Обычно *CTRL* \;

*SIGSEGV* – ошибка сегментации;

*SIGSTOP* – остановить процесс;

*SIGSYS* – некорректный системный вызов;

*SIGTERM* – завершение процесса. Обычно *kill<pid*>; *SIGTRAP* – сигнал трассировочного прерывания; *SIGTSTP* – остановить процесс. Обычно *CTRL+Z*;

*SIGTTIN* – остановить фоновый процесс, если он пытается читать данные со своего управляющего терминала;

*SIGTTOU* – сигнал о попытке вывода на терминал фоновым процессом; *SIGURG* – сигнал о поступлении в буфер сокета срочных данных; *SIGUSR1 SIGUSR2* – определяется пользователем;

*SIGVTALRM* – сигнал виртуального таймера;

*SIGXCPU* – сигнал о превышении лимита процессорного времени;

*SIGXFSZ* – сигнал о превышении предела на размер файла.

Сигналы обеспечивают простой метод программного прерывания работы процессов.

**Процесс может выполнять три действия с сигналами, а именно:**

* **изменять свою реакцию на поступление определенного сигнала (изменять обработку сигналов);**
* **блокировать сигналы (откладывать их обработку) на время выполнения определенных критических участков кода;**
* **посылать сигналы другим процессам.**

Вопрос 2. Алгоритм обработки сигналов

Ядро обрабатывает сигналы в контексте того процесса, который получает их, поэтому, чтобы обработать сигналы, нужно запустить процесс.

При получении сигнала процесс может выполнить одно из трех действий:

* + выполнить действие по умолчанию;
  + игнорировать сигнал и продолжать выполнение;
  + выполнить определенное пользователем действие.

Реакцией по умолчанию со стороны процесса, исполняемого в режиме ядра, является вызов функции *exit*, однако с помощью функции *signal* процесс может указать другие специальные действия, принимаемые по получении тех или иных сигналов.

В пространстве процесса поддерживается массив полей для обработки сигналов. По одному полю на каждый определенный в системе сигнал. В поле, соответствующем сигналу с указанным номером, ядро сохраняет адрес пользовательской функции, вызываемой по получении сигнала процессом. Способ обработки сигналов одного типа не влияет на обработку сигналов других типов.

Обрабатывая сигнал, ядро определяет тип сигнала и очищает (гасит) разряд в записи таблицы процессов, соответствующий данному типу сигнала и установленный в момент получения сигнала процессом. Если функции обработки сигнала присвоено значение по умолчанию, ядро в отдельных случаях перед завершением процесса сбрасывает на внешний носитель (дампирует) образ процесса в памяти. Дампирование удобно для программистов тем, что позволяет установить причину завершения процесса и посредством этого вести отладку программ. Ядро дампирует состояние памяти при поступлении сигналов, которые сообщают о каких-нибудь ошибках в выполнении процессов, как, например, попытка исполнения запрещенной команды или обращение к адресу, находящемуся за пределами виртуального адресного пространства процесса. Ядро не дампирует состояние памяти, если сигнал не связан с программной ошибкой. Если процесс получает сигнал, на который было решено не обращать внимание, выполнение процесса продолжается так, словно сигнала и не было. Поскольку ядро не сбрасывает значение соответствующего поля, свидетельствующего о необходимости игнорирования сигнала данного типа, то когда сигнал поступит вновь, процесс опять не обратит на него внимание.

Если процесс получает сигнал, реагирование на который было признано необходимым, сразу по возвращении процесса в режим задачи выполняется заранее условленное действие, однако прежде чем перевести процесс в режим задачи, ядро еще должно предпринять следующие шаги:

* ядро обращается к сохраненному регистровому контексту задачи и выбирает значения счетчика команд и указателя вершины стека, которые будут возвращены пользовательскому процессу;
* сбрасывает в пространстве процесса прежнее значение поля функции обработки сигнала и присваивает ему значение по умолчанию;
* создает новую запись в стеке задачи, в которую при необходимости выделяя дополнительную память, переписывает значения счетчика команд и указателя вершины стека, выбранные ранее из сохраненного регистрового контекста задачи. Стек задачи будет выглядеть так, как будто процесс произвел обращение к пользовательской функции (обработка сигнала) в той точке, где он вызывал системную функцию или где ядро прервало его выполнение (перед опознанием сигнала);
* вносит изменения в сохраненный регистровый контекст задачи, устанавливает значение счетчика команд равным адресу функции обработки сигнала, а значение указателя вершины стека равным глубине стека задачи.

Эту ситуацию можно разобрать следующим образом. Процесс обращается к системной функции *signal* для того, чтобы дать указание принимать сигналы о прерываниях и исполнять по их получении функцию *sigcatcher*. Затем он порождает новый процесс, запускает системную функцию *nice*, позволяющую сделать приоритет запуска процесса-родителя ниже приоритета его потомка, и входит в бесконечный цикл. Порожденный процесс задерживает свое выполнение на 5 секунд, чтобы дать родительскому процессу время исполнить системную функцию nice и снизить свой приоритет. После этого порожденный процесс входит в цикл, в каждой итерации которого он посылает родительскому процессу сигнал о прерывании (посредством обращения к функции *kill*). Если в результате ошибки, например, из-за того, что родительский процесс больше не существует, *kill* завершается, то завершается и порожденный процесс.

Вся идея состоит в том, что родительскому процессу следует запускать функцию обработки сигнала при каждом получении сигнала о прерывании. Функция обработки сигнала выводит сообщение и снова обращается к функции *signal* при очередном появлении сигнала о прерывании, родительский же процесс продолжает исполнять циклический набор команд.

Однако возможна и следующая очередность наступления событий:

* порожденный процесс посылает родительскому процессу сигнал о прерывании;
* родительский процесс принимает сигнал и вызывает функцию обработки сигнала, но резервируется ядром, которое производит переключение контекста до того, как функция *signal* будет вызвана повторно;
* снова запускается порожденный процесс, который посылает родительскому процессу еще один сигнал о прерывании;
* родительский процесс получает второй сигнал о прерывании, но перед тем он не успел сделать никаких распоряжений относительно способа обработки сигнала. Когда выполнение родительского процесса будет возобновлено, он завершится.

Второе несоответствие в обработке сигналов связано с приемом сигналов, поступающих во время исполнения системной функции, когда процесс приостановлен с допускающим прерывания приоритетом. Сигнал побуждает процесс выйти из приостанова (с помощью *longjump*), вернуться в режим задачи и вызвать функцию обработки сигнала. Когда функция обработки сигнала завершает работу, происходит то, что процесс выходит из системной функции с ошибкой, сообщающей о прерывании ее выполнения. Узнав об ошибке, пользователь запускает системную функцию повторно, однако более удобно было бы, если бы это действие автоматически выполнялось ядром, как в системе BSD.

Третье несоответствие проявляется в том случае, когда процесс игнорирует поступивший сигнал. Если сигнал поступает в то время, когда процесс находится в состоянии приостанова с допускающим прерывания приоритетом, процесс возобновляется, но не выполняет *longjump*. Другими словами, ядро узнает о том, что процесс проигнорировал поступивший сигнал только после возобновления его выполнения. Логичнее было бы оставить процесс в состоянии приостанова. Однако, в момент посылки сигнала к пространству процесса, в котором ядро хранит адрес функции обработки сигнала, может отсутствовать доступ. Эта проблема может быть решена путем запоминания адреса функции обработки сигнала в записи таблицы процессов, обращаясь к которой, ядро получало бы возможность решать вопрос о необходимости возобновления процесса по получении сигнала.

С другой стороны, процесс может немедленно вернуться в состояние приостанова (по алгоритм *sleep*), если обнаружит, что в его возобновлении не было необходимости. Однако пользовательские процессы не имеют возможности осознавать собственное возобновление, поскольку ядро располагает точку входа в алгоритм *sleep* внутри цикла с условием продолжения, переводя процесс вновь в состояние приостанова, если ожидаемое процессом событие в действительности не имело места.

Ко всему сказанному выше следует добавить, что ядро обрабатывает сигналы типа «гибель потомка» не так, как другие сигналы. В частности, когда процесс узнает о получении сигнала «гибель потомка», он выключает индикацию сигнала в соответствующем поле записи таблицы процессов и по умолчанию действует так, словно никакого сигнала и не поступало. Назначение сигнала «гибель потомка» состоит в возобновлении выполнения процесса, приостановленного с допускающим прерывания приоритетом. Если процесс принимает такой сигнал, он, как и во всех остальных случаях, запускает функцию обработки сигнала. Действия, предпринимаемые ядром в том случае, когда процесс игнорирует поступивший сигнал этого типа, будут описаны позже. Наконец, когда процесс вызвал функцию *signal* с параметром «гибель потомка» (*death of child*), ядро посылает ему соответствующий сигнал, если он имеет потомков, прекративших существование.

Для посылки сигналов процессы используют системную функцию *kill*.

Во всех случаях, если процесс, пославший сигнал, исполняется под кодом идентификации пользователя, не являющегося суперпользователем, или если коды идентификации пользователя (реальный и исполнительный) у этого процесса не совпадают с соответствующими кодами процесса, принимающего сигнал, *kill* завершается неудачно.

Вопрос 3. Посылка и блокировка сигналов

Наборы сигналов являются одним из основных параметров, передаваемых работающим с сигналами системным вызовам. Они просто задают список сигналов, которые необходимо передать системному вызову.

Наборы сигналов определяются при помощи типа *sigset\_t*, который определен в заголовочном файле <*signal.h*>. Размер типа задан так, чтобы в нем мог поместиться весь набор определенных в системе сигналов.

Выбрать определенные сигналы можно, начав либо с полного набора сигналов и удалив ненужные сигналы, либо с пустого набора, включив в него нужные.

Инициализация пустого и полного набора сигналов выполняется при помощи процедур *sigemptyset* и *sigfillset* соответственно. После инициализации с набором сигналов можно оперировать при помощи процедур *sigaddset* и *sigdelset*, соответственно добавляющих и удаляющих указанные вами сигналы:

#*include* <*signal.h*>

/\*Инициализация\*/

*int sigemptyset* (*sigset\_t \*set*); *int sigfillset* (*sigset\_t \*set*);

/\*Добавление и удаление сигналов\*/ *int sigaddset* (*sigset\_t \*set, int signo*); *int sigdelset* (*sigset\_t \*set, int signo*);

Процедуры sigemptyset (*sigset\_t \*set*); и *sigfillset* (*sigset\_t \*set*); имеют единственный параметр – указатель на переменную типа *sigset\_t*.

Приложения должны вызывать *sigemptyset* (*sigset\_t \*set*); и *sigfillset* (*sigset\_t*

*\*set*); хотя бы один раз для каждой переменной типа sigset\_t.

Процедуры *sigaddset* (*sigset\_t \*set, int signo*); и *sigdelset* (*sigset\_t \*set, int signo*); принимают в качестве параметров указатель на инициализированный набор сигналов и номер сигнала, который должен быть добавлен или удален. Второй параметр, *signo*, может быть символическим именем константы, таким как *SIGINT*, или настоящим номером сигнала, но в этом случае программа окажется системно-зависимой.

После определения списка сигналов можно задать определенный метод обработки сигнала при помощи процедуры *sigaction*:

Описание

#*include* <*signal.h*>

*int sigaction* (*int signo, const struct sigaction \*act, struct sigaction \*oact*);

Первый параметр *signo* задает отдельный сигнал, для которого нужно определить действие. Чтобы это действие выполнялось, процедура *sigaction* должна быть вызвана до получения сигнала типа *signo*. Значением переменной *signo* может быть любое из ранее определенных имен сигналов, за исключением *SIGSTOP* и *SIGKILL*, которые предназначены только для остановки или завершения процесса и не могут обрабатываться по-другому.

Второй параметр, *act*, определяет обработчика сигнала *signo*.

Третий параметр, *oact*, если не равен *NULL*, указывает на структуру, куда будет помещено описание старого метода обработки сигнала.

*Первое* поле, *sa\_handler*, задает обработчик сигнала *signo*. Это поле может иметь три вида значений:

* *SIG\_DFL* – константа, сообщающая, что нужно восстановить обработку сигнала по умолчанию (для большинства сигналов это завершение процесса);
* *SIG\_IGN* – константа, сообщающая, что нужно игнорировать данный сигнал. Не может использоваться для сигналов *SIGSTOP* и *SIGKILL*;
* адрес функции, принимающей аргумент типа *int*. Если функция объявлена в тексте программы до заполнения *sigaction*, то полю *sa\_handler* можно просто присвоить имя функции. Компилятор поймет, что имелся в виду ее адрес. Эта функция будет выполняться при получении сигнала *signo*, а само значение *signo* будет передано в качестве аргумента вызываемой функции. Управление будет передано функции, как только процесс получит сигнал, какой бы участок программы при этом не выполнялся. После возврата из функции управление будет снова передано процессу и продолжится с точки, в которой выполнение процесса было прервано.

*Второе* поле, *sa\_mask*, демонстрирует первое практическое использование набора сигналов. Сигналы, заданные в поле *sa\_mask*, будут блокироваться во время выполнения функции, заданной полем *sa\_handler*. Это не означает, что эти сигналы будут игнорироваться, просто их обработка будет отложена до завершения функции. При входе в функцию перехваченных сигнал также будет неявно добавлен к текущей маске сигналов. Все это делает механизм сигналов более надежным.

Поле *sa\_flags* может использоваться для изменения характера реакции на сигнал *signo*. Например, после возврата из обработчика можно вернуть обработчик по умолчанию *SIG\_DFL*, установив значение поля *sa\_flags* равным *SA\_RESETHAND*. Если же значение поля *sa\_flags* установлено равным *SA\_SIGINFO*, то обработчику сигнала будет передаваться дополнительная информация. В этом случае поле *sa\_handler* является избыточным и используется поле *sa\_sigaction*.

Передаваемая этому обработчику структура *siginfo\_t* содержит дополнительную информацию о сигнале, такую как:

* его номер;
* идентификатор пославшего сигнал процесса и действующий идентификатор пользователя этого процесса.