Снижение стоимости разработки промышленных систем управления посредством использования адаптивной декомпозиции системных ресурсов

Роман Кипрушенков

С ростом сложности и объёма программного кода вероятность проявления проблем нехватки процессорного времени в конечном продукте возрастает — усложняется этап отладки и интеграции разрабатываемой системы при финальной сборке, увеличиваются затраты по выявлению и устранению дефектов. Использование запатентованной технологии адаптивной декомпозиции позволяет обеспечить приложения гарантированным реальным временем процессорной обработки, нейтрализовать угрозы, связанные с захватом процессорных ресурсов вредоносным и сбойным программным обеспечением. Происходит сокращение издержек по устранению неполадок, вызванных дефицитом процессорных ресурсов, продукт становится более надёжным и защищённым, выход конечного продукта на рынок происходит быстрее.

Возрастающая сложность

Совсем недавно большинство промышленных систем управления имело ограниченные требования к программному обеспечению – как правило, оно содержало лишь несколько тысяч строк исходного кода. В настоящее время, однако, встраиваемая система управления может содержать в себе сотни тысяч или даже миллионы строк исходного кода и использовать большое количество взаимодействующих сложным образом программных компонентов, и это при условии ограниченного объёма памяти и процессорного времени.

Чтобы ускорить процесс разработки сложных систем, проект разделяется между многочисленными группами разработчиков, каждая из которых в ходе исполнения имеет свои собственные цели, схемы назначения приоритетов задач и подходы к оптимизации процесса выполнения. При данном параллельном способе разработки в процессе интегрирования подсистем, созданных каждой группой разработчиков, неизменно возникают вопросы, связанные с производительностью, прежде всего из-за того, что различные подсистемы начинают конкурировать между собой за системные ресурсы. Подсистемы, прекрасно работающие в изолированном режиме, в этом случае начинают медленно реагировать на запросы, а порой и вообще дают сбои. Все усугубляется еще и тем, что многие из этих проблем возникают только в процессе интеграции и проверочного тестирования, когда стоимость перепроектирования и перекодирования программного обеспечения очень высока.

Диагностика и решение таких проблем чрезвычайно трудны. Разработчики должны умело манипулировать приоритетами задач с возможным изменением поведения потоков в системе, а затем проводить повторное тестирование и уточнять собственные модификации. На этот процесс может уйти несколько рабочих недель, что приведёт к увеличению затрат и задержке выпуска продукта.

Разделение как метод решения

В последнее время понятие разделения стало рассматриваться как способ управления сложностью системы и обеспечения более высокой системной доступности. Кратко это можно определить как подход, дающий возможность командам разработчиков делить программное обеспечение на отдельные группы, где каждой группе назначается выделенная часть (или запас) системных ресурсов.

Как результат каждая составная часть обеспечивает стабильную, известную среду исполнения, которую команды разработчиков могут формировать и проверять индивидуально. Если программный процесс в пределах выделенной области без сбоев выполняется в ходе тестирования, можно с полной уверенностью сказать, что и на стадии интеграции он покажет такую же производительность.

Если деление статического ресурса, такого как память, на разделы не вызывает особых трудностей, разделить время процессорной обработки гораздо сложней. Так как стоимость и энергопотребление для встраиваемых систем изначально имеют ограничения, процессоры для таких систем имеют сравни-

тельно небольшой запас производительности. Принимая во внимание эти условия, можно сказать, что процессорное время — это ограниченный ресурс, распоряжаться которым нужно очень бережно, особенно во время высокой загрузки процессора. Адаптивное распределение процессорного времени позволяет достичь этой цели.

Планировщик ОС РВ

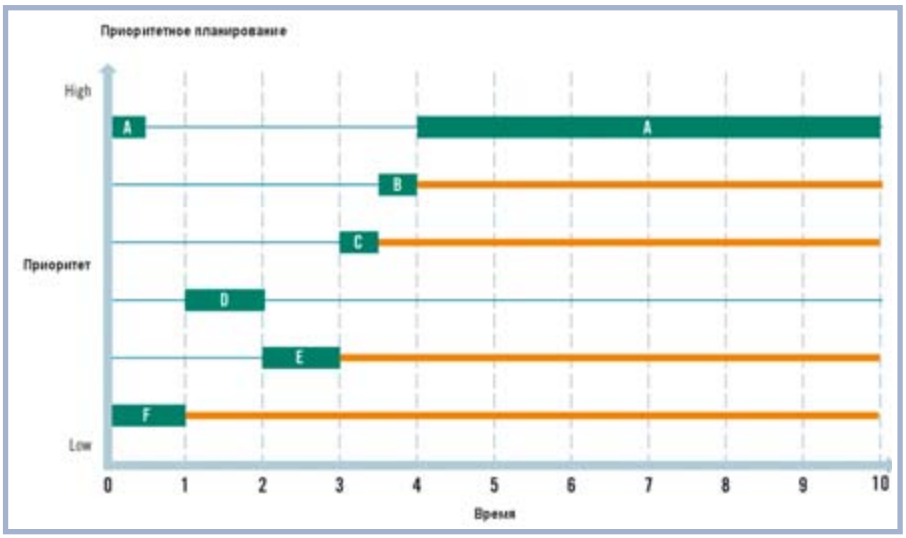
Чтобы понять необходимость распределения процессорного времени, нужно рассмотреть роль планирования во встраиваемой системе реального времени.

Для обеспечения переносимости кода большинство современных операционных систем поддерживает интерфейс прикладного программирования POSIX. Этот открытый стандарт определяет потоковую модель, в которой единичный процесс может содержать один или несколько исполняемых потоков. Чтобы достичь параллельности выполнения нескольких операций и производительности реального времени, необходимой для встраиваемых систем, операционные системы реального времени, основанные на стандарте POSIX, используют потоковый планировщик с приоритетными прерываниями.

Такой тип планировщика всегда позволяет назначить процессорное время потоку с более высоким приоритетом, требующему обработки. Когда поток с более высоким приоритетом (обычно запускаемый внешним событием) готов к обработке, происходит замещение им любого запущенного потока с более низким приоритетом.

Планировщик, основанный на приоритетах, обладает рядом преимуществ, включая

- предсказуемость периодов ожидания: назначая требующие немедленного исполнения функции потокам с высокими приоритетами, разработчики могут чётко контролировать время, которое понадобится системе для обработки внешних событий;
- параллельность выполнения и гибкость: используя приоритетный планировщик, встраиваемые системы могут одновременно обрабатывать различные типы задач (регулярно возникающие задачи с установленным временем исполнения, высокоприоритетные событийно-управляемые задачи и задачи, выполняемые в фоновом режиме);
- привычность и надёжность: приоритетный планировщик широко используется в промышленных приложениях и хорошо понятен для разработчиков встраиваемых систем.



Условные обозначения:

в процессе выполнения; в режиме ожидания; в режиме готовности.

Рис. 1. Распределение задач при приоритетном планировании

Когда поток обработан и готов к запуску, он располагается в очереди готовых потоков с одинаковым уровнем приоритетов. Политика работы планировщика, установленная для каждого потока разработчиком, определяет, какой поток из очереди готовых к запуску потоков будет выполняться следующим. Стандарт POSIX описывает три политики работы планировщика.

- Правило «первым пришёл – первым обслужен» (FIFO) гарантирует, что поток, выбранный для запуска, будет выполняться, пока не остановится или не будет вытеснен потоком с более высоким приоритетом.
- Правило циклического алгоритма (round robin) гарантирует, что поток будет выполняться, пока не остановится, не будет вытеснен потоком с более высоким приоритетом или не закончится отведённое ему время выполнения.
- Спорадическое правило определяет, как долго процесс будет выполняться в пределах отведённого ему времени.

При работе планировщика в спорадическом режиме поток имеет два назначенных приоритета – нормальный и низкий — и запускается с нормальным приоритетом в определённый период времени. Когда время работы с нормальным приоритетом заканчивается, поток запускается с низким приоритетом. Значение периода работы с нормальным приоритетом регулярно обновляется. Такой подход помогает предотвратить работу единичного потока с высоким приоритетом в течение продолжительного времени. Тем не менее часто разработчики ограничивают использование спорадического режима работы планировщика, так как его трудно использовать в системах с большим количеством потоков.

Управление приоритетами

Планировщик, основанный на приоритетах, функционирует не совсем «честно». Высокоприоритетная задача, требующая обработки, может захватить всё процессорное время, лишив обработки все другие задачи. В результате разработчикам приходится очень осторожно назначать и тестировать приоритеты задач во всей системе.

Когда возрастает сложность системы и увеличивается количество разработчиков, назначение и учёт приоритетов для большого количества потоков становится непосильно трудной задачей и причиной разногласий среди разработчиков. Понимая, что неограниченное назначение приоритетов может привести к хаосу (или к прекращению функционирования системы), разработчики часто сокращают количество используемых приоритетов. Однако такое решение имеет нежелательный побочный эффект — увеличение периодов ожидания. Так как большое количество потоков имеет одинаковый приоритет, очередь готовых к запуску потоков становится очень длинной. Готовый к запуску поток ожидает выполнения, пока до него не дойдет очередь.

Приоритетное планирование гарантирует, что наиболее критические задачи получают доступ к ресурсам процес-

сора, но также может быть причиной проблем в случае, когда высокоприоритетная задача неумышленно или намеренно потребляет все доступные циклы процессора (рис. 1). Например, задача А препятствует всем другим задачам в получении доступа к процессору с момента своего запуска (4-я единица времени).

Более того, планировщик, основанный на приоритетах, может допустить полную загрузку процессора вредоносным программным обеспечением или процессом отказа от обслуживания поступивших запросов (DoS attack), делая таким образом систему недоступной для пользователей.

В целях предотвращения таких проблем разработчики программного обеспечения выбирают одно из двух возможных решений:

- использование управляющего процесса-диспетчера или «сторожевого» процесса, который отслеживает загрузку процессора потоками.
- использование алгоритма выделения процессорного времени, управляемого операционной системой, который назначает определённое процессорное время каждому потоку.

«Сторожевые» процессы

Иногда группа связанных потоков лишает другие потоки процессорного времени. В других случаях недостаток времени процессорной обработки — это результат зацикливания одного высокоприоритетного потока. Чтобы предотвратить такие моменты, «сторожевой» процесс следит за загрузкой процессора и осуществляет корректирующие действия в случае обнаружения превышения потоком бюджета установленного времени процессорной обработки.

Тем не менее не существует простого способа нейтрализовать использование процессора вышедшим из-под

Рис. 2. Пример упрощённой системы автоматизации

контроля потоком. В связи с этим «сторожевой» процесс должен использовать кардинальные меры, такие как перезапуск защищённого процесса (и всех его порождённых процессов) или снижать приоритеты защищённых потоков или процессов.

Более того, метод «сторожевого» процесса не работает гладко со всеми процессами и потоками. Например, некоторым процессам необходимо полностью использовать процессор в определенные моменты времени, это порождает необходимость обращения «сторожевого» процесса к списку исключений, чтобы поддерживать правильное функционирование системы.

«Сторожевой» процесс может также поглощать значительный объём процессорного времени. Например, приходится через определённые интервалы времени регулярно посылать запрос операционной системе, чтобы правильно определять использование процессора всеми потоками, а затем сравнивать полученную информацию с данными предыдущего интервала. Также необходимо учитывать все случаи исключения, упомянутые ранее. Так как «сторожевой» процесс выполняется с более высоким приоритетом, чем все другие потоки, за которыми он следит, он и сам может стать причиной лишения задач процессорного времени.

«Сторожевой» процесс обладает и другими побочными эффектами, включая замедленное время отклика, ограничение использования процессорного времени «законным» процессом.

Алгоритм выделения процессорного времени, управляемого операционной системой

Некоторые операционные системы реального времени поддерживают фиксированное выделение процессорных ресурсов группам потоков. Такой подход обеспечивает некий контейнер для процессов и потоков, называемый разделом, которому выделяется фиксированная доля процессорного времени. Например, для группы потоков, имеющих общее предназначение, разработчик выделяет раздел с фиксированным бюджетом в 5% от общего объема процессорного времени. Планировщик раздела будет в этом случае гарантировать, что данный раздел получит назначенный объём процессорного времени.

Используя подход временного разделения, при разработке можно задать бюджет гарантированного процессорного времени для каж-

дой значимой подсистемы программного продукта. Разработчик таким образом получает гарантию того, что полная загрузка системных ресурсов одной подсистемой не повлияет на функционирование других подсистем. Такой подход предотвращает поглощение всех ресурсов процессора одним потоком, даже если поток запущен с самым высоким уровнем приоритета.

В пределах раздела обработка потоков планируется с учётом традиционных правил прерывания планировщика, основанного на приоритетах. К разделу применяются стандартные правила планировщика: FIFO, round robin и спорадическое правило. В результате каждый раздел становится мини-средой исполнения.

Работа планировщиков разделов различается. Некоторые чётко распределяют бюджетное время процессорной обработки, так что каждый раздел получает установленный бюджет процессорных ресурсов, даже когда в этом нет особой необходимости. Другие могут динамически распределять неиспользуемые ресурсы процессора другим разделам, максимизируя таким образом использование процессора и позволяя системе справляться с пиками запросов.

Упрощение процесса разработки и интеграционного тестирования

Распределение процессорного времени облегчает процесс параллельной разработки, позволяя разработчику назначить гарантированный бюджет процессорных ресурсов для каждой субсистемы. Такой подход устраняет необходимость применять глобальные схемы приоритетов и позволяет разработчикам определять схемы приоритетов на уровне субсистем в зависимости от потребностей. В результате становится возможным ведение процесса параллельной разработки.

При тестировании функционирования субсистемы внутри раздела разработчики могут создавать условия полной загрузки процессора вне этого раздела, симулируя таким образом работу системы при полной загрузке процессора. Это позволяет разработчикам производить отладку своего программного кода в неблагоприятных условиях полного использования системных ресурсов. В результате решается большое количество проблем, связанных с производительностью и использованием процессорного времени до стадии интеграции системы.

Чтобы оценить все преимущества разделения процессорного времени, рассмотрим относительно простую систему, спроектированную без использования алгоритма распределения процессорного времени. Система, изображённая на рис. 2, содержит следующие процессы:

- процесс со средним приоритетом, ответственный за функционирование локального человеко-машинного интерфейса;
- процесс со средним приоритетом, производящий периодическое сенсорное сканирование;
- процесс с высоким приоритетом, ответственный за управление двигателем;
- процесс с низким приоритетом, ответственный за функционирование удалённой системы мониторинга, которая посылает обновлённые данные центральной системе мониторинга, основанной на Интернет-технологии.

На этапе интеграции, когда вся система собрана, система web-мониторинга работает прекрасно до тех пор, пока не используется локальный человеко-машинный интерфейс. При его использовании система мониторинга «замирает» и перестаёт отображать обновленные данные. Проблемы возникают, когда к процессу управления двигателем, работающему с высоким приоритетом, добавляется процесс, отвечающий за команды локального человеко-машинного интерфейса, абсолютно вытесняя процесс мониторинга с более низким приоритетом. Анализ приоритетов объясняет, почему это происходит. При полной загрузке процессора процессы с малым приоритетом не получают процессорного времени обработки.

Чтобы решить данную проблему, разработчик назначает процессу, отвечающему за команды локального человеко-машинного интерфейса, более низкий приоритет, чем процессу мониторинга. Однако это делает невозможным адекватную работу человеко-машинного интерфейса. Назначение среднего приоритета трём процессам: процессу, производящему сенсорное сканирование, процессу, ответственному за функционирование удалённой системы мониторинга, и процессу, отвечающему за команды локального человеко-машинного интерфейса, – также ничего не даёт – производительность трёх процессов снижается. Так как переназначение приоритетов не даёт должного результата, разработчик должен сделать следующий шаг и попытаться изменить поведение потока, а это дорогостоящее решение на стадии интеграции.

Разделение процессорного времени позволяет избежать всех этих проблем. Например, разработчик может установить бюджет процессорного времени для каждого из четырех разделов (рис. 3): 10% для процесса, отвечающего за команды локального человеко-машинного интерфейса, 10% для процесса, ответственного за функционирование удалённой системы мониторинга, 30% для процесса, производящего сенсорное сканирование, и 50% для процесса, ответственного за управление двигателем.

При таком подходе каждый раздел будет функционировать в соответствии с выделенным бюджетом процессорного времени. При сборке системы на стадии интеграции все процессы гарантированно получат выделенную бюджетом долю процессорного времени. В результате ни один процесс не будет обделён процессорными ресурсами и, более того, у разработчика появится возможность настройки производительности системы простой регулировкой бюджетов процессорного времени для каждого из разделов.

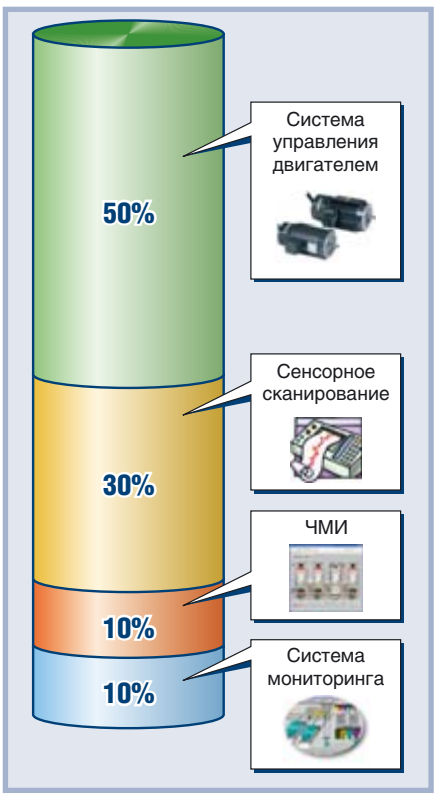


Рис. 3. Распределение бюджетов процессорного времени между процессами

Подсчёт экономической эффективности

Использовать схему «создание, тестирование, обнаружение, отладка» в условиях, когда задачам недостаёт процессорных ресурсов, – это очень дорого. Часто недостаток системных ресурсов — это следствие сбойного, необъяснимого поведения системы, а не серьёзных поломок. В результате достаточно трудно собрать адекватную информацию о возникших проблемах. Обычно поиск такого рода неисправностей требует как наличия детальных знаний об устройстве системы, так и большого объёма кропотливой работы — в итоге требуется целая команда специалистов, чтобы обнаружить проблему и найти пути её решения. Процесс включает в себя следующие действия.

1. Тестер создаёт отчет о проблеме, описывающий неожиданное поведение системы при тестировании. Так как проблему сложно воспроизвести, тестер не может собрать достаточную информацию, способную облегчить решение проблемы.
2. Разработчик производит серию тестовых запусков, пытаясь воспроизвести описанную проблему. Как правило, разработчик обнаруживает, что дело не в конкретном процессе, а в некотором другом, который полностью загружает процессор, лишая таким образом все другие процессы процессорного времени обработки.
3. На этом этапе границы поиска решения проблемы расширяются, в процесс вовлекается больше специалистов. Решение может состоять в настройке приоритетов потока или в изменении поведения процесса.

Таблица 1

Затраты времени на решение проблемы

Задача	Необходимое время
1. Проверка и создание отчёта о проблеме (1 человек тестирует и создает отчёт)	1 день
2. Первоначальный этап выявления неисправности (1 человек отвечает за устранение проблемы)	2 дня
3. Совместное выявление неисправности (3 человека принимают участие)	3 дня
4. Совместное решение проблемы (3 человека принимают участие, каждый тратит 3 рабочих дня)	9 дней
5. Перепроверка (1 человек заново тестирует систему)	1 день
Всего трудозатрат	**16 дней**

4. Каждый привлеченный к процессу разработчик выполняет необходимые изменения и тестирует свою часть проекта, затем интегрирует изменения в систему.
5. Тестер выполняет повторное тестирование и закрывает отчёт об ошибке при условии, что в ходе этих мероприятий не было выявлено дополнительных проблем или ошибок.

Исходя из описанного алгоритма поиска неисправностей, составляем таблицу затрат на отладку одной проблемы (табл. 1).

Из этого примера видно, как недостаток процессорного времени проекта обработки может увеличить затраты на разработку и задержать сдачу проекта; в нашем случае это две-три календарные недели. И это при том, что в нашем примере рассматривается система только с четырьмя потоками, в то время как многие промышленные системы содержат сотни и даже тысячи потоков, для каждого из которых существует сотня способов занять всё процессорное время.

Так как стадия интеграции системы занимает больше всего времени при разработке проекта, оптимизация этой стадии приводит к сокращению издержек производства и тем самым способствует более быстрому выходу конечного продукта на рынок.

Минимум усилий

С ростом сложности и объёма программного кода вероятность появления проблем нехватки процессорного времени и других дефектов в конечном продукте возрастает. Стоимость устранения таких неполадок, после того как система была собрана, возрастает многократно, не говоря уже об ударе по репутации разработчика и других финансовых издержках, с этим связанных. Фирмы-поставщики и разработчики, которые создают программные продукты для систем промышленной автоматизации, должны использовать любые возможности и методы, находящиеся в их распоряжении, чтобы обеспечить корректность отладки и полноценное тестирование своих программ.

Более сложная задача, однако, найти и воплотить технологии разработки, которые минимизируют усилия разработчиков и используемые компьютерные ресурсы. При условии правильного применения технология адаптивного распределения процессорного времени является таким решением. Более того, она

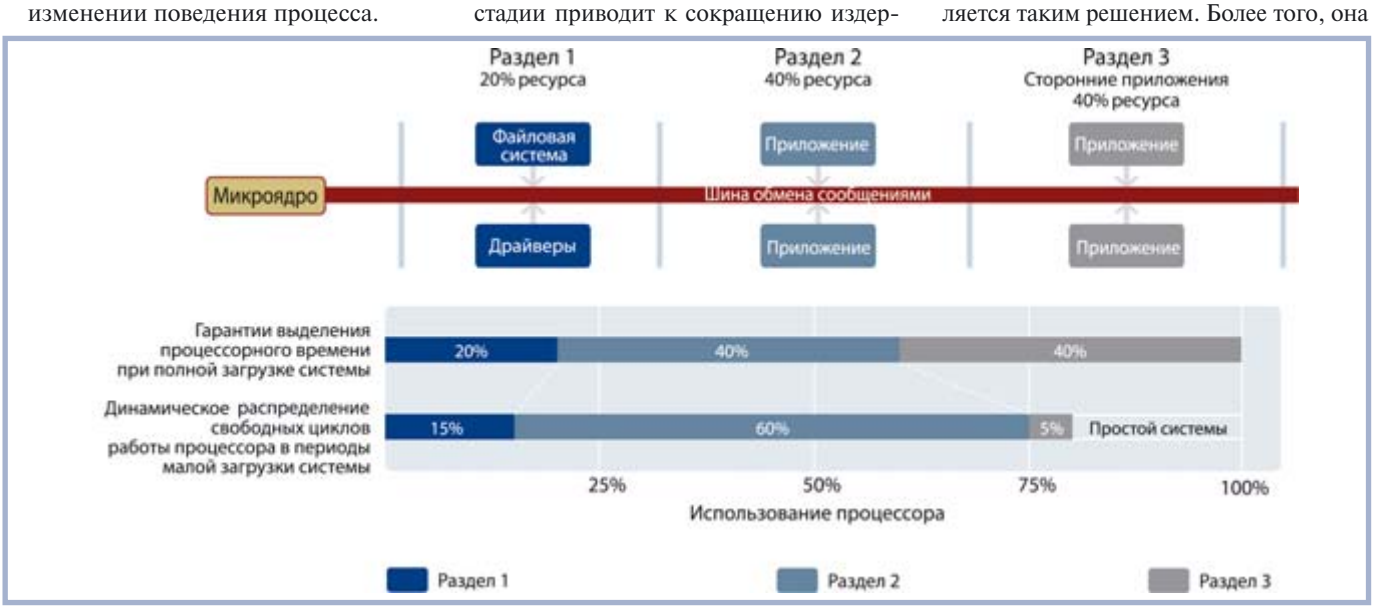


Рис. 4. Распределение процессорных ресурсов при использовании технологии адаптивной декомпозиции

обеспечивает повышение надёжности и отказоустойчивости систем, предотвращая захват ресурсов процессора вредоносными программами или процессами отказа от обслуживания поступивших запросов (DoS attack). Таким образом, мы имеем технологию, которая позволяет разработчикам встраиваемых приложений создавать не только хорошо интегрируемые, но и хорошо защищённые системы.

Технология адаптивной декомпозиции

Запатентованная технология адаптивной декомпозиции QNX позволяет обеспечить вашим приложениям гарантированное реальное время выполнения, нейтрализовать угрозы и защитить вашу систему.

Каких целей можно достичь путём выполнения применения технологии адаптивной декомпозиции QNX при разработке промышленных систем управления?

Создавать защищённые системы: вредоносное программное обеспечение может помешать нормальной работе важных системных функций, ограничивая время их доступа к процессорной обработке. Чтобы избежать этого, технология адаптивной декомпозиции QNX позволяет вам создавать раздел между ключевыми процессами вашей системы и сторонним программным обеспечением, таким образом защищая систему.

Повысить эффективность использования процессора: в отличие от статичных подходов к декомпозиции с помощью циклического планировщика, технология адаптивной декомпозиции распознаёт циклы загрузки процессора и периоды, когда система работает вхолостую. С технологией адаптивной декомпозиции исключается неэффективное расходование времени при простое системы. Так, неиспользуемые циклы работы процессора с одного блока процессов могут быть динамически перераспределены на другие, следовательно, общее время эффективной работы процессора увеличивается. Технология адаптивной декомпозиции обеспечивает простое надёжное решение для систем, выполняющих ограниченный круг задач с интенсивной загрузкой процессора.

Быстро начать работу: технология адаптивной декомпозиции использует стандартную модель программирования POSIX, у вас будет возможность использовать знакомую среду разработки, привычную для любой встраиваемой системы технику программирования и отладки. Вы можете применить технологию адаптивной декомпозиции, просто определив размер блока и решив, какое приложение или процесс будет использовать определённый блок. С технологией адаптивной декомпозиции QNX приложения и системные службы будут запускаться в собственных блоках.

Запатентованная технология адаптивной декомпозиции от компании QNX Software Systems позволяет выделять приложениям и процессам гарантированный ресурс процессорного времени при полной загрузке системы, а также динамически распределяет свободные циклы работы процессора в периоды его малой загрузки (рис. 4).

Обеспечивая гарантированное время процессорной обработки для каждой программной подсистемы, адаптивная декомпозиция заметно сокращает трудозатраты на интеграцию всей системы. ●

Автор — сотрудник компании SWD Software
Телефон: +7 (812) 373-0260, 702-0833
Факс: +7 (812) 373-0497