Федеральное государственное бюджетное образовательное учреждение

высшего профессионального образования

|  |  |
| --- | --- |
| **Gerb-BMSTU_01** | ***«Московский государственный технический университет  имени Н.Э. Баумана»***  ***(МГТУ им. Н.Э. Баумана)*** |

ФАКУЛЬТЕТ Информатика и системы управления (ИУ)

КАФЕДРА Операционные системы (ИУ7)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**РАСЧЁТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**к курсовому проекту на тему:**

         Разработка планировщика периодических задач для ядра Linux

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Студент            группы ИУ7-71 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_   Баранин А.В.

(Подпись, дата) (И.О.Фамилия)

Руководитель курсового проекта \_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Подпись, дата) (И.О.Фамилия)

Москва, 2016

Реферат

В ходе выполнения курсового проекта разработан класс планирования процессов SCHED\_FREQUENCY для ядра Linux версии 3.16 на базе исходных кодов дистрибутива Debian 8.1 (amd64), предназначенный для управления выполнением периодических задач реального времени мягкого характера.

Проведено:

1. Анализ исходного кода ядра, формата системных вызовов и структуры планировщика.

2. Определение необходимых для решения задачи новых структур данных.

3. Реализован собственно планировщик и изменены части кода ядра для его использования.

Расчётно-пояснительная записка содержит 23 листа формата А4, 1 изображение.

Приложение содержит 1 оптический диск, содержащий изменённые исходные тексты ядра в соответствующей оригиналу структуре папок.

Оглавление

[Реферат 3](#_Toc444474361)

[Оглавление 4](#_Toc444474362)

[Введение 4](#_Toc444474363)

[1. Аналитическая часть 6](#_Toc444474364)

[1.1 Описание предметной области 6](#_Toc444474365)

[1.2 Описание программного окружения 6](#_Toc444474366)

[1.3 Определение необходимых структур данных 7](#_Toc444474367)

[1.4 Описание политики планирования 9](#_Toc444474368)

[2. Конструкторская часть 13](#_Toc444474369)

[2.1 Обеспечение доступа к планировщику 13](#_Toc444474370)

[2.2 Дополнительные шаги для интеграции в ядро 14](#_Toc444474371)

[2.3 Реализация основных методов класса 16](#_Toc444474372)

[3. Технологическая часть 19](#_Toc444474373)

[3.1 Компиляция и установка 19](#_Toc444474374)

[3.2 Отладка 20](#_Toc444474375)

[3.3 Ошибки и недостатки 20](#_Toc444474376)

[Заключение 21](#_Toc444474377)

[Список использованной литературы 22](#_Toc444474378)

Введение

Системы реального времени используются в различных отраслях человеческой деятельности, начиная от систем управления летательными аппаратами и заканчивая цифровой частью хирургических аппаратов. Способность вычислительной системы выполнять временные требования к обработке информации – востребованное качество, оно достигается вниманием со стороны разработчиков операционных систем. Особым образом должен быть организован вычислительный процесс, в наиболее популярных операционных системах за это отвечает часть ядра – планировщик, управляющая выделением процессорного времени системным и пользовательским задачам.

Стандарт POSIX, реализуемый ядром Linux, предписывает наличие двух классов задач реального времени SCHED\_FIFO и SCHED\_RR, предназначенных для ухода от политики обеспечения минимального времени отклика пользователя в сторону беспрекословного выделения максимально-возможного процессорного времени конкретным задачам. С 2009 года в ядро Linux также входит новый класс SCHED\_DEADLINE (бывший SCHED\_EDF), реализующий алгоритм планирования по ближайшему сроку завершения.

Данная работа посвящена реализации очень простого класса планирования SCHED\_FREQUENCY, предназначенного для постановки на выполнение задач с заданным пользователем периодом активации. В процессе выполнения задачи был получен опыт изменения, компиляции и отладки ядра Linux, а так же точное представление о подсистеме планирования задач в Linux и особенностях различных политик. Исходный код нового планировщика представляет собой один новый .c файл и несколько изменённых .c, .h и make-файлов оригинального ядра.

1. Аналитическая часть

1.1 Описание предметной области

Система реального времени — это система, которая должна реагировать на события во внешней по отношению к системе среде или воздействовать на среду в рамках требуемых временных ограничений.

Операционная система реального времени, ОСРВ — тип операционной системы, основное назначение которой — предоставление необходимого и достаточного набора функций для работы систем реального времени на конкретном аппаратном оборудовании.

Реальное время в операционных системах — это способность операционной системы обеспечить требуемый уровень сервиса в определённый промежуток времени.

В данной работе временные требования содержат лишь один пункт: частоту выполнения. Частота задаётся пользователем при активации нового планировщика и может изменяться в процессе выполнения с помощью системного вызова. На планировщик с целью облегчения реализации не накладываются требования минимизации ошибки тактирования задач, или же умышленного игнорирования слишком долгих задач. В реальности такой планировщик обладал бы положительными свойствами для задач управления, алгоритмы которых выигрывают в точности от равномерности отрезков времени выполнения, но слабой надёжностью из-за отсутствия механизмов борьбы с перегрузкой и пересекающимися тактами задач.

1.2 Описание программного окружения

Выделением процессорного времени пользовательским и системным задачам в Linux версии 3.16 занимается подсистема ядра – планировщик. В неизменённом ядре эта задача решается с помощью 5-и «классов» планирования, организованных в односвязный список и обладающих безоговорочным приоритетом друг над другом. Каждый из классов (структура, поля которой – указатели на «методы») реализует определённую логику планирования процессов в конкретном, непересекающемся с другими, диапазоне приоритетов.

Эти классы: STOP (особые задачи остановки процессоров), DEADLINE (планирование по ближайшему сроку завершения), RT (политики FIFO и round-robin), CFS («полностью справедливый» алгоритм для обычных процессов пользовательского уровня), IDLE (процессор не занят, можно минимизировать нагрузку и тепловыделение). Лишь три центральных из них обладают отражением в интерфейсе ядра, только RT и CFS описаны в POSIX.

stop\_sched\_class → deadline\_sched\_class → rt\_sched\_class → fair\_sched\_class → idle\_sched\_class.

Такая структура внешне модульна, но при реализации возникли явные проблемы архитектурного характера, мешающие лёгкому добавлению новых политик. Классы абстрагируют функцию планирования лишь от половины работы, остальная часть всё равно сильно зависит от конкретных классов и функций, определённых для них вне отдельного файла.

К классам планирования ядром предъявляется требование в реализации определённого набора функций, используемых ядром в течение работы планировщика, а также предоставлен набор необязательных к обработке событий, информирующих класс об определённых изменениях в системе.

Каждый процессор обладает своим экземпляром структуры **struct rq** (run queue), задача которой – хранить контекст планирования, необходимый для работы классов и остальной системы. Большая часть классов планирования имеют собственные поля в этой структуре, необходимые для хранения коллекций задач, приоритетов, временных параметров.

Локальные для каждой задачи данные, необходимые для обработки задачи каждым классом, хранятся как поля в структуре **struct task\_struct**, откуда они доступны планировщику и конкретному классу.

Пример обращения к классам - листинг части кода ядра, определяющего следующую на выполнение задачу:

again:

for\_each\_class(class) {

p = class->pick\_next\_task(rq, prev);

if (p) {

if (unlikely(p == RETRY\_TASK))

goto again;

return p;

}

}

1.3 Определение необходимых структур данных

Вышеописанная архитектура предполагает реализацию нового класса планирования, а так же добавление необходимых для его работы полей в локальную для каждого процессора «очередь выполнения» run queue (struct rq). Также для реализации требуются изменения в структуре task\_struct для учёта в теле задачи различных параметров планирования нового алгоритма.

Исходный код описания нового класса планирования:

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

.next = &fair\_sched\_class, // следующий в списке класс

.enqueue\_task = enqueue\_task\_fq, // добавление задачи в очередь

.dequeue\_task = dequeue\_task\_fq, // удаление задачи из очереди

.yield\_task = yield\_task\_fq, // сдача процессора задачей

.check\_preempt\_curr = check\_preempt\_curr\_fq, // проверка на вытеснение новой задачи

.pick\_next\_task = pick\_next\_task\_fq, // выбор задачи на выполнение

.put\_prev\_task = put\_prev\_task\_fq, // обработчик снятия задачи с выполнения

#ifdef CONFIG\_SMP

// для многопроцессорных систем

.select\_task\_rq = select\_task\_rq\_fq, // выбор процессора для новой задачи

.rq\_online = rq\_online\_fq, // Пробуждение процессора

.rq\_offline = rq\_offline\_fq, // Засыпание процессора

#endif

.set\_curr\_task = set\_curr\_task\_fq, // обработчик постановки задачи на выполнение

.task\_tick = task\_tick\_fq, // обработчик тика таймера

.task\_dead = task\_dead\_fq, // обработчик завершения задачи

.prio\_changed = prio\_changed\_fq, // обработчик смены приоритета

.switched\_from = switched\_from\_fq, // смена политики с FQ на иную

.switched\_to = switched\_to\_fq, // смена политики с иной на FQ

};

Поля структуры класса инициализируются указателями на собственно функции, которые и составляют политику планирования. Они же являются точками входа в новый планировщик.

Так как процесс выбора задачи для выполнения требует сортировки по времени пробуждения, наш класс будет хранить список задач в красно-чёрном дереве, реализация которого уже присутствует в ядре Linux. Это обеспечит логарифмическую сложность операций обновления очереди после изменений в задачах или их количестве. Красно-чёрные деревья требуют наличие записи корня в структуре struct rq и узла дерева в task\_struct. Второе дерево добавлено в очередь по примеру класса DEADLINE, так как далеко не всегда задачи правее текущего в главном дереве могут мигрировать (например, ограниченный набор доступных для процесса процессоров), и лучше хранить доступные для миграции задачи в отдельном дереве.

В struct rq будет добавлено поле fq типа struct fq\_rq:

struct fq\_rq {

struct rb\_root rb\_root; // корень дерева задач

struct rb\_node \*rb\_leftmost; // кэшированное значение левого узла

unsigned long fq\_nr\_running; // число задач в дереве

#ifdef CONFIG\_SMP

struct {

u64 curr\_fin; // ожидаемое время завершения текущей задачи

u64 next\_wakeup; // кешированное значение следующего пробуждения

} earliest\_wakeup; // используется при миграции

unsigned long fq\_nr\_migratory; // число доступных для миграции задач

int overloaded; // флаг, установлен, если более одной задачи

u64 pull\_time; // время последней миграции

struct rb\_root pushable\_fq\_tasks\_root; // корень дерева мигрирующих задач

struct rb\_node \*pushable\_fq\_tasks\_leftmost;

};

struct rq {

...

struct fq\_rq fq;

...

};

Структура task\_struct содержит так называемые «единицы планирования» (scheduling entities) – поля-структуры, используемые различными классами планирования для своих нужд. Мы добавим свою единицу под названием sched\_fq\_entity:

struct sched\_fq\_entity {

struct rb\_node rb\_node; // узел дерева

u64 fq\_period; // тактовый период

s64 runtime; // время выполнения с начала последнего такта

u64 prev\_runtime; // время, потраченное на выполнение задачи в пред. такте

u64 wakeup; // абсолютное время пробуждения

// флаги новой задачи, наследования приоритетов, сдачи процессорного времени

int fq\_new, fq\_boosted, fq\_yielded;

};

struct task\_struct {

...

struct sched\_fq\_entity fq;

...

};

Иных новых структур и типов для реализации не требуется.

1.4 Описание политики планирования

От класса планирования требуется обязательная реализация базового набора функций. Среди них наиболее важные: выбор задачи на выполнение, добавление новой задачи в очередь, подчинённую классу, удаление задачи из очереди и добровольная сдача процессора задачей. Сдача важна, так как периодическим задачи реального времени удобней пользоваться не тяжеловесным вызовом exit(), завершающий процесс и требующий создание нового для выполнения программы в следующий такт, а вызовом sched\_yield(). Событие сдачи процессора будем считать сообщением процесса ядру об окончании такта и необходимостью запланировать его через период.

Словесное описание алгоритма выбора задачи на выполнение:

Запрос из планировщика на выбор задачи для выполнения:

1. Начало

2. Если дерево частотных задач пусто

3. Вернуть NULL

4. Иначе

5. ЛУ := крайний левый узел дерева fq-задач

6. Если время пробуждения ЛУ не позже текущего времени

7. Вернуть ЛУ

8. Иначе

9. Вернуть NULL

10. Конец

Словесное описание события добавления новой задачи в частотную очередь:

Запрос на добавление задачи ЗЧ в очередь fq-планировщика:

1. Начало

2. Если ЗЧ только что создана

3. Время пробуждения ЗЧ := текущее\_время + период

4. Если время пробуждения ЗЧ предшествует текущему

5. Перенести время пробуждения (ВП) ЗЧ за текущее на интервал, кратный периоду:

ВП ЗЧ := ((текущее время - ВП ЗЧ) / период + 1) \* период

6. Установить счётчик времени выполнения задачи в 0

7. Добавить ЗЧ в дерево fq-задач

8. Увеличить счётчики задач вообще и fq-задач

9. Обновить данные о второй задаче в дереве

10.Если ЗЧ не выполняется в данный момент

11. Увеличить счётчик доступных к миграции fq-задач

12. Добавить ЗЧ в дерево доступных к миграции fq-задач

13.Конец

Словесное описание события удаления задачи из частотной очереди:

Запрос на удаление задачи ЗЧ из очереди fq-планировщика:

1. Начало

2. Обновить счётчики времени fq-очереди и задачи ЗЧ

3. Удалить ЗЧ из дерева fq-задач

4. Уменьшить счётчики задач вообще и fq-задач

5. Обновить данные о второй задаче в дереве

6. Если ЗЧ не выполняется в данный момент

7. Уменьшить счётчик доступных к миграции fq-задач

8. Удалить ЗЧ из дерева доступных к миграции fq-задач

9. Конец

Словесное описание события сдачи процессора fq-задачей:

Задача ЗЧ выполняет системный вызов yield():

1. Начало

2. Обновить счётчики времени fq-очереди и задачи ЗЧ

3. Запомнить время выполнения задачи

4. Установить флаг сдачи

5. Перенести время пробуждения (ВП) ЗЧ за текущее на интервал, кратный периоду:

ВП ЗЧ := ((текущее время - ВП ЗЧ) / период + 1) \* период

6. Удалить ЗЧ из дерева fq-задач

7. Добавить ЗЧ в дерево fq-задач на новую позицию

8. Сигнализировать планировщику о необходимости перепланирования

Наш планировщик лишён приоритетов, поэтому вытеснение выполняющейся в данный момент fq-задачи будет выполнятся только в случае наличия задачи более высокого класса, например FIFO.

Выбор процессора для новой fq-задачи реализуем просто: вернём процессор с наименьшим числом fq-задач в очереди.

Обработчики пробуждения процессоров не используются в данной реализации, поэтому функции заполнены по аналогии с этими же функциями класса DEADLINE.

Обработчик события начала выполнения задачи на процессоре просто обновляет запись о времени начала выполнения, а также удаляет задачу из дерева доступных к миграции задач.

Обработчик события тика системного таймера просто обновляет временные параметры. Перепланирование или какая-либо другая сложная логика отсутствуют.

Обработчики смены приоритета и очереди оповещают другие классы о событии, что не имеет отношения к собственно политике планирования.

Схема простой итерации планировщика на одной задаче изображена на рисунке 1.

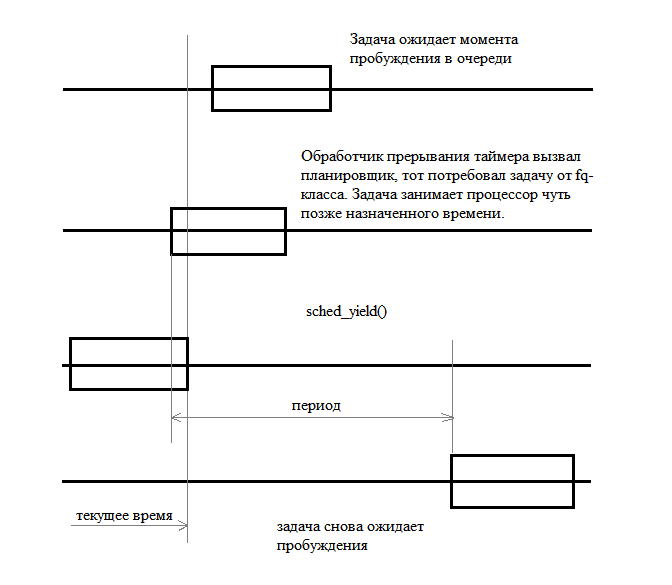


рис. 1

2. Конструкторская часть

2.1 Обеспечение доступа к планировщику

Регистрация класса планировки в ядре системы начинается с его включения в очередь классов. Мы расположим наш класс ниже REAL\_TIME, но выше CFS. На этом же месте возникает проблема выбора цифрового значения приоритета, так как приоритеты RT 0..99 и CFS 100..139 непосредственно прилегают друг к другу. Легко эта проблема не решается, слишком много ссылок на эти константы в ядре подразумевают их последовательное расположение. Был сделан выбор остаться на 99 приоритете и разделять это число с RT.

Изменения в /kernel/sched/rt.c:

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

.next = &fq\_sched\_class, // регистрируем наш класс

...

}

// аналогично, ссылка на cfs\_sched\_class занимает поле .next в нашем fq\_sched\_class.

Далее необходимо адаптировать какой-либо системный вызов для присвоения процессу политики частотного планирования. Вызов sched\_setscheduler (POSIX) позволяет установить тип планировщика, но принимает одним из параметров «обрезанную» структуру sched\_params, не содержащую поля period. Специально для работы с DEADLINE планировщиком ядро экспортирует системный вызов sched\_setattr, принимающий полный набор параметров. Этот вызов не предписан стандартом и потому не представлен в Си библиотеке режима пользователя, поэтому обращение к нему будет проходить с помощью универсального вызова syscall. Пример:

syscall(\_\_NR\_sched\_setattr, pid, &params, 0);

Определим константу SCHED\_FREQUENCY в /include/uapi/linux/sched.h.

Далее адаптируем функцию ядра \_\_sched\_setscheduler (/kernel/sched/core.c) для назначения частотного планировщика задаче:

// исправим выход по неизвестной политике

if (policy != SCHED\_DEADLINE &&

policy != SCHED\_FIFO && policy != SCHED\_RR &&

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

policy != SCHED\_IDLE && policy != SCHED\_FREQUENCY)

return -EINVAL;

а так же \_\_setscheduler(...)

{

...

// добавим присвоение класса задаче

if (dl\_prio(p->prio))

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

else if (fq\_policy(p->policy))

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

...

}

// Инициализатор единицы частотного планирования

static void

\_\_setparam\_fq(struct task\_struct \*p, const struct sched\_attr \*attr)

{

struct sched\_fq\_entity \*fq\_se = &p->fq;

fq\_se->runtime = 0;

fq\_se->prev\_runtime = 0;

fq\_se->wakeup = 0;

fq\_se->fq\_period = attr->sched\_period;

fq\_se->fq\_boosted = 0;

fq\_se->fq\_new = 1;

fq\_se->fq\_yielded = 0;

}

// дополним функцию присвоения параметров

static void \_\_setscheduler\_params(struct task\_struct \*p,

const struct sched\_attr \*attr)

{

...

else if (fq\_policy(policy))

\_\_setparam\_fq(p, attr);

...

}

// дополним инициализатор приоритета

static inline int normal\_prio(struct task\_struct \*p)

{

int prio;

if (task\_has\_dl\_policy(p))

prio = MAX\_DL\_PRIO-1;

else if (task\_has\_rt\_policy(p))

prio = MAX\_RT\_PRIO-1 - p->rt\_priority;

else if (task\_has\_fq\_policy(p))

prio = MAX\_RT\_PRIO-1; // вернёт 99

else

prio = \_\_normal\_prio(p);

return prio;

}

2.2 Дополнительные шаги для интеграции в ядро

К сожалению, задания нового класса и его реализации недостаточно для его нормальной работы. Также нужно учесть механизмы инициализации очереди выполнения, единицы планирования, механизмов наследования приоритета и др.

Инициализация **rq** происходит в функции void sched\_init(void) (/kernel/sched/core.c):

void \_\_init sched\_init(void)

{

...

for\_each\_possible\_cpu(i)

{

struct rq \*rq;

rq = cpu\_rq(i);

raw\_spin\_lock\_init(&rq->lock);

...

init\_cfs\_rq(&rq->cfs);

init\_rt\_rq(&rq->rt, rq);

init\_dl\_rq(&rq->dl, rq);

init\_fq\_rq(&rq->fq, rq);

...

}

...

}

Инициализация принадлежащих процессу структур происходит в функции создания процесса – fork, а конкретно в \_\_sched\_fork(...):

static void \_\_sched\_fork(unsigned long clone\_flags, struct task\_struct \*p)

{

...

RB\_CLEAR\_NODE(&p->fq.rb\_node); // инициализация вершин деревьев

p->fq.fq\_period = 0;

p->fq.runtime = 0;

p->fq.prev\_runtime = 0;

p->fq.wakeup = 0;

...

RB\_CLEAR\_NODE(&p->pushable\_fq\_tasks);

...

}

Механизм наследования приоритетов, необходимый для работы примитива rt\_mutex, также требует корректировки:

void rt\_mutex\_setprio(struct task\_struct \*p, int prio)

{

...

// любой вызов этой функции сопровождается изменением приоритета

// поэтому установим флаг

if (fq\_policy(p->policy))

p->fq.fq\_boosted = 1;

...

if (rt\_prio(prio)) {

...

// мы на территории приоритетов класса RT, аккуратно учтём

if (fq\_prio(prio) && p->fq.fq\_boosted)

{

// мы вернулись на оригинальный уровень

p->fq.fq\_boosted = 0;

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

}

else

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

...

}

2.3 Реализация основных методов класса

Рассмотрим реализацию наиболее важных функций частотного класса, начиная с кода добавления новой задачи в очередь (дерево).

static void enqueue\_task\_fq(struct rq \*rq, struct task\_struct \*p, int flags)

{

enqueue\_fq\_entity(&p->fq, flags); // добавить в очередь задачу

// если она не в стадии выполнения и не привязана к одному процессору

if (!task\_current(rq, p) && p->nr\_cpus\_allowed > 1)

{

// увеличить счётчики миграции и добавить в дерево

inc\_fq\_migration(&p->fq, &rq->fq);

// потенциальных эмигрантов, аналогична \_\_enqueue\_fq\_entity(...)

enqueue\_pushable\_fq\_task(rq, p);

}

}

static void enqueue\_fq\_entity(struct sched\_fq\_entity \*fq\_se, int flags)

{

update\_fq\_entity(fq\_se); // обновить временные характеристики задачи

\_\_enqueue\_fq\_entity(fq\_se); // добавить в дерево

}

static void update\_fq\_entity(struct sched\_fq\_entity \*fq\_se)

{

struct fq\_rq \*fq\_rq = fq\_rq\_of\_se(fq\_se);

struct rq \*rq = rq\_of\_fq\_rq(fq\_rq);

if (fq\_se->fq\_new) {

setup\_new\_fq\_entity(fq\_se); // инициализировать новую задачу

return;

}

// проверить, не опаздывает ли задача

if (fq\_time\_before(fq\_se->wakeup, rq\_clock(rq)))

{

// перенести в будущее

fq\_se->wakeup = fq\_se->wakeup +

(1 + (rq\_clock(rq) - fq\_se->wakeup) / fq\_se->fq\_period)

\* fq\_se->fq\_period;

fq\_se->runtime = 0; // обнулить временя такта

}

}

static void \_\_enqueue\_fq\_entity(struct sched\_fq\_entity \*fq\_se)

{

struct fq\_rq \*fq\_rq = fq\_rq\_of\_se(fq\_se);

struct rb\_node \*\*link = &fq\_rq->rb\_root.rb\_node;

struct rb\_node \*parent = NULL;

struct sched\_fq\_entity \*entry;

int leftmost = 1;

// для отладки, валит систему при добавлении задачи, уже находящейся в fq-дереве

BUG\_ON(!RB\_EMPTY\_NODE(&fq\_se->rb\_node));

// спуск по дереву согласно сравнению времени пробуждения

while (\*link) {

parent = \*link;

entry = rb\_entry(parent, struct sched\_fq\_entity, rb\_node);

// если новая задача проснётся раньше узла, спуск влево

if (fq\_time\_before(fq\_se->wakeup, entry->wakeup))

link = &parent->rb\_left;

else

{

link = &parent->rb\_right;

leftmost = 0;

}

}

if (leftmost)

fq\_rq->rb\_leftmost = &fq\_se->rb\_node;

// вставка в дерево

rb\_link\_node(&fq\_se->rb\_node, parent, link);

// и обновление дерева

rb\_insert\_color(&fq\_se->rb\_node, &fq\_rq->rb\_root);

inc\_fq\_tasks(fq\_se, fq\_rq);

}

static inline void setup\_new\_fq\_entity(struct sched\_fq\_entity \*fq\_se)

{

struct fq\_rq \*fq\_rq = fq\_rq\_of\_se(fq\_se);

struct rq \*rq = rq\_of\_fq\_rq(fq\_rq);

// в первый раз запустим задачу через период

fq\_se->wakeup = rq\_clock(rq) + fq\_se->fq\_period;

fq\_se->runtime = 0;

fq\_se->prev\_runtime = 0;

fq\_se->fq\_new = 0;

}

Задача выбора процесса для выполнения решается проще благодаря деревьям, сформированным выше.

struct task\_struct \*pick\_next\_task\_fq(struct rq \*rq, struct task\_struct \*prev)

{

struct sched\_fq\_entity \*fq\_se;

struct task\_struct \*p;

struct fq\_rq \*fq\_rq;

fq\_rq = &rq->fq;

if (rq\_clock(rq) - fq\_rq->pull\_time > (u64)FREQ\_PULL\_PERIOD)

{

// периодически будем подтягивать задачи с других процессоров

fq\_rq->pull\_time = rq\_clock(rq);

pull\_fq\_task(rq);

// RCU-блокировка в pull может пропустить ненужную задачу в очередь

if (rq->stop && rq->stop->on\_rq)

return RETRY\_TASK;

}

// предыдущая задача может нуждаться в обновлении временных характеристик

if (prev->sched\_class == &fq\_sched\_class)

update\_curr\_fq(rq);

// если нет задач, вернём NULL, планировщик обратится к менее приоритетным классам

if (!fq\_rq->fq\_nr\_running)

return NULL;

fq\_se = pick\_next\_fq\_entity(rq, fq\_rq); // простая функция выбора

if (!fq\_se)

return NULL;

put\_prev\_task(rq, prev); // укажем о снятии с процессора предыдущей задачи

p = fq\_task\_of(fq\_se);

p->se.exec\_start = rq\_clock(rq);

dequeue\_pushable\_fq\_task(rq, p); // изымаем новую задачу из эмигрантов

return p;

}

static struct sched\_fq\_entity \*pick\_next\_fq\_entity(struct rq \*rq, struct fq\_rq \*fq\_rq)

{

struct sched\_fq\_entity \*ret;

struct rb\_node \*left = fq\_rq->rb\_leftmost;

if (!left)

return NULL;

// берём левый лист

ret = rb\_entry(left, struct sched\_fq\_entity, rb\_node);

if (ret)

{

// если время активации в прошлом, выбираем задачу, иначе - NULL

if (ret->wakeup > rq\_clock(rq))

return NULL;

else

return ret;

}

return NULL;

}

Изъятие из очереди (содержимое функций – зеркало функций добавления)

static void dequeue\_task\_fq(struct rq \*rq, struct task\_struct \*p, int flags)

{

update\_curr\_fq(rq);

\_\_dequeue\_task\_fq(rq, p, flags);

}

static void \_\_dequeue\_task\_fq(struct rq \*rq, struct task\_struct \*p, int flags)

{

dequeue\_fq\_entity(&p->fq);

if ((p->nr\_cpus\_allowed > 1) && !task\_current(rq, p))

{

dec\_fq\_migration(&p->fq, &rq->fq);

dequeue\_pushable\_fq\_task(rq, p);

}

}

static void dequeue\_fq\_entity(struct sched\_fq\_entity \*fq\_se)

{

struct fq\_rq \*fq\_rq = fq\_rq\_of\_se(fq\_se);

if (RB\_EMPTY\_NODE(&fq\_se->rb\_node))

return;

if (fq\_rq->rb\_leftmost == &fq\_se->rb\_node) {

struct rb\_node \*next\_node;

next\_node = rb\_next(&fq\_se->rb\_node);

fq\_rq->rb\_leftmost = next\_node;

}

// удаление из кч-дерева требует лишь одного вызова

rb\_erase(&fq\_se->rb\_node, &fq\_rq->rb\_root);

RB\_CLEAR\_NODE(&fq\_se->rb\_node);

dec\_fq\_tasks(fq\_se, fq\_rq);

}

И, наконец, обработчик yield():

static void yield\_task\_fq(struct rq \*rq)

{

struct task\_struct \*p = rq->curr;

// актуализируем временные параметры завершающей такт задачи

update\_curr\_fq(rq);

if (p->fq.runtime > 0)

{

p->fq.prev\_runtime = p->fq.runtime; // сохраним время выполнения

p->fq.fq\_yielded = 1;

p->fq.runtime = 0;

}

// перенесём в будущее кратно периоду

if (p->fq.fq\_period > 0)

p->fq.wakeup += ((rq\_clock(rq) - p->fq.wakeup) / p->fq.fq\_period + 1) \*

p->fq.fq\_period;

else

p->fq.wakeup = rq\_clock(rq) + (u64)10000000;

// перестроим дерево

\_\_dequeue\_task\_fq(rq, p, 0);

enqueue\_task\_fq(rq, p, ENQUEUE\_REPLENISH);

}

3. Технологическая часть

3.1 Компиляция и установка

Исходный код Linux распространяется в стандартной для сообщества форме сжатого .tar архива. Конфигурация, компиляция и установка полностью производятся с помощью GNU make. В процессе выполнения работы использована 64-битная конфигурация с установленным флагом SMP. Система работает на проприетарной виртуальной машине VMWare, поддерживающей аппаратную виртуализацию. К ОС-хосту система присоединена через виртуальную сеть и эмулятор COM-порта, что позволяет устанавливать ssh-соединение.

Полное время сборки и установки на моём домашнем компьютере составляло один час, незначительные изменения только в .c файлах (не трогая файлы заголовков) позволили свести итерацию в район восьми-десяти минут.

3.2 Отладка

Отладка операционной системы в режиме ядра – сложная задача. Начиная с версии ядра 2.6 Linux распространяется со встроенным ассемблерным отладчиком kdb и его интерфейсом kgdb, предназначенным для отладки на уровне исходного кода языка Си через терминал или последовательный порт. Несмотря на полноту их возможностей и отсутствие альтернатив, пришлось прибегнуть к пробной версии утилиты VisualKernel, предоставляющей возможности удалённой разработки модулей ядра в окружении Microsoft Visual Studio. На гостевой машине процессом сборки и отладки руководит всё так же make и kdb, однако управляет ими программа из операционной системы-хоста.

Visual Studio предоставляет удобные средства навигации по дереву исходного кода, а плагин VisualKernel тесно интегрирован с отладочным средством VMWare – vmware debug stub. Такой вариант отладки показал более высокую скорость и надёжность, чем стандартный интерфейс kgdb + COM.

Из-за невозможности компиляции ядра без установленных флагов оптимизации, объектные файлы предоставляют тяжело отлаживаемый, перемещённый, сокращённый набор инструкций. Локальные переменные и параметры функций почти всегда недоступны для чтения отладчиком, так как не имеют физического представления в памяти и регистрах виртуальной машины. Порядок операций в функциях также существенно изменён, многие функции опущены или совмещены с другими операциями. Это сводит функции отладчика собственно ядра к трассировке и созданию breakpoint-ов, что, несмотря на выше обозначенные проблемы, всё равно существенно облегчает разработку и поиск ошибок.

3.3 Ошибки и недостатки

Выполненная реализация слабо отлажена. Наблюдаются проблемы в обработке частотными процессами сигналов, а также наглядны проблемы с блокировкой cfs-процессов во время работы частотных. Стандартные функции ввода-вывода демонстрируют долгое время отклика, что может говорить об ещё не обнаруженных ошибках реализации, а так же о плохой кооперации механизмов синхронизации оригинального ядра и нового класса планирования. Также, в процессе отладки не было ни одной остановки выполнения на breakpoint-е в коде наследования приоритетов, чему объяснения я дать не могу.

Хотя порядок и время выполнения тестового примера говорят о том, что разветвлённые процессы выполняются в верном порядке и с необходимой частотой, планировщик очень далёк от отлаженного состояния.

Заключение

В процессе реализации программы был получен ценный опыт модификации ядра Linux, а также представление об уровне компетенции, требуемом от системного разработчика. Хотя работоспособность частотного планировщика и сомнительна, большая часть его функционала отлажена и демонстрирует «нормальный» порядок работы без аварийного завершения системы или её блокировки. Исправление существующих недостатков требует гораздо более широкого понимания всех механизмов ядра и взаимосвязей между ними, так как степень абстракции компонентов в исходном коде Linux сравнительно мала. Хороший уровень осведомлённости только о подсистеме планировки не может гарантировать легкой разработки изменений в ней, так как она активно используется в других частях ядра, нередко используя излишне предосудительные интерфейсы.

Список использованной литературы

1. «Linux Kernel Development» Robert Love, 2010
2. «Process Scheduling in Linux» Volker Seeker – University of Edinburgh, 2013

3. «Real-time Linux (Xenomai)» брошюра автора Harco Kuppens, 2011

4. «Implementing a new real-time scheduling policy for Linux» Paulo Baltarejo Sousa и Luis Lino Ferreira, Политехнический институт города Порто, 2010

5. «ProfessionalLinux Kernel Architecture» Wolfgang Mauerer, изд. Wiley Publishing, Inc., 2008

6. Документация и обзорные материалы с <https://www.kernel.org>

7. «UNIX изнутри» Ю. Вахалия 2003

8. «Operating System Concepts» A. Silberschatz, P.B.Galvin, G.Gagne, Wiley Publishing Inc. 2013