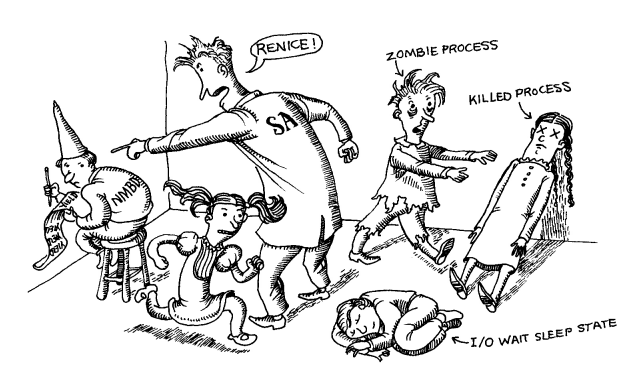
***4 Контроль над процессами***



Процесс представляет собой запущенную программу. Это абстракция, с помощью которой можно управлять и следить за памятью, процессорным временем и ресурсами ввода-вывода.

Это аксиома философии UNIX, что как можно больше работы должно быть сделано в контексте процессов, а не обрабатываться специально ядром. Системные и пользовательские процессы следуют одним и тем же правилам, поэтому вы можете использовать один набор инструментов для управления ими обоими.

**4.1 Компоненты процесса**

Процесс состоит из адресного пространства и набора структур данных внутри ядра. Адресное пространство - это набор страниц памяти, которые ядро выделило для использования процессом1. Эти страницы содержат код и библиотеки, которые выполняет процесс, переменные процесса, его стеки и различную дополнительную информацию, необходимую ядру во время выполнения процесса. Виртуальное адресное пространство процесса случайным образом располагается в физической памяти и отслеживается таблицами страниц ядра.

*1. Страницы - это единицы измерения, в которых осуществляется управление памятью. Они обычно имеют размер 4 KiB или 8 KiB*

Внутренние структуры данных ядра записывают различные фрагменты информации о каждом процессе. Вот некоторые наиболее важные из них:

* Карта адресного пространства процесса
* Текущее состояние процесса (спящий, остановленный, запущенный и т.д.)
* Приоритет выполнения процесса
* Информация о ресурсах, которые использовал процесс (процессор, память и т.д.)
* Информация о файлах и сетевых портах, открытых процессом
* Маска сигнала процесса (запись о том, какие сигналы блокируются)
* Владелец процесса

"Поток" - это контекст выполнения внутри процесса. Каждый процесс имеет по крайней мере один поток, но некоторые процессы имеют много потоков. Каждый поток имеет свой собственный стек и контекст процессора, но работает в адресном пространстве своего заключающего процесса.

Современное компьютерное оборудование включает в себя несколько процессоров и несколько ядер на процессор. Потоки процесса могут выполняться одновременно на разных ядрах. Многопоточные приложения, такие как BIND и Apache, получают довольно много преимуществ от этой архитектуры, поскольку она позволяет им распределять запросы по отдельным потокам.

Многие параметры, связанные с процессом, непосредственно влияют на его выполнение: количество получаемого процессорного времени, файлы, к которым он может получить доступ, и так далее. В следующих разделах мы обсудим значение и значимость параметров, которые наиболее интересны с точки зрения системного администратора. Эти атрибуты являются общими для всех версий UNIX и Linux.

**PID: ID процесса**

Ядро присваивает каждому процессу уникальный идентификационный номер. Большинство команд и системных вызовов, которые управляют процессами, требуют указания PID для определения целевого объекта операции. PID’ы назначаются в порядке создания процессов.

Linux в настоящее время определяет понятие "пространства имен" процесса, которое еще больше ограничивает способность процессов видеть и влиять друг на друга. Контейнерные реализации используют эту функцию, чтобы разделять процессы. Один из побочных эффектов заключается в том, что процесс может иметь различные PID в зависимости от пространства имен наблюдателя. Это что-то вроде теории относительности Эйнштейна для идентификаторов процессов. Дополнительную информацию см. В Главе 25 "Контейнеры".

**PPID: родительский PID**

Ни в UNIX, ни в Linux нет системного вызова, который инициирует новый процесс, запускающий определенную программу. Вместо этого все делается в два отдельных этапа. Во-первых, существующий процесс должен клонировать себя, чтобы создать новый процесс. Затем клон может заменить запущенную программу на другую.

Когда процесс был клонирован, исходный процесс становится родительским, а копия - дочерним. Атрибут PPID процесса - это PID родителя, из которого он был клонирован2.

*2. По крайней мере, поначалу. Если исходный родитель умирает, init или systemd (процесс 1) становится новым родителем. Смотрите страницу 94.*

Родительский PID - это полезная информация, когда вы сталкиваетесь с неизвестным (и, возможно, плохо работающим) процессом. Прослеживание процесса до его истока (будь то оболочка или какая-то другая программа) может дать вам лучшее представление о его цели и значении.

**UID и EUID: реальный и эффективный ID пользователя**

UID процесса-это идентификационный номер пользователя, создавшего его, или, точнее, это копия значения UID родительского процесса. Обычно только создатель (он же владелец) и суперпользователь могут управлять процессом.

*Дополнительную информацию о UID смотрите на стр. 248.*

EUID - это "эффективный" идентификатор пользователя, дополнительный UID, который определяет, к каким ресурсам и файлам процесс имеет доступ в данный момент. Для большинства процессов UID и EUID являются одинаковыми, обычным исключением являются setuid программы.

*Дополнительную информацию о выполнении setuid смотрите на стр. 68.*

Для чего же нужны UID и EUID? Просто для того, что полезно поддерживать различие между идентификацией и разрешением, а также потому, что программа setuid может не захотеть работать с расширенными разрешениями все время. В большинстве систем эффективный UID можно установить и сбросить, чтобы включить или ограничить дополнительные разрешения, которые он предоставляет.

Большинство систем также отслеживают "сохраненный UID", который является копией EUID процесса в момент, когда процесс впервые начинает выполняться. Если процесс не предпримет шаги для удаления этого сохраненного UID, он остается доступным для использования в качестве реального или эффективного UID. Поэтому консервативно написанная программа setuid может отказаться от своих особых привилегий на большую часть времени выполнения и обращаться к ним только в тех случаях, когда требуются дополнительные привилегии.

Linux также определяет нестандартный параметр процесса FSUID, который управляет определением разрешений файловой системы. Он редко используется вне ядра и не переносится в другие системы UNIX.

**GID и EGID: реальный и эффективный идентификатор группы**

GID - это групповой идентификационный номер процесса. EGID связан с GID таким же образом, как EUID связан с UID, поскольку он может быть "обновлен" путем выполнения программы setgid. Как и в случае с сохраненным UID, ядро поддерживает сохраненный GID для каждого процесса.

*Дополнительную информацию о группах смотрите на стр. 249*

Атрибут GID процесса в значительной степени рудиментарен. Для целей определения доступа процесс может быть членом сразу нескольких групп. Полный список групп хранится отдельно от выделенных GID и EGID. Определение прав доступа обычно учитывает EGID и дополнительный список групп, но не сам GID.

Единственный раз, когда GID действительно важен, - это когда процесс создает новые файлы. В зависимости от того, как были установлены разрешения файловой системы, новые файлы могут по умолчанию принимать GID создающего процесса. Подробнее см. стр. 134.

**Niceness**

Приоритет планирования процесса определяет, сколько процессорного времени он получает. Ядро вычисляет приоритеты с помощью динамического алгоритма, который учитывает количество процессорного времени, которое процесс недавно использовал, и продолжительность времени, которое он ожидал перед запуском. Ядро также обращает внимание на административно заданное значение, которое обычно называется "хорошим значением" или "niceness" (вежливость), потому что оно определяет, насколько хорошо вы планируете относиться к другим пользователям системы. Мы подробно обсуждаем niceness на странице 102.

**Управляющий терминал**

Большинство процессов, недемонов, имеют связанный терминал управления. Управляющий терминал определяет связи по умолчанию для каналов стандартного ввода (STDIN), стандартного вывода (STDOUT) и стандартных ошибок (STDERR). Он также передает сигналы процессам в ответ на события, такие как <Control-C>; см. обсуждение, начинающееся на стр. 94.

*Дополнительную информацию о стандартных каналах связи смотрите на стр. 190*

Конечно, настоящие терминалы в наши дни редко встречаются за пределами компьютерных музеев. Тем не менее, они продолжают существовать в виде псевдотерминалов, которые до сих пор широко используются в системах UNIX и Linux. Например, при запуске команды из командной консоли окно терминала обычно становится терминалом управления процессом.

Конечно, настоящие терминалы в наши дни редко встречаются за пределами компьютерных музеев. Тем не менее, они продолжают существовать в виде псевдотерминалов, которые до сих пор широко используются в системах UNIX и Linux. Например, при запуске команды из командной консоли окно терминала обычно становится терминалом управления процессом.

**4.2 Жизненный цикл процесса**

Чтобы создать новый процесс, процесс копирует себя с помощью системного вызова **fork**3. **fork** создает копию исходного процесса, и эта копия в значительной степени идентична родительской. Новый процесс имеет отдельный PID и имеет свою собственную учетную информацию.

*3. Технически, системы Linux используют* ***clone****, расширенный вариант* ***fork****, который обрабатывает потоки и включает в себя дополнительные функции.* ***fork*** *остается в ядре для обратной совместимости, но негласно вызывает* ***clone****.*

**fork** обладает уникальным свойством возвращать два разных значения. С точки зрения потомка, он возвращает ноль. Родитель получает PID вновь созданного дочернего объекта. Поскольку эти два процесса в остальном идентичны, они оба должны изучить возвращаемое значение, чтобы выяснить, какую роль они должны играть.

После того, как отработает **fork**, дочерний процесс часто использует одну из подпрограмм семейства **exec** для начала выполнения новой программы. Эти вызовы изменяют программу, которую выполняет процесс, и сбрасывают сегменты памяти в предварительно определенное начальное состояние. Различные формы **exec** различаются только тем, как они задают аргументы командной строки и окружение, которое будет дано новой программе.

Когда система загружается, ядро автономно создает и устанавливает несколько процессов. Наиболее заметными из них являются **init** или **systemd**, которые всегда является процессом номер 1. Этот процесс выполняет скрипты запуска системы, хотя действительный способ работы немного различается между UNIX и Linux. Все процессы, кроме тех, которые создает ядро, являются потомками этого изначального процесса. Смотрите Главу 2 для получения дополнительной информации о загрузке и возможностях демона **init**.

**init** (или **systemd**) также играет еще одну важную роль в управлении процессами. Когда процесс завершается, он вызывает процедуру с именем **\_exit**, чтобы уведомить ядро о том, что он готов умереть. Она предоставляет код выхода (целое число), который говорит, почему процесс завершается. По соглашению, ноль указывает на нормальное или "успешное" завершение.

Прежде чем мертвому процессу будет разрешено исчезнуть полностью, ядро требует, чтобы его смерть была признана родительским процессом, что родитель и делает с помощью вызова **wait**. Родитель получает копию кода выхода потомка (или, если потомок не завершился добровольно, указание на то, почему он был убит), а также может получить сводку об использовании ресурсов дочернего процесса, если он пожелает.

Эта схема прекрасно работает, если родители существуют дольше своих потомков и добросовестно используют **wait**, чтобы можно было избавиться от мертвых процессов. Однако если родитель умирает раньше своих потомков, ядро распознает, что никакого **wait** уже не будет. Ядро настраивает сиротские процессы, чтобы сделать их потомками **init** или **systemd**, которые вежливо выполняют **wait**, чтобы избавиться от них, когда они умрут.

**Сигналы**

Сигналы - это запросы на прерывание уровня процесса. Определено около тридцати различных видов сигналов, и они используются различными способами:

* Они могут передаваться между процессами как средство коммуникации.
* Они могут быть отправлены драйвером терминала для уничтожения, прерывания или приостановки процессов при нажатии комбинаций клавиш <Control-C> и <Control-Z>4.
* Они могут быть отправлены администратором (с помощью **kill**) для достижения различных целей.
* Они могут быть отправлены ядром, когда процесс совершает нарушение, такое как деление на ноль.
* Они могут быть отправлены ядром, чтобы уведомить процесс об "интересном" состоянии, таком как смерть дочернего процесса или доступность данных на канале ввода-вывода.

*4. Функции <Control-Z> и <Control-C> могут быть переназначены другим клавишам с помощью команды* ***stty****, но на практике это происходит редко. В этой главе мы обращаемся к ним через их условные привязки.*

Когда сигнал получен, может произойти одно из двух событий. Если принимающий процесс назначил подпрограмму обработчика для этого конкретного сигнала, обработчик вызывается с информацией о контексте, в котором был доставлен сигнал. В противном случае ядро выполняет какое-либо действие по умолчанию от имени процесса. Действие по умолчанию варьируется от сигнала к сигналу. Многие сигналы завершают процесс; некоторые также генерируют дампы памяти (если они не были отключены).

*Дамп памяти - это копия образа памяти процесса, которая иногда бывает полезна для отладки.*

Указание процедуры обработчика для сигнала называется перехватом сигнала. Когда обработчик завершается, выполнение перезапускается с точки, в которой был получен сигнал.

Чтобы предотвратить поступление сигналов, программы могут запросить их игнорирование или блокировку. Сигнал, который игнорируется, просто отбрасывается и не оказывает никакого влияния на процесс. Заблокированный сигнал помещается в очередь для доставки, но ядро не требует, чтобы процесс реагировал на него до тех пор, пока сигнал не будет явно разблокирован. Обработчик вновь разблокированного сигнала вызывается только один раз, даже если сигнал был получен несколько раз во время блокировки приема.

В таблице 4.1 перечислены некоторые сигналы, с которыми должны быть знакомы администраторы. Соглашение об использовании верхнего регистра для имен происходит из традиции языка С. Вы также можете увидеть имена сигналов, написанные с префиксом SIG (например, SIGHUP) по аналогичным причинам.

**Таблица 4.1 - Сигналы, которые должен знать каждый администратора**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **#b** | **Имя** | **Описание** | **Действие по умолчанию** | **Может перехватить?** | **Может блокировать?** | **Дамп памяти?** |
| 1 | HUP | Закрытие терминала | Завершение | Да | Да | Нет |
| 2 | INT | Прерывание (Ctrl-C) | Завершение | Да | Да | Нет |
| 3 | QUIT | Сигнал «Quit» с терминала | Завершение | Да | Да | Да |
| 9 | KILL | Безусловное завершение | Завершение | Нет | Нет | Нет |
| 10 | BUS | Ошибка шины | Завершение | Да | Да | Да |
| 11 | SEGV | Ошибка при обращении к памяти | Завершение | Да | Да | Да |
| 15 | TERM | Прекращение работы ПО | Завершение | Да | Да | Нет |
| 17 | STOP | Остановка выполнения процесса | Остановка | Нет | Нет | Нет |
| 18 | TSTP | Сигнал остановки с терминала (Ctrl-Z) | Остановка | Да | Да | Нет |
| 19 | CONT | Продолжить выполнение ранее остановленного процесса | Игнорировать | Да | Нет | Нет |
| 28 | WINCH | Изменение размеров окна терминала | Игнорировать | Да | Да | Нет |
| 30 | USR1 | Пользовательский сигнал #1 | Завершение | Да | Да | Нет |
| 31 | USR2 | Пользовательский сигнал #2 | Завершение | Да | Да | Нет |

1. Список имен и номеров сигналов также доступен из встроенной команды bash **kill-l**.
2. Может отличаться в некоторых системах. Смотрите в **/usr/include/signal.h** или **man signal** для получения дополнительной информации

Другие сигналы, не показанные в таблице 4.1, в основном сообщают о непонятных ошибках, таких как "недопустимая инструкция". Обработка таких сигналов по умолчанию - Завершение с дампом памяти. Перехват и блокирование обычно разрешены, потому что некоторые программы достаточно умны, чтобы попытаться очистить любую проблему, вызвавшую ошибку, прежде чем продолжить.

Сигналы BUS и SEGV также являются сигналами ошибок. Мы включили их в таблицу, потому что они очень распространены: когда программа выходит из строя, обычно именно один из этих двух сигналов в конечном итоге приводит ее к сбою. Сами по себе эти сигналы не имеют особой диагностической ценности. Оба они указывают на попытку неправильного использования памяти или доступа к ней.

Сигналы с именами KILL и STOP не могут быть перехвачены, заблокированы или проигнорированы. Сигнал KILL уничтожает принимающий процесс, а STOP приостанавливает его выполнение до тех пор, пока не будет получен сигнал CONT. CONT может быть перехвачен или проигнорирован, но не заблокирован.

TSTP - это "мягкая" версия STOP, которую лучше всего описать как запрос на остановку. Это сигнал, генерируемый драйвером терминала при вводе <Control-Z> на клавиатуре. Программы, которые перехватывают этот сигнал, обычно очищают свое состояние, а затем посылают себе сигнал остановки для завершения операции остановки. В ином случае, программы могут игнорировать TSTP, чтобы предотвратить их остановку с клавиатуры.

Сигналы KILL, INT, TERM, HUP и QUIT все звучат так, как будто они означают примерно одно и то же, но используются они совершенно по-разному. Досадно, что для них была выбрана такая расплывчатая терминология. Вот руководство по декодированию:

* KILL не блокируется и завершает процесс на уровне ядра. Процесс никогда не сможет на самом деле получить или обработать этот сигнал.
* INT отправляется драйвером терминала, когда пользователь нажимает <Control-C>. Это запрос на прекращение текущей операции. Простые программы должны выйти (если они получают сигнал) или просто позволить себя убить, что является поведением по умолчанию, если сигнал не получен. Программы, имеющие интерактивные командные строки (например, shell), должны прекратить то, что они делают, очистить и ждать ввода данных пользователем.
* TERM - это запрос на полное прекращение выполнения. Ожидается, что принимающий процесс очистит свое состояние и закроется.
* HUP имеет два толкования. Во-первых, он понимается как запрос на перезагрузку многими демонами. HUP обычно может вызвать такое поведение если демон способен перечитать свой конфигурационный файл и приспособиться к изменениям без перезапуска.

Во-вторых, сигналы HUP иногда генерируются драйвером терминала в попытке "очистить" (т. е. убить) процессы, подключенные к определенному терминалу. Такое поведение в значительной степени является пережитком со времен проводных терминалов и модемных соединений, отсюда и название "прекращение" (hangup).

Оболочки семейства C shell (**tcsh** и др.) обычно делают фоновые процессы невосприимчивыми к сигналам HUP, чтобы они могли продолжать работать после выхода пользователя из системы. Пользователи оболочек Bourne (**ksh**, **bash** и др.) могут эмулировать это поведение с помощью команды **nohup**.

* QUIT похож на TERM, за исключением того, что он по умолчанию создает дамп памяти, если не перехвачен. Несколько программ разбирают этот сигнал и интерпретируют его по-другому.

Сигналы USR1 и USR2 не имеют заданного значения. Они доступны для программ, чтобы использовать их так, как они хотят. Например, веб-сервер Apache интерпретирует сигнал HUP как запрос на немедленную перезагрузку. Сигнал USR1 инициирует более изящный переход, в котором существующие клиентские соединения не разрываются немедленно, а завершаются по инициативе пользователей.

**kill: посыл сигналов**

Как следует из названия, команда **kill** чаще всего используется для завершения процессов. **kill** может послать любой сигнал,но по умолчанию он посылает TERM. **kill** может использоваться обычными пользователями на их собственных процессах или пользователем root на любом процессе. Синтаксис таков:

**kill** *[-signal] pid*

где *signal* - это номер или символическое имя посылаемого сигнала (как показано в таблице 4.1), а *pid* - идентификационный номер целевого процесса.

**kill** без номера сигнала не гарантирует, что процесс умрет, потому что сигнал TERM может быть перехвачен, заблокирован или проигнорирован. Следующая команда "гарантирует", что процесс умрет, потому что сигнал 9, KILL, не может быть перехвачен.

$ **kill -9** *pid*

Используйте **kill -9**, только если "вежливая просьба" не будет выполнена. Мы ставим кавычки вокруг "гарантирует", потому что процессы иногда могут настолько заклиниваться, что даже KILL не может повлиять на них. Обычно такое случается из-за невозможности блокировки устройств ввода-вывода, например, ожидание исчезнувшего диска. Перезагрузка обычно является единственным способом избавиться от таких процессов.

**killall** убивает процессы по имени. Например, следующая команда убивает все процессы веб-сервера Apache:

$ **sudo killall httpd**

Команда **pkill** выполняет поиск процессов по имени (или другим атрибутам, таким как EUID) и отправляет указанный сигнал. Например, следующая команда посылает сигнал TERM всем процессам, запущенным от имени пользователя ben:

$ **sudo pkill -u ben**

**Состояния процессов и потоков**

Как вы видели в предыдущем разделе, процесс может быть приостановлен с помощью сигнала STOP и возвращен к активной работе с помощью сигнала CONT. Состояние приостановки или запуска применяется к процессу в целом и наследуется всеми потоками процесса5.

*5. Отдельные потоки фактически могут управляться аналогично. Однако эти объекты в первую очередь представляют интерес для разработчиков; системные администраторы не должны беспокоиться о них.*

Даже когда потоки могут выполняться, они часто должны ждать, пока ядро завершит некоторую фоновую работу для них, прежде чем они смогут продолжить выполнение. Например, когда поток считывает данные из файла, ядро должно запросить соответствующие дисковые блоки и затем организовать доставку их содержимого в адресное пространство запрашивающего процесса. В течение этого времени запрашивающий поток переходит в состояние кратковременного сна, в котором он не может быть выполнен. Однако другие потоки в том же процессе могут продолжать работать.

Иногда вы будете видеть целые процессы со статусом "спящие" (например, в выводе **ps** - cм. cледующий раздел). Поскольку спящий режим является атрибутом уровня потока, его применение немного обманчиво. Процесс обычно становится "спящим", когда все его потоки спят. Однако, это различие является спорным в случае однопоточных процессов, которые наиболее распространены.

Интерактивные оболочки и системные демоны большую часть времени проводят в спящем режиме, ожидая ввода с терминала или сетевых подключений. Поскольку спящий поток эффективно блокируется до тех пор, пока его запрос не будет удовлетворен, его процесс обычно не получает процессорного времени, если он не получает сигнал или ответ на один из своих запросов ввода-вывода.

*Дополнительную информацию о жестко монтируемых файловых системах NFS см. на стр. 809.*

Некоторые операции могут привести к переходу процессов или потоков в состояние бесперебойного сна. Это состояние обычно является переходным и не выводится при использовании **ps** (обозначается *D* в столбце *STAT*; см. таблицу 4.2 на стр. 100). Однако несколько дегенеративных ситуаций могут привести к его сохранению. Наиболее распространенная причина связана с проблемами сервера в файловой системе NFS, смонтированной с параметром *hard*. Поскольку процессы в состоянии непрерывного сна не могут быть пробуждены даже для обслуживания сигнала, они не могут быть убиты. Чтобы избавиться от них, необходимо устранить основную проблему или перезагрузить компьютер.

В реальных условиях иногда вы можете увидеть "зомби"-процессы, которые завершили выполнение, но их статус еще не был обработан их родительским процессом (или **init** или **systemd**). Если вы видите, что "зомби" бродят в вашей системе, проверьте их PPID с помощью **ps**, чтобы узнать, откуда они приходят.

**4.3 PS: мониторинг процессов**

Команда **ps** является основным инструментом системного администратора для мониторинга процессов. Хотя версии **ps** отличаются используемыми аргументами и отображением информации, все они предоставляют по существу одну и ту же информацию. Часть огромных различий между версиями **ps** можно проследить в различиях между версиями в истории развития UNIX. Однако **ps** - это также команда, которую поставщики склонны настраивать по другим причинам. Она тесно связана с обработкой процессов ядром, поэтому имеет тенденцию отражать все основные изменения ядра поставщика.

**ps** может показывать PID, UID, приоритет и терминал управления процессами. Она также сообщает вам, сколько памяти использует процесс, сколько процессорного времени он потребил,и каково его текущее состояние (работает, остановлен, спит и т.д.). Зомби появляются в списке **ps** как <exiting> или <defunct>.

Реализация **ps** с годами только усложняется. Некоторые поставщики отказались от попыток определить значимые дисплеи и сделали свои **ps** полностью настраиваемыми. После небольших кастомизаций можно получить практически любой желаемый результат.

В качестве примера можно привести используемую в Linux **ps**, которая является высоко полиморфной версией. Она понимает наборы опций нескольких дистрибутивов. Почти уникальная особенность команд UNIX - способность **ps** принимать флаги командной строки с тире или без тире, но она также может назначить различные интерпретации этим формам. Например, **ps -a** - это не то же самое, что **ps a**.

Пусть вас не пугает вся эта сложность: она предназначена в основном для разработчиков, а не для системных администраторов. Хотя вы будете часто использовать **ps**, вам нужно знать только несколько конкретных "заклинаний".

Вы можете получить полезную информацию обо всех процессах, запущенных в системе с помощью **ps aux**. Опция **a** говорит **ps** показать все процессы, **x** - показать процессы, которые не имеют терминала управления; **u** выбирает формат вывода "ориентированный на пользователя". Вот пример вывода **ps aux** на машине под управлением Red Hat:

redhat$ **ps aux**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| USER | PID | %CPU | %MEM | VSZ | RSS | TTY | STAT | TIME | COMMAND |
| root  root  root  root  root  root  root  …  root  root  root  root  root  root  root  root  root  root  root  ... | 1  2  3  4  5  18  28  196  1050  1472  1646  1733  2124  2182  2186  2519  2384  2419 | 0.1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0.0  0  0 | 0.2  0  0  0  0  0  0  0  0.1  0.3  0.3  0  0.3  0.2  0.1  0.0  0.6  1.1 | 3356  0  0  0  0  0  0  0  2652  3048  3012  0  3004  2264  2952  17036  4080  7776 | 560  0  0  0  0  0  0  0  448  1008  1012  0  1008  596  484  380  1660  3004 | ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ? | S  SN  S<  S<  S<  S<  S  S  S<s  S<s  S<s  S  Ss  Ss  Ss  Ss  Ss  Ss | 0:00  0:00  0:00  0:00  0:00  0:00  0:00  0:00  0:00  0:00  0:00  0:00  0:00  0:00  0:00  0:00  0:00  0:00 | init [5]  [ksoftirqd/0]  [events/0]  [khelper]  [kacpid]  [kblockd/0]  [pdflush]  [kjournald]  udevd  /sbin/dhclient -1  /sbin/dhclient -1  [kjournald]  /sbin/dhclient -1  rsyslog -m 0  klogd -x  /usr/sbin/atd  /usr/sbin/sshd  sendmail: accept |

Имена команд в скобках на самом деле вообще не являются командами, а скорее потоками ядра, запланированными как процессы. Значение каждого поля показано в таблице 4.2.

Еще один полезный набор аргументов - это **lax**, который выводит больше технической информации. Параметры **a** и **x** приведены выше (показывает все процессы), а **l** выбирает "длинный" формат вывода. **ps** **lax** может работать немного быстрее, чем **ps aux**, потому что ей не нужно переводить каждый UID в имя пользователя—эффективность может быть важна, если система уже захлебывается.

**Таблица 4.2 - Объяснение вывода ps aux**

|  |  |
| --- | --- |
| **Поле** | **Содержание** |
| USER | Имя пользователя-владельца процесса |
| PID | ID процесса |
| %CPU | % использования CPU процессом |
| %MEM | % использования реальной памяти процессом |
| VSZ | Виртуальный размер процесса |
| RSS | Resident set size (количество страниц в памяти) |
| TTY | ID контрольного терминала |
| STAT | Текущий статус процесса:  R = запущен D = In uninterruptible sleep  S = спит (< 20 сек) T = Traced or stopped  Z = зомби  Дополнительные флаги:  W = Процесс засвопился  < = Процесс имеет больший приоритет чем обычно  N = Процесс имеет меньший приоритет чем обычно  L = Некоторые страницы заблокированы в ядре  s = Процесс является лидером сессии |
| TIME | Время CPU, затраченное на этот процесс |
| COMMAND | Имя команды и аргументыа |

1. Программы могут изменять эту информацию, так что это не обязательно точное представление фактической командной строки.

В представленном далее сокращенном примере, **ps lax** включает такие поля, как ID родительского процесса (PPID), вежливость (NI) и тип ресурса, который ожидает процесс (WCHAN, сокращение от "wait channel").

redhat$ **ps lax**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| F | UID | PID | PPID | PRI | NI | VSZ | RSS | WCHAN | STAT | TIME | COMMAND |
| 4  1  1  1  5  5  5  1  1  5  1  5  ... | 0  0  0  0  0  32  29  0  0  0  0  0 | 1  2  3  4  2186  2207  2227  2260  2336  2384  2399  2419 | 0  1  1  3  1  1  1  1  1  1  1  1 | 16  34  5  5  16  15  18  16  21  17  15  16 | 0  19  -10  -10  0  0  0  0  0  0  0  0 | 3356  0  0  0  2952  2824  2100  5668  3268  4080  2780  7776 | 560  0  0  0  484  580  760  1084  556  1660  828  3004 | select  ksofti  worker  worker  syslog  -  select  -  select  select  select  select | S  SN  S<  S<  Ss  Ss  Ss  Ss  Ss  Ss  Ss  Ss | 0:00  0:00  0:00  0:00  0:00  0:00  0:00  0:00  0:00  0:00  0:00  0:00 | init [5]  [ksoftirqd/0]  [events/0]  [khelper]  klog -x  portmap  rpc.statd  rpc.idmapd  acpid  sshd  xinetd -sta  sendmail: a |

Команды с длинными списками аргументов могут иметь отключенный вывод командной строки. Добавьте **w** в список флагов, чтобы отобразить больше столбцов в выводе. Добавьте **w** дважды для неограниченной ширины столбца, что удобно для тех процессов, которые имеют исключительно длинные аргументы командной строки, такие как некоторые приложения **java**.

Администраторам часто бывает необходимо определить PID процесса. Вы можете найти PID используя **grep** в выходных данных **ps**:

$ **ps aux | grep sshd**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| root  bwhaley | 6811  13961 | 0.0  0.0 | 0.0  0.0 | 78056  110408 | 1340  868 | ?  pts/1 | Ss  S+ | 16:04  20:37 | 0:00  0:00 | /usr/sbin/sshd  grep /usr/sbin/sshd |

Обратите внимание, что вывод **ps** включает в себя саму команду **grep**, так как **grep** была активна в списке процессов во время выполнения **ps**. Вы можете удалить эту строку из вывода с помощью **grep -v**:

$ **ps aux | grep -v grep | grep sshd**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| root | 6811 | 0.0 | 0.0 | 78056 | 1340 | ? | Ss | 16:04 | 0:00 | /usr/sbin/sshd |

Вы также можете определить PID процесса с помощью команды **pidof**:

$ **pidof /usr/sbin/sshd**

6811

Или с помощью утилиты **pgrep**:

$ **pgrep sshd**

6811

**pidof** и **pgrep** показывают все процессы, соответствующие заданному значению. Мы часто используем простой **grep** для максимальной гибкости, хотя он иногда он быть немного более подробным чем необходимо.

**4.4 Интерактивный мониторинг с помощью top**

Такие команды, как **ps**, показывают вам снимок системы, на момент выполнения команды. Часто эта ограниченная выборка оказывается недостаточной для того, чтобы передать общую картину происходящего. **top** - это своего рода динамическая **ps**, которая дает регулярно обновляемую интерактивную сводку по процессам и использование ими ресурсов. Например:

redhat$ **top**

top - 20:07:43 up 1:59, 3 users, load average: 0.45, 0.16, 0.09

Tasks: 251 total, 1 running, 250 sleeping, 0 stopped, 0 zombie

%Cpu(s): 0.7 us, 1.2 sy, 0.0 ni, 98.0 id, 0.0 wa, 0.0 hi, 0.2 si, 0.0 st

KiB Mem : 1013672 total, 128304 free, 547176 used, 338192 buff/cache

KiB Swap: 2097148 total, 2089188 free, 7960 used. 242556 avail Mem

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PID | USER | PR | NI | VIRT | RES | SHR | S | %CPU | %MEM | TIME+ | COMMAND |
| 2731  25721  25296  747  846  1647  10246  1  2  3  5  7  8  9  ... | root  ulsah  ulsah  root  root  root  ulsah  root  root  root  root  root  root  root | 20  20  20  20  20  20  20  20  20  20  0  rt  20  20 | 0  0  0  0  0  0  0  0  0  0  -20  0  0  0 | 193316  619412  260724  4372  141744  177436  130156  59620  0  0  0  0  0  0 | 34848  27216  6068  604  384  3656  1936  5472  0  0  0  0  0  0 | 15184  17636  3268  504  192  2632  1256  3348  0  0  0  0  0  0 | S  S  S  S  S  S  R  S  S  S  S  S  S  S | 1.7  1.0  0.7  0.3  0.3  0.3  0.3  0.0  0.0  0.0  0.0  0.0  0.0  0.0 | 3.4  2.7  0.6  0.1  0.0  0.4  0.2  0.5  0.0  0.0  0.0  0.0  0.0  0.0 | 0:30.39  0:03.67  0:17.78  0:02.68  0:01.74  0:04.47  0:00.10  0:02.09  0:00.02  0:00.03  0:00.00  0:00.20  0:00.00  0:00.00 | Xorg  konsole  prlcc  rngd  prltoolsd  cupsd  top  systemd  kthreadd  ksoftirqd/0  kworker/0:+  migration/0  rcu\_bh  rcuob/0 |

По умолчанию дисплей обновляется каждые 1-2 секунды, в зависимости от системы. Наиболее нагружающие процессор процессы появляются в верхней части. **top** также принимает ввод с клавиатуры для отправки сигналов и проведения **renice** для процессов (см. Следующий раздел). Вы можете наблюдать за тем, как ваши действия влияют на общее состояние системы.

*Вы можете узнать, как интерпретировать информацию о процессоре, памяти и нагрузке в Главе 29.*

Сводная информация в первых нескольких строках верхнего вывода - это одно из первых мест, на которое следует обратить внимание для анализа состояния системы. В сжатом виде она показывает загрузку системы, использование памяти, количество процессов и сортирует их по использованию процессора.

В многоядерных системах загрузка ЦП показывает среднюю нагрузку для всех ядер в системе. В Linux нажмите 1, когда открыт **top**, чтобы переключиться на режим отображения отдельных ядер. Во FreeBSD запустите **top -P** для достижения того же эффекта6.

*6. В системах FreeBSD вы можете установить переменную окружения TOP для передачи дополнительных аргументов в* ***top****. Мы рекомендуем использовать* ***-H****, чтобы показать все потоки для многопоточных процессов, и* ***-P****, чтобы отобразить все ядра процессора. Добавьте export TOP= "- HP" в файл инициализации shell, чтобы сделать эти изменения постоянными.*

Root может запускать **top** с опцией **-q**, чтобы поднять его приоритет до максимально возможного. Эта опция может быть полезна, когда вы пытаетесь отследить процесс, который уже поставил систему на колени.

Нам также нравится **htop**, открытый, кросс-платформенный, интерактивный просмотрщик процессов, который предлагает больше возможностей и имеет более приятный интерфейс, чем у **top**. Он еще не доступен в виде пакета для наших примеров систем, но вы можете скачать бинарный файл или исходный код с веб-сайта разработчика по адресу hisham.hm/htop.

**4.5 nice и renice: приоритет планирования влияния**

"Вежливость" процесса - это числовая подсказка ядру о том, как этот процесс следует рассматривать по отношению к другим процессам, претендующим на использование CPU7. Такое странное название происходит от того, что оно определяет, насколько хорошо вы будете относиться к другим пользователям системы. Высокая вежливость означает низкий приоритет для вашего процесса: вы достаточно вежливы. Низкое или отрицательное значение означает высокий приоритет: вы не вежливы.

*7.* ***nice*** *управляет только приоритетом процессора. Чтобы установить приоритет устройств ввода-вывода, используйте* ***ionice****.*

В наши дни крайне необычно расставлять приоритеты вручную. В слабых системах, во времена появления UNIX, производительность в значительной степени зависела от того, какой процесс использовал процессор. Сегодня, когда настольные компьютеры имеют более чем достаточно мощные процессоры, планировщик выполняет хорошую работу по управлению большинством рабочих нагрузок. Добавление классов планирования дает разработчикам дополнительный контроль, когда необходимо быстро получать отклики.

Диапазон допустимых значений вежливости варьируется в разных системах по-разному. В Linux этот диапазон составляет от -20 до +19, а во FreeBSD - от -20 до +20.

Если пользователь не предпримет специальных действий, вновь созданный процесс наследует вежливость своего родительского процесса. Владелец процесса может увеличить его вежливость, но не может понизить ее, даже чтобы вернуть процесс к значению вежливости по умолчанию. Это ограничение не позволяет процессам, запущенным с низким приоритетом, иметь высокоприоритетные дочерние элементы. Однако суперпользователь может устанавливать данные значения как угодно.

Производительность устройств ввода-вывода не успевает за все более быстрыми процессорами. Даже с современными высокопроизводительными SSD пропускная способность диска остается бутылочным горлышком в большинстве систем. К сожалению, вежливость процесса никак не влияет на управление ядром его памятью или I/O; высокоприоритетные процессы все еще могут монополизировать непропорционально большую долю этих ресурсов.

Вежливость процесса может быть установлена во время создания с помощью команды **nice** и скорректирована позже с помощью команды **renice**. **nice** принимает значения командной строки в качестве аргумента, а **renice** принимает PID или (иногда) имя пользователя.

Примеры:

|  |  |
| --- | --- |
| $ **nice -n 5 ~/bin/longtask**  $ **sudo renice -5 8829**  $ **sudo renice 5 -u boggs** | // Понижает приоритет (увеличивает вежливость) на 5  // Устанавливает вежливость в -5  // Устанавливает вежливость процессов boggs в 5 |

К сожалению, между системами существует мало соглашений относительно того, как следует определять желаемые приоритеты; на самом деле даже **nice** и **renice** из одной и той же системы обычно не имеют соглашений. Чтобы усложнить ситуацию, **nice** встроена в C shell и некоторые другие распространенные оболочки (но не в **bash**). Если вы не введете полный путь к **nice**, вы запустите версию **nice** из оболочки, а не операционной системы. Чтобы обойти эту двусмысленность, мы предлагаем использовать полный путь к системной версии **nice**, находящейся по пути **/usr/bin/nice**.

В таблице 4.3 приводится сводная информация о различных вариациях вежливости. *prio* является абсолютной вежливостью, а *incr* отображает вежливость в оболочке, из которой запускаются **nice** и **renice**. Только **nice** оболочки понимает знак плюс (на самом деле, она его требует); в других случаях он не требуется.

**Таблица 4.3 как выразить приоритеты в nice и renice**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Система** | **Пределы** | **nice в ОС** | **csh nice** | **renice** |
| Linux | от -20 до 19 | -n incr | +incr или -incr | prio или -n prio |
| FreeBSD | от -20 до 20 | -n incr | +incr или -incr | incr или -n incr |

**4.6 Файловая система /proc**

Linux-версии **ps** и **top** считывают информацию о состоянии своих процессов из каталога **/proc**, псевдофайловой системы, в которой ядро предоставляет разнообразную интересную информацию о состоянии системы.

Несмотря на имя **/proc** (и имя базового типа файловой системы "proc"), данные не ограничиваются информацией о процессе — здесь представлены различные сведения о состоянии и статистические данные, генерируемые ядром. Вы даже можете изменить некоторые параметры, записав их в соответствующий файл **/proc**. Некоторые примеры см. на стр. 339.

Хотя большую часть информации проще всего получить через команды **vmstat** и **ps**, некоторые из наиболее непонятных моментов должны быть прочитаны непосредственно из **/proc**. Стоит покопаться в этой директории, чтобы ознакомиться со всем, что там есть. **man proc** содержит исчерпывающую документацию.

Поскольку ядро создает содержимое файлов **/proc** на лету (по мере их чтения), большинство из них кажутся пустыми, 0-байтовыми файлами, когда они перечислены с помощью **ls -l**. Вам придется использовать **cat** или **less** чтобы увидеть, что они на самом деле содержат. Но будьте осторожны — некоторые файлы содержат или ссылаются на двоичные данные, которые могут сбить с толку ваш эмулятор терминала при непосредственном просмотре.

Информация, относящаяся к процессам, делится на подкаталоги, именуемые PID этих процессов. Например, **/proc/1** всегда является каталогом, содержащим информацию об **init**. В таблице 4.4 перечислены наиболее полезные файлы каждого процесса.

**Таблица 4.4 - Файлы с информацией о процессе в папке /proc в Linux (нумерованные подкаталоги)**

|  |  |
| --- | --- |
| **Файл** | **Содержание** |
| cgroup | Контрольные группы, к которым относится процесс |
| cmd | Команда или программа, которую выполняет процесс |
| cmdlineа | Содержит команду, которая первоначально запустила процесс |
| cwd | Символическая ссылка на текущий рабочий каталог процесса |
| environ | Переменные окружения процесса (разделенные нулем) |
| exe | Символическая ссылка на оригинальный исполняемый файл |
| fd | Подкаталог, содержащий ссылки для каждого открытого файлового дескриптора |
| fdinfo | Подкаталог, содержащий дополнительную информацию для каждого открытого файлового дескриптора |
| maps | Информация о сопоставлении памяти (общие сегменты, библиотеки и т. д ) |
| ns | Подкаталог со ссылками на каждое пространство имен, используемое процессом. |
| root | Символическая ссылка на корневой каталог процесса (устанавливается с помощью **chroot**) |
| stat | Общая информация о состоянии процесса (лучше всего декодируется с помощью **ps**) |
| statm | Информация об использовании памяти |

1. Может быть недоступна, если процесс засвопился

Отдельные компоненты, содержащиеся в файлах **cmdline** и **environ**, разделяются нулевыми символами, а не новыми строками. Вы можете фильтровать их содержимое через **tr "\000" "\n"**, чтобы сделать их более удобочитаемыми.

Подкаталог **fd** содержит открытые файлы в виде символьных ссылок. Файловые дескрипторы, подключенные к конвейерам или сетевым сокетам, не имеют соответствующего имени файла. Вместо этого ядро предоставляет общее описание в качестве целевой ссылки.

Файл **maps** может быть полезен для определения библиотек, с которыми связана программа или зависит от них.

FreeBSD включает в себя аналогичную, но отличающуюся реализацию **/proc**. Однако его использование было признано ненадежным из-за ошибок в коде и исторических проблем с безопасностью. Он по-прежнему доступен для совместимости, но по умолчанию не монтируется. Чтобы смонтировать его, используйте команду8

|  |
| --- |
| freebsd$ **sudo mount -t procfs proc /proc** |

*8. Чтобы автоматически смонтировать файловую систему* ***/proc*** *во время загрузки, добавьте строку proc /proc procfs rw 0 0 в файл /etc/fstab*

Компоновка файловой системы аналогична (но не идентична) Linux-версии procfs. Информация для процесса включает его статус, символическую ссылку на исполняемый файл, сведения о виртуальной памяти процесса и другую низкоуровневую информацию. Смотрите также **man procfs**.

**4.7 STRACE и TRUSS: трассировка сигналов и системные вызовы**

Часто бывает трудно понять, что же на самом деле делает тот или иной процесс. Первый шаг обычно состоит в том, чтобы сделать обоснованное предположение, основанное на косвенных данных, собранных из файловой системы, журналов и инструментов, таких как **ps**.

Если эти источники информации окажутся недостаточными, вы можете следить за процессом на более низком уровне с помощью команды **strace** (в Linux; обычно дополнительный пакет) или **truss** (FreeBSD). Эти команды отображают каждый системный вызов, который делает процесс, и каждый сигнал, который он получает. Вы можете прикрепить **strace** или **truss** к запущенному процессу, понаблюдать некоторое время, а затем отсоединиться от процесса, не беспокоя его9.

*9. Ну, как правило.* ***strace*** *может прерывать системные вызовы. Затем отслеживаемый процесс должен быть готов к их перезапуску. Это стандартное правило гигиены ПО в UNIX, но оно не всегда соблюдается.*

Несмотря на то, что системные вызовы происходят на относительно низком уровне абстракции, вы обычно можете довольно много рассказать об активности процесса из трассировки вызовов. Например, следующий отчет был подготовлен **strace** запущенном в активной копии **top** (которая работает с PID 5810):

redhat$ **sudo strace -p 5810**

|  |  |
| --- | --- |
| gettimeofday( {1116193814, 213881}, {300, 0} )  open("/proc", O\_RDONLY|O\_NONBLOCK|O\_LARGEFILE|O\_DIRECTORY)  fstat64(7, {st\_mode=S\_IFDIR|0555, st\_size=0, ...} )  fcntl64(7, F\_SETFD, FD\_CLOEXEC)  getdents64(7, /\* 36 entries \*/, 1024)  getdents64(7, /\* 39 entries \*/, 1024)  stat64("/proc/1", {st\_mode=S\_IFDIR|0555, st\_size=0, ...} )  open("/proc/1/stat", O\_RDONLY)  read(8, "1 (init) S 0 0 0 0 -1 4194560 73"..., 1023)  close(8) | = 0  = 7  = 0  = 0  = 1016  = 1016  = 0  = 8  = 191  = 0 |

**strace** не только показывает вам имя каждого системного вызова, выполненного процессом, но также декодирует аргументы и показывает код результата, который возвращает ядро.

В приведенном выше примере **top** начинает работу с проверки текущего времени. Затем он открывает и регистрирует каталог **/proc** и читает содержимое каталога, получая таким образом список запущенных процессов. **top** переходит к stat каталогу, представляющему процесс **init**, а затем открывает **/proc/1/stat**, чтобы прочитать информацию о состоянии **init**.

Вывод системного вызова часто может выявить ошибки, о которых не сообщает сам процесс. Например, ошибки разрешения файловой системы или конфликты сокетов обычно довольно очевидны в выходных данных **strace** или **truss**. Ищите системные вызовы, которые возвращают ошибки, и проверяйте ненулевые значения.

**strace** содержит много полезных вещей, большинство из которых задокументировано в man-странице. Например, флаг **-f** отслеживает форкнутые процессы. Эта функция полезна для отслеживания демонов (таких как httpd), которые порождают много потомков. Параметр **-e trace=file** отображает только операции, связанные с файлами. Эта функция особенно удобна для обнаружения местоположения неуловимых конфигурационных файлов.

Вот аналогичный пример из FreeBSD, который использует **truss**. В этом случае мы проследим, как **cp** копирует файл:

freebsd$ **truss cp /etc/passwd /tmp/pw**

…

|  |  |
| --- | --- |
| lstat("/etc/passwd",{ mode=-rw-r--r-- ,inode=13576,size=2380, blksize=4096 })  umask(0x1ff)  umask(0x12)  fstatat(AT\_FDCWD,"/etc/passwd",{ mode=-rw-r--r-- ,inode=13576,size=2380,blksize=4096 },0x0)  stat("/tmp/pw",0x7fffffffe440) ERR#2 'No such file or directory'  openat(AT\_FDCWD,"/etc/passwd",O\_RDONLY,00)  openat(AT\_FDCWD,"/tmp/pw",O\_WRONLY|O\_CREAT,0100644)  mmap(0x0,2380,PROT\_READ,MAP\_SHARED,3,0x0)  write(4,"# $FreeBSD: releng/11.0/etc/mast"...,2380)  close(4)  close(3) | = 0 (0x0)  = 18 (0x12)  = 511 (0x1ff)  = 0 (0x0)  = 3 (0x3)  = 4 (0x4)  = 34366304256 (0x800643000)  = 2380 (0x94c)  = 0 (0x0)  = 0 (0x0) |

...

После выделения памяти и открытия библиотечных зависимостей (не показаны) **cp** использует системный вызов **lstat** для проверки текущего состояния файла **/etc/passwd**. Затем он запускает **stat** по пути предполагаемой копии, **/tmp/pw**. Этот файл еще не существует, поэтому **stat** отрабатывает с ошибкой и **truss** декодирует ошибку для вас как "Нет такого файла или каталога".

Затем **cp** вызывает системный вызов **openat** (с параметром O\_RDONLY) для чтения содержимого файла **/etc/passwd**, а затем **openat** **/tmp/pw** с параметром O\_WRONLY для создания нового файла назначения. Затем он переносит содержимое файла **/etc/passwd** в память (с помощью **mmap**) и записывает данные с помощью **write**. Наконец, **cp** подчищает за собой путем закрытия обоих обработчиков файлов.

Трассировка системных вызовов - это мощный инструмент отладки для администраторов. Обратитесь к этим инструментам после того, как более традиционные способы, такие как изучение файлов журналов и настройка процесса для большего вывода, будут исчерпаны. Пусть вас не пугает огромный и нечитаемый output; обычно достаточно сосредоточиться на удобочитаемых фрагментах.

**4.8 Неконтролируемые процессы**

"Неконтролируемые" процессы - это те, которые поглощают значительно больше ресурсов процессора, диска или сети системы, чем можно было бы ожидать от их обычной роли или поведения. Иногда такие программы имеют свои собственные баги, которые привели к дегенеративному поведению. В других случаях они не справляются должным образом с вышестоящими сбоями и зацикливаются. Например, процесс может снова и снова пытаться выполнить одну и ту же неудачную операцию, приводя к исчерпанию ресурсов процессора. В еще одной категории случаев ошибки как таковой нет, но программное обеспечение просто неэффективно в своей реализации и жадно расходует ресурсы системы.

Все эти ситуации заслуживают изучения системным администратором не только потому, что неконтролируемый процесс, скорее всего, неисправен, но и потому, что он обычно влияет на работу других процессов, запущенных в системе.

Грань между патологическим и нормальным поведением при большой нагрузке размыта. Часто первым шагом в диагностике является выяснение того, какое из этих явлений вы наблюдаете на самом деле. Как правило, системные процессы всегда должны вести себя разумно, поэтому очевидное неправильное поведение со стороны одного из этих процессов автоматически вызывает подозрение. Пользовательские процессы, такие как веб-серверы и базы данных, могут быть просто перегружены.

Вы можете определить процессы, которые чрезмерно используют время процессора, посмотрев на выходные данные **ps** или **top**. Также проверьте средние значения загрузки системы, о которых сообщает команда **uptime**. Традиционно эти значения количественно определяют среднее число процессов, которые были запущены за предыдущие 1-, 5- и 15-минутные интервалы. В Linux средняя нагрузка также учитывает загрузку, вызванную дисковым трафиком и другими устройствами ввода-вывода.

Для систем, связанных с ЦП, средние значения нагрузки должны быть меньше общего числа ядер ЦП, доступных в вашей системе. Если это не так, то система перегружена. В Linux проверьте общую загрузку ЦП с помощью **top** или **ps**, чтобы определить, связаны ли высокие значения средней нагрузки с утилизацией ЦП или с устройствами ввода-вывода. Если загрузка ЦП близка к 100%, вероятно, именно он является узким местом.

Процессы, которые используют чрезмерное количество памяти относительно физической оперативной памяти системы, могут вызвать серьезные проблемы с производительностью. Вы можете проверить объем памяти процессов, запустив **top**. Столбец VIRT показывает общий объем виртуальной памяти, выделенной каждому процессу, а столбец RES показывает часть этой памяти, которая в настоящее время сопоставлена с определенными страницами памяти ("резидентный набор").

Оба эти числа могут включать общие ресурсы, такие как библиотеки, и поэтому потенциально могут вводить в заблуждение. Более прямое измерение потребления памяти для конкретного процесса находится в столбце DATA, который по умолчанию не отображается. Чтобы добавить этот столбец на дисплей **top**, нажмите клавишу **f** после запуска **top** и выберите DATA из списка, нажав пробел. Значение DATA указывает объем памяти в сегментах данных и стека каждого процесса, поэтому оно относительно специфично для отдельных процессов (сегменты общей памяти по модулю). Ищите рост с течением времени, а также абсолютный размер. Эквивалентным столбцом во FreeBSD является SIZE и отображается по умолчанию.

Сделайте согласованные усилия, чтобы понять, что происходит, прежде чем остановить, казалось бы, безудержный процесс. Лучший способ отладить проблему и предотвратить ее повторение - иметь живой пример, который вы можете исследовать. После того, как вы убьете неправильный процесс, большинство доступных улик исчезнет.

Также помните о возможности взлома. Вредоносное программное обеспечение обычно не проверяется на корректность в различных средах, поэтому вероятность того, что оно войдет в какое-то дегенеративное состояние, выше среднего. Если вы подозреваете неправомерную деятельность, сделайте трассировку системного вызова с помощью **strace** или **truss**, чтобы получить представление о том, что делает процесс (например, перебирает пароли) и где хранятся его данные.

Неконтролируемые процессы, у которых есть output, могут заполнить всю файловую систему, вызывая многочисленные проблемы. Когда файловая система заполняется, на консоль записывается множество сообщений, и при попытках записи в файловую систему появляются сообщения об ошибках.

Первое, что нужно сделать в этой ситуации, это определить, какая файловая система заполнена и какой файл ее заполняет. Команда **df -h** показывает использование диска с файловой системой в удобном виде. Найдите файловую систему, которая заполнена на 100% или более10. Используйте команду **du -h** в найденной файловой системе, чтобы определить, какой каталог занимает больше места. Повторите **du** в "большом" каталоге чтобы найти большие файлы.

*10. Большинство реализаций файловых систем резервируют часть (около 5%) дискового пространства для* "*передышки*"*, но процессы, запущенные с правами root, могут вторгаться в это пространство, что приводит к использованию, о котором сообщается, что оно более 100%.*

**df** и **du** сообщают об использовании диска немного по-разному. **df** сообщает о дисковом пространстве, используемом смонтированной файловой системой, в соответствии с общим количеством блоков диска в метаданных файловой системы. **du** суммирует размеры всех файлов в указанном каталоге. Если файл не связан (удален) с файловой системой, но на него все еще ссылается какой-то запущенный процесс, **df** учитывает его размер, а **du** - нет. Это несоответствие сохраняется до тех пор, пока дескриптор открытого файла не будет закрыт или файл не будет удален. Если вы не можете определить, какой процесс использует файл, попробуйте выполнить команды **fuser** и **lsof** (подробно описано на стр. 124), чтобы получить больше информации.

**4.9 Периодические процессы**

Часто бывает полезно выполнить скрипт или команду без вмешательства человека. Распространенные случаи использования включают резервное копирование по расписанию, обслуживание базы данных или выполнение пакетных заданий на ночь. Как это типично для UNIX и Linux, существует более одного способа достижения этой цели.

**cron: выполнение команд по расписанию**

Демон **cron** - это традиционный инструмент для запуска команд по заданному расписанию. Он запускается при загрузке системы и работает все время, пока работает система. Существует несколько реализаций **cron**, но, к счастью для администраторов, синтаксис и функциональность различных версий практически идентичны.

По непонятным причинам **cron** был переименован в **crond** в Red Hat. Но это все тот же самый **cron**, который мы все знаем и любим.

**cron** читает файлы конфигурации, содержащие списки командных строк и время, когда они должны быть вызваны. Командные строки выполняются с помощью **sh**, поэтому почти все, что вы можете сделать вручную из оболочки, также можно сделать с помощью **cron**. Вы можете даже настроить **cron** для использования другой оболочки, если пожелаете.

Файл конфигурации **cron** называется "crontab", сокращенно от "cron table". Crontab для отдельных пользователей хранятся в **/var/spool/cron** (Linux) или **/var/cron/tabs** (FreeBSD). В основном для каждого пользователя существует один файл crontab. Файлы crontab - это простые текстовые файлы, названные именами пользователей, которым они принадлежат. **cron** использует эти имена файлов (и владельца файла), чтобы выяснить, какой UID использовать при выполнении команд, содержащихся в каждом файле. Команда **crontab** передает файлы crontab в и из этого каталога.

**cron** пытается свести к минимуму время, которое он тратит на повторную обработку файлов конфигурации и вычисление времени. Команда **crontab** помогает поддерживать эффективность **cron**, уведомляя **cron** об изменении файлов crontab. Поэтому вы не должны редактировать файлы crontab напрямую, потому что такой подход может привести к тому, что **cron** не заметит ваши изменения. Если вы попадаете в ситуацию, когда **cron** не замечает измененный crontab - отправьте сигнал HUP процессу **cron**, который насильно заставит его перезагрузиться в большинстве систем.

Обычно **cron** тихо выполняет свою работу, без вывода сообщений, но большинство его версий могут записывать действия в файл журнала (обычно в **/var/log/cron**), в котором перечислены команды, которые были выполнены, и время, в которое они выполнялись. Посмотрите файл журнала **cron**, если у вас возникли проблемы с заданием **cron**, и вы не можете понять причину их появления.

*См. Главу 10 для получения дополнительной информации о syslog.*

*Формат файлов crontab*

Все файлы crontab в системе имеют одинаковый формат. Комментарии начинаются со знака # в начале строки. Каждая строка без комментариев содержит шесть полей и представляет одну команду:

*минута час дни\_месяца месяц день\_недели команда*

Первые пять полей сообщают **cron**, когда нужно выполнить команду. Они разделены пробелами, но внутри поля команды пробелы передаются в оболочку. Поля в спецификации времени интерпретируются, как показано в таблице 4.5. Запись в crontab в разговорной речи называется "задание cron".

**Таблица 4.5 спецификации времени crontab**

|  |  |
| --- | --- |
| **Поле** | **Диапазон** |
| *минута* | от 0 до 59 |
| *час* | от 0 до 23 |
| *дни месяца* | от 1 до 31 |
| *месяц* | от 1 до 12 |
| *день недели* | от 0 до 6 (0 = Воскресенье) |

Каждое из связанных со временем полей может содержать:

* Звездочку, которая соответствует всем значениям
* Одно целое число, которое соответствует одному значению
* Два целых числа, разделенные чертой, соответствующие диапазону значений
* Диапазон, за которым следует косая черта и значение шага, например, 1-10/2
* Разделенный запятыми список целых чисел или диапазонов, соответствующих любому значению

Например, указание времени:

|  |
| --- |
| 45 10 \* \* 1-5 |

означает "10:45 утра, с понедельника по пятницу". Подсказка: никогда не используйте звездочки в каждом поле, если вы не хотите, чтобы команда запускалась каждую минуту, что полезно только в сценариях тестирования. Одна минута - минимальное значение гранулярности, доступное для заданий **cron**.

Временные диапазоны в файлах crontab могут включать значение шага. Например, ряды 0,3,6,9,12,15,18 можно записать более кратко, как 0-18/3. Вы также можете использовать трехбуквенную текстовую мнемонику для названий месяцев и дней, но не в сочетании с диапазонами. Насколько нам известно, эта функция работает только с английскими именами.

Существует потенциальная двусмысленность, за которой нужно следить в полях дни недели и день месяца. Каждый день является и днем недели, и днем месяца. Если указаны и день недели и день месяца, то день должен удовлетворять только одному из двух условий, из которых необходимо выбрать.

Например:

|  |
| --- |
| 0,30 \* 13 \* 5 |

означает "каждые полчаса в пятницу и каждые полчаса каждое 13-е число месяца", а не "каждые полчаса в пятницу 13-го".

Команда - это командная строка **sh**, которая должна быть выполнена. Это может быть любая допустимая shell команда, и ее не следует заключать в кавычки. Считается, что команда продолжается до конца строки и может содержать пробелы или символы табуляции.

Знаки процента (%) указывают на новые строки в поле команды. Только текст до первого знака процента фактически включен в команду. Остальные строки передаются команде в качестве стандартного input’а. Используйте обратную косую черту (\) в качестве escape-символа в командах, имеющих знак процента, например, *date +* \*% s*.

Хотя **sh** задействован в выполнении команды, shell не действует как оболочка входа в систему и не читает содержимое **~/.profile** или **~/.bash\_profile**. В результате переменные окружения команды могут быть настроены несколько иначе, чем вы от них ожидаете. Если кажется, что команда работает нормально при выполнении из оболочки, но при вводе в файл crontab происходит сбой, вероятным виновником являются переменные. При необходимости вы всегда можете обернуть команду скриптом, который устанавливает соответствующие переменные.

Мы также предлагаем использовать полный путь к команде, гарантируя, что задание будет работать должным образом, даже если PATH не установлен должным образом. Например, следующая команда записывает дату и время работы в файл в домашнем каталоге пользователя каждую минуту:

|  |
| --- |
| \* \* \* \* \* echo $(/bin/date) - $(/usr/bin/uptime) >> ~/uptime.log |

Кроме того, вы можете явно установить переменные окружения в верхней части crontab:

|  |
| --- |
| PATH=/bin:/usr/bin  \* \* \* \* \* echo $(date) - $(uptime) >> ~/uptime.log |

Вот еще несколько примеров допустимых записей в crontab:

|  |
| --- |
| \*/10 \* \* \* 1,3,5 echo ruok | /usr/bin/nc localhost 2181 |  mail -s "TCP port 2181 status" ben@admin.com |

Эта строка отправляет по электронной почте результаты проверки подключения на порт 2181 каждые 10 минут по понедельникам, средам и пятницам. Поскольку **cron** выполняет поле *команда* с помощью **sh**, специальные символы оболочки, такие как вертикальная черта и перенаправления, функционируют должным образом.

|  |
| --- |
| 0 4 \* \* Sun (/usr/bin/mysqlcheck -u maintenance --optimize  --all-databases) |

Эта запись запускает программу обслуживания **mysqlcheck** по воскресеньям в 4:00 утра. Поскольку выходные данные не сохраняются в файл или отбрасываются другим способом, они будут отправлены владельцу crontab по электронной почте.

|  |
| --- |
| 20 1 \* \* \* find /tmp -mtime +7 -type f -exec rm -f { } ';' |

Эта команда выполняется в 1:20 каждое утро. Она удаляет все файлы в каталоге **/tmp**, которые не были изменены в течение 7 дней. ';' в конце строки отмечает конец аргументов подкоманды **find**.

**cron** не пытается перезапускать команды, пропущенные во то время, когда система была выключена или не работала по иным причинам. Тем не менее, он достаточно умен чтобы корректировать время, например, переходить на летнее время и обратно.

Если ваша задача cron является скриптом, убедитесь, что он является исполняемым (с помощью **chmod +x**), иначе **cron** не сможет его выполнить. В качестве альтернативы, установите команду cron для непосредственного вызова оболочки в вашем скрипте (например, **bash -c ~/bin/myscript.sh**).

*Управление crontab*

**crontab** *filename* устанавливает новый crontab файл, заменяя любую предыдущую версию. **crontab -e** извлекает копию вашего crontab, вызывает ваш редактор (какой именно, указано в переменной среды EDITOR), а затем повторно отправляет его в каталог crontab. **crontab -l** выводит содержимое вашего crontab в stdout, а **crontab -r** удаляет его, оставляя вас вообще без файла crontab.

Root может предоставить аргумент имени пользователя для редактирования или просмотра файлов crontab других пользователей. Например, **crontab -r jsmith** удаляет crontab, принадлежащий пользователю jsmith, а **crontab -e jsmith** редактирует его. Linux допускает использование как имени пользователя, так и имени файла в качестве аргумента в одной и той же команде, поэтому перед именем пользователя необходимо указать префикс **-u** (например, **crontab -u jsmith crontab.new**).

Без аргументов командной строки большинство версий **crontab** пытаются прочитать crontab из стандартного input’а. Если вы войдете в этот режим случайно, не пытайтесь выйти с помощью <Control-D>; это сотрет весь ваш crontab. Вместо этого используйте <Control-C>. FreeBSD требует, чтобы вы указали дефис в качестве аргумента имени файла, чтобы **crontab** обращал внимание на его стандартный input. Умно.

Многие сайты испытывают незначительные, но периодически возникающие сбои в работе сети, возникающие из-за того, что администраторы настроили **cron** на одновременное выполнение одной и той же команды на сотнях машин, что вызывает задержки или чрезмерную загрузку. Синхронизация часов с NTP усугубляет проблему. Эту проблему легко исправить с помощью скриптов со случайной задержкой.

**cron** регистрирует свои действия с помощью syslog, используя средство "cron", при этом большинство сообщений отправляется на уровне "info". Настроенный по умолчанию syslog обычно отправляет данные логов **cron** в свой собственный файл.

*Другие файлы crontab*

В дополнение к поиску пользовательских crontab, **cron** также подчиняется системным записям crontab, находящимся по пути **/etc/crontab** и **/etc/cron.d**. Эти файлы имеют немного иной формат, чем файлы crontab пользователей: они позволяют запускать команды от произвольных пользователей. Дополнительное поле *username* находится перед именем команды. Поле *username* отсутствует в различных версиях файлов crontab, поскольку имя файла crontab содержит ту же информацию.

В общем, **/etc/crontab** - это файл, который системные администраторы должны поддерживать вручную, тогда как **/etc/cron.d** - это своего рода хранилище, в которое пакеты программного обеспечения могут устанавливать любые записи crontab, которые им могут понадобиться. Файлы в **/etc/cron.d** по соглашению называются по имени пакетов, которые их устанавливают, но cron не заботится о данном соглашении и не применяет его.

Дистрибутивы Linux также предварительно устанавливают записи crontab, которые запускают скрипты из набора хорошо известных каталогов, тем самым предоставляя программным пакетам еще один способ устанавливать периодические задания без какого-либо редактирования файла crontab. Например, скрипты в **/etc/cron.hourly**, **/etc/cron.daily** и **/etc/cron.weekly** запускаются ежечасно, ежедневно и еженедельно, соответственно.

***cron****, контроль доступа*

Два файла конфигурации указывают, какие пользователи могут работать с файлами crontab. Для Linux это файлы **/etc/cron.{allow,deny}**, а для FreeBSD - **/var/cron/{allow,deny}**. Многие стандарты безопасности требуют, чтобы файлы crontab были доступны только для сервисных учетных записей или пользователей с законными потребностями бизнеса. Файлы **allow** и **deny** облегчают соблюдение этих требований.

Если файл **cron.allow** существует, то он содержит список всех пользователей, которым разрешено работать с crontab, по одному в строке. Ни один человек, не включенный в список, не может вызвать команду **crontab**. Если файл cron.allow не существует, проверяется файл **cron.deny**. Он тоже содержит список пользователей, но его значение меняется на противоположное: всем, кроме перечисленных пользователей, разрешен доступ.

Если не существует ни **cron.allow** ни **cron.deny**, система по умолчанию (по-видимому, наугад, поскольку нет доминирующего соглашения) либо разрешает всем пользователям работать с crontab, либо предоставляет доступ к crontab только для root. На практике начальная конфигурация обычно включается при стандартной установке ОС, поэтому вопрос о том, как **crontab** может вести себя без файлов конфигурации, является спорным. Большинство конфигураций по умолчанию разрешают всем пользователям доступ к **cron**.

Важно отметить, что в большинстве систем управление доступом осуществляется **crontab**, а не **cron**. Если пользователь может вставить файл crontab в соответствующий каталог другими способами, **cron** будет слепо выполнять содержащиеся в нем команды. Поэтому крайне важно поддерживать права root для **/var/spool/cron** и **/var/cron/tabs**. Дистрибутивы ОС по умолчанию всегда правильно устанавливают разрешения.

**Таймеры systemd**

*См. Главу 2, для введения в systemd и юниты*

В соответствии со своей задачей дублировать функции всех других подсистем Linux, **systemd** включает в себя концепцию таймеров, которые активируют службу **systemd** по заранее заданному расписанию. Таймеры более мощные, чем записи в crontab, но они также более сложны в настройке и управлении. Некоторые дистрибутивы Linux (например, CoreOS) полностью отказались от **cron** в пользу таймеров **systemd**, но все наши примеры систем все еще включают в своем составе **cron** и используют его по умолчанию.

У нас нет полезных советов относительно выбора между таймерами **systemd** и записями crontab. Используйте то, что вам нравится больше. К сожалению, у вас нет реальной возможности стандартизировать использование одной или другой системы, потому что пакеты программного обеспечения добавляют свои задания в произвольную систему по своему выбору. Вам всегда нужно проверять обе системы, когда вы пытаетесь выяснить, как выполняется конкретное задание.

*Структура таймеров* ***systemd***

Таймер **systemd** состоит из двух файлов:

* Таймер юнит, который описывает расписание и юнит для активации
* Сервис юнит, который определяет детали того, что запускать

В отличие от записей crontab, таймеры **systemd** могут быть описаны как в абсолютных значениях календаря ("среда в 10:00 утра"), так и в относительных значениях ("30 секунд после загрузки системы"). Опции могут комбинироваться, чтобы создать мощные выражения, которые не подвержены тем же ограничениям, что и задания **cron**. Таблица 4.6 описывает параметры временных выражений.

**Таблица 4.6 Типы таймеров systemd**

|  |  |
| --- | --- |
| **Тип** | **Основа времени** |
| OnActiveSec | Относительно времени, в которое активируется сам таймер |
| OnBootSec | Относительно времени загрузки системы |
| OnStartupSec | Относительно времени запуска **systemd** |
| OnUnitActiveSec | Относительно последнего активного периода активности указанного unit’а |
| OnUnitInactiveSec | Относительно последнего неактивного периода активности указанного unit’а |
| OnCalendar | Конкретный день и время |

Как следует из имен, значения для этих параметров таймера даны в секундах. Например, OnActiveSec = 30 составляет 30 секунд после активации таймера. Фактически это может быть любое допустимое выражение времени systemd (более подробно обсуждается, начиная со стр. 116).

*Пример таймера* ***systemd***

Red Hat и CentOS включают в свой состав предварительно сконфигурированный таймер **systemd**, который очищает системные временные файлы один раз в день. Ниже мы более подробно рассмотрим пример. Сначала мы перечислим все существующие таймеры командой **systemctl**. (Мы повернули таблицу вывода ниже, чтобы сделать ее читабельной. Обычно каждый таймер имеет одну длинную строку вывода).

redhat$ **systemctl list-timers**

|  |  |
| --- | --- |
| NEXT  LEFT  LAST  PASSED  UNIT  ACTIVATES | Sun 2017-06-18 10:24:33 UTC  18h left  Sat 2017-06-17 00:45:29 UTC  15h ago  systemd-tmpfiles-clean.timer  systemd-tmpfiles-clean.service |

Output содержит как имя модуля таймера, так и имя юнит-файла сервиса, который он активирует. Так как это системный таймер по умолчанию, юнит-файл находится в стандартном каталоге юнита systemd, **/usr/lib/systemd/system**. Рассмотрим юнит-файл таймера:

redhat$ **cat /usr/lib/systemd/system/systemd-tmpfiles-clean.timer**

|  |
| --- |
| [Unit]  Description=Daily Cleanup of Temporary Directories  [Timer]  OnBootSec=15min  OnUnitActiveSec=1d |

Сначала таймер активируется через 15 минут после загрузки, а затем срабатывает один раз в день. Обратите внимание, что всегда необходим какой-либо триггер для начальной активации (здесь OnBootSec). Не существует единой спецификации, которая сама по себе обеспечивает эффект "каждые X минут".

Проницательные наблюдатели заметят, что таймер на самом деле не указывает, какой юнит запускать. По умолчанию **systemd** ищет юнит-файл сервиса, имя которого совпадает с именем таймера. Вы можете указать целевой юнит в явном виде с помощью опции Unit.

В данном случае соответствующий юнит сервиса не содержит никаких сюрпризов:

redhat$ **cat /usr/lib/systemd/system/systemd-tmpfiles-clean.service**

|  |
| --- |
| [Unit]  Description=Cleanup of Temporary Directories  DefaultDependencies=no  Conflicts=shutdown.target  After=systemd-readahead-collect.service systemd-readahead-replay.service  local-fs.target time-sync.target  Before=shutdown.target  [Service]  Type=simple  ExecStart=/usr/bin/systemd-tmpfiles --clean  IOSchedulingClass=idle |

Вы можете запустить целевой сервис напрямую (то есть независимо от таймера) с помощью **systemctl start systemd-tmpfiles-clean**, как и любой другой сервис. Этот факт значительно облегчает отладку запланированных задач, которые могут быть причиной многих страданий администраторов при использовании **cron**.

Чтобы создать свой собственный таймер, удалите файлы **.timer** и **.service** в **/etc/systemd/system**. Если вы хотите, чтобы таймер запускался при загрузке, добавьте в конец юнит-файла таймера следующую запись:

|  |
| --- |
| [Install]  WantedBy=multi-user.target |

Не забудьте включить таймер при загрузке системы с помощью **systemctl enable**. (Вы также можете сразу запустить таймер с помощью **systemctl start**).

Опция таймера AccuracySec задерживает его активацию на случайное количество времени в течение указанного временного окна. Эта функция удобна, когда таймер работает на большой группе компьютеров в сети, и вы хотите избежать одновременного срабатывания всех таймеров. (Вспомните, что для достижения этого эффекта с помощью **cron** вам необходимо использовать скрипт произвольной задержки).

AccuracySec по умолчанию составляет 60 секунд. Если вы хотите, чтобы ваш таймер работал точно по расписанию, используйте AccuracySec=1ns (по крайней мере достаточно близко к этому значению). Обратите внимание, что на самом деле вы не получите такую точность.

*Временные выражения* ***systemd***

Таймеры позволяют гибко указывать даты, время и интервалы. man-страница systemd.time является авторитетной ссылкой для описания грамматики.

Вы можете использовать интервальные выражения вместо задания секунд для относительного времени, используемого в качестве значений OnActiveSec и OnBootSec. Например, следующие формы являются допустимыми:

|  |
| --- |
| OnBootSec=2h 1m  OnStartupSec=1week 2days 3hours  OnActiveSec=1hr20m30sec10msec |

Использование пробелов является необязательным во временных выражениях. Минимальное значение составляет 1 наносекунду, но если ваш таймер срабатывает слишком часто (чаще, чем раз в две секунды), **systemd** временно отключает его.

В дополнение к запуску триггеров через определенные промежутки времени можно запланировать активацию таймеров в определенное время, включив параметр OnCalendar. Эта функция предлагает наиболее близкое соответствие синтаксису традиционного заданий **cron**, но его синтаксис более выразителен и гибок. В таблице 4.7 приведены некоторые примеры спецификаций времени, которые можно использовать в качестве значения OnCalendar.

**Таблица 4.7 Время systemd и примеры декодирования дат**

|  |  |
| --- | --- |
| **Временные спецификации** | **Значение** |
| 2017-07-04 | 4 июля 2017 года в 00:00:00 (полночь) |
| Fri-Mon \*-7-4 | 4 июля каждого года, но только если он выпадает на пт – пн |
| Mon-Wed \*-\*-\* 12:00:00 | Каждый понедельник, вторник и среду в полдень |
| Mon 17:00:00 | Каждый понедельник в 5:00 вечера |
| weekly | Каждый понедельник в 00:00:00 (полночь) |
| monthly | Каждый 1-й день месяца в 00:00:00 (полночь) |
| \*:0/10 | Каждые 10 минут, начиная с 0-й минуты |
| \*-\*-\* 11/12:10:0 | В 11:10 и 23:10 каждый день |

Во временных выражениях звезды - это заполнители, которые соответствуют любому вероятному значению. Как и в файлах crontab, косые черты вводят значение приращения. Точный синтаксис немного отличается от используемого в файлах crontab: crontab хочет, чтобы инкрементируемый объект был диапазоном (например, 9-17/2, "каждые два часа с 9:00 до 17:00"), но выражения времени **systemd** принимают только начальное значение (например, 9/2, "каждые два часа, начиная с 9:00").

*Временные таймеры*

Вы можете использовать команду **systemd-run**, чтобы запланировать выполнение команды в соответствии с любым из стандартных типов таймеров **systemd**, но без создания файлов таймера и юнит-файла сервиса для каждой задачи. Пример команды, чтобы делать pull Git-репозитория каждые десять минут:

|  |
| --- |
| $ **systemd-run --on-calendar '\*:0/10' /bin/sh -c "cd /app && git pull"**  Running timer as unit run-8823.timer.  Will run service as unit run-8823.service. |

**systemd** возвращает идентификатор временного юнита, который вы можете увидеть с помощью **systemctl**. Здесь мы еще раз профильтровали output…

$ **systemctl list-timers run-8823.timer**

|  |  |
| --- | --- |
| NEXT  LEFT  LAST  PASSED | Sat 2017-06-17 20:40:07 UTC  9min left  Sat 2017-06-17 20:30:07 UTC  18s ago |

$ **systemctl list-units run-8823.timer**

|  |  |
| --- | --- |
| UNIT  LOAD  ACTIVE  SUB  DESCRIPTION | run-8823.timer  loaded  active  waiting  /bin/sh -c "cd /app && git pull" |

Чтобы отменить и удалить временный таймер, просто остановите его, запустив **systemctl stop**:

$ **sudo systemctl stop run-8823.timer**

Работа **systemd-run** заключается в создании таймера и юнит-файла для вас в подкаталогах **/run/systemd/system**. Однако временные таймеры не сохраняются после перезагрузки. Чтобы сделать их постоянными, вы можете выловить их из **/run**, настроить как необходимо и установить в **/etc/systemd/system**. Обязательно остановите временный таймер перед запуском или включением постоянной версии.

**Обычное использование запланированных задач**

В этом разделе мы рассмотрим пару общих задач, которые часто автоматизируются с помощью **cron** или **systemd**.

*Отправка почты*

Следующая запись crontab реализует простое напоминание по электронной почте. Вы можете использовать такую запись, чтобы автоматически отправлять по электронной почте output ежедневного отчета или результаты выполнения команд (мы перенесли вывод по строкам, чтобы уместить всю информацию на странице, в действительности он представляет собой одну длинную запись).

|  |
| --- |
| 30 4 25 \* \* /usr/bin/mail -s "Time to do the TPS reports"  ben@admin.com%TPS reports are due at the end of the month! Get  busy!%%Sincerely,%cron% |

Обратите внимание на использование символа % как для отделения команды от текста input’а, так и для пометки концов строк в input’е. Эта запись отправляет электронное письмо в 4:30 утра 25 числа каждого месяца.

*Очистка файловой системы*

В случае сбоя программы ядро может записать файл (обычно называемый **core**.pid, **core** или program.**core**), содержащий образ адресного пространства программы. Core-файлы полезны для разработчиков, но для администраторов они попросту занимают свободное пространство на диске. Пользователи часто не знают о существовании core-файлов, поэтому они, как правило, не отключают их создание и не удаляют их самостоятельно. Вы можете использовать задание **cron** для очистки core-файлов или других следов, оставленных неправильным поведением и сбойными процессами.

*Ротация log-файлов*

Системы различаются по качеству управления файлами журналов, и вам, вероятно, придется изменить настройки по умолчанию в соответствии с вашими локальными политиками. "Ротация" log-файла означает разделение его на сегменты по размеру или по дате, сохраняя при этом несколько более старых версий логов. Поскольку ротация логов является повторяющимся и регулярно происходящим событием, это идеальная задача для выполнения ее по расписанию. См. Управление и ротация log-файлов на странице 319 для получения более подробной информации.

*Запуск пакетных заданий*

Некоторые долгосрочные вычисления лучше всего выполнять как пакетные задания. Например, сообщения могут накапливаться в очереди или базе данных. Вы можете использовать задание **cron** для одновременной обработки всех сообщений из очереди в виде ETL (извлечения, преобразования и загрузки) в другое место, например, в хранилище данных.

Некоторые базы данных выигрывают от рутинного обслуживания. Например, распределенная база данных с открытым исходным кодом Cassandra имеет функцию восстановления, которая синхронизирует узлы в кластере. Эти задачи обслуживания являются хорошими кандидатами для выполнения через **cron** или **systemd**.

*Резервное копирование и зеркалирование*

Вы можете использовать запланированное задание для автоматического резервного копирования каталога на удаленную систему. Мы предлагаем запускать полное резервное копирование один раз в неделю и инкрементальное резервное копирование каждую ночь. Резервное копирование запускайте поздно ночью, когда нагрузка на систему, вероятно, будет низкой.

Зеркала - это побайтовые копии файловых систем или каталогов, которые размещены в другой системе. Их можно использовать как форму резервного копирования или как способ сделать файлы доступными в нескольких локациях. Веб-сайты и хранилища программного обеспечения часто зеркалируются, чтобы обеспечить лучшую избыточность и обеспечить более быстрый доступ для пользователей, которые физически удалены от основного сайта. Периодическое запускайте команду **rsync**, чтобы поддерживать зеркала и обновлять их.