ПРОГРАММИРОВАНИЕ, 2024, № 5, с. 14–30

ПАРАЛЛЕЛЬНОЕ И РАСПРЕДЕЛЕННОЕ ПРОГРАММИРОВАНИЕ

УДК 004.031.6

СОВМЕСТНОЕ ИСПОЛЬЗОВАНИЕ СОПРОГРАММ И СОБЫТИЙ ДЛЯ ПРОГРАММИРОВАНИЯ ВСТРОЕННЫХ СИСТЕМ

© 2024 г. П. В. Минин[a],[*]

[a]ООО “КБ “ДОРС”

111399 Москва, Федеративный пр., д. 5, к. 2, Россия,

[*]E-mail: p.minin@dors.ru

Поступила в редакцию: 24.03.2023 г.
После доработки: 04.11.2023 г.
Принята к публикации: 12.05.2024 г.

Предложена новая методология построения программ для встроенных систем реального времени. Программа, написанная на языке C/C++, исполняется без помощи операционной системы и позволяет реализовать как событийно-ориентированный подход, так и параллельное выполнение с помощью сопрограмм. Обработка событий и выполнение сопрограмм производится в ядре мягкого реального времени на уровне приоритета программных прерываний. Сопрограмма рассматривается как частный случай события, где возобновляемая функция сопрограммы выполняется в качестве обработчика события. Для возобновления сопрограммы после приостановки ее событие повторно отправляется на обработку, до тех пор пока возобновляемая функция не будет полностью выполнена. Таким образом, в ядре происходят как обработка простых событий, так и планирование выполнения сопрограмм. Событиям могут назначаться различные приоритеты обработки. Ядро мягкого реального времени может быть расширено для работы в симметричной многопроцессорной системе. Сочетание сопрограмм и простых событий позволяет реализовать технологию fork/join, а также средства параллельного программирования, сходные с инструментами языков Go и occam. Новая методология была воплощена на языке C++ в виде библиотеки DORSECC, адаптированной для различных процессоров ARM и Blackfin. С ее помощью были созданы встроенные вычислительные системы реального времени, применяемые в серийно выпускаемых машинах для сортировки банкнот. Эти системы используются как для распознавания образов методом каскада классификаторов, так и для управления датчиками и приводами механизма с применением автоматного программирования. Их общее количество в эксплуатации превысило 20000. Ядро мягкого реального времени в указанных системах обеспечивает среднюю длительность обработки события на уровне десятков микросекунд при субмикросекундных накладных затратах.

Ключевые слова: режим реального времени, встроенный, “голое железо”, параллелизм, управляемый событиями, сопрограмма, C++, программное прерывание, синхронизация, многопоточный код

DOI: 10.31857/S0132347424050027, **EDN:** OMADOW

1. ВВЕДЕНИЕ

Чаще всего, при создании встроенных систем используется самодостаточная (bare metal) программа, написанная с использованием языков C, C++ или их комбинации. Самодостаточная программа содержит в себе все необходимые компоненты и не требует операционной системы (ОС). Значительно реже используются программы, работающие под управлением ОС реального времени (FreeRTOS, ThreadX, WxWorks, LynxOS, PikeOS и другие) либо даже ОС общего назначения Linux. Отказ от применения ОС в пользу самодостаточной программы обычно объясняется ресурсными ограничениями аппаратной платформы или же затруднениями при управлении сложной аппаратурой в реальном времени. Самодостаточная программа позволяет избежать накладных затрат, связанных с операционной системой. В конечном счете, она обеспечивает решение, наиболее оптимальное с точки зрения быстродействия и требуемых аппаратных ресурсов. Однако, платой за эту оптимизацию оказывается сложность программирования. Наибольшие трудности, как показывает опыт, связаны с реализацией параллельного выполнения кода, а также асинхронной обработки.

ОС реального времени предоставляют возможность использовать концепцию потоков в качестве основного механизма параллельной обработки. Для самодостаточной программы с использованием C/ C++, до последнего

времени, самим языком программирования не предусматривались средства параллельного выполнения. Только недавно в C++ был реализован механизм сопрограмм (coroutine, начиная со спецификации C++ 20). Однако, сопрограммы C++ во встроенных системах пока еще не нашли широкого применения, отчасти из-за сложности самой концепции и ее непривычной терминологии.

1.1 Сопрограммы

Сопрограммы [1] не являются потоками в современном понимании, поскольку реализуют параллельное исполнение нескольких последовательностей операторов с явной передачей управления между ними, без возможности вытеснения. Достаточно давно известны библиотечные реализации сопрограмм на языках C [2, 3] и C++ [4, 5, 6], которые делятся на стековые и бесстековые. Концепция стековой сопрограммы, по своим выразительным возможностям, максимально приближена к традиционному потоку. Обратной стороной высоких выразительных возможностей оказываются большие накладные расходы на управление сопрограммой. Для бесстековых сопрограмм не обеспечивается сохранение стековых переменных одной сопрограммы на период передачи управления от нее другой сопрограмме. Это накладывает существенные ограничения на стиль программирования. С другой стороны, бесстековые сопрограммы обеспечивают минимальные требования по памяти и затраты времени, поскольку при переключении от одной сопрограммы к другой не требуется сохранения и восстановления стека и связанных с ним элементов контекста. Сравнение стековых и бесстековых сопрограмм вызвало интенсивное обсуждение, которое частично отражено, например, в [7]. Обзор по применимости сопрограмм во встроенных системах приведен в [8].

В некоторых реализациях сопрограммам назначается величина приоритета. Приоритетный запуск сопрограммы на исполнение достигается двумя независимыми способами: с помощью диспетчера сопрограмм и с использованием приоритета потока исполнения. После того, как очередная сопрограмма прерывает свое исполнение, диспетчер передает управление наиболее приоритетной сопрограмме, готовой к выполнению. Если же ОС представляет программе потоки исполнения с разными приоритетами, то для выполнения более приоритетных сопрограмм может выделяться более приоритетный поток. Примерами можно назвать язык Kotlin [9] и ОС реального времени FreeRTOS [10]. Отметим, что в самодостаточной программе имеется единственный поток исполнения, поэтому для обеспечения приоритетов сопрограмм невозможно использовать потоки с разными приоритетами.

На сегодняшний день, существуют десятки реализаций сопрограмм для языков C/C++, ни одна из которых не смогла пока стать стандартом де-факто. Обращает на себя внимание тот факт, что в подавляющем большинстве случаев в них не задействован такой мощный асинхронный инструмент реального времени, как система прерываний процессора. Более того, обслуживание прерываний обычно полностью изолируется от выполнения детерминированной последовательности процессорных команд, создаваемой компилятором для кода сопрограмм.

1.2 Обработка событий

Значительная часть задач управления физическим устройством сводится к описанию в виде конечного автомата. Для конечного автомата естественным представлением является система, управляемая событиями [11]. Использование потоков (threads) для решения многих задач приводит к утяжеленным, неоптимальным решениям, в то время как эти же задачи наиболее просто решаются в концепции управления событиями [12, 13].

В простых встроенных системах реального времени обработка событий, возникающих в аппаратуре, традиционно реализуется с помощью взаимосвязанных обработчиков прерываний. Подобный подход позволяет получить максимальное быстродействие, но основан на приемах программирования, часто напоминающих трюки, порождающих скрытые ошибки и сложных для верификации. Для функционально сложного управления встроенной системой по событиям используются специализированные библиотеки программных инструментов (frameworks), например, Quantum Leaps [14, 15]. Кроме того, в практическом программировании часто применяется шаблон проектирования, известный как “Наблюдатель” (Observer) [16] и используемый в качестве основы менеджера событий. Для каждого получаемого события такой менеджер обеспечивает вызов соответствующего ему обработчика.

В операционных системах общего назначения используется механизм отложенной обработки данных, формируемых в аппаратных прерываниях, на менее приоритетном уровне. По своей сути, он также относится к обработке событий. В качестве примеров можно привести отложенный

вызов процедуры (DPC) в MS Windows [17], или отложенные задачи в Linux [18], которые выполняются в потоке ядра (Workqueue) или в контексте программного прерывания (Tasklet).

Для организации последовательной обработки асинхронно возникающих событий обычно применяется входная очередь системы обработки событий.

2. МОТИВАЦИЯ К РАЗРАБОТКЕ

Бесстековые сопрограммы представляют существенный интерес как средство обеспечения многозадачности во встроенных системах, где не используется ОС. Главное преимущество бесстековой сопрограммы перед потоком ОС состоит в значительно меньших накладных затратах на переключение выполнения задач. Оно связано с размером переключаемого контекста, который в случае бесстековой сопрограммы может быть уменьшен до одного-двух слов процессора.

Однако, классическая сопрограмма плохо подходит для работы в условиях реального времени, поскольку не может обеспечить гарантированное время реакции (латентность, latency) на асинхронный стимул, такой, как аппаратное прерывание. Это можно увидеть в сравнении с приоритетными вытесняющими потоками (задачами, процессами), применяемыми в ОС реального времени. Латентность запуска потока, ранее остановленного в ожидании прерывания, при гарантии вытеснения составляет

$$T_{TL} = T_{IL} + T_{TSW} + T_{OVH} \quad (1).$$

Здесь T_{IL} обозначает латентность обработки прерывания, а T_{TSW} – время переключения контекста с одного потока на другой. При помощи T_{OVH} обозначена общая продолжительность вспомогательных операций, включающих в себя выполнение драйвера прерывания до момента отправки сигнала потоку, а также накладные расходы на синхронизацию и диспетчеризацию. Видно, что T_{TL} складывается как сумма продолжительностей системных операций ОС, которые невелики и хорошо детерминированы.

Классические сопрограммы в самодостаточной программе выполняются в единственном имеющемся потоке исполнения и обеспечивают кооперативный тип многозадачности. Чтобы отреагировать на стимул запуском соответствующей сопрограммы, в потоке исполнения должен работать диспетчер, управляющий выполнением набора сопрограмм. Диспетчер может отправить сопрограмму на исполнение, только дождавшись, чтобы предшествующая сопрограмма самостоятельно уступила процессор. Латентность запуска бесстековой сопрограммы в ответ на стимул составляет

$$T_{CL} = T_{IL} + T_{CY} + T_{OVH} \quad (2),$$

где T_{CY} есть время выполнения текущей сопрограммы от момента обработки стимула в прерывании до момента освобождения процессора. Текущая сопрограмма уступает процессору, когда ее последовательность исполнения доходит до специальной операции YIELD или TRANSFER, без какой-либо синхронизации с появлением стимула. Поэтому $max(T_{CY}) = max(T_{CQ})$, где T_{CQ} есть квант времени, в течение которого сопрограмма владеет процессором. Этот квант определяется только расстановкой YIELD или TRANSFER в коде исполняемых сопрограмм. Иначе говоря, T_{CY} ограничивается сверху наибольшим значением кванта T_{CQ} для всех выполняемых сопрограмм, но, за счет асинхронного появления стимула, может случайным образом принимать любые меньшие значения. Заметим, что квант T_{CQ} должен существенно превышать время переключения сопрограмм, поскольку при слишком частом переключении недопустимо растет доля накладных расходов процессорного времени. За счет всех указанных факторов, латентность T_{CL} оказывается плохо предсказуемой и достаточно большой.

Латентность реакции на стимул считается основным параметром, по которому можно судить о возможности работы программы в реальном времени. По этой характеристике классические сопрограммы значительно уступают вытесняющим потокам. Достижение стабильно малой латентности запуска сопрограммы представляет собой важную проблему, пока еще не нашедшую адекватного разрешения. Вытесняющие сопрограммы ввода-вывода в языке Modula-2 [19] могли бы показаться подходящим решением. Однако, при внимательном рассмотрении видно, что они обладают всеми свойствами потока в современном понимании, и потому их неправомерно называть сопрограммами.

С другой стороны, обработка событий позволяет достичь малой и предсказуемой латентности реакции на внешний стимул. Для наиболее быстрого запуска и исполнения функции-обработчика может использоваться либо программное прерывание, либо вытесняющий поток реального времени. В качестве примеров можно привести упомянутые ранее механизмы Tasklet и Workqueue [18].

В отсутствии других событий в очереди, при использовании программного прерывания, латентность можно приблизительно представить в виде

$$T_{EL} = 2T_{IL} + T_{OVH} \quad (3),$$

где $2T_{IL}$ соответствует сумме латентностей аппаратного и программного прерывания. Когда используется обработка в вытесняющем потоке, то для оценки латентности может использоваться формула (1). В обоих случаях, затраты времени на диспетчеризацию с использованием очереди событий включены в T_{OVH}. Вход в прерывание выполняется быстрее переключения контекста ($T_{IL} \ll T_{TSW}$), благодаря чему программное прерывание обеспечивает существенно более низкую латентность, чем вытесняющий поток. Для самодостаточной программы, работающей без ОС и потоков, обработка в программном прерывании остается наилучшим возможным выбором.

В то же время, использование программного прерывания ограничивает круг задач, которые могут быть решены с помощью обработки событий. Как и для любого обработчика прерывания, в функции обработки события не допускается ожидание заданного условия или готовности примитива синхронизации. Кроме того, обработка должна быть ограничена по длительности, чтобы не нарушить своевременную обработку других событий.

Естественным решением перечисленных проблем было бы объединение высокой скорости реакции на асинхронный стимул, характерной для обработки событий, с возможностями сопрограмм для параллельной обработки и выполнения синхронизации. Насколько нам известно, подобный подход ранее не рассматривался. При его реализации важно сохранить такое фундаментальное преимущество бесстековых сопрограмм, как очень малые накладные затраты времени на переключение.

Мы постарались создать простую методологию для совместного применения сопрограмм и управления событиями в рамках самодостаточной встроенной программы, целиком написанной на C/C++. На ее основе нами была создано переносимое ядро обработки событий и сопрограмм в виде библиотеки DORSECC (DORS Event and Coroutine Core).

Важными требованиями к методологии были ее независимость от архитектуры процессора и отказ от использования ассемблерных вставок, за исключением стандартного кода для вызова функции обработчика прерывания, а также для возбуждения программного прерывания. Возможность дальнейшего масштабирования для использования в симметричных мультипроцессорных системах рассматривалась как обязательное требование.

3. ПРЕДЛАГАЕМОЕ РЕШЕНИЕ

Предполагается, что самодостаточная программа имеет три основных уровня исполнения кода. На наименее приоритетном уровне исполняется код главной функции *main*(). Во встроенных системах эту функцию часто называют суперциклом. Она обеспечивает первоначальную инициализацию и последующее циклическое выполнение действий, не критичных по времени, например таких, которые обеспечивают работу пользовательского интерфейса или встроенного веб-сервера. В случае мультипроцессорной системы, на этом уровне для каждого из ядер исполняется код главной функции, назначенной заданному ядру.

На наиболее приоритетном уровне выполняются обработчики прерываний от внешних устройств системы. Этот уровень распадается на несколько отдельных подуровней, каждому из которых соответствуют аппаратные прерывания, инициируемые определенными внешними устройствами. Организация обработчиков прерываний, которые выполняют действия, наиболее критичные по времени, здесь не отличается от общепринятой. Время нахождения в обработчике прерывания ограничено, и в нем допустимо выполнять только те действия, которые следует выполнить немедленно.

Отличия от общеизвестных встроенных систем возникают на промежуточном уровне приоритета, лежащем между уровнем аппаратных прерываний и уровнем *main*(). Промежуточный уровень соответствует программным прерываниям и предназначен для работы системы событий. Для тех архитектур, где отдельных программных прерываний не существует, может применяться низкоприоритетное аппаратное прерывание, выделенное для отсутствующего устройства и возбуждаемое через установку бита запроса в контроллере прерываний.

Во встроенных системах нет возможности задействовать все выразительные средства языка C++ из-за ограничений по использованию динамического выделения памяти, а также по скорости выполнения. При разработке DORSECC мы ограничились использованием механизма классов с небольшими дополнениями.

3.1. Система событий

Система событий реализована внутри базового класса события. Для каждого вида событий создается отдельный класс, который наследует базовому классу. Экземпляр отдельного класса события соответствует событию этого вида, произошедшему в конкретный момент и с конкретными параметрами. Например, события периодического таймера реализуются отдельным классом, причем каждый экземпляр этого класса соответствует определенному импульсу таймера и содержит номер импульса в качестве параметра.

Обращения к системе событий выполнены как методы экземпляра класса события. Источник события может находиться на любом уровне приоритета. Он создает экземпляр события *Event* с помощью *new*, при необходимости наполняет его параметрами, и передает на обработку вызовом функции *pEvent->Post*(). Эта функция помещает событие *Event* в очередь системы событий (см. рис. 1).

В простейшем случае очередь представляет собой обычную структуру FIFO. Вызов *Post*() может происходить из кода, исполняемого на любом уровне приоритета. Обработка события *Event* состоит в извлечении его из очереди и вызове обработчика *pEvent->Exec*(), индивидуально определенного для каждого отдельного класса события. Когда очередь событий не пуста, и никакое другое событие не обрабатывается, то диспетчер событий запускает на обработку первое по порядку событие в очереди. Весь код обработки события выполняется на промежуточном уровне.

Более конкретно, событие помещается в очередь с помощью *pEvent->Post*(). Если очередь до этого момента была пустой, то также возбуждается программное прерывание. В этом программном прерывании запускается диспетчер событий, который извлекает событие из очереди и вызывает его обработчик *pEvent->Exec*(). Если, после завершения обработчика, в очереди еще остаются другие события, то они обрабатываются одно за другим. Как только очередь оказывается пустой, диспетчер событий прекращает работу и программное прерывание завершается.

Функция обработчика возвращает статус события, который определяет его дальнейшую судьбу. Обычно, обработанное событие уничтожается (статус *done* на рис. 1). Однако, для специальных целей, как будет описано далее, событие может быть сохранено и отправлено на обработку повторно.

Для обеспечения требований реального времени, при создании и уничтожении события не

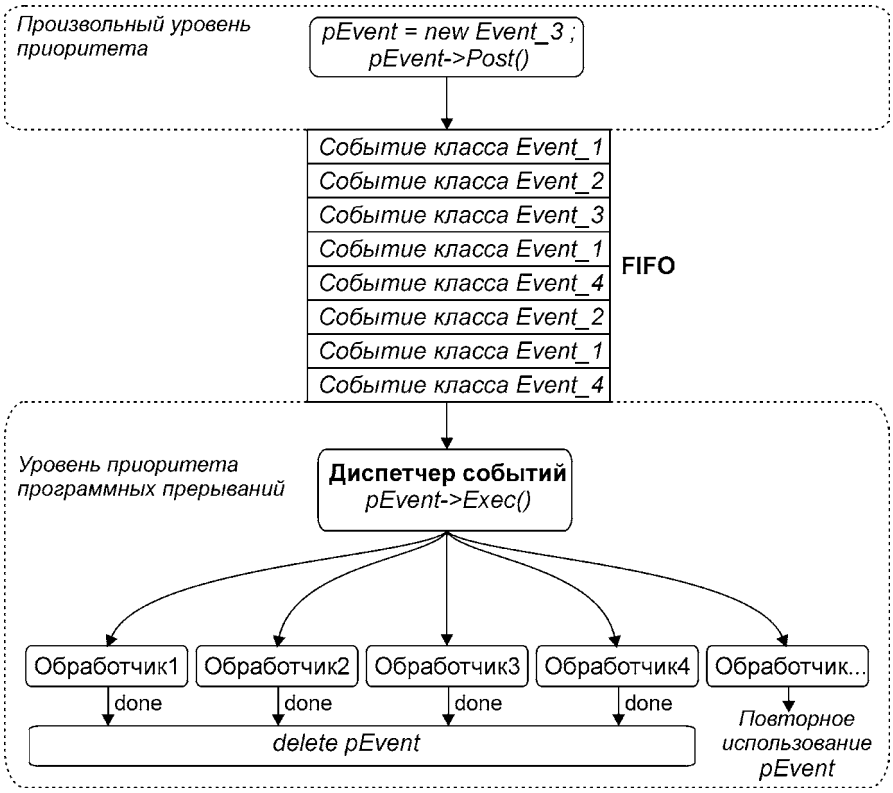

Рис. 1. Система событий. Показана отправка события класса *Event_3* в очередь FIFO (без заполнения события параметрами).

происходит обращения к менеджеру памяти. Вместо этого, заранее создается пул событий, которые не уничтожаются, а только помечаются как неиспользуемые. Затем, они повторно используются при создании события. Для такого решения в C++ имеются средства в виде контейнеров, а также возможности переопределения операторов *new* и *delete*. Если требования реального времени не очень жесткие и менеджер памяти безопасно работает в прерываниях, то можно обойтись без заранее созданного пула. Событие может либо выделяться из числа ранее удаленных, либо создаваться заново в памяти, когда ранее удаленное событие не доступно. Такой вариант обеспечивает автоматическую подстройку использования памяти к реальным потребностям.

Обработка событий диспетчером прерывает выполнение кода уровня *main*(), поскольку приоритет программного прерывания более высокий. Обработка события должна быть ограниченной во времени, чтобы не блокировать обработку следующих событий в очереди. В то же время, она не может заблокировать реакцию на аппаратные прерывания, так как имеет более низкий приоритет. Таким образом, в отличие от аппаратных прерываний, обеспечивающих режим жесткого реального времени, система событий работает в мягком режиме реального времени.

Одна из основных функций системы событий относится к вычислительно сложной обработке аппаратного события. Обработчик аппаратного прерывания быстро реагирует на запрос от периферийного устройства и производит немедленную обработку, которую нельзя отложить. Однако, если требуются дополнительные сложные действия, которые недопустимо выполнять в обработчике аппаратного прерывания по соображениям большой продолжительности, то такие действия выполняет система событий. Для этого, обработчик аппаратного прерывания создает событие по *new* и помещает его в очередь с помощью *pEvent->Post*(). Диспетчер событий далее запускает обработку события *pEvent->Exec*() на промежуточном уровне приоритета, не блокируя аппаратные прерывания.

Система событий позволяет реализовать переходы конечного автомата по событиям. При обработке события выполняется перевод автомата в следующее состояние, соответствующее графу переходов. Источником такого события может быть как низкоприоритетный код уровня *main*() (например, в результате действий в интерфейсе пользователя), так и высокоприоритетное аппаратное прерывание. Кроме того, событие может быть отправлено с промежуточного уровня, то есть, из самой системы событий.

3.2. Сопрограммы

Для реализации бесстековых сопрограмм нами была выбрана концепция протопотоков (protothreads) Дункельса [2], основанная на методе Даффа (Duff's device) и работе Тэтема [20]. Она относится к наиболее часто применяемому способу реализации сопрограмм, в котором используются так называемые возобновляемые функции (resumable functions, подробную дискуссию о них в контексте C++ см. в [7]). Сопрограмма представляет собой последовательность операторов, показанную на рис. 2. С формальной точки зрения, она есть тело функции, в которой необходимые действия сопрограммы реализованы последовательностью операторов Op.1 – Op.12. Служебные участки кода BEGIN, YIELD и END реализованы с помощью макроопределений. Вся сопрограмма целиком выполняется за несколько последовательных вызовов этой функции. Благодаря такому поведению функция получила название возобновляемой. При каждом вызове выполняется только часть операторов, после чего происходит выход из возобновляемой функции.

Начальный участок кода BEGIN представляет собой локальный диспетчер, который направляет выполнение кода на продолжение сопрограммы с того места, где оно завершилось при предшествующем вызове возобновляемой функции. Служебный участок YIELD приостанавливает

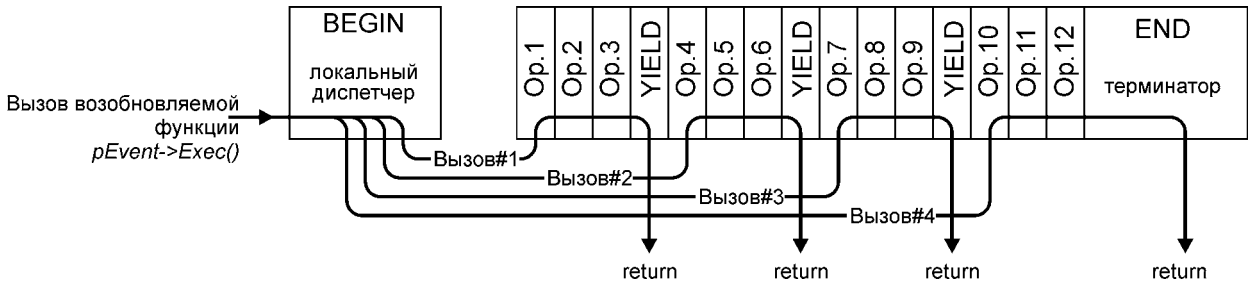

Рис. 2. Последовательность операторов возобновляемой функции сопрограммы.

выполнение сопрограммы, завершая текущий вызов возобновляемой функции. При первом вызове выполняются операторы Op.1 – Op.3, при следующем Op.4 – Op.6, и так далее. Выполнение сопрограммы оканчивается терминатором END, в котором состояние сопрограммы помечается как завершенное. При необходимости повторного исполнения в терминаторе может быть реализован возврат к началу сопрограммы.

Служебные участки кода используют скрытые переменные состояния, которые нужны для сохранения точки возврата в сопрограмму при следующем вызове возобновляемой функции, а также текущего статуса выполнения сопрограммы. Эти участки построены на основе оператора *switch*, где все точки возобновления выполнения сопрограммы отмечены с помощью *case*. Однако конструкция *switch – case* скрыта внутри макроопределений и не видна программисту.

Макрос YIELD, как это изначально было определено Дункельсом, дает возможность выполниться другим сопрограммам. Помимо YIELD, существуют дополнительные макроопределения служебных участков кода, которые могут быть использованы для проверки определенного условия, и остановки выполнения сопрограммы в случае, когда условие не выполнено. Например, макрос WAIT_UNTIL(*condition*) проверяет условие *condition* и в случае его невыполнения приостанавливает выполнение сопрограммы. При этом точка возобновления устанавливается на начало макроса. После выполнения условия *condition* при очередном вызове возобновляемой функции, выполнение сопрограммы продвигается дальше по коду.

Из сопрограммы при помощи макроса SPAWN может быть запущена дочерняя сопрограмма. Такой запуск является синхронным и останавливает выполнение основной сопрограммы до завершения дочерней.

На протяжении всего выполнения сопрограммы, только статически определенные переменные и объекты, выделенные в куче (heap), могут сохранить свои значения. Стековые переменные допустимо использовать исключительно между двумя служебными участками кода, поскольку их значение будет утеряно при следующем вызове функции сопрограммы. Результат выполнения сопрограммы должен, в том или ином виде, сохраняться в куче. Благодаря невозможности полноценно применять стековые переменные, возобновляемая функция не реентерабельна. Эти ограничения можно считать естественным для всех видов бесстековых сопрограмм.

Подобная реализация сопрограмм привносит очень малые накладные расходы процессорного времени и памяти. Это было убедительно показано при разработке ОС реального времени Contiki [21] и сетевого стека uIP [22, 23], применяемых во встроенных системах с весьма ограниченными ресурсами. В частности, высокое быстродействие локального диспетчера достигается за счет табличной реализацией переходов, в которую в ходе компиляции переводится конструкция *switch – case*.

3.3. Сопрограммы как события

Исходная реализация Дункельса [2] имеет ряд существенных недостатков, которые касаются способа вызова функции сопрограммы. В этой реализации, вызов возобновляемых функций всех объявленных сопрограмм последовательно делался из карусели (round robin) суперцикла, работающего на низкоприоритетном уровне *main*(). Суперцикл реальных встроенных систем обычно включает в себя множество действий, выполняемых без привлечения сопрограмм, не относящихся к категории реального времени и снижающих общую частоту повторения. Из-за этого, для сопрограмм было достаточно сложно обеспечить высокую скорость реакции. Кроме того, ожидание условия по типу WAIT_UNTIL(*condition*), хотя и не блокирует работу других сопрограмм, но является активным и потому непродуктивно расходует время процессора на постоянные проверки условия *condition*. Благодаря этому, синхронизации сопрограмм сопряжена со значительными потерями времени процессора.

Нами был предложен иной подход к вызову возобновляемой функции сопрограммы, который полностью свободен от указанных недостатков. Он основан на нескольких принципах, перечисленных ниже.

Класс сопрограммы наследует от класса события, так что каждая сопрограмма представляет собой частный случай события. Возобновляемая функция сопрограммы оформляется как обработчик события *pEvent->Exec*(). Поскольку возобновляемая функция сопрограммы не реентерабельна и не имеет смысла вне соответствующего ей события, то предпочтительно создавать ее, вместе с экземпляром события, как анонимную функцию (лямбда-выражение в C++ 11).

Запуск сопрограммы производится ее отправкой в очередь системы событий с помощью *pEvent->Post*(). После извлечения сопрограммы из очереди, ее возобновляемая функция *pEvent->Exec*() вызывается диспетчером системы событий.

После выхода из *pEvent->Exec*() по выполнению макроса YIELD, событие сопрограммы не уничтожается, но заново отправляется в очередь системы событий. Таким образом, после каждого YIELD возобновляемая функция повторно вызывается системой событий, пока исполнение не дойдет до конца сопрограммы, отмеченного END.

При выполнении YIELD проверяется состояние очереди системы событий. Если она оказывается пустой, то выход из возобновляемой функции не выполняется и исполнение *pEvent->Exec*() продолжается до следующего служебного макроса. Благодаря такой проверке, приостановка сопрограммы производится только тогда, когда это реально требуется, а накладные затраты системы событий снижаются.

Когда сопрограмму необходимо остановить в ожидании какого-либо условия, а затем продолжить ее выполнение по наступлению этого условия, то используется механизм возобновления. Как только внешний код обнаруживает наступление ожидаемого условия, он возобновляет работу сопрограммы, инициируя отправку ее события в очередь.

При выполнении этих принципов, система событий становится одновременно планировщиком и диспетчером сопрограмм. В то же время, полностью сохраняется возможность обработки простых событий, обработчик которых является обычной непрерывной функцией. Простые события и события сопрограммы стоят в общей очереди и единообразно обрабатываются диспетчером событий. Когда работают несколько сопрограмм, то в очереди присутствует смесь событий этих сопрограмм и простых событий, а их последовательная обработка обеспечивает параллельное выполнение сопрограмм.

Грань между простым событием и событием сопрограммы оказывается достаточно размытой. Давно отмечено [24], что подпрограмма представляет собой частный случай сопрограммы. А именно, подпрограмма – это сопрограмма, которая не приостанавливается в ходе выполнения. Аналогичным образом, обработчик простого события можно рассматривать как частный случай сопрограммы, которая не выполняет YIELD.

Если определенное простое событие возникает достаточно часто, то в очереди системы событий одновременно может находиться несколько экземпляров простых событий одного класса, возможно, отличающихся параметрами. Вызов возобновляемой функции сопрограммы также является повторяющимся, но в очереди системы событий в каждый момент времени может быть не более одного события, соответствующего данной сопрограмме. Разница тут в том, что появление экземпляров одного и того же простого события определяется внешними асинхронными действиями, а сопрограмма самостоятельно и синхронным образом планирует исполнение самой себя (см. рис. 3).

При развитии функциональности программного кода может выясниться, что обработка некоторого простого события стала выполняться недопустимо долго и блокирует обработку других событий на слишком большое время. Тогда, это простое событие можно превратить в сопрограмму, сделав функцию обработчика возобновляемой и вставив в ее код один или несколько YIELD. За счет такой вставки общее время обработки события несколько увеличится, но время блокировки им обработки других событий кратно уменьшится.

Для работы на самом низком уровне приоритета, была сохранена возможность вызова возобновляемых функций в суперцикле *main*(), в соответствии с первоначальным подходом Дункельса. Они не обрабатываются в системе событий и не работают в реальном времени, но позволяют структурировать параллельное выполнение медленных задач суперцикла.

3.4. Синхронизация параллельного выполнения

Представленный нами механизм остановки и возобновления сопрограммы реализует пассивное ожидание, не расходующее процессорное время. Он полностью совместим со всеми классическими способами синхронизации, включая семафоры, мониторы, каналы передачи сообщений, мьютексы, а также ожидание готовности объекта. Указатель на событие сопрограммы при остановке и возобновлении используется совершенно аналогично идентификатору (handle) процесса или потока в традиционных операционных системах.

Событие, обработка которого завершилась, считается готовым. Это относится, в том числе, и к завершению сопрограммы. Статус завершения сопрограммы можно получить опросом *pEvent->GetState*(). Такой способ наиболее пригоден для проверки состояния в суперцикле *main*(), в том числе, при активном ожидании. Когда используется управление встроенной системой с помощью конечного автомата, то естественным результатом обработки события и подтверждением ее завершения является изменение состояния автомата самим кодом обработчика.

Кроме того, предусмотрена подача объекту синхронизации сигнала о наступлении готовности события. Для этого, соответствующий объ-

Рис. 3. Обработка простых событий и сопрограмм в системе событий. *Event_XX* может быть классом как простого события, так и сопрограммы.

ект синхронизации должен быть предварительно подписан на сигнал от события *Event*, то есть указатель на него должен быть занесен в поле события *pEvent->SignalOnReady*. Использование функционального обратного вызова (callback) в запускающий код могло бы показаться более удобным способом извещения о готовности события, но оно не рекомендуется нами по соображениям безопасности. Дело в том, что распространение передачи управления при обратном вызове плохо контролируется средствами верификации кода. Обратный вызов исполняется на промежуточном уровне приоритета и может продлиться слишком долго, блокируя другие события.

В реализации Дункельса имеется только синхронный блокирующий запуск дочерней сопрограммы SPAWN, при котором возобновляемая функция родительской сопрограммы вызывает возобновляемую функцию дочерней. Предлагаемый нами подход позволяет асинхронно запускать дочернее событие при помощи макроса FORK, который просто отправляет дочернее событие в очередь на обработку, не останавливая родительской сопрограммы.

В связи с малыми затратами памяти и времени на порождение даже сравнительно большого количества событий, появляется возможность простой реализации методологии fork/join [25]. Для выполнения операции join, то есть ожидания завершения ранее запущенных событий, нами используется специальный объект синхронизации *Joint* с методом *Signal*(). В объекте хранятся две переменные, задаваемые извне: счетчик ожидаемых событий *waited* и указатель на событие продолжения *pContinuation*. Первоначально, в счетчик заносится известное количество ожидаемых событий. Объект *Joint* подписывается на сигналы от тех событий, завершения обработки которых ему необходимо дождаться. По каждому вызову *Signal*() происходит декремент *waited*. Как только *waited* достигнет нуля, что обозначает завершение обработки всех ожидаемых событий, *Joint* запускает событие продолжения по указателю *pContinuation*. В описанной схеме fork/join

простые события и сопрограммы могут сочетаться произвольным образом.

Для родительской сопрограммы возможно организовать пассивное ожидание завершения join. Она должна записать саму себя в качестве *pContinuation* и остановиться после запуска дочерних событий. Когда обработка всех запущенных событий завершится, сопрограмма будет возобновлена.

В класс *cJoint* был добавлен метод *Fork*(*pEvent*), который подписывает объект *Joint* на дочернее событие *Event,* инкрементирует счетчик *waited* и отправляет дочернее событие в очередь при помощи *pEvent->Post*(). С помощью *Fork* полностью автоматизируется запуск дочерних событий и подготавливается ожидание их завершения.

Нужно отметить, что описанные средства управления параллельным исполнением сходны с конструкциями языков Go и occam [26, 27], основанных на теории взаимодействующих последовательных процессов (CSP) Хоара [28]. Так, FORK действует аналогично оператору *go* языка Go, а базовый вариант *Joint* близок к стандартному примитиву синхронизации *sync.WaitGroup* этого языка. Использование *Fork*(*pEvent*) позволяет получить семантику, аналогичную оператору параллельного исполнения *PAR* в языке occam.

Как можно видеть, *Joint* через поле *pContinuation* связан с сопрограммой, в коде которой он используется для ожидания. Данный объект синхронизации естественно включить в объект сопрограммы, что и было выполнено в DORSECC. В результате, сама сопрограмма становится объектом синхронизации. Когда сопрограмма *Event* остановилась в ожидании сигнала и *waited*=1, то обращение *pEvent->Signal*() имеет тот же результат, что и *pEvent->Post*(). Если же сопрограмма не остановлена в ожидании, то поданный ей сигнал просто теряется. Таким образом, сопрограмма может ожидать готовности не только других событий, но также и любых объектов, которые могут отправить ей сигнал при входе в ожидаемое состояние. Подобный способ синхронизации можно считать безопасной заменой обратного вызова функции, который очень широко, но без должной дисциплины используется в программировании встроенных систем.

Рассмотрим пример, когда макрос WAIT_UNTIL(*condition*) в возобновляемой функции обнаруживает невыполнение условия и останавливает сопрограмму в ожидании сигнала. После прихода сигнала, возобновление происходит с первого оператора в макросе WAIT_UNTIL(*condition*), так что сразу же выполняется повторная проверка условия *condition*. Если условие не выполнено, то сопрограмма вновь останавливается. Благодаря этой проверке, сигнал может подаваться как знак не только гарантированного, но и даже лишь вероятного выполнения *condition* (например, при освобождении ресурса, за который конкурируют несколько сопрограмм). Кроме того, дополнительная проверка позволяет учесть изменение условия, если оно произошло уже после подачи сигнала. Этот подход известен как монитор типа Mesa [29].

Макрос JOINT, используемый для ожидания завершения всех запущенных дочерних событий, эквивалентен WAIT_UNTIL(*this->waited* <= 0). Он позволяет корректно обработать ситуацию быстрого завершения дочерних событий, когда родительскую сопрограмму останавливать уже не нужно.

Использование сопрограммы в качестве объекта синхронизации не отменяет, но дополняет ранее перечисленные классические способы, в которых примитив синхронизации существует независимо и хранит в себе указатель на ожидающую сопрограмму для ее прямого возобновления по *pEvent->Post*().

3.5. Приоритеты и вытеснение

При необходимости, в DORSECC может быть разрешена поддержка приоритета события. Для обработки событий с учетом приоритета, система событий клонируется на несколько промежуточных уровней приоритета прерываний, так чтобы каждому уровню приоритета соответствовала своя очередь FIFO и свой диспетчер событий. В этом случае, приоритет обработки события определяется уровнем приоритета программного прерывания, на котором он выполняется. В типовом варианте, установлены два уровня приоритетов: нормальный и высокий.

Отправка на обработку высокоприоритетного события инициирует программное прерывание с высоким приоритетом, и таким образом прерывает обработку менее приоритетного события. Благодаря этому, становится возможным немедленное вытеснение одной сопрограммы более приоритетной. Такое поведение невозможно в классической схеме приоритетов сопрограмм [9, 10], если только для ее реализации не используются приоритетные потоки ОС.

Латентность начала обработки уединенной сопрограммы, как и всякого события, описывается формулой (3). Наличие других событий в очереди увеличивает латентность в соответствии со временем обработки этих событий. Однако, когда в DORSECC используются приоритеты событий, то для наиболее приоритетных событий, всегда

обеспечивается минимальная латентность согласно (3), если только в очереди нет других событий с тем же приоритетом.

Указанный эффект аналогичен вытеснению исполняемого потока ОС более приоритетным, но реализуется без использования ОС и потоков. Поскольку вход в прерывание всегда выполняется быстрее переключения контекста ($T_{IL} \ll T_{TSW}$), то DORSECC обеспечивает меньшую латентность не только в сравнении с классическими сопрограммами согласно (2), но также и с вытесняющими потоками ОС в соответствии с (1).

Сопрограмма с высоким приоритетом является мощным инструментом реального времени, но при этом несет с собой опасность блокировки системы событий. Пока такая сопрограмма выполняется, все события с более низким приоритетом остаются заблокированными. Поэтому, сопрограмма с высоким приоритетом должна быстро завершаться либо останавливаться, что требует дополнительного внимания от программиста. Для устранения риска блокировки нами был реализован так называемый приоритет пробуждения (wakeup priority), который используется в дополнение к обычному приоритету события. При запуске или при возобновлении сопрограммы после остановки, ее событие помещается в очередь диспетчера, соответствующего приоритету пробуждения. Когда в сопрограмме выполняется YIELD, то ее событие перемещается в очередь диспетчера обычного уровня приоритета. Таким образом, сопрограмма начинает или возобновляет работу на уровне приоритета пробуждения, но после первого же вызова YIELD переходит на обычный уровень.

В качестве примера рассмотрим сопрограмму, для которой важна малая латентность запуска в ответ на внешний асинхронный стимул, но которая не должна блокировать выполнение сопрограмм с нормальным уровнем приоритета. Для нее устанавливается нормальный уровень обычного приоритета и высокий уровень приоритета пробуждения. При возникновении внешнего стимула, событие сопрограммы помещается в очередь диспетчера высокого приоритета, и вытесняет выполняемый в этот момент обработчик простого события или сопрограмму с нормальным приоритетом. Однако, как только выполнение сопрограммы на высоком уровне приоритета дойдет до первого YIELD, она перемещается в очередь диспетчера нормального уровня приоритета, и продолжает выполняться на этом уровне до завершения или остановки. Таким образом, достигается малая латентность реакции сопрограммы согласно формуле (3), но блокировка обработки на нормальном уровне приоритета не возникает. Конечно же, сохраняется потенциальный риск блокировки, если программист вовсе не будет использовать YIELD, но такой риск присущ всем системам с кооперативным параллельным исполнением.

3.6. Мультипроцессорная обработка

Предлагаемая методология может быть распространена на случай симметричной мультипроцессорной системы. В самом простом варианте, если очередь событий не пуста, и хотя бы на одном ядре процессора никакое событие не обрабатывается, то диспетчер событий запускает на обработку этим ядром первое по порядку событие в очереди. Это обеспечивает равномерную загрузку промежуточного уровня приоритета для всех ядер процессора.

В мультипроцессорной системе должны быть предусмотрены средства для обеспечения аффинности между ядром процессора и определенным событием. В отношении сопрограмм, аффинность к ядру аналогична хорошо изученному понятию аффинности потока и ядра в операционных системах, например, см. [30, 31]. За счет обеспечения аффинности между ядром процессора и сопрограммой снижаются затраты времени и энергии на поддержание когерентности кэша ядер. Когда речь идет об оптимизации работы кэша, то аффинность носит для диспетчера мягкий, рекомендательный характер, поскольку ее нарушение не приводит к нарушению логики работы программы.

С другой стороны, соблюдение аффинности между ядром процессора и определенным классом простого события может быть строго необходимым для поддержания последовательности обработки событий в том порядке, как они поступали в систему событий. Если несколько экземпляров события одного класса непосредственно следуют в очереди друг за другом, то, без соблюдения аффинности, их обработка может почти одновременно начаться на нескольких ядрах. В некоторых случаях, такое нарушение очередности (race condition) может приводить к разрушению логики обработки событий определенного класса. Жестко заданная афинность событий этого класса к конкретному ядру позволяет сохранить неизменной последовательность обработки.

4. ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ

Библиотека DORSECC была использована при создании самодостаточных программ встроенных систем для трех различных платформ: ARM9, Blackfin и ARM Cortex V7-A. С помощью

этих систем, решалась техническая задача высокоскоростной сортировки банкнот. Были разработаны два вида взаимодействующих встроенных систем: контроллер механизма сортировки и модуль проверки банкнот.

Первая из этих систем в реальном времени осуществляла управление датчиками и приводными устройствами механизма, а также поддерживала графический интерфейс пользователя и работу в сети TCP/IP. Вначале она была реализована на процессоре ARM926 с частотой ядра 400 МГц и объемом оперативной памяти 128 МБ (применялся компилятор IAR с максимальной оптимизацией по скорости). Задача управления в реальном времени решалась с помощью двух взаимосвязанных конечных автоматов, графы переходов которых суммарно имели более 90 состояний и были реализованы через систему событий. Наибольший поток событий составил около 12000 в секунду. Сопрограммы применялись для формирования отчетов о работе системы, автоматического регулирования в механизме, а также обеспечения печати и сетевой коммуникации. Во время работы механизма в наиболее нагруженном режиме процессор находился на промежуточном уровне приоритета примерно в течение 36% всего времени, из которых 9% приходилось на выполнение сопрограмм. Собственные накладные расходы времени системы событий (без учета выполнения самого обработчика) для первого события в очереди составляли до 1,5 мкс. Для второго и последующих событий, находящихся очереди, затраты на снижались до 0,5–1,0 мкс, поскольку для них не требовался выход из программного прерывания и повторный вход в него. Обработка подавляющего большинства событий занимала от 20 до 90 мкс, в среднем составляя около 30 мкс. В очереди обычно находилось от 0 до 2 событий, и в редких случаях это число увеличивалось до 5. Латентность обработки события, в среднем, составляла 10–12 мкс, но иногда достигала 220 мкс за счет накопления в очереди часто следующих событий.

Позднее, описанная система была перенесена на процессор ARM Cortex A7 с частотой ядра 1 ГГц (использовался компилятор GCC и уровень оптимизации –o3). Для этой платформы, все накладные расходы системы событий снизились до субмикросекундных величин.

Вторая система использовалась для мультиспектрального сканирования банкноты линейными датчиками изображения и последующего анализа полученных изображений. При распознавании образов применялся метод каскада классификаторов, реализованный с помощью специализированной виртуальной машины. Каскад классификаторов записывался в виде байт-кода, где каждому классификатору соответствовала отдельная инструкция виртуальной машины. Интерпретация байт-кода была реализована с помощью сопрограммы, где после выполнения каждой инструкции вызывался YIELD. В ходе анализа образа банкноты, независимо друг от друга и с перекрытием по времени работали до 3 виртуальных машин. Кроме того, с помощью простых событий, в ходе сканирования выполнялся поворот изображения для компенсации его перекоса. В целом, обработка простых событий занимала не более одной пятой части времени нахождения в программных прерываниях.

Система сканирования и анализа изображений была первоначально выполнена на двухъядерном процессоре Analog Devices Blackfin ADSP-BF607, работающем на частоте ядра 500 МГц (использовался компилятор CrossCore Studio с максимальной оптимизацией по скорости). Позднее, она была перенесена на ранее упомянутую платформу ARM Cortex A7 с частотой ядра 1 ГГц. В обоих вариантах, обработка событий выполнялась на одном из двух ядер процессора. Время исполнения инструкций байт-кода виртуальной машиной на платформе Cortex A7 лежало в пределах от единиц до десятков микросекунд, и только в редких случаях достигало нескольких сотен микросекунд. При пиковой нагрузке во время распознавания, аппаратные и программные прерывания занимали все время процессора. Такая нагрузка, в виде отдельных интервалов продолжительностью от 2 до 10 миллисекунд, в разных режимах отбирала 25–40% общего времени работы. Приведенные значения дают представление о гранулярности распределения процессорного времени, а также о весьма малой доле накладных расходов системы событий.

Мы выявили важный эффект, который можно назвать автоматической балансировкой нагрузки между сопрограммами и простыми событиями. Суть его сводится к следующему. Сопрограмма, выполняющая распознавание образа во второй системе, работает без остановки для ожидания и постоянно перепланирует продолжение своего выполнения при помощи YIELD. Таким образом, она занимает все доступное ей время процессора на промежуточном уровне приоритета прерываний. За счет этого и возникает пиковая нагрузка, при которой для уровня *main*() время не выделяется. Было обнаружено, что при пиковой нагрузке часто повторяющиеся простые события автоматически получают некоторый приоритет перед вычислительными сопрограммами.

Это можно объяснить следующим образом. Простые события обычно синхронизированы со временем или же запускаются извне встроенной системы (например, от срабатывания датчиков). Их частота, таким образом, есть данность, на которую количество событий в очереди не оказывает влияния. Напротив, частота возобновления работы сопрограммы после выполнения YIELD зависит от продолжительности обработки всех событий, находящихся в очереди, и падает при их большом количестве. Таким образом, когда времени для обработки простых событий не хватает, они накапливаются в очереди и снижают долю времени, выделяемую сопрограмме. В результате, сопрограмма получает процессорное время по остаточному принципу, после того, как оно расходуется для обслуживания потока поступающих простых событий.

Из проведенного рассмотрения можно сделать общие выводы о структуре накладных расходов времени на обработку события. Эти расходы складываются, главным образом, из суммарного времени T_{IP} входа в прерывание и завершения прерывания, а также суммарного времени T_{QUEUE} постановки в очередь и получения из очереди. При начале обработки уединенного простого события либо сопрограммы обязательно происходит программное прерывание, что соответствует накладным расходам времени на его обработку $T_{IP} + T_{QUEUE}$. Для простого события к этому добавляется время выполнения *new* и *delete*.

Однако, запуск на параллельное выполнение еще одной сопрограммы не требует вызова программного прерывания, поскольку это прерывание уже обрабатывается. То же относится к возобновлению любой сопрограммы после YIELD. Не требуется прерывание и для простых событий, которые либо возникают с интервалами, меньшими времени их обработки, либо обрабатываются в смеси с сопрограммами. Во всех перечисленных случаях накладные расходы определяются только T_{QUEUE}.

Наименьшие накладные расходы возникают при выполнении YIELD уединенно работающей сопрограммы. В этом случае, в системе событий лишь проверяется отсутствие событий в очереди, а расходы времени T_{IP} и T_{QUEUE} не возникают.

Важное следствие состоит в том, что накладные расходы на переключение между сопрограммами в предложенной методологии близки к расходам на переключение классических сопрограмм. Для классических сопрограмм переключение обычно осуществляется по принципу карусели (round robin), для реализации которой чаще всего применяется связанный кольцевой список сопрограмм. Эта структура данных похожа на структуру данных очереди, и вносит сопоставимые накладные расходы времени. Поскольку программные прерывания для переключения не требуются, T_{IP} в расходы на переключение между сопрограммами не включается.

По мере роста загрузки системы событий и роста заполнения очереди удельные накладные расходы на обработку одного события снижаются. Это происходит потому, что частота программных прерываний F_I и связанная с ней часть общих накладных расходов $F_I T_{IP}$ растут более медленно, чем полная частота событий F_E. Соответственно, уменьшается компонент $F_I T_{IP}/F_E$ удельных накладных расходов.

Обе описанные здесь встроенные системы были созданы для применения в счетно-сортировальных машинах DORS, выпускаемых крупносерийно. На момент написания статьи, их общее количество в эксплуатации превысило 20000 и продолжает увеличиваться.

Небольшие различия кода для реализации системы событий и сопрограмм на используемых платформах связаны с особенностями применяемых компиляторов, а также способом возбуждения программного прерывания. На платформе ARM9 это прерывание возбуждалось эмуляцией аппаратного прерывания от отсутствующего устройства с помощью специального бита запроса прерывания в контроллере прерывания AIC. Процессор Blackfin имеет отдельную инструкцию RAISE для возбуждения программного прерывания. На платформе ARM Cortex V7-A имеется развитая система команд контроллера прерываний GIC-400, которая позволяет гибко управлять возбуждением программных прерываний (SGI).

5. ОБСУЖДЕНИЕ И ВЫВОДЫ

Предложенная методология позволяет построить компактное ядро для обработки событий и выполнения бесстековых сопрограмм в самодостаточной программе встроенной системы реального времени. Обеспечивается малая латентность реакции на внешний стимул не только в сравнении с классическими сопрограммами, но также и с вытесняющими потоками ОС. Накладные расходы на переключение остаются низкими, как и у классических бесстековых сопрограмм.

Мы намеренно использовали слово “методология”, чтобы указать на возможность различных воплощений одного и того же замысла. Для реализации библиотеки DORSECC был выбран язык C++, хотя, при необходимости, можно было бы

ограничиться классическим ANSI C. Применение C++ скрывает детали внутренних механизмов системы событий и выполнения сопрограммы. Использованные нами и намеренно ограниченные выразительные средства языка C++ позволяют не только получить быстрый и оптимизированный ассемблерный код, но и достичь весьма высокого уровня абстракции.

Вместо протопотоков Дункельса, могут быть использованы практически любые виды сопрограмм, реализуемые в виде возобновляемой функции. Перечислим основные отличия нашего подхода от известных реализаций сопрограмм:

• представление сопрограммы как события, а ее возобновляемой функции в качестве обработчика этого события;

• обработку событий, в том числе сопрограмм, в контексте низкоприоритетного программного прерывания;

• обеспечение нескольких уровней приоритета событий и сопрограмм с помощью различных приоритетов программных прерываний для их обработки;

• выполнение YIELD путем повторной отправки события сопрограммы на обработку;

• возможность немедленного вытеснения одной сопрограммы более приоритетной сопрограммой или событием при асинхронном возникновении внешнего стимула, с малым риском блокировки системы событий.

Самодостаточная встроенная программа разделяется на три уровня приоритета исполнения кода. На самом высокоприоритетном уровне жесткого реального времени производится обработка аппаратных прерываний, с субмикросекундными задержками и длительностью на уровне единиц или десятков микросекунд. На промежуточном уровне мягкого реального времени, с характерными временами в десятки или сотни микросекунд, происходит обработка простых событий, а также выполняются сопрограммы. И, наконец, на самом низком уровне приоритета, соответствующего основной функции *main*(), выполняются постоянные действия с миллисекундными разрывами в исполнении, не требующие реального масштаба времени. Загрузка промежуточного уровня автоматически балансируется между простыми событиями и сопрограммами таким образом, что при необходимости обработки интенсивного потока простых событий, процессорное время для этого отбирается у сопрограмм.

Накладные расходы времени для обработки события, в наихудшем случае, заметно превосходят накладные расходы обработки прерывания. Однако, для большинства случаев обработки, эти расходы оказываются существенно меньшими. Как правило, они не превышают единиц процентов от длительности самих обработчиков событий.

Вызов возобновляемой функции в контексте программного прерывания решает вопрос о способе вызова, который был предметом активного обсуждения при выработке стандарта C++ 20 [7]. Когда сопрограмма представляет собой просто конструкцию языка программирования, то ее возобновляемую функцию поневоле приходится синхронно вызывать из контекста потока исполнения программы. В предлагаемом подходе, сопрограмма работает изолированно и асинхронно в контексте программного прерывания. Она запускается путем отправки в систему событий, но ее возобновляемая функция обычно вызывается вне запускающего контекста, то есть асинхронным образом по отношению к запускающему коду. С точки зрения аппаратуры, вызов возобновляемой функции инициируется контроллером прерывания, а не ядром процессора. Он не связан цепочкой вызовов с единственным потоком исполнения самодостаточной программы.

Дополнительное преимущество использования C++ проявилось в возможности динамического создания большого количества простых событий и сопрограмм при малых накладных затратах. Это важно для декомпозиции сложных алгоритмов по принципу fork/join, который пока еще редко используется во встроенных системах реального времени. Для управления параллельным исполнением могут применяться как классические способы синхронизации, так и средства, характерные для языков Go и occam. Поскольку средства и механизмы синхронизации следуют теории взаимодействующих последовательных процессов Хоара [28], то для верификации программ могут применяться формальные методы, основанные на этой теории.

Проведенная реализация сложных самодостаточных программ реального времени для массово выпускаемых устройств с использованием библиотеки DORSECC показала надежность предлагаемого подхода. Протопоток Дункельса, выбранный за основу сопрограммы, обладает неоспоримыми преимуществами: независимостью от типа компилятора и версии языка, высокой скоростью исполнения и малыми затратами памяти. Однако он накладывает ограничения на синтаксис возобновляемой функции и не предполагает автоматических средств проверки соблюдения этих ограничений. В будущем имеет смысл опробовать предлагаемую методологию с другими воплощениями сопрограмм, которые были бы более безопасны и удобны для программиста.

Предложенный нами механизм событий имеет явное сходство с механизмом *sender – executor – receiver* (отправитель – исполнитель – получатель), который планируется к введению в новый стандарт языка C++ [32]. *Executor* есть инструмент доступа к определенному контексту исполнения, который понимается очень широко: от потока программы до сопроцессора SIMD. Представляется важным опробовать абстракцию *executor* в применении к контексту исполнения в программном прерывании.

При внимательном рассмотрении можно заметить, что буфер FIFO системы событий в каждый момент времени содержит динамически генерируемую и исполняемую последовательность символов, известную как шитый код (threaded code). Вообще, символы шитого кода представляют собой единообразно оформленные ссылки на подпрограммы. В данном случае это указатели на объекты события, каждый со своей подпрограммой *Exec*(). Шитый код близок к понятию байт-кода и известен высокой скоростью исполнения, приближающейся к скорости исполнения нативного кода процессора. Понятие шитого кода было введено Беллом [33]. Затем, оно было развито Муром в языке FORTH [34], а также Верноком и Гешке в языке PostScript [35] для хранения тела процедуры в откомпилированной форме. Описанное нами порождение событий, а также планирование и диспетчеризацию их обработки можно рассматривать как особый случай машинной генерации программы в шитом коде для ее немедленного исполнения на выделенном уровне приоритета прерывания.

СПИСОК ЛИТЕРАТУРЫ

1. *Knuth D.* The Art of Computer Programming: Fundamental Algorithms. Addison-Wesley, Reading, Massachusetts, the USA. 1997. P. 193–200.
2. *Dunkels A., Schmidt O., Voigt T., Ali M.* Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006), Boulder, Colorado, the USA, November 2006.
3. Libtask: a Coroutine Library for C and Unix. https://swtch.com/libtask/. Accessed 21.03.2023.
4. The Boost C++ Libraries. Chapter 51: Boost.Coroutine. https://theboostcpplibraries.com/boost.coroutine. Accessed 21.03.2023.
5. The Boost C++ Libraries. Chapter 32: Boost.Asio. https://theboostcpplibraries.com/boost.asio. Accessed 21.03.2023.
6. CO2 Coroutine. https://github.com/jamboree/co2. Accessed 21.03.2023.
7. *Goodspeed N.* Stackful Coroutines and Stackless Resumable Functions. The C++ Standards Committee – ISOCPP Document N4232. December 13, 2014. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4232.pdf.
8. *Belson B., Holdsworth J., Xiang W., Philippa B.* A Survey of Asynchronous Programming Using Coroutines in the Internet of Things and Embedded Systems. ACM Transactions on Embedded Computer Systems. April 2019. V. 18. № 3. Article 21. https://doi.org/10.1145/3319618.
9. *Moskała M.* Kotlin Coroutines: Deep Dive. Kt. Academy, Warsaw, Poland. 2022.
10. FreeRTOS Kernel. Co-routines. https://www.freertos.org/croutine.html. Accessed 01.11.2023.
11. *Шалыто А.А.* Парадигма автоматного программирования. Научно-Технический вестник ИТМО. 2008. Выпуск 53: Автоматное программирование.
12. *Ousterhout J.* Why Threads are a Bad Idea (for most purposes). January 1996. USENIX Winter Technical Conference, San Diego, California, the USA.
13. *Lee E.* The Problem with Threads. Computer. 2006. V. 39. P. 33–42.
14. *Samek M.* Who Moved My State? Dr. Dobb's (online). April 1, 2003. https://drdobbs.com/who-moved-my-state/184401643
15. *Samek M.* Practical Statecharts in C/C++: Quantum Programming for Embedded Systems. CMP Books, San-Francisco, the USA. 2002.
16. *Gamma E., Helm R., Johnson R., Vlissides J.* Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts, the USA. 1995. P. 293.
17. Kernel-Mode Driver Architecture Design Guide: Introduction to DPC Objects. https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-dpc-objects. Accessed 21.03.2023.
18. Deferred Work – The Linux Kernel Documentation. https://linux-kernel-labs.github.io/refs/heads/master/labs/deferred_work.html. Accessed 21.03.2023.
19. *Nicholl R.A.* A Specification of Modula-2 Process (Coroutine) Management. Journal of Pascal, Ada & Modula-2. 1988. V. 7. № 5. P. 16–22.
20. *Tatham S.* Coroutines in C (online). https://www.chiark.greenend.org.uk/~sgtatham/coroutines.html. 2000.
21. *Dunkels A., Grönvall B., Voigt T.* Contiki – A Lightweight and Flexible Operating System for Tiny Networked Sensors. Proceedings of the Conference on Local Computer Networks. 2004. P. 455–462. https://doi.org/10.1109/LCN.2004.38
22. *Barnett D., Massa A.* Inside the uIP Stack. Dr Dobbs Journal (online). February 1, 2005. https://www.drdobbs.com/inside-the-uip-stack/184405971
23. *Dunkels A.* Programming Memory-Constrained Networked Embedded Systems. Swedish Institute of Computer Science Doctoral Thesis (online). SICS Dissertation Series 47. February 2007.

http://www.diva-potal.org/smash/get/diva2:1041306/FULLTEXT01.pdf

24. *Perlis A.* Epigrams on programming. ACM SIGPLAN Notices. September 1982. V. 17. № 9. P. 7–13. https://doi.org/10.1145/947955.1083808
25. *McCool M., Reinders J., Robison A.* Structured Parallel Programming: Patterns for Efficient Computation. Morgan Kaufmann, Burlington, Massachusetts, the USA. 2012. P. 209–252.
26. *Cox-Buday K.* Concurrency in Go. O'Reily Media, Sebastopol, California, the USA. 2017.
27. *Roscoe A., Hoare C.A.R.* The Laws of Occam Programming. Theoretical Computer Science. 1988. V. 60. P. 177–229. https://doi.org/10.1016/0304-3975(88)90049-7
28. *Hoare C.A.R.* Communicating sequential processes. Prentice-Hall, Englewood Cliffs, New Jersey, the USA. 1985.
29. *Lampson B.W., Redell D.D.* Experience with processes and monitors in Mesa. Comm. of the ACM. 1980. V. 23. № 2. P. 105–117.
30. *Love R.* CPU Affinity. Linux Journal (online). July 1, 2003. https://www.linuxjournal.com/article/6799
31. *Gujarati A., Cerqueira F., Brandenburg B.* Multiprocessor real-time scheduling with arbitrary processor affinities: from practice to theory. Real-Time Systems. 2015. V. 51. P. 440–483. https://doi.org/10.1007/s11241-014-9205-9
32. *Hoberock J., Garland M., Kohloff C. et al.* A Unified Executors Proposal for C++. The C++ Standards Committee – ISOCPP Document P0443R14. September 15, 2020. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0443r14.html
33. *Bell J.* Threaded code. Communications of the ACM. 1973. V. 16. № 6. P. 370–372. https://doi.org/10.1145/362248.362270.
34. *Rather E., Colburn D., Moore C.* The evolution of Forth. In: History of programming languages – II. ACM Other Books. 1 January 1996. P. 625–670. https://doi.org/10.1145/234286.1057832 ISBN 9780201895025
35. *Reid G.* Thinking in PostScript. Addison-Wesley, Reading, Massachusets, the USA. 1990. P. 105–118.

UNIFIED PROCESSING OF EVENTS AND COROUTINES IN EMBEDDED PROGRAM

© 2024 P. V. Minin[a]

[a] *DORS R&D LLC*

7–2, Federativny prospect, Moscow, 111399, Russia

A new architectural approach to real-time embedded programming is described. A bare-metal program is written in C/C++ and combines event-driven technique with concurrency based on co-routines. Events are processed by soft real-time core at the priority level of software generated interrupts. Event is first posted to input queue of the core and then processed by invocation of its event handler. A special case of event is co-routine, its resumable function being a co-routine event handler. The co-routine that either yielded or needs to be resumed is queued for event processing once again. As a result, it is processed multiple times until execution of resumable function comes to the end of its operator sequence. Different levels of processing priority may be assigned to an event. Soft real-time core could be further expanded to run on symmetrical multiprocessor hardware. A combination of co-routines and basic events could easily be used in fork/join model. Concurrency constructs resemble those of Go and occam languages. Virtually all classic types of synchronization primitives could be implemented. The new approach was implemented for various ARM and Blackfin processors in C++ language as portable DORSECC library. This library was further used to program real-time embedded systems for mass-produced banknote sorting machines. One type of systems was used to recognize and validate banknote images by the method of cascade of one-class classifiers. The other system worked as a motion controller and used finite automata to control sensors and actuators. The total number of systems in operation is currently over 20000. The event and co-routine core in these systems provides average event processing time in the range of dozens of microseconds with sub-microsecond overhead time per each event.

Keywords: real-time, embedded, bare metal, concurrency, event-driven, co-routine, C++, software interrupt, synchronization, threaded code

REFERENCES

1. *Knuth D.* The Art of Computer Programming: Fundamental Algorithms. Addison-Wesley, Reading, Massachusetts, the USA. 1997. P. 193–200.
2. *Dunkels A., Schmidt O., Voigt T., Ali M.* Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006), Boulder, Colorado, the USA, November 2006.
3. Libtask: a Coroutine Library for C and Unix. https://swtch.com/libtask/. Accessed 21.03.2023.

4. The Boost C++ Libraries. Chapter 51: Boost.Coroutine. https://theboostcpplibraries.com/boost.coroutine. Accessed 21.03.2023.
5. The Boost C++ Libraries. Chapter 32: Boost.Asio. https://theboostcpplibraries.com/boost.asio. Accessed 21.03.2023.
6. CO2 Coroutine. https://github.com/jamboree/co2. Accessed 21.03.2023.
7. *Goodspeed N.* Stackful Coroutines and Stackless Resumable Functions. The C++ Standards Committee – ISOCPP Document N4232. December 13, 2014. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4232.pdf.
8. *Belson B., Holdsworth J., Xiang W., Philippa B.* A Survey of Asynchronous Programming Using Coroutines in the Internet of Things and Embedded Systems. ACM Transactions on Embedded Computer Systems. April 2019. V. 18. № 3. Article 21. https://doi.org/10.1145/3319618.
9. *Moskała M.* Kotlin Coroutines: Deep Dive. Kt. Academy, Warsaw, Poland. 2022.
10. FreeRTOS Kernel. Co-routines. https://www.freertos.org/croutine.html. Accessed 01.11.2023.
11. *Shalyto, A.A.* Paradigm for automata based programming, in Nauchno-tekhnicheskii vestnik ITMO (ScientificTechnical Herald of ITMO Univ.). Issue 53: Avtomatnoe programmirovanie (Automata Based Programming), St. Petersburg, 2008.
12. *Ousterhout J.* Why Threads are a Bad Idea (for most purposes). January 1996. USENIX Winter Technical Conference, San Diego, California, the USA.
13. *Lee E.* The Problem with Threads. Computer. 2006. V. 39. P. 33–42.
14. *Samek M.* Who Moved My State? Dr. Dobb's (online). April 1, 2003. https://drdobbs.com/who-moved-my-state/184401643
15. *Samek M.* Practical Statecharts in C/C++: Quantum Programming for Embedded Systems. CMP Books, San-Francisco, the USA. 2002.
16. *Gamma E., Helm R., Johnson R., Vlissides J.* Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts, the USA. 1995. P. 293.
17. Kernel-Mode Driver Architecture Design Guide: Introduction to DPC Objects. https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-dpc-objects. Accessed 21.03.2023.
18. Deferred Work – The Linux Kernel Documentation. https://linux-kernel-labs.github.io/refs/heads/master/labs/deferred_work.html. Accessed 21.03.2023.
19. *Nicholl R.A.* A Specification of Modula-2 Process (Coroutine) Management. Journal of Pascal, Ada & Modula-2. 1988. V. 7. № 5. P. 16–22.
20. *Tatham S.* Coroutines in C (online). https://www.chiark.greenend.org.uk/~sgtatham/coroutines.html. 2000.
21. *Dunkels A., Grönvall B., Voigt T.* Contiki – A Lightweight and Flexible Operating System for Tiny Networked Sensors. Proceedings of the Conference on Local Computer Networks. 2004. P. 455–462. https://doi.org/10.1109/LCN.2004.38
22. *Barnett D., Massa A.* Inside the uIP Stack. Dr Dobbs Journal (online). February 1, 2005. https://www.drdobbs.com/inside-the-uip-stack/184405971
23. *Dunkels A.* Programming Memory-Constrained Networked Embedded Systems. Swedish Institute of Computer Science Doctoral Thesis (online). SICS Dissertation Series 47. February 2007. http://www.diva-potal.org/smash/get/diva2:1041306/FULLTEXT01.pdf
24. *Perlis A.* Epigrams on programming. ACM SIGPLAN Notices. September 1982. V. 17. № 9. P. 7–13. https://doi.org/10.1145/947955.1083808
25. *McCool M., Reinders J., Robison A.* Structured Parallel Programming: Patterns for Efficient Computation. Morgan Kaufmann, Burlington, Massachusetts, the USA. 2012. P. 209–252.
26. *Cox-Buday K.* Concurrency in Go. O'Reily Media, Sebastopol, California, the USA. 2017.
27. *Roscoe A., Hoare C.A.R.* The Laws of Occam Programming. Theoretical Computer Science. 1988. V. 60. P. 177–229. https://doi.org/10.1016/0304-3975(88)90049-7
28. *Hoare C.A.R.* Communicating sequential processes. Prentice-Hall, Englewood Cliffs, New Jersey, the USA. 1985.
29. *Lampson B.W., Redell D.D.* Experience with processes and monitors in Mesa. Comm. of the ACM. 1980. V. 23. № 2. P. 105–117.
30. *Love R.* CPU Affinity. Linux Journal (online). July 1, 2003. https://www.linuxjournal.com/article/6799
31. *Gujarati A., Cerqueira F., Brandenburg B.* Multiprocessor real-time scheduling with arbitrary processor affinities: from practice to theory. Real-Time Systems. 2015. V. 51. P. 440–483. https://doi.org/10.1007/s11241-014-9205-9
32. *Hoberock J., Garland M., Kohloff C. et al.* A Unified Executors Proposal for C++. The C++ Standards Committee – ISOCPP Document P0443R14. September 15, 2020. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0443r14.html
33. *Bell J.* Threaded code. Communications of the ACM. 1973. V. 16. № 6. P. 370–372. https://doi.org/10.1145/362248.362270.
34. *Rather E., Colburn D., Moore C.* The evolution of Forth. In: History of programming languages – II. ACM Other Books. 1 January 1996. P. 625–670. https://doi.org/10.1145/234286.1057832 ISBN 9780201895025
35. *Reid G.* Thinking in PostScript. Addison-Wesley, Reading, Massachusets, the USA. 1990. P. 105–118.