Лекция 4. Управление процессами и нитями, ввод/вывод в ОС UNIX. Прерывания и особые ситуации.

4.1 Среда выполнения процессов в ОС UNIX
4.2 Пользовательская и ядерная составляющие процессов. Принципы организации многопользовательского режима
4.3 Традиционный механизм управления процессами на уровне пользователя. Понятие нити (threads).
4.4 Принципы системной буферизации ввода/вывода
4.5 Прерывания и особые ситуации
4.6 Перспективные ОС, поддерживающие среду ОС UNIX

4.1 Среда выполнения процессов в ОС UNIX

Порождение процессов

Порождение процессов в системе UNIX происходит следующим образом. При создании процесса строится образ порожденного процесса, являющийся точной копией образа породившего процесса. Сегмент данных и сегмент стека отца действительно копируются на новое место, образуя сегменты данных и стека сына. Процедурный сегмент копируется только тогда, когда он не является разделяемым. Наследуются все характеристики процесса, содержащиеся в контексте.

Таким образом, в UNIX порождение нового процесса происходит в два этапа

1. сначала создается копия процесса-родителя, то есть дублируется дескриптор, контекст и образ процесса.
2. Затем у нового процесса производится замена кодового сегмента на заданный.

Вновь созданному процессу операционная система присваивает целочисленный идентификатор, уникальный за весь период функционирования системы.

В операционной системе UNIX традиционно поддерживается классическая схема мультипрограммирования. Система поддерживает возможность параллельного (или квази-параллельного в случае наличия только одного аппаратного процессора) выполнения нескольких пользовательских программ. Каждому такому выполнению соответствует процесс операционной системы. Каждый процесс выполняется в собственной виртуальной памяти, и, тем самым, процессы защищены один от другого, т.е. один процесс не в состоянии неконтролируемым образом прочитать что-либо из памяти другого процесса или записать в нее. Однако контролируемые взаимодействия процессов допускаются системой, в том числе за счет возможности разделения одного сегмента памяти между виртуальной памятью нескольких процессов.

Не менее важно защищать саму операционную систему от возможности ее повреждения каким бы то ни было пользовательским процессом. В ОС UNIX это достигается за счет того, что ядро системы работает в собственном "ядерном" виртуальном пространстве, к которому не может иметь доступа ни один пользовательский процесс.

Ядро системы предоставляет возможности (набор системных вызовов) для порождения новых процессов, отслеживания окончания порожденных процессов и т.д. С другой стороны, в ОС UNIX ядро системы - это полностью пассивный набор программ и данных. Любая программа ядра может начать работать только по инициативе некоторого пользовательского процесса (при выполнении системного вызова), либо по причине внутреннего или внешнего прерывания (примером внутреннего прерывания может быть прерывание из-за отсутствия в основной памяти требуемой страницы виртуальной памяти пользовательского процесса; примером внешнего прерывания является любое прерывание процессора по инициативе внешнего устройства). В любом случае считается, что выполняется ядерная часть обратившегося или прерванного процесса, т.е. ядро всегда работает в контексте некоторого процесса.

В последние годы в связи с широким распространением так называемых симметричных мультипроцессорных архитектур компьютеров (Symmetric Multiprocessor Architectures - SMP) в ОС UNIX был внедрен механизм легковесных процессов (light-weight processes), или нитей, или потоков управления (threads). Говоря по-простому, нить - это процесс, выполняющийся в

виртуальной памяти, используемой совместно с другими нитями того же "тяжеловесного" (т.е. обладающего отдельной виртуальной памятью) процесса.

Планирование процессов

В системе UNIX System V Release 4 реализована ***вытесняющая многозадачность, основанная на использовании приоритетов и квантования***.

Все процессы разбиты на несколько групп, называемых классами приоритетов. Каждая группа имеет свои характеристики планирования процессов.

Созданный процесс наследует характеристики планирования процесса-родителя, которые включают класс приоритета и величину приоритета в этом классе. Процесс остается в данном классе до тех пор, пока не будет выполнен системный вызов, изменяющий его класс.

В UNIX System V Release 4 возможно включение новых классов приоритетов при инсталляции системы. В настоящее время имеется три приоритетных класса:

- класс реального времени,
- класс системных процессов
- класс процессов разделения времени.

В отличие от ранних версий UNIX приоритетность (привилегии) процесса тем выше, чем больше число, выражающее приоритет. Значения приоритетов определяются для разных классов по разному. Процессы системного класса используют стратегию фиксированных приоритетов. Системный класс зарезервирован для процессов ядра. Уровень приоритета процессу назначается ядром и никогда не изменяется. Заметим, что пользовательский процесс, перешедший в системную фазу, не переходит при этом в системный класс приоритетов.

Процессы реального времени также используют стратегию фиксированных приоритетов, но пользователь может их изменять. Так как при наличии готовых к выполнению процессов реального времени другие процессы не рассматриваются, то процессы реального времени надо тщательно проектировать, чтобы они не захватывали процессор на слишком долгое время. Характеристики планирования процессов реального времени включают две величины: уровень глобального приоритета и квант времени. Для каждого уровня приоритета имеется по умолчанию своя величина кванта времени. Процессу разрешается захватывать процессор на указанный квант времени, а по его истечении планировщик снимает процесс с выполнения.

Подсистема управления процессами

В UNIX System V Release 4 реализован механизм виртуальной файловой системы VFS (Virtual File System), который позволяет ядру системы одновременно поддерживать несколько различных типов файловых систем. Механизм VFS поддерживает для ядра некоторое абстрактное представление о файловой системе, скрывая от него конкретные особенности каждой файловой системы.

Типы файловых систем, поддерживаемых в UNIX System V Release 4:

- **s5** - традиционная файловая система UNIX System V, поддерживаемая в ранних версиях UNIX System V от AT&T;
- **ufs** - файловая система, используемая по умолчанию в UNIX System V Release 4, которая ведет происхождение от файловой системы SunOS, которая в свою очередь, происходит от файловой системы Berkeley Fast File System (FFS);
- **nfs** - адаптация известной файловой системы NFS фирмы Sun Microsystems, которая позволяет разделять файлы и каталоги в гетерогенных сетях;
- **rfs** - файловая система Remote File Sharing из UNIX System V Release 3. По функциональным возможностям близка к NFS, но требует на каждом компьютере установки UNIX System V Release 3 или более поздних версий этой ОС;
- **Veritas** - отказоустойчивая файловая система с транзакционным механизмом операций;
- **specfs** - этот новый тип файловой системы обеспечивает единый интерфейс ко всем специальным файлам, описываемым в каталоге /dev;
- **fifofs** - эта новая файловая система использует механизм VFS для реализации файлов FIFO, также известных как конвейеры (pipes), в среде STREAMS;

- **bfs** - загрузочная файловая система. Предназначена для быстрой и простой загрузки и поэтому представляет собой очень простую плоскую файловую систему, состоящую из одного каталога;
- **/proc** - файловая система этого типа обеспечивает доступ к образу адресного пространства каждого активного процесса системы, обычно используется для отладки и трассировки;
- **/dev/fd** - этот тип файловой системы обеспечивает удобный метод ссылок на дескрипторы открытых файлов.

Не во всех коммерческих реализациях поддерживаются все эти файловые системы, отдельные производители могут предоставлять только некоторые из них.

Программой называется исполняемый файл, а процессом называется последовательность операций программы или часть программы при ее выполнении. В системе UNIX может одновременно выполняться множество процессов, при чем их число логически не ограничивается. Различные системные операции позволяют процессам порождать новые процессы, завершают процессы, синхронизируют выполнение этапов процесса и управляют реакцией на наступление различных событий. Благодаря различным обращениям к операционной системе, процессы выполняются независимо друг от друга.

Элементы конструкционных блоков

Как уже говорилось ранее, концепция разработки системы UNIX заключалась в построении операционной системы из элементов, которые позволили бы пользователю создавать небольшие программные модули, выступающие в качестве конструкционных блоков при создании более сложных программ.

Одним из таких элементов, с которым часто сталкиваются пользователи при работе с командным процессором shell, ***является возможность переназначения ввода-вывода***. Говоря условно, процессы имеют доступ к трем файлам: они читают из файла стандартного ввода, записывают в файл стандартного вывода и выводят сообщения об ошибках в стандартный файл ошибок. Процессы, запускаемые с терминала, обычно используют терминал вместо всех этих трех файлов, однако каждый файл независимо от других может быть "переназначен".

Вторым конструкционным элементом является канал, механизм, обеспечивающий информационный обмен между процессами, выполнение которых связано с операциями чтения и записи. Процессы могут переназначать выводной поток со стандартного вывода на канал для чтения с него другими процессами, переназначившими на канал свой стандартный ввод. Данные, посылаемые в канал первыми процессами, являются входными для вторых процессов. Вторые процессы так же могут переназначить свой выводной поток и так далее, в зависимости от пожеланий программиста. И снова, так же как и в вышеуказанном случае, процессам нет необходимости знать, какого типа файл используется в качестве файла стандартного вывода; их выполнение не зависит от того, будет ли файлом стандартного вывода обычный файл, канал или устройство. В процессе построения больших и сложных программ из конструкционных элементов меньшего размера программисты часто используют каналы и переназначение ввода-вывода при сборке и соединении отдельных частей. И действительно, такой стиль программирования находит поддержку в системе, благодаря чему новые программы могут работать вместе с существующими программами.

4.2 Пользовательская и ядерная составляющие процессов. Принципы организации многопользовательского режима

Пользовательская и ядерная составляющие процессов.

Каждому процессу соответствует контекст, в котором он выполняется. Этот контекст включает:

1. содержимое пользовательского адресного пространства - ***пользовательский контекст*** (т.е. содержимое сегментов программного кода, данных, стека, разделяемых сегментов и сегментов файлов, отображаемых в виртуальную память),

2. содержимое аппаратных регистров - ***регистровый контекст*** (регистр счетчика команд, регистр состояния процессора, регистр указателя стека и регистры общего назначения),

3. а также структуры данных ядра (***контекст системного уровня***), связанные с этим процессом.

Контекст процесса системного уровня в ОС UNIX состоит из "***статической***" и "***динамических***" частей. Для каждого процесса имеется одна статическая часть контекста системного уровня и переменное число динамических частей.

Статическая часть контекста процесса системного уровня включает следующее:

1. Описатель процесса, т.е. элемент таблицы описателей существующих в системе процессов. Описатель процесса включает, в частности, следующую информацию:
   - состояние процесса;
   - физический адрес в основной или внешней памяти u-области процесса;
   - идентификаторы пользователя, от имени которого запущен процесс;
   - идентификатор процесса;
   - прочую информацию, связанную с управлением процессом.
2. U-область (u-area) - индивидуальная для каждого процесса область пространства ядра, обладающая тем свойством, что хотя u-область каждого процесса располагается в отдельном месте физической памяти, u-области всех процессов имеют один и тот же виртуальный адрес в адресном пространстве ядра. Именно это означает, что какая бы программа ядра не выполнялась, она всегда выполняется как ядерная часть некоторого пользовательского процесса, и именно того процесса, u-область которого является "видимой" для ядра в данный момент времени. U-область процесса содержит:
   - указатель на описатель процесса;
   - идентификаторы пользователя;
   - счетчик времени, в течение которого процесс реально выполнялся (т.е. занимал процессор) в режиме пользователя и режиме ядра;
   - параметры системного вызова;
   - результаты системного вызова;
   - таблица дескрипторов открытых файлов;
   - предельные размеры адресного пространства процесса;
   - предельные размеры файла, в который процесс может писать;
   - и т.д.

Динамическая часть контекста процесса - это один или несколько стеков, которые используются процессом при его выполнении в режиме ядра. Число ядерных стеков процесса соответствует числу уровней прерывания, поддерживаемых конкретной аппаратурой.

Принципы организации многопользовательского режима.

Основной проблемой организации многопользовательского (правильнее сказать, мультипрограммного) режима в любой операционной системе является организация планирования "параллельного" выполнения нескольких процессов. Исторически ОС UNIX является системой разделения времени, т.е. система должна прежде всего "справедливо" разделять ресурсы процессора(ов) между процессами, относящимися к разным пользователям, причем таким образом, чтобы время реакции каждого действия интерактивного пользователя находилось в допустимых пределах.

Наиболее распространенным алгоритмом планирования в системах разделения времени является кольцевой режим (round robin). Основной смысл алгоритма состоит в том, что время процессора делится на кванты фиксированного размера, а процессы, готовые к выполнению, выстраиваются в кольцевую очередь. У этой очереди имеются два указателя - начала и конца. Когда процесс, выполняющийся на процессоре, исчерпывает свой квант процессорного времени, он снимается с процессора, ставится в конец очереди, а ресурсы процессора отдаются процессу, находящемуся в начале очереди. Если выполняющийся на процессоре процесс откладывается (например, по причине обмена с некоторым внешним устройством) до того, как он исчерпает свой квант, то после повторной активизации он становится в конец очереди (не смог доработать - не

вина системы). Это прекрасная схема разделения времени в случае, когда все процессы одновременно помещаются в оперативной памяти.

Однако ОС UNIX всегда была рассчитана на то, чтобы обслуживать больше процессов, чем можно одновременно разместить в основной памяти Поэтому требовалась несколько более гибкая схема планирования разделения ресурсов процессора(ов). В результате было введено понятие приоритета. В ОС UNIX значение приоритета определяет, во-первых, возможность процесса пребывать в основной памяти и на равных конкурировать за процессор. Во-вторых, от значения приоритета процесса, вообще говоря, зависит размер временного кванта, который предоставляется процессу для работы на процессоре при достижении своей очереди. В-третьих, значение приоритета, влияет на место процесса в общей очереди процессов к ресурсу процессора(ов).

Схема разделения времени между процессами с приоритетами в общем случае выглядит следующим образом. Готовые к выполнению процессы выстраиваются в очередь к процессору в порядке уменьшения своих приоритетов. Если некоторый процесс отработал свой квант процессорного времени, но при этом остался готовым к выполнению, то он становится в очередь к процессору впереди любого процесса с более низким приоритетом, но вслед за любым процессом, обладающим тем же приоритетом. Если некоторый процесс активизируется, то он также ставится в очередь вслед за процессом, обладающим тем же приоритетом. Весь вопрос в том, когда принимать решение о своппинге процесса, и когда возвращать в оперативную память процесс, содержимое памяти которого было ранее перемещено во внешнюю память.

Традиционное решение ОС UNIX состоит в использовании динамически изменяющихся приоритетов. Каждый процесс при своем образовании получает некоторый устанавливаемый системой статический приоритет, который в дальнейшем может быть изменен с помощью системного вызова nice. Этот статический приоритет является основой начального значения динамического приоритета процесса, являющегося реальным критерием планирования. Все процессы с динамическим приоритетом не ниже порогового участвуют в конкуренции за процессор. Однако каждый раз, когда процесс успешно отрабатывает свой квант на процессоре, его динамический приоритет уменьшается (величина уменьшения зависит от статического приоритета процесса). Если значение динамического приоритета процесса достигает некоторого нижнего предела, он становится кандидатом на откачку (своппинг) и больше не конкурирует за процессор.

Процесс, образ памяти которого перемещен во внешнюю память, также обладает динамическим приоритетом. Этот приоритет не дает процессу право конкурировать за процессор (да это и невозможно, поскольку образ памяти процесса не находится в основной памяти), но он изменяется, давая в конце концов процессу возможность вновь вернуться в основную память и принять участие в конкуренции за процессор. Правила изменения динамического приоритета для процесса, перемещенного во внешнюю память, в принципе, очень просты. Чем дольше образ процесса находится во внешней памяти, тем более высок его динамический приоритет (конкретное значение динамического приоритета, конечно, зависит от его статического приоритета). Конечно, раньше или позже значение динамического приоритета такого процесса перешагнет через некоторый порог, и тогда система принимает решение о необходимости возврата образа процесса в основную память. После того, как в результате своппинга будет освобождена достаточная по размерам область основной памяти, процесс с приоритетом, достигшим критического значения, будет перемещен в основную память и будет в соответствии со своим приоритетом конкурировать за процессор.

4.3 Традиционный механизм управления процессами на уровне пользователя. Понятие нити (threads).

Традиционный механизм управления процессами на уровне пользователя.

Как свойственно операционной системе UNIX вообще, имеются две возможности управления процессами - с использованием командного языка (того или другого варианта Shell) и с использованием языка программирования с непосредственным использованием системных вызовов ядра операционной системы.

Общая схема возможностей пользователя, связанных с управлением процессами: Каждый процесс может образовать полностью идентичный подчиненный процесс с помощью системного вызова fork() и дожидаться окончания выполнения своих подчиненных процессов с помощью системного вызова wait. Каждый процесс в любой момент времени может полностью изменить содержимое своего образа памяти с помощью одной из разновидностей системного вызова exec (сменить образ памяти в соответствии с содержимым указанного файла, хранящего образ процесса (выполняемого файла)). Каждый процесс может установить свою собственную реакцию на "сигналы", производимые операционной системой в соответствие с внешними или внутренними событиями. Наконец, каждый процесс может повлиять на значение своего статического (а тем самым и динамического) приоритета с помощью системного вызова nice.

Для создания нового процесса используется системный вызов fork. В среде программирования нужно относиться к этому системному вызову как к вызову функции, возвращающей целое значение - идентификатор порожденного процесса, который затем может использоваться для управления (в ограниченном смысле) порожденным процессом. Реально, все процессы системы UNIX, кроме начального, запускаемого при раскрутке системы, образуются при помощи системного вызова fork.

Понятие нити (threads).

"Нить" (thread) - это независимый поток управления, выполняемый в контексте некоторого процесса. Фактически, понятие контекста процесса изменяется следующим образом. Все, что не относится к потоку управления (виртуальная память, дескрипторы открытых файлов и т.д.), остается в общем контексте процесса. Вещи, которые характерны для потока управления (регистровый контекст, стеки разного уровня и т.д.), переходят из контекста процесса в контекст нити.

Все нити процесса выполняются в его контексте, но каждая нить имеет свой собственный контекст. Контекст нити, как и контекст процесса, состоит из пользовательской и ядерной составляющих. Пользовательская составляющая контекста нити включает индивидуальный стек нити. Поскольку нити одного процесса выполняются в общей виртуальной памяти (все нити процесса имеют равные права доступа к любым частям виртуальной памяти процесса), стек (сегмент стека) любой нити процесса в принципе не защищен от произвольного (например, по причине ошибки) доступа со стороны других нитей. Ядерная составляющая контекста нити включает ее регистровый контекст (в частности, содержимое регистра счетчика команд) и динамически создаваемые ядерные стеки.

Хотя концептуально реализации механизма нитей в разных современных вариантах практически эквивалентны технически и, к сожалению, в отношении интерфейсов эти реализации различаются.

Хотя концептуально реализации механизма нитей в разных современных вариантах практически эквивалентны (да и что особенное можно придумать по поводу легковесных процессов?), технически и, к сожалению, в отношении интерфейсов эти реализации различаются. Мы не ставим здесь перед собой цели описать в деталях какую-либо реализацию, однако постараемся в общих чертах охарактеризовать разные подходы.

Начнем с того, что разнообразие механизмов нитей в современных вариантах ОС UNIX само по себе представляет проблему. Сейчас достаточно трудно говорить о возможности мобильного параллельного программирования в среде UNIX-ориентированных операционных систем. Если программист хочет добиться предельной эффективности (а он должен этого хотеть, если для целей его проекта приобретен дорогостоящий мультипроцессор), то он вынужден использовать все уникальные возможности используемой им операционной системы.

Для всех очевидно, что сегодняшняя ситуация далека от идеальной. Однако, по-видимому, ее было невозможно избежать, поскольку поставщики мультипроцессорных симметричных архитектур должны были как можно раньше предоставить своим покупателям возможности эффективного программирования, и времени на согласование решений просто не было (любых поставщиков прежде всего интересует объем продаж, а проблемы будущего оставляются на будущее).

Применяемые в настоящее время подходы зависят от того, насколько внимательно разработчики ОС относились к проблемам реального времени. (Возвращаясь к введению этого

раздела, еще раз отметим, что здесь мы имеем в виду "мягкое" реальное время, т.е. программно-аппаратные системы, которые обеспечивают быструю реакцию на внешние события, но время реакции не установлено абсолютно строго.) Типичная система реального времени состоит из общего монитора, который отслеживает общее состояние системы и реагирует на внешние и внутренние события, и совокупности обработчиков событий, которые, желательно параллельно, выполняют основные функции системы.

Понятно, что от возможностей реального распараллеливания функций обработчиков зависят общие временные показатели системы. Если, например, при проектировании системы замечено, что типичной картиной является "одновременное" поступление в систему N внешних событий, то желательно гарантировать наличие реальных N устройств обработки, на которых могут базироваться обработчики. На этих наблюдениях основан подход компании Sun Microsystems.

4.4 Принципы системной буферизации ввода/вывода

Традиционно в ОС UNIX выделяются три типа организации ввода/вывода и, соответственно, три типа драйверов.

Блочный ввод/вывод главным образом предназначен для работы с каталогами и обычными файлами файловой системы, которые на базовом уровне имеют блочную структуру. Блочный ввод/вывод, кроме того, поддерживается системной.

Символьный ввод/вывод служит для прямого (без буферизации) выполнения обменов между адресным пространством пользователя и соответствующим устройством. Общей для всех символьных драйверов поддержкой ядра является обеспечение функций пересылки данных между пользовательскими и ядерным адресными пространствами.

Потоковый ввод/вывод похож на символьный ввод/вывод, но по причине наличия возможности включения в поток промежуточных обрабатывающих модулей обладает существенно большей гибкостью.

Традиционным способом снижения накладных расходов при выполнении обменов с устройствами внешней памяти, имеющими блочную структуру, является буферизация блочного ввода/вывода. Это означает, что любой блок устройства внешней памяти считывается прежде всего в некоторый буфер области основной памяти, называемой в ОС UNIX системным кэшем, и уже оттуда полностью или частично (в зависимости от вида обмена) копируется в соответствующее пользовательское пространство.

Принципами организации традиционного механизма буферизации является, во-первых, то, что копия содержимого блока удерживается в системном буфере до тех пор, пока не возникнет необходимость ее замещения по причине нехватки буферов. Во-вторых, при выполнении записи любого блока устройства внешней памяти реально выполняется лишь обновление (или образование и наполнение) буфера кэша. Действительный обмен с устройством выполняется либо при выталкивании буфера вследствие замещения его содержимого, либо при выполнении специального системного вызова sync (или fsync), поддерживаемого специально для насильственного выталкивания во внешнюю память обновленных буферов кэша.

Эта традиционная схема буферизации вошла в противоречие с развитыми в современных вариантах ОС UNIX средствами управления виртуальной памятью и в особенности с механизмом отображения файлов в сегменты виртуальной памяти Поэтому в System V Release 4 появилась новая схема буферизации, пока используемая параллельно со старой схемой. Суть новой схемы состоит в том, что на уровне ядра фактически воспроизводится механизм отображения файлов в сегменты виртуальной памяти. Новая схема буферизации в ядре ОС UNIX главным образом основывается на том, что для организации буферизации можно не делать почти ничего специального. Когда один из пользовательских процессов открывает не открытый до этого времени файл, ядро образует новый сегмент и подключает к этому сегменту открываемый файл. После этого (независимо от того, будет ли пользовательский процесс работать с файлом в традиционном режиме с использованием системных вызовов read и write или подключит файл к сегменту своей виртуальной памяти) на уровне ядра работа будет производиться с тем ядерным сегментом, к которому подключен файл на уровне ядра. Основная идея нового подхода состоит в

том, что устраняется разрыв между управлением виртуальной памятью и общесистемной буферизацией.

Семафоры.

Семафор может рассматриваться как целочисленная переменная, которая принимает только неотрицательные значения.

Так, процесс, требующий защищенного семафором ресурса, вынужден ожидать до тех пор, пока семафор не станет доступным, что свидетельствует об освобождении ожидаемого ресурса, и, захватив ресурс, установить семафор. В свою очередь, другие процессы также будут ожидать доступа к ресурсу вплоть до того момента, когда семафор возвратит соответствующий ресурс системе распределения ресурсов.

Семафор в ОС UNIX состоит из следующих элементов:

- значение семафора;
- идентификатор процесса, который хронологически последним работал с семафором;
- число процессов, ожидающих увеличения значения семафора;
- число процессов, ожидающих нулевого значения семафора.

Для работы с семафорами поддерживаются три системных вызова:

- **semget** для создания и получения доступа к набору семафоров;
- **semop** для манипулирования значениями семафоров (это именно тот системный вызов, который позволяет процессам синхронизоваться на основе использования семафоров);
- **semctl** для выполнения разнообразных управляющих операций над набором семафоров.

Основным поводом для введения массовых операций над семафорами было стремление дать программистам возможность избегать тупиковых ситуаций в связи с семафорной синхронизацией. Это обеспечивается тем, что системный вызов semop, каким бы длинным он не был, выполняется как атомарная операция, т.е. во время выполнения semop ни один другой процесс не может изменить значение какого-либо семафора.

Очереди сообщений.

Для обеспечения возможности обмена сообщениями между процессами этот механизм поддерживается следующими системными вызовами:

- **msgget** для образования новой очереди сообщений или получения дескриптора существующей очереди;
- **msgsnd** для посылки сообщения (вернее, для его постановки в указанную очередь сообщений);
- **msgrcv** для приема сообщения (вернее, для выборки сообщения из очереди сообщений);
- **msgctl** для выполнения ряда управляющих действий.

Как обычно, при выполнении системного вызова msgget ядро ОС UNIX либо создает новую очередь сообщений, помещая ее заголовок в таблицу очередей сообщений и возвращая пользователю дескриптор вновь созданной очереди, либо находит элемент таблицы очередей сообщений, содержащий указанный ключ, и возвращает соответствующий дескриптор очереди.

Для того чтобы ядро успешно поставило указанное сообщение в указанную очередь сообщений, должны быть выполнены следующие условия:

- обращающийся процесс должен иметь соответствующие права по записи в данную очередь сообщений;
- длина сообщения не должна превосходить установленный в системе верхний предел;
- общая длина сообщений (включая вновь посылаемое) не должна превосходить установленный предел;
- указанный в сообщении тип сообщения должен быть положительным целым числом.

В этом случае обратившийся процесс успешно продолжает свое выполнение, оставив отправленное сообщение в буфере очереди сообщений. Тогда ядро активизирует (пробуждает) все процессы, ожидающие поступления сообщений из данной очереди.

Если же оказывается, что новое сообщение невозможно буферизовать в ядре по причине превышения верхнего предела суммарной длины сообщений, находящихся в одной очереди сообщений, то обратившийся процесс откладывается (усыпляется) до тех пор, пока очередь сообщений не разгрузится процессами, ожидающими получения сообщений.

4.5 Прерывания и особые ситуации

Система UNIX позволяет таким устройства, как внешние устройства ввода-вывода и системные часы, асинхронно прерывать работу центрального процессора. По получении сигнала прерывания ядро операционной системы сохраняет свой текущий контекст (застывший образ выполняемого процесса), устанавливает причину прерывания и обрабатывает прерывание. После того, как прерывание будет обработано ядром, прерванный контекст восстановится и работа продолжится так, как будто ничего не случилось. Устройствам обычно приписываются приоритеты в соответствии с очередностью обработки прерываний. В процессе обработки прерываний ядро учитывает их приоритеты и блокирует обслуживание прерывания с низким приоритетом на время обработки прерывания с более высоким приоритетом.

Особые ситуации связаны с возникновением незапланированных событий, вызванных процессом, таких как недопустимая адресация, задание привилегированных команд, деление на ноль и т.д. Они отличаются от прерываний, которые вызываются событиями, внешними по отношению к процессу. Особые ситуации возникают прямо "посредине" выполнения команды, и система, обработав особую ситуацию, пытается перезапустить команду; считается, что прерывания возникают между выполнением двух команд, при этом система после обработки прерывания продолжает выполнение процесса уже начиная со следующей команды. Для обработки прерываний и особых ситуаций в системе UNIX используется один и тот же механизм.

Уровни прерывания процессора

Ядро иногда обязано предупреждать возникновение прерываний во время критических действий, могущих в случае прерывания запортить информацию. Например, во время обработки списка с указателями возникновение прерывания от диска для ядра нежелательно, т.к. при обработке прерывания можно запортить указатели. Обычно имеется ряд привилегированных команд, устанавливающих уровень прерывания процессора в слове состояния процессора. Установка уровня прерывания на определенное значение отсекает прерывания этого и более низких уровней, разрешая обработку только прерываний с более высоким приоритетом. На Рисунке 4 показана последовательность уровней прерывания. Если ядро игнорирует прерывания от диска, в этом случае игнорируются и все остальные прерывания, кроме прерываний от часов и машинных сбоев.

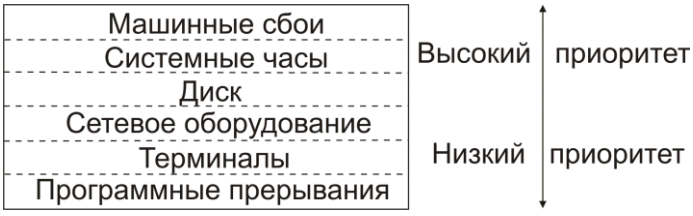

Рисунок 1 - Стандартные уровни прерываний

4.6 Перспективные ОС, поддерживающие среду ОС UNIX

Микроядро - это минимальная стержневая часть операционной системы, служащая основой модульных и переносимых расширений. По-видимому, большинство операционных систем следующего поколения будут обладать микроядрами.

В широкий обиход понятие микроядра ввела компания Next, в операционной системе которой использовалось микроядро Mach. Небольшое привилегированное ядро этой ОС, вокруг которого располагались подсистемы, выполняемые в режиме пользователя, теоретически должно было обеспечить небывалую гибкость и модульность системы. Но на практике это преимущество было несколько обесценено наличием монолитного сервера, реализующего операционную систему UNIX BSD 4.3, которую компания Next выбрала в качестве оболочки микроядра Mach.

Следующей микроядерной операционной системой была Windows NT компании Microsoft, в которой ключевым преимуществом использования микроядра должна была стать не только модульность, но и переносимость. ОС NT была построена таким образом, чтобы ее можно было применять в одно- и мультипроцессорных системах, основанных на процессорах Intel, Mips и Alpha. Поскольку в среде NT должны были выполняться программы, написанные для DOS, Windows, OS/2 и систем, совместимых со стандартами Posix, компания Microsoft использовала присущую микроядерному подходу модульность для создания общей структуры NT, не повторяющей ни одну из существующих операционных систем. Каждая операционная система эмулируется в виде отдельного модуля или подсистемы.

Позднее микроядерные архитектуры операционных систем были объявлены компаниями Novell/USL, Open Software Foundation (OSF), IBM, Apple и другими. Одним из основных конкурентов NT в области микроядерных ОС является Mach 3.0, система, созданная в университете Карнеги-Меллон, которую как IBM, так и OSF взялись довести до коммерческого вида

Очевидна тенденция к переходу от монолитных к микроядерным системам. Это совсем не новость для компаний QNX Software Systems и Unisys, которые уже в течение нескольких лет выпускают пользующиеся успехом микроядерные операционные системы. ОС QNX пользуется спросом на рынке систем реального времени, а CTOS фирмы Unisys популярна в области банковского дела. В обеих системах успешно использована модульность, присущая микроядерным ОС.

Средства графического интерфейса пользователей.

Во всех современных вариантах ОС UNIX поддерживаются графические интерфейсы пользователя с системой, а пользователям предоставляются инструментальные средства для разработки графических интерфейсов с разрабатываемыми ими программами. С точки зрения конечного пользователя средства графического интерфейса, поддерживаемого в разных вариантах ОС UNIX, да и в других системах (например, MS Windows или Windows NT), примерно одинаковы по своему стилю.

Во-первых, во всех случаях поддерживается многооконный режим работы с экраном терминала. В любой момент времени пользователь может образовать новое окно и связать его с нужной программой, которая работает с этим окном как с отдельным терминалом. Окна можно перемещать, изменять их размер, временно закрывать и т.д.

Во-вторых, во всех современных разновидностях графического интерфейса поддерживается управление мышью.

В-третьих, такое распространение "мышиного" стиля работы оказывается возможным за счет использования интерфейсных средств, основанных на пиктограммах (icons) и меню. В большинстве случаев, программа, работающая в некотором окне, предлагает пользователю выбрать какую-либо выполняемую ей функцию либо путем отображения в окне набора символических образов возможных функций (пиктограмм), либо посредством предложения многоуровневого меню. В любом случае для дальнейшего выбора оказывается достаточным управления курсором соответствующего окна с помощью мыши.

Наконец, современные графические интерфейсы обладают "дружественностью по отношению к пользователю", обеспечивая возможность немедленного получения интерактивной подсказки по любому поводу.

Оконная система X как базовое средство графических интерфейсов в среде ОС UNIX.

Для нормальной организации работы пользовательских программ с графическими терминалами требуется наличие некоторого базового слоя программного обеспечения,

скрывающего аппаратные особенности терминала; обеспечивающего создание окон на экране терминала, управление этими окнами и работу с ними со стороны пользовательской программы; дающего возможность пользовательской программе реагировать на события, происходящие в соответствующем окне (ввод с клавиатуры, движение курсора, нажатие клавиш мыши и т.д.). Такой базовый слой графического программного обеспечения принято называть оконной системой.

На этих идеях построена и оконная система X. На стороне пользовательского терминала находится сервер системы X, обеспечивающий единообразное управление графическим терминалом вне зависимости от его специфических аппаратных характеристик. В других компьютерах сети (которые, фактически, являются серверами с точки зрения организации вычислительного процесса) установлены клиентские части системы X, создающие впечатление у выполняемой программы, что она взаимодействует с локальным терминалом, а на самом деле поддерживающие точно специфицированный протокол взаимодействий с сервером системы X.

Клиентская и серверная части оконной системы X, хотя в целом соответствуют идеологии архитектуры "клиент-сервер", обладают тем своеобразием, что серверная часть системы находится вблизи пользователя (т.е. основного клиента вычислительной сети), а клиентская часть системы базируется на мощных серверах сети. В зависимости от конфигурации системы X-сервер может обслуживать один или несколько графических экранов, клавиатуру и мышь, реально представляя собой процесс, группу процессов или выделенное компьютерное устройство (X-терминал).

Для обеспечения требуемой гибкости, взаимодействия клиентской и пользовательской частей системы X по мере возможностей не должны были зависеть от используемых сетевой среды передачи данных и сетевых протоколов.

Одним из клиентов оконной системы обычно является так называемый "оконный менеджер" (window manager). Это специально выделенный клиент оконной системы, обладающий полномочиями на управление расположением окон на экране терминала. Некоторые из возможностей X-протокола (связанные, например, с перемещением окон) доступны только клиентам с полномочиями оконного менеджера. Во всем остальном оконный менеджер является обычным клиентом.

Системы, основанные на System V Release 4.

На базе System V возникло много коммерческих компаний. Их продукты мы кратко рассмотрим.

1 Solaris компании Sun Microsystems.

В течение многих лет основой операционных систем (SunOS) компании Sun являлся UNIX BSD. Однако, начиная с SunOS 4.0, произошел полный переход на System V 4.0. Sun Microsystems внесла ряд существенных расширений в SVR 4.0. Прежде всего это касается обеспечения распараллеливания программ при использовании симметричных мультипроцессорных компьютеров (механизм потоков управления - threads).

Solaris является внешней оболочкой SunOS и дополнительно включает средства графического пользовательского интерфейса и высокоуровневые средства сетевого взаимодействия (в частности, средства вызова удаленных процедур - RPC).

2 HP/UX компании Hewlett-Packard, DG/UX компании Data General, AIX компании IBM.

HP/UX, DG/UX и AIX обладают многими отличиями. В частности, в этих версиях ОС UNIX поддерживаются разные средства генерации графических пользовательских интерфейсов (хотя все они основаны на использовании оконной системы X), по-разному реализованы threads и т.д. Однако все эти системы объединяет тот факт, что в основе каждой из них находится SVR 4.x. Поэтому основной набор системных и библиотечных вызовов в этих реализациях совпадает.

3 Santa Cruz Operation и SCO UNIX.

Варианты ОС UNIX, производимые компанией SCO и предназначенные исключительно для использования на Intel-платформах, до сих пор базируются на лицензированных исходных

текстах System V 3.2. Однако SCO довела свои продукты до уровня полной совместимости со всеми основными стандартами (в тех позициях, для которых существуют стандарты).

Консерватизм компании объясняется прежде всего тем, что ее реализация ОС UNIX включает наибольшее количество драйверов внешних устройств и поэтому может быть установлена практически на любой Intel-платформе.

4 Open Software Foundation и OSF-1.

OSF была первой коммерческой компанией, решившейся на полную реализацию ОС UNIX на базе микроядра Mach. Результатом этой работы явилось создание ОС OSF-1. На сегодняшний день наиболее серьезным потребителем OSF-1 является компания Digital Equipment на своих платформах, основанных на микропроцессорах Alpha. В OSF-1 поддерживаются все основные стандарты ОС UNIX, хотя многие утверждают, что пока система работает не очень устойчиво.

5 Свободно распространяемые и коммерческие варианты ОС UNIX семейства BSD.

Многие годы варианты ОС UNIX, разработанные в Калифорнийском университете г. Беркли, являлись реальной альтернативой AT&T UNIX. Например, ОС UNIX BSD 4.2 была бесплатно доступна в исходных текстах и достаточно широко использовалась даже в нашей стране на оригинальных и воспроизведенных машинах линии DEC.

Группа BSD оказала огромное влияние на общее развитие ОС UNIX. До появления SVR 4.0 проблемой для пользователей являлась несовместимость наборов системных вызовов BSD и System V.

Несколько лет назад группа BSD разделилась на коммерческую и некоммерческую части. Новая коммерческая компания получила название BSDI. Обе подгруппы выпустили варианты ОС UNIX для Intel-платформ под названиями 386BSD и BSD386, причем коммерческий вариант был гораздо более полным.

Сегодня популярен новый свободно распространяемый вариант ОС UNIX, называемый FreeBSD. Ведутся работы над более развитыми версиями BSDNet.

Системы семейства BSD на сегодняшний день не являются единственными свободно доступными вариантами ОС UNIX.

6 Linux университета Хельсинки.

LINUX - это оригинальная реализация ОС UNIX для Intel-платформ, выполненная молодым сотрудником университета Хельсинки Линусом Торвальдом.

Ядро системы написано в традиционной технологии (т.е. без использования микроядра). Однако по отзывам любителей ядро LINUX отличается высоким качеством кода и хорошей модульностью. Кроме того, утверждается, что при аккуратном программировании прикладные программы, созданные в среде LINUX, без особых проблем переносятся в среду коммерческих систем, базирующихся на System V.

7 Hurd Free Software Foundation.

Проект системы Hurd явился попыткой довести до логического завершения знаменитый проект GNU Ричарда Столлмана, основателя и президента Фонда свободного программного обеспечения (Free Software Foundation - FSF). Основной идеей проекта Hurd было использование в качестве основы системы готового варианта микроядра Mach, бесплатно распространяемого университетом Карнеги-Меллон. Сам Ричард Столлман рекомендует пока использовать LINUX совместно с продуктами линии GNU.