Оглавление

1 Основные черты UNIX.

1. Код системы написан **на языке высокого уровня С**, что сделало ее простой для понимания, изменений и переноса на другие платформы. По оценкам одного из создателей UNIX, Дэнниса Ритчи, система на языке С имела на 20—40% больший размер, а производительность ее была на 20% ниже аналогичной системы, написанной на ассемблере. Несмотря на то, что большинство UNIX поставляется сегодня не в исходных текстах, а в виде бинарных файлов, система остается легко расширяемой и настраиваемой.

2. UNIX — **многозадачная многопользовательская система** с широким спектром услуг. Один мощный сервер может обслуживать запросы большого количества пользователей. При этом необходимо администрирование только одной системы. ОС может выполнять различные функции - работать как вычислительный сервер, сервер базы данных, сетевой сервер, поддерживающий важнейшие сервисы сети (telnet, ftp, электронную почту, службу имен DNS и т. д.), как сетевой маршрутизатор.

3. **Наличие стандартов**. Несмотря на многообразие версий UNIX, основой всего семейства являются принципиально одинаковая архитектура и ряд стандартных интерфейсов. Опытный администратор без большого труда сможет обслужить другую версию системы, для пользователей переход на другую версию и вовсе может оказаться незаметным.

4. Простой, но мощный **модульный пользовательский интерфейс**. Имея в своем распоряжении набор утилит, каждая из которых решает узкую специализированную задачу, можно конструировать из них сложные комплексы. Для пользователя запускается командный интерпретатор (один из них) shell
графический интерфейс X Window

5. Использование единой, легко обслуживаемой **иерархической файловой системы.** Файловая система - это не только доступ к данным, хранящимся на диске. Через унифицированный интерфейс файловой системы осуществляется доступ к терминалам, принтерам, магнитным лентам, сети и даже к памяти.

6. Очень **большое количество приложений**, в том числе свободно распространяемых, начиная от простейших текстовых редакторов и заканчивая мощными системами управления базами данных.

2 Структура системы и ядро UNIX

2.1 Общая архитектура системы UNIX

Классическая UNIX представляет собой многопользовательскую операционную систему, основным компонентом которой является ядро, отвечающее за

- **управление устройствами от имени приложений,**
- **планирование доступа к ресурсам,**
- **обеспечение защиты процессов друг от друга.**

2.2 Основные модули ядра:

Интерфейс системного вызова - позволяет пользовательским процессам обращаться к сервисам операционной системы;

Модули символьного и блочного ввода-вывода, а также драйверов устройств -используются при реализации файловых систем и для доступа к устройствам;

Буферный кэш - отвечает за кэширование данных, над которыми выполняются блочные операции ввода-вывода; он повышает производительность системы;

Файловая подсистема - управляет иерархическим пространством имен файлов, каталогов и устройств ввода-вывода, именование которых унифицировано

Модуль управления памятью - поддерживает виртуальную память UNIX;

Модуль управления процессами — отвечает за создание и планирование процессов, прекращение их работы и поддержку базовых средств их взаимодействия.

2.3 Системные вызовы и выполнение кода ядра

В ядре операционной системы UNIX реализован ряд сервисов, доступ к которым осуществляется из процессов прикладного уровня посредством системного вызова.

Интерфейс системного вызова - это стандартный управляемый способ входа прикладных процессов в ядро операционной системы с переключением в привилегированный режим выполнения.

Когда процесс входит в ядро, идентификатор данного процесса остается неизменным, но сам он получает полный доступ к памяти и устройствам компьютера.

Таким образом, можно сказать, что ядро UNIX выполняется процедурно, поскольку при системном вызове происходит нечто похожее на выполнение процедуры в обычной прикладной программе.

Данная архитектура противоположна архитектуре на основе передачи сообщений, при которой ядро может функционировать параллельно с

пользовательскими процессами, взаимодействующими с ним путем отправки и получения сообщений.

Ядро операционной системы UNIX выполняется без вытеснения.

Это означает, что процесс, функционирующий в режиме ядра, не может быть вытеснен планировщиком с целью передачи управления другому процессу.

Однако при этом процесс ядра может вызвать блокирующую операцию, в частности, когда он выполняет ввод-вывод и вынужден приостановить работу до тех пор, пока устройство не удовлетворит запрос.

Если во время выполнения процесса ядра происходит прерывание, то после его обработки управление возвращается тому же процессу.

В отличие от процесса ядра пользовательский процесс в такой ситуации может быть вытеснен с процессора, особенно в том случае, когда прерывание сигнализирует о завершении блокирующей операции ввода-вывода.

Наличие перечисленных выше ограничений объясняется тем обстоятельством, что UNIX первоначально разрабатывалась для использования на однопроцессорных компьютерах, где переключение между процессами существенно замедляет работу.

Невытесняющее планирование процессов ядра помогает избежать многих сложностей, связанных с управлением параллельным выполнением.
Исходная ориентация на однопроцессорную платформу определяет всю архитектуру классической UNIX.

При такой архитектуре доступ процессов к общим структурам данных ядра не представляет проблемы.

Войдя в ядро, процесс выполняется до тех пор, пока не примет решение о блокировании, следовательно, может выполнять свои критические секции без вмешательства со стороны других процессов. Если же существует вероятность модификации программой обработки прерывания некоторой структуры данных, с которой процесс работает, он может просто запретить прерывания на время выполнения критического кода.

Обычно в UNIX-системах определены приоритеты прерываний. Например, если нужно без помех обрабатывать важные для системы прерывания от таймера, можно так определить приоритеты, чтобы во время обработки другого прерывания, скажем, от сетевого устройства, было разрешено поступление прерываний от таймера, но не наоборот.

3 Управление процессами

Процессом в системе UNIX называется единица исполнения программного кода (классическая UNIX не поддерживает потоки).

В основе UNIX лежит концепция процесса - единицы управления и единицы потребления ресурсов.

Процесс представляет собой программу в состоянии выполнения, причем в UNIX в рамках одного процесса не могут выполняться никакие параллельные действия.

Каждый процесс выполняется в своем виртуальном адресном пространстве. Совокупность участков физической памяти, отображаемых на виртуальные адреса процесса, называется **образом процесса**.

3.1 Основные структуры данных процесса.

При управлении процессами операционная система использует два основных типа информационных структур:

- **дескриптор процесса (структура proc)**
- **и контекст процесса (структура user).**

Дескриптор процесса содержит такую информацию о процессе, которая необходима ядру в течение всего жизненного цикла процесса, независимо от того, находится ли он в активном или пассивном состоянии, находится ли образ процесса в оперативной памяти или выгружен на диск.

Дескрипторы отдельных процессов объединены в список, образующий системную таблицу процессов.

Память для таблицы процессов отводится динамически в области ядра.

На основании информации, содержащейся в таблице процессов, операционная система осуществляет планирование и синхронизацию процессов.

В дескрипторе прямо или косвенно (через указатели на связанные с ним структуры) содержится информация о

- **состоянии процесса,**
- **расположении образа процесса в оперативной памяти и на диске,**
- **о значении отдельных составляющих приоритета, а также его итоговое значение - глобальный приоритет,**
- **идентификатор пользователя, создавшего процесс,**
- **информация о родственных процессах, о событиях, осуществления которых ожидает данный процесс**
- **некоторая другая информация.**

Контекст процесса содержит менее оперативную, но более объемную часть информации о процессе, необходимую для возобновления выполнения процесса с прерванного места:

- **содержимое регистров процессора,**
- **коды ошибок выполняемых процессором системных вызовов,**
- **информацию о всех открытых данным процессом файлов и незавершенных операциях ввода-вывода (указатели на структуры file)**
- **другие данные, характеризующие состояние вычислительной среды в момент прерывания.**

Контекст, так же как и дескриптор процесса, доступен только программам ядра, то есть находится в виртуальном адресном пространстве операционной системы, однако он хранится не в области ядра, а непосредственно примыкает к образу процесса и перемещается вместе с ним, если это необходимо, из оперативной памяти на диск.

В UNIX для процессов предусмотрены два режима выполнения: привилегированный режим - "система" и обычный режим - "пользователь".

В режиме "пользователь" запрещено выполнение действий, связанных с управлением ресурсами системы, в частности, корректировка системных таблиц, управление внешними устройствами, маскирование прерываний, обработка прерываний.

В режиме "система" выполняются программы ядра, а в режиме "пользователь" - оболочка и прикладные программы.

При необходимости выполнить привилегированные действия пользовательский процесс обращается с запросом к ядру в форме системного вызова. В результате системного вызова управление передается соответствующей программе ядра. С момента начала выполнения системного вызова процесс считается системным. Таким образом, один и тот же процесс может находиться в пользовательской и системной фазах. Эти фазы никогда не выполняются одновременно.

В классической UNIX процесс, работающий в режиме системы, не мог быть вытеснен другим процессом. Из-за этого организация ядра, которое составляет привилегированную общую часть всех процессов, упрощалась, т.к. все функции ядра не были реентерабельными. Однако, при этом реактивность системы страдала - любой процесс, даже низкоприоритетный, войдя в системную фазу, мог оставаться в ней сколь угодно долго. Из-за этого свойства UNIX не мог использоваться в качестве ОС реального времени.

В более поздних версиях организация ядра усложнилась и процесс можно вытеснить и в системной фазе, но не в произвольный момент времени, а только в определенные периоды его работы, когда процесс сам разрешает это сделать установкой специального сигнала.

3.2 Состояния процесса

Сначала процесс создается и направляется на выполнение:

• **с-гп**: только что созданному процессу выделяется некоторый объем основной памяти;

• **гп-вя**: процесс направляется на выполнение с некоторым начальным приоритетом;

• **вя-вп**: перешел из режима ядра в пользовательский режим.

Далее процесс работает в пользовательском режиме - он либо будет прерван либо выполнит системный вызов:

• **вп-вя**: процесс, работающий в пользовательском режиме, произвел системный вызов и теперь выполняет код ядра или же произошло прерывание, например, от устройства или таймера, и это прерывание обрабатывается в контексте данного процесса;

• **вя-вп**: возврат из системного состояния в пользовательское после обработки прерывания или по окончании функционирования вызванной системной процедуры (либо сразу, либо после периода сна, когда процесс пробуждается и направляется на выполнение: вя-сп-гп-вя);

• **вя-в**: процесс вытеснен в тот момент, когда должен был перейти в пользовательское состояние (это могло произойти из-за того, что процесс ядра был переведен в состояние готовности в результате прерывания или же по причине того, что после прерывания от таймера выяснилось, что время данного процесса истекло;

• **в- вп**: ни один из системных процессов не находится в состоянии готовности, поэтому на выполнение направляется вытесненный процесс;

• **вя-вя**: произошло прерывание, которое обрабатывается в контексте прерванного процесса; если этот процесс выполнялся в пользовательском режиме, механизм прерывания изменяет его состояние на системное;

• **вя-сп**: процесс, выполняющий код ядра, засыпает в ожидании события;

• **сп-гп**: произошло событие, которого ожидал процесс, и он переводится в состояние готовности;

• гп-вп: процесс направляется на выполнение в соответствии с тем приоритетом, который был у него на момент блокировки;

• сп-со: спящий процесс откачан из памяти на диск;

• со-го: произошло событие, которого ждал процесс, и он переводится в состояние готовности, но все еще находится на диске;

• го-гп: процесс загружается с диска в память;

• гп-го: готовый к выполнению процесс откачивается из памяти на диск;

• с-го: для нового процесса недостаточно памяти;

• ря-з: процесс выполняет системный вызов exit() (вп-вс) и удаляется из системы; он остается «зомби» до тех пор, пока его код не будет присоединен родительским процессом.

При переходах вя-сп и гп-вя производится проверка сигналов, связанных со взаимодействием процессов; кроме того, сигналы проверяются и обрабатываются, когда процесс намерен перейти в пользовательское состояние (вя-вп или в-вп).

Системные вызовы выполняются в вызывающем процессе, который лишь переходит в привилегированное состояние.

Выделенный процесс для обработки прерываний от устройств отсутствует, поэтому все такого рода прерывания обрабатываются в контексте прерванного процесса.

Механизм прерываний гарантирует их обработку в привилегированном режиме.

Пользовательский процесс никогда не блокируется непосредственно: он делает это либо после попытки выполнения ввода или вывода, либо в результате вызова для взаимодействия с другим процессом, причем в обоих случаях сначала входит в ядро.

Процесс должен войти в ядро и для того, чтобы завершить свою работу.

3.3 Системные вызовы управления процессами

3.3.1 Порождение процесса

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. В противном случае сын становится еще одним процессом, разделяющим данный процедурный сегмент.

После выполнения системного вызова fork оба процесса продолжают выполнение с одной и той же точки. Чтобы процесс мог опознать, является ли он отцом или сыном, системный вызов fork возвращает в качестве своего значения в породивший процесс идентификатор порожденного процесса, а в порожденный процесс NULL. Типичное разветвление на языке С записывается так:

```
if( fork() )  {  действия отца }
else   { действия сына }
```

Идентификатор сына может быть присвоен переменной, входящей в контекст процесса-отца. Так как контекст процесса наследуется его потомками, то дети могут узнать идентификаторы своих старших братьев, так образом сумма знаний наследуется при порождении и может быть распространена между родственными процессами. Наследуются все характеристики процесса, содержащиеся в контексте.

На независимости идентификатора процесса от выполняемой процессом программы построен механизм, позволяющий процессу придти к выполнению другой программы с помощью системного вызова exec.

Таким образом в UNIX порождение нового процесса происходит в два этапа - сначала создается копия процесса-родителя, то есть дублируется дескриптор, контекст и образ процесса. Затем у нового процесса производится замена кодового сегмента на заданный.

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

Любой процесс создается с помощью системного вызова fork(), возвращающего идентификатор процесса.

В ответ на вызов UNIX создает новое адресное пространство с полной копией содержимого адресного пространства родительского процесса.

Затем в адресное пространство с помощью системного вызова execve() может быть загружена для выполнения программа.

В этом случае существующее содержимое сегмента кода процесса заменяется кодом новой программы, а содержимое сегмента данных обновляется. Поскольку процесс часто создается для выполнения новой программы, работа по копированию адресного пространства родительского процесса оказывается напрасной.

3.3.2 Синхронизация между родительским и дочерним процессами

Если родительский процесс создает для выполнения определенной подзадачи дочерний процесс, ему нужно знать, когда тот завершит свою работу. С этой целью применяется следующий системный вызов, выполняемый дочерним процессом:

exit (информация о состоянии)

В ответ на него из системы удаляется вся информация адресного пространства процесса, за исключением управляющего блока, где хранится предназначенная для родительского процесса информация о состоянии.

Такой процесс в UNIX называется «зомби».

Родительский процесс, в свою очередь, выполняет блокирующий системный вызов wait(), указывая идентификатор дочернего процесса, а в ответ система возвращает байт состояния и некоторую дополнительную информацию.

3.4 Планирование процессов. Уровни приоритетов

В UNIX реализовано планирование на основе динамического приоритета процессов, согласно которому для выполнения каждый раз выбирается процесс с наивысшим приоритетом. При этом планировщик вычисляет приоритеты

процессов с учетом их поведения, а не назначает им статические приоритеты при создании.

- Процесс, который в настоящий момент находится в ядре, имеет более высокий приоритет, чем процессы пользователя. Процессы ядра планируются без вытеснения и поэтому, выполняя системный код, процесс работает до тех пор, пока не произведет некоторый блокирующий вызов или не будет временно прерван устройством.
- При блокировании процессу назначается приоритет, соответствующий событию, которого он ждет.
- Пользовательские процессы планируются в соответствии с принципом квантования по времени. Алгоритм планирования учитывает количество использованного процессом времени процессора и на основании этого динамически назначает ему приоритет.

Планировщик, равно как и большинство других компонентов ядре UNIX, выполняется в контексте пользовательского процесса. Как правило, это процесс, который был вытеснен или заблокирован последним.

Когда процесс выполняется в режиме ядра, ему иногда приходится дожидаться какого-либо события или освобождения необходимого ресурса (устройства ввода или буфера).

В таком случае процесс вызывает примитив **sleep(),** который переводит его в блокированное состояние.

Управление передается планировщику (в контексте этого же процесса), и тот направляет на процессор следующий процесс.

Если происходит ожидаемое процессом событие или освобождается ресурс, выполняется примитив **wakeup().** Он может быть вызван из обработчика прерывания, связанного с данным событием, и здесь снова имеет место процедурное выполнение ядра.

Примитив **wakeup()** переводит все заблокированные процессы в состояние готовности.

Вместо того, чтобы записывать информацию об ожидаемых и произошедших событиях в дескрипторы процессов, UNIX определяет для каждого события адрес в ядре.

Соотношение между событиями и адресами не обязательно должно быть «один-к-одному», так что процессы, ожидающие разных событий, которым соответствует один и тот же адрес, будут пробуждены, когда процедура wakeup()обратится по этому адресу.

Невытесняющее планирование процессов ядра гарантирует, что процесс может завершить выполнение примитива sleep () или wakeup() без вмешательства со стороны других процессов. А для того чтобы в это время не происходили прерывания, таковые отключаются.