Правительство Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Санкт-Петербургский государственный университет»

Кафедра Системного Программирования

Логинова Вита Владимировна

Разработка планировщика ОСРВ с совмещением кооперативной и вытесняющей многозадачности

Дипломная работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

Научный руководитель:
асп. Козлов А. П.

Рецензент:
д. ф.-м. н., профессор Терехов А. Н.

Санкт-Петербург
2015

SAINT-PETERSBURG STATE UNIVERSITY

Software Engineering Chair

Vita Loginova

Development of RTOS scheduler combining preemptive and cooperative multitasking

Graduation Thesis

Admitted for defence.
Head of the chair:
professor A. N. Terekhov

Scientific supervisor:
postgraduate A. P. Kozlov

Reviewer:
professor A. N. Terekhov

Saint-Petersburg
2015

Оглавление

Введение

Системы реального времени (СРВ) — это системы, которые взаимодействуют со средой посредством обработки поступающей информации и предоставлением результата вычислений за конечное и определенное время [14].

Системы реального времени часто выступают как встраиваемые и используются в самых разных устройствах: в промышленных системах, в бытовой технике, развлекательных устройствах, периферийных устройствах, в медицинском, военном и космическом оборудовании [18].

Аппаратное обеспечение встраиваемых систем выбирается для решения определенных задач и не предусматривает возможности расширения функциональности. Такой подход позволяет снизить стоимость всей системы, но влечет за собой существенные ограничения в ресурсах [18].

Встраиваемые системы используются для разного рода задач. Во-первых, это задачи, для которых требуется реакция на внешние асинхронные события, например, обработка прерываний или сетевых пакетов. Во-вторых, задачи последовательной обработки данных, такие как кодирование и декодирование, передача объемных данных, шифрование. Сложные встраиваемые системы решают как те, так и другие задачи.

При разработке программного обеспечения для встраиваемых систем используются специализированные операционные системы (ОС). Неотъемлемой частью любой современной ОС является многозадачность.

Существует два основных типа многозадачности: кооперативная и вытесняющая [15]. Вытесняющая от кооперативной отличается главным образом тем, что перепланирование происходит не только при ожидании и добровольной приостановке потока[1], но также при независящих

[1]Обычно в теории операционных систем негласно подразумевается, что поток имеет стек. В контексте данной дипломной работы под потоком будет иметься в виду минимальный объект, управляемый планировщиком, вне зависимости от свойств и деталей реализации. Термины “поток” и “единица планирования” выступают синонимами.

от потока событий (появление более приоритетного потока, истечение кванта времени). Кооперативная многозадачность, помимо прочего, может использоваться и в событийно-ориентированной модели[1], [12], [2], в которой многозадачность может быть реализована с помощью потоков без стека, что позволяет сильно сократить используемые ресурсы.

Наличие лишь кооперативной многозадачности, как, например, в ОС Contiki [3] или TinyOS [13], делает систему применимой только для узкого класса устройств. Поэтому часто ОС поддерживает вытесняемую многозадачность (например, QNX [11]) или же оба типа. Потоки ядра в таких ОС, как правило, вытесняемые, и ими заведует центральный планировщик системы. Однако некоторое множество единиц планирования, например, средства для отложенной обработки прерываний, часто контролируются другими, специализированными планировщиками. Часть из них изолирована и не поддерживает синхронизацию с основными потоками, как softirq и tasklet Linux [8] или coroutine FreeRTOS [5], а часть интегрирована в основной планировщик, например, workqueue Linux [8].

Наличие в ядре ОС единого централизованного планировщика, координирующего взаимодействие как между вытесняемыми единицами, так и между кооперативными, могло бы упростить разработку ПО для встраиваемых систем. Реализация классических средств синхронизации для единиц без стека позволила бы расширить круг их применения, а также использовать меньший объем оперативной памяти. Кроме того, присутствие разных по типу потоков в единой приоритетной очереди дало бы возможность более гибко распределять ресурсы процессора между ними.

На кафедре системного программирования уже много лет ведутся исследования и разработки, связанные со встраиваемыми системами. В частности, в сотрудничестве с кафедрой разрабатывается операционная система Embox [4], [17].

Embox — это конфигурируемая ОС для встраиваемых решений. Такие ее свойства, как модульность и несложная переносимость, позволяют использовать ее для широкого класса устройств.

На момент написания данной дипломной работы в Embox разные планировщики заведовали разными типами потоков. Таким образом, была поставлена задача разработки планировщика, совмещающего различные типы потоков, и его реализации в рамках проекта Embox.

1. Постановка задачи

Целью данной работы является разработка планировщика, который совмещает разные типы многозадачности и использует общую очередь для различных типов единиц планирования. Для достижения цели были сформулированы следующие задачи:

1. Разработать архитектуру планировщика, удовлетворяющего указанным требованиям;
2. Реализовать разработанный планировщик в ядре ОСРВ Embox;
3. Произвести оценку полученной реализации.

2. Обзор

Для разработки архитектуры решения необходимо было изучить различные подходы к планированию и реализации планировщиков в современных ОСРВ и встраиваемых ОС. Каждая многозадачная операционная система включает в себя один и более планировщиков. При рассмотрении каждой системы и единиц планирования необходимо было ответить на вопросы:

1. Для какого класса устройств подходит ОС?
2. Какие типы многозадачности поддерживает ОС, за какой тип отвечают конкретные потоки?
3. Насколько тяжеловесны потоки, имеют ли свой стек?
4. Возможна ли синхронизация потоков между собой, с другими типами потоков?
5. Изолированны ли разные типы потоков друг от друга, используется ли для их планирования общая очередь?

2.1. Используемая терминология

Сопрограмма — обобщение понятие подпрограммы, поддерживающее несколько входных точек, остановку и последующее выполнение с заданного места [16].

Отложенная обработка прерываний — механизм, позволяющий разделить обработку аппаратного прерывания на две части: одна часть выполняется непосредственно в момент прерывания, а вторая откладывается для выполнения в менее строгом контексте [9].

2.2. ОС Linux

ОС Linux [8] хоть и не считается операционной системой реального времени, но подходы, которые в ней применяются, в том или ином

виде используются и в более специализированных системах. В Linux есть несколько различных видов планировщиков. Основой планировщик системы в полной мере поддерживает вытесняющую многозадачность. Он достаточно сложен и тяжеловесен, так как содержит множество оптимизаций, связанных, например, с многоядерной архитектурой и механизмом синхронизации RCU. Поэтому в обзоре уделяется внимание более легковесным и специфичным планировщикам, которые обрабатывают отложенные прерывания.

2.2.1. Softirq

Softirq — это один из базовых механизмов отложенной обработки прерываний в ядре Linux [8]. Обработчики softirq регистрируются статически с фиксированным приоритетом. Они выполняются в специальном вычислительном контексте и могут вытесняться только аппаратными прерываниями.

Механизм softirq имеет слишком много ограничений и считается устаревшим. Более того, уже давно ведутся разговоры о том, чтобы исключить этот механизм из состава ОС окончательно, особенно в контексте реального времени [7]. Однако softirq связан с тасклетами, которые рассматриваются в следующем подразделе.

2.2.2. Tasklet

Тасклеты (tasklet) ядра Linux [8] по своей структуре напоминают облегченную версию потоков. Они выполняются атомарно и не могут использовать никакие примитивы синхронизации, реализованные на механизме ожидания [9].

Для планирования тасклетов существует специальная функция-планировщик, предоставляющая примитивное кооперативное планирование. Эта функция-планировщик регистрируется как обработчик softirq. Тасклеты по приоритету делятся на два типа: на обычные и с высоким приоритетом. Разделение идет за счет использования разных очередей и двух экземпляров планировщика.

За счет простого устройства тасклеты обрабатываются быстро и занимают мало памяти.

2.2.3. Workqueue

Так как функциональность тасклетов очень ограничена, а при обработке прерываний возникает необходимость в синхронизации, в ядре Linux реализован альтернативный механизм — workqueue [8].

Механизм workqueue имеет сравнительно сложную реализацию со множеством оптимизаций, связанных с мультипроцессорной архитектурой и специфичными возможностями ядра Linux [9]. Однако саму концепцию можно представить достаточно просто. Она представлена на рис. 1.

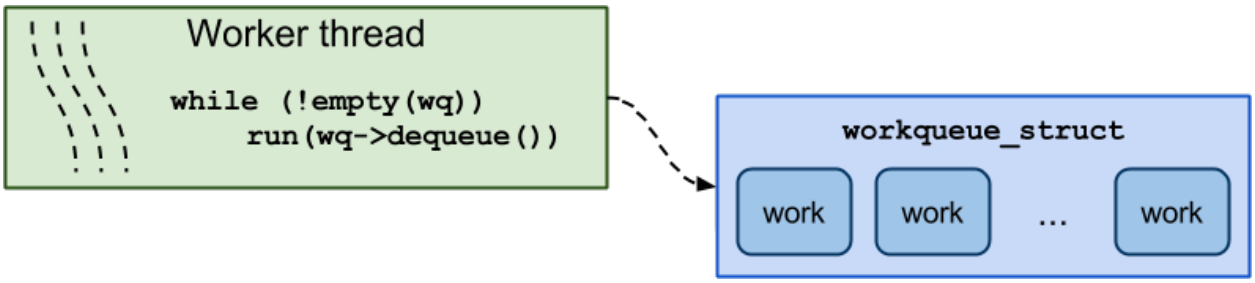

Рис. 1: Упрощенная иллюстрация работы механизма workqueue в ОС Linux.

Работа (work) — легковесный аналог потока. При планировании работа помещается в очередь workqueue. Роль планировщика здесь выполняют специальные потоки ядра (kernel thread), называемые worker. Пока такой поток не завершит конкретную работу, он не может начать выполнять следующую. Однако worker-потоков может быть несколько, их число масштабируется в зависимости от загруженности системы. За счет того, что это обычные потоки, они поддерживают все виды примитивов синхронизации и позволяют различным работам синхронизироваться между собой.

Таким образом, за один квант времени один worker-поток может выполнить сразу несколько работ, и переключение контекста при этом не требуется. В этом случае работы отрабатываются достаточно быстро. Однако, если функция работы достаточно большая или требуется

синхронизация, происходит переключение между worker-потоками, то есть, как между обычными потоками. Поэтому, хоть в целом этот механизм работает быстрее обычных потоков, но все же он куда медленнее тасклетов и потребляет больше памяти.

В пределах одного worker-потока работы исполняются строго последовательно, что напоминает кооперативную многозадачность. Однако worker-потоки, будучи потоками ядра, вытесняются согласно стратегии основного планировщика.

2.3. ОС Contiki и protothreads

Contiki — это компактная многозадачная ОС [3], используемая на устройствах с ограниченными ресурсами. В Contiki существуют механизмы для поддержания и вытесняющей, и кооперативной, событийно-ориентированной многозадачности. Однако как таковую вытесняющую многозадачность ОС не предоставляет, вместо этого она реализуется как отдельная, опционально подключаемая библиотека. Поэтому в обзоре рассматривается только кооперативная многозадачность.

Protothreads [10] — это часть Contiki, которая включает в себя легкие потоки, называемые протопотоками, планировщик и различные функции взаимодействия с ними. Реализация протопотоков основана на сопрограммах. Все основные функции управления протопотоками: создание, манипуляции с сохранением состояния, синхронизация и так далее — реализованы с помощью макросов. Это делает код с одной стороны более читабельным, с другой стороны — менее очевидным.

Протопотоки управляются событийно-ориентированным кооперативным планировщиком, поддерживающим приоритеты. У протопотоков нет стека, и их запуск эквивалентен запуску функции. Как следствие, они экономны как с точки зрения производительности, так и с точки зрения используемой памяти.

Синхронизация между протопотоками происходит с помощью аналогов таких примитивов как мьютексы, семафоры, события.

Одно из важных замечаний по использованию протопотоков и неко-

торых других реализаций сопрограмм — это необходимость аккуратно обходиться с локальными переменными. Отсутствие стека у сопрограмм влечет за собой невалидные значения локальных переменных при повторном входе в сопрограмму.

2.4. TinyOS

TinyOS [13], как и Contiki, основывается на событийно-ориентированной парадигме и используется для систем с ограниченными ресурсами, в основном для беспроводных систем датчиков. В отличие от других ОС, рассматриваемых в этом обзоре, TinyOS написана на своем языке nesC, диалекте C. Это позволяет средствами языка определять свойства функций, секций кода и т.д.

В качестве планируемых объектов в TinyOS присутствуют легковесные бесстековые задачи, они выполняются атомарно и координируются кооперативным планировщиком. Задачи относятся к синхронным объектам системы.

Кроме синхронных объектов в ОС есть асинхронные (относительно задач), например, обработчики прерываний. Такие объекты системы, как команды и события могут быть как синхронными, так и асинхронными, что определяется ключевым словом языка nesC. Причем, асинхронные объекты не могут вызывать синхронные.

Так как в ОС присутствуют только кооперативные задачи, нет особой необходимости в средствах синхронизации. Те секции кода, запись и чтение в которых должны быть защищены от асинхронных объектов, окружаются специальным блоком, помечающим код как атомарный.

2.5. FreeRTOS

В операционной системе FreeRTOS [5] реализованы два основных планировщика: вытесняющий для задач и кооперативный для сопрограмм. В контексте FreeRTOS под сопрограммой будет подразумеваться конкретная реализация концепции. Сопрограммы в этой операционной системе очень похожи на протопотоки и имеют те же ограничения

и преимущества.

В документации FreeRTOS предлагается запускать планировщик для сопрограмм либо отдельно, либо внутри самой низкой по приоритету задачи, так называемой idle-задачи. Во втором случае происходит совмещение кооперативной и классической многозадачности. Однако в таком случае сопрограммы будут выполняться после всех остальных задач. Есть возможность поместить планировщик сопрограмм в другую задачу, но это все равно не обеспечит гибкой настройки приоритетов, так как планирование происходит в разных очередях. Кроме того, средства синхронизации реализованы только для задач, и взаимодействовать сопрограммы могут только друг с другом.

2.6. QNX

QNX [11] — это распространенная ОСРВ для встраиваемых систем. Многозадачность в QNX организуется одним центральным вытесняющим планировщиком. Для взаимодействия между потоками реализованы различные средства синхронизации.

Роль обработчиков отложенных прерываний в QNX играют обычные потоки. Таким образом, в QNX можно пользоваться преимуществами общей очереди. Однако, так как каждому потоку требуется свой стек, такая реализация неприменима на устройствах с сильно ограниченными ресурсами.

2.7. Сравнение

В таблице 1 приведена сводка сравнения поддерживаемой многозадачности в рассмотренных системах. По таблице видно, что ни в одной операционной системе многозадачность не удовлетворяет всем поставленным требованиям.

В тех ОС, где планирование осуществляется единым планировщиком, не поддерживаются оба типа многозадачности. В других же системах разные типы потоков изолированы друг от друга и не поддерживают синхронизации друг с другом. Изолированность означает так-

		Тип многозадачности	Стек потоков	Синхронизация	Общая ли очередь?
Linux	Tasklet	Кооп.	Нет	Нет	Нет
	Workqueue	Вытесн.	Есть	Да	
FreeRTOS	Task	Вытесн.	Есть	Да	Нет
	Coroutine	Кооп.	Нет	Нет	
Contiki	Protothread	Кооп.	Нет	Да	Да
TinyOS		Кооп.	Нет	Нет	Да
QNX		Вытесн.	Есть	Да	Да

Таблица 1: Сравнение подходов к многозадачности в современных ОСРВ и Linux.

же то, что, по сути, одному планировщику назначается более высокий приоритет, а другому — более низкий, то есть приоритеты жестко разграничены.

2.8. Embox

На момент написания данной дипломной работы в Embox было два планировщика: softirq, наподобие аналогичного в Linux, и основной планировщик системы, поддерживающий классическую многозадачность.

Кроме того, на студенческом проекте 2013-2014 учебного года, где я выступала одним из руководителей, началась разработка прототипа облегченной версии потока, так называемых легких потоков. Легкие потоки были реализованы как атомарные функции. По функциональности они мало отличались от механизма softirq, но планирование происходило уже в общей очереди планировщика с помощью абстрактной структуры. И хоть прототип не вошел в основную версию Embox, работа над ним показала работоспособность идеи.

Таким образом, передо мной стояла задача проработать архитектуру обобщенного планировщика, реализовать легкие потоки, опираясь на прототип, но уже с более широкими возможностями.

3. Архитектура совмещенного планировщика

Архитектура совмещенного планировщика должна быть сконструирована таким образом, чтобы итоговый планировщик распределял ресурсы процессора между потоками различных типов, абстрагируясь от конкретных реализаций. При этом потоки должны лежать в одной очереди, чтобы потоку можно был назначить любое допустимое значение приоритета вне зависимости от типа.

3.1. Обобщенная единица планирования

Для этих целей была разработана обобщенная единица планирования. Она служит для поддержания низкой связности компонент архитектуры за счет наследования. Диаграмма классов планировщика представлена на рис 2.

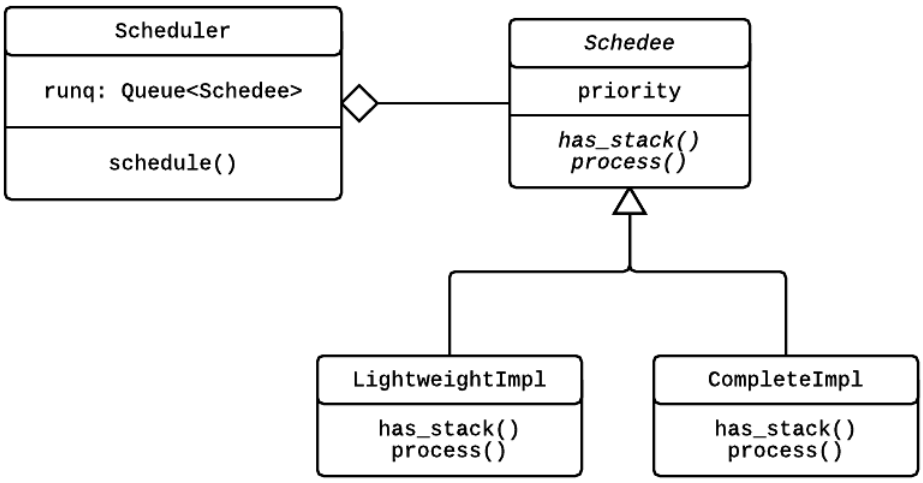

Рис. 2: Диаграмма классов совмещенного планировщика.

На диаграмме `schedee` — это родительский абстрактный класс различных типов единиц планирования. Все манипуляции с потоками в совмещенном планировщике происходят посредством этого класса. Кроме общих для всех типов свойств (ссылка в очереди планирования, приоритет и т.д.) планировщику понадобится обращение к абстрактным

свойствам `schedee`, реализуемым индивидуально для каждого типа. Рассмотрим различные свойства потоков:

- Наличие поддержки вытеснения. Если поток вытесняемый, то перепланирование происходит при явной передачи управления планировщику, при появлении более приоритетной сущности или по истечению кванта времени. Без поддержки вытеснения перепланирование инициируется только явным вызовом планировщика. С точки зрения планировщика разница есть только в том, когда он вызывается, так что нет необходимости в обращении к этому свойству;

- Наличие стека. Наличие стека подразумевает смену контекста при переключении потоков. Контекст включает в себя указатель на стек, выполняемую инструкцию и т.д. Отсутствие стека делает невозможным ожидание со сменой контекста и, как следствие, вытеснение (по крайней мере, полноценное). Наличие или отсутствие стека является принципиальным для планировщика, поэтому `schedee` должен как-то предоставить доступ к этому свойству (абстрактная функция `has_stack()` на диаграмме).

Задача планировщика — определить, какой поток будет исполняться следующим, провести подготовку и делегировать обработку конкретному `schedee`, вызвав абстрактную функцию `process()`. Эта функция выполняет все специальные действия по обработке конкретного `schedee`.

3.2. Схема алгоритма

Все схемы в этом разделе приведены только в общем виде без технических подробностей. Реальная реализация будет содержать, например, проверки корректности, синхронизацию с критическими контекстами, поддержку многоядерности.

Прежде чем перейти к схеме алгоритма обобщенного планировщика, рассмотрим отдельно схемы планировщиков для потоков со стеком

и без. Когда планировщик работает со стековыми потоками (листинг 1), он выполняет две функции: определяет следующий на исполнение поток и делает смену контекста. Планировщик вызывается на стеке текущего потока, поэтому по выходу из функции поток продолжает свое исполнение с того места, где произошло переключение. Потоки со стеком могут как поддерживать вытеснение, так и не поддерживать. На рис. 3 представлены различные сценарии вызова планировщика для потока со стеком. Видно, что во всех случаях планировщик ведет себя одинаково.

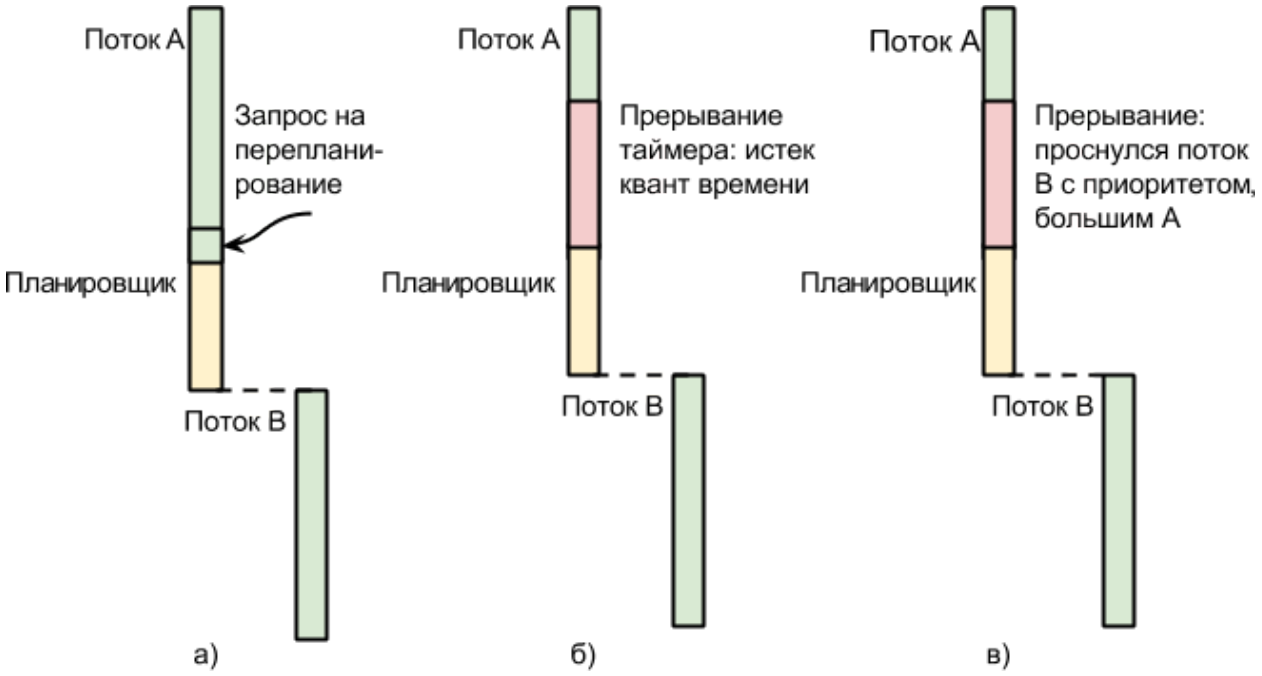

Рис. 3: Сценарии вызова планировщика для потоков со стеком. Перепланирование происходит а) по запросу, б) по истечению кванта времени, в) при появлении более приоритетного потока.

```
schedule() {
  prev = get_prev_schedee();
  if (!prev.is_sleeping())
    runq.insert(prev);
  next = runq.extract();
  switch_context(prev, next);
}
```

Листинг 1: Схема функции планировщика для потоков со стеком.

В листинге 2 представлена схема планировщика для бесстековых потоков. Здесь обработка потоков происходит прямо в планировщике, в цикле. По такой схеме работают тасклеты [8] и протопотоки [10].

```
schedule() {
  while (true)
    run(runq.extract());
}
```

Листинг 2: Схема функции планировщика для потоков без стека.

В совмещенной схеме в зависимости от того, есть стек или нет, нужно либо сменить контекст и выйти из функции планировщика, либо выполнить основную функцию. Какие именно операции будут произведены — останется скрытым от планировщика. Итоговая схема представлена в листинге 3.

```
schedule() {
  prev = get_prev_schedee();
  if (!prev.is_sleeping())
    runq.insert(prev);
  do {
    next = runq.extract();
    next.process();
  } while(next.has_stack());
}
```

Листинг 3: Схема функции совмещенного планировщика.

3.3. Отложенные прерывания

Совмещенный планировщик позволяет планировать в общей очереди не только различные типы потоков ядра и пользовательские потоки, но также и отложенные прерывания. Таким образом, потокам, решающим задачу реального времени, могут быть назначены более высокие приоритеты, чем некоторым отложенным прерываниям.

В качестве примера такой задачи можно привести станок ЧПУ, который, помимо основной программы, предоставляет информацию о статусе работы станка через веб-интерфейс, выступая в качестве веб-сервера. Так как эта функциональность не является основной, она не должна мешать выполнению основной программы. Приоритеты в системе должны быть распределены согласно рис. 4. Такого распределения не получилось бы достичь без использования общей очереди.

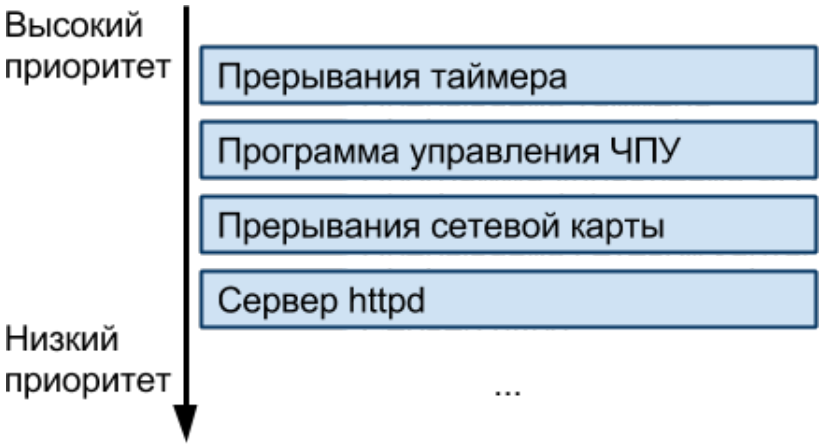

Рис. 4: Распределение приоритетов на примере задачи станка ЧПУ.

Общая очередь также позволяет реализовать средства синхронизации, которыми можно пользоваться из разных типов потоков, в том числе и из отложенных прерываний. Инверсия приоритетов, проблема которой имеет большое значение в СРВ, избегается средствами механизма наследования приоритетов, невозможного без общей очереди.

4. Реализация планировщика

Предложенная архитектура была реализована в рамках проекта Embox. Программная реализация планировщика непосредственно включает в себя интерфейс для взаимодействия с обобщенной единицей планирования и алгоритм планирования. Помимо этого, было необходимо разработать и внедрить новый тип единиц планирования, кооперативные бесстековые легкие потоки.

4.1. Обобщенная единица планирования

В качестве обобщенной единицы планирования, с которой взаимодействует планировщик, в Embox выступает структура `schedee`. Как уже было сказано в обзоре, в Embox уже существовали вытесняемые потоки. Для того чтобы изменения, вызванные реализацией совмещенного планировщика, были минимальны, поля структуры `schedee` были выделены из структуры потоков. Структура `schedee`, которая представлена в листинге 4, включает в себя только необходимые непосредственно планировщику компоненты.

```
struct schedee {
  /* Ссылка в очереди планирования. */
  runq_item_t runq_link;
  /* Поле для блокировки структуры при некоторых операциях
     планировщика. */
  spinlock_t lock;
  /* Специфичная функция, которую вызывает планировщик для
     обработки конкретного schedee. Возвращает указатель на
     schedee, в контексте которого продолжается исполнение. */
  struct schedee *(*process)(struct schedee *prev,
     struct schedee *next);
  /* Поля состояния schedee в машине состояний планировщика. */
  unsigned int active;
  unsigned int ready;
```

```
    unsigned int waiting;
    /* Поле, специфичное для мультипроцессорной архитектуры. */
    struct affinity affinity;
    /* Для вычисления времени исполнения конкретного schedee. */
    struct sched_timing sched_timing;
    /* Приоритет в очереди планирования. */
    struct schedee_priority priority;
    /* Ссылка в очереди ожидания. */
    struct waitq_link waitq_link;
};
```

Листинг 4: Структура schedee

Чтобы планировщик мог отличать единицы планирования со стеком и без, используется специальная сигнатура функции `process()`. Функция возвращает ссылку на структуру `schedee`, на стек которой произошло переключение. Если стек у потока отсутствует, то функция возвращает нулевой указатель.

Инфраструктура планировщика включает в себя различные модули, которые реализуют интерфейс для управления приоритетом единиц планирования (в том числе наследование приоритетов), базовый механизм ожидания, стратегию планирования и так далее. До появления `schedee` все перечисленные модули работали только с полноценными потоками, структурой `thread`. Теперь их интерфейс обновлен и взаимодействует со `schedee` и, следовательно, доступен для использования разными типами потоков.

4.2. Алгоритм планировщика

Алгоритм планировщика переработан согласно рассмотренной в предыдущем разделе схеме. Функция планировщика представлена в приложении А.

В текущей реализации потоки без стека выполняются на стеке последнего стекового потока. Такое решение небезопасно с точки зрения переполнения стека (в Linux, например, для подобных целей выделяет-

ся специальный стек для обработки softirq [8]), но уместно в контексте встраиваемых систем с ограниченными ресурсами. В будущем в Embox можно будет реализовать и другое поведение без существенных изменений в функции планировщика.

4.3. Легкие потоки

Как уже было сказано в обзоре, на момент написания работы был реализован прототип легких потоков. В рамках дипломной работы прототип был полностью переработан и расширен, в частности, были добавлены интерфейс сопрограмм и поддержка синхронизации.

Легкие потоки Embox кооперативные, не имеют стека и поддерживают несколько точек входа, то есть являются сопрограммами. Структура легких потоков `lthread` представлена в листинге 5.

```
struct lthread {
  /* Обобщенная единица планирования. */
  struct schedee schedee;
  /* Основаня функция легкого потока. */
  int (*run)(struct lthread *);
  /* Смещение относительно начальной метки, для управления
     множественными точками входа. */
  ptrdiff_t label_offset;
  /* Специфичная структура, содержащая необходимую
     информацию для механизма ожидания. */
  struct sched_wait_info info;
  /* Ссылка на schedee, ожидающего завершения легкого потока. */
  struct schedee *joining;
};
```

Листинг 5: Структура легких потоков Embox.

Основная функция исполнения легких потоков задается функцией `run()`. В качестве входного параметра функция принимает указатель на свой, текущий легкий поток. Такая сигнатура выбрана согласно

предположению, что указатель на текущий легкий поток будет использоваться практически всегда. Например, так как у легких потоков отсутствует свой стек, часто структура легких потоков будет содержаться в другой структуре. В этой внешней структуре будет храниться информация, которую необходимо сохранять от вызова к вызову. Доступ к ней будет осуществляться за счет приведения типов. Также множество функций API легких потоков и `schedee` принимают на вход в качестве параметра указатель на конкретный легкий поток или его `schedee`, чтобы лишний раз не обращаться к глобальной переменной процессора, где хранится указатель на исполняемый `schedee`.

Работа с несколькими входными точками осуществляется за счет использования меток и переходов к ним. Для этого в структуре легких потоков сохраняется отступ от начальной входной точки до той, с которой необходимо начать исполнение функции. Это не стандартная возможность языка C, а расширение Labels as Values компилятора GCC [6], который используется в Embox. Это решение похоже на переходы с помощью оператора `switch`, который используется в библиотеке Protothread [10], где в качестве состояния используется номер строки. Предлагаемое в дипломной работе решение более явное, так как от программиста не скрывается, в каком месте и куда осуществляется переход. Функции для работы с метками легких потоков представлены в листинге 6.

```
/* Шаблон использования: goto lthread_resume(lt, start); */
void *lthread_resume(struct lthread *lt, void *
    start_lbl) {
  return start_lbl + lt->label_offset;
}

/* Шаблон использования: return lthread_yield(start,
    resume_point); */
int lthread_yield(void *start_lbl, void *target_lbl) {
  return target_lbl - start_lbl;
```

```
}
```

Листинг 6: Функции для работы с метками легких потоков Embox.

Также с помощью меток реализовано взаимодействие с механизмом ожидания и, в частности, со средствами синхронизации. Если легкому или обычному потоку нужно заснуть, он обязан передать управление планировщику. Так как легкие потоки кооперативные и не имеют стека, то осуществляется это явным выходом из функции. Поэтому в случае необходимости передать управление планировщику, все функции, связанные с ожиданием легких потоков, возвращают код ошибки `EAGAIN`. В следующий раз продолжить исполнение необходимо с вызова этой же функции. Пример работы с мьютексом представлен в листинге 7.

```
static int example(struct lthread *self) {
  goto *lthread_resume(self, &&start);
start:
  // мьютекс еще не захвачен

mutex_retry:
  if (mutex_trylock_lthread(self, &mtx) == -EAGAIN) {
    return lthread_yield(&&start, &&mutex_retry);
  }
  // код выполняется с захваченным мьютексом
  ...

  mutex_unlock_lthread(self, &mtx);

  // мьютекс уже освобожден
  return 0;
}
```

Листинг 7: Пример использования мьютексов в легких потоках Embox.

4.4. Обработка отложенных прерываний

После реализации обобщенного планировщика и легких потоков обработка отложенных прерываний была переведена на легкие потоки. Таким образом, теперь в Embox можно пользоваться преимуществами, описанными в разделе 3.3.

Кроме того, раньше отложенными прерываниями заведовал механизм softirq, то есть обработчики и их приоритеты задавались статически. Теперь же можно менять их приоритеты прямо во время исполнения.

5. Эксперименты

5.1. Сравнение с FreeRTOS, Contiki и QNX

Для сравнения были выбраны ОС с тремя разными подходами: QNX с поддержкой только вытесняемой многозадачности, Contiki только с кооперативной и FreeRTOS с обеими.

В QNX обычные потоки обрабатывают отложенные прерывания, поэтому, как и в Embox, можно гибко распределять приоритеты между отложенными прерываниями и пользовательскими задачами. Главным недостатком QNX по сравнению Embox является то, что невозможно использовать многозадачность QNX на устройствах с ограниченными ресурсами из-за необходимости каждому потоку выделять свой стек.

FreeRTOS не имеет этого недостатка. В ней поддерживаются и вытесняемые потоки, и кооперативные, сопрограммы. Но во FreeRTOS, в отличие от Embox, разные классы потоков изолированы друг от друга и не могут синхронизироваться между собой и планируются в разных очередях. Возможно, именно это делает сопрограммы FreeRTOS редко применимыми на практике [5].

Contiki разрабатывалась для работы на устройствах с малым объемом памяти, поэтому поддержка вытесняемой многозадачности здесь присутствует только за счет сторонних библиотек. И даже если использовать оба типа многозадачности, планировщики будут изолированы друг друга. Тем не менее, в силу своей компактности, достигаемой за счет узкой функциональности, в том числе в рамках многозадачности, Contiki может применяться на тех устройствах, на которых Embox не поместится даже с минимальной конфигурацией. Таким образом, Embox и Contiki по-прежнему применимы для разных классов устройств, хоть и множества этих классов пересекаются после появления в Embox легких потоков.

5.2. Оценка производительности

Для оценки производительности легких потоков по сравнению с полноценными было написано приложение, запускающее подряд N потоков, каждый из которых завершает исполнение сразу после запуска. Фиксировалось время перед запуском потоков и сразу после их отработки. Таким образом, оценивается жизненный цикл потока, который для основной функции является накладными расходами. Кроме того, измеряется время, на которое отключаются аппаратные прерывания во время работы планировщика. Измерения проводились на компьютере с процессором Intel Core i7 3517U 1900 МГц. Ядро операционной системы собиралось компилятором GCC с флагом оптимизации O2.

В таблице 2 представлены результаты измерений, полученные от 6 запусков тестов. Стандартное отклонение не превышает 1,1% для данных о легких потоках и 4,9% о полноценных. Относительно больший разброс значений в случае с полноценными потоками обусловлен более сложной обработкой и невозможностью оптимизации компилятором и процессором некоторых участков кода.

	Число потоков	Полноценные потоки	Легкие потоки	Разница
Среднее время исполнения одного потока	3	5414 (∼0,3%)	418 (∼0,5%)	∼92,3%
	10	5307 (∼0,4%)	274 (∼0,2%)	∼94,8%
	30	5360 (∼2,4%)	235 (∼0,2%)	∼95,6%
	50	5285 (∼1,1%)	226 (∼0,2%)	∼95,7%
	100	5278 (∼0,6%)	221 (∼0,2%)	∼95,8%
Средняя длительность блокировки прерываний	3	405 (∼3,4%)	169 (∼1,0%)	∼58,3%
	10	432 (∼4,8%)	128 (∼0,4%)	∼70,4%
	30	436 (∼1,6%)	115 (∼0,9%)	∼73,6%
	50	419 (∼1,8%)	113 (∼0,4%)	∼73,0%
	100	428 (∼1,9%)	112 (∼0,5%)	∼73,8%

Таблица 2: Результаты тестов на производительность легких и полноценных потоков. Данные указаны в тактах (в скобках значение стандартного отклонения).

Легкие потоки имеют куда меньшую задержку (выигрыш более 90%) по сравнению с полноценными потоками и на более короткое время

блокируют прерывания (60—70%). Это также связано с более сложной обработкой потоков. Переключение контекста, к тому же, происходит с отключенными прерываниями. По таблице видно, что в случае легких потоков полученные величины зависят обратно от числа потоков. Это связано с тем, что обработка бесстековых потоков происходит в цикле внутри планировщика, а, значит, накладные расходы на начало планирования и его завершение участвуют в обработке единожды.

5.3. Сравнение образов разной конфигурации

Одним из преимуществ совмещенного планировщика Embox является его конфигурируемость, то есть возможность включить в образ только необходимые типы потоков. Для сравнения характеристик образов использовалось демо-приложение, работающее на двух параллельных потоках. В одном образе это были легкие потоки, в другом — полноценные. Приложение с обеими конфигурациями было запущено на плате STM32VLDISCOVERY с 8 КБ RAM и 128 КБ flash памяти. В результате образы обладали характеристиками, подставленными в таблице 3. Объем стеков подбирался оптимальным образом, чтобы не возникло переполнения, но без лишнего запаса.

		Оба типа	Только легкие	Разница
Flash	Text Os	30152	28724	1428 (4,7%)
RAM	data	436	436	3040 (37%)
	bss	3102	3168	
	Основной стек+ стек потоков	1078 + 1782*2	1536 + 0	

Таблица 3: Характеристики образов Embox с разными конфигурациями планировщика для платы STM32VLDISCOVERY. Данные указаны в байтах.

Хоть в итоге и удалось запустить Embox на этой плате с вытесняющей многозадачностью, уместить больше двух потоков на ней бы не получилось. Однако в случае с системой, основанной на легких потоках, остается еще почти 3 КБ RAM для более сложного приложения с большим числом легких потоков.

Заключение

В рамках дипломной работы разработан планировщик, который совмещает кооперативную и вытесняющую многозадачность и использует общую приоритетную очередь для разных типов единиц планирования. Архитектура планировщика включает в себя обобщенную единицу планирования и алгоритм с поддержкой различных типов единиц планирования. Такая архитектура дает преимущества в обработке отложенных прерываний и хорошо подходит для устройств с ограниченными ресурсами. Использование общей приоритетной очереди позволяет исполнять задачи реального времени вперед некоторых отложенных прерываний.

Предложенная архитектура реализована в ядре ОСРВ Embox. В частности, переработан существующий планировщик ОС, разработаны кооперативные единицы планирования и добавлен программный интерфейс для управления ими.

Было выполнено сравнение планировщика Embox с FreeRTOS, Contiki и QNX. Сравнение показало, что, в отличие от других систем, планировщик Embox в сочетании с реализацией легких потоков позволяет на общих правах взаимодействовать как вытесняемым, так кооперативными единицами.

Кроме того, были проведены эксперименты в рамках Embox. Сравнение производительности полного жизненного цикла легких и полноценных потоков Embox показало выигрыш порядка 90% в случае использования легких потоков. Помимо этого, легкие потоки блокируют обработку аппаратных прерываний на существенно меньшее время, разница составляет примерно 60—70%.

Для устройства STM32VLDISCOVERY проведено сравнение образов Embox с различными конфигурациями планировщика: только с легкими потоками и с обоими типами. Использование только легких потоков позволило сэкономить 1428 байт ROM и 3040 байт RAM, большую часть которой занимали стеки потоков. Сравнение показало возможность использования Embox с поддержкой многозадачности для устройств с ограниченными ресурсами.

Список литературы

[1] Adam Dunkels Oliver Schmidt Thiemo Voigt-Muneeb Ali. Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems // Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006).

[2] Atul Adya Jon Howell Marvin Theimer-William J. Bolosky John R. Douceur. Cooperative Task Management without Manual Stack Management or, Event-driven Programming is Not the Opposite of Threaded Programming // Proceedings of the 2002 Usenix Annual Technical Conference.

[3] Contiki website. — URL: `http://www.contiki-os.org/` (online; accessed: 05.06.2015).

[4] Embox source code ropository. — URL: `https://github.com/embox/embox` (online; accessed: 05.06.2015).

[5] FreeRTOS website. — URL: `http://www.freertos.org/index.html` (online; accessed: 05.06.2015).

[6] GNU compilers manual website. — URL: `https://gcc.gnu.org/onlinedocs/gcc/index.html` (online; accessed: 05.06.2015).

[7] Gleixner Thomas. The 3.6.1-rt1 release announce. — 2012. — URL: `https://lkml.org/lkml/2012/10/9/408` (online; accessed: 05.06.2015).

[8] Inc Linux Kernel Organization. Linux source code 3.19.6. — URL: `https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.19.6.tar.xz` (online; accessed: 05.06.2015).

[9] Kroah-Hartman Jonathan Corbet Alessandro Rubini Greg. Linux Device Drivers. — O'Reilly Media, 2005.

[10] Protothread website. — URL: `http://dunkels.com/adam/pt/` (online; accessed: 05.06.2015).

[11] QNX documentation website. — URL: http://www.qnx.com/developers/docs/am11/index.jsp (online; accessed: 05.06.2015).

[12] Silvana Rossetto Noemi Rodriguez. A cooperative multitasking model for networked sensors // Proceedings of the 26th IEEE International Conference on Distributed Computing Systems Workshops, 2006.

[13] TinyOS website. — URL: http://www.tinyos.net/ (online; accessed: 05.06.2015).

[14] Young S. J. Real Time Languages: Design and Development. — Chichester : Ellis Horwood, 1982.

[15] А Таненбаум Э. Вудхалл. Операционные системы. Разработка и реализация. — СПб. : Питер, 2006.

[16] Кнут Дональд. Искусство программирования, том 1. Основные алгоритмы. — М. : Вильямс, 2006.

[17] Сайт кафедры Системного Программирования. — URL: http://se.math.spbu.ru/SE (online; accessed: 05.06.2015).

[18] Э. Таненбаум. Архитектура компьютера. — СПб. : Питер, 2013.

A. Функция планировщика Embox

```
static void __schedule(int preempt) {
  struct schedee *prev, *next;

  prev = schedee_get_current();

  assert(!sched_in_interrupt());
  spin_lock_ipl(&rq.lock);

  if (!preempt && prev->waiting)
    prev->ready = false;
  else
    __sched_enqueue(prev);

  sched_timing_stop(prev);

  while (1) {
    next = runq_extract(&rq.queue);
    spin_unlock(&rq.lock);

    schedee_set_current(next);

    next = next->process(prev, next);
    if (next) {
      break;
    }

    spin_lock_ipl(&rq.lock);
  }

  sched_timing_start(next);
}
```