Заглянем под капот FX-RTOS

Дмитрий Алексеев (ЭРЕМЕКС)

Многозадачность – ключевая функция операционных систем и главный фактор их возникновения как класса ПО. Использование операционных систем во встроенном программном обеспечении микроконтроллеров широко распространено, но полемика вокруг целесообразности этого подхода ведётся до сих пор, а сама тема сохраняет атмосферу загадочности. В статье разбирается многозадачность на примере ядра российской ОСРВ с открытым исходным кодом FX-RTOS Nanokernel Lite [1]. Рассматриваются технические решения, которые реализованы для удовлетворения потребностей систем реального времени.
В данной статье практически не затрагивается взаимодействие потоков, прерывания, программные таймеры и другие аспекты. Но модульная архитектура позволяет, абстрагируясь от других компонентов, последовательно изучить каждую подсистему по мере необходимости. Понимание логики работы внутренних механизмов ОС, реализующих многозадачность, необходимо, чтобы создавать корректные и эффективные программы.

Введение

В этой статье будет говориться о многозадачности как о многопоточности в том смысле, что на каждую задачу приходятся один или несколько потоков исполнения. Поток исполнения (или «нить», от англ. thread) – часть программы, выполняющаяся непрерывно, как будто на отдельном «виртуальном» процессоре параллельно другим потокам. Текущее состояние процессора характеризуется содержимым его регистров, называемым контекстом. Смена потока достигается за счёт переключения контекста (Context Switch). В ходе исполнения нить неоднократно вытесняется и продолжается с того места, где была прервана (рис. 1). Набор правил, который определяет порядок выполнения задач, называется алгоритмом планирования. Процедура выделения процессора для потока, выбранного алгоритмом планирования, называется диспетчеризацией [2, 3].

Программная реализация отдельных процессов, как правило, не применяется в микроконтроллерных ОС ввиду отсутствия аппаратной поддержки виртуальной памяти и ограниченности ресурсов, и управление задачами сводится к планированию нитей. Потоки находятся в едином физическом адресном пространстве, взаимодействуя через программные объекты, расположенные в совместно используемой памяти. Рассмотрим классический паттерн производитель – потребитель для двух нитей: один поток (назовём его «производитель») снимает информацию с датчика, обрабатывает и записывает её в определённую структуру. Второй поток (потребитель) периодически считывает эту структуру и отображает в текстовом виде на дисплее. Допустим, такая схема взаимодействия работает безошибочно 99% времени, но в редких случаях на экране возникают ошибки вследствие того, что потребитель забирает данные, когда они ещё не полностью записаны. Про такой алгоритм взаимодействия говорится, что он содержит гонки или условия гонок (*Race Condition*). Для того чтобы обеспечить корректность доступа к общим данным, необходимо упорядочить поведение потоков с помощью специальных объектов, предоставляемых ядром ОС: примитивов синхронизации. Один из таких объектов называется «семафор» – по аналогии с железнодорожным семафором, сигнализирующим о занятом участке пути [4]. Любые обращения потока к примитивам синхронизации, как и к другим службам ядра, выполняются через специальные процедуры: системные вызовы. Системный вызов может возвратить ответ одномоментно либо заблокировать поток в ожидании запрошенного ресурса.

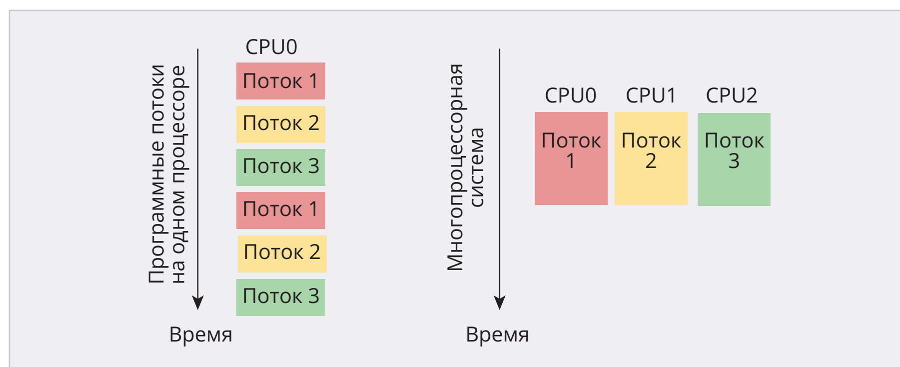

Рис. 1. Разделение процессорного времени между задачами создаёт абстракцию параллельности их выполнения

Политики планирования

Определением следующего потока исполнения и момента времени диспетчеризации занимается компонент ядра ОС – планировщик. Алгоритмы планирования исходят из критерия выбора следующего потока, наиболее применимого в конкретном случае. Самые распространённые политики планирования: первым поступил – первым обслужен (FIFO), по наименьшему остающемуся времени до дедлайна (EDF), циклическое планирование (Round-Robin), частотно-монотонное планирование (RMS) [5]. Разработчики ОСРВ реализуют наиболее унифицированные алгоритмы, базирующиеся на вытеснении текущего потока наиболее высокоприоритетным и распределении времени между потоками одного приоритета. Такие планировщики называются вытесняющими. Одновременно может происходить передача управления между потоками одного приоритета по собственной инициативе (кооперативное планирование). Если код потока инициирует передачу управления или блокируется в ожидании некоторого события, диспетчеризация происходит незамедлительно, планировщик выбирает следующий поток с наивысшим приоритетом либо первый в очереди среди потоков с одинаково высоким приоритетом (рис. 2).

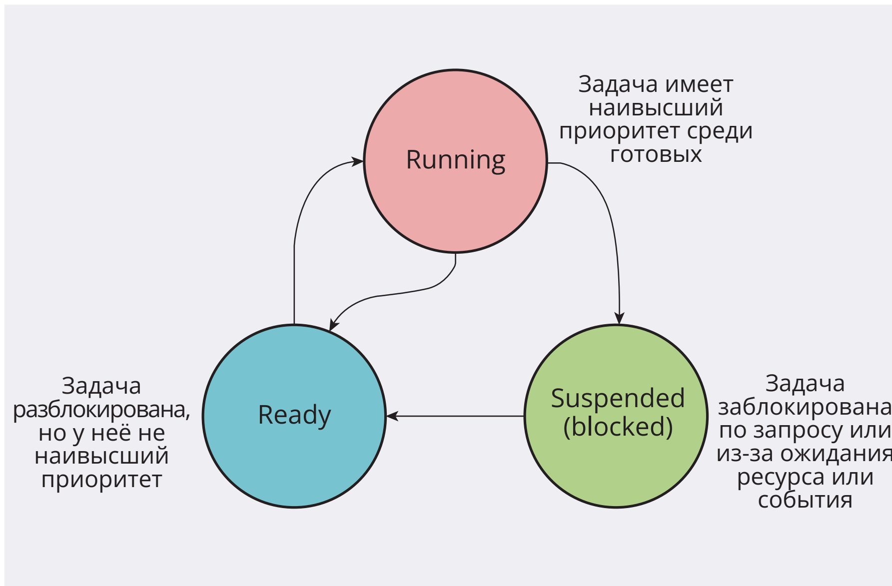

Рис. 2. Диаграмма состояний выполняющейся задачи

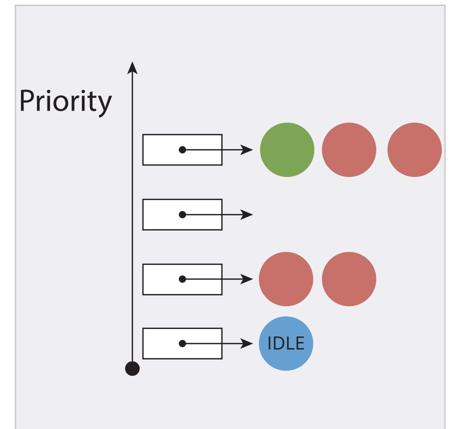

Рис. 3. Схема работы алгоритма планирования MPQ. Зелёным кругом обозначен выполняющийся активный поток, красные круги – потоки, ожидающие в очередях. Синий круг – системный idle-поток

В FX-RTOS реализован алгоритм планирования MLQ (Multi-Level Priority Queue), действующий согласно следующим принципам.

1. Неактивные, но готовые к выполнению потоки группируются в отдельные очереди с одинаковым приоритетом. Количество очередей равно количеству приоритетов потоков, определённых в системе. Заблокированные потоки отсутствуют в очередях (рис. 3).
2. Первой обрабатывается очередь с наивысшим приоритетом до полного исчерпания потоков, готовых к выполнению.
3. Диспетчеризация потоков внутри одной очереди может быть устроена по порядку (FIFO) либо с разделением по времени (Round-Robin).
4. При активации более приоритетного потока выполняется незамедлительное вытеснение менее значимого.
5. Приостановленные потоки в планировании не участвуют.

К основным недостаткам данного подхода можно отнести отсутствие со стороны планировщика гарантий выполнения потоков с наименьшими приоритетами при повышенной активности высокоприоритетных.

Структура исходных текстов FX-RTOS

Модули FX-RTOS можно классифицировать по группам: FX (кроссплатформенные высокоуровневые компоненты ОСРВ), RTL (библиотека универсальных алгоритмов и структур данных), HAL (слой реализации низкоуровневых функций ОС для различного аппаратного обеспечения), HW (низкоуровневые функции для взаимодействия с аппаратурой через регистры процессора). Далее внимание будет уделяться файлам с префиксами *fx_thread, fx_sched, fx_spl, hal* и *hw*. Иерархическая структура системы показана на рис. 4.

Синхронизация внутри ядра и блокировки

Многопоточный код обязан отвечать требованиям надёжности (отсутствие ошибок и целостность данных как в примере производителя и потребителя) и жизнеспособности (способности всех взаимодействующих потоков завершить работу). Ядро FX-RTOS является вытесняемым, то есть допускающим прерывания без нарушения целостности управляющих структур данных и алгоритмов. Естественно, само ядро не имеет возможности использовать примитивы синхронизации, предоставляемые в пользовательском API, а использует более низкоуровневые механизмы. В однопроцессорной системе источником асинхронности являются прерывания и смена контекста выполняющихся потоков. Например, системный вызов, прерванный в момент модификации очереди к объекту, приведёт к неопределённому поведению при обращении к тому же объекту из другого потока.

Запрещение прерываний в однопроцессорных системах обеспечивает взаимное исключение как с потоками, так и с обработчиками прерываний. Этот простой способ, выполняемый

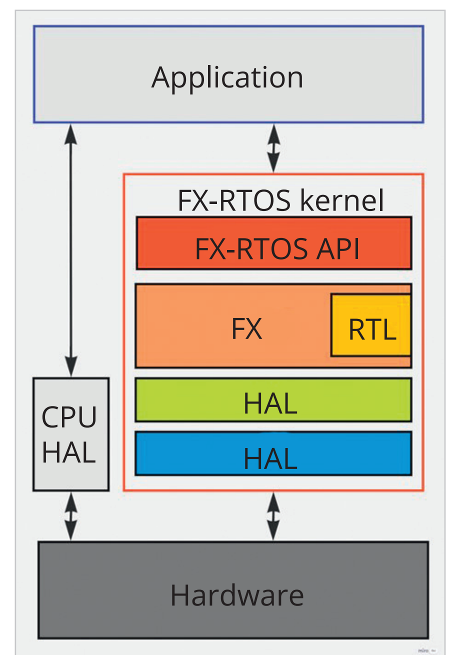

Рис. 4. Иерархия компонентов FX-RTOS во встроенном приложении

за 1 инструкцию, имеет высокую гранулярность, так как не позволяет выполняться в том числе и тем потокам, которые не используют защищаемый ресурс. В FX-RTOS используется концепция *приоритета системы по отношению к асинхронным событиям (SPL)*. Различные сервисы ОС должны выполняться на минимальном уровне SPL, достаточном, чтобы в это время не могли выполняться другие потоки, пользующиеся теми же управляющими структурами. Запрещение всех прерываний осуществляется на самом высоком уровне, в терминах HAL называющемся SPL_SYNC [6]. В системах реального времени продол-

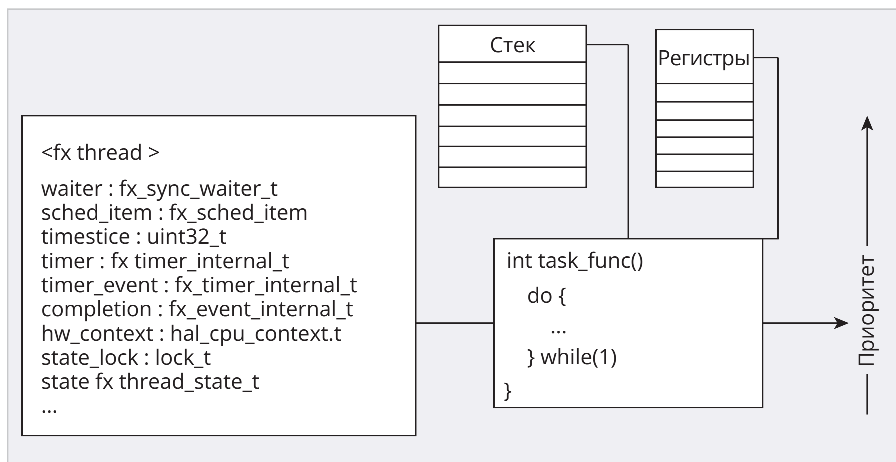

Рис. 5. Параметры потока

жительность выполнения секции кода при запрещённых прерываниях должна быть строго ограничена константой, чтобы разработчик мог рассчитать заранее, сможет ли задача среагировать на событие с заданной максимальной задержкой [7]. Уровень SPL_SYNC используется при изменении данных центральных сервисов ядра, таких как планировщик. Исполнение пользовательских нитей происходит на уровне SPL_LOW, позволяющем прерываться любыми источниками. За консистентность данных пользовательских приложений отвечают примитивы синхронизации, предоставляемые ядром. В некоторых конфигурациях FX-RTOS возможно использование дополнительных промежуточных уровней SPL, что усложняет реализацию, но позволяет гибче и быстрее реагировать на события.

Помимо управления прерываниями, при наличии аппаратной поддержки используются более сложные механизмы, такие как неблокирующая синхронизация с атомарными инструкциями и спин-блокировки в симметричных многопроцессорных системах (SMP).

Реализация потоков

Входом в ядро является функция *fx_kernel_entry*, вызывающая *fx_thread_init*, где инициализируются контейнеры системных объектов: список планируемых потоков, список таймеров. Создаётся и добавляется в очередь планировщика поток *idle* с уникальным, наименьшим приоритетом из возможных. Поток *idle* необходим для корректного функционирования ОС, чтобы замещать пользовательские потоки при отсутствии активных задач. Перед входом в нормальный режим работы устанавливается пользовательский уровень *SPL_LOW*. Это означает, что обработка любых прерываний разрешена, и потоки будут вытесняться другими потоками и прерываниями. На заключительном шаге инициализации вызывается функция *fx_app_init()*, которую определяет программист для создания начальных пользовательских объектов и задач.

Каждая нить имеет персональные секции под код и стек. Указатели на функцию входа и область, выделенную под стек, передаются в конструктор потока *fx_thread_init*, который создаёт контекст в памяти, инициализирует структуру fx_thread_t, хранящую состояние нити. Здесь хранятся параметры планирования (приоритет, квант), ссылка для подключения в очередь ожидания, структуры для манипуляций программным таймером, контекст (рис. 5).

Глобальная переменная *g_current_thread* хранит указатель на структуру *fx_thread_t* текущего активного потока.

Архитектура планировщика

Планировщик обладает внутренним состоянием *fx_sched_context_t*, адресуемым глобальной переменной global_domain. Экземпляр хранит ссылку на планировочную структуру активного потока, ссылку на очереди готовых потоков (*g_domain*) и другие поля (рис. 6).

Атрибут *g_domain* типа *fx_sched_container_t* содержит информацию обо всех потоках, готовых к выполнению, и их приоритетах. Массив из очередей потоков использует структуру *rtl_queue*, реализующую абстрактный связный список, способный содержать любые объекты [8]. Двухуровневая битовая карта приоритетов хранит все номера приоритетов (0…1023 в 32-битной системе) готовых потоков и позволяет за кратчайшее время определить наиболее приоритетную очередь (рис. 7).

При добавлении нового потока его приоритет отмечается единичными битами в Map1 и Map2 с индексами:

$$Idx_{map1}={}^{\text{Priority}}/_{\text{Bitwidth}_{\text{int}}}$$

Процедура планирования (вызов *fx_sched_container_get*) сводится к определению приоритетнейшей непустой очереди и извлечении первого элемента из неё. Необходимо заметить, что фактический приоритет обратно пропорционален его номеру. Наивысший приоритет определяется по самым

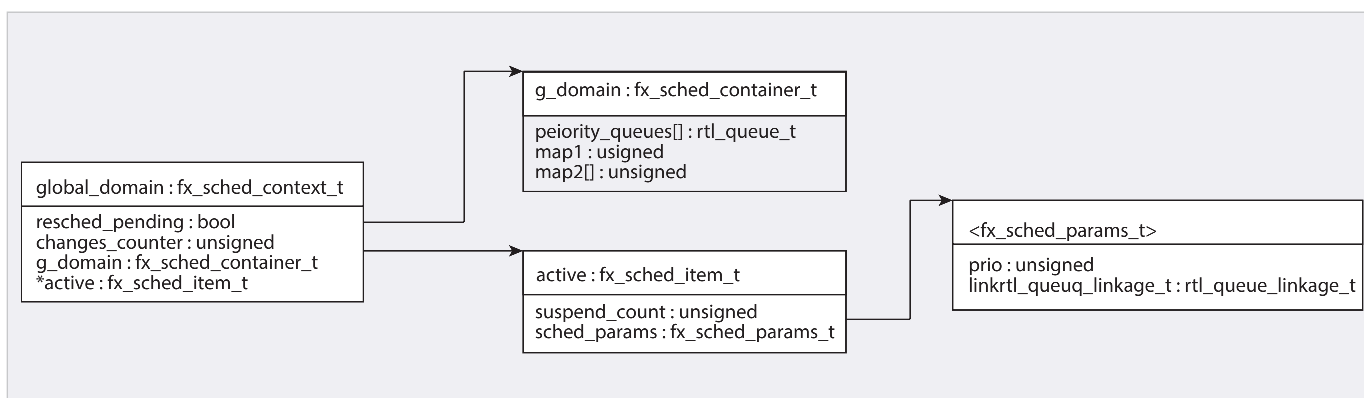

Рис. 6. Диаграмма классов управляющих структур планировщика

младшим установленным битам в Map1 и Map2:

$$Idx_{SchedQueue\,max} = Idx_{map1\,max} \times \mathrm{BitWidth}_{int} + Idx_{map2\,max}$$

Для определения наименьшего установленного бита используется процессорная инструкция CTZ (*Count Trailing Zeroes*), реализованная в слое HW, считающая число нулевых битов в регистре после последнего единичного. Таким образом, планирование выполняется за константное время, не зависящее от числа всевозможных приоритетов и потоков, что является критической характеристикой для систем жёсткого реального времени.

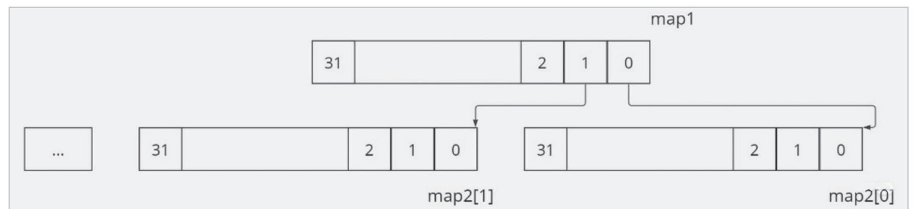

Рис. 7. Структура двухуровневой карты приоритетов

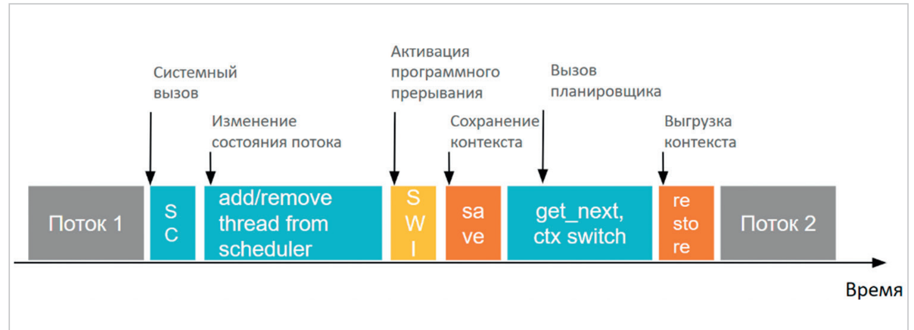

Рис. 8. Последовательность системного вызова, диспетчеризации и смены контекста

Диспетчеризация

Планировщик вызывается каждый раз при изменении состояния объектов, влияющих на выполнение потоков. Необходимость планирования может возникнуть в результате системного вызова или внешнего события. Пример: чтобы заблокировать поток в ожидании ресурса, системный вызов обращается к функции *fx_sched_item_suspend()*, которая удаляет поток из очереди планировщика, а затем устанавливает флаг *resched_pending* и делает запрос программного прерывания. Обработчик программного прерывания отработает не сразу, а когда приоритет системы будет понижен с SPL_SYNC до SPL_LOW (рис. 8).

При входе в прерывание контекст потока сохраняется в стек, обработчик прерывания выполняется на отдельном стеке. Затем вызывается функция *fx_dispatch_handler()*, где проверяется флаг *resched_pending* и вызывается планировщик. Глобальная переменная *g_current_thread* перезаписывается указателем на TCB нового потока. Указатель на стековый фрейм с сохранёнными регистрами подменяется на сохранённый стек другого потока. При выходе из прерывания в регистры будет загружен уже контекст нового потока (или прежнего, если он остался самым приоритетным).

Типичное аппаратное прерывание – от системного таймера, отсчитывающего фиксированные периоды времени, «тики». По завершении каждого такого периода индексируются значения счётчиков, отмеряющих кванты работы потоков и определяющих моменты срабатывания программных таймеров. По окончании обработки вызывается диспетчеризация, как в предыдущем случае.

Во время работы планировщика могут возникать другие прерывания, меняющие состояние потоков. Чтобы их учитывать, в вызовах функций планировщика присутствуют точки синхронизации, где прерывания разрешены и планировщик может быть вытеснен. Таким образом, возможно неоднократное вхождение в функцию планировщика, в результате корректной работы которой управление будет передано наиболее приоритетному потоку, первому в очереди.

Заключение

Ядро операционной системы – это программное обеспечение, которое управляет временем процессора, реализуя абстракцию параллельного выполнения потоков, управляющих объектами окружающей среды. Одни задачи реального мира имеют периодический, другие – апериодический характер. При работе с гибридными наборами задач основная цель ядра – гарантировать планирование всех потоков реального времени в наихудших условиях и обеспечить достаточно хорошее среднее время отклика для действий, выполняемых не в жёстких ограничениях. Гарантия своевременной обработки апериодических задач, вызываемых событиями, может быть достигнута только путём принятия правильных предположений о минимальном промежутке времени между двумя последовательными экземплярами событий. В ОСРВ FX-RTOS применены все современные архитектурные и технические решения для решения задач жёсткого реального времени. Исследуемый в статье исходный код версии Lite предоставлен в свободном доступе [1]. Полный набор конфигураций ядра ОСРВ FX-RTOS доступен на сайте компании-разработчика.

Литература

1. Исходные тексты и примеры ОСРВ FX-RTOS Lite // URL: https://github.com/Eremex/fxrtos-lite.
2. *Столлингс В.* Операционные системы. Внутренняя структура и принципы проектирования. 9-е издание. М.: Диалектика-Вильямс, 2020.
3. *Дейкстра Э.* Языки программирования / пер. с англ. под ред. В.М. Курочкина. М.: Мир, 1972.
4. *Downey A.* Little book of semaphores // URL: https://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf.
5. *Buttazzo G.C.* Hard Real-Time Computing Systems. Predictable Scheduling Algorithms and Applications. Third Edition. Springer, 2011.
6. FX-RTOS. Руководство по эксплуатации // URL: https://www.eremex.ru/products/fx-rtos/#database.
7. Разработка встроенных операционных систем реального времени // URL: https://www.eremex.ru/upload/iblock/eb6/rtos_dev_book.pdf.
8. Объяснение работы абстрактного списка list_entry // URL: https://stackoverflow.com/questions/5550404/list-entry-in-linux.