Содержание

1 Введение

Когда компьютер работает в многозадачном режиме (большинство современных вычислительных систем и многие встроенные системы используют многозадачность), на нем могут быть активными несколько процессов, пытающихся одновременно получить доступ к процессору. Эта ситуация возникает при наличии двух и более процессов в состоянии готовности. Если доступен только один процессор, необходимо выбирать между процессами. Отвечающая за это часть операционной системы называется *планировщиком*, а используемый алгоритм — *алгоритмом планирования*.

1.1 Общие сведения о планировщиках

1.1.1 Задачи планирования

Во времена систем пакетной обработки, использовавших отображение содержимого перфокарт на магнитной ленте в качестве устройства ввода, алгоритм планирования был прост: запустить следующую задачу на ленте. С появлением систем с разделением времени (time-sharing) алгоритм планирования усложнился, поскольку теперь несколько задач одновременно ожидали обслуживания. На некоторых мэйнфреймах до сих пор совмещаются системы пакетной обработки и службы разделения времени. В результате планировщик должен решать, запустить ли следующим пакетное задание или предоставить процессор интерактивному пользователю за терминалом. Поскольку время процессора является в такой системе дефицитным ресурсом, хороший планировщик может существенно изменить ощущаемую пользователем производительность работы и, следовательно, степень удовлетворения пользователя. Поэтому много усилий было потрачено на разработку умных и эффективных алгоритмов планирования.

Помимо правильного выбора следующего процесса, планировщик также должен заботиться об эффективном использовании процессора, поскольку переключение между процессами требует затрат. Во-первых, необходимо переключиться из режима пользователя в режим ядра. Затем следует сохранить текущее состояние текущего процесса, включая сохранение регистров в таблице процессов, чтобы их можно было загрузить заново позже. Затем нужно выбрать следующий процесс, запустив алгоритм планирования. После этого нужно запустить новый процесс.

Все это занимает много процессорного времени, особенно при слишком частом переключении между процессами, поэтому в этом рекомендуется соблюдать умеренность.

1.1.2 Обоснование целесообразности применения

Типичная встроенная вычислительная система должна решать множество задач одновременно, например:

- следить за показаниями датчиков
- выполнять требуемые алгоритмом вычисления
- оказывать управляющее воздействие на объект управления
- обеспечивать интерактивное взаимодействие с оператором

Рассмотрим, как можно реализовать необходимую функциональность без применения многозадачности (а следовательно, и планировщика).

При этом применяется традиционный метод бесконечного цикла, где каждый компонент приложения соответствует функции, которая выполняется до завершения.

В идеальном случае можно было бы использовать аппаратный таймер для решения задач управления критически важными процессами, на практике же необходимость ожидания данных и выполнения сложных вычислений делает эту задачу неподходящей для выполнения в процедуре обработки прерываний.

Для приоритезации задач можно изменять порядок и частоту исполнения отдельных функций в главном бесконечном цикле.

У этого подхода есть такие достоинства как:

- малый размер кода
- независимость от программ третьих лиц (при использовании готового планировщика)
- снижение требований к количеству ОЗУ, ПЗУ и вычислительным ресурсам

Однако среди недостатков у него:

- проблемы обеспечения сложных требований по времени

- расширение функциональности приводит к значительному увеличению сложности
- сложность оценки временных интервалов работы системы в связи с взаимозависимостями между функциями

Таким образом, получается, что использование бесконечного цикла хорошо только для простых приложений и приложений с гибкими требованиями по времени. Для более сложных применений он становится избыточно сложным и ненадежным.

1.2 Категории алгоритмов планирования

В различных средах требуются различные алгоритмы планирования. Это связано с тем, что различные операционные системы и различные приложения ориентированы на разные задачи. Другими словами, то, для чего следует оптимизировать планировщик, различно в разных системах. Можно выделить три среды:

1. Системы пакетной обработки данных
2. Интерактивные системы
3. Системы реального времени

В системах пакетной обработки нет пользователей, сидящих за терминалами и ожидающих ответа. В таких системах приемлемы алгоритмы без переключений или с переключениями, но с большим временем, отводимым каждому процессу. Такой метод уменьшает количество переключений между процессами и улучшает эффективность.

В интерактивных системах необходимы алгоритмы планирования с переключениями, чтобы предотвратить захват процессора одним процессом. Даже если ни один процесс не захватывает процессор на неопределенно долгий срок намеренно, из-за ошибки в программе один процесс может заблокировать остальные. Для исключения подобных ситуаций используется планирование с переключениями (т.е. не кооперативная многозадачность, а принудительная).

В системах с ограничениями реального времени приоритетность, как это ни странно, не всегда обязательна, поскольку процессы знают, что их время ограничено, и быстро выполняют работу, а затем блокируются. Отличие от интерактивных систем в том, что в системах реального

времени работают только программы, предназначенные для содействия конкретным приложениям. Интерактивные системы являются универсальными системами. В них могут работать произвольные программы, не сотрудничающие друг с другом и даже враждебные по отношению друг к другу.

При всех обстоятельствах необходимо справедливое распределение процессорного времени. Сопоставимые процессы должны получать сопоставимое обслуживание. Выделить одному процессу намного больше времени процессора, чем другому, эквивалентному, несправедливо. Разумеется, с различными категориями процессов можно обращаться весьма по-разному. Возьмем, например, задачи обеспечения безопасности и начисления заработной платы в компьютерном центре атомной электростанции.

Еще одной глобальной задачей является контроль занятости всех частей системы. Если устройства ввода-вывода и процессор все время работают, в единицу времени окажется выполнено гораздо больше полезной деятельности, чем если отдельные компоненты будут простаивать. Например, в системах пакетной обработки планировщик выбирает, какие задачи загрузить в память для работы. Гораздо лучше иметь в памяти одновременно несколько процессов, ограниченных возможностями устройств ввода-вывода, чем сначала загрузить и запустить несколько процессов, ограниченных возможностями процессора, и только потом несколько процессов, ограниченных возможностями устройств ввода-вывода. В последнем случае во время работы процессов, ограниченных возможностями процессора, будет простаивать диск, а во время работы процессов, ограниченных возможностями устройств ввода-вывода, будет простаивать процессор. Правильнее будет заставить работать всю систему, аккуратно перемешав процессы.

1.2.1 Планирование в системах реального времени

Системы реального времени обладают не такими свойствами, как интерактивные системы, соответственно и задачи планирования будут разными. Характерной особенностью систем реального времени является наличие жестких сроков выполнения определенных действий. Например, если компьютер управляет устройством, выдающим данные с постоянной скоростью, неспособность своевременно запустить процесс, обрабатывающий данные, приведет к потере данных. Поэтому первоочередной

задачей в системах реального времени является строгое соблюдение всех (или большинства) требований по срокам.

Чаще всего одно или несколько внешних физических устройств генерируют входные сигналы, и компьютер должен адекватно на них реагировать в течение заданного промежутка времени. Например, компьютер в проигрывателе компакт-дисков получает биты от дисковода и должен за очень маленький промежуток времени конвертировать их в музыку. Если процесс конвертации будет слишком долгим, звук окажется искаженным. Подобные системы также используются для наблюдениями за пациентами в палатах интенсивной терапии, в качестве автопилота самолета, для управления роботами на автоматизированном производстве. В любом из этих случаев запоздалая реакция ничуть не лучше, чем отсутствие реакции.

Системы реального времени делятся на *жесткие системы реального времени*, что означает наличие жестких сроков для каждой задачи (в них обязательно надо укладываться), и *гибкие системы реального времени*, в которых нарушения временного графика нежелательны, но допустимы. В обоих случаях реализуется разделение программы на несколько процессов, каждый из которых предсказуем. Эти процессы чаще всего бывают короткими и завершают свою работу в течение секунды. Когда появляется внешний сигнал, именно планировщик должен обеспечить соблюдение графика.

Внешние события, на которые система должна реагировать, можно разделить на *периодические* (возникающие через регулярные интервалы времени) и *непериодические* (возникающие непредсказуемо). Возможно наличие нескольких периодических потоков событий, которые система должна обрабатывать. В зависимости от времени, затрачиваемого на обработку каждого из событий, может оказаться, что система не в состоянии своевременно обработать все события. Если в систему поступает m периодических событий, событие с номером i поступает с периодом P_i, и на его обработку уходит C_i секунд работы процессора, все потоки могут быть своевременно обработаны только при выполнении условия

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leqslant 1.$$

Системы реального времени, удовлетворяющие этому условию, называются поддающимися планированию или *планируемыми*.

Алгоритмы планирования для систем реального времени могут быть как статическими, так и динамическими. В первом случае все решения планирования принимаются заранее, еще до запуска системы. Во втором случае решения планирования принимаются по ходу дела. Статическое планирование применимо только при наличии достоверной информации о работе, которую необходимо выполнить, и о временном графике, которого нужно придерживаться. Динамическое планирование не нуждается в подобных ограничениях.

1.2.2 Алгоритмы планирования для интерактивных систем

Циклическое планирование Одним из наиболее старых, простых, справедливых и часто используемых является алгоритм *циклического планирования*. Каждому процессу предоставляется некоторый интервал времени процессора, так называемый *квант* времени. Если к концу кванта времени процесс все еще работает, он прерывается, а управление передается другому процессу. Разумеется, если процесс блокируется или прекращает работу раньше, переход управления происходит в этот момент. Реализация циклического планирования проста. Планировщику нужно всего лишь поддерживать список процессов в состоянии готовности. Когда процесс исчерпал свой лимит времени, он отправляется в конец списка.

Единственным интересным моментом этого алгоритма является длина кванта. Переключение с одного процесса на другой занимает некоторое время — необходимо сохранить и перезагрузить кэш памяти и т.п. Представим, что *переключение процессов* или *переключение контекста*, как его иногда называют, занимает 1 мс, включая переключение карт памяти, перезагрузку кэша и т.п. Пусть размер кванта установлен в 4 мс. В таком случае 20% процессорного времени уйдет на администрирование — это слишком много.

Для увеличения эффективности назначим размер кванта, скажем, 100 мс. Теперь пропадает только 1% времени. Но представьте, что будет в системе с разделением времени, если 10 пользователей одновременно нажмут клавишу возврата каретки. В список будет занесено 10 процессов. Если процессор был свободен, первый процесс будет запущен немедленно, второму придется ждать 100 мс, и т.д. Последнему процессу, возможно, придется ждать целую секунду, если все остальные не блокируются за время кванта. Большинству пользователей секундная задержка вряд

ли понравится.

Важен и тот фактор, что если установленное значение кванта больше среднего интервала работы процессора, переключение процессоров будет происходить редко. Напротив, большинство процессов будут совершать блокирующую операцию прежде, чем истечет длительность кванта, вызывая переключение процессов. Устранение принудительных переключений процессов улучшает производительность системы, так как переключения процессов будут происходить только тогда, когда это логически необходимо, то есть когда процесс заблокировался и не может продолжить работу.

Вывод можно сформулировать следующим образом: слишком малый квант приведет к частому переключению процессов и небольшой эффективности (за счет *накладных расходов*), но слишком большой квант может привести к медленному реагированию на короткие интерактивные запросы. Значение кванта около 20–50 мс (при *стоимости* переключения 4 мс) часто является разумным компромиссом.

Приоритетное планирование В циклическом алгоритме планирования есть важное допущение о том, что все процессы равнозначны. Зачастую это не так, например, процесс управления двигателем должен иметь больший приоритет, чем интерактивное взаимодействие с пользователем, которое, в свою очередь, приоритетнее записи в журнал.

Чтобы предотвратить бесконечную работу процессов с высоким приоритетом, планировщик может уменьшать приоритет процесса с каждым тактом часов (то есть при каждом прерывании по таймеру). Если в результате приоритет текущего процесса окажется ниже, чем приоритет следующего процесса, произойдет переключение. Возможно предоставление каждому процессу максимального отрезка времени работы. Как только время кончилось, управление передается следующему по приоритету процессу.

Приоритеты процессам могут присваиваться статически или динамически. В системе Unix есть команда *nice*, позволяющая пользователю добровольно снизить приоритет своих процессов, чтобы быть вежливыми по отношению к остальным пользователям. Этой командой никто никогда не пользуется.

Система может динамически назначать приоритеты для достижения своих целей. Например, некоторые процессы сильно ограничены возможностями устройств ввода-вывода и большую часть времени проводят в

ожидании завершения операций ввода-вывода. Когда бы ни потребовался процессор такому процессу, его следует немедленно предоставить, чтобы процесс мог начать следующий запрос ввода-вывода, который будет выполняться параллельно с вычислениями другого процесса. Если заставить процесс, ограниченный возможностями устройств ввода-вывода, длительное время ждать доступа к процессору, он будет неоправданно долго находиться в памяти. Простой алгоритм обслуживания процессов, ограниченных возможностями устройств ввода-вывода, состоит в установке приоритета, равного $1/f$, где f — часть использованного в последний раз кванта. Процесс, использовавший всего 1 мс из 50 мс кванта, получит приоритет 50, процесс, использовавший 25 мс, получит приоритет 2, а процесс, использовавший весь квант, получит приоритет 1.

Часто бывает удобно сгруппировать процессы в классы по приоритетам и использовать приоритетное планирование среди классов, но циклическое планирование внутри каждого класса.

Несколько очередей Один из первых приоритетных планировщиков был реализован в системе CTSS (compatible time-shared system). Основной проблемой системы CTSS было слишком медленное переключение процессов, поскольку в памяти компьютера IBM 7094 мог находиться только один процесс. Каждое переключение означало выгрузку текущего процесса на диск и считывание нового процесса с диска. Разработчики CTSS быстро сообразили, что эффективность будет выше, если процессам, ограниченным возможностями процессора, выделять больший квант времени, чем если предоставлять им небольшие кванты, но часто. С одной стороны, это уменьшит количество перекачек из памяти на диск, с другой — приведет к ухудшению времени отклика. В результате было разработано решение с классами приоритетов. Процессам класса с высшим приоритетом выделялся один квант, процессам следующего класса — два кванта, следующего — четыре кванта и т.д. Когда процесс использовал все отведенное ему время, он перемещался на класс ниже.

Чтобы процессу, который при запуске считался долгим, но позже стал интерактивным, не погибнуть в недрах планирования, была разработана следующая стратегия. Как только с терминала приходит сигнал возврата каретки, процесс, соответствующий этому терминалу, переносится в класс высшего приоритета, поскольку предполагается, что он становится интерактивным.

«Самый короткий процесс — следующий» Поскольку алгоритм «Кратчайшая задача — первая» минимизирует среднее оборотное время в системах пакетной обработки, хотелось бы использовать его и в интерактивных системах. В известной степени это возможно. Интерактивные процессы чаще всего следуют схеме «ожидание команды, исполнение команды, ожидание команды, исполнение команды...». Если рассматривать выполнение каждой команды как отдельную задачу, можно минимизировать общее среднее время отклика, запуская первой самую короткую задачу. Единственная проблема состоит в том, чтобы понять, какой из ожидающих процессов самый короткий.

Один из методов оценивается на оценке длины процесса, базирующейся на предыдущем проведении процесса. При этом запускается процесс, у которого оцененное время самое маленькое. Допустим, что предполагаемое время исполнения команды равно T_0 и предполагаемое время следующего запуска равно T_1. Можно улучшить оценку времени, взяв взвешенную сумму этих времен: $aT_0 + (1 - a)T_1$. Выбирая соответствующее значение a, мы можем заставить алгоритм оценки быстро забывать о предыдущих запусках или, наоборот, помнить о них в течении долгого времени. Взяв $a = 1/2$, мы получим серию оценок:

$$T_0, T_0/2 + T_1, T_0/4 + T_1/4 + T_2/2, T_0/8 + T_1/8 + T_2/4 + T_3/2.$$

После трех запусков вес T_0 в оценке уменьшится до $1/8$.

Метод оценки следующего значения серии через взвешенное среднее предыдущего значения и предыдущей оценки часто называют *старением*. Этот метод применим во многих ситуациях, где необходима оценка по предыдущим значениям. Проще реализовать старение при $a = 1/2$. На каждом шаге нужно лишь добавить к текущей оценке новое значение и разделить сумму пополам (сдвинув вправо на 1 бит).

Гарантированное планирование Принципиально другим подходом к планированию является предоставление пользователям реальных обещаний и затем их выполнение. Вот одно обещание, которое легко произнести и легко выполнить: если вместе с вами процессором пользуются n пользователей, вам будет предоставлено $1/n$ мощности процессора. И в системе с одним пользователем и n запущенными процессами каждому достанется $1/n$ циклов процессора.

Чтобы выполнить это обещание, система должна отслеживать распределение процессора между процессами с момента создания каждого

процесса. Затем система рассчитывает количество ресурсов процессора, на которое процесс имеет право, например время с момента создания, деленное на n. Теперь можно сосчитать отношение времени, предоставленное процессу, к времени, на которое он имеет право. Затем запускается процесс, у которого это отношение наименьшее, пока оно не станет больше, чем у его ближайшего соседа.

Лотерейное планирование Хотя идея обещаний пользователям и их выполнения хороша, но ее трудно реализовать. Для более простой реализации предсказуемых результатов используется другой алгоритм, называемый *лотерейным планированием*.

В основе алгоритма лежит раздача процессам лотерейных билетов на доступ к различным ресурсам, в том числе и к процессору. Когда планировщику необходимо принять решение, выбирается случайным образом лотерейный билет, и его обладатель получает доступ к ресурсу. Что касается доступа к процессору, «лотерея» может происходить 50 раз в секунду и победитель получает 20 мс времени процессора.

Более важным процессам можно раздать дополнительные билеты, чтобы увеличить вероятность выигрыша. Если всего 100 билетов и 20 из них находятся у одного процесса, то ему достанется 20% времени процессора. В отличие от приоритетного планировщика, в котором очень трудно оценить, что означает, скажем, приоритет 40, в лотерейном планировании все очевидно. Каждый процесс получит процент ресурсов, примерно равный проценту имеющихся у него билетов.

Лотерейное планирование характеризуется несколькими интересными свойствами. Например, если при создании процессу достается несколько билетов, то уже в следующей лотерее его шансы на выигрыш пропорциональны количеству билетов. Другими словами, лотерейное планирование обладает высокой отзывчивостью.

Взаимодействующие процессы могут при необходимости обмениваться билетами. Так, если клиентский процесс посылает сообщение серверному процессу и затем блокируется, он может передать все свои билеты серверному процессу, чтобы увеличить шанс запуска сервера. Когда серверный процесс заканчивает работу, он может вернуть все билеты обратно. Действительно, если клиентов нет, то серверу билеты вовсе не нужны.

1.2.3 Алгоритмы планирования для систем реального времени

В некоторых системах реального времени процессы являются прерываемыми, тогда как в других системах — нет. Если процессы прерываемы, это означает, что процесс, которому угрожает невыполнение задачи в срок, может прервать работающий процесс прежде, чем тот успеет закончить свою работу. Затем управление может быть возвращено прерванному процессу.

Алгоритмы реального времени могут быть статическими или динамическими. Статические алгоритмы заранее назначают каждому процессу фиксированный приоритет, после чего выполняют приоритетное планирование с переключениями. У динамических алгоритмов нет фиксированных приоритетов.

Алгоритм планирования RMS Классическим примером статического алгоритма планирования реального времени для прерываемых, периодических процессов является алгоритм RMS (Rate Monotonic Scheduling — планирование с приоритетом, пропорциональным частоте). Этот алгоритм может использоваться для процессов, удовлетворяющих следующим условиям:

1. каждый периодический процесс должен быть завершен за время его периода;
2. ни один процесс не должен зависеть от любого другого процесса;
3. каждому процессу требуется одинаковое процессорное время на каждом интервале;
4. у непериодических процессов нет жестких сроков;
5. прерывание процесса происходит мгновенно, без накладных расходов.

Первые четыре требования разумны. Последнее, естественно, нет, но оно значительно упрощает модель системы. Алгоритм RMS работает, назначая каждому процессу фиксированный приоритет, равный частоте возникновения событий процесса. Таким образом, приоритеты пропорциональны частоте. Отсюда и название алгоритма. Во время работы планировщик всегда запускает готовый к работе процесс с наивысшим

приоритетом, прерывая при необходимости работающий процесс. Авторы алгоритма доказали, что RMS является оптимальным решением в классе статических алгоритмов планирования.

Алгоритм планирования EDF Другим популярным алгоритмом планирования является алгоритм EDF (Earliest Deadline First — процесс с ближайшим сроком завершения в первую очередь). Алгоритм EDF представляет собой динамический алгоритм, в отличие от предыдущего алгоритма не требующий от процессов периодичности. Он также не требует и постоянства временных интервалов использования центрального процессора. Каждый раз, когда процессу требуется процессорное время, он объявляет о своем присутствии и о своем сроке выполнения задания. Планировщик хранит список процессов, сортированный по срокам выполнения заданий. Алгоритм запускает первый процесс в списке, то есть тот, у которого самый близкий по времени срок выполнения. Когда новый процесс переходит в состояние готовности, система сравнивает его срок выполнения со сроком выполнения со сроком текущего процесса. Если у нового процесса график более жесткий, он прерывает работу текущего процесса.

Одно из преимуществ использования EDF по сравнению с RMS обусловлено тем, что использование статических приоритетов работает только при не слишком высокой загруженности центрального процессора. Было показано, что алгоритм RMS гарантированно работает в любой системе периодических процессов при условии

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leqslant m \left(2^{1/m} - 1\right).$$

Для 3, 4, 5, 10, 20 и 100 процессов максимальная разрешенная загруженность процессора составляет 0.780, 0.757, 0.743, 0.718, 0.705 и 0.696. При $m \to \infty$ значение максимальной загруженности процессора асимптотически стремится к $\ln 2$.

Алгоритм EDF, напротив, всегда работает с любым набором процессов, для которого возможно планирование. Коэффициент загруженности центрального процессора для алгоритма EDF может достигать 100%. Платой за это является использование более сложного алгоритма.

2 Описание алгоритмов планирования в распространенных планировщиках

2.1 Планировщик из состава FreeRTOS

FreeRTOS — легко портируемая свободная мини-ОС реального времени. Весь системо-независимый код из состава FreeRTOS написан на C, что позволило разработчикам создать большое число портов на различные архитектуры и средства разработки. Ядро FreeRTOS поддерживает вытесняющую, добровольную и гибридные модели многозадачности. Поддерживаются как обычные процессы, так и корутины. Имеются возможности по отладке, контролю переполнения стека. Отсутствуют ограничения на количество процессов и приоритетов. Для синхронизации можно применять очереди сообщений, бинарные и семафоры с счетчиком, а также мьютексы (с наследованием приоритетов).

Для хранения информации о задачах FreeRTOS использует несколько очередей, в частности очередь для задач, находящихся в состоянии ожидания, очередь для задач, ожидающих перевода в состояние готовых к исполнению, и массива очередей с задачами, готовыми к исполнению. Число элементов этого массива равно числу приоритетов, используемых программистом, и задается в конфигурационном файле проекта. Выбор типа планирования также задается в этом конфигурационном файле; эти параметры не могут быть изменены во время исполнения.

2.1.1 Добровольное планирование

В каждом процессе программист должен предусмотреть точки вызова планировщика, причем время между этими вызовами не должно быть слишком большим, так как в противном случае переключение контекстов будет происходить слишком редко, и некоторые процессы могут не успеть выполнить в срок возложенные на них задачи. Активация планировщика осуществляется вызовом функции `vPortYield`, которая для порта на ATMega (AVR-микроконтроллер фирмы Atmel) имеет следующий вид:

```
/*
 * Manual context switch.  The first thing we do is save
 * the registers so we
 * can use a naked attribute.
 */
```

```
void vPortYield( void ) __attribute__ ( ( naked ) );
void vPortYield( void )
{
        portSAVE_CONTEXT();
        vTaskSwitchContext();
        portRESTORE_CONTEXT();

        asm volatile ( "ret" );
}
```

В ней `portSAVE_CONTEXT` и `portRESTORE_CONTEXT` являются платформозависимыми макросами для сохранения контекста процессов (текущих значений регистров и др.), а `vTaskSwitchContext` — платформонезависимая функция, осуществляющая планирование:

```
void vTaskSwitchContext( void )
{
...
        /* Find the highest priority queue that contains ready
        tasks. */
        while( listLIST_IS_EMPTY(
               &( pxReadyTasksLists[ uxTopReadyPriority ] ) ) )
        {
                --uxTopReadyPriority;
        }

        /* listGET_OWNER_OF_NEXT_ENTRY walks through the list,
        so the tasks of the
        same priority get an equal share of the processor time. */
        listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB,
                &( pxReadyTasksLists[ uxTopReadyPriority ] ) );
...
}
```

Данная функция реализует довольно простой алгоритм: начиная с самого большого приоритета (`uxTopReadyPriority` изменяется при добавлении процесса в очередь готовых к исполнению, поэтому почти всегда содержит максимальное значение приоритета для процессов, готовых к исполнению) ищется непустая очередь с процессом в состоянии готов-

ности. Макрос `listGET_OWNER_OF_NEXT_ENTRY` осуществляет последовательную выборку элементов из очереди.

Из данного описания видно, что если в каждый момент времени хотя бы один высокоприоритетный процесс находится в состоянии готовности, то низкоприоритетные процессы никогда не получат управления (за исключением случая, когда приоритет владеющего мьютексом низкоприоритетного процесса временно повышается).

2.1.2 Preemption

Для реализации вытесняющей многозадачности используется следующий механизм: один из системных таймеров настраивается на генерацию прерывания заданное число раз в секунду, обработчик прерывания для ATMega выглядит следующим образом:

```
#if configUSE_PREEMPTION == 1

        /*
         * Tick ISR for preemptive scheduler.  We can use a naked
         * attribute as
         * the context is saved at the start of vPortYieldFromTick().
         * The tick
         * count is incremented after the context is saved.
         */
        void SIG_OUTPUT_COMPARE1A( void ) __attribute__ ( ( signal,
                naked ) );
        void SIG_OUTPUT_COMPARE1A( void )
        {
                vPortYieldFromTick();
                asm volatile ( "reti" );
        }
#else

        /*
         * Tick ISR for the cooperative scheduler.  All this does is
         * increment the
         * tick count.  We don't need to switch context, this can only
         * be done by
         * manual calls to taskYIELD();
```

```
         */
        void SIG_OUTPUT_COMPARE1A( void ) __attribute__ ( ( signal ) );
        void SIG_OUTPUT_COMPARE1A( void )
        {
                vTaskIncrementTick();
        }
#endif
...
/*
 * Context switch function used by the tick.  This must be identical to
 * vPortYield() from the call to vTaskSwitchContext() onwards.  The only
 * difference from vPortYield() is the tick count is incremented as the
 * call comes from the tick ISR.
 */
void vPortYieldFromTick( void ) __attribute__ ( ( naked ) );
void vPortYieldFromTick( void )
{
        portSAVE_CONTEXT();
        vTaskIncrementTick();
        vTaskSwitchContext();
        portRESTORE_CONTEXT();

        asm volatile ( "ret" );
}
```

Таким образом, по каждому прерыванию в случае использования *preemption* вызывается функция `vPortYieldFromTick`, аналогичная `vPortYield`, но включающая в себя увеличение счетчика тактов (который, в свою очередь, выполняет проверку наступления событий и может перевести к переводу процессов из состояния ожидания в состояние готовности к исполнению, переместив их в соответствующую очередь). Сам алгоритм планирования остается неизменным.

2.1.3 Оценка сложности O() алгоритма

Так как переменная `uxTopReadyPriority` в подавляющем большинстве случаев уже содержит приоритет готового к исполнению процесса, а структура очереди предусматривает указатель для последовательного обхода всех элементов, время работы планировщика не будет зависеть

от числа задач и будет фиксированным, то есть $O(1)$.

Можно также отметить, что добавление процесса в очередь готовых к исполнению также производится за фиксированное (причем весьма малое) время:

```
#define prvAddTaskToReadyQueue( pxTCB ) { \
    if( pxTCB->uxPriority > uxTopReadyPriority )
        { \
                uxTopReadyPriority = pxTCB->uxPriority; \
        } \
    vListInsertEnd( ( xList * ) &( pxReadyTasksLists[
      pxTCB->uxPriority ] ), &( pxTCB->xGenericListItem ) ); \
}
```

Здесь функция `vListInsertEnd` не сортирует лист, а добавляет элемент в конец, так, что он исполнится после всех уже имеющихся в списке.

Из рассмотрения исходного кода FreeRTOS можно сделать вывод, что разработчикам удалось создать гибкую, легко портируемую систему, с довольно малыми накладными расходами на планирование, при этом имеющую довольно богатые возможности (в частности, *priority inheritance*, наследование приоритетов, необходимое для корректной работы мьютексов).

2.2 Планировщик AvrX

AvrX — легковесная ОС реального времени для AVR-микроконтроллеров. В ее состав входит планировщик, поддерживающий до 16 уровней приоритета; процессы с одинаковым приоритетом исполняются по очереди, планируясь добровольно; процесс с большим приоритетом вытесняет процесс с меньшим.

Также в состав AvrX входят семафоры, очереди сообщений, очередь событий по таймеру. AvrX поддерживает пошаговую отладку исполняемых процессов; в ее состав входит монитор, использующий системные API для остановки, запуска и пошагового исполнения процессов.

AvrX написана на ассемблере, но ее можно применять и для написания проектов на C. Для полной функциональности требуется всего от 700 до 1000 слов программной памяти. Время обработки системного тика — 234 процессорных циклов.

Планировщик AvrX работает следующим образом: для восстановления контекста используется функция `Epilog`, которая переключает стек на процесс, находящийся первым в очереди готовых к исполнению:

```
; _Epilog
;
; Restore previous context (kernel or user).
...
        ldi     Zl, lo8(AvrXKernelData)
        ldi     Zh, hi8(AvrXKernelData)
        BeginCritical
...
        ldd     Yh, Z+RunQueue+NextH
        ldd     Yl, Z+RunQueue+NextL
        std     Z+Running+NextH, Yh
        std     Z+Running+NextL, Yl   ; Update current running task
        adiw    Yl, 0
        breq    _IdleTask

        ldd     Xh, Y+PidSP+NextH
        ldd     Xl, Y+PidSP+NextL
        out     _SFR_IO_ADDR(SPL), Xl
        out     _SFR_IO_ADDR(SPH), Xh         ; 20 cycles
...
```

Таким образом, алгоритм планирования работает за фиксированное время, не зависящее от количества процессов, т.е. за $O(1)$. Естественно, для работы такого алгоритма очередь `RunQueue` должна быть соответственно отсортирована, что происходит в функции `_QueuePid`:

```
; _QueuePid
;
; Takes a PID and inserts it into the run queue.  The run queue is sorted
; by priority.  Lower numbers go first.  If there are multiple tasks
; of equal
; priority, then the new task is appended to the list of equals (round
; robin)
...
        _FUNCTION(_QueuePid)
```

```
_QueuePid:                              ; Kernel entry point only
        mov     Zl, p1l
        mov     Zh, p1h
        ldi     tmp1, lo8(-1)

        ldd     tmp0, Z+PidState          ; Xh = Priority & Flags
        andi    tmp0, (BV(SuspendBit) | BV(IdleBit)) ; if marked
                                              ; Suspended or idle
        brne    _qpSUSPEND

        push    Yl              ; 9/13/04
        push    Yh              ; 9/13/04

        ldd     tmp2, Z+PidPriority
        ldi     Yl, lo8(AvrXKernelData+RunQueue)
        ldi     Yh, hi8(AvrXKernelData+RunQueue)
        in      tmp0, _SFR_IO_ADDR(SREG)
        cli             ; 9/13/04
_qp00:
        inc     tmp1                    ; Tmp1 = counter of insertion
                                        ; point.
        mov     Zl, Yl                  ; 0 = head of run queue.
        mov     Zh, Yh
        ldd     Yl, Z+PidNext+NextL
        ldd     Yh, Z+PidNext+NextH     ; Z = current, X = Next
        adiw    Yl, 0
        breq    _qp01                   ; End of queue, continue
        ldd     tmp3, Y+PidPriority
        cp      tmp2, tmp3
        brsh    _qp00                   ; Loop until pri > PID to
                                        ; queue
_qp01:
        std     Z+NextH, p1h
        std     Z+NextL, p1l    ; Prev->Next = Object
        mov     Zh, p1h
        mov     Zl, p1l
        std     Z+NextH, Yh     ; Object->Next = Next
```

```
std     Z+NextL, Yl
pop     Yh              ; 9/13/05
pop     Yl              ; 9/13/04
mov     r1l, tmp1
out     _SFR_IO_ADDR(SREG), tmp0
ret                     ; 9/13/04
```

Для экономии памяти для очередей применяется простой однонаправленный список, поэтому при добавлении процесса в очередь требуется двигаться от головы списка до тех пор, пока не будут пройдены все процессы с приоритетом большим или равным добавляемому. Следовательно если n — число процессов в очереди с приоритетом большим или равным добавляемому, то время исполнения такого алгоритма пропроционально n, т.е. сложность этого алгоритма $O(n)$.

3 Методика исследования

Анализ алгоритмов функционирования планирования в FreeRTOS и AvrX показал, что фактически решение о планировании принимается не в момент вызова планировщика (т.к. в обоих случаях он просто запускает первый процесс, стоящий в очереди), а в момент добавления процесса в очередь.

Для того чтобы выявить разницу между работой планировщиков этих систем, было решено использовать варьирующееся количество задач одинакового приоритета, планирующихся добровольно. Это позволило измерить время выполнения одной итерации каждого процесса, включая вызов планировщика, осущесвляющего сохранение контекста, повторную постановку процесса в очередь готовых и восстановление контекста следующего процесса.

3.1 Выбор критериев оценки для данного исследования

В качестве критерия было выбрано время полной итерации одного процесса, включая работу планировщика. Анализ исходного кода предсказывал, что это время будет увеличиваться для AvrX с ростом числа процессов, т.к. время добавления в очередь для этой системы пропорционально числу процессов с приоритетом равным или выше добавляемому.

Для FreeRTOS данная зависимость должна отсутствовать из-за другой схемы организации очереди.

Сравнительный анализ количества используемой Flash и RAM памяти не проводился, т.к. функциональность систем значительно различается и для справедливого сравнения потребовались бы модификации исходного кода самих систем.

3.2 Детали практической реализации эксперимента

Для практических измерений было использовано устройство с микроконтроллером ATmega128, к одному из выводов которого был подключен цифровой осциллограф. Вывод настроен на выход, запускающиеся задачи поочередно переводят его в состояние 0 и 1. Таким образом на осциллограф подавался меандр со скважностью 2, т.к. все задачи были равноценны и планировались одинаково. С учетом разницы в наименованиях функций используемые для теста программы абсолютно идентичны. Исходный код FreeRTOS был модифицирован для отключения ненужной в данном случае инициализации таймера, т.к. вносимая прерыванием задержка отражалась на результатах измерений.

Частота меандра отображалась осциллографом на экране, затем пересчитывалась в время работы одной итерации процесса и заносилось в таблицу.

Полный исходный код тестовых программ дан в Приложениях A и B.

4 Результаты испытаний

4.1 FreeRTOS

Как и ожидалось, измерения показали независимость производительности планировщика FreeRTOS от количества процессов. Было измерено время для 2, 4, 8 и 16 процессов. Во всех случаях оно было равно 30.7 микросекунд.

4.2 AvrX

Результаты AvrX оказались предсказуемо хуже, чем FreeRTOS. Для тех же условий были получены результаты:

Число процессов	Время итерации
2	42.9 μs
4	46.6 μs
8	54.1 μs
16	69.1 μs

Мы видим, что увеличение количества процессов на 1 приводит к увеличению времени работы планировщика на 1.87 микросекунд. Гипотеза о линейной зависимости подтверждена.

В процессе испытаний было отмечено, что FreeRTOS показала значительно лучший результат по времени планирования, но при этом расход RAM и Flash памяти был больше, что еще раз подтверждает важность учета компромисса между производительностью и использованием памяти при разработке и использовании систем для встроенных применений.

5 Выводы

Наше исследование наглядно продемонстрировало, что от конкретных решений при построении RTOS может напрямую зависеть производительность. В следствие этого разработчику рекомендуется перед выбором RTOS для своего проекта внимательно изучить документацию и исходный код для того чтобы выявить слабые и сильные места и определить применимость к своей задаче.

5.1 Зависимость оптимальной схемы планирования от специфики задачи

Существует широкий круг задач, решаемых с помощью встроенных микропроцессорных систем, которым достаточно коротких и быстрых обработчиков прерываний (которые обеспечивают вытеснение автоматически, по своей сути) и нескольких задач с равным приоритетом, планирующихся добровольно. Иногда в таких случаях применение даже простейшего планировщика неоправданно и может привести к ненужному усложнению кода и снижению производительности или надежности.

Но по мере усложнения устройств и постоянного увеличения требований к функциональности, применение планировщика становится логичным и естественным шагом, так как позволяет программисту разбить программу на независимые модули с четко продуманной схемой взаимо-

действия, тем самым обеспечив себе возможность относительно простого наращивания функциональности без ущерба для надежности системы.

Во многих случаях все задачи можно свести к набору процессов, исполняющихся с определенной частотой, что позволяет применить для планирования схему RMS. Для реализации RMS достаточно вытесняющего планировщика со статическими приоритетами (число возможных приоритетов должно соответствовать количеству задач с разными частотами). Математически строго доказано, что эта схема — лучшая схема со статическими приоритетами, однако не стоит забывать, что она не дает возможность использовать процессор со 100% занятостью, из-за чего при превышении определенного порога может наступить «голодание» и, как следствие, невозможность системы удовлетворить требованиям по времени и функциональности. В виду этого перед применением рекомендуется применить анализ (RMA)[6] для того, чтобы убедиться, что получившаяся система будет планируемой.

5.2 Зависимость оптимальной схемы планирования от имеющихся аппаратных ресурсов

Скорость и архитектура процессора, объем доступной оперативной и flash-памяти, другие характеристики выбранной платформы также могут оказывать решающее влияние на выбор планировщика и детали реализации. Например, во FreeRTOS увеличение числа приоритетов ведет к значительному увеличению количества используемой RAM из-за того, что создаются дополнительные очереди для выполняющихся процессов.

Можно заметить, что AvrX, которая создавалась с расчетом на минимизацию используемых аппаратных ресурсов (для исполнения на младших представителях семейства AVR), имеет гораздо более простой (а следовательно, и менее гибкий планировщик), чем FreeRTOS, которая позиционируется для гораздо более мощных систем (в числе поддерживаемых архитектур помимо AVR есть ARM7, ARM9, Cortex M-3, x86, AVR32 и др.). Большое количество поддерживаемых архитектур и гибкость планировщика FreeRTOS позволяют проектам, использующим ее, активно и относительно беспроблемно расширяться, в том числе значительно облегчается миграция на более мощную архитектуру (например, с AVR на ARM7).

5.3 Возможные модификации/оптимизации планировщиков

В каждом конкретном проекте программист может столкнуться с различными ограничениями, поэтому решения об оптимизации схем планирования во встроенных системах приходится принимать по каждому случаю отдельно. Например, может быть целесообразным изменить число приоритетов в системе, или метод организации очередей; в некоторых случаях необходимо реализовать наследование приоритетов, а в других оно будет лишь неоправданно расходовать ресурсы; часто бывает, что для решения задачи вполне достаточно использования RMS, но в других случаях может оказаться экономически выгодно реализовать схему EDF, чтобы осуществить максимальную утилизацию ресурсов конкретного процессора.

Следует обратить внимание, что в реальных проектах во многих случаях производительность всей системы будет определяться даже не планировщиком, а методами межпроцессного взаимодействия, в связи с этим при выборе ОС для проекта стоит проанализировать имеющиеся инструменты для синхронизации, обмена сообщениями и т.д.

5.4 Тенденции в развитии планировщиков

Ресурсы, доступные разработчикам встроенных систем, постоянно увеличиваются. В настоящее время разница между стоимостью 8-битных и 32-битных архитектур неуклонно снижается, в частности, младшие ARM практически сравнялись по стоимости со старшими AVR, обладая при этом гораздо большими возможностями (больший объем памяти, производительность, возможности по подключению ethernet и usb и многое другое). С другой стороны, многие устройства не требуют «жесткого» реального времени (hard real time), что сближает их с современными интерактивными системами (персональные компьютеры и др.).

Эти два фактора привели к тому, что уже сейчас в огромном количестве устройств используется *Linux* (ядро свободной операционной системы *GNU/Linux*, широко распространенной на персональных компьютерах и серверах) и специальная модификация для процессоров, не имеющих модуля управления памятью MMU (например, ARM7) — *μCLinux*. Эти ядра обеспечивают «гибкое» (soft) реальное время с помощью стандартного планировщика (сложностью $O(1)$)[11],[7] (в версиях 2.6.23 и вы-

ше применяется CFS[13]).

Если для задачи все же необходимо «жесткое» реальное время, можно воспользоваться гибридной концепцией, которую предлагают *Xenomai*, *RTAI* и *RTLinux*. В них можно реализовывать свои процессы, планируемые по желаемой схеме реального времени, а Linux (или μCLinux) исполняется как один из процессов с низким приоритетом. Это позволяет использовать огромное число уже написанных и отлаженных драйверов, пакетов для математических вычислений, систем пользовательского интерфейса и других приложений совместно с несколькими процессами, временные требования которых должны быть строго гарантированы. Обзор планировщика RTLinux (а также сравнение с VxWorks) можно найти в [8] и [9]. Примеры использования RMS и EDF в RTAI (а также большое количество других учебных материалов) представлены в [10].

Можно заметить, что в силу специфики задач для систем реального времени новых концепций, получивших сколько-нибудь массовое распространение, обнаружить не удалось. Однако в современных интерактивных системах происходит постоянная исследовательская и инженерная работа, благодаря которой характеристики этих систем значительно улучшились за последнее время. Например, в Linux версий 2.6.x планировщик был полностью переписан (по сравнению с 2.4.x), результат этого заметен даже обычным пользователям (в частности, при воспроизведении и записи multimedia)[11]. Совсем недавно вопрос о планировщике Linux широко обсуждался в прессе (см., например [12]), а в версии 2.6.23 появился полностью новый, написанный с нуля Инго Моллнаром (под влиянием идей Кона Коливаса) планировщик CFS (Completely Fair Scheduler)[13].

6 Список литературы

[1] Таненбаум Э.: *Современные операционные системы. 2-е изд.* — СПб.: Питер, 2002.

[2] Таненбаум Э., Вудхалл А.: *Операционные системы: разработка и реализация* — СПб.: Питер, 2006.

[3] Dr. Jürgen Sauermann, Melanie Thelen: *Concepts and Implementation of Microkernels for Embedded Systems* (доступна для скачивания, например, с *http://ihtik.lib.ru/dreamhost_electrotehn_4janv2007.html*)

[4] Richard Barry: *Official FreeRTOS documentation — http://www.freertos.org/*

[5] Larry Barello: *Official AvrX documentation — http://www.barello.net/avrx/Theory.htm*

[6] Mark Klein, Carnegie Mellon University: *Rate Monotonic Analysis — http://www.sei.cmu.edu/str/descriptions/rma_body.html*

[7] M. Tim Jones: *Inside the Linux scheduler — http://www.ibm.com/developerworks/linux/library/l-scheduler/*

[8] Victor Yodaiken and Michael Barabanov: *RTLinux Version Two — http://rtportal.upv.es/tutorial/documentacion/design.pdf*

[9] Oskar Hermansson and Stefan Holmer: *A comparison between the scheduling algorithms in RTLinux and in VxWorks - both from a and a contextual view — http://www.ida.liu.se/~TDDB72/rtproj/reports2006/04-v2-oskhe171steho564-RTLinux_VxWorks_scheduling.pdf*

[10] Harco Kuppens: *Realtime Linux (RTAI) - Radboud University Nijmegen Exercise #9 RMS & EDF — http://www.cs.ru.nl/lab/rtai/exercises/9.RMS-EDF/Exercise-9.html*

[11] *Wikipedia: O(1) scheduler — http://en.wikipedia.org/wiki/O%281%29_scheduler*

[12] corbet: *The Rotating Staircase Deadline Scheduler*, — *http://lwn.net/Articles/224865/*

[13] *Wikipedia: Completely Fair Scheduler* — *http://en.wikipedia.org/wiki/Completely_Fair_Scheduler*

A Тестовая программа для FreeRTOS

```
#include <stdlib.h>

/* Scheduler include files. */
#include "FreeRTOS.h"
#include "task.h"

#define LED     PORTE
#define LEDDDR  DDRE
#define LEDPIN  6

/*-----------------------------------------------------------*/
static void task00( void *pvParameters )
{
        while ( 1 ) {
                LED &= ~_BV( LEDPIN );
                taskYIELD();
        }
}
/*-----------------------------------------------------------*/
static void task10( void *pvParameters )
{
        while ( 1 ) {
                LED |= _BV( LEDPIN );
                taskYIELD();
        }
}
/*-----------------------------------------------------------*/
static void task20( void *pvParameters )
{
        while ( 1 ) {
                LED &= ~_BV( LEDPIN );
                taskYIELD();
        }
}
/*-----------------------------------------------------------*/
static void task30( void *pvParameters )
```

```
{
        while ( 1 ) {
                LED |= _BV( LEDPIN );
                taskYIELD();
        }
}
/*-----------------------------------------------------------*/
static void task01( void *pvParameters )
{
        while ( 1 ) {
                LED &= ~_BV( LEDPIN );
                taskYIELD();
        }
}
/*-----------------------------------------------------------*/
static void task11( void *pvParameters )
{
        while ( 1 ) {
                LED |= _BV( LEDPIN );
                taskYIELD();
        }
}
/*-----------------------------------------------------------*/
static void task21( void *pvParameters )
{
        while ( 1 ) {
                LED &= ~_BV( LEDPIN );
                taskYIELD();
        }
}
/*-----------------------------------------------------------*/
static void task31( void *pvParameters )
{
        while ( 1 ) {
                LED |= _BV( LEDPIN );
                taskYIELD();
        }
}
```

```
/*-----------------------------------------------------------*/
static void task02( void *pvParameters )
{
        while ( 1 ) {
                LED &= ~_BV( LEDPIN );
                taskYIELD();
        }
}
/*-----------------------------------------------------------*/
static void task12( void *pvParameters )
{
        while ( 1 ) {
                LED |= _BV( LEDPIN );
                taskYIELD();
        }
}
/*-----------------------------------------------------------*/
static void task22( void *pvParameters )
{
        while ( 1 ) {
                LED &= ~_BV( LEDPIN );
                taskYIELD();
        }
}
/*-----------------------------------------------------------*/
static void task32( void *pvParameters )
{
        while ( 1 ) {
                LED |= _BV( LEDPIN );
                taskYIELD();
        }
}
/*-----------------------------------------------------------*/
static void task03( void *pvParameters )
{
        while ( 1 ) {
                LED &= ~_BV( LEDPIN );
                taskYIELD();
```

```
        }
}
/*-----------------------------------------------------------*/
static void task13( void *pvParameters )
{
        while ( 1 ) {
                LED |= _BV( LEDPIN );
                taskYIELD();
        }
}
/*-----------------------------------------------------------*/
static void task23( void *pvParameters )
{
        while ( 1 ) {
                LED &= ~_BV( LEDPIN );
                taskYIELD();
        }
}
/*-----------------------------------------------------------*/
static void task33( void *pvParameters )
{
        while ( 1 ) {
                LED |= _BV( LEDPIN );
                taskYIELD();
        }
}

portSHORT main( void )
{
        // Enable LEDPIN for output
        LEDDDR |= _BV( LEDPIN );

        xTaskCreate( task00, NULL, configMINIMAL_STACK_SIZE,
                     NULL, tskIDLE_PRIORITY+1, NULL );
        xTaskCreate( task10, NULL, configMINIMAL_STACK_SIZE,
                     NULL, tskIDLE_PRIORITY+1, NULL );
        xTaskCreate( task20, NULL, configMINIMAL_STACK_SIZE,
                     NULL, tskIDLE_PRIORITY+1, NULL );
```

```
        xTaskCreate( task30, NULL, configMINIMAL_STACK_SIZE,
                     NULL, tskIDLE_PRIORITY+1, NULL );
        xTaskCreate( task01, NULL, configMINIMAL_STACK_SIZE,
                     NULL, tskIDLE_PRIORITY+1, NULL );
        xTaskCreate( task11, NULL, configMINIMAL_STACK_SIZE,
                     NULL, tskIDLE_PRIORITY+1, NULL );
        xTaskCreate( task21, NULL, configMINIMAL_STACK_SIZE,
                     NULL, tskIDLE_PRIORITY+1, NULL );
        xTaskCreate( task31, NULL, configMINIMAL_STACK_SIZE,
                     NULL, tskIDLE_PRIORITY+1, NULL );
        xTaskCreate( task02, NULL, configMINIMAL_STACK_SIZE,
                     NULL, tskIDLE_PRIORITY+1, NULL );
        xTaskCreate( task12, NULL, configMINIMAL_STACK_SIZE,
                     NULL, tskIDLE_PRIORITY+1, NULL );
        xTaskCreate( task22, NULL, configMINIMAL_STACK_SIZE,
                     NULL, tskIDLE_PRIORITY+1, NULL );
        xTaskCreate( task32, NULL, configMINIMAL_STACK_SIZE,
                     NULL, tskIDLE_PRIORITY+1, NULL );
        xTaskCreate( task03, NULL, configMINIMAL_STACK_SIZE,
                     NULL, tskIDLE_PRIORITY+1, NULL );
        xTaskCreate( task13, NULL, configMINIMAL_STACK_SIZE,
                     NULL, tskIDLE_PRIORITY+1, NULL );
        xTaskCreate( task23, NULL, configMINIMAL_STACK_SIZE,
                     NULL, tskIDLE_PRIORITY+1, NULL );
        xTaskCreate( task33, NULL, configMINIMAL_STACK_SIZE,
                     NULL, tskIDLE_PRIORITY+1, NULL );

        vTaskStartScheduler();

        return 0;
}
```

B Тестовая программа для AvrX

```
#include <avrx-io.h>
#include "avrx.h"
#include "hardware.h"

AVRX_GCC_TASKDEF(task00, 8, 0)
{
    while (1)
    {
    LED &= ~_BV( LEDPIN );
    AvrXYield();
    }
}
AVRX_GCC_TASKDEF(task10, 8, 0)
{
    while (1)
    {
    LED |= _BV( LEDPIN );
    AvrXYield();
    }
}
AVRX_GCC_TASKDEF(task20, 8, 0)
{
    while (1)
    {
    LED &= ~_BV( LEDPIN );
    AvrXYield();
    }
}
AVRX_GCC_TASKDEF(task30, 8, 0)
{
    while (1)
    {
    LED |= _BV( LEDPIN );
    AvrXYield();
    }
}
```

```
AVRX_GCC_TASKDEF(task01, 8, 0)
{
    while (1)
    {
    LED &= ~_BV( LEDPIN );
    AvrXYield();
    }
}
AVRX_GCC_TASKDEF(task11, 8, 0)
{
    while (1)
    {
    LED |= _BV( LEDPIN );
    AvrXYield();
    }
}
AVRX_GCC_TASKDEF(task21, 8, 0)
{
    while (1)
    {
    LED &= ~_BV( LEDPIN );
    AvrXYield();
    }
}
AVRX_GCC_TASKDEF(task31, 8, 0)
{
    while (1)
    {
    LED |= _BV( LEDPIN );
    AvrXYield();
    }
}
AVRX_GCC_TASKDEF(task02, 8, 0)
{
    while (1)
    {
    LED &= ~_BV( LEDPIN );
    AvrXYield();
```

```
    }
}
AVRX_GCC_TASKDEF(task12, 8, 0)
{
    while (1)
    {
    LED |= _BV( LEDPIN );
    AvrXYield();
    }
}
AVRX_GCC_TASKDEF(task22, 8, 0)
{
    while (1)
    {
    LED &= ~_BV( LEDPIN );
    AvrXYield();
    }
}
AVRX_GCC_TASKDEF(task32, 8, 0)
{
    while (1)
    {
    LED |= _BV( LEDPIN );
    AvrXYield();
    }
}
AVRX_GCC_TASKDEF(task03, 8, 0)
{
    while (1)
    {
    LED &= ~_BV( LEDPIN );
    AvrXYield();
    }
}
AVRX_GCC_TASKDEF(task13, 8, 0)
{
    while (1)
    {
```

```
    LED |= _BV( LEDPIN );
    AvrXYield();
    }
}
AVRX_GCC_TASKDEF(task23, 8, 0)
{
    while (1)
    {
    LED &= ~_BV( LEDPIN );
    AvrXYield();
    }
}
AVRX_GCC_TASKDEF(task33, 8, 0)
{
    while (1)
    {
    LED |= _BV( LEDPIN );
    AvrXYield();
    }
}

int main(void)                  // Main runs under the AvrX Stack
{
    AvrXSetKernelStack(0);

    LEDDDR |= _BV( LEDPIN );

    AvrXRunTask(TCB(task00));
    AvrXRunTask(TCB(task10));
    AvrXRunTask(TCB(task20));
    AvrXRunTask(TCB(task30));
    AvrXRunTask(TCB(task01));
    AvrXRunTask(TCB(task11));
    AvrXRunTask(TCB(task21));
    AvrXRunTask(TCB(task31));
    AvrXRunTask(TCB(task02));
    AvrXRunTask(TCB(task12));
    AvrXRunTask(TCB(task22));
```

```
    AvrXRunTask(TCB(task32));
    AvrXRunTask(TCB(task03));
    AvrXRunTask(TCB(task13));
    AvrXRunTask(TCB(task23));
    AvrXRunTask(TCB(task33));

    Epilog();                   // Switch from AvrX Stack to first task
}
```

C Иллюстрации