Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

**Отчёт по курсовому проекту**

**Дисциплина**: Проектирование ОС и компонентов

**Тема**: Встраивание планировщика EDF в ядро ОС Linux

Выполнил студент гр. 13541/4 Степанов Д.С.

(подпись)

Руководитель Душутина Е.В.

(подпись)

“ ” 2017 г.

Санкт - Петербург

2017

**Содержание**

Введение ………………………………………………………………………..…3

1. Политики планирования в ОС Linux ……………………………………..6
   1. Completely Fair Scheduler …………………………………………...9
   2. Earliest Deadline Scheduler ………………………………………...10
2. Разработка и внедрение планировщика EDF ……………...……………12
   1. Красно-чёрное дерево ……………………………………………. 24
3. Сборка и внедрение ядра ……………………………………………….. 28
4. Тестирование ……………………………………….……………………….30

Заключение ………………………………………………………………………32

Список использованных источников …………………………………………..33

**Введение**

Ключевым моментом в обеспечении корректной работы ОС Linux является планировщик выполнения задач. Архитектура планировщика в ОС Linux непрерывно изменялась.

ОС Linux является открытой, что позволяет провести анализ текущих механизмов планирования и внедрить свой усовершенствованный алгоритм. Приведем анализ разработок планировщиков задач за последнее десятилетие.

Основная задача планировщика состоит в выполнении максимального объёма работы в условиях ограничений. Автором планировщика ОС Linux для ядер серии 2.6.x является Инго Молнар (Ingo Molnar) [1;2].

На вход планировщику поступает очередь задач, которую необходимо выполнить. Длина этой очереди определяет время работы планировщика ОС Linux ядер серии 2.4.x. но не 2.6.x серии.

* 0(n) - эта запись используется для отображения того факта, что с возрастанием количества входных данных, возрастает время работы самого планировщика.
* 0(1) - запись показывает что, время работы алгоритма константное. Алгоритм, который характеризуется 0(1), гарантирует завершиться за некий период времени вне зависимости от размера входящей очереди задач. Каждая часть такого алгоритма выполняется за некое константное время.

Работа планировщика ядра 2.6.x базируется на использовании следующих понятий:

1. Runqueue - очередь задач готовых к выполнению. Каждый ЦПУ имеет свою очередь задач.
2. Priority array - массив приоритетов. Очередь задач связана с двумя массивами приоритетов: один активный другой пассивный. Массивы приоритетов меняются местами, когда активный массив приоритетов опустошается. Задача массива приоритетов сводится к нахождению задачи с наибольшим приоритетом за константное время. Массив приоритетов представляет собой битовое поле, которое отображает приоритеты для активных задач.
3. Интерактивные кредиты - позволяют лучше распределять процессорное время между интерактивными задачами. Задача получает интерактивный кредит, когда долго спала, и наоборот, теряет интерактивный кредит, когда значительное время занимала ЦПУ.
4. Домен планирования (scheduler domain) - включает несколько вычислительных ресурсов. Каждый домен планировщика разделяет все свои доступные ЦПУ на группы. В многопроцессорной системе или одно процессорной системе каждый физический ЦПУ образует отдельную группу. Балансировка нагрузки происходит только внутри домена. Задачи перемещаются внутри домена между группами. Нагрузка группы равна нагрузке всех ЦПУ в этой группе. С целью предотвращения частых очисток кэша, каждая задача выполнятся на одном и том же ЦПУ. Тем не менее, иногда наступают моменты, когда некий ЦПУ имеет больше задач на выполнение чем другие ЦПУ в системе. Тогда планировщик пытается провести балансировку нагрузки, и равномерно распределить задачи между процессорами.

Классификации нитей является одним из ключевых мест работы планировщика ядра 2.6.x. Планировщик, для работы на пользовательских системах, ставит нитям В/В высокий приоритет доступа к ЦПУ. Это делается с целью повышения интерактивности. Без этого подхода, при большой нагрузке ЦПУ, в моменты больших вычислений, пользователь будет ощущать не достаточно быструю реакцию компьютера на свои действия.

Также известно, что нити В/В занимают много времени и их лучше запустить на выполнение как можно раньше. Например, программа, ожидающая данные из дискового накопителя, значительное время будет находиться в режиме ожидания, прежде чем она получит свои данные. Пока будут извлекаться данные из диска, ЦПУ сможет выполнить другую полезную работу.

Планировщик ОС Linux не делает различия между нитями которые работают с системой В/В, так не может знать, ожидает ли нить данных с клавиатуры или с дискового накопителя. Планировщик рассматривает все нити В/В одинаково.

Планировщик ядра ОС Linux 2.6.x позволяет приложениям реального времени (РВ) указывать очень жёсткие по времени предельные сроки выполнения (deadline), но не гарантирует их соблюдения. Под мягким планированием (soft RT scheduling) понимают, отсутствие гарантии, что задача будет выполнена к некоторому желаемому сроку (deadline).

Нити РВ выделяются в отдельный класс. Планировщик дает нитям РВ приоритет над всеми другими нитями в системе:

* задачи РВ имеют приоритеты в диапазоне 0-99;
* обычные задачи отображаются в диапазон 100-140.

1. **Политики планирования в ОС Linux**

Поскольку задачи РВ имеют приоритет "ниже" чем обычные задачи, они всегда вытесняют обычные задачи. В момент выполнения задач РВ, выполнения других задач невозможно. Задачи РВ работают с другими схемами планирования:

* SCHED\_FIFO
* SCHED\_RR

Обычные задачи маркируются как SCHED\_NORMAL.

1. Планирование задач РВ по схеме SCHED\_FIFO. Если процессор содержит SCHED\_FIFO (First In First Out) задачу она вытесняет любую другую задачу и выполняется так долго, сколько ей необходимо. Такая задача не имеет ограничений по времени выполнения. Если в очереди задач несколько SCHED\_FIFO задач, то планировщик распределит выполнение задач по приоритетам: задачи с более высоким приоритетом вытеснят задачи с более низким приоритетом.
2. Планирование задач РВ по схеме SCHED\_RR. Механизм планирования SCHED\_RR (Round-Robin) задач очень похож на планирование SCHED\_FIFO задач. Отличие состоит в том что:
3. для каждой SCHED\_RR задачи указано как много она может использовать процессорного времени (timeslice);
4. любая SCHED\_RR задача будет безоговорочно вытеснена SCHED\_FIFO задачей. SCHED\_FIFO нити имеют более высокий приоритет, чем SCHED\_RR нити.

SCHED\_RR задачи планируются согласно своим приоритетам по кругу (round-robin). Задача, которая использовала отведённое ей процессорное время, помещается в конец очереди.

Каждая SCHED\_RR задача согласно своему приоритету выполняется в течение отведённого ей процессорного времени, а потом заносится в конец списка очереди согласно своему приоритету (priority array queue).

В начале файла kernel/sched.c определено несколько макросов. Изменяя определения этих макросов можно достичь желаемой работы планировщика.

Рассмотрим более раннюю реализацию планировщика ОС Linux в серии ядер 2.4.х. Он имеет надёжный дизайн. Но существует ряд нежелательных характеристик. Планировщик делит время на эпохи. Эпоха - период времени, за которое каждая задача может использовать своё процессорное время. В начале наступления новой эпохи для каждой задачи планировщик подсчитывает сколько времени она может выполнятся O(n) итераций.

Каждой задачи присваивается базовое значение в течении которого она может использовать ЦПУ. Это значение определяется пользователем с помощью утилиты nice. Установленное значение масштабируется в некое количество тактов микропроцессора. Значение 0 примерно масштабируется в 200 мили секунд.

Когда происходит подсчёт сколько процессорного времени выделить для задачи, это базовое значение может изменится в зависимости является ли задача В/В. Каждая задача имеет переменную counter. Эта переменная содержит количество оставшихся тактов микропроцессора. За всю эпоху, задача могла не использовать все выделенное ей процессорное время, т. е. значение переменной counter > 0. Это могло произойти например если задача спала находилась в ожидании операции В/В. Тогда в конце эпохи задаче будет выделено больше процессорного времени.

Когда задача порождает потомка, то, процессорное время родителя разделяется с потомком, что предотвращает захват ЦПУ размножением.

Функуия schedule() чтобы выбрать задачу для выполнения, вызывает функцию goodness(). Функция goodness() проверяет наличие задачи, которая может выполнится в текущем адресном пространстве (p->mm == prev->mm), и если есть, то ей отдаётся предпочтение.

Основные проблемы:

* Плохая масштабируемость (scalability) == 0(n). Планировщик назначает всем задачам приоритеты (цикл через все активные задачи).
* Планировщик ищет задачу с наивысшим приоритетом (также цикл через все задачи).
* Если существует 100 активных нитей, и одна нить с более низким приоритетом чем все остальные нити, то этой нити придётся ждать своего выполнения 20 сек: 200 мс \* 100
* Если существует несколько интерактивных задач с более высоким приоритетом, обычные задачи будут голодать и долго ожидать времени выполнения. Пример: WEB-сервер получив данные c ресурса В/В будет очень долго формировать ответ.

Автор реализации планировщика в ядре 2.6 Инго Молнар предложил полностью переписать подсистему планирования в ядре Linux. Тем не менее, Молнар заметил, что планировщик должен оставаться универсальным и пригодным для использования в пользовательских настольных системах и высокопроизводительных серверах.

Молнар предложил построить систему планирования по модульному принципу. Предполагается, что модули инкапсулируют (заключают) детали политики планирования. Базовая часть кода является инвариантной и представляет интерфейс, позволяющий подключать различные алгоритмы планирования.

Особенностью предложенного Молнаром подхода является то, что выбор алгоритма должен происходить на этапе компиляции ядра Linux.

* 1. **Completely Fair Scheduler**

В качестве базового модуля Молнар предлагает использовать Completely Fair Scheduler (CFS) - полностью справедливый планировщик. Замысел алгоритма CFS вполне радикален: он не использует очередей задач (runqueues), а использует отсортированную во времени структуру Red Black Tree (сбалансированное красно-черное дерево). Эта структура используется для построения очередности выполнения задач. По заявлению автора, этот алгоритм избавлен недостатков текущего планировщика, источником которых были переключаемые массивы задач. Задачи помещаются в дерево согласно их приоритету (времени, оставшемуся от timeslice), и дерево постоянно перемешивается (подобно тому, как раньше массивы менялись местами). Алгоритм подсчитывает время, оставшееся от выделенного процессу на исполнение в данный интервал, что и является приоритетом.

Планировщик CFS использует наносекундный учет времени выполнения задач и не зависит от других рабочих характеристик ЦПУ, например его частоты.

Алгоритм CFS не учитывает процессорное время, выделенное задаче, или эвристические данные о потоке выполнения задач. Единственный способ управлять работой планировщика предусмотрен изменением параметра в файле */proc/sys/kernel/sched\_granularity\_ns.* Таким образом, изменяя значение этого параметра, можно подстраивать функционирование ОС для работы в режиме:

* сервера - задачи хорошо укомплектованы;
* рабочей станции - минимальное время реакции на действия пользователя.

Время работы алгоритма CFS оценивается в O(log(n)) по сравнению с 0(1) в текущей реализации планировщика.

В настоящее время, помимо планировщика CFS также большую популярность получил алгоритм планирования EDF(earliest deadline first).

* 1. **Deadline Scheduler**

Deadline Scheduler был написан Дженсом Аксбо (Jens Axboe). Основополагающим принципом его работы является гарантированное время запуска запросов ввода-вывода на обслуживание. Он сочетает в себе такие возможности как слияние запросов, однонаправленный алгоритм лифта[[1]](#footnote-1) и устанавливает предельный срок на обслуживание всех запросов (отсюда и такое название). Он поддерживает две специальные «очереди сроков выполнения» (deadline queues) в дополнение к двум отдельным «отсортированным очередям» на чтение и запись (sorted queues). Задания в очереди сроков выполнения сортируются по времени исполнения запросов по принципу «меньшее время — более раннее обслуживание — ближе к началу очереди». Очереди на чтение и запись сортируются на основе запрашиваемого ими номера сектора (алгоритм лифта).

Данный планировщик действительно помогает пропускной способности в случаях чтения с секторов с большим номером блока. Операции чтения могут иногда заблокировать приложения, поскольку пока они выполняются, приложения ждут их завершения. С другой стороны, операции записи могут выполняться гораздо быстрее, поскольку они производятся в дисковый кэш (если только вы не отключили его использование). Более медленные операции чтения с внешних областей диска перемещаются к концу очереди, а более быстрые операции чтения с более близких областей поступят на обслуживание раньше. Такой алгоритм планировщика позволяет быстрее обслужить все запросы на чтение-запись, даже если существуют запросы на чтение с дальних областей диска.

Схема работы планировщика достаточно прямолинейна. Планировщик сначала решает какую очередь использовать первой. Более высокий приоритет установлен для операций чтения, поскольку, как уже упоминалось, приложения обычно блокируются при запросах на чтение. Затем он проверяет истекло ли время исполнения первого запроса. Если так — данный запрос сразу же обслуживается. В противном случае, планировщик обслуживает пакет запросов из отсортированной очереди. В обоих случаях, планировщик также обслуживает пакет запросов следующий за выбранным в отсортированной очереди.

Deadline scheduler очень полезен для некоторых приложений. В частности, в системах реального времени используется данный планировщик, поскольку в большинстве случаев, он сохраняет низкое время отклика (все запросы обслуживаются в короткий временной период).

Приложение реального времени обычно состоит из нескольких задач с разными уровнями критичности. Жесткие задания в реальном времени не могут пропустить ни одного крайнего срока, в противном случае могут возникнуть нежелательные или смертельные результаты, тогда как задачи в режиме реального времени могут пропустить некоторые крайние сроки, и система все еще может работать правильно.

Например, кардиостимулятор сердца - это жесткая система реального времени, потому что задержанный сигнал может привести к смерти человека, тогда как аудио-видео система обычно классифицируется как система с мягким реальным временем, потому что отсутствие предельного срока приводит к ухудшению качества, но система может продолжать работать.

1. **Разработка и внедрение планировщика EDF**

Прежде чем мы изучить детали реализации планировщика, важно понять некоторые из наиболее важных параметров задачи реального времени. Задача реального времени (ti) характеризуется следующими параметрами: время готовности (Ri), наихудшее время выполнения (Ci), дедлайн (Di) и периодичность (Ti).

Ri - это время, когда задача готова к обработке. Ci - это процессорное время, необходимое для выполнения задачи без перерыва. Di - это время, когда задача должна быть выполнена, чтобы избежать повреждения системы. Ti - это временной интервал, с которым задачи освобождаются в системе.

Рассмотрим пример (рис.1.). В последнем столбце таблицы указано смещение (Oi) первого задания для каждой задачи. То есть момент времени, когда первое задание освобождается.

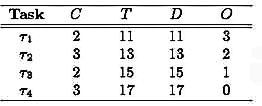


Рис.1. Исходные данные для примера использования алгоритма EDF

Результат работы алгоритма представлен на рисунке 2. Выполнение заданий представлено серыми прямоугольниками, а черный круг - завершением выполнения задания.

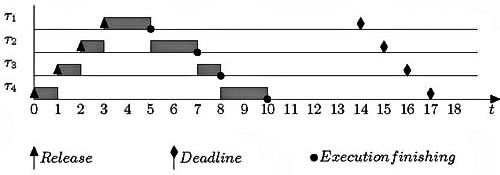


Рис.2. Результат планирования

Как можно увидеть, назначение приоритета выполняется в соответствии с абсолютным сроком (который является суммой Ri + Di). То есть, в определенный момент времени работа, которая выполняется, является той, у которой самый ранний крайний срок для всех активных заданий.

Первым шагом является добавление новой записи параметра конфигурации, которая используется для переноса всего необходимого кода SCLS. Поскольку хост-система основана на архитектуре x86, эту запись параметра конфигурации необходимо добавить в файл /kernel\_source\_code /arch/x86/Kconfig.

Ниже показано содержимое новой записи параметра конфигурации. Определяющее имя (SCHED\_MY\_POLICY) определяет, что представляет собой эта опция. Тем не менее, эта запись о параметрах конфигурации указывается в коде с помощью CONFIG\_SCHED\_MY\_POLICY. Префикс CONFIG\_ предполагается, но не записывается.

menu "EDF scheduler"

config SCHED\_EDF\_POLICY

bool "EDF scheduling policy"

default y

endmenu

В директиве bool указано, что этот параметр является функцией и может быть задан только с помощью двух значений (y или n). Цитируемый текст, следующий за директивой, предоставляет имя этой опции в различных утилитах конфигурации, таких как make menuconfig. Значение по умолчанию этой опции определяется с использованием директивы default.

Второй шаг - определить макрос для определения политики планирования. Для этого нам нужно изменить /kernel-source-code/include/linux/sched.h и /usr/include/bitssched.h файлы, чтобы определить этот макрос:

#ifdef CONFIG\_SCHED\_EDF\_POLICY

#define SCHED\_EDF 6

#endif

Структуры task\_struct и struct rq являются двумя центральными структурами данных в системе. Процесс Linux - это экземпляр выполняемой программы . Для управления процессами ядро ​​поддерживает информацию о каждом процессе в дескрипторе процесса. Информация, хранящаяся в каждом дескрипторе процесса (struct task\_struct, определенная в / kernel source-code / include / linux / sched.h), связана с состоянием процесса, его адресным пространством, списком открытых файлов, приоритетом процесса и его классом планирования. Для решения задач нашего планировщика в эту структуру данных необходимо добавить некоторые поля.

В листинге ниже показаны добавленные поля. Поле my\_id используется для установки логического идентификатора задачи. Относительный срок (с использованием наносекундной единицы времени) каждой задачи устанавливается в крайний срок.

#ifdef CONFIG\_SCHED\_EDF\_POLICY

unsigned int edf\_id;

unsigned long long deadline;

#endif

Каждый процессор содержит очередь выполнения всех выполняемых процессов. Политика планирования использует эту run-queue для выбора «лучшего» процесса, который должен быть выполнен. Информация для этих процессов хранится в структуре данных для каждого процессора, называемой struct rq, которая объявлена ​​в /kernel\_source\_code/kernel/sched.c.

Информация о каждой задаче сохраняется с использованием структуры данных struct edf:

#ifdef CONFIG\_SCHED\_edf\_POLICY

struct edf\_task{

struct rb\_node edf\_rb\_node;

unsigned long long absolute\_deadline;

struct list\_head edf\_list\_node;

struct task\_struct \*task;

};

struct edf\_rq {

struct rb\_root edf\_rb\_root;

struct list\_head edf\_list\_head;

atomic\_t nr\_running;

};

#endif

Таким образом, поле задачи является указателем на запись дескриптора процесса. Абсолютный крайний срок сохраняется в поле абсолютного предельного значения.

Поле типа struct rb node типа данных необходимо для организации задач на красно-черном дереве. В ядро Linux уже реализовано красно-черное дерево (/ kernel-source-code / include / linux / rbtree.h).

Как известно, красно-черные деревья являются сбалансированными бинарными деревьями, узлы которых отсортированы по ключу, следовательно, большинство операций выполняются за O (log (n)), таким образом, красно-черное дерево подходит для частой ротации узлов.

Поле struct edf\_list\_node необходимо для организации всех задач, имеющихся в системе, в двойном связанном списке.

Структура данных struct-head определена в файле / kernel\_source\_code / include / linux / list.h, который реализует простой в использовании двойной связанный список с использованием языка программирования C.

Все задачи, назначенные одному процессору, управляются с помощью структуры данных struct edf\_rq. Они хранятся в связанном списке.

Корень красно-черного дерева является корнем задачи поля edf. И nr\_running используется для указания количества задач в очереди выполнения. Поле данных типа struct edf\_rq должно было быть добавлено в структуру данных struct rq rq.

Структуры данных планировщика и очереди выполнения инициализируются функцией sched\_init, которая определена в файле / kernel\_source\_ code / kernel / sched.c. Поскольку в struct rq rq было добавлено новое поле, необходимо инициализировать это поле данных:

#ifdef CONFIG\_SCHED\_EDF\_POLICY

init\_edf\_rq(&rq->edf\_rq);

#endif

Чтобы добавить новую политику планирования в ядро Linux, необходимо создать новый модуль. После добавления нового модуля с наивысшим приоритетом модули планировщика становятся иерархически организованными, как показано в рисунке 3:



Рис.3. Иерархия модулей

Согласно модульным правилам планирования, каждый модуль должен реализовывать набор функций, указанных в структуре sched\_class. В листинге ниже показано определение edf\_sched\_class, которое реализует модуль EDF.

Первое поле этой структуры (далее) является указателем на sched\_class, который используется для организации модулей планировщика по приоритету в связанном списке, а ядро планировщика, начиная с модуля планировщика с наивысшим приоритетом, будет искать выполняемую задачу каждого модуля в порядке убывания приоритета.

В этом случае, поскольку edf\_sched\_class является модулем планировщика с наивысшим приоритетом, это поле указывает на следующий модуль планировщика с низким приоритетом, который является структурой rt\_sched\_class, которая, в свою очередь, реализует модуль RT (поскольку в данный момент этот модуль имеет наивысший приоритет)

const struct sched\_class edf\_sched\_class = {

.next = &rt\_sched\_class,

.enqueue\_task = enqueue\_task\_edf,

.dequeue\_task = dequeue\_task\_edf,

.check\_preempt\_curr = check\_preempt\_curr\_edf,

.pick\_next\_task = pick\_next\_task\_edf,

.put\_prev\_task = put\_prev\_task\_edf,

#ifdef CONFIG\_SMP

.load\_balance = load\_balance\_edf,

.move\_one\_task = move\_one\_task\_edf,

#endif

.set\_curr\_task = set\_curr\_task\_edf,

.task\_tick = task\_tick\_edf,

};.

Другие поля являются функциями, которые действуют как обратные вызовы для определенных событий, которые описаны в следующих разделах. В описании следующих функций предполагается, что задачи уже хранятся в связанном списке, который является полем списка edf struct edf\_rq.

Enqueue\_task\_edf (листинг ниже) вызывается всякий раз, когда задача входит в состояние выполнения. Он получает два указателя: один для очереди выполнения процессора, на котором выполняется этот код (rq), а другой - для задачи, которая входит в состояние runnable (p).

Эта функция, вызывая функцию списка задач find\_edf, получает указатель на структуру struct edf, хранящуюся в связанном списке struct edf\_rq, который указывает на задачу p. Затем он обновляет абсолютный срок и вставляет edf\_task в красно-черное дерево (insert\_edf\_task\_rb\_tree).

Вызов native sched\_clock возвращает текущее время в наносекундах (эта функция определена в файле / kernel-source-code / kernel / sched.c). Кроме того, он регистрирует это событие в системе ведения журнала edf .

static void enqueue\_task\_edf(struct rq \*rq, struct task\_struct \*p, int wakeup)

{

struct edf\_task \*t=NULL;

char msg[edf\_MSG\_SIZE];

if(p){

t=find\_edf\_task\_list(&rq->edf\_rq,p);

if(t){

t->absolute\_deadline=sched\_clock()+ p->deadline;

insert\_edf\_task\_rb\_tree(&rq->edf\_rq, t);

atomic\_inc(&rq->edf\_rq.nr\_running);

snprintf(msg,edf\_MSG\_SIZE,"(%d:%d:%llu)",p->edf\_id,p->pid,t->absolute\_deadline);

register\_edf\_event(sched\_clock(), msg, edf\_ENQUEUE);

}

else{

printk(KERN\_ALERT "enqueue\_task\_edf\n");

}

}

}

Когда задача больше не запускается, вызывается функция dequeue\_task\_casio, которая отменяет работу функции enqueue\_task\_casio. Обратите внимание: если задача не выполняется, она удаляется из связанного списка (rem\_casio\_task\_list).

static void dequeue\_task\_edf(struct rq \*rq, struct task\_struct \*p, int sleep)

{

struct edf\_task \*t=NULL;

char msg[edf\_MSG\_SIZE];

if(p){

t=find\_edf\_task\_list(&rq->edf\_rq,p);

if(t){

snprintf(msg,edf\_MSG\_SIZE,"(%d:%d:%llu)",t->task->edf\_id,t->task->pid,t->absolute\_deadline);

register\_edf\_event(sched\_clock(), msg, edf\_DEQUEUE);

remove\_edf\_task\_rb\_tree(&rq->edf\_rq, t);

atomic\_dec(&rq->edf\_rq.nr\_running);

if(t->task->state==TASK\_DEAD || t->task->state==EXIT\_DEAD || t->task->state==EXIT\_ZOMBIE){

rem\_edf\_task\_list(&rq->edf\_rq,t->task);

}

}

else{

printk(KERN\_ALERT "dequeue\_task\_edf\n");

}

}

}

Как следует из названия, функция check\_preempt\_curr\_casio проверяет, должна ли выполняться текущая запущенная задача. Эта функция вызывается после постановки на очередь или снятия с очереди задачи и только устанавливает флаг, который указывает ядру планировщика, что текущая выполняемая задача должна быть вытеснена (с помощью функции resched\_task). Указатель struct task\_struct rq-> curr указывает на задачу, которая в данный момент выполняется на процессоре.

Текущая задача должна быть выгружена (листинг ниже): если в очереди выполнения и задаче, назначенной в данный момент для процессора (rq-> curr) есть хотя бы одна задача, а также если на красно-чёрном дереве есть задача с более ранним сроком.

static void check\_preempt\_curr\_EDF(struct rq \*rq, struct task\_struct \*p)

{

struct EDF\_task \*t=NULL,\*curr=NULL;

if(rq->curr->policy!=SCHED\_EDF){

resched\_task(rq->curr);

}

else{

t=earliest\_deadline\_EDF\_task\_rb\_tree(&rq->EDF\_rq);

if(t){

curr=find\_EDF\_task\_list(&rq->EDF\_rq,rq->curr);

if(curr){

if(t->absolute\_deadline < curr->absolute\_deadline)

resched\_task(rq->curr);

}

else{

printk(KERN\_ALERT "check\_preempt\_curr\_EDF\n");

}

}

}

}

Функция pick\_next\_task\_edf выбирает задачу, которая будет выполнена текущим процессором. Эта функция вызывается ядром планировщика всякий раз, когда текущая задача помечена для вытеснения.

Данная функции выбирает задачу с самым ранним абсолютным предельным сроком (earliest\_deadline\_edf\_task\_rb\_tree). Стоит обратить внимание, что если на процессоре нет задачи, то эта функция возвращает NULL, и таким образом ядро планировщика пытается найти одну задачу в следующем классе планирования с низким приоритетом.

static struct task\_struct \*pick\_next\_task\_edf(struct rq \*rq)

{

struct edf\_task \*t=NULL;

t=earliest\_deadline\_edf\_task\_rb\_tree(&rq->edf\_rq);

if(t){

return t->task;

}

return NULL;

}

После написания нового модуля планировщика необходимо включить его в основной файл планировщика (sched.c), а также сообщить ядру планировщика, что edf\_sched\_class является наиболее приоритетным планировщиком

В листинге ниже показаны изменения в sched.c для включения файла, реализующего модуль EDF (sched\_edf.c), а также для информирования ядра планировщика о том, что edf\_sched\_class теперь является планировщиком с наивысшим приоритетом.

#include "sched\_rt.c"

#ifdef CONFIG\_SCHED\_DEBUG

#include "sched\_debug.c"

#endif

#ifdef CONFIG\_SCHED\_EDF\_POLICY

#include "sched\_edf.c"

#endif

#ifdef CONFIG\_SCHED\_EDF\_POLICY

#define sched\_class\_highest (&edf\_sched\_class)

#else

#define sched\_class\_highest (&rt\_sched\_class)

#endif

Для планирования задачи, во-первых, она должна присутствовать в системе. Первоначально задача создается как любая задача в системе, используя системный вызов fork. После этого для того, чтобы быть задачей EDF, необходимо изменить свою политику планирования.

Чтобы задать требуемые параметры планирования задачи, необходимо изменить структуру данных struct sched\_param. Это нужно сделать в двух файлах: /usr/include/bits/sched.h и /kernel\_source-code/include/linux/sched.h

В структуру struct sched\_param были добавлены два поля. Одним из них является идентификатор (edf\_id), а другой используется для установки относительного срока (крайнего срока) задачи.

struct sched\_param {

int sched\_priority;

#ifdef CONFIG\_SCHED\_EDF\_POLICY

unsigned int edf\_id;

unsigned long long deadline;

#endif

};

Исходная функция ядра rt\_policy (определенная в файле /kernel-source-code/kernel/sched.c) используется для определения, относится ли данная политика планирования к классу реального времени (SCHED\_RR и SCHED\_FIFO). Теперь включим в неё политику SCHED\_EDF:

static inline int rt\_policy(int policy)

{

if (unlikely(policy == SCHED\_FIFO) || unlikely(policy == SCHED\_RR)

#ifdef CONFIG\_SCHED\_CASIO\_POLICY

|| unlikely(policy == SCHED\_EDF)

#endif

){

return 1;

}

return 0;

}

Статический приоритет - это приоритет, назначенный процессу при его запуске. Он может быть изменен с помощью системных вызовов nice и sched\_setscheduler. Это делается путем установки поля класса sched для переменной struct, которая представляет задачу в системе с адресом новой переменной класса планирования.

static void

\_\_setscheduler(struct rq \*rq, struct task\_struct \*p, int policy, int prio)

{

BUG\_ON(p->se.on\_rq);

p->policy = policy;

switch (p->policy) {

case SCHED\_NORMAL:

case SCHED\_BATCH:

case SCHED\_IDLE:

p->sched\_class = &fair\_sched\_class;

break;

case SCHED\_FIFO:

case SCHED\_RR:

p->sched\_class = &rt\_sched\_class;

break;

#ifdef CONFIG\_SCHED\_EDF\_POLICY

case SCHED\_EDF:

p->sched\_class = &edf\_sched\_class;

break;

#endif

}

p->rt\_priority = prio;

p->normal\_prio = normal\_prio(p);

/\* we are holding p->pi\_lock already \*/

p->prio = rt\_mutex\_getprio(p);

set\_load\_weight(p);

}

Функция вызывается функцией sched\_setscheduller, в которую также необходимо внести изменения:

int sched\_setscheduler(struct task\_struct \*p, int policy,

struct sched\_param \*param)

{

/\* double check policy once rq lock held \*/

if (policy < 0)

policy = oldpolicy = p->policy;

else if (policy != SCHED\_FIFO && policy != SCHED\_RR &&

policy != SCHED\_NORMAL && policy != SCHED\_BATCH &&

policy != SCHED\_IDLE

#ifdef CONFIG\_SCHED\_EDF\_POLICY

&& policy!=SCHED\_EDF

#endif

)

return -EINVAL;

...

#ifdef CONFIG\_SCHED\_EDF\_POLICY

if(policy==SCHED\_EDF){

p->deadline=param->deadline;

p->edf\_id=param->edf\_id;

}

#endif

...

#ifdef CONFIG\_SCHED\_EDF\_POLICY

if(policy==SCHED\_EDF){

add\_edf\_task\_2\_list(&rq->edf\_rq, p);

}

#endif

...

}

* 1. **Красно-чёрное дерево**

Красно-черное дерево (англ. red-black tree) - это еще одна форма сбалансированного бинарного поискового дерева. Впервые оно было представлено в 1972 году как еще одна разновидность сбалансированного бинарного дерева. Время поиска, вставки или удаления узла для красно-черного дерева является логарифмической функцией от числа узлов.

Данный тип деревьев отличается от других реализаций следующими свойствами:

* каждый узел ассоциируется с определенным цветом - красным или черным;
* корневой узел может быть любого цвета;
* красные узлы могут иметь только черные дочерние узлы;
* все пути от узла до любого листа, расположенного ниже в дереве, содержат одно и то же количество черных узлов.

Высота красно-черного дерева, состоящего из **N** узлов, лежит в диапазоне от двоичного логарифма log(N+1) до 2 \* log(N+1).

В листинге 1 приведены структуры на языке Си, описывающие узлы красно-черного дерева.

##### Листинг 1. Исходный код узлов, использующихся красно-черном дереве.

/\* структура, описывающая узел красно-черного дерева \*/

struct rb\_node

{

int red;

int data;

struct rb\_node \*link[2];

};

Также для работы с красно-черным деревом потребуется вспомогательная структура из листинга 2.

##### Листинг 2. Структура, описывающая красно-черное дерево

struct rb\_tree

{

struct rb\_node \*root; // указатель на корневой узел

int count; // количество узлов в дереве

};

## Вставка узла в красно-черное дерево

При вставке нового узла в цветное дерево (другое название красно-черного дерева) для него изначально устанавливается красный цвет. Затем для добавляемого элемента выполняется поиск родительского узла и проверка его цвета. Если цвет родителя - черный, то основной критерий цветного дерева сохраняется. Если же родитель - красного цвета, то выполняется итеративная (нерекурсивная) балансировка дерева. В худшем случае время вставки может достичь значения логарифма от числа узлов в красном дереве.

Для упрощения алгоритма предполагается, что листья (узлы, не имеющие потомков) имеют черный цвет. В листинге 3 приведен исходный код функции для определения цвета узла.

##### Листинг 3. Функция для определения цвета узла

int is\_red ( struct rb\_node \*node )

{

return node != NULL && node->red == 1;

}

Для ротации узлов в дереве будут использоваться функции, представленные в листинге 4. Первая функция меняет местами два узла, вторая функция выполняет два таких обмена:

##### Листинг 4. Функции для ротации узлов в красно-черном дереве

/\* функция для однократного поворота узла \*/

struct rb\_node \*rb\_single ( struct rb\_node \*root, int dir )

{

struct rb\_node \*save = root->link[!dir];

root->link[!dir] = save->link[dir];

save->link[dir] = root;

root->red = 1;

save->red = 0;

return save;

}

/\* функция для двукратного поворота узла \*/

struct rb\_node \*rb\_double ( struct rb\_node \*root, int dir )

{

root->link[!dir] = rb\_single ( root->link[!dir], !dir );

return rb\_single ( root, dir );

}

В листинге 5 приведен исходный код функции для создания нового узла.

##### Листинг 5. Функция для создания нового узла

struct rb\_node \*make\_node ( int data )

{

struct rb\_node \*rn = malloc ( sizeof \*rn );

if ( rn != NULL ) {

rn->data = data;

rn->red = 1; /\* –инициализация красным цветом \*/

rn->link[0] = NULL;

rn->link[1] = NULL;

}

return rn;

}

При вставке узла в цветное дерево не требуется прибегать к рекурсии, так как это можно сделать за один проход. В общем случае при вставке нового узла возможны три варианта.

1. происходит изменение цвета;
2. требуется сделать один поворот;
3. требуется сделать двойной поворот.

Для этого в памяти кроме текущего узла нужно хранить еще три уровня дерева: «родителя», «деда» и «прадеда» текущего узла.

## Удаление узла из красно-черного дерева

При удалении узла из цветного дерева цвет родительского узла не изменяется. Удаление красного узла не влечет никаких последствий, коллизию может вызвать только удаление узла черного цвета. Поэтому при удалении черного узла для обеспечения целостности дерева необходимо использовать различные операции: простую смену цвета (если это возможно) или одну или несколько ротаций.

Если при удалении черного узла, его «брат» (узел, находящийся на том же уровне, что и удаляемый) и все их четыре потомка имеют черный, то выполняется изменение цвета. Если «брат» удаляемого узла окрашен в красный цвет, то производится ротация, изображенная на варианте II. Если удаляемый узел и его «брат» черного цвета, а правый потомок «брата» - красного, то выполняется двойная ротация. Если левый потомок «брата» красного, то выполняется одиночная ротация, как в пятом варианте.

1. **Сборка и встраивание ядра**

Перед настройкой ядра необходимо загрузить следующее программное обеспечение для сборки ядра:

* linux-source-2.6.24
* kernel-package
* libncurses5-dev
* build-essential

1. С помощью утилиты wget загружаем исходный код ядра:

sudo wget https://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.24.tar.bz2

1. Распаковываем архив:

tar –xjvf linux-2.6.24.tar.bz2

1. Создаем конфигурационный файл из текущего системного конфигурационного файла:

sudo cp /boot/config-2.6.24 .config

1. Меняем параметр extraversion в Makefile, чтобы отличать собираемое ядро от других версий

EXTRAVERSION= -edf

1. Компилируем ядро:

sudo make oldconfig

sudo make-kpkg –initrd kernel\_image 2>../errors

1. Начинается компиляция ядра, и если все идет хорошо, создается сжатый образ ядра, в противном случае можно проверить ошибки в файле ошибки, созданном в домашнем каталоге
2. Установим скомпилированную версию ядра, сгенерированную предыдущим шагом

sudo dpkg -i kernel\_image-2.6.24\_xxxx.deb

1. Для завершения процесса необходимо перезагрузить систему и выбрать загрузку нового ядра:



Рис.4. Загрузка с новой версии ядра

**Тестирование**

Для тестирования был написан следующий код, устанавливающий процессу политику планирования SCHED\_EDF:

#include <sched.h>

#include <stdio.h>

void printPolicy(){

int policy = sched\_getscheduler(0);

switch (policy) {

case SCHED\_OTHER:

printf ("Обычная политика\n");

break;

case SCHED\_RR:

printf ("Политика карусели\n");

break;

case SCHED\_FIFO:

printf("Политика FIFO\n");

break;

case SCHED\_DEADLINE:

printf("Политика Deadline\n");

break;

case -1:

printf("sched\_getscheduler");

break;

default:

printf("Неизвестная политика!\n");

}

}

int main(){

printPolicy();

struct sched\_param sp = { .sched\_priority = 1 };

int ret;

ret = sched\_setscheduler(0, SCHED\_RR, &sp);

printPolicy();

if (sched\_yield() == 0)

printf("sched\_yield\n");

return 0;

}

Также данный код был апробирован в версиях ядра 3.16 и 4.11, где уже реализована политика планирования EDF под названием SCHED\_DEADLINE.

В завершение был использован набор тестов[5] для проверки политики планирования. Результаты меняются в зависимости от задаваемых приоритетов и временных отрезков:

[1455]

missed deadlines = 2856

missed periods = 821

Total adjustments = 48064

us deadline : 1000

us runtime : 400 us

nr\_periods : 6368

[1460]

missed deadlines = 3802

missed periods = 1026

Total adjustments = 20386 us

deadline : 1000 us

runtime : 400 us

nr\_periods : 5188

**Заключение**

Модульная платформа планирования Linux поставляется с версией ядра Linux 2.6.23. Эта структура стала реализацией новых политик планирования для Linux проще, чем предыдущая. Однако в официальной литературе не было найдено документов, объясняющих, как реализовать новую политику планирования для Linux. Обычно можно найти документацию общего назначения, в которой объясняются все части Linux.

В этом работе были подробно описаны все шаги, необходимые для реализации новой политики планирования. Для этого была изменена версия ядра Linux 2.6.24, чтобы она поддерживала задачи в реальном времени, планируемые в соответствии с алгоритмом EDF.

Представленная в работе реализация является простой. Но решение таких проблем, как прерывания, таймеры и многопроцессорные системы выходят за рамки этой работы.

**Список использованных источников**

1. Лав Р. Ядро Linux //Описание процесса разработки: Пер. с англ.–3-е изд.− М.: Вильямс. – 2013.
2. Лав Р. L. Системное программирование. 2-е изд./Роберт Лав.— Санкт-Петербург: Питер, 2014.— 448 с.
3. Интернет-источник. Implementing a new real-time scheduling policy for Linux: Part 1. http://www.embedded.com/design/operating-systems/4204929/Real-Time-Linux-Scheduling-Part-1/ Дата обращения - 30.04.2017
4. Интернет-источник. Википедия. SCHED\_DEADLINE. https://ru.wikipedia.org/wiki/SCHED\_DEADLINE / Дата обращения - 30.04.2017
5. Интернет-источник. Набор тестов для политики SCHED\_DEADLINE. https://github.com/rostedt/sched\_deadline Дата обращения – 30.04.2017

1. Запросы ввода-вывода упорядочиваются исходя из физического размещения данных на диске таким образом, чтобы головки диска как можно больше перемещались в одном направлении и как можно более упорядоченно. [↑](#footnote-ref-1)