Астра Линукс

План:

1. Определение

2. Архитектура ОС

3. Управление памятью

4. Управление процессами

5. А чем занимается оболочка?

6. Подведение итогов

1. Определение

Astra Linux — операционная система на базе ядра Linux, которая внедряется в России в качестве альтернативы Microsoft Windows в государственных организациях.

2. Архитектура ОС

Давайте вспомним что из себя представляет архитектура ОС.

Как правило все модули ОС разделяют на 2 группы: ядро и оболочка. Ядро выполняет все основные функции: управление памятью, управление процессами. А оболочка – интерфейс взаимодействия пользователя с ядром.

3. Управление памятью в ядре

**Как устроена память в Linux?**

Базовой единицей в организации памяти для систем UNIX/Linux является страница памяти. Обладающая размером от 4 Кбайт, которому соответствует объём физического пространства в оперативной или виртуальной (область подкачки на диске или другом устройстве хранения) памяти. При запуске процессов, они запрашивают у системы (т. е. у ядра посредством соответствующих системных вызовов) память для своей работы. А в ответ на это ядро выделяет для них достаточное количество страниц памяти. Виртуальная память или как её ещё называют, «резервное ЗУ» (резервное запоминающее устройство) для страниц памяти. Которые содержат, к примеру, исходный текст исполняемого приложения, представляют собой обычные исполняемые файлы на диске. Равно как и для других файлов данных резервным ЗУ являются сами файлы. Информация о том как взаимосвязаны страницы физической и виртуальной памяти хранится в соответствующих таблицах страниц памяти.

Для работы с памятью в Linux (как и в других UNIX-подобных системах) характерно такое явление как «страничный обмен» (paging). Оно заключается в том, что ядро выделяет процессам столько памяти, сколько им необходимо. В том смысле, чтобы её (памяти) всегда хватало. Это достигается за счёт расширения физической памяти за счёт виртуальной, т. е. «подкачки». Поскольку выполнение процессов должно происходить в реальной физической памяти. То ядро постоянно перемещает страницы памяти процессов между физической и виртуальной памятью. Забегая вперёд, следует отметить, что в виртуальной памяти хранятся «неактивные» страницы. Которые не задействованы процессом в данный момент, но необходимые ему для полноценной работы впоследствии.

**Основные принципы управления памятью в Linux**

Первое, на что следует обратить внимание, это то, что ядро старается управлять памятью таким образом, чтобы недавно используемые процессом страницы находились в физической памяти. И в свою очередь, «неактивные» или редко используемые страницы перемещаются и хранятся в виртуальной памяти в области «подкачки». Такой механизм распределения памяти называется LRU (least recently used) — замещение наиболее редко используемых страниц.

Вторым важнейшим аспектом в работе памяти является использование кеш-буфера страниц. Это вытекает из работы алгоритма LRU, который довольно сложен в своей реализации. Поскольку следить за всеми обращениями к страницам — это в некоторых случаях, довольно ощутимые потери в производительности системы. Использование же страничного кеш-буфера куда проще в своей реализации при тех же самых результатах. К тому же данный подход имеет огромный модернизационный потенциал (в отличие от LRU) и алгоритмы анализа содержимого кеш-буфера (для определения, какие страницы должны быть перемещены из виртуальной памяти) постоянно совершенствуются. Что заметно сказывается на производительности и эффективности управления памятью.

Когда процессу не хватает памяти, то ядро начинает искать «занятые» страницы. Которые можно использовать для «голодающего» процесса. Обычно такими страницами являются те, что давно не были использованы. Ядро проверяет их на предмет модификации каким-либо процессом. Для этого существуют определённые признаки, при последнем обращении и если изменения были, то такие страницы помечаются ядром как «грязные». Т. е. такие, которые ещё нужны процессам. Для повторного использования памяти такие страницы сначала обязательно переносятся в виртуальную память. Все же остальные страницы являются «чистыми». И поэтому ядро их использует для предоставления другим или «голодающим» процессам.

**Особенности управления памятью в Linux**

Когда происходит обращение к страницам памяти, которые некоторое или долгое время не использовалис, т. е. к «неактивным» страницам. То ядро выполняет с ними несколько важных задач:

* возвращает ссылки на эти страницы в соответствующей таблице страниц;
* сбрасывает в нулевое значение время «неиспользования» этих страниц;
* помечает эти страницы как «активные».

Со страницами, находящимися в виртуальной памяти не всё так однозначно. Дело в том, что для того, чтобы «активизировать» такие страницы, они должны быть предварительно прочитаны с диска.

Системное ядро комплектуется специализированными модулями. Которые содержат алгоритмы и даже целые технологии. С помощью которых система довольно эффективно «предсказывает», сколько может потребоваться памяти при разной степени активности и загруженности процессов. Эти алгоритмы имеют своей целью обеспечение процессов свободной памятью с максимальной эффективностью. Т. е. так, чтобы процессам как можно реже приходилось простаивать в «ожидании» выгрузки очередной страницы в свободную память. Таким образом, наблюдая за состоянием страничного обмена во время рабочей нагрузки системы, можно делать выводы о том, нужна ли ей дополнительная память. Если страничный обмен интенсивный — то однозначно следует установить дополнительные модули ОЗУ.

Если же происходит так, что процессам не хватает ни реальной физической, ни виртуальной памяти. Т. е. когда память полностью исчерпана, то система начинает завершать (а точнее уничтожать) целые процессы. Либо запрещает создание новых. Конечно в этом случае в первую очередь уничтожаются наиболее «безболезненные» для системы процессы. Однако в таких случаях даже «на глаз» и по собственным ощущениям видно что она большую часть времени тратит на управление памятью, а не на выполнение рабочих задач.

В Linux можно настроить параметр, который задаёт, насколько быстро ядро должно «отбирать»страницы памяти у процессов. Которым они менее нужны для процессов, которым они на данный момент необходимы. Этот параметр содержится в файле */proc/sys/vm/swappiness* и по-умолчанию равен 60. Если задать его меньшим значением (например 0). То ядро будет забирать страницы процесса в самую последнюю очередь. Используя вместо этого любые другие варианты. Если это значение в пределах между 60 и 100. То страницы будут отбираться у процессов с более высокой вероятностью. Вариант с изменением данного параметра на самом деле говорит о том, что необходимо либо снизить нагрузку на систему. Адаптировав её для других менее производительных задач, либо увеличить объём ОЗУ.

4. Управление процессами

Планирование – это процесс распределения ресурсов системы для выполнения задач. В статье мы рассмотрим его вариант, в котором ресурсом является одно или несколько ядер процессора, а задачи представлены потоками или процессами, которые нужно выполнить.  
  
Само планирование осуществляется планировщиком, который нацелен:

* Максимизировать пропускную способность, то есть количество задач, выполняемых за единицу времени.
* Минимизировать время ожидания, то есть время, прошедшее с момента готовности процесса до начала его выполнения.
* Минимизировать время ответа, то есть время, прошедшее с момента готовности процесса до завершения его выполнения.
* Максимизировать равнодоступность, то есть справедливое распределение ресурсов между задачами.

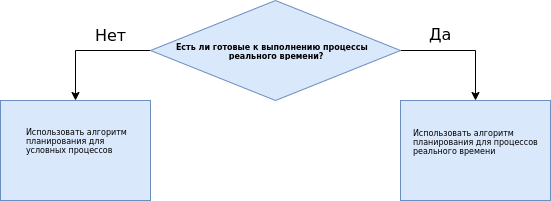
Если с этими метриками вы не знакомы, то предлагаю просмотреть несколько примеров в [другой моей статье](http://codingkaiser.blog/2021/07/26/cpu-scheduling-algorithms/) (англ.), посвященной алгоритмам планировщика.

Типы процессов в Linux

В Linux процессы делятся на два типа:

* Процессы реального времени.
* Условные процессы.

Процессы реального времени должны вписываться в границы времени ответа, независимо от загрузки системы. Иначе говоря, такие процессы являются срочными и ни при каких условиях не откладываются.  
  
В качестве примера можно привести процесс переноса, отвечающий за распределение рабочей нагрузки между ядрами ЦПУ.  
  
Условные же процессы не ограничиваются строгими рамками времени ответа и в случае занятости системы могут подвергаться задержкам.  
  
В пример можно привести процесс браузера, который позволяет вам читать эту статью.  
  
У каждого типа процессов есть свой алгоритм планирования. При этом пока есть готовые к выполнению процессы реального времени, выполняться будут они, оставляя условные процессы в ожидании.



Планирование в реальном времени

В случае с планированием в реальном времени используются две политики, SCHED\_RR и SCHED\_FIFO.  
  
Политика определяет количество выделяемого процессу времени, а также принцип организации очереди на выполнение.  
  
Немного поясню.  
  
Суть в том, что готовые к выполнению процессы хранятся в очереди, откуда выбираются планировщиком на основе той или иной политики.

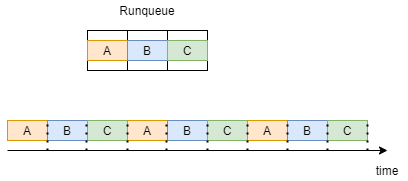
SCHED\_FIFO

В данной политике планировщик выбирает процесс, ориентируясь на время его поступления (FIFO = первым вошел, первым вышел).  
  
Процесс с политикой планирования SCHED\_FIFO может «освободить» ЦПУ в нескольких случаях:

* Процесс ожидает, к примеру, операции ввода/вывода, после чего по возвращению в состояние «готов» помещается в конец очереди.
* Процесс уступил ЦПУ через системный вызов sched\_yield, после чего он тут же возвращается в конец очереди.

SCHED\_RR

SCHED\_RR подразумевает циклическое планирование.  
  
В этой политике каждый процесс в очереди получает интервал времени (квант) и выполняется в свою очередь (исходя из приоритета) по циклическому принципу.  
  
Для лучшего понимания рассмотрим пример, где в очереди находятся три процесса, A B C, все из которых работают по политике SCHED\_RR.  
  
Как показано ниже, каждый процесс получает квант времени и выполняется в свою очередь. После однократного выполнения всех процессов они повторяются в той же последовательности.



Обобщение по планированию в реальном времени

Процесс реального времени может планироваться по двум разным политикам, SCHED\_FIFO и SCHED\_RR.  
  
Политика влияет на принцип работы очереди и определяет, сколько времени нужно выделить тому или иному процессу.

Условное планирование

Здесь мы знакомимся с Completely Fair Scheduler (CFS, абсолютно справедливый планировщик), представляющим алгоритм планирования условных процессов, появившийся в версии Linux 2.6.23.  
  
Помните метрики планирования, которые мы затронули в начале статьи? Так вот CFS фокусируется на одной из них – он стремится к максимальной равноправности процессов, то есть обеспечивает выделение всем процессам равных квантов времени ЦПУ.  
  
Обратите внимание, что процессы с повышенным приоритетом все равно могут получать на обработку больше времени.  
  
Для лучшего понимания принципа работы CFS нужно познакомиться с новым термином – виртуальное время выполнения (vruntime).

Виртуальное время выполнения

Виртуальное время выполнения процесса – это количество времени, потраченного именно на выполнение, без учета любых ожиданий.  
  
Как было сказано, CFS стремится быть максимально справедливым, в связи с чем по очереди планирует готовый к выполнению процесс с минимальным виртуальным временем.  
  
CFS задействует переменные, содержащие максимальное и минимальное виртуальное время выполнения, и чуть позже станет ясно зачем.

CFS —Абсолютно справедливый планировщик

Прежде чем перейти к принципу работы этого алгоритма, нужно понять, какие структуры данных он использует.  
  
CFS задействует красно-черное дерево, представляющее бинарное дерево поиска – то есть добавление, удаление и поиск выполняются за O(logN), где N выражает количество процессов.  
  
Ключом в этом дереве выступает виртуальное время выполнения процесса. Новые процессы или процесс, возвращающиеся из ожидания в состояние готовности, добавляются в дерево с ключом vruntime = min\_vruntime. Это очень важный момент, который позволяет избежать дефицита внимания ЦПУ для старых процессов.  
  
Вернемся к самому алгоритму. В первую очередь он устанавливает себе лимит времени – sched\_latency.  
  
В течение этого времени алгоритм стремится выполнить все готовые процессы – N. Это означает, что каждый процесс получит интервал времени равный временному лимиту, поделенному на количество процессов: Qi = sched\_latency/N.  
  
Когда процесс исчерпывает свой интервал (Qi), алгоритм выбирает в дереве следующий процесс с наименьшим виртуальным временем.  
  
Рассмотрим ситуацию, которая может стать проблематичной для такой схемы работы алгоритма.  
  
Предположим, что алгоритм выбрал лимит времени 48мс при наличии 6 процессов – в этом случае каждый процесс получит на выполнение по 8мс.  
  
Но что произойдет, если система окажется перегружена процессами? Предположим, что лимит времени остается равен 48мс, но теперь у нас 32 процесса. В результате каждый получит уже всего по 1.5мс на выполнение, что приведет к замедлению работы всей системы.

Почему?

Все дело в переключении контекста, которое подразумевает сохранение состояния процесса или потока с последующим его восстановлением и продолжением выполнения.  
  
Каждый раз, когда процесс исчерпывает свое время на выполнение, и планируется очередной процесс, активируется переключение контекста, которое также занимает некоторое время.  
  
Предположим, что на него уходит 1мс. В первом примере, где каждому процессу у нас отводилось по 8мс, это вполне допустимо. Так мы тратим 1мс на переключение контекста и 7мс на фактическое выполнение процесса.  
  
А вот во втором примере на выполнение каждого процесса останется уже всего по 0.5мс – то есть большая часть времени уходит на переключение контекста, отсюда и проблема с выполнением.  
  
Для того, чтобы исправить ситуацию, мы вводим новую переменную, которая определит минимальную протяженность кванта времени выполнения – min\_granularity.  
  
Представим, что min\_granularity = 6мс, и вернемся к нашему примеру, где лимит времени равен 48мс при наличии 32 процессов.  
  
С помощью той же формулы, что и прежде, мы получаем по 1.5мс на каждый процесс, но теперь такой вариант не допускается, так как min\_granularity задает минимальный квант времени, который должен получить каждый процесс.  
  
В данном случае, где Qi < min\_granularity, мы берем Qi равным min\_granularity, то есть 6мс, и соответствующим образом изменяем временной лимит. В результате он составит Qi x N = 6мс x 32 = 192мс.  
  
На данный момент отличия между CFS и RR могут оказаться недостаточно наглядны, поскольку они оба определяют временные интервалы и организуют порядок выполнения процессов. Для лучшего обобщения и понимания различий между этими алгоритмами я приведу краткую таблицу:

|  |  |
| --- | --- |
| **RR – циклический список** | **CFS – абсолютно справедливый планировщик** |
| * Квант времени статичен и не зависит от количества процессов в системе. | * Квант времени динамичен и может изменяться в соответствии с количеством процессов в системе. |
| * По исчерпанию процессом его кванта времени, RR выбирает очередной процесс с наименьшим виртуальным временем из циклического списка. | * По исчерпанию процессом его кванта времени, CFS выбирает очередной процесс с наименьшим виртуальным временем из красно-черного дерева. |

5. Чем занимается оболочка

Она занимается трансляцией

Дебиан - это операционная система, основанная на ядре Linux. Для взаимодействия с ядром Linux, Debian использует специальные программные интерфейсы, предоставляемые ядром. Эти интерфейсы позволяют Debian управлять ресурсами компьютера, такими как процессор, память, ввод-выводные устройства, сетевые соединения и т.д. Debian также использует системные вызовы, которые предоставляются ядром, для выполнения различных операций, таких как создание процессов, работа с файловой системой, управление сетевыми соединениями и т.д.

Кроме того, Debian содержит драйверы устройств, которые предоставляют интерфейс между аппаратным обеспечением компьютера и ядром Linux. Это позволяет Debian взаимодействовать напрямую с аппаратным обеспечением компьютера через ядро.

Таким образом, Debian взаимодействует с ядром Linux посредством использования системных вызовов, программных интерфейсов ядра и драйверов устройств, чтобы обеспечить исполнение задач и управление компьютером.

Для взаимодействия с программными интерфейсами ядра в операционной системе, в том числе в Debian, программы используют системные вызовы. Системные вызовы - это специальные функции, предоставляемые ядром, которые позволяют выполнить привилегированные операции, такие как чтение/запись в файлы, управление процессами, работа с сетью и другие операции.

Системные вызовы не выполняются на уровне машинного кода приложения, а используются через средства библиотеки glibc (GNU C Library) или других библиотек, предоставляющих интерфейс между программой и ядром.

Когда программа вызывает системный вызов, библиотека преобразует этот вызов в машинный код и передает управление ядру, которое выполняет запрос и возвращает результат обратно программе.

Таким образом, программы в Debian и других операционных системах Linux взаимодействуют с программными интерфейсами ядра через системные вызовы, а не непосредственно машинный код.