Bossa, framework для разработки планировщиков
Среда для разработки планировщиков, рассмотрение работы планировщиков
Кристоф Ожие (Christophe Augier), Пятница, 23 Июль 2004, 00:25

Краткий обзор

В последнее время основная активность в группе разработчиков ядра Linux вызвана введением в строй нового планировщика, автором которого является Инго Молнар (Ingo Molnar). Этот факт подчеркивает для обычных пользователей Linux важность планировщиков в современных операционных системах. В этой статье вы поверхностно познакомитесь с разработкой планировщиков, получив возможность самому написаться свой собственных планировщик для Linux, с помощью Bossa, framework (англ. - каркас) специально предназначеннымдля этих целей.

Bossa, framework для разработки планировщиков

Работа планировщика

Большинство продаваемых в данный момент компьютеров могут исполнять одновременно только один процесс, и, тем не менее, пользователю кажется, что множество программ работают параллельно. Эта иллюзия создана планировщиком операционной системы. Для создания такой иллюзии, планировщик заставляет систему быстро переключаться между процессами. Каждый процесс запускается на определенное время и, когда время выполнения процесса подходит к концу, планировщик выбирает, какой из процессов CPU будет обрабатывать следующим. Поэтому, поведение системы главным образом зависит от планировщика. Равноправие процессов, эффективность, латентность - планировщик должен балансировать между этими качествами, чтобы поддерживать соответствующий режим работы. Следовательно, в то время как планировщик, нацеленный на применение в настольных системах должен особое внимание уделить комплексности, планировщик, используемый в серверных системах, должен иметь хорошую производительноть и эффективно решать проблемы масштабируемост. При реализации алгоритма планировщика, разработчик должен принять во внимание тот факт, что каждое из этих свойств тесно связано с другими.
Большинство наиболее распространенны операционных систем в данный момент являются системами общего назначения, и до тех пор, пока они могут быть запущены на различных типах систем (серверы, настольные системы, встраиваемые устройства), в них, как правило, реализуется уникальный планировщик, который, соответственно, должен работать хорошо на всех этих платформах. Именно таким является новый планировщиком Linux. Реализация новых O(1) алгоритмов и переписанная поддержка мультипроцессорости, увеличивает масштабируемост серверов, построенных на основе Linux 2.6. С другой стороны, новый планировщик предлагает механизмы для приблизительног подсчета комплексности процессов, который улучшает "" системы в настольных системах.
Параллельно с индустриальным миром, исследования на поприще разработки планировщиков были горячей темой с первых дней существования многозадачных систем, и признаком этого стало возрождение темы всвязи с ростом мультимедийных приложений. Несмотря на создание большого количества новых алгоритмов, они не нашли реализации в планировщиках, используемых коммерческими операционными системами. Задачи ясны:

мультимедиа, работа в реальном времени, контроль ресурсов и т.д., и все же существующие алгоритмы работы планировщиков не нашли широкого применения. Удивительно? Нет. Одной из главных проблем, с которой сталкиваются разработчики при реализации политики планировщика в стандартной ОС состоит в том, что этот процесс требует хороших знаний сразу в двух областях: разработчик планировщика должен должен хорошо изучить ядро системы для последующей его модификации и быть в курсе исследований на этом поприще.

Bossa

Bossa - framework, призванный облегчить разработку и введение новых политик для планировщиков. Bossa разработан в Ecole des Mines de Nantes [http://webi.emn.fr] и Университете Копенгагена [http://www.diku.dk]. Основная цель разработки была буквально следующей: необходимо позволить людям делать свою работу. Если задача одних разрабатывать алгоритмы работы планировщиков, то у других необходимость ковыряться в коде ядра ОС должна отпасть.

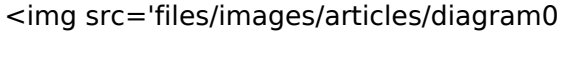
Текущая версия framework основана на Linux и заменяет разбросанный по по всему ядру код фиксированным интерфейсом для событий планировщика. Интеграция новой политики сводится к лин ковке к ядру модуля, реализующего обработчики для этих событий. Реинжениринг ядра производиться только однажды (для каждой ОС), и затем интеграция планировщиков существенно упрощается.

Bossa включает платформо-зависимый язык (DSL), который обеспечивает высокий уровень абстракции для разработчика планировщиков. Этот язык используется для написания политики планировщика, и намного более подходит для этой задачи, чем C. После этого, специальный компилятор проверяет написанный на Bossa DSL код на совместимость с целевым ядром и интерпретирует этот код в код на C, после чего может быть собран как модуль ядра Linux.

Кроме упрощения разработки политик работы планировщика, использование DSL обеспечивает лучшие гарантии безопасности. Bossa DSL накладывает больше ограничений в это плане, чем C или C++. Например, указатели (которые, как известно, являются причиной многих ошибок) просто отсутствуют, а бесконечные циклы отлавливаются самими интерпретатором Кроме того, компилятор Bossa выполняет проверку политик.

Bossa также добавляет ядру Linux иерархию планировщиков. Эта концепция заключается в построения дерева планировщиков, что позволяет различным типам приложений запускаться под различными политиками. Листья дерева - планировщики процессов, и узлы - планировщики планировщиков (так же известные, как виртуальные планировщики).

На диаграмме ниже показан пример иерархии планировщиков. В этом примере есть два типа приложений: обычные и работающие в реальном времени. В то время, как обычные приложения обрабатываются стандартным планировщиком Linux, приложения, работающие в реальном времени обрабатываются другим планировщиком, реального времени. И оба эти планировщика процессов обрабатываются виртуальным планировщиком (планировщиком планировщиков).



Дерево планировщиков

Bossa распространяется абсолютно бесплатно и доступен на собственном web-сайте:
http://www.emn.fr/x-info/bossa/ [http://www.emn.fr/x-info/bossa/]
Прямые ссылки на загрузку:
Модифицированно для поддержки Bossa ядро Linux.

[http://www.emn.fr/x-info/bossa/versions/bossa2424.2406.tar.gz]
Загрузочный компакт-диск Bossa, основанный на Knoppix.
[http://www.emn.fr/x-info/bossa/knoppix.iso]

Алгоритм кругового обслуживания

Дабы иллюстрировать процесс разработки политики планирования, используя Bossa, мы рассмотрим один из самых простых и все же наиболее широко используемыми алгоритмов: алгоритм кругового обслуживания. Этот алгоритм назначает каждому процессу некоторый промежуток времени, называемый квантом, который определяет время, в течение которого процесс работает. Если процесс все еще запущен по истечению своего кванта, он выгружается ядром, и CPU начинает обрабатывать другой процесс. Процессы, ожидающие выполнения, сохраняются в очереди, и первый процесс в очереди, соответственно, будет исполнен первым при следующем освобождении процессора. После того, как процесс выгружен, он помещается в самый конец очереди, что подразумевает, что ему придется ждать исполнение всех других процессов впереди себя в течении их квантов, чтобы получить очередную порцию своего времени. На рисунке, расположенном чуть ниже, показана схема исполнения четырех процессов по данному алгоритму.

Исполнение процессов по алгоритму кругового обслуживания с квантом, равным 3 мс.

Далее мы объясним, как написать планировщик, основанный на алгоритме кругового обслуживания, используя Bossa. Кроме того, мы посмотрим, как загрузить этот планировщик в модифицированно Bossa ядро Linux и запустим пару испытательных программ, чтобы показать, что он работает.
Реализация алгоритма кругового обслуживания на языке Bossa
Bossa вводит новый язык программировани для написания политик планировщиков. В этой секции мы сделаем краткий обзор этого языка, в процессе реализации алгоритма кругового обслуживания.
[newpage]
Политика Bossa разделена на пять основных частей:
1.Определение структуры процесса
2.Объявление различных состояний
3.Критерии, по которым выбирается следующий процесс для запуска
4.Обработка событий планировщика
5.Интерфейс, позволяющий пользовательски программам управлять некоторыми аспектами политики
Каждая из этих частей подробно рассмотрена ниже.
Определение структуры процесса
Поскольку наша политика кругового обслуживания представляет собой планировщик процессов, мы должны объявить атрибуты процесса, которые будут использоваться политикой.

```
process = {
   int time_slice;
```

}

Это объявление означает, что каждый процесс имеет атрибут time_slice, который будет использоваться для хранения времени, в течении которого обрабатывается процесс. Управляя процессом p мы можем получить доступ к времени его выполнения, используя p.time_slice.

Состояния

Состояния жизни процесса

Состояния процесса за период его жизни могут упрощено быть охарактеризован, как: запущен, готов, блокирован и остановлен. В Bossa присутствует два типа состояний:
1.Классы состояний: Запущен, Готов, Блокирован, Остановлен. Это основные состояния жизни процесса с точки зрения ядра.
2.Состояния: это состояния, объявленные разработчиком, и используемые в политике планировщика. Состояние всегда ассоциируется с классом состояния.
Давайте посмотрим, как в Bossa объявляются состояния:

```
states = {
   RUNNING running : process;
   READY active : sorted fifo queue;
   BLOCKED blocked : queue;
   TERMINATED terminated;
}
```

Выше, мы объявили четыре состояния: запущен(running), активен(active), блокирован(blocked), и остановлен(terminated). Состояние запущен(running) оссоциируется с классом состояния RUNNING и ссылается на уникальный процесс. Состояние активен(active) оссоциируется с классом состояния READY(готов) и ссылается на очередь процессов, которая обслуживается в режиме fifo (first-in first-out - "первым пришел - первым "). (sorted будет обсуждаться в следующем разделе).
running ссылается на процесс, запущенный в данный момент, в то время как active ссылается на очередь, содержащую все процессы, готовые быть выполненными, blocked ссылается на блокированные процессы, a terminated не ссылается ни на один процесс, т.е. похоже на /dev/null, так как изменение состояния процесса на terminated уничтожает все ссылки на него.

Выбор

Выбор - вероятно главная часть всех планировщиков, поскольку поведение (правильное выделение времени процессу, комплексность, и т.д.) политики планирования зависит, от способа выбора следующего запущенного процесса. Язык Босса обеспечивает возможность упростить спецификацию выбора процесса. Он позволяет пользователю определять

критерии, по которым будет выбран следующий процесс из ряда. Разработчик использует аннотацию, sorted, чтобы указать состояние (и структуру данных), к которому должны быть применены критерии выбора. В предыдущем разделе, мы объявляли очередь, содержащую процессы с состоянием active как sorted. Выбор, соответственно, будет произведен только из процессов с этим состоянием.
Критерии определяются при помощи ключевого слово ordering_criteria. Например, если следующим запущенным процессом должен быть процессом с самым высоким приоритетом, это должно быть определено как:
ordering_criteria = { highest priority }
В случае, если существует несколько процессов с самым высоким приоритетом, следующий процесс будет выбран в соответствии с положением этих процессов в очереди.
Вернемся к нашему примеру. Планировщик при осуществлении алгоритма кругового обслуживания всего лишь получает следующий процесс из очереди процессов, так что нет необходимости объявлять какие бы то ни было критерии.

```
ordering_criteria = { }
```

Обработка событий

Bossa, как уже было сказано выше, заменяет разбросанный по по всему ядру код фиксированным интерфейсом событий планировщика. В этой секции мы рассмотрим, какие события доступны и как с ними работать. Поскольку эта статья - лишь краткий обзор возможностей Bossa, мы рассмотрим только основные события: new, block, unblock, clocktick, schedule и end .
Сначала давайте посмотрим, как объявляется набор обработчиков событий:

```
handler(event e) {

   On event_name_1 {
      /* какие-то действия */
   }

   On event_name_2 {
      e.target.attribute = X;
       //e.target – процесс, на который нацелено событие e
      e.source.attribute = Y;
       //e.source – процесс, сгенерировавшийсобытие e
   }
}
```

new

Событие new - первый шаг в жизни процесса. Ядро Linux выполняет всю работу по размещению процесса и других ресурсов в памяти. Политика должна всего лишь инициализироватьданные для планировщика и состояние процесса.

```
On process.new  {
   e.target.time_slice = 30;
   e.target => active;
}
```

Данный код присваивает свойству time_slice 30 единиц и помечает процесс как active, соответственно, теперь он может быть запущен.

block

Это событие весьма часто во время жизни процессов, основаных на вводе/выводе. Оно происходит, когда процесс блокируется, например, для ожидания ввода пользователя.

```
 On block.* { // для обработки всех событий блокирования можно использовать маску *
   e.target => blocked;
}
```

Все заблокированныепроцессы мы помечаем, как blocked, в результате чего они не могут быть запущены.

unblock

Как противовес событию block.*, событие unblock.* происходит, когда процесс разблокируется, к примеру, если пользователь что-то напечатал.

```
On unblock.* {
   if (e.target in blocked) {  // если e.target блокирован
      e.target => active;
   }
}
```

Данный обработчик проверяет, находится ли процесс в состоянии blocked. Если да, мы изменяем его состояние с blocked на active. Оператор => автоматически удаляет процесс из структуры данных (очередь, переменная процесса, список, дерево...), ассоциируемой с его текущим состоянием.
[newpage]

clocktick

Событие clocktick возникает при каждом тике часов CPU. Как правило, планировщик может использовать clocktick как единицу времени, в нашем случае, это необходимо для управления атрибутом time_slice. Выше, в обработчике события new мы установили свойство time_slice в 30, что подразумевает, что процесс может исполняться а течении 30 тиков. Ниже способ обработки этого события, применительно к алгоритму кругового обслуживания:

```
On system.clocktick {
   running.time_slice--;                  // процесс уже потратил один тик
   if (running.time_slice <= 0) {         // время процесса вышло
      running.time_slice = 30;        // восстанавливаемого время
      running => active;              // и выгружаем его
   }
}
```

Если процесс израсходовал свое время, он выгружается.

schedule

Мы знаем, что процесс может быть выгружен в течение блокировки и тика процессора. Когда запущенный процесс был выгружен, планировщик должен выбрать следующий процесс для запуска. Алгоритм кругового обслуживания всего лишь выбирает следующий процесс с состоянием active. Таким образом, реализация события schedule очень проста:

```
On bossa.schedule {
   select() => running;
}
```

select(), возвращает процесс из списка sorted, выбранный согласно ordering_criteria и любой дополнительной информации о порядке этого списка (в данном случае, fifo). Изменяя состояние этого процесса на running (используя =>) мы объявляем, что выбранный процесс должен быть выполнен (мы изменяем его состояние с active на running).

end

Наконец, событие end указывает на завершение процесса. Чаще всего мы не заботимся о законченных процессах, так что обработчик этого события, как правило будет содержать лишь:

```
On process.end {
   e.target => terminated;
}
```

Этот обработчик как правило более сложен в случае политик, в которых критерии приема основаны на текущей загрузке планировщика (например, EDF[1]). Эти политики должны знать, когда процессы умирают, таким образом уменьшая нагрузку.

Хватит теории! Покажите мне, как это работает.

В описании выше вы получили краткий обзор того, как происходит написание политики планировщика, используя Bossa. В этом разделе, мы проверим наш планировщик. Полную реализацию и некоторые тестовые программы можно найти в архиве rr.tar.gz [http://www.emn.fr/x-info/bossa/rr.tar.gz] . Конечно, чтобы эти программы запустились правильно, первым делом необходимо загрузить и запустить ядро Bossa.
1.Распакуйте, соберите, установите, и загрузитесь с ядром Bossa [http://www.emn.fr/x-info/bossa/versions/bossa2424.2406.tar.gz] (Примечание: Вы должны использовать make install, чтобы установить ядро и инструменты Bossa), или загрузитесь, используя компакт-диск Bossa [http://www.emn.fr/x-info/bossa/knoppix.iso]
2.Распакуйте архив:

```
 $ tar zxf RR.tar.gz
    $ cd RR
    $ ls
    loop10.c  Proportion.bossa  RR30.bossa     testRR.sh
    loop30.c  RR10.bossa        setup_test.sh
    $
```

RR10.bossa и RR30.bossa - две реализации алгоритма кругового обслуживания. В RR10 time_slice каждого процесса - 10, в то время, как в RR30 - 30.

Дерево планировщиков
3.Запустите скрипт setup_test.sh с правами супер-пользователя . Этот скрипт собирает две политики кругового обслуживания, как модули Linux, используя утилиту bossa_install, включенную в поставку ядра Bossa. Затем скрипт загружает модули в ядро и формирует дерево планировщиков, как показано на диаграмме.

```
 $ ./setup_test.sh
```

4.Запустите тестовый скрипт.

```
$ ./testRR.sh
```

Скрипт загружает три бесконечных цикла параллельно, каждый цикл печатает число от

одного до трех. Исходный текст может быть найден в файлах loop10.c и loop30.c. После запуска программа присоединяется к планировщику кругового обслуживания и затем исполняется.

We run three loops (1, 2, 3) in parallel. They are
scheduled by the round-robin scheduler with time_slice=30.
11111111111111111112222222222222222233333333333333333311111111111111111122222
22222222222223333333333333333333311111111111111111112222222222222222233333333
33333333331111111111111111112222222222222222223333333333333333333311111111111
11111122222222222222222233333333333333333311111111111111111112222

The round robin scheduling effect can clearly be seen.

We run three loops (1, 2, 3) in parallel. They are scheduled by the round-robin
scheduler with time_slice=10.
11111222222333333111111222222333333111111222222333333111111222222333333111122

22223333331111112222223333331111112222223333331111112222223333331111112222223333

33111111222222333333111111222222333333111111222222333333111111222222333333111111

22222333333311111122222233333311111122222233333311111122

The round robin scheduling effect can clearly be seen.

То, что все отлично работает видно из вывода на консоль.

Заключение

В этой статье мы рассмотрели Bossa, framework для разработки политик работы планировщиков. После прочтения статьи вы должны понимать процесс разработки политик работы планировщика, а так же преимущества платформо-зависимого языка. Bossa существенно упрощает разработку, и большинство алгоритмов работы планировщика может быть реализовано всего лишь несколькими сотнями строк кода.
Кроме того, поскольку Bossa упрощает разработку и интеграцию планировщиков в ядро ОС, она может использоваться для обучения студентов работе планировщиков. Экспериментируя с различными политиками работы планировщика, используя Bossa, студенты могут на практике понять, как работает планировщик.
Об авторе: Кристоф Ожие (Christophe Augier) – студент Ecole des Mines de Nantes (Франция), где он изучает операционные системы и компьютерные сети. В качестве магистерской диссертации, он пишет специальный язык для диспетчеров сетевых пакетов.
Большое спасибо Джулии Лоуэлу (Julia Lawall) и Жильу Мюлле (Gilles Muller) за поддержку в написании этой статьи.

Ссылки

1 The Earliest-Deadline First scheduling policy:
Bossa-версия политики EDF: EDF.bossa [http://x-info.emn.fr/bossa/policies/EDF.bossa]
Исследовательсkя работа, описывающая EDF:
Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment,
[http://portal.acm.org/citation.cfm?id=321743&dl=ACM&coll=portal]
C. L. Liu and James W. Layland
Journal of the ACM (JACM)
Volume 20, Issue 1 (January 1973)