Лекция 7. Управление процессами и нитями в ОС UNIX

План :

1 Среда выполнения процессов в ОС UNIX

2 Пользовательская и ядерная составляющие процессов. Принципы организации многопользовательского режима

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

1 Управление процессами и нитями

1.1.1 Среда выполнения процессов

1.1.1.1 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.

1.1.1.2 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.

1.1.1.3 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.

1.1.1.4 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.

2 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.

1.1.2 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 внешнем 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.

3 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.

1.1.3 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.

Упражнения к лекции.

1. Как в ОС UNIX происходит порождение процессов?
2. Сколько классов приоритетов и какие существуют в ОС UNIX?
3. Какие элементы конструкционных блоков используются в ОС UNIX?
4. Пользовательская и ядерная составляющие процессов.
5. Какой алгоритм планирования процессов используется в ОС UNIX? Дайте его краткое описание.
6. Какой традиционный механизм управления процессами на уровне пользователя?
7. Понятие нити в ОС UNIX.

Литература к лекции.

5.1 А.В.Гордеев, А.Ю.Молчанов. Системное программное обеспечение. — "Питер", 2002. — 736с.

5.2 Кристиан К. Введение в операционную систему Unix: пер. с англ. — М. Финансы и статистика, 1985. – 360с.

5.3 Робачевский А.М., Немнюгин С.А., Стесик О.Л. Операционная система Unix. 2-е изд.– СПб.: БХВ – Петербург, 2005. – 635с.