Операционные системы реального времени

И.Б. Бурдонов, А.С. Косачев, В.Н. Пономаренко

1. Введение: Особенности операционных систем реального времени

Операционные системы реального времени (ОСРВ) предназначены для обеспечения интерфейса к ресурсам критических по времени систем реального времени. Основной задачей в таких системах является своевременность (timeliness) выполнения обработки данных.

В качестве основного требования к ОСРВ выдвигается требование обеспечения **предсказуемости** или **детерминированности** поведения системы в наихудших внешних условиях, что резко отличается от требований к производительности и быстродействию универсальных ОС. Хорошая ОСРВ имеет предсказуемое поведение при всех сценариях системной загрузки (одновременные прерывания и выполнение потоков).

Существует некое различие между системами реального времени и встроенными системами. Не всегда от встроенной системы требуется, чтобы она имела предсказуемое поведение, и в таком случае она не является системой реального времени. Однако даже беглый взгляд на возможные встроенные системы позволяет утверждать, что большинство встроенных систем нуждается в предсказуемом поведении по крайней мере для некоторой функциональности, и таким образом, эти системы можно отнести к системам реального времени.

Принято различать системы мягкого (soft) и жесткого (hard) реального времени. В системах жесткого реального времени неспособность обеспечить реакцию на какие-либо события в заданное время ведет к отказам и невозможности выполнения поставленной задачи. В большинстве русскоязычной литературы такие системы называют системами с детерминированным временем. При практическом применении время реакции должно быть минимальным. Системами мягкого реального времени называются системы, не попадающие под определение "жесткие", т.к. в литературе четкого определения для них пока нет. Системы мягкого реального времени могут не успевать решать задачу, но это не приводит к отказу системы в целом. В системах реального времени необходимо введение некоторого директивного срока задачи (в англоязычной литературе – deadline), до которого задача должна обязательно (для систем мягкого реального времени –

желательно) выполниться. Этот директивный срок используется планировщиком задач как для назначения приоритета задачи при ее запуске, так и при выборе задачи на выполнение.

Мартин Тиммерман сформулировал следующие необходимые требования для ОСРВ [DEDSYS]:

- ОС должна быть многозадачной и допускающей вытеснение (pre-emptable),
- ОС должна обладать понятием приоритета для потоков,
- ОС должна поддерживать предсказуемые механизмы синхронизации,
- ОС должна обеспечивать механизм наследования приоритетов,
- поведение ОС должно быть известным и предсказуемым (задержки обработки прерываний, задержки переключения задач, задержки драйверов и т.д.), это значит, что должно быть определено максимальное время отклика во всех сценариях рабочей нагрузки системы.

В течение последних 25-30 лет структура операционных систем эволюционировала от монолитной к многослойной структуре ОС и далее к архитектуре клиент-сервер. При монолитной структуре ОС состоит из набора модулей, и изменения одного модуля влияют на другие модули. Чем больше модулей, тем больше хаоса при эксплуатации такой системы. Кроме того, невозможно распределить ОС в многопроцессорной системе. В многослойной структуре изменения одного слоя влияют на соседние слои, кроме того, обращение через слой невозможно. Для систем реального времени должно быть обеспечено прямое обращение к каждому слою ОС, а иногда напрямую к аппаратуре.

Основной идеей клиент-серверной технологии в ОС является сведение базиса ОС к минимуму (планировщик и примитивы синхронизации). Вся остальная функциональность выносится на другой уровень и реализуется через потоки или задачи. Совокупность таких серверных задач отвечает за системные вызовы. Приложения являются клиентами, которые запрашивают сервисы через системные вызовы.

Клиент-серверная технология позволяет создавать масштабируемые ОС и упрощает распределение в многопроцессорной системе. При эксплуатации системы замена одного модуля не вызывает эффекта “снежного кома”, кроме того, сбой модуля не всегда влечет за собой отказ системы в целом. Появилась возможность динамической загрузки и отгрузки модулей. Главной проблемой в этой модели является защита памяти, поскольку серверные процессы должны быть защищены. При каждом запросе сервиса система должна переключаться с контекста приложения на контекст сервера. При поддержке защиты памяти время переключения с одного процесса на другой увеличивается.

Как правило, большинство современных ОСРВ построены на основе микроядра (kernel или nucleus), которое обеспечивает планирование и диспетчеризацию задач, а также осуществляет их взаимодействие. Несмотря на

сведение к минимуму в ядре абстракций ОС, микроядро все же должно иметь представление об абстракции процесса. Все остальные концептуальные абстракции операционных систем вынесены за пределы ядра, вызываются по запросу и выполняются как приложения.

Рассмотрим концептуальные абстракции операционной системы через призму требований к системам реального времени.

1.1. Процессы, потоки, задачи

Концепция многозадачности (псевдо-параллелизм) является существенной для системы реального времени с одним процессором, приложения которой должны быть способны обрабатывать многочисленные внешние события, происходящие практически одновременно. Концепция процесса, пришедшая из мира UNIX, плохо реализуется в многозадачной системе, поскольку процесс имеет тяжелый контекст. Возникает понятие потока (thread), который понимается как подпроцесс или легковесный процесс (light-weight process). Потоки существуют в одном контексте процесса, поэтому переключение между потоками происходит очень быстро, а вопросов безопасности не существует. Потоки являются легковесными, потому что их регистровый контекст меньше, т.е. их управляющие блоки намного компактнее. Уменьшаются накладные расходы, вызванные сохранением и восстановлением управляющих блоков прерываемых потоков. Объем управляющих блоков зависит от конфигурации памяти. Если потоки выполняются в разных адресных пространствах, система должна поддерживать отображение памяти для каждого набора потоков.

Итак, в системах реального времени процесс распадается на задачи или потоки. В любом случае каждый процесс рассматривается как приложение. Между этими приложениями не должно быть слишком много взаимодействий, и в большинстве случаев они имеют различную природу – жесткого реального времени, мягкого реального времени, не реального времени.

1.2. Планирование, приоритеты

В связи с проблемой дедлайнов главной проблемой в ОСРВ становится планирование задач (scheduling), которое обеспечивало бы предсказуемое поведение системы при всех обстоятельствах. Процесс с дедлайнами должен стартовать и выполняться так, чтобы он не пропустил ни одного своего дедлайна. Если это невозможно, процесс должен быть отклонен.

В связи с проблемами планирования в ОСРВ изучаются и развиваются два подхода – статические алгоритмы планирования (RMS – Rate Monotonic Scheduling) [LL73] и динамические алгоритмы планирования (EDF – Earliest Deadline First).

RMS используется для формального доказательства условий предсказуемости системы. Для реализации этой теории необходимо планирование на основе приоритетов прерывающих обслуживание (preemptive priority scheduling). В теории RMS приоритет заранее назначается каждому процессу. Процессы должны удовлетворять следующим условиям:

- процесс должен быть завершен за время его периода,
- процессы не зависят друг от друга,
- каждому процессу требуется одинаковое процессорное время на каждом интервале,
- у непериодических процессов нет жестких сроков,
- прерывание процесса происходит за ограниченное время.

Процессы выполняются по приоритету. При планировании RMS предпочтение отдается задачам с самыми короткими периодами выполнения.

В EDF приоритет присваивается динамически, и наибольший приоритет выставляется процессу, у которого осталось наименьшее время выполнения. При больших загрузках системы EDF имеет преимущества перед RMS.

Во всех системах реального времени требуется политика планирования, управляемая дедлайнами (deadline-driven scheduling). Однако этот подход находится в стадии разработки.

Обычно в ОСРВ используется планирование с приоритетами, прерывающими обслуживание, которое основано на RMS. Приоритетное прерывание обслуживания (preemption) является неотъемлемой составляющей ОСРВ, т.к. в системе реального времени должны существовать гарантии того, что событие с высоким приоритетом будет обработано перед событием более низкого приоритета. Все это ведет к тому, что ОСРВ нуждается не только в механизме планирования на основе приоритетов, прерывающих обслуживание, но также и в соответствующем механизме управления прерываниями. Более того, ОСРВ должна быть способна запрещать прерывания, когда необходимо выполнить критический код, который нельзя прерывать. Длительность обработки прерываний должна быть сведена к минимуму.

ОСРВ должна обладать развитой системой приоритетов. Во-первых, потому, что система сама может рассматриваться как набор серверных приложений, подразделяющихся на потоки, и несколько высоких уровней приоритетов должно быть выделено системным процессам и потокам. Во-вторых, в сложных приложениях необходимо все потоки реального времени помещать на разные приоритетные уровни, а потоки не реального времени помещать на один уровень (ниже, чем любые потоки реального времени). При этом потоки не реального времени можно обрабатывать в режиме циклического планирования (RRS – round-robin scheduling), при котором каждому процессу предоставляется квант времени процессора, а когда квант заканчивается, его контекст сохраняется, и процесс ставится в конец очереди. Многие ОСРВ используют RRS для планирования задач на одном уровне. Приоритетный уровень 0 обычно используется для холостого режима.

При планировании на основе приоритетов необходимо решить две обязательные проблемы:

- обеспечить выполнение процесса с наивысшим приоритетом,

- не допустить инверсии приоритетов, когда задачи с высокими приоритетами ожидают ресурсы, захваченные задачами с более низкими приоритетами.

Для борьбы с инверсией приоритетов в ОСРВ часто используется механизм наследования приоритетов, однако при этом приходится отказываться от планирования на основе RMS, поскольку приоритеты становятся динамическими.

1.3. Память

Как уже упоминалось выше, задержка на переключение контекста потока напрямую зависит от конфигурации памяти, т.е. от модели защиты памяти. Рассмотрим четыре наиболее распространенных в ОСРВ модели защиты памяти.

- *Модель без защиты* – системное и пользовательское адресные пространства не защищены друг от друга, используется два сегмента памяти: для кода и для данных, при этом не требуется никакого управления памятью от системы, не требуется MMU (memory management unit – специальное аппаратное устройство для поддержки управления виртуальной памятью).
- *Модель защиты система/пользователь* – системное адресное пространство защищено от адресного пространства пользователя, системные и пользовательские процессы выполняются в общем виртуальном адресном пространстве, при этом требуется MMU. Защита обеспечивается страничным механизмом защиты. Различаются системные и пользовательские страницы. Пользовательские приложения никак не защищены друг от друга. Процессор находится в режиме супервизора, если текущий сегмент имеет уровень 0, 1 или 2. Если уровень сегмента – 3, тогда процессор находится в пользовательском режиме. В этой модели необходимы четыре сегмента – два сегмента на уровне 0 (для кода и данных) и два сегмента на уровне 3. Механизм страничной защиты не добавляет накладных расходов, т.к. защита проверяется одновременно с преобразованием адреса, которое выполняет MMU, при этом ОС не нуждается в управлении памятью.
- *Модель защиты пользователь/пользователь* – к *модели система/пользователь* добавляется защита между пользовательскими процессами, требуется MMU. Как и в предыдущей модели, используется механизм страничной защиты. Все страницы помечаются как привилегированные, за исключением страниц текущего процесса, которые помечаются как пользовательские. Таким образом, выполняющийся поток не может обратиться за пределы своего адресного пространства. ОС отвечает за обновление флага привилегированности для конкретной страницы в таблице страниц при переключении процесса. Как и в предыдущей модели используются четыре сегмента.

- *Модель защиты виртуальной памяти* – каждый процесс выполняется в своей собственной виртуальной памяти, требуется MMU. Каждый процесс имеет свои собственные сегменты и, следовательно, свою таблицу описателей. ОС несет ответственность за поддержку таблиц описателей. Адресуемое пространство может превышать размеры физической памяти, если используется страничная организация памяти совместно с подкачкой. Однако в системах реального времени подкачка обычно не применяется из-за ее непредсказуемости. Для решения этой проблемы доступная память разбивается на фиксированное число логических адресных пространств равного размера. Число одновременно выполняющихся процессов в системе становится ограниченным.

Фундаментальное требование для памяти в системе реального времени заключается в том, что время доступа к ней должно быть ограничено (или, другими словами, предсказуемо). Прямым следствием становится запрет на использование техники вызова страниц по запросу (подкачка с диска) для процессов реального времени. Поэтому системы, обеспечивающие механизм виртуальной памяти, должны уметь блокировать процесс в оперативной памяти, не допуская подкачки. Итак, подкачка недопустима в ОСРВ, потому что непредсказуема.

Если поддерживается страничная организация памяти (paging), соответствующее отображение страниц в физические адреса должно быть частью контекста процесса. Иначе опять появляется непредсказуемость, неприемлемая для ОСРВ.

Для процессов, не являющихся процессами жесткого реального времени, возможно использование механизма динамического распределения памяти, однако при этом ОСРВ должна поддерживать обработку таймаута на запрос памяти, т.е. ограничение на предсказуемое время ожидания.

В обычных ОС при использовании механизма сегментации памяти для борьбы с фрагментацией применяется процедура уплотнения после сборки мусора. Однако такой подход неприменим в среде реального времени, т.к. во время уплотнения перемещаемые задачи не могут выполняться, что ведет к непредсказуемости системы. В этом состоит основная проблема применимости объектно-ориентированного подхода к системам реального времени. До тех пор, пока проблема уплотнения не будет решена, C++ и JAVA останутся не самым лучшим выбором для систем жесткого реального времени.

В системах жесткого реального времени обычно применяется статическое распределение памяти. В системах мягкого реального времени возможно динамическое распределение памяти, без виртуальной памяти и без уплотнения.

1.4. Прерывания

При описании управления прерываниями обычно различают две процедуры, а именно:

- программа обработки прерывания (ISR – interrupt servicing routine) –

программа низкого уровня в ядре с ограниченными системными вызовами,

- поток обработки прерывания (IST – interrupt servicing thread) – поток уровня приложения, который управляет прерыванием, с доступом ко всем системным вызовам.

Обычно ISR реализуются производителем аппаратуры, а драйверы устройств выполняют управление прерываниями с помощью IST. Потоки обработки прерываний действуют как любые другие потоки и используют ту же самую систему приоритетов. Это означает, что проектировщик системы может придать IST более низкий приоритет, чем приоритет потока приложения.

1.5. Часы и таймеры

В ОСРВ используются различные службы времени. Операционная система отслеживает текущее время, в определенное время запускает задачи и потоки и приостанавливает их на определенные интервалы. Временные службы ОСРВ используют часы реального времени. Обычно используются высокоточные аппаратные часы. Для отсчета временных интервалов на основе часов реального времени создаются таймеры.

Для каждого процесса и потока определяются часы процессорного времени. На базе этих часов создаются таймеры; которые измеряют перерасход времени процессом или потоком, позволяя динамически выявлять программные ошибки или ошибки вычисления максимально возможного времени выполнения. в высоконадежных, критичных ко времени системах важно выявление ситуаций, при которых задача превышает максимально возможное время своего выполнения, т.к. при этом работа системы может выйти за рамки допустимого времени отклика. Часы времени выполнения позволяют выявить возникновение перерасхода времени и активизировать соответствующие действия по обработке ошибок.

Большинство ОСРВ оперируют относительным временем. Что-то происходит “до” и “после” некоторого другого события. В системе, полностью управляемой событиями, необходим часовой механизм (ticker), т.к. там нет квантования времени (time slicing). Однако, если нужны временные метки для некоторых событий или необходим системный вызов типа “ждать одну секунду”, тогда нужен тактовый генератор и/или таймер.

Синхронизация в ОСРВ осуществляется с помощью механизма блокирования (или ожидания) до наступления некоторого события. Абсолютное время не используется.

Реализации в ОСРВ других концептуальных абстракций подобны их реализациям в традиционных ОС.

1.6. Стандарты ОСРВ

Большие различия в спецификациях ОСРВ и огромное количество существующих микроконтроллеров выдвигают на передний план проблему стандартизации в области систем реального.

Наиболее ранним и распространенным стандартом ОСРВ является стандарт **POSIX** (IEEE Portable Operating System Interface for Computer Environments, IEEE 1003.1). Первоначальный вариант стандарта POSIX появился в 1990 г. и был предназначен для UNIX-систем, первые версии которых появились в 70-х годах прошлого века. Спецификации POSIX определяют стандартный механизм взаимодействия прикладной программы и операционной системы и в настоящее время включают набор более чем из 30 стандартов. Для ОСРВ наиболее важны семь из них (1003.1a, 1003.1b, 1003.1c, 1003.1d, 1003.1j, 1003.21, 1003.2h), но широкую поддержку в коммерческих ОС получили только три первых.

Несмотря на явно устаревшие положения стандарта **POSIX** и большую востребованность обновлений стандартизации для ОСРВ, заметного продвижения в этом направлении не наблюдается.

Некоторые наиболее успешные компании в области систем реального времени объявляют о своем решении принять в качестве стандарта спецификации одной из своих продвинутых ОСРВ. Так поступила TRON (the RTOS Nucleus), которая в 1987г. выпустила в свет первые ITRON спецификации – ITRON1. Далее в 1989г. она разработала и выпустила спецификации μITRON для 8- и 16- битовых микроконтроллеров, а также спецификации ITRON2 для 32-битовых процессоров. Более подробно ОСРВ ITRON описана в соответствующем разделе. Этот стандарт является очень распространенным в Японии. Спецификация ITRON описана в разделе операционных систем.

Военная и аэрокосмическая отрасли предъявляют жесткие требования к вычислительным средствам, влияющим на степень безопасности целевой системы. В настоящее время имеются следующие стандарты для ОСРВ в авиации – стандарт **DO-178B** и стандарт **ARINC-653**. Поскольку эти стандарты разработаны в США, стоит отметить еще европейский стандарт ED-12B, который является аналогом **DO-178B.**

Распространенным также является стандарт **OSEK/VDX** [OSEK], который первоначально развивался для систем автомобильной индустрии.

1.6.1. POSIX

Стандарт **POSIX** был создан как стандартный интерфейс сервисов операционных систем. Этот стандарт дает возможность создавать переносимые приложения. Впоследствии этот стандарт был расширен особенностями режима реального времени [POSIX].

Спецификации **POSIX** задают стандартный механизм взаимодействия приложения и ОС. Необходимо отметить, что стандарт **POSIX** тесно связан с ОС Unix, тем не менее, разработчики многих ОСРВ стараются выдержать соответствие этому стандарту. Соответствие стандарту **POSIX** для ОС и аппаратной платформы должно быть сертифицировано с помощью прогона на них тестовых наборов [POSIXTestSuite]. Однако если ОС не является Unix-подобной, выдержать это требование становится непростой задачей. Тестовые наборы существуют только для POSIX 1003.1a. Поскольку структура POSIX

является совокупностью необязательных возможностей, поставщики ОС могут реализовать только часть стандартного интерфейса, и при этом говорить о POSIX-комплиантности своей системы.

Несмотря на то, что стандарт POSIX вырос из Unix'а, он затрагивает основополагающие абстракции операционных систем, а расширения реального времени применимы ко всем ОСРВ.

К настоящему времени стандарт **POSIX** рассматривается как семейство родственных стандартов: IEEE Std 1003.n (где n – это номер).

Стандарт 1003.1a (OS Definition) содержит базовые интерфейсы ОС – поддержку единственного процесса, поддержку многих процессов, управление заданиями, сигналами, группами пользователей, файловой системой, файловыми атрибутами, управление файловыми устройствами, блокировками файлов, устройствами ввода/вывода, устройствами специального назначения, системными базами данных, каналами, очередями FIFO, а также поддержку языка С.

Стандарт 1003.1b (Realtime Extensions) содержит расширения реального времени – сигналы реального времени, планирование выполнения (с учетом приоритетов, циклическое планирование), таймеры, синхронный и асинхронный ввод/вывод, ввод/вывод с приоритетами, синхронизация файлов, блокировка памяти, разделяемая память, передача сообщений, семафоры. Для того, чтобы стать POSIX-комплиантной, ОС должна реализовать не менее 32 уровней приоритетов. POSIX определяет три политики планирования обработки процессов:

- SCHED_FIFO – процессы обрабатываются в режиме FIFO и выполняются до завершения,
- SCHED_RR – round robin – каждому процессу выделяется квант времени,
- SCHED_OTHER – произвольная реализационно-зависимая политика, которая не переносима на другие платформы.

Стандарт 1003.1c (Threads) касается функций поддержки многопоточной обработки внутри процесса – управление потоками, планирование с учетом приоритетов, мьютексы (специальные синхронизирующие объекты в межпроцессном взаимодействии, подающие сигнал, когда они не захвачены каким-либо потоком), приоритетное наследование в мьютексах, переменные состояния (condition variables).

Стандарт 1003.1d включает поддержку дополнительных расширений реального времени – семантика порождения новых процессов (spawn), спорадическое сер-верное планирование, мониторинг процессов и потоков времени выполнения, таймауты функций блокировки, управление устройствами и прерываниями.

Стандарт 1003.21 касается распределенных систем реального времени и включает функции поддержки распределенного взаимодействия, организации буферизации данных, посылки управляющих блоков, синхронных и асинхронных операций, ограниченной блокировки, приоритетов сообщений, меток сообщений, и реализаций протоколов.

Стандарт 1003.2h касается сервисов, отвечающих за работоспособность системы.

1.6.2. DO-178B

Стандарт **DO-178B**, создан Радиотехнической комиссией по аэронавтике (RTCA, Radio Technical Commission for Aeronautics) для разработки ПО бортовых авиационных систем [DO178B]. Первая его версия была принята в 1982 г., вторая (DO-178A) - в 1985-м, текущая DO-178B - в 1992 г. Готовится принятие новой версии, DO-178C. Стандартом предусмотрено пять уровней серьезности отказа, и для каждого из них определен набор требований к программному обеспечению, которые должны гарантировать работоспособность всей системы в целом при возникновении отказов данного уровня серьезности

Данный стандарт определяет следующие уровни сертификации:

- A (катастрофический),
- B (опасный),
- C (существенный),
- D (несущественный)
- E (не влияющий).

До тех пор пока все жесткие требования этого стандарта не будут выполнены, вычислительные системы, влияющие на безопасность, никогда не поднимутся в воздух.

1.6.3. ARINC-653

Стандарт **ARINC-653** (Avionics Application Software Standard Interface) разработан компанией ARINC в 1997 г. Этот стандарт определяет универсальный программный интерфейс APEX (Application/Executive) между ОС авиационного компьютера и прикладным ПО. Требования к интерфейсу между прикладным ПО и сервисами операционной системы определяются таким образом, чтобы разрешить прикладному ПО контролировать диспетчеризацию, связь и состояние внутренних обрабатываемых элементов. В 2003 г. принята новая редакция этого стандарта. ARINC-653 в качестве одного из основных требований для ОСРВ в авиации вводит архитектуру изолированных (partitioning) виртуальных машин.

1.6.4. OSEK

Стандарт **OSEK/VDX** является комбинацией стандартов, которые изначально разрабатывались в двух отдельных консорциумах, слившихся впоследствии. OSEK берет свое название от немецкого акронима консорциума, в состав которого входили ведущие немецкие производители автомобилей – BMW, Bosch, Daimler Benz (теперь Daimler Chrysler), Opel, Siemens, и Volkswagen, а также университет в Карлсруэ (Германия). Проект VDX (Vehicle Distributed eXecutive) развивался совместными усилиями французских компаний PSA и Renault. Команды OSEK и VDX слились в 1994г.

Первоначально проект **OSEK/VDX** предназначался для разработки стандарта открытой архитектуры ОС и стандарта API для систем, применяющихся в автомобильной промышленности. Однако разработанный стандарт получился более абстрактным и не ограничивается использованием только в автомобильной индустрии.

Стандарт **OSEK/VDX** состоит из трех частей – стандарт для операционной системы (OS), коммуникационный стандарт (COM), и стандарт для сетевого менеджера (NM). В дополнение к этим стандартам определяется некий реализационный язык (OIL). Первой компонентой стандарта **OSEK** является стандарт для ОС, поэтому часто стандарт **OSEK** ошибочно воспринимается как ОСРВ. Хотя ОС и есть большая порция данного стандарта, мощность его состоит в интеграции всех его компонент.

В данной работе рассматривается только стандарт для операционной системы, и его описание приводится в пункте **OSEK/VDX** в разделе операционных систем.

1.6.5. Стандарты безопасности

В связи со стандартами для ОСРВ стоит отметить широко известный стандарт критериев оценки пригодности компьютерных систем (Trusted Computer System Evaluation Criteria – TCSEC) [DoD85]. Этот стандарт разработан Министерством обороны США и известен также под названием "Оранжевая книга" (Orange Book – из-за цвета обложки).

Аналогичные критерии были разработаны в ряде других стран, на основе которых был разработан международный стандарт “Общие критерии оценки безопасности информационных технологий” (далее просто – Общие критерии) (Common Criteria for IT Security Evaluation, ISO/IEC 15408) [CC99].

В "Оранжевой книге" перечислены семь уровней защиты:

- A1 – верифицированная разработка. Этот уровень требует, чтобы методы формальной верификации гарантировали защиту секретной и другой критичной информации средствами управления безопасностью.
- B3 – домены безопасности. Этот уровень предназначен для защиты систем от опытных программистов.
- B2 – структурированная защита. В систему с этим уровнем защиты нельзя допустить проникновение хакеров.
- B1 – мандатный контроль доступа. Защиту этого уровня возможно и удастся преодолеть опытному хакеру, но никак не рядовым пользователям.
- C2 – дискреционный контроль доступа. Уровень C2 обеспечивает защиту процедур входа, позволяет производить контроль за событиями, имеющими отношение к безопасности, а также изолировать ресурсы.
- C1 – избирательная защита. Этот уровень дает пользователям возможность защитить личные данные или информацию о проекте,

установив средства управления доступом.

- D – минимальная защита. Этот нижний уровень защиты оставлен для систем, которые проходили тестирование, но не смогли удовлетворить требованиям более высокого класса.
- Что касается Общих критериев, то в них введены похожие требования обеспечения безопасности в виде оценочных уровней (Evaluation Assurance Levels – EAL). Их также семь:
- EAL7 – самый высокий уровень предполагает формальную верификацию модели объекта оценки. Он применим к системам очень высокого риска.
- EAL6 определяется, как полуформально верифицированный и протестированный. На уровне EAL6 реализация должна быть представлена в структурированном виде, анализ соответствия распространяется на проект нижнего уровня, проводится строгий анализ покрытия, анализ и тестирование небезопасных состояний.
- EAL5 определяется, как полуформально спроектированный и протестированный. Он предусматривает создание полуформальных функциональной спецификации и проекта высокого уровня с демонстрацией соответствия между ними, формальной модели политики безопасности, стандартизованной модели жизненного цикла, а также проведение анализа скрытых каналов.
- EAL4 определяется, как методически спроектированный, протестированный и пересмотренный. Он предполагает наличие автоматизации управления конфигурацией, полной спецификации интерфейсов, описательного проекта нижнего уровня, подмножества реализаций функций безопасности, неформальной модели политики безопасности, модели жизненного цикла, анализ валидации, независимый анализ уязвимостей. По всей вероятности, это самый высокий уровень, которого можно достичь на данном этапе развития технологии программирования с приемлемыми затратами.
- EAL3 определяется, как методически протестированный и проверенный. На уровне EAL3 осуществляется более полное, по отношению к уровню EAL2, тестирование покрытия функций безопасности, а также контроль среды разработки и управление конфигурацией объекта оценки.
- EAL2 определяется, как структурно протестированный. Он предусматривает создание описательного проекта верхнего уровня объекта оценки, описания процедур инсталляции и поставки, руководств администратора и пользователя, функциональное и независимое тестирование, оценка прочности функций безопасности, анализ уязвимостей разработчиками.
- EAL1 определяется, как функционально протестированный. Он обеспечивает анализ функций безопасности с использованием

функциональной спецификации и спецификации интерфейсов, руководящей документации, а также независимое тестирование. На этом уровне угрозы не рассматриваются как серьезные.

В соответствии с требованиями Общих критериев, продукты определенного класса (например, операционные системы) оцениваются на соответствие ряду функциональных критериев и критериев доверия – профилей защиты. Существуют различные определения профилей защиты в отношении операционных систем, брандмауэров, смарт-карт и прочих продуктов, которые должны соответствовать определенным требованиям в области безопасности. Например, профиль защиты систем с разграничением доступа (Controlled Access Protection Profile) действует в отношении операционных систем и призван заменить старый уровень защиты C2, определявшийся в соответствии с американским стандартом TCSEC. В соответствии с оценочными уровнями доверия сертификация на более высокий уровень означает более высокую степень уверенности в том, что система защиты продукта работает правильно и эффективно, и, согласно условиям Общих критериев, уровни 5-7 рассчитаны на тестирование продуктов, созданных с применением специализированных технологий безопасности.

Следует отметить, что большинство усилий по оценке продуктов безопасности сосредоточены на уровне 4 стандарта Общих критериев и ниже, что говорит об ограниченном применении формальных методов в этой области.

С точки зрения программиста Общие критерии можно рассматривать как набор библиотек, с помощью которых пишутся задания по безопасности, типовые профили защиты и т.п. Следует отметить, что требования могут быть параметризованы.

1.7. Настраиваемость операционных систем

В последнее время одной из главных тем исследовательских работ в области операционных систем стало исследование настраиваемости (customizability) или адаптируемости операционной системы. Настраиваемой или адаптируемой операционной системой называется операционная система, позволяющая гибкую модификацию основных механизмов, стратегий и политик системы. В зависимости от контекста, настраиваемость системы может преследовать различные цели. В операционных системах общего назначения, как правило, такой целью является производительность системы в целом. Для встроенных систем настраиваемость служит целям энергосбережения и/или сокращения объема программного обеспечения. Детальный систематический обзор исследовательских операционных систем с точки зрения их настраиваемости дается в работе Дениса и др. [DPM02].

В ранних ОС присутствовала некая форма настройки, чаще всего она заключалась в возможности настраивать систему на этапе ее генерации. Однако в последнее время появились исследования и других способов адаптации ОС – это касается инициатора настройки и времени ее осуществления. Инициатором адаптации может быть администратор или проектировщик ОС (т.е. человек),

приложение или сама операционная система. В последнем случае адаптация называется автоматической. Что касается времени настройки, то она может происходить на этапе проектирования, компоновки или инсталляции (статическая адаптация), а также во время загрузки и даже во время выполнения (динамическая адаптация).

2. Краткие характеристики наиболее распространенных ОСРВ

Большинство распространенных ОСРВ являются проприетарными, поэтому информация о них не всегда доступна. В этом разделе описаны наиболее распространенные ОСРВ в порядке объема собранных о них сведений.

2.1. VxWorks

Операционные системы реального времени семейства **VxWorks** корпорации **WindRiver Systems** предназначены для разработки программного обеспечения (ПО) встраиваемых компьютеров, работающих в системах жесткого реального времени [VxWorks]. Операционная система **VxWorks** обладает кросс-средствами разработки программного обеспечения (ПО), т.е. разработка ведется на инструментальном компьютере (host) в среде Tornado для дальнейшего ее использования на целевом компьютере (target) под управлением системы **VxWorks**.

Операционная система **VxWorks** имеет архитектуру клиент-сервер, и построена в соответствии с технологией микроядра, т.е. на самом нижнем непрерываемом уровне ядра (**WIND** *Microkernel*) обрабатываются только планирование задач и управление их взаимодействием/синхронизацией. Вся остальная функциональность операционного ядра – управление памятью, вводом/выводом и пр. – выполняется на более высоком уровне и реализуется через процессы. Это обеспечивает быстродействие и детерминированность ядра, а также масштабируемость системы.

VxWorks может быть скомпонована как для небольших встраиваемых систем с жесткими ограничениями для памяти, так и для сложных систем с развитой функциональностью. Более того, отдельные модули сами являются масштабируемыми. Конкретные функции можно убрать при сборке, а специфические ядерные объекты синхронизации можно опустить, если приложение в них не нуждается.

Хотя система **VxWorks** является конфигурируемой, т.е. отдельные модули можно загружать статически или динамически, нельзя сказать, что она использует подход, основанный на компонентах. Все модули построены над базовым ядром и спроектированы таким образом, что не могут использоваться в других средах.

Основными параметрами ядра **VxWorks** являются следующие характеристики:

- количество задач не ограничено,
- число уровней приоритетов задач – 256,
- планирование задач возможно двумя способами – вытеснение по приоритетам и циклическое,

- средствами взаимодействия задач служат очереди сообщений, семафоры, события и каналы (для взаимодействия задач внутри CPU), сокеты и удаленные вызовы процедур (для сетевого взаимодействия), сигналы (для управления исключительными ситуациями) и разделяемая память (для разделения данных),
- для управления критическими системными ресурсами обеспечивается несколько типов семафоров: двоичные, вычислительные (counting) и взаимно исключающие с приоритетным наследованием,
- детерминированное переключение контекста.

VxWorks предлагает как POSIX-й, так и собственный механизмы планирования (wind scheduling). Оба варианта включают вытесняющее и циклическое планирование. Различие между ними состоит в том, что wind scheduling применяется на системном базисе, в то время как алгоритмы POSIX-планирования применяются на базисе процесс-за-процессом.

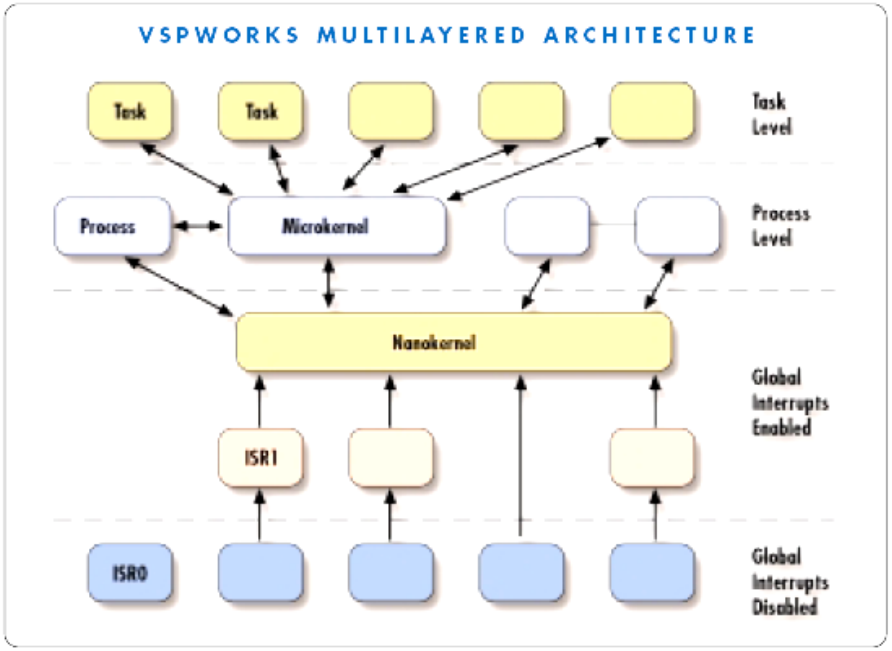

*Рис. 1. Многослойная архитектура **VSPWorks.***

В **VxWorks** все задачи системы и приложений разделяют единственное адресное пространство, что чревато нарушением стабильности системы из-за неисправности какого-либо приложения. Необязательный компонент VxVMI дает возможность каждому процессу иметь свою собственную виртуальную память.

Для того, чтобы достичь быстрой обработки внешних прерываний, программы обработки прерываний (ISRs – interrupt service routines) в **VxWorks**

выполняются в специальном контексте вне контекстов потоков, что позволяет выиграть время, которое обычно тратится на переключение контекстов. Следует отметить, что С-функция, которую пользователь присоединяет к вектору прерывания, на самом деле не является фактической ISR. Прерывания не могут непосредственно обращаться к С-функциям. Адрес ISR запоминается в таблице векторов прерываний, которая вызывается аппаратно. ISR выполняет некую начальную обработку (сохранение регистров и подготовку стека), а затем вызывается С-функция, которая была присоединена пользователем.

VSPWorks [VSPWorks] – это весьма популярная и достаточно мощная ОС на основе **VxWorks**. **VSPWorks** спроектирована специально для систем, основанных на DSP. Она обеспечивает многозадачный режим с приоритетами и поддержку быстрых прерываний на процессорах DSP и ASIC. ОСРВ **VSPWorks** следует модели единственного виртуального процессора, что значительно упрощает распределение приложений в многопроцессорной системе, сохраняя при этом производительность жесткого реального времени. **VSPWorks** является модульной и масштабируемой.

ОСРВ **VSPWorks** обладает многослойной структурой, что служит хорошей основой для абстрагирования и переносимости. Центром системы служит сильно оптимизированное наноядро (nanokernel), которое способно управлять совокупностью процессов. Ниже наноядра находятся программы, обслуживающие прерывания, выше наноядра располагается микроядро, которое управляет многозадачным режимом с приоритетами C/C++ задач.

Управление прерываниями имеет два уровня. Нижний уровень (уровень 1) используется для обработки аппаратных прерываний. Во время обработки таких прерываний все остальные прерывания блокируются. Код, выполняющийся на этом уровне, написан на языке ассемблера, и ответственность за сохранение соответствующих регистров на стеке ложится на программиста. На этом уровне может быть обработано прерывание, которое требует малого времени для обработки. Если обработка прерывания является более сложной и требует большего времени, тогда прерывание обрабатывается на более высоком уровне (уровень 2), где разрешено прерывание прерывания и, таким образом, они могут быть вложенными. Переход на более высокий уровень прерываний происходит по системному вызову.

Процессы наноядра (уровень 3) пишутся на языке ассемблера и имеют сокращенный контекст (т.е. используют меньше регистров). Эти процессы могут быть загружены и разгружены с процессора очень быстро. Каждому процессу присваивается приоритет. Уровень 3 идеален для написания драйверов для интерфейсов аппаратуры низкого уровня.

Микроядро находится на уровне 4. Микроядро написано на языке С и имеет свыше 100 сервисов. Обработка задач на этом уровне ведется в режиме приоритетного прерывания, и планирование управляется приоритетами.

Сетевые средства. VxWorks поддерживает все сетевые средства, стандартные для UNIX: TCP/zero-copyTCP/UDP/ICMP/IP/ARP, SLIP/CSLIP/PPP, Sockets, telnet/rlogin/rpc/rsh, ftp/tftp/bootp, NFS (Network File System) (клиент и сервер).

В сетевые средства для VxWorks входят также функции, необходимые при разработке устройств, подключаемых к Internet: IP multicasting уровня 0,1 или 2; long fat pipe; CIDR (Classless Inter-Domain Routing); DHCP (Dynamic Host Configuration Protocol) в конфигурациях server, client и relay agent; DNS client (Domain Naming System); SNTP (Simple Network Time Protocol). **VxWorks** поддерживает протоколы маршрутизации RIPv1/RIPv2 (Routing Information Protocol), а также OSPF (Open Shortest Path First) версии 2. Протокол RIP входит в стандартную поставку **VxWorks**, OSPF поставляется как дополнительный продукт. SNMP-агент для **VxWorks** поддерживает протокол SNMP (Simple Network Management Protocol) как версии v1, так и v2c. MIB (Management Information Base) компилятор поддерживает объекты MIB-II и расширения. STREAMS – стандартный интерфейс для подключения переносимых сетевых протоколов к операционным системам. Под **VxWorks** можно инсталлировать любой протокол, имеющий STREAMS-реализацию: как стандартный (Novell SPX/IPX, Decnet, AppleTalk, SNA и т.п.), так и специализированный. **VxWorks** поддерживает STREAMS версии UNIX System V.4.

Графические пакеты и встроенный Интернет. Графические приложения для встраиваемых компьютеров с ОСРВ **VxWorks** могут быть разработаны как на языке C/C++, так и на языках Java и HTML. Для разработки графических пользовательских интерфейсов (GUI) на языке C++ поставляется программный продукт Zinc for **VxWorks** , для разработки на языке Java – PersonalJWorks и для разработки на языке HTML – HTMLWorks/eNavigator. Все три GUI для VxWorks используют один и тот же универсальный API к графической аппаратуре (графическому контроллеру, фрэйм-буферу и устройству ввода), который называется UGL (Universal Graphics Library). UGL – это набор графических 2D примитивов, драйверы популярных графических контроллеров и средства разработки собственных пользовательских графических драйверов. UGL входит в состав каждого GUI-продукта и поставляется в исходных текстах.

Zinc for **VxWorks** – это C++ API, предоставляющий широкий набор графических объектов с задаваемыми пользователем параметрами. Для разработки GUI используется Zinc Designer – WYSIWYG-редактор, который входит в комплект поставки. Графический интерфейс может быть разработан на языке Java с использованием стандартного pAWT (Abstract Windowing Toolkit), входящего в состав PersonalJWorks. Для разработки GUI используется любой инструментарий разработки Java-приложений. Пользовательский интерфейс может быть разработан с использованием графических возможностей языка HTML (фрэймы, изображения, таблицы, формы) и динамических возможностей JavaScript. HTMLWorks – это интерпретатор HTML/JavaScript-страниц, которые могут находиться в постоянной памяти или быть загружены по сети. Для разработки GUI используется любой инструментарий web-дизайна. Если встраиваемый компьютер с HTML GUI должен уметь выполнять web-серфинг, то совместно с HTMLWorks может быть использован браузер для встраиваемых приложений eNavigator.

Средства построения мультипроцессорных систем. VxWorks поддерживает два вида мультипроцессинга: слабосвязанный – через распределенные очереди сообщений и сильносвязанный – через объекты в разделяемой памяти. Слабосвязанный мультипроцессинг через распределенные очереди сообщений реализован в библиотеке VxFusion, которая является отдельным продуктом. VxFusion применяется для обмена между процессорами, не имеющими общей памяти (например, между узлами сети). Сильносвязанный мультипроцессинг через объекты в разделяемой памяти реализован в библиотеке VxMP, которая также является отдельным продуктом. VxMP применяется для обмена между процессорами, имеющими общую область памяти (например, находящимися на одной шине).

Средства портирования. Все аппаратно-зависимые части **VxWorks** вынесены в отдельные модули для того, чтобы разработчик встраиваемой компьютерной системы мог сам портировать **VxWorks** на свой нестандартный целевой компьютер. Этот комплект конфигурационных и инициализационных модулей называется BSP (Board Support Package) и поставляется для стандартных компьютеров (VME-процессор, PC или Sparcstation) в исходных текстах. Разработчик нестандартного компьютера может взять за образец BSP наиболее близкого по архитектуре стандартного компьютера и портировать **VxWorks** на свой компьютер путем разработки собственного BSP с помощью BSP Developer's Kit.

Промежуточное ПО (middleware). Модель компонентных объектов COM (Component Object Model) и ее расширение для распределенных систем DCOM (Distributed COM) являются стандартными интерфейсами обмена между приложениями для Windows. VxDCOM – DCOM для операционной системы **VxWorks** – это первая реализация модели распределенных компонентных объектов для систем реального времени. Теперь нет необходимости в разработке специализированных драйверов ввода/вывода при интеграции нижнего и верхних уровней распределенной системы управления. VxDCOM поддерживает также OPC-интерфейсы (OLE for Process Control), что позволяет разрабатывать OPC-серверы для встраиваемых систем, работающих под управлением ОСРВ **VxWorks**.

Файловая система для флэш-памяти. Файловая система TrueFFS предназначена для эмуляции жесткого диска, работающего под управлением файловых систем **VxWorks**: DOS-FS и NFS (Network File System). TrueFFS удовлетворяет стандарту PCMCIA FTL (Flash Translation Level) и поддерживает PC-cards, MiniatureCards и микросхемы флэш-памяти Intel 28F0xx, AMD 29F0xx, и Samsung 29Vxx000.

2.2. QNX Neutrino RTOS

Операционная система QNX Neutrino Realtime Operating System (RTOS) [QNXNeutrino] корпорации QNX Software Systems является микроядерной операционной системой, которая обеспечивает многозадачный режим с приоритетами. QNX Neutrino RTOS имеет клиент-серверную архитектуру. Под

QNX Neutrino каждый драйвер, приложение, протокол и файловая система выполняются вне ядра, в защищенном адресном пространстве. В случае сбоя любого компонента он может автоматически перезапуститься без влияния на другие компоненты или ядро. Хотя система QNX является конфигурируемой, т.е. отдельные модули можно загружать статически или динамически, нельзя сказать, что она использует подход, основанный на компонентах. Все модули полагаются на базовое ядро и спроектированы таким образом, что не могут использоваться в других средах.

QNX Neutrino RTOS состоит из ядра, планировщика процессов (process manager) и расширенных сервисов на уровне пользователя. Как истинная микроядерная операционная система **QNX Neutrino RTOS** реализует только наиболее фундаментальные сервисы в ядре ОС, такие как передача сообщений, сигналы, таймеры, планирование потоков, объекты синхронизации. Все другие сервисы ОС, драйверы и приложения выполняются как отдельные процессы, которые взаимодействуют через синхронную передачу сообщений.

Ядро **QNX Neutrino RTOS** выполняется на уровне 0, управляющие программы и драйверы устройств выполняются на уровнях 1 и 2, совершая операции ввода/вывода. Приложения выполняются на уровне 3.

Планировщик процессов строится на базисе ядра и обеспечивает дополнительную семантику уровня процессов, управление памятью и путями доступа к файлам. Все другие компоненты – файловые системы, набор протоколов, очереди сообщений, приложения – выполняются в защищенном адресном пространстве, и являются расширенными сервисами. Взаимодействие компонентов осуществляется через передачу сообщений. Передача сообщений играет роль виртуальной “программной шины”, которая позволяет оперативно динамически подгружать и отгружать любой компонент. Как следствие, любой модуль, даже драйвер устройства, может быть замещен или перезапущен оперативно, что в большинстве ОСРВ требует перезапуска системы. Сообщения передаются прозрачно через границы процессора, обеспечивая бесшовный доступ к любому ресурсу в сети.

Обладая вытесняющим микроядром и планировщиком с приоритетным обслуживанием **QNX Neutrino RTOS** способна реагировать быстро и с высокой предсказуемостью на события реального времени. Высокоприоритетные потоки обрабатывают дедлайны своевременно, даже при большой загрузке системы (см. рис. 2).

QNX Neutrino RTOS имеет малые времена обработки прерываний, быстрое переключение контекстов. Инверсия приоритетов преодолевается с помощью распределенного наследования приоритетов. Упрощенное моделирование активностей реального времени проводится через синхронную передачу сообщений. Вложенные прерывания и фиксированная верхняя граница времени обработки прерывания гарантируют, что высокоприоритетные прерывания обрабатываются быстро с предсказуемым временем.

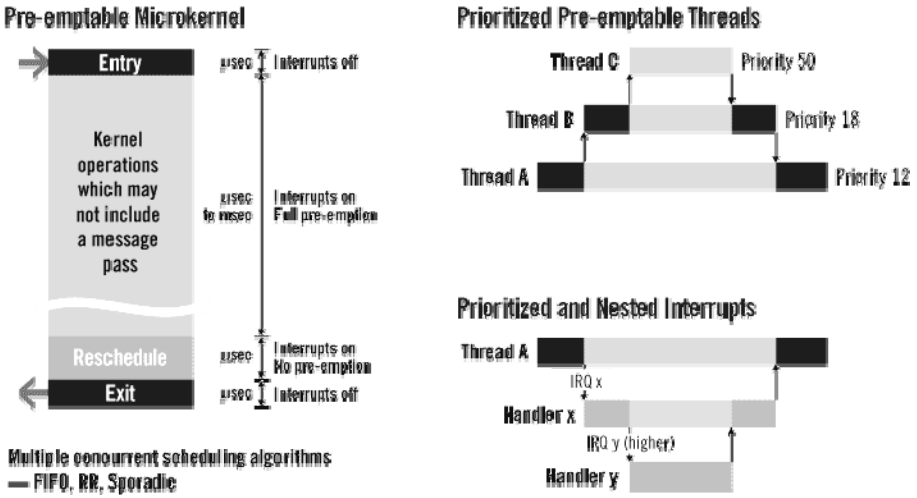

Рис. 2. Производительность реального времени QNX Neutrino RTOS.

2.3. RTEMS

RTEMS (Real-Time Executive for Multiprocessor Systems) – это некоммерческая операционная система реального времени для глубоко встраиваемых систем [RTEMS]. Разработчик системы компания OAR (On-Line Applications Research Corporation, США). Система была создана по заказу министерства обороны США для использования в системах управления ракетными комплексами. Система разрабатывается для многопроцессорных систем на основе открытого исходного кода в противовес аналогичным системам с закрытым кодом. Система рассчитана на платформы MS-Windows и Unix (GNU/Linux, FreeBSD, Solaris, MacOS X).

Ядро **RTEMS** обеспечивает базовую функциональность систем реального времени. В эти возможности входят

- мультизадачная обработка;
- работа в гомогенных и гетерогенных системах;
- планирование, управляемое событиями, на основе приоритетов;
- возможно планирование с монотонной скоростью;
- взаимодействие задач и синхронизация;
- приоритетное наследование;
- управление ответным прерыванием;
- распределение динамической памяти;
- возможность конфигурирования системы для уполномоченных пользователей;
- переносимость на многие целевые платформы.

Ядро RTEMS отвечает за управление основной памятью компьютера и виртуальной памятью выполняемых процессов, за управление процессором и планирование распределения процессорных ресурсов между совместно выполняемыми процессами, за управление внешними устройствами и, наконец, за обеспечение базовых средств синхронизации и взаимодействия процессов. При этом ядро использует соответствующие менеджеры. В состав RTEMS входит набор следующих менеджеров: инициализации, задач, прерываний, часов реального времени, таймера, семафоров, сообщений, событий, сигналов, разделов, регионов, двухпортовой памяти, ввода/вывода, неисправимых ошибок, монотонной частоты, расширений пользователя, много-процессорности. Привязка ОСРВ к аппаратуре производится с помощью специальной библиотеки подпрограмм BSP (board support package) и специализированных подпрограмм для различных архитектур. В состав BSP входят программа инициализации аппаратуры и драйверы устройств. Поддержка мультипроцессорных систем в RTEMS позволяет использовать ее для управления как однородными, так и неоднородными системами Ядро RTEMS автоматически учитывает различия в архитектуре используемых процессоров, выполняя в случае необходимости перестановку байтов и другие процедуры. Это позволяет осуществлять переход на другое семейство процессоров без значительных изменений системы.

ОСРВ RTEMS можно рассматривать как набор компонентов, обеспечивающих ряд базовых сервисных функций для программ пользователя. Программный интерфейс приложения состоит из директив, распределенных по логическим наборам соответствующих менеджеров. Функции, используемые несколькими менеджерами, такие как распределение процессорного времени, диспетчеризация и управление объектами, реализованы в ядре. Ядро содержит также небольшой набор процедур, зависящих от типа используемого процессора: доступ к физической памяти, инициализация контроллера прерываний и периферийных устройств, специфичных для данного процессорного ядра, и ряд других функций.

ОСРВ RTEMS реализует следующие виды межпроцессорного взаимодействия:

- обмен данными между задачами;
- обмен данными между задачами и программами обработки прерываний;
- синхронизация между задачами;
- синхронизация между задачами и программами обработки прерываний.

Функции, позволяющие осуществлять те или иные виды межпроцессорного взаимодействия, входят в большинство менеджеров RTEMS. Из них четыре менеджера семафоров, сообщений, событий, сигналов предназначены исключительно для осуществления межпроцессорного взаимодействия.

Менеджер семафоров. RTEMS поддерживает стандартные двоичные и семафоры со счетчиками, обеспечивающие синхронизацию и эксклюзивный доступ к ресурсам.

Менеджер событий. Служит для синхронизации выполнения задач. Флаг события используется задачей для того, чтобы информировать другую задачу о возникновении определенного события. Каждой задаче соответствуют 32 флага событий. Совокупность одного или более флагов называется набором событий. Одна задача может послать другой задаче набор событий, а также выяснить состояние набора событий соответствующей функцией.

Менеджер сообщений. Служит для обмена сообщениями между задачами. Сообщение это буфер переменной длины, используемый для обмена данными. Сообщения передаются в виде очередей типа FIFO ("первым пришёл, первым обслужен"). Имеется возможность посылки срочного сообщения. Для каждой очереди задается максимальная длина сообщения. Сообщения могут использоваться для синхронизации задач. Задача может ожидать прихода определенного сообщения или проверять наличие сообщения в очереди.

Менеджер сигналов. Используется для асинхронного взаимодействия между задачами. Задача может включать в себя процедуру обработки асинхронного сигнала, которой передается управление при получении сигнала. Флаг сигнала используется задачей для того, чтобы проинформировать другую задачу о возникновении нештатной ситуации. Каждой задаче соответствуют 32 флага сигналов. Совокупность одного или более флагов называется набором сигналов.

Менеджер задач. Обеспечивает полный набор функций для создания, удаления и управления задачами. С точки зрения RTEMS, задачей является наименьшая последовательность команд, которая может самостоятельно конкурировать за использование системных ресурсов. Каждой задаче соответствует блока контроля задачи TCB (Task Control Block). Этот блок является структурой, которая содержит всю информацию, относящуюся к выполнению задачи. В процессе инициализации RTEMS выделяет TCB для каждой задачи, имеющейся в системе. Элементы TCB изменяются в соответствии с систем-ными вызовами, которые выполняются приложением в ответ на внешние запросы. Блок TCB единственная внутренняя структура данных RTEMS, доступная приложению через дополнительные процедуры пользователя. При переключении задач в TCB сохраняется контекст задачи. При возвращении управления задаче ее контекст восстанавливается. При перезапуске задачи исходный контекст задачи восстанавливается в соответствии со стартовым контекстом, хранящемся в TCB. Задача может находиться в одном из пяти состояний: выполнение; готовность к выполнению (управление может быть передано задаче); остановка (задача заблокирована); спящий режим (созданная, но не запущенная задача); отсутствие задачи (задача не создана или удалена).

Ядро реального времени RTEMS поддерживает 255 уровней приоритетов. Чем больше значение приоритета, тем более привилегированной является задача. Количество задач, имеющих одинаковый приоритет, не ограничено. Каждая задача всегда имеет какой-либо уровень приоритета, начальное значение которого присваивается при создании задачи и в дальнейшем может быть изменено. Режим выполнения задачи определяется следующими параметрами:

вытесняемость; обработка асинхронных запросов ASR (Asynchronous Signal Request); квантование времени; уровень прерывания. Эти параметры используются для распределения процессорного времени и изменения контекста задачи. Они задаются пользователем при компиляции системы.

Параметр вытесняемость определяет порядок передачи управления между задачами. Если он включен, то задача сохранит контроль над процессором, пока она находится в состоянии выполнения, даже если готова к выполнению более привилегированная задача. Если этот параметр выключен, то управление будет немедленно передано задаче, имеющей более высокий приоритет.

Параметр квантование времени определяет, как происходит распределение процессорного времени между задачами с одинаковым приоритетом. Если он включен, то RTEMS ограничит время выполнения задачи при наличии другой задачи с таким же приоритетом, готовой к выполнению. Время, выделяемое каждой такой задаче, определяется в таблице конфигурации системы. Если квантование времени выключено, то задача будет выполняться до тех пор, пока не будет готова к выполнению задача с более высоким приоритетом. В том случае, если параметр вытесняемость выключен, параметр квантование времени не учитывается.

Параметр обработка асинхронных сигналов (запросов) ASR определяет порядок обработки получаемых задачей сигналов (запросов). Если он включен, то посланные задаче сигналы будут обработаны, если выключен – сигналы будут обработаны только после включения этого параметра. Данный параметр влияет только на задачи, имеющие процедуры обработки внешних сигналов.

Параметр уровень прерывания определяет, какие прерывания могут обрабатываться при выполнении задачи.

Менеджер инициализации. Отвечает за запуск и остановку RTEMS. Инициализация RTEMS производится путем создания и запуска всех инициализирующих задач и инициализирующих процедур для каждого драйвера. В случае мультипроцессорной системы происходит также инициализация механизмов межпроцессорного взаимодействия. Инициализирующие задачи отличаются от остальных задач тем, что они присутствуют в таблице инициализирующих задач пользователя и автоматически создаются RTEMS в процессе инициализации. Чтобы эти задачи выполнялись до запуска остальных задач, они должны иметь более высокий приоритет. После окончания инициализации RTEMS не удаляет инициализирующие задачи, поэтому такие задачи должны либо сами удалить себя, либо трансформироваться в "обычную" задачу. В любой системе должна быть, как минимум, одна инициализирующая задача.

Менеджер прерываний позволяет быстро реагировать на прерывания, обеспечивая возможность "вытеснения" задачи сразу после выхода из процедуры обработки прерывания. Менеджер прерываний дает также возможность программе пользователя подключить процедуру обработки к соответствующему вектору прерывания. Когда поступает запрос прерывания, процессор передает его ядру RTEMS. При обслуживании запросов прерывания RTEMS сохраняет и восстанавливает содержимое всех регистров, сохранение

которых не предусмотрено правилами языка С, а затем вызывает пользовательскую процедуру обработки прерывания. Для минимизации времени, в течение которого не обслуживаются запросы прерывания равного или более низкого уровня, процедура обработки должна выполнять лишь минимальный набор необходимых действий. Дальнейшая обработка должна осуществляться программой пользователя. Менеджер прерываний гарантирует правильное распределение процессорного времени между задачами после завершения процедуры обработки прерывания. Системный вызов, сделанный из процедуры обработки прерывания, может перевести в состояние готовности к исполнению задачу с большим приоритетом, чем прерванная задача. Поэтому необходимо произвести отложенную диспетчеризацию после завершения процедуры обработки прерывания. Вызов директив RTEMS из процедуры обработки прерывания не сопровождается диспетчеризацией.

Для правильного распределения процессорного времени между задачами должно выполняться следующее условие: все процедуры обработки прерываний, которые могут быть прерваны процедурами обработки прерываний, вызывающими директивы RTEMS с большим приоритетом, должны использовать менеджер прерываний. Если при обработке прерывания поступает новый запрос на прерывание, его обработка происходит сразу после завершения текущей процедуры обработки. Отложенная диспетчеризация производится только после того, как будут обслужены все запросы. ОСРВ RTEMS поддерживает 256 уровней прерываний, которые транслируются в уровни прерываний процессора.

При выполнении определенных директив RTEMS может возникнуть необходимость отключения обработки прерываний, чтобы обеспечить непрерывное выполнение критических сегментов программы. Система RTEMS отключает все маскируемые прерывания перед выполнением этих сегментов. Максимальное время отключения прерываний различно для разных процессоров и указывается в документации RTEMS для соответствующего процессора. Немаскируемые прерывания не отключаются, поэтому в процедурах их обработки не должны использоваться директивы RTEMS.

Менеджер ввода/вывода. Обеспечивает определенный механизм доступа к драйверам устройств. Если в системе используется этот менеджер, то в конфигурационной таблице должен быть указан адрес таблицы драйверов устройств, которая содержит входные точки каждого драйвера. Драйвер может иметь следующие точки входа: инициализации, открытия, закрытия, чтения, записи, контроля.

Менеджер доступа к памяти. Для работы с памятью служат менеджеры разделов и регионов. Раздел это область памяти, состоящая из буферов фиксированной длины. Каждый из этих буферов может быть выделен для использования с помощью директив менеджера разделов. Регион это область памяти переменной длины, кратной размеру страницы для данного региона. Раздел представляет собой список буферов. При запросе на выделение буфера он выделяется из начала цепи свободных буферов. Когда буфер освобождается, он помещается в конец этой цепи. Регион состоит из блоков

памяти различного размера, который кратен размеру страницы для данного региона. При поступлении запроса на выделение блока памяти размер запрошенного блока округляется до целого количества страниц и при наличии свободного блока соответствующего размера этот блок выделяется. Менеджер доступа к памяти реализует следующий набор функций: создание, удаление, установка значений, освобождение, захват областей регионов/разделов и буферов, содержащихся в них. Для регионов реализуется возможность добавления памяти.

Менеджер таймеров обеспечивает работу с таймерами: создание и удаление таймеров, доступ к таймерам, запуск подпрограмм по событию/сигналу от таймера. Этот менеджер может быть использован для создания сторожевого таймера.

Менеджер часов реального времени используется для информирования пользователя о текущей дате. Обеспечивает также формирование и обработку сигналов об истечении минимальных промежутков времени, которые задаются на этапе конфигурирования системы и равны целому числу микросекунд.

RTEMS не поддерживает динамическую загрузку приложений и модулей, поэтому сферой ее применения являются встраиваемые системы, в которых не предполагается частая модификация программного обеспечения. ОСРВ RTEMS обеспечивает достаточно слабую поддержку файловых систем, что ограничивает область ее возможного применения в сфере систем централизованного сбора и хранения данных стандартными высокоуровневыми средствами. На настоящий момент RTEMS поддерживает только файловые системы IMFS и TFTP, что явно недостаточно. Поэтому для создания на базе RTEMS файл-серверов требуется разработка специального протокола. Понимая эту проблему, разработчики RTEMS ведут активную работу по реализации систем поддержки широко используемых файловых систем (в первую очередь сетевых). В RTEMS фактически отсутствуют резидентные средства отладки. Имеются только стандартные функции rtems_panic и printf, которые позволяют выводить отладочную информацию на терминал в процессе работы системы. Следует, однако, отметить, что наличие мощных средств кросс-разработки делает этот недостаток не очень существенным.

2.4. ChorusOS

Операционная система **ChorusOS –** это масштабируемая встраиваемая ОС, широко применяемая в телекоммуникационной индустрии. В настоящее время этот бренд развивается и распространяется корпорацией Sun Microsystems [CHORUSOS]. Для того, чтобы компоновать и развертывать ОС **ChorusOS** на конкретных телекоммуникационных платформах, Sun Microsystems предлагает к использованию среду разработки Sun Embedded Workshop. Корпорация Sun Microsystems представляет ОС **ChorusOS** как встраиваемую основу для Sun'овской сети, управляемой сервисами (Sun's Service-Driven Network). В сочетании с широким набором сервисов, полной интеграцией ПО и аппаратуры, удобным администрированием и поддержкой Java-технологии, которая посвящена нуждам телекоммуникации, ОС **ChorusOS** дает

возможность эффективно развертывать новые возможности и приложения, поддерживая надежность и функциональность современных сетей.

ОС **ChorusOS** поддерживает на одной аппаратной платформе широкий набор телекоммуникационных протоколов, унаследованных приложений, приложений режима реального времени и Java-технологии.

ОС **ChorusOS** моделирует три сорта приложений:

- POSIX-процессы составляют большинство приложений ChorusOS, эти приложения имеют доступ к чисто POSIX APIs, нескольким POSIX-подобным расширенным APIs и небольшому числу ограниченных системных вызовов микроядра,
- Акторы ChorusOS – эти приложения выполняются на верхушке микроядра и ограничиваются API микроядра, акторы включают драйверы, события подсистем и протокольные стеки,
- Унаследованные приложения ChorusOS поддерживаются для совместимости с приложениями, разработанными для более ранних версий ChorusOS.

Архитектура ОС **ChorusOS** является многослойной, основанной на компонентах (component-based). Микроядро содержит минимальный набор компонент, необходимых для функционирования ОС:

- kern – реализует интерфейс микроядра и содержит актор KERN, вспомогательную библиотеку и заголовочные файлы,
- менеджер приватных данных (pd) реализует интерфейс между подсистемами микроядра,
- менеджер постоянной памяти (pmm) реализует интерфейс постоянной памяти,
- core executive обеспечивает существенную часть поддержки реального времени.

Компонент диспетчер ядра (core executive) обеспечивает следующую функциональность:

- поддержка многочисленных независимых приложений,
- поддержка пользовательских и системных приложений,
- поддержка актора – единицы модуляризации приложений,
- поддержка единицы исполнения – потока,
- операции управления потоками,
- управление Local Access Point (LAP),
- сервисы управления исключительными ситуациями,
- минимальный сервис управления прерываниями.

В core executive отсутствует управление такими вещами, как синхронизация, планирование, время, память. Политики управления этими понятиями обеспечиваются дополнительными компонентами, которые выбираются пользователем в зависимости от требований аппаратных и программных средств. Core executive всегда присутствует в исполняемом экземпляре ОС

ChorusOS, остальные компоненты конфигурируются и добавляются по необходимости. Размер резидентной часть ядра составляет 10Kb.

Понятие “актор” в **ChorusOS** определяется как единица загрузки для приложения. Оно также служит единицей инкапсуляции для того, чтобы сопоставить все системные ресурсы, используемые приложением, и потоки, выполняющиеся внутри актора. Примерами таких ресурсов являются потоки, регионы памяти и конечные точки взаимодействия.

Необязательные компоненты ОС **ChorusOS 5.0** разбиваются в соответствии с функциональностью:

- Управление деятельностью (Actor management) включает поддержку расширения режима пользователя, динамические библиотеки, управление сжатыми файлами;
- Планирование (Scheduling) включает планирование в стиле FIFO (first-in-first-out), разностильное планирование (multi-class scheduling), циклическое планирование (round-robin), планирование в режиме реального времени;
- Управление памятью включает кроме распределения памяти, поддержки аппаратной защиты и подкачки еще и статистику микроядра, события системы Solaris, метрики операционной системы;
- Работоспособность (High Availability) включает горячий рестарт, сторожевой таймер (Watchdog timer), черный ящик, дамп системы;
- Синхронизация потоков включает семафоры, наборы флагов событий, мьютексы, монопольные блокировки, обеспечивающие отсутствие инверсии приоритетов;
- Управление включает периодические таймеры, потоковый виртуальный таймер, дата и время, датчик реального времени, переменные окружения;
- Взаимодействие потоков включает независимое от местонахождения взаимодействие, поддержку удаленного взаимодействия, механизм взаимодействия через почтовые ящики, синхронизацию между потоками, приватные данные потока, такие средства взаимодействия системы POSIX, как семафоры, сокеты, потоки, таймеры, очереди сообщений, объекты разделяемой памяти, сигналы реального времени;
- Инструментальная поддержка включает системную журнализацию, регистрацию ошибок, поддержку профилирования и контрольных точек, мониторинг системы, отладку системы, дамп ядра;
- Поддержка языка C включает командный интерпретатор на целевом компьютере, удаленный shell;
- Поддержка файловой системы включает именованные каналы, NFS-клиент, NFS-сервер, файловые системы MS-DOS, PDE, /proc, UFS, ISO9000;
- Управление вводом/выводом включает поддержку драйверов некоторых устройств;
- Сетевая поддержка включает поддержку некоторых сетевых протоколов.

Выделение управления памятью в отдельный необязательный компонент позволяет легко адаптировать систему к разным аппаратным платформам.

ОС **ChorusOS 5.0** лежит в основе операционной среды Solaris и поддерживает следующие целевые платформы:

- UltraSPARC II (CP1500 и CP20x0),
- Intel x86, Pentium,
- Motorola PowerPC 750 и семейство процессоров 74x0 (mpc7xx),
- Motorola PowerQUICC I (mpc8xx) и PowerQUICC II (mpc8260) (микроконтроллеры).

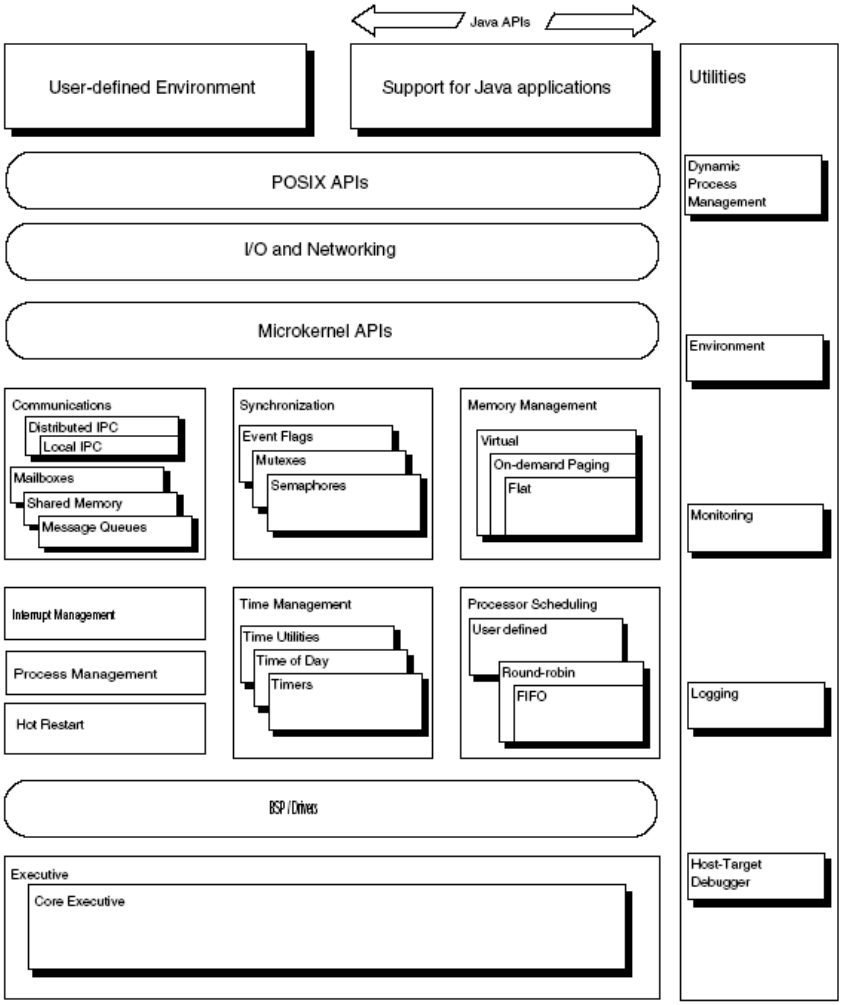

*Рис. 3. Архитектура **ChorusOS**.*

2.5. Расширения реального времени для Windows NT

Windows NT проектировалась и, в основном, используется как универсальная ОС. Однако, на рынке систем реального времени четко прослеживается тенденция использовать Windows NT и ее расширения в специализированных системах. На это существует несколько причин:

- Windows NT проектировалась согласно современным технологиям построения ОС,
- программный интерфейс приложений (API) для Win32 стал де-факто стандартом для программистов,
- графический пользовательский интерфейс (GUI) стал настолько популярным, что другие ОС стараются обеспечить похожий интерфейс,
- доступно большое количество драйверов устройств,
- доступны многие мощные интегрированные среды разработки.

Сама по себе Windows NT не подходит для применения в системах реального времени, поскольку в ней слишком мало приоритетных уровней, отсутствует механизм наследования приоритетов. Для минимизации времени обработки прерываний (ISR) в Windows NT введена концепция отложенного вызова процедуры (DPC – deferred procedure call), приоритет которой выше, чем приоритет пользовательских и системных потоков, в то время как все DPCs имеют одинаковый приоритет. Это приводит к тому, что все DPCs ставятся в FIFO-очередь, и DPC с высокоуровневым прерыванием сможет выполниться только после того, как все другие DPCs, стоящие в очереди перед ней, будут выполнены. Такие ситуации ведут к непредсказуемым временам отклика, что несовместимо с требованиями к ОСРВ. Управление памятью в Windows NT основано на механизме виртуальной памяти. Это тянет за собой защиту памяти, трансляцию адресов и подкачку, которая неприемлема в ОСРВ.

2.5.1. RTX для Windows NT

Расширение реального времени RTX (Real Time Extension) для ОС Windows NT (разработано корпорацией VenturCom) позволяет создавать приложения для высокоскоростного управления с детерминированным временем реакции на внешние события [RTX].

RTX глубоко интегрировано в ядро Windows NT и для обеспечения необходимых функций использует сервис Windows NT и WIN32 API. Ядро реального времени (nucleus) интегрировано в ядро NT (kernel). Каждый процесс RTX выполняется как драйвер устройства ядра NT, при этом процессы не защищены друг от друга. Такая реализация приводит к быстрому переключению контекста, но небезопасна с точки зрения конфиденциальности.

Расширения реального времени добавляют к Windows NT специфическую для реального времени функциональность:

- возможность создавать процессы реального времени, управляемые собственным планировщиком. Этот планировщик работает уже по правилам реального времени и использует алгоритм вытеснения по приоритетам. Кроме того, процессы реального времени имеют преимущество перед стандартными процессами Win32, вытесняя их. Процессы реального времени имеют совсем иную, по сравнению со стандартными процессами Windows NT, степень надежности и специфическую функциональность;
- процессы реального времени и стандартные процессы Win32 имеют средства взаимодействия друг с другом;
- процессы реального времени имеют свой собственный программный интерфейс RTAPI, реализующий развитый набор средств, характерный для программных интерфейсов (API) ОСРВ;
- приложение может использовать как стандартные функции Win32, так и специфические функции API реального времени (RTAPI), что позволяет выделять критические участки кода приложений Windows NT и контролировать время и надежность их выполнения;
- возможность контроля над работоспособностью и временами реакции системы. Зависания стандартных приложений Windows NT или крах системы не приводят к зависанию приложений реального времени;
- возможность работы с быстрыми часами и таймерами высокого разрешения;
- возможность прямого доступа к памяти и физическим устройствам.

RTX включает в себя следующие компоненты:

- уровень аппаратных абстракций HAL (Hardware Abstraction Layer) реального времени (Real-Time HAL). HAL является программным компонентом самого низкого уровня при взаимодействии драйверов ядра с аппаратурой. В частности, именно на уровне HAL происходит первоначальная обработка прерываний от таймера,
- подсистему реального времени RTSS (Real-Time Subsystem),
- программный интерфейс расширений реального времени RTAPI (Real-Time Application Programming Interface). HAL реального времени подменяет стандартный HAL Windows NT.

Подсистема реального времени RTSS обеспечивает выполнение большинства функций и управление ресурсами расширений реального времени. С точки зрения реализации, RTSS выглядит как драйвер Windows NT и выполняется в режиме ядра. Это позволяет достаточно простым способом устроить взаимодействие между процессами реального времени и процессами Windows NT. RTSS обеспечивает выполнение функций RTAPI и содержит планировщик потоков реального времени со 128-ю фиксированными приоритетами. RTSS содержит также менеджер объектов, предоставляющий унифицированные механизмы использования системных ресурсов. По сравнению с набором

объектов Windows NT, добавлены такие, как таймеры и обработчики прерываний.

Работа с прерываниями Real-Time HAL. Перехватывая аппаратные прерывания, Real-Time HAL различает прерывания, относящиеся к обработчикам реального времени и обработчикам Windows NT. Прерывания, которые должны обрабатываться драйверами Windows NT, отправляются по стандартной цепочке. При этом Real-Time HAL следит за тем, чтобы прерывания не маскировались драйверами Windows NT более чем на 5 мкс, исключая возможность пропуска критического события.

Быстрые часы и таймерные службы. Для измерения временных интервалов или для генерации прерываний Real-Time HAL позволяет работать с тиккером, разрешение которого 1 мкс. Системный таймер синхронизирован с тиккером, и может работать с периодом 100 мкс или быстрее, обеспечивая работу как стандартных таймерных сервисов, так и дополнительных, входящих в состав подсистемы реального времени.

Поддержка подсистемы реального времени (RTSS). Кроме перечисленных выше функций (прерывания и таймеры), Real-Time HAL содержит поддержку функционирования всей подсистемы реального времени. Так, на основе прерываний от таймера строится диспетчер процессов реального времени, Real-Time HAL отвечает также за выполнение функций ввода-вывода подсистемы реального времени и пр.

Программный интерфейс реального времени RTAPI является расширением Win32 и содержит, прежде всего, набор функций, необходимых для управления устройствами. RTAPI реализован в двух видах – как подмножество подсистемы реального времени (RTSS) и как динамическая библиотека (DLL), которая может вызываться из Win32-приложений. RTAPI содержит следующие группы функций:

- управление процессами и потоками – предоставляет Win32-совместимый интерфейс для управления, создания, изменения приоритетов, профилирования и завершения потоков реального времени,
- управление объектами RTSS – предоставляет возможности унифицированного управления объектами RTSS (создание, закрытие, доступ). Объектами RTSS являются: таймеры, обработчики прерываний и исключительных ситуаций (startup, shutdown, blue screen), потоки, процессы, семафоры, мьютексы (mutex), разделяемая память, почтовые ящики, консольный и файловый ввод-вывод, регистры.

Взаимодействие между процессами. RTAPI использует семафоры, мьютексы и разделяемую память для взаимодействия как потоков реального времени между собой, так и для взаимодействия процессов реального времени с процессами WIN32.

Управление памятью позволяет фиксировать приложения в памяти, запрещая их выгрузку в файл подкачки.

Доступ к физической памяти: приложение пользователя получает возможность доступа к данным по физическим адресам памяти.

Управление прерываниями позволяет назначать и запрещать обработчики прерываний, разрешать и запрещать прерывания.

Управление часами и таймерами разрешает создавать, удалять, отменять, инициализировать таймеры, назначать обработчики прерываний.

Управление вводом-выводом RTAPI предоставляет два способа управления устройствами ввода-вывода. Во-первых, приложения пользователя получают возможность непосредственного доступа к адресам портов ввода-вывода, что позволяет программировать работу устройств напрямую. Кроме того, внешнее устройство может управляться специальными (легко разрабатываемыми) драйверами, для работы с которыми RTAPI предоставляет специальный интерфейс.

2.5.2. INtime

Система **INtime** является расширением реального времени Windows, которое было разработано корпорацией Radisys Corporation, а в настоящее время поддерживается корпорацией TenAsys [INTIME].

INtime комбинирует возможности ОСРВ жесткого реального времени со стандартными ОС Windows, включая Windows XP, Windows XP Embedded, Windows 2000, Windows NT и Windows NT Embedded, не требуя дополнительной аппаратуры. INtime специально разработана под архитектуру процессора x86. Приложения реального времени и не реального времени выполняются на разных виртуальных машинах на единственном компьютере (см. рис. 4).

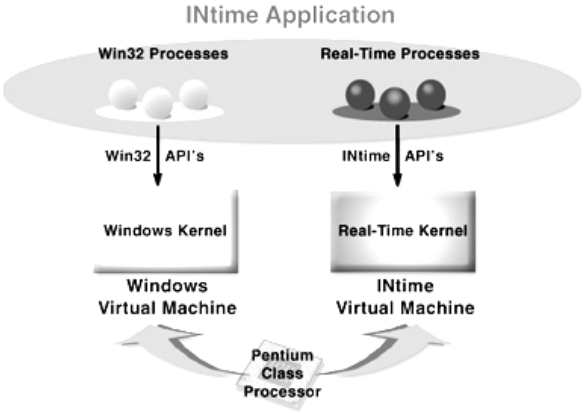

*Рис. 4. Структура **INtime**.*

INtime в отличие от RTX слабо связана с NT. Архитектура **INtime** основана на механизме аппаратного обслуживания задач (hardware tasking), которое обеспечивается процессором Intel. Получается, что два ядра выполняются на одной аппаратуре. Поскольку они разделяют одну аппаратуру, потребовались

некоторые модификации NT HAL. Такой подход позволяет защитить и отделить среду выполнения и область памяти от Windows. Внутри **INtime** каждый процесс приложения имеет свое собственное адресное пространство. Кроме того, ядро и приложения выполняются на разных приоритетных уровнях, что позволяет защитить их друг от друга.

INtime показывает предсказуемое поведение, однако ее сложная архитектура не позволяет достичь системе хорошей производительности. Из-за сегментационных ограничений **INtime** подходит не для всех систем реального времени.

2.5.3. Microsoft Windows Embedded

Операционные системы **Microsoft Windows Embedded** для встраиваемых систем имеют две разновидности в соответствии с версиями ОС Windows – NT и XP [MSEmb]. Версии систем Embedded корпорации Microsoft состоят из многочисленных конфигурируемых частей, которые позволяют легко манипулировать набором установленного программного обеспечения.

Windows NT Embedded использует технические ресурсы Windows NT и позволяет разрабатывать приложения, которые могут быть легко интегрированы в существующую информационную инфраструктуру.

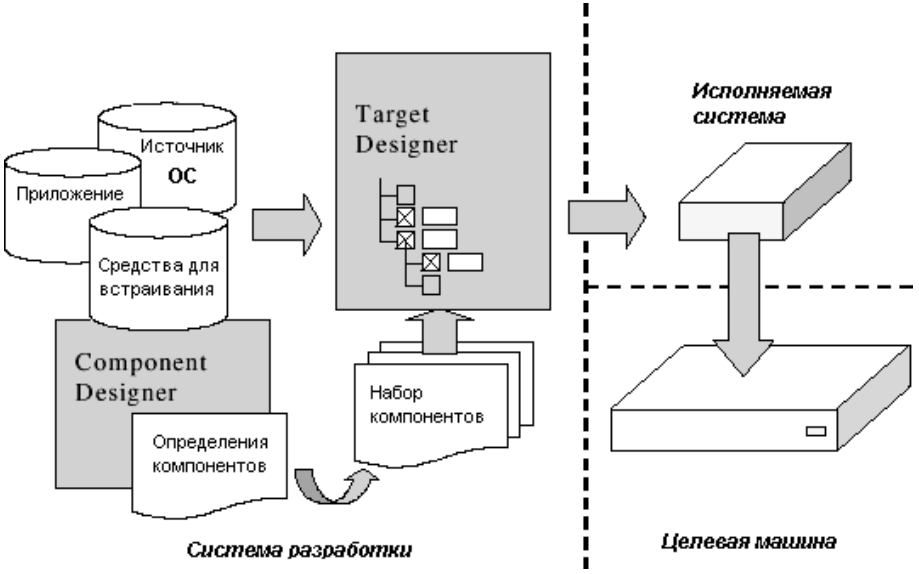

*Рис. 5. Процесс разработки встраиваемого ПО на основе **Windows NT Embedded**.*

Набор средств разработки – Target Designer и Component Designer – позволяет OEM (original equipment manufacturer) производителям конфигурировать и создавать операционную систему для конкретной аппаратной платформы. **Windows NT Embedded** обладает специфическими компонентами для создания

встраиваемых систем, которые позволяют работать в системах без видеоадаптера, осуществлять загрузку и работу накопителей в режиме "только чтение", выполнять удаленное администрирование и предоставляют дополнительные средства обработки ошибок и восстановления. **Windows NT Embedded** дает возможность создавать устройства, с которыми работать так же просто, как и со стандартными ПК на основе Windows, и управлять этими новыми устройствами на основе существующих профессиональных продуктов, таких как Microsoft Systems Management Сервер, HP OpenView, IBM Tivoli, CA Unicenter TNG, и др.

Разработчик встраиваемых систем использует Target Designer для конфигурирования ОС, используя готовый двоичный код Windows NT, дополнительные компоненты для встраивания и дополнительные приложения. В случае необходимости, Component Designer может использоваться для создания новых компонентов, не входящих в состав продукта (например, драйверов устройств, приложений и пр.). Вновь созданные новые компоненты могут быть импортированы в Target Designer и включены в состав целевой ОС. После конфигурирования ОС с помощью Target Designer происходит проверка взаимосвязей компонентов и строится образ системы, готовый к загрузке и исполнению на целевой системе.

Windows XP Embedded насчитывает до 10000 отдельных компонентов, а в **Windows NT Embedded** их было чуть больше 300. Основной отличительной чертой **Windows XP Embedded** является четкое разграничение компонент системы, что позволяет разработчикам встраиваемого набора функций при создании образа системы включать только необходимые файлы и максимально сократить размер результирующей системы. Этими компонентами служат отдельные части системы Windows XP Professional.

Компоненты **Windows XP Embedded** представлены сервисами, приложениями, библиотеками и драйверами – разработчику нужно сконфигурировать необходимый набор функций и собрать из компонент необходимую конфигурацию в образ среды исполнения (runtime image). Все опции конфигурации собраны воедино в базу данных компонент. Разработчик имеет к ней доступ и может ее редактировать с помощью специального инструмента – Component Database Manager.

Для каждого компонента в процессе создания определяется ряд параметров:

- платформа, на которой будет выполняться данный компонент, определяет порядок компиляции и сборки;
- описание и схема подключения компонента;
- список ассоциированных ресурсов, таких как файлы и ключи реестра;
- зависимости компонента от других компонент (например, от DirectX или NET runtime);
- указатель на хранилище файлов (чаще всего это просто локальный каталог, но может быть и сетевым ресурсом);

- принадлежность к группе для упрощения обращения сразу к нескольким компонентам как к целому.

Сама база данных представляет собой БД в терминах MS SQL и может быть расположена как локально, на компьютере разработчика, так и на сервере.

2.6. TinyOS

Разработка операционной системы **TinyOS** [HSW00] связана с появлением новой концепции беспроводной связи – Motes. Motes (в переводе с английского – пылинки, соринки) – это реализация идеи "smart-dust" ("распыленной разумности"), предложенной оборонным агентством Darpa (Defense Advanced Research Projects Agency) для отслеживания передвижений противника.

Motes разработаны Калифорнийским Университетом в Беркли совместно с Intel, и в настоящее время ведутся испытания этих самоорганизующихся сетей, построенных на основе открытых технологий Intel Mote и программного обеспечения **TinyOS**, TinyDB.

Умные сенсоры Motes, распределенные в пространстве, могут самостоятельно связываться друг с другом, образуя распределенную беспроводную информационную сеть. "Пылинка разума" состоит из 8-битового микро-контроллера семейства Amtel AVR, приемопередающего интегрального модуля TR1000 и двух микросхем средней степени интеграции – энергонезависимой памяти и дополнительного загрузочного микроконтроллера, позволяющего по радиоканалу обновлять ПО центрального процессора – AVR.

"Smart-dust" создавалась для динамических, изменяющихся как в пространстве, так и во времени сетей – для той области, в которой абсолютно неприменимы ни традиционные алгоритмы управления, ни отработанные принципы маршрутизации, ни архитектурные решения, лежащие в основе традиционного системного ПО. Стремление конструкторов сделать ее как можно более компактной (в перспективе – 1 мм3) влечет за собой ряд существенных ограничений, в первую очередь энергетических. Ограниченные вычислительные ресурсы и динамический характер сети приводят к тому, что функциональность "пылинки" надо время от времени изменять, что может быть достигнуто только одним способом – передачей по радиоканалу нужного ПО. С другой стороны, энергетическая дороговизна передачи информации требует сверхкомпактного представления передаваемого кода, в противном случае "пылинки" просто не будут работать из-за быстрого истощения крохотных автономных источников питания.

При проектировании ОС **TinyOS** основными требованиями являлись достижение энергетической эффективности и создание высокого уровня абстракции системных вызовов для упрощения разработки программ. Эта система обладает всеми отличительными признаками развитой ОС – в первую очередь, крайне простой, но достаточно развитой компонентной моделью. Однако специфика предназначения этой компонентной модели существенно отличается от традиционных разработок, поскольку главной целью компонентности является не облегчение подбора интерфейсов,

соответствующих требованиям запрашивающего компонента, а обеспечение развитых и надежных механизмов параллельного выполнения задач в условиях крайне ограниченных ресурсов.

Вышеописанные причины привели разработчиков **TinyOS** к выбору событийной модели, которая позволяет управлять высокой степенью параллельности выполнения задач в ограниченном пространстве памяти. Подход к управлению многопоточности, основанный на стеках, потребовал бы значительно больших ресурсов памяти, чем предполагалось в данном проекте. Для каждого контекста исполнения потребовалось бы выделение памяти для наихудшего варианта, либо нужно было бы применить какой-либо слишком изощренный и сложный метод управления памятью.

Архитектура **TinyOS** объединяет такое привычную составляющую, как планировщик задач (scheduler), и оригинальное понятие – компонентный граф. Термин "компонент" здесь одновременно и соответствует общепринятому пониманию, и существенно расширяет его. Так, интерфейс компонента **TinyOS** состоит из двух множеств – верхнего, предоставляемого им, и нижнего, требуемого для его функционирования. Каждое из этих множеств содержит описания команд и событий – синхронных и асинхронных процессов.

Согласно описанию системы компонент имеет 4 взаимосвязанные части – набор команд, набор обработчиков событий, инкапсулированный фрейм фиксированного размера и пучок простых потоков. Потоки, команды и обработчики событий выполняются в контексте данного фрейма и воздействуют на его состояние. Кроме того, каждый компонент декларирует команды, которые он использует и события, о которых он сигнализирует. Эти декларации используются при компоновке для конфигурации системы, настроенной на определенный класс приложений. Процесс композиции создает слои компонентов, где каждый более высокий уровень выдает команды к нижележащему уровню, а нижележащий уровень обращается к более высокому с помощью сигналов, что понимается в данной системе как события. Аппаратное обеспечение является самым низким слоем компонентов.

Написана **TinyOS** с использованием структурированного подмножества языка C. Использование статического распределения памяти позволяет определять требования к памяти на уровне компиляции и избегать накладных расходов, связанных с динамическим распределением. Кроме того, этот подход позволяет сокращать время выполнения благодаря статическому размещению переменных во время компиляции вместо доступа к ним по указателю во время выполнения.

Команды являются неблокируемыми запросами к компонентам нижележащего слоя. Команда сохраняет необходимые параметры в своем фрейме и может инициировать поток для его последующего выполнения. Для того чтобы не возникало неопределенных задержек, время ответа от вызванной команды не должно превышать заданного интервала времени, при этом команда должна вернуть статус, указывающий успешно она завершилась или нет. Команда не может подавать сигналы о событиях.

Обработчики событий прямо или косвенно имеют дело с аппаратными событиями. Самый нижний слой компонент содержит обработчики, непосредственно связанные с аппаратными прерываниями. Обработчик событий может положить информацию в свой фрейм, запустить потоки, подать сигнал вышележащему уровню о событиях или вызвать команды нижележащего слоя. Аппаратное событие инициирует фонтан обработки, которая распространяется вверх по уровням через события и может вернуться вниз через команды. Для того, чтобы избежать циклов в цепочке команд/событий, команды не могут подавать сигналы о событиях. Как команды, так и события предназначены для выполнения небольшой, строго фиксированной порции обработки, которая возникает внутри контекста выполняющегося потока.

Основная работа возлагается на потоки. Потоки в **TinyOS** являются атомарными, и в отличие от потоков в других ОС выполняются до завершения, хотя они и могут быть вытеснены событиями. Потоки могут вызывать команды нижележащего уровня, сигнализировать о событиях более высокому уровню и планировать другие потоки внутри компонента. Семантика потока “выполнение до завершения” позволяет иметь один стек, который выделяется выполняющемуся потоку, что очень существенно в системах с ограниченной памятью. Потоки дают возможность симулировать параллельную обработку внутри каждого компонента, т.к. они выполняются асинхронно по отношению к событиям. Однако потоки не должны блокироваться или простаивать в ожидании, потому в таких случаях они будут препятствовать развитию обработки в других компонентах. Пучки потоков обеспечивают средство для встраивания произвольных вычислительных обработок в модель, управляемую событиями.

В системе предусмотрена также отдельная абстракция задачи, понимаемая как продолжительный вычислительный процесс. Взаимоотношение между понятиями “команда” и “задача” следующее: команда – это атомарная составляющая задачи. Команда ставится в очередь на исполнение планировщика, затем она выполняется и может быть временно прервана обработкой события.

Планировщик работает по принципу очереди FIFO, т.е. для передачи управления следующей задаче требуется полное завершение предыдущей. Однако в зависимости от требований приложения могут использоваться и более сложные механизмы планирования, основанные на приоритетах или на дедлайнах. Ключевым моментом является то, что планировщик ориентирован на энергосбережение: процессор засыпает, если очередь планировщика пуста, а периферийные устройства работают, и каждое из них может разбудить систему. Когда очередь становится пустой, новый поток может быть запущен на исполнение только как результат какого-либо события, которое может возникнуть только в аппаратных устройствах. Планировщик имеет крайне малые размеры – всего 178 байтов, данные планировщика занимают только 16 байтов.

В **TinyOS** полностью отсутствуют механизмы блокирования исполнения, что означает необходимость введения индикации завершения продолжительной операции соответствующим асинхронным событием. Традиционные приемы построения ОС реального времени и привычные отработанные архитектурные решения здесь оказались неприменимы. В результате вся ОС и ее компоненты построены по принципу конечных автоматов – переходов из состояния в состояние.

Итак, **TinyOS** состоит из набора компонентов (каждый размером примерно 200 байт), из которых разработчики собирают систему для каждого конкретного сенсора. Для компоновки системы из набора компонентов, которые статически линкуются с ядром, используется специальный описательный язык. После проведения компоновки модификация системы не возможна.

Для того, чтобы обеспечить динамичность во время выполнения была разработана виртуальная машина, которая является надстройкой над ОС **TinyOS**. Эта виртуальная машина решает проблему потребление – компактность представления программ. Код для виртуальной машины можно загру-зить в систему во время выполнения. Для работы этой виртуальной машины необходимы 600 байт оперативной памяти и менее 8 KB памяти команд 8-битового микроконтроллера. Программы виртуальной машины представляются 8-битовыми инструкциями всего трех типов, объединяемых в "капсулы" – атомарные последовательности не более чем двадцати четырех инструкций.

Иерархическая структура сети получается автоматически благодаря тому, что все сенсоры следуют простым правилам, заложенным в **TinyOS**. Правила эти, к примеру, определяют способ поиска кратчайшего пути до ближайшего стационарного узла, а уже в зависимости от того, где и как расположены сенсоры, сеть принимает привычную для системных администраторов древообразную форму. В **TinyOS** учитывается также и то, что некоторые виды сенсоров могут работать от солнечных батарей или иных источников энергии, зависящих от погоды, поэтому при потере связи с ближайшим узлом сети происходит смена маршрута, по которому пересылаются пакеты.

2.7. OSEK/VDX

Как уже упоминалось в разделе о стандартах, **OSEK/VDX** является комбинацией стандартов компьютерных систем реального времени, разработанных консорциумами **OSEK** и **VDX** для автомобильной промышленности. В данной работе рассматривается только стандарт **OSEK**, касающийся архитектуры операционной системы.

ОС **OSEK** оперирует такими объектами, как задачи, события, ресурсы. Кроме того, обеспечиваются такие возможности, как управление ошибками и средства для пользовательских функций отслеживания изменений в состоянии системы.

ОС **OSEK** обеспечивает определенный набор интерфейсов для пользователя. Интерфейсы используются сущностями, конкурирующими за центральный процессор. ОС **OSEK** оперирует двумя типами таких сущностей – задачи и прерывания – и определяет три уровня обработки – уровень прерываний,

логический уровень планировщика и уровень задач. Задачи выбираются на выполнение в соответствии с присвоенным им приоритетам.

Задача в ОС **OSEK** может быть

- базовой или расширенной,
- вытесняемой или невытесняемой.

Главным различием между базовой и расширенной задачами заключается в том, может ли она впасть в состояние ожидания (в котором она ждет появления события). Только расширенная задача может ожидать события. Вытесняемая задача может быть вытеснена задачей более высокого приоритета или прервана прерыванием. Невытесняемая задача может быть вытеснена только с помощью прерывания (когда прерывания не запрещены).

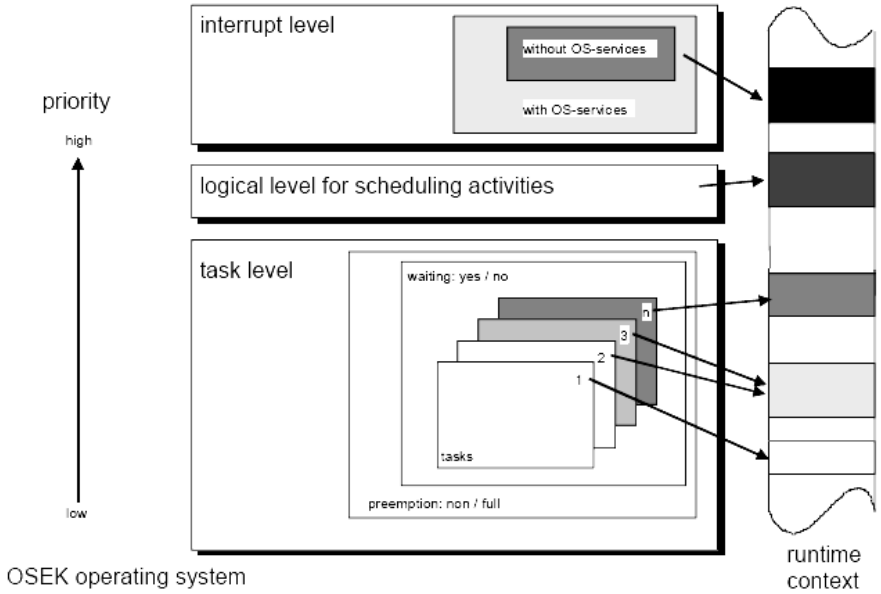

*Рис. 6. Уровни обработки в ОС **OSEK.***

Концепция двух типов задач потребовала введения нового понятия – класс соответствия (conformance class) для описания своеобразной реализации ОС **OSEK** и системных сервисов. Определяются четыре класса соответствия – два для базового соответствия (BCC1 и BCC2 – Basic conformance Classes 1 и 2) и два для расширенного (ECC1 и ECC2 – Extended Conformance Classes 1 и 2). Реализации, которые соответствуют базовым классам, требуют использования только базовых задач, в то время как расширенные классы требуют как расширенных, так и базовых задач. Числа 1 и 2 в именах указывают количество запросов на задачу для базовых задач и количество задач на приоритет для всех задач. Таким образом, BCC1 и ECC1 имеют только одну задачу на приоритет и базовые задачи могут быть запрошены только один раз. BCC2 и ECC2

допускают множественность задач на приоритет и множественное запрашивание базовых задач.

Каждая задача должна находиться в одном из четырех состояний:

- Выполняющаяся – только одна задача может быть в этом состоянии,
- Готовая к выполнению – планировщик может выбрать ее на выполнение на основании приоритетов и правил вытеснения,
- Ожидающая – задача ждет появления события,
- Приостановленная – задача в пассивном состоянии и ждет активации.

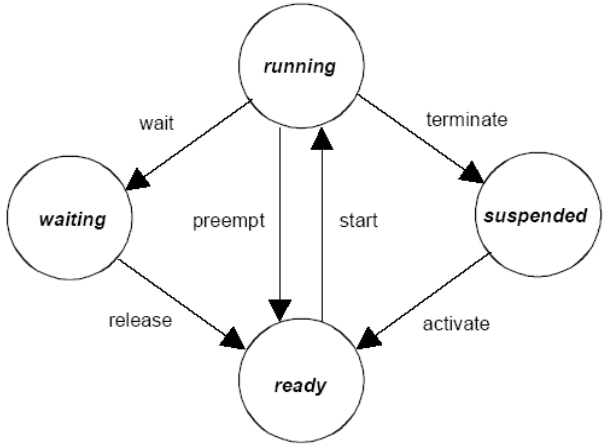

*Рис. 7. Модель состояний задачи в ОС **OSEK**.*

Каждая задача имеет приоритет. Стандарт ОС **OSEK** не ограничивает максимальное количество приоритетов – это определяет реализация.

ОС **OSEK** определяет два уровня программ управления прерываниями, которые различаются возможностями вызова системных сервисов. Прерывания уровня 1 выполняются независимо от ОС очень быстро. Уровень 2 обеспечивает выполнение функций приложений, которые содержат вызовы ОС.

События в ОС **OSEK** используются для синхронизации различных задач. События являются собственностью задач. Любая задача, в том числе и базовая, может установить событие, и только собственник события может ожидать или снять его.

Управление ресурсами обеспечивает доступ к разделяемым ресурсам, таким как память, аппаратура и т.п. Планировщик также считается специальным ресурсом, который может быть захвачен задачами. Для того, чтобы избежать инверсии приоритетов и тупиковые ситуации, **OSEK** применяет потолочный протокол приоритетов. Согласно этому протоколу задаче, захватившей ресурс, временно повышается приоритет, и таким образом, никакие другие задачи, обращающиеся к данному ресурсу, не смогут выполняться до тех пор, пока ресурс остается захваченным. Однако, все задачи с более высоким

приоритетом, чем приоритет задачи, захватившей ресурс, все еще могут выполняться.

Аварийные сигналы и счетчики в **OSEK** используются для синхронизации активации задач с повторяющимися событиями. Аварийный сигнал статически присваивается счетчику, задаче и воздействию. Воздействие может либо активировать задачу, либо установить событие. Счетчики оперируют тактами и могут представлять время, количество принятых импульсов и т.п. Каждая реализация обеспечивает один временной счетчик, который используется для планирования периодических событий. Все другие счетчики управляются через API, и являются специфическими для конкретной реализации и не могут быть переносимыми.

В **OSEK** существует два типа аварийных сигналов: циклические и одинарные. Циклические аварийные сигналы применяются для диспетчеризации задачи, которая должна запускаться периодически. Счетчик аварийного сигнала может быть установлен в относительное или абсолютное значение. Параметры цикла и значение счетчика могут переустанавливаться динамически.

ОС **OSEK** обеспечивает минимальные средства для управления ошибками времени выполнения. Однако, есть возможность дополнительного управления ошибками во время разработки благодаря расширенной функциональности возврата управления. Причина такого решения состоит в том, что после того как продукт запущен в производство, большинство возможных ошибок могут быть выявлены во время тестирования (такие как “неверный идентификатор задачи”, “ресурс занят”, “непредусмотренный вызов с уровня прерываний” и т.д.). Во время выполнения большинство системных сервисов не возвращают ошибки, но некоторые сервисы, такие как аварийные сигналы, которые могут стартовать и останавливаться динамически, возвращают ошибку, если данный аварийный сигнал уже использовался.

ОС **OSEK** определяет два типа ошибок – ошибки приложения и фатальные ошибки. При ошибке приложения целостность внутренних данных все еще сохраняется, в то время как приложение пытается выполнить несанкционированную операцию (например, активизировать несуществующую задачу). Фатальные ошибки возникают, если ОС обнаруживает нарушение целостности внутренних данных. Такие ошибки вызывают сервис завершения работы ОС.

2.8. OSE RTOS

Операционная система реального времени **OSE RTOS**, разработанная в корпорации ENEA, имеет ядро с приоритетным планированием [OSERTOS]. Это ядро сильно оптимизировано для обеспечения высокой производительности и достаточно компактно для использования во встраиваемых системах. **OSE** имеет архитектуру, управляемую сообщениями, с простыми системными вызовами. Передача сообщений в **OSE** служит концептуальным шлюзом в распределенных многопроцессорных встраиваемых системах. Задачи посылают сообщения друг другу напрямую через ОС без поддержки очередей, почтовых ящиков или других промежуточных механизмов. **OSE RTOS** поддерживает подкачку,

дублирование, динамическое обновление кода и многие коммуникационные протоколы.

OSE RTOS предлагает три варианта ядра, построенные по одному принципу. **OSE** Epsilon – для глубоко встраиваемой и SoC (system-on-chip) разработки. OSEck – компактное ядро для DSP. OSE Link Handler – для многочисленных смешанных CPU/DSP проектов. Все они имеют очень маленькое количество системных вызовов – от шести до восьми.

Архитектура **OSE RTOS** основана на многослойной модели (рис.8).

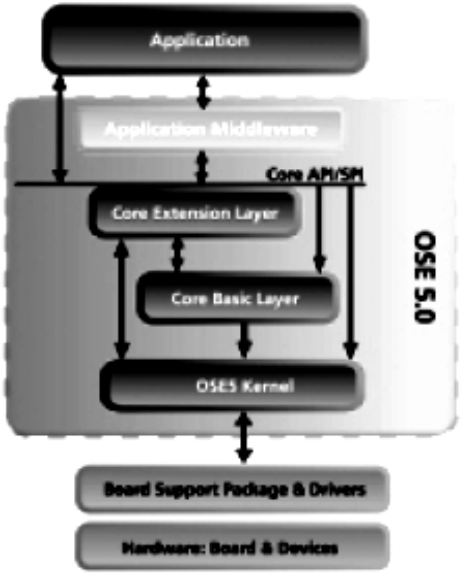

*Рис. 8. Многослойная архитектура **OSE RTOS**.*

Единицей выполнения в **OSE RTOS** является процесс. Процессы могут быть сгруппированы в блок, который может иметь собственный пул памяти. В ядре **OSE RTOS** адресное пространство принадлежит сегменту, который может включать один или больше блоков. Отображение блоков в сегменты и отображение пулов в регионы дает возможность достичь полной защиты памяти и изоляции программы. Блоки и пулы могут размещаться в одном или нескольких сегментах.

OSE RTOS оперирует разными типами и категориями процессов.

Типы процессов:

- процессы прерывания возникают в ответ на аппаратные или программные прерывания, выполняются до конца, имеют самый высокий приоритет и такой же контекст, как и все другие процессы,

- таймерные процессы прерывания аналогичны процессам прерывания, за исключением того, что они предусматриваются планировщиком периодически в соответствии с указанным периодов времени,
- приоритетные процессы являются самыми распространенными процессами в **OSE RTOS** и выполняются до тех пор, пока не будут вытеснены процессом прерывания или процессом с более высоким приоритетом,
- фоновые процессы выполняются строго в режиме циклического обслуживания с квантованием времени на приоритетном уровне, который находится ниже всех приоритетных процессов.

Под категориями процессов в **OSE RTOS** понимается разделение процессов на динамические и статические. Статические процессы создаются ядром, когда система стартует, и существуют на всем протяжении существования системы. Динамические процессы создаются и уничтожаются во время выполнения.

Источником потенциальных возможностей **OSE RTOS** является механизм прямой передачи сообщений. Сообщение, посланное одним процессом другому, содержит идентификатор, адреса отправителя и получателя и данные. Как только сообщение послано, отправитель уже не имеет к нему доступа, т.е. собственность сообщения никогда не разделяется. Это важное свойство исключает конфликты доступа к памяти. Прямая передача сообщений концептуально более проста, чем стандартная косвенная модель, а уникальная разработка такой передачи оказалась чрезвычайно эффективной.

2.9. Contiki

Операционная система **Contiki** [DGV04] разработана в Швеции (Swedish Institute of Computer Science) для систем с ограниченной памятью. Система **Contiki** позволяет динамически загружать и отгружать приложения и сервисы. С целью минимизации размеров операционной системы было спроектировано ядро **Contiki**, которое основано на модели управления событиями [HSW00].

В традиционных системах, управляемых событиями, процессы моделируются как обработчики событий, которые выполняются до завершения. Поскольку обработчик событий не может быть заблокирован, все процессы могут использовать один и тот же стек, разделяя дефицитные ресурсы памяти. К тому же не нужны механизмы блокировки, т.к. два обработчика событий никогда не выполняются параллельно. В ОС, управляемой событиями, длинные обработки монополизируют центральный процессор, не давая возможности реагировать на происходящие внешние события. Однако, если ОС снабжена механизмом многопоточной обработки с прерываниями, этот недостаток сглаживается, что и сделано в **Contiki**.

Многопоточный режим с приоритетами в системе **Contiki** реализован с помощью библиотеки приложений, которые выполняются над ядром, управляемым событиями. Приложения, обеспечивающие многопоточную обработку, компонуются с выполняющимся приложением по мере необходимости, т.е. если оно явно требует многопоточной модели вычислений.

Выполняющаяся система **Contiki** разделяется на две части – сердцевину (core) и загруженные программы. Сердцевина (core) состоит из собственно ядра (kernel), базовых сервисов и фрагментов библиотек поддержки, в том числе языковой поддержки времени выполнения. Разделяемая функциональность реализуется через сервисы как некоторая форма разделяемых библиотек. Эти сервисы можно обновлять или замещать динамически независимо друг от друга во время выполнения, что по мнению разработчиков ведет к гибкой структуре системы.

Реализация **Contiki** показала, что многопоточная обработка с приоритетами необязательно должна быть упрятана на самый нижний приоритетный уровень ядра, а может быть реализована как библиотека приложений над ядром, управляемым событиями. Такой подход позволяет выполнять потоковые программы над ядром без накладных расходов реентерабельности или многочисленных стеков во всех частях системы.

Системы, управляемые событиями, имеют свои проблемы. Модель программирования, управляемая состояниями, сложна для программистов. К тому же не все программы укладываются в конечно-автоматную модель.

Contiki не поддерживает никаких механизмов защиты, т.к. аппаратура, для которой она проектировалась, не поддерживает защиту памяти.

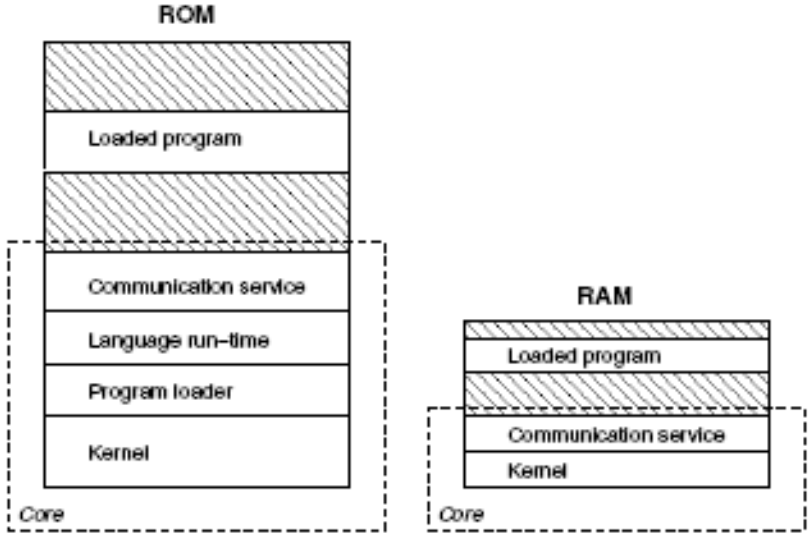

*Рис. 9. Сердцевина **Contiki** и загруженные программы.*

Что касается архитектуры ядра ОС **Contiki**, то ядро этой системы состоит из облегченного планировщика, который осуществляет диспетчеризацию событий для выполняющихся процессов и периодически вызывает обработчики опроса процессов. Выполнение программы переключается либо в соответствии с

событиями, регулируемыми ядром, либо через механизм опроса. Если для обработки был выбран обработчик события, ядро не прерывает его работу до тех пор, пока он не завершится. Однако, обработчики событий могут использовать внутренние механизмы для выполнения прерывания. Ядро поддерживает два вида событий – асинхронные и синхронные. Асинхронные события являются некоторой формой отложенного вызова процедуры – асинхронные события ядро ставит в очередь, и они направляются целевому процессу некоторое время спустя. Синхронные события обрабатываются почти также как асинхронные, только направляются целевому процессу сразу. Управление возвращается посылающему процессу только после того, как целевой процесс завершил обработку события. Это можно рассматривать как вызов процедуры внутри процесса.

Contiki написана на языке C и адаптирована для ряда микроконтроллерных архитектур, включая Texas Instruments MSP430 и Atmel AVR, а также для платформы ESB.

2.10. pSOS

ОСРВ **pSOS** была разработана корпорацией Integrated Systems. В настоящее время она принадлежит корпорации **WindRiver** [PSOS], которая ее купила, видимо для того, чтобы она не мешалась на рынке сбыта ОСРВ.

Имя **pSOSsystem** присвоено операционной системе, имя **pSOS+** – ее ядру. **pRISM+** – это интегрированная среда разработки для создания приложений.

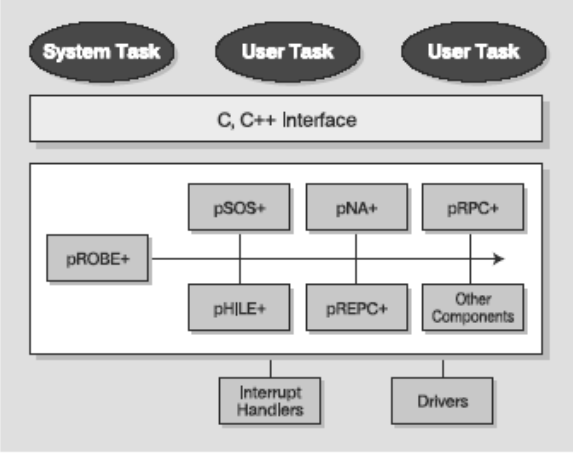

*Рис. 10. Компоненты **pSOSsystem**.*

pSOS+ – это маленькое ядро встраиваемых приложений, представляющее собой некий вариант клиент-серверной архитектуры. Однако оно не имеет протокола взаимодействия, основанного на сообщениях. Для взаимодействия модулей используется программная шина (software bus). Есть возможность

выбрать и встроить модули в систему во время компиляции. Такими модулями могут быть файловая система (pHILE+), отладчик (pROBE+), сетевые протоколы (pNA+), библиотека удаленных вызовов процедур (pRPC+) и стандартная библиотека ANSI C (pREPC+). Эти компоненты показаны на рис. 10.

Вызовы различных приложений осуществляются через программные прерывания.

pSOS+m является многопроцессорной версией ядра **pSOS+**. Она требует, чтобы один узел был главным, а остальные – подчиненными. К этому ядру добавлены системные вызовы, позволяющие оперировать через границы процессора.

pSOS+ не имеет понятия процесса, вместо них она оперирует задачами, что соответствует понятию потоков, выполняющихся в одном процессе. Все системные объекты разделяются между всеми потоками. Так как все потоки разделяют один и тот же контекст, время переключения потоков становится очень малым.

pSOSsystem имеет несегментированную модель памяти. Защита памяти может быть обеспечена через библиотеку управления памятью. Код, данные и стеки можно защитить с помощью определения отображений защиты памяти для каждой задачи. При этом ответственность ложится на разработчика приложений, а это является непростой задачей. pSOSsystem предлагает две абстракции для управления памятью – регионы и разделы. Регионы – это куски памяти нефиксированного размера, в то время как разделы – куски фиксированного размера. Управление памятью с помощью разделов обеспечивает быстрое выделение памяти.

Управление прерываниями в pSOSsystem довольно примитивное. Кроме того, отсутствуют мьютексы и механизм наследования приоритетов, что может привести к инверсии приоритетов.

2.11. INTEGRITY

Продукт **INTEGRITY** (компания Green Hills Software) [INTEGRITY] – это ОСРВ с предсказуемым временем отклика, рассчитанная на применение в тех ситуациях, когда необходимы масштабируемость ОС, её компактность и возможность работы в режиме реального времени. Платформа **INTEGRITY** построена на базе микроядра velOSity [Velosity] и хорошо подходит для использования в недорогих устройствах с ограниченными аппаратными ресурсами (сюда относится большая часть потребительской электроники). Для своей операционной системы компания Green Hills предлагает интегрированную среду разработки MULTI, полностью автоматизирующую процесс создания ПО. Поддерживая многоязыковую разработку и отладку, графический интерфейс пакета MULTI дает пользователю быстрый и удобный доступ к оптимизирующим C/C++ компиляторам и функциям MISRA C. В этом инструментальном пакете содержится отладчик уровня входного языка, компоновщик, анализатор событий, профилировщик производительности,

программа обнаружения ошибок периода исполнения и средство отладки, не нарушающее основного режима функционирования.

Объектно-ориентированный подход к проектированию **INTEGRITY** обеспечивает строгий контроль доступа и верификацию безопасности и целостности данных, взаимодействий, компонент и системы в целом. **INTEGRITY** использует аппаратную защиту памяти и обеспечивает поддержку многочисленных защищенных виртуальных адресных пространств, каждое из которых может содержать несколько задач приложения. Ядро **INTEGRITY** оперирует в своем собственном защищенном адресном пространстве.

Для управления памятью **INTEGRITY** использует механизм виртуальной памяти. Для того, чтобы гарантировать абсолютное минимальное время обработки прерываний, ядро никогда не блокирует прерывание, даже при обработке критических структур данных. Ядро также избегает длинных обработок прерываний. В качестве примера таких прерываний упоминаются операции деления и обработки строк.

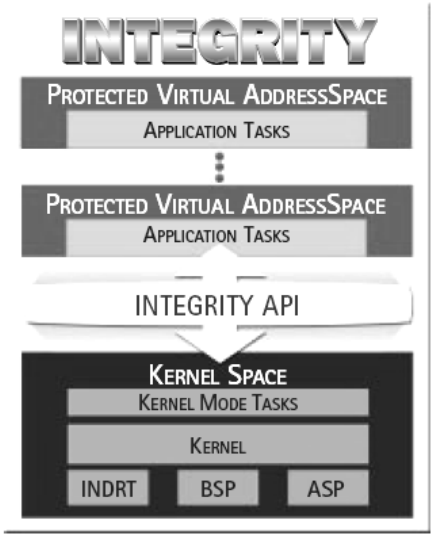

*Рис. 11. Структура **INTEGRITY.***

ОСРВ **INTEGRITY** включает двухуровневый планировщик ARINC-653, основанный на сегментации (Partition Scheduler), который обеспечивает гарантированное временное окно центрального процессора для каждой выполняющейся задачи. Например, если выполняются две задачи, A и B, и каждой предоставлено по 50% времени, то порождение задачей B задач B1 и B2 не повлияет на выполнение задачи A, поскольку время центрального процессора, выделенного для задачи B, – 50%, разделится на 3 для задач B, B1 и B2, а для задачи A останутся ее прежние 50%. Таким образом, действия

одной задачи никогда не смогут повлиять на выполнение других задач, что позволяет избегать воздействия злоумышленного кода, вирусов, проникновения хакера или просто ошибок в других адресных пространствах.

2.12. LynxOS

Операционная система **LynxOS® RTOS** (LynuxWorks, Inc.) является операционной системой жесткого реального времени, которая предназначена для специализированной и телекоммуникационной аппаратуры [LynxOS]. Эта ОС является полностью детерминированной и обладает POSIX-, UNIX- и Linux-совместимостью. Областями применения ОС **LynxOS** являются также сложные системы безопасности.

Последняя выпущенная версия этого бренда ОС **LynxOS-178 2.0** характеризуется производителем как коммерческая операционная система, обеспечивающая высокий уровень надежности и оперативности, необходимый для встраиваемых приложений с особыми требованиями к безопасности. В **LynxOS-178 2.0** реализована поддержка интерфейса APEX (APlication/EXecutive – интерфейс приложения/управляющей программы) спецификации ARINC-653. Это означает, что данная операционная система отвечает самым строгим требованиям к безопасности и надежности электронных систем для военной и гражданской авиации. Система **LynxOS-178 2.0** полностью соответствует положениям уровня A спецификации DO-178B.

ОСРВ **LynxOS-178 2.0** соответствует требованиям стандартов POSIX и ARINC-653, а также DO-178B, что означает гарантию переносимости прикладного кода встраиваемых систем, многократного использования созданных программ, а также соответствие самым строгим нормативам операционных систем с повышенными требованиями к безопасности. Использование **LynxOS-178 2.**0 позволяет применять любые ранее сертифицированные программы и разработки.

2.13. Microware OS-9

Операционная система реального времени **OS-9** корпорации Microware System является многозадачной, многопользовательской операционной системой для встраиваемых приложений, работающих в режиме реального времени [OS-9]. Эта система предназначена для работы в таких системах, как мобильные телекоммуникационные устройства, встраиваемые терминалы доступа в Интернет, интерактивные цифровые телевизионные приставки. **OS-9** работает на таких процессорах, как Motorola 68K, ARM/StrongARM, Intel® IXP1200 Network Processor, MIPS, PowerPC, Hitachi SuperH, x86 or Intel® Pentium®, Intel® IXC1100 XScale®.

Ядро **OS-9** является масштабируемым, полностью вытесняемым, поддерживает функционирование до 65535 процессов, предоставляет 65535 уровней приоритета и обеспечивает работу до 255 пользователей. Ядро **OS-9** содержит более 90 системных вызовов, которые дают возможность управлять динамическим режимом диспетчеризации, распределением памяти,

межпроцессорной коммуникацией и т.д. – вплоть до управления встраиваемым в ядро ОС режимом экономичного потребления питания. Характеристики производительности ядра: 5,6 мкс – время задержки прерывания (Interrupt Latence Time), 14 мкс – время переключения контекста процесса (для процессора MC68040, 30MHz).

Система ввода-вывода ОС поддерживает следующие форматы устройств массовой памяти и основных интерфейсов периферийных устройств: Raw, MS-DOS, True FFS, CardSoft PCMCIA, USB, IrDA.

Среда **OS-9** поддерживает несколько программных коммуникационных платформ – mwSoftStax (Microware), Harris & Jeffries, Trillium. Благодаря наличию стандартизованной коммуникационной среды, в OS-9 доступны современные и наиболее перспективные коммуникационные протоколы: ISDN, ATM, X.25, MPEG-2, FR, SS7 и т.д.

Графические средства в **OS-9** представлены разнообразными продуктами – от компактных минимизированных по ресурсам программных модулей поддержки графики Multimedia Applications User Interface (MAUI) фирмы Microware до полнофункциональных клиент-серверных графических систем G-Windows (GESPAC), XiBase9 GUI (XiSys), MGR (Reccoware).

Корпорация Microware одной из первых лицензировала Java для встраиваемых приложений и является лидером по предложению разнообразных средств и приложений в рамках **OS-9** для различных классов устройств. В **OS-9** пользователю предлагается Java VM, Java-Compiler/JIT, Java-ROMizer, Java Applets Lib, Embedded Java, Personal Java.

В различных областях применения для портирования **OS-9** на аппаратную платформу производителя используются следующие программные пакеты:

- **OS-9** for Embedded Systems Kit,
- **OS-9** for Communications Systems,
- **OS-9** for Consumer Devices (Wireless Devices),
- **OS-9** for Interactive Digital TV,
- **OS-9** Java Starter Kit.

В качестве интегрированной кросс-среды разработки приложений для **OS-9** корпорация Microware разработала среду Hawk, которая функционирует на платформе MS Windows NT. Hawk является открытой средой и предоставляет сторонним разработчикам инструментальных средств более сотни API, позволяющих включать в состав среды Hawk продукты известных фирм разработчиков инструментального ПО.

Для нужд совместной программно-аппаратной разработки в Hawk встроены средства для работы с внутрисхемными эмуляторами серии visionICE фирмы EST. Есть средства отладки в режиме реального времени.

Для тестирования и верификации ПО разработано средство верификации программного обеспечения CodeTEST (Applied Microsystems), встраиваемое в Hawk. Это средство дает возможность осуществлять трассировку

встраиваемого ПО и контролировать его характеристики, а также ход выполнения тестов и распределение памяти.

2.14. GRACE-OS

Система **GRACE-OS** представляет собой планировщик CPU в режиме мягкого реального времени для мобильных устройств, выполняющих, главным образом, мультимедийные приложения [YN03]. Система **GRACE-OS** разработана в Иллинойском университете (University of Illinois, Department of Computer Science). При проектировании системы первоочередными целями ставились задачи поддержки качества сервиса и сбережения энергии. Для достижения поставленных целей **GRACE-OS** интегрирует динамическое масштабирование напряжения в диспетчеризацию на основе модели мягкого реального времени и определяет, как быстро, когда и как долго должно осуществляться выполнение приложений. Планировщик **GRACE-OS** реализован внутри ядра Linux, и апробирован на лэптопе HP Pavilion.

Планировщик **GRACE-OS** состоит из трех основных компонентов – профайлера, планировщика SRT (soft real-time) и адаптера скорости, как показано на рис. 12.

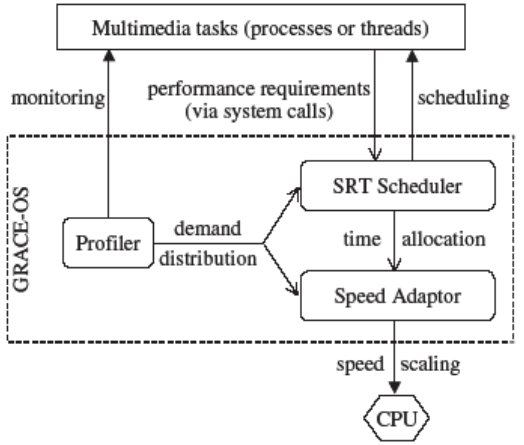

*Рис. 12. Архитектура **GRACE-OS***

Усовершенствованный планировщик выполняет планирование в режиме мягкого реального времени и динамическое масштабирование напряжения.

Профайлер осуществляет мониторинг коэффициента загрузки цикла отдельных задач и автоматически получает распределение вероятности их запросов внутри цикла в зависимости от коэффициента загрузки. Планировщик SRT отвечает за выделение циклов задачам и их планирование, обеспечивая необходимую производительность. Планирование в режиме мягкого реального

времени основано на статистических требованиях производительности и распределении запросов каждой задачи. Адаптер скорости динамически регулирует скорость CPU, обеспечивая экономию энергии. Он адаптирует скорость выполнения каждой задачи на основе распределения выделяемого задачам времени, что обеспечивается планировщиком SRT, и распределения запросов, что обеспечивается профайлером.

2.15. C EXECUTIVE

C EXECUTIVE (JMI Software Systems, INC.) [CEXEC] – это многозадачное ядро реального времени для встраиваемых систем, работающее на 8-, 16- и 32-битовых CISC процессорах, на широком диапазоне RISC процессоров и DSP (Digital Signal Processor). Это ядро обеспечивает быстрое переключение контекста, имеет маленький размер. Над ядром можно надстраивать DOS-совместимую файловую систему, TCP/IP и SNMP.

Ядро **C EXECUTIVE** обладает высокой степенью масштабируемости, можно даже сказать, что масштабируемость внутренне присуща такому ядру, поскольку набор системных вызовов компонуется из библиотеки во время создания системы, и выполняющийся экземпляр системы будет содержать только те системные вызовы, которые используются конкретным приложением. К тому же такое ядро можно конфигурировать с или без квантования времени, генератора тактовых импульсов, сигналов и т.п., таким образом, позволяя осуществлять крайне высокую оптимизацию системной конфигурации для небольших целевых систем.

Ядра реального времени компании JMI применяются в сотнях встраиваемых приложений, включая лазерные принтеры, электронные кассовые аппараты, медицинскую аппаратуру, устройства коммуникации, военные и космические приложения и другие критические по времени системы.

2.16. CMX-RTX

Операционная система **CMX-RTX** [CMXRTX] является многозадачной операционной системой реального времени для микроконтроллеров, микропроцессоров, микрокомпьютеров и DSP (Digital Signal Processor). Эта система поддерживает вложенные прерывания, имеет быстрое время переключения контекстов, низкие времена задержек прерываний и имеет крайне малые размеры. Планировщик задач и компонент управления прерываниями написаны на языке ассемблера для ускорения вычислительного процесса. **CMX-RTX** имеет компоненты управления задачами, событиями, временем, сообщениями, очередями, ресурсами, семафорами, фиксированными блоками памяти, автоматическим выключением питания, асинхронной последовательной передачей данных (UART – universal asynchronous receiver-transmitter), приоритетными прерываниями.

2.16.1. CMX-TINY+

CMX-TINY+ [CMXTINY] является многозадачной операционной системой реального времени для широкого ряда микропроцессоров и микрокомпьютеров, которая создана для разработки приложений, выполняющихся под ОСРВ и использующих только встраиваемую память процессора. Эта система обеспечивает незначительно меньшую функциональность, чем система **CMX-RTX**. Она создавалась для того, чтобы ее можно было поместить внутри небольшой бортовой памяти RAM (random access memory) в чипе, которая имеет размер 512 байтов и больше.

2.17. Inferno

Inferno (корпорация Lucent) – это компактная операционная система, созданная для построения распределенных и сетевых систем на широком диапазоне устройств и платформ [INFERNO]. Эта система обладает межплатформенной переносимостью и может выполняться как пользовательское приложение или как независимая операционная система. Поддерживается для большинства широко распространенных операционных систем и платформ. Каждая система **Inferno** предоставляет пользователю идентичную среду разработки независимо от основной операционной системы или архитектуры, разрешая работать в гомогенной среде с множеством различных платформ.

Inferno – это не только операционная система, но она также является полноценной средой разработки, обеспечивая все средства, необходимые для создания, отладки и тестирования приложений. Приложения, создаваемые в среде **Inferno** пишутся на языке Limbo, который является модульным параллельным языком программирования с С-подобным синтаксисом. Код на Limbo компилируется в архитектурно-независимый байтовый код, который затем может быть выполнен в режиме интерпретации (или код компилируется оперативно) для целевого процессора. Таким образом, Inferno-приложения выполняются идентично на всех Inferno-платформах.

Inferno предлагает полную прозрачность ресурсов и данных, применяя некую систему именного пространства. Ресурсы представляются как файлы, применяется один стандартный коммуникационный протокол. Благодаря этому, такие ресурсы как хранилища данных, сервисы и внешние устройства могут разделяться между различными Inferno-системами. Интерфейс ресурса можно импортировать в локальную систему и им могут пользоваться приложения, которые не знают, локальный данный ресурс или удаленный.

Безопасность высокого уровня также является частью Inferno-системы. Благодаря тому, что для всей сети используется один стандартный коммуникационный протокол, безопасность обеспечивается на системном уровне. **Inferno** предлагает также поддержку аутентификации, основанной на шифровании.

3. ОС, разработанные специально для портативных устройств

3.1. ITRON

ITRON (Industrial The Real-time Operating system Nucleus) – это широко распространенная в Японии операционная система для встроенных систем, которая используется в роботах, аппаратах факсимильной связи, цифровых камерах, а также в хостах разнообразных устройств [ITRON]. По сути **ITRON** является открытым стандартом ОСРВ для встроенных систем. После создания в Японии спецификации μITRON (micro-ITRON) и быстро возросшей ее популярности корпорация Accelerated Technology (разработчик серии ОСРВ Nucleus с открытым исходным кодом) разработала ядро реального времени Nucleus μiPLUS, которое соответствует стандартам интерфейса μITRON, что позволяет переносить ранее созданные приложения, соответствующие этому стандарту, на широкий ряд процессоров, выполняющих Nucleus μiPLUS.

Создание ОС **ITRON** было вызвано необходимостью введения каких-либо стандартов в море несовместимых между собой операционных систем для встроенных систем. **ITRON** предлагает спецификации для стандартного ядра реального времени, которое с незначительными настройками могло бы выполняться на различных устройствах. Согласно оценкам японской прессы от трех до четырех биллионов микропроцессоров работают под ОС **ITRON**.

Несмотря на эту популярность, ОС **ITRON** имеет большой дефект. Широкие возможности по модификации ОС **ITRON**, данные разработчикам для того, чтобы они могли подогнать спецификации ядра под свои требования, основано на концепции “слабой стандартизации”. Слабая стандартизация приводит к большим трудностям в создании унифицированной среды разработки ОС **ITRON**.

При проектировании спецификаций **ITRON** учитывались следующие технические требования к спецификациям ОСРВ для встроенных систем:

- извлекать максимальную производительность из аппаратуры,
- способствовать улучшению эффективности программного обеспечения,
- обеспечивать масштабируемость некоторого набора систем.

Для того, чтобы спецификации **ITRON** удовлетворяли этим требованиям, они проектировались в соответствии со следующими принципами:

- Увеличить адаптируемость к аппаратуре, избегая чрезмерной виртуализации аппаратуры. Адаптация к аппаратуре означает изменение спецификаций ОСРВ и внутренних реализационных методов, приводящее к возрастанию производительности всей системы в целом. Более точно, спецификации **ITRON** вводят явное разграничение между аспектами, которые следует стандартизовать через аппаратную архитектуру, и вопросами, которые должны быть решены оптимально на основе природы аппаратуры и ее

производительности. Среди аспектов, которые стандартизуются, выделяются правила планирования задач; имена и функции системных вызовов; имена, порядок и значение параметров; имена и значения кодов ошибок. Во всех других вопросах стандартизация и виртуализация не проводится, т.к. это может привести к снижению производительности во время выполнения. Это относится к размеру параметра в битах и методам, стартующим обработку прерываний, – такие вопросы решаются отдельно для каждой реализации.

- Учитывать адаптируемость к приложениям. Адаптация к приложению означает изменение спецификаций ядра и внутренних реализационных методов, опирающихся на функции ядра и производительность, требуемую приложением, приводящее к возрастанию производительности всей системы в целом. В случае встроенной системы, объектный код ОС генерируется отдельно для каждого приложения, таким образом, адаптация к приложению работает особенно хорошо (рис. 13). При проектировании спецификаций **ITRON** такая адаптация сопровождается обеспечением независимости функций ядра друг от друга, насколько это возможно, тем самым разрешая каждому приложению выбрать только те функции, которые ему необходимы. На практике это выражается в том, что большинство μITRON-спецификационных ядер поставляются в библиотечном формате. Каждый системный вызов обеспечивается единственной функцией, что позволяет легко встраивать только необходимые функции.
- Обеспечить простоту обучения разработчика программного обеспечения. Спецификации **ITRON** применяют стандартизацию как способ облегчения приобретения разработчиками необходимых навыков. Согласованность в использовании терминологии, именования системных вызовов и удобная справочная система гарантируют быстрое усвоение основных положений и широкое применение их впоследствии. Возможен и другой способ обучения – на основе изучения текстовых материалов.
- Организация спецификаций в серии и разделение на уровни. Для того, чтобы обеспечить адаптацию к широкому многообразию аппаратуры, спецификации организуются в серии и подразделяются на уровни. Например, спецификация μITRON (версия 2.0) была создана, главным образом, для использования в системах с 8- или 16-битовых MCU, в то время как спецификация ITRON2 предназначена для 32-битовых процессоров. Каждая спецификация далее разбивается на уровни, основанные на степени востребованности каждой функции. При реализации ядра соответствующий уровень выбирается на основе предназначения приложений и требуемых для них функций. Последняя реализованная спецификация μITRON3.0 подразделяет системные вызовы на три уровня, что дает возможность этой одной спецификацией покрывать диапазон от маломасштабных до крупных

процессоров. Спецификации для распределенных и многопроцессорных систем также могут быть стандартизованы с помощью серий **ITRON**-спецификаций.

- Обеспечивать широкий набор функциональностей. Примитивы ядра не ограничиваются малым количеством функций, напротив, они покрывают широкий диапазон разнообразных возможностей. Выбирая примитивы, которые хорошо подходят для данного типа приложения и аппаратуры, системные разработчики смогут быстро и легко создавать программы, обеспечивающие высокую производительность времени выполнения.

Из доступных версий спецификаций **ITRON** самой последней является спецификация µITRON4.0. Рассмотрим ее подробно.

Под термином "задача" в системе **ITRON** понимается единица параллельной обработки. Переключение выполнения с одной задачи на другую называется диспетчеризацией. Процесс выбора следующей задачи для выполнения называется планированием.

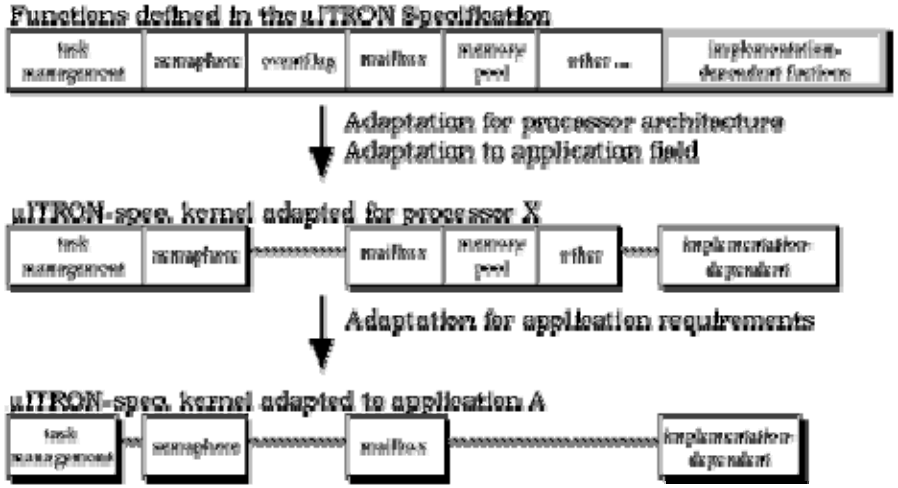

Рис. 13. Адаптация в спецификации µITRON.

Система ITRON оперирует следующими понятиями для описания **состояния задач**:

- Выполняющаяся (running),
- Готовая к выполнению (ready),
- Блокированная (bloked)
  - Ждущая (waiting) – ожидается выполнение каких-либо условий,
  - Приостановленная (suspended) – остановлена другой задачей или самой собой,

  - Ждущая-приостановленная (waiting-suspended) – ожидаются условия, и приостановлена,
- Спящая (dormant) – еще не выполнялась или уже завершилась,
- Несуществующая (non-existent) – не существует в системе, или и не создавалась, или уже уничтожена.

Планирование задач основано на приоритетах, присвоенных задачам, и является вытесняющим (preemptive). Совокупность задач с одинаковым приоритетом обслуживается по принципу – "первый пришел, первым обслужен" (FCFS – first come, first served).

Управление задачами. Функции управления задачами включают создание и уничтожение задачи, активацию и завершение задачи, отмену запросов на активацию и получение информации о состоянии задачи. Задача является объектом с уникальным идентификатором (ID).

Обработка прерываний. В спецификации µITRON4.0 обработка внешних прерываний описывается с помощью обработчиков прерываний (interrupt handlers) и программ обслуживания прерываний (interrupt service routines). Обработчики прерываний управляют устройствами IRC (Interrupt Request Controller) и зависят от архитектуры прерываний процессора. Они не могут быть переносимыми на другие платформы без изменений. Программы обслуживания прерываний запускаются обработчиками прерываний, и были введены в спецификации µITRON4.0 для улучшения переносимости обработки прерываний.

Спецификация µITRON4.0 определяет интерфейсы приложения для регистрации обработчика прерываний и интерфейсы приложения для программы обслуживания прерываний. Реализация должна обеспечивать либо один набор интерфейсов, либо оба. Если обеспечиваются интерфейсы только для регистрации обработчика прерываний, ядро может обеспечить связующую программу для обработчика прерываний, которая включает процессы, запускающиеся до и после выполнения обработчика прерываний. Если обеспечиваются интерфейсы только для регистрации программы обслуживания прерываний, ядро должно обеспечить для этой программы обработчик прерываний. Поведение системы с использованием обоих интерфейсов определяется реализацией.

Ядро не управляет прерываниями с приоритетами выше порогового приоритетного уровня. Такие прерывания называются неядерными прерываниями. Метод определения порогового приоритетного уровня определяется реализацией. Из обработчиков прерываний, запущенных неядерными прерываниями, нельзя сделать сервисный вызов (вызов функции ядра или программного компонента).

В спецификации µITRON4.0 существует два способа для указания прерывания – с помощью номера прерывания и через номер обработчика прерывания. К тому же программа обслуживания прерывания идентифицируется уникальным идентификатором объекта (ID).

Управление исключительными ситуациями. Спецификация µITRON4.0 определяет управление исключительными ситуациями центрального процессора (CPU) и функции управления исключительными ситуациями на уровне задач.

Обработчик исключительных ситуаций CPU запускается, когда процессор обнаруживает исключительную ситуацию. Обработчик исключительных ситуаций CPU может быть зарегистрирован приложением для каждого вида исключительной ситуации CPU. Ядро может обеспечивать связующую программу (glue routine) для обработчика исключительной ситуации CPU, которая включает процессы, запускаемые до и после выполнения обработчика исключительной ситуации CPU.

Поскольку обработчики исключительных ситуаций CPU являются общими для всей системы, контекст и состояние в точке возникновения исключительной ситуации CPU исследуются самим обработчиком исключительной ситуации CPU. Если исключительная ситуация CPU возникает в задаче, обработчик исключительной ситуации CPU может позволить провести обработку этой исключительной ситуации соответствующей программе задачи.

Функции управления исключительными ситуациями задачи используются для остановки нормального выполнения задачи и запуска программы управления исключительной ситуации задачи, которая выполняется в контексте данной задачи. После возвращения из этой программы будет продолжено прерванное выполнение задачи. Приложение может зарегистрировать одну программу управления исключительными ситуациями для каждой задачи.

Обработчик исключительных ситуаций CPU определяется реализацией, поскольку сильно зависит от архитектуры управления исключительными ситуациями и реализации ядра. Он не переносим на другие платформы.

Сервисные вызовы, которые могут быть встретиться в обработчике исключительных ситуаций CPU, определяются реализацией. Однако, обработчик исключительных ситуаций CPU должен выполнять следующие операции (метод выполнения определяется реализацией):

- Читать контекст и системное состояние при возникновении исключительной ситуации CPU. Ядро должно обеспечивать метод ссылки к информации о системном состоянии.
- Читать ID задачи, в которой возникла исключительная ситуация CPU.
- Запросить управление исключительными ситуациями задачи.

Контекст и системное состояние. В спецификации µITRON4.0 ядро управляет выполнением следующих единиц обработки:

- Обработчики прерываний.
  - Программы обслуживания прерываний.
- Обработчики временных событий.
- Обработчики исключительных ситуаций CPU.
- Программы расширенных сервисных вызовов.

- Задачи.
  - Программы управления исключительными ситуациями задачи.

Обработчики прерываний и программы обслуживания прерываний выполняются в своих собственных независимых контекстах.

Обработчики временных событий запускаются по временному триггеру и выполняются в своих собственных независимых контекстах. Рассматриваются три вида обработчиков временных событий – циклические, аварийные и по переполнению.

Обработчики исключительных ситуаций CPU выполняются в независимом контексте, определяемом исключительной ситуацией CPU и контекстом, в котором она возникла.

Программы расширенных сервисных вызовов регистрируются приложением и запускаются расширенными сервисными вызовами. Программа расширенного сервисного вызова выполняется в независимом контексте, определяемом расширенным сервисным вызовом и контекстом, из которого произошел этот вызов.

Задачи выполняются в своих собственных независимых контекстах. Программа управления исключительными ситуациями задачи выполняется в ассоциированном контексте задачи.

Процессы ядра не классифицируются как единицы обработки, упомянутые выше. Процессы ядра включают выполнение сервисных вызовов, диспетчер, связующие программы для обработчиков прерываний (или программ обслуживания прерываний), связующие программы для обработчиков исключительных ситуаций CPU. Контекст выполнения процессов ядра никак не влияет на поведение приложения.

Предшествование выполнения каждой единицы обработки специфицируется следующим образом:

1. Обработчики прерываний, обработчики временных событий, обработчики исключительных ситуаций CPU
2. Диспетчер (процесс ядра)
3. Задачи

Предшествование в первой группе определяется реализацией. Относительное предшествование задач определяется с помощью правил планирования.

Сервисные вызовы ядра, главным образом, выполняются атомарно, и состояние действующих процессов сервисных вызовов скрыто. Однако реализация может изменить это положение, чтобы улучшить реакцию системы. В этом случае операция сервисного вызова должна все же выполняться атомарно, насколько приложение может определить использование сервисного вызова. Такое поведение называется гарантией атомарности сервисного вызова. Однако иногда это трудно гарантировать при поддержке высокого уровня реакции с реализационно-специфическими функциями, которые не

покрывает данная спецификация. В таком случае разрешается ослабление атомарности сервисного вызова.

При атомарном выполнении сервисных вызовов их предшествование является самым высоким. При ослаблении атомарности предшествование сервисных вызовов становится реализационно-зависимым до тех пор, пока их предшествование выше приоритета единицы обработки, вызвавшей эти сервисные вызовы.

Другие процессы ядра, такие как диспетчер и связующие программы для обработчиков прерываний и исключительных ситуаций обрабатываются аналогично.

Состояние CPU может быть блокированным или разблокированным. В спецификации μITRON4.0 блокированное состояние CPU рассматривается как состояние, независимое от прерываний и диспетчеризации задач. В блокированное состояние CPU могут быть вызваны только несколько сервисных вызовов.

В заблокированном состоянии обработчики прерываний (за исключением тех, которые были запущены неядерным прерыванием) и обработчики временных событий не запускаются и диспетчеризация не совершается. Блокированное состояние CPU можно рассматривать как состояние, в котором предшествование выполняющейся единицы обработки является наивысшим. Возможно существование и промежуточного состояния, которое не является ни блокированным, ни разблокированным, – такая реализация тоже имеет место.

Состояние CPU после запуска обработчика прерывания является реализационно-зависимым. Однако, как попасть в разблокированное состояние, определяется реализацией обработчиков прерываний. Реализация определяет также, как корректно вернуться из обработчиков прерываний после того, как система перешла в разблокированное состояние. Поведение не определено, если обработчики прерываний не делают возврат так, как специфицировано в реализации.

После запуска программ обслуживания прерываний и обработчиков временных событий система переходит в разблокированное состояние. При возврате из них, система также должна быть в разблокированном состоянии. Поведение не определено, если система после возврата из них оказалась в блокированном состоянии.

Запуск и возврат из обработчиков исключительных ситуаций CPU не изменяют состояния CPU. Если состояние изменяется в обработчике исключительной ситуации CPU, его следует возвращать в предыдущее состояние перед возвратом из обработчика. Поведение не определено, если это не совершается.

Запуск и возврат из программ расширенных сервисных вызовов не изменяют состояния CPU.

После запуска задачи система находится в разблокированном состоянии. После ее окончания система должна быть в разблокированном состоянии. Поведение не определено, если задача закончилась, а система в блокированном состоянии.

Запуск и возврат из программ управления исключительными состояниями задачи не изменяют состояния CPU. Однако не специфицируется, в каком состоянии были запущены программы управления исключительными состояниями задачи. После возврата из программы, состояние CPU остается тем же самым, как установлено программой.

Прерывания обычно (но не всегда) разрешены в разблокированном состоянии CPU.

Диспетчеризация может пребывать в двух состояниях – разрешенном или запрещенном. В запрещенном состоянии диспетчеризация не совершается. Запрещенное состояние диспетчеризации может рассматриваться как состояние, в котором предшествование выполняющейся единицы обработки выше, чем у диспетчера. Реализация может допускать существование промежуточного состояния.

Переход к запрещенному состоянию диспетчеризации называется “отключение диспетчеризации”, переход к разрешенному состоянию диспетчеризации называется “включение диспетчеризации”.

В запрещенном состоянии диспетчеризации сервисные вызовы, которые могут быть вызваны из контекста задачи, имеют следующие ограничения. Если вызванный в запрещенном состоянии диспетчеризации сервисный вызов привел к тому, что вызвавшая его задача перешла в блокированное состояние, поведение не определено, если не специфировано иное. Сервисные вызовы, вызванные из незадачного контекста, не имеют ограничений даже в запрещенном состоянии диспетчеризации.

Запуск и возврат из обработчиков прерываний, программ обслуживания прерываний, обработчиков временных событий и исключительных ситуаций CPU не изменяют состояния диспетчеризации. Поведение не определено, если при возврате из этих обработчиков/программ состояние диспетчеризации не возвращается в предыдущее состояние.

Запуск и возврат из программ расширенных сервисных вызовов не изменяют состояния диспетчеризации.

После запуска задачи систем находится в разрешенном состоянии диспетчеризации. После окончания задачи система должна быть в разрешенном состоянии диспетчеризации. Поведение не определено, если задача заканчивается в запрещенном состоянии диспетчеризации.

Запуск и возврат из программ управления исключительными состояниями задачи не изменяют состояния диспетчеризации.

Состояние диспетчеризации рассматривается независимо от состояния CPU.

В спецификации μITRON4.0 не существует сервисных вызовов в незадачном контексте, которые изменяют состояние диспетчеризации. Следовательно, невозможно изменить состояние диспетчеризации внутри обработчиков прерываний и временных событий, если это не обеспечивается реализационно-специфическим расширением. Эти правила распространяются и на

обработчики исключительных ситуаций CPU, когда они выполняются в незадачном контексте.

Диспетчеризация не производится во время выполнения единицы обработки с более высоким предшествованием, чем у диспетчера, в заблокированном состоянии CPU или в запрещенном состоянии диспетчеризации. Эти три состояния называются состоянием задержки диспетчеризации. Состояние задачи во время задержки диспетчеризации можно изменить только с помощью реализационно-специфических расширений. Такие расширения могут делать сервисные вызовы из незадачного контекста, что позволит перевести задачу из состояния “выполняющаяся” в состояние “приостановленная” или “спящая”. Кроме того, расширения могут позволять сервисным вызовам выводить задачу из состояния “приостановленная” в запрещенном состоянии диспетчеризации.

Если задачу в состоянии “выполняющаяся” нужно перевести в состояние “приостановленная” или “спящая”, переход задерживается до тех пор, пока система не выполнит диспетчеризацию. Во время этой задержки считается, что выполняющаяся задача находится в промежуточном состоянии. Интерпретация задачи в этом промежуточном состоянии зависит от реализации. Задача, которая должна поступить следующей на выполнение, остается в состоянии “готова” до тех пор, пока не произойдет диспетчеризация.

Сервисные вызовы классифицируются согласно следующим трем категориям:

- Сервисные вызовы для незадачных контекстов.
- Сервисные вызовы, которые могут быть вызваны из любых контекстов.
- Сервисные вызовы для задачных контекстов.

Выполнение сервисных вызовов из незадачных контекстов может быть отложено до тех пор, пока не закончится выполнение единиц обработки с более высоким предшествованием, чем у диспетчера. Это позволяет гарантировать атомарность системных вызовов без запрещения прерываний на слишком долгое время. Такой подход называется отложенным выполнением сервисных вызовов. Однако, существует группа сервисных вызовов, для которых не разрешается откладывать их выполнение.

В спецификации µITRON4.0 специфицируется также процедура инициализации системы, а также процедуры регистрации объектов и их уничтожения. Объект идентифицируется уникальным идентификатором и регистрируется ядром с помощью статического вызова некоторого метода интерфейса приложения или сервисного вызова, которые создают этот объект.

Спецификации µITRON4.0 специфицируют формат написания на языке C следующих единиц обработки: программ обслуживания прерываний, обработчиков временных событий, программ расширенных сервисных вызовов, задач и программ управления исключительными ситуациями задачи. Однако не специфицируется формат написания единиц обработки на языке ассемблера. Формат для написания обработчиков прерываний, обработчиков исключительных ситуаций CPU и атрибутов объектов, которые используются

для их регистрации в ядре, определяются реализацией и не специфицируется в спецификации µITRON4.0.

Приложения могут использовать константы и макросы, описывающие конфигурацию ядра, с целью улучшения переносимости приложений. Метод, который используется для определения констант и макросов конфигурации ядра, является реализациннно-зависимым. Обычно они определяются в заголовочных файлах ядра или могут быть сгенерированы конфигуратором. Как альтернатива, они могут быть определены в заголовочных файлах приложения и затем использованы для конфигурации ядра.

3.2. Windows CE

Системы семейства Microsoft **Windows CE** [WinCE] являются открытыми, масштабируемыми ОС, позволяющими компоновать ОС для широкого диапазона современных небольших устройств, которые соединяют в себе компьютерные, телефонные и сетевые возможности. Устройство, на которое может быть установлена **Windows CE**, обычно проектируется для специализированного использования, часто работает автономно и требует маленькой ОС, которая имеет детерминированные реакции на прерывания.

Последней версией из этого семейства является система Microsoft **Windows CE** 5.0, в которой объединены возможности систем реального времени и последние технологии Windows. По сравнению с другими ОСРВ **Windows CE** проектировалась так, чтобы она была совместимой с универсальными ОС.

ОСРВ **Windows CE** является модульной с небольшим ядром и необязательными модулями, которые выполняются как независимые процессы. Планирование в **Windows CE** осуществляется на основе приоритетов. Поддерживается защита ядра и процессов друг от друга. Кроме того, возможен режим работы, когда отсутствует защита между процессами и ядром. Следует отметить, что прерывания выполняются как потоки и имеют уровни приоритетов потоков. **Windows CE** поддерживает также нити (fiber), являющиеся потоками, которыми ядро не управляет. Каждая нить выполняется в контексте потока, который ее создал, ее можно использовать для создания планировщика внутри потока. Такие нити используются в экзотических или унаследованных приложениях, но они непригодны в системах реального времени.

Windows CE имеет ограничение на физическую память – 512MB. Microsoft ввел это ограничение для того, чтобы **Windows CE** могла выполняться на большом диапазоне встраиваемых процессоров без проблем совместимости, поскольку некоторые из этих процессоров способны управлять физической памятью в 512MB. Однако физическая память может быть отображена в адресные регионы, размер которых превышает 512MB.

RAM в устройстве **Windows CE** разделяется на две области – хранилище объектов и программная память. Хранилище объектов напоминает постоянный виртуальный диск RAM. Данные в таком хранилище запоминаются во время приостановки или операции частичной переустановки (soft reset). Когда операция возобновляется, система находит ранее созданное хранилище объ-

ектов и использует его. Программная память состоит из оставшейся RAM, она работает как RAM в персональном компьютере – запоминает стеки и области для динамически выделяемой памяти (heaps) выполняющихся приложений.

Во время старта **Windows CE** создает единое виртуальное адресное пространство в 4GB, которое затем разделяется на две секции – ядро и пользовательское пространство, как и в универсальной ОС Windows. Далее пользовательское пространство делится на 64 слота по 32MB, из которых 32 резервируются для процессов (отсюда ограничение на число процессов в системе). Все процессы разделяют виртуальное адресное пространство, но не имеют доступа друг к другу. В виртуальном адресном пространстве в 32MB находится все, что нужно процессу – программа, данные, область динамической памяти (heap). Если процесс имеет соответствующие права доступа, он может получить память сверх ограничения в 32MB, обратившись к специальному процессу (VirtualAlloc) или используя файлы отображенной памяти (memory mapped files).

Windows CE реализует постраничное управление виртуальной памятью. Размер страницы зависит от платформы, но по возможности используется размер в 4KB. Есть возможность запретить страничную организацию, что важно для систем реального времени. В этом режиме модуль целиком загружается в память перед выполнением. Таким образом, сбои страничной организации (paging) не повлияют на выполнение приложения.

В обычной конфигурации для защиты ядра и процессов друг от друга используется MMU. Есть возможность сконфигурировать **Windows CE** без защиты памяти между процессами и ядром, что позволяет достичь большей производительности системы.

В **Windows CE** механизмы синхронизации можно разделить на две категории:

- механизмы защиты от одновременного доступа – критические секции, мьютексы и семафоры,
- механизмы взаимодействия – события и очереди сообщений.

В отличие от других ОСРВ **Windows CE** поддерживает обобщенные функции ожидания для различных типов объектов (мьютексов, семафоров, событий, процессов и потоков). Преимущество таких функций состоит в том, что можно ожидать многие объекты сразу, пока один из них не подаст сигнал. Критические секции можно использовать только внутри одного процесса. Вычислительные семафоры и мьютексы могут быть использованы как внутри одного процесса, так и между процессами.

Windows CE использует наследование приоритетов для того, чтобы избежать проблемы инверсии приоритетов.

Windows CE позволяет построить, отладить и внедрить настроенную ОС из предлагаемого набора компонент с помощью инструмента Platform Builder. Процесс разработки ОС **Windows CE** показан на рис. 14.

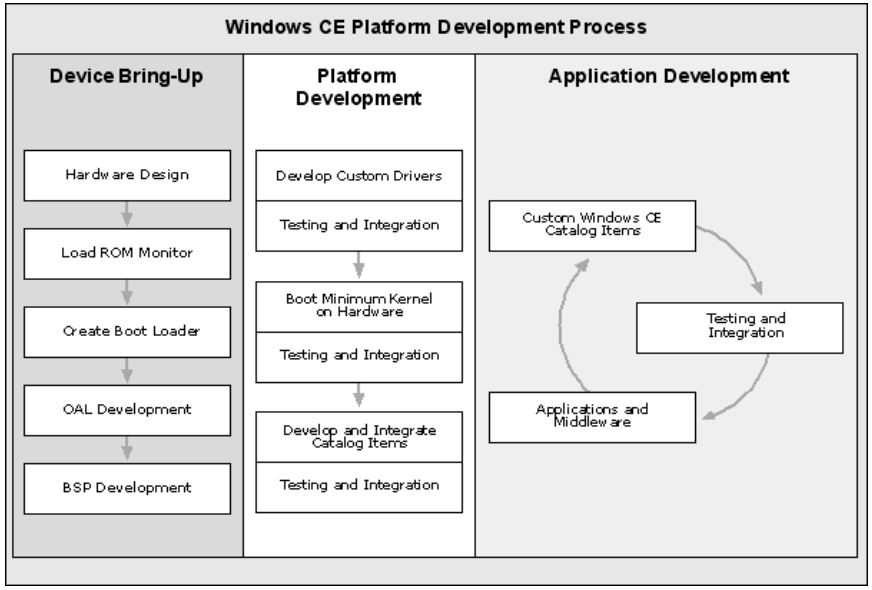

*Рис.14. Процесс разработки ОС **Windows CE**.*

Ниже приведена архитектура ОС **Windows CE**.

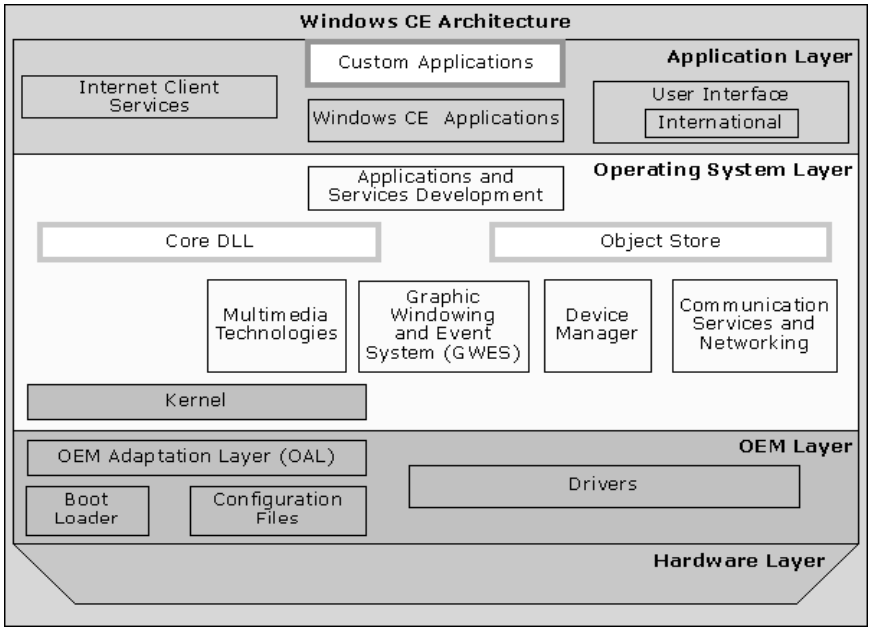

*Рис. 15. Архитектура ОС **Windows CE.***

Введение адаптационного уровня производителя в архитектуру Windows CE позволило повысить ее эффективность.

Хотя **Windows CE** имеет модульную структуру, которая позволяет создавать минимальные конфигурации для небольших систем, она все-таки остается сложной и требует относительно большого пространства на диске, поэтому она не является хорошим выбором для глубоко встраиваемых систем.

3.3. JavaOS

JavaOS – это семейство небольших по размеру, эффективных операционных систем, оптимизированных для поддержки Java-среды, предназначенных выполняться на тонких клиентах – сетевых компьютерах [SM99]. **JavaOS** проектировалась для выполнения приложений, написанных на языке Java, и была создана корпорацией Sun Microsystems для того, чтобы виртуальная машина Java (JVM) непосредственно выполнялась на микропроцессорах, без участия резидентной операционной системы.

По сравнению с реализацией Java над универсальными ОС, **JavaOS** обеспечивает не только экономию ресурсов (что важно в первую очередь для встраиваемых систем), но и снижение затрат на администрирование (что сулит существенную экономию для корпоративных систем).

Семейство **JavaOS** включает три разновидности ОС:

- **JavaOS for Business** (развивается совместно с корпорацией IBM);
- **JavaOS for Consumers,**
- **JavaOS for network computers.**

Вообще говоря, для обеспечения конфигурации **JavaOS** имеет базу данных, состоящую из именованных Java-объектов. Эта база данных помогает поддерживать динамическую реконфигурацию.

Характерной чертой **JavaOS** является стремление к максимальной независимости от платформы. Такая независимость способствует мобильности самой **JavaO**S и построенных на ее основе программных систем, что очень важно в условиях большого разнообразия аппаратных модификаций и частой смены моделей. Для достижения независимости от платформы внутри **JavaOS** выделены технологические интерфейсы – с платформой (**JavaOS** Platform Interface, JPI) и периферийными устройствами (**JavaOS** Device Interface, JDI). Все, что выше этих интерфейсов, может быть написано на Java и сделано мобильным.

JavaOS построена по принципу многослойной архитектуры, в которой каждый слой может обновляться независимо от всех остальных. Архитектура **JavaOS** состоит из микроядра и диспетчера памяти, драйверов устройств, виртуальной машины Java, систем **JavaOS** Graphics и JavaOS Windowing, сетевых классов и средств поддержки всех интерфейсов прикладного программирования (API) Java. Приложения, написанные для **JavaOS**, совместимы с браузерами и операционными системами, соответствующими стандартам Java.

Многослойная архитектура **JavaOS** разделяет коды на платформенные и платформенно-независимые. Платформенный код, который компилируется в исходный, состоит из ядра и виртуальной машины Java. Платформенно-независимая часть **JavaOS** (написанная на Java) состоит из систем **JavaOS** Window и Graphics, драйверов устройств JavaOS и сетевых классов JavaOS.

Микроядро JavaOS поддерживает начальную загрузку, управление прерываниями, многопоточность, управление перехватами и распределением динамической памяти. Микроядро также поддерживает ряд сервисов сеанса работы, которые включают Java виртуальную машину, сборщик мусора и сервисный загрузчик.

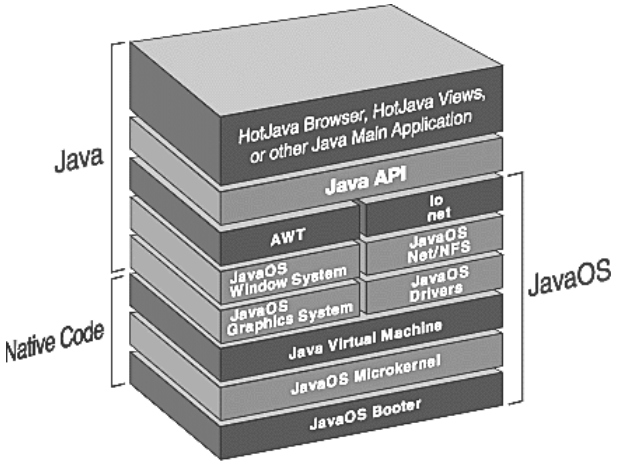

*Рис. 16. Многослойная архитектура **JavaOS**.*

Драйверы устройств JavaOS написаны на языке Java и являются переносимыми и расширяемыми. Сетевые классы **JavaOS**, также написанные на Java, включают стандартные промышленные сетевые протоколы, такие как TCP/IP и др.

Уменьшение объемов платформенно-зависимых частей **JavaOS** упрощает администрирование корпоративных конфигураций, делая их более однородными. Это важное средство снижения общей стоимости владения клиентскими частями информационных систем.

Стоит отметить, что микроядро **JavaOS** для эффективной привязки к аппаратному обеспечению или к другой операционной системе создается на языке C и соответствующем языке ассемблера.

Для **JavaOS** характерна распределенная работа ее компонентов, безопасность данных, а также контроль использования ресурсов, как сервера, так и клиента.

JavaOS перенесена на системы с микропроцессорами SPARC, x86 и ARM. Для полной сетевой реализации с поддержкой рабочей среды Java (Java Runtime Environment) нужно 2.4 MB памяти. **JavaOS** с браузером HotJava требует минимального объема памяти в 4 MB.

3.4. Jbed

Система **Jbed** фирмы Oberon Microsystems, является ОСРВ с ядром, ориентированным на Java технологию [Jbed98], и может рассматриваться как Java платформа для встроенных систем и систем реального времени. Другими кандидатами Java платформ являются такие системы, как EmbeddedJava и JavaCard. В основном они различаются средствами поддержки потоков, сбора мусора, чисел с плавающей запятой и т.п. В **Jbed** ядро в действительности и есть Java машина, которая также называется runtime системой. **Jbed** runtime система как минимально возможная система обеспечивает планирование потоков, распределение памяти и сбор мусора. Такая конфигурация требует 64KB оперативной памяти. Другие возможные конфигурации могут включать минимальную систему с TCP/IO и веб-сервером (что требует 128KB) и минимальную систему с сетевым загрузчиком и компилятором для трансляции Java-кода в машинный код целевого компьютера (что требует 256KB). В терминах настраиваемости операционных систем это означает, что компоненты ядра обладают крупным уровнем детализации, и к тому же в **Jbed** отсутствует возможность динамической конфигурации на уровне ядра.

Микроядро **Jbed** обладает специфическими особенностями, необходимыми для встроенных систем и систем реального времени, такими как малое количество требуемой памяти, поддержка потоков реального времени и управление дедлайнами.

Над ядром выстраиваются такие компоненты, как драйверы периферийных устройств, драйверы устройств связи, сетевые загрузчики, библиотеки. Эти компоненты называются встроенными приложениями и подгружаются они по требованию. Таким образом, на этом уровне **Jbed** обеспечивает динамическую настраиваемость. Кроме того, уровень приложений поддерживает клиент-серверную модель. На этом уровне такие приложения, как управление процессами, удаленная диагностика и система сигнализации, называются клиентами. Серверные программы управляют клиентскими компонентами (встроенными приложениями), которые могут осуществлять отладку, удаленное управление или работать как веб-серверы. Серверы позволяют клиентам удаленно диагностировать встроенные приложения, замещать компоненты и удаленно управлять встроенными системами с некоторого персонального компьютера. Однако эти возможности не доступны на уровне ядра операционной системы. Получается, что на уровне ядра конфигурация заключается просто в выборе одной из упомянутых выше конфигураций.

3.5. Nucleus RTOS

Операционная система **Nucleus**, разработанная корпорацией Accelerated Technology, предназначена для встраиваемых приложений [NUCLEUS]. **Nucleus** является кросс-системой, т.е. программный продукт создается на одной программно-аппаратной платформе, а выполняется на другой. ОСРВ **Nucleus** поставляется вместе с открытым кодом.

Ядро ОСРВ **Nucleus,** Nucleus PLUS, обеспечивает многозадачную обработку, является переносимым и масштабируемым. Ядро реализовано как библиотека функций на языке C. Nucleus PLUS предоставляет такие возможности как управление взаимодействием задач (почтовые ящики, очереди, конвейеры, семафоры, события, сигналы), а также управление памятью, таймерами, прерываниями. Планирование задач осуществляется на основе приоритетов, а также по алгоритму FIFO. При выполнении системного вызова выполнение задачи может приостанавливаться на неопределенное время, на заданный интервал, или не приостанавливаться. Все объекты в системе могут создаваться и удаляться динамически.

3.6. EMERALDS

EMERALDS (Extensible Microkernel for embedded, ReAL-time, Distributed Systems) [ZS01] – это микроядро реального времени, написанное на языке C++, которое предназначено для малых и средних по размеру встраиваемых систем. Система **EMERALDS** является научной разработкой Мичиганского университета (University of Michigan).

EMERALDS обеспечивает обработку многопоточных процессов. Процесс в **EMERALDS** является пассивной сущностью, характеризующейся защищенным адресным пространством, в котором выполняются потоки. Каждый поток имеет приоритет, присвоенный ему пользователем, на основе которого ядро осуществляет его планирование. Ядро также снабжено системным вызовом, способным изменить приоритет потока во время выполнения. Для обеспечения эффективной защиты памяти ядро отображает себя в адресное пространство каждого процесса. При таком отображении переключение из приложения в ядро вызывает прерывание (TRAP), которое переключает центральный процессор из приложения в режим ядра, и совершается переход на соответствующий адрес внутри того же адресного пространства.

Ядро обеспечивает такие сервисы, как семафоры, таймеры, управление памятью и пр. В качестве механизма взаимодействия процессов **EMERALDS** использует обмен сообщениями через почтовые ящики, как для внутри-процессного, так и для межпроцессного взаимодействия. Проблема инверсии приоритетов решается с помощью введения наследования приоритетов.

3.7. CORTEX

CORTEX – это многозадачная ОСРВ для встраиваемых приложений, разработанная корпорацией ARTESYS (Australian Real Time Embedded Systems). Исходный код системы находится в свободном распространении для образовательных и некоммерческих целей.

Управление задачами включает временную поддержку, реентерабельность, вытеснение, основано на управлении событиями, является детерминированным и поддерживает приоритеты. Доступны три разных политики планирования. Поддерживается 62 уровня приоритетов для задач. Приоритетное прерывание обслуживания может осуществляться непосредственно через сервисы

управления вытеснением или косвенно с помощью взаимодействия между задачами и примитивов синхронизации. Поддерживается механизм наследования приоритетов.

Синхронизация задач и их защита осуществляется через рекурсивные блокировки ресурсов, мьютексы и условия, мониторы и условия, вычислительные семафоры, события.

3.8. DeltaOS

DeltaOS является ОСРВ для встраиваемых приложений, разработанной китайской корпорацией CoreTek Systems. Система поддерживается для наиболее популярных семейств микропроцессоров, таких как PowerPC, Intel X86, ARM, MIPS. **DeltaOS** настраивается и масштабируется в широком диапазоне систем реального времени и может применяться как в простых автономных устройствах, так и в сложных отказоустойчивых многопроцессорных системах.

DeltaOS основана на ядре DeltaCore, которое построено на надежной технологии реального времени, включающей приоритетное прерывание обслуживания, реентерабельность, многозадачность и детерминированное поведение. Ядро проектировалось на основе объектно-ориентированной парадигме. Объектами, которыми оперирует **DeltaOS,** являются задачи, семафоры, события, таймеры, очереди сообщений, сегменты памяти. Планирование основано на приоритетах. Поддерживается планирование с вытеснением, циклическое и на основе разделения времени. Обработка прерываний осуществляется через ISRs вне ядра. Для обеспечения быстрой обработки прерывания можно передавать непосредственно в ISRs. Системные вызовы, сделанные из ISR возвращают управление в ISR, таким образом сокращая временные затраты механизма планирования в ядре.

3.9. Palm OS

Операционные системы серии **Palm OS** (корпорация PalmPC) являются одним из популярных программных продуктов для портативных устройств и смартфонов [PALMOS]. К последним разработкам относятся **Palm OS** Garnet и **Palm OS** Cobalt 6.1.

Palm OS Garnet – это усовершенствованная версия системы **Palm OS** 5, которая обеспечивает дополнительные возможности, такие как динамическая область ввода, улучшенное сетевое взаимодействие и поддержка для широкого диапазона разрешающей способности экрана.

Palm OS Cobalt 6.1 – это следующее поколение системы **Palm OS**. Эта система позволяет создание новых классов устройств для коммуникации, предприятий, обучения и развлечений. **Palm OS** Cobalt 6.1 обеспечивает интегрированные телефонные функции, поддержку для сетей типа WiFi и Bluetooth и расширенные возможности для пользовательского интерфейса.

3.10. Symbian OS (EPOC)

EPOC – это операционная система от корпорации Psion Software, разработанная специально для мобильных, основанных на ROM-памяти (read-only memory), компьютерных устройств [EPOC]. **EPOC**16 – устаревшая 16-битовая версия этой ОС. **EPOC**32 – более современная 32-битовая ОС, которая поддерживает многозадачный режим с приоритетами. ОС **EPOC** распространяется корпорацией Symbian и часто выступает под названием Symbian OS, поскольку Symbian является совместным предприятием фирм Psion, Ericsson, Motorola и Nokia. ОС **EPOC** является ближайшим конкурентом **Windows CE** на рынке устройств PDA (personal digital assistant)**.**

4. Настраиваемость операционных систем

Рассмотрим более подробно основные подходы к настраиваемости операционных систем. В первую очередь здесь обсуждаются вопросы ОС под набор приложений, хотя упоминаются и вопросы настройки на аппаратную платформу.

При обсуждении настраиваемости ОС часто применяются понятия распространения (spread) и уровня детализации настройки.

4.1. Адаптация, осуществляемая человеком

Системы в категории статической адаптации являются, по сути, каркасами операционных систем (framework), которые проектировщик может настраивать в соответствии с целями будущей ОС. Если область приложений, которые будут использовать создаваемую ОС, заранее известна, можно создать специфическую ОС, которая будет поддерживать необходимую функциональность и ничего более. Такой подход приводит к созданию высокопроизводительной специфической ОС. После компоновки такой системы ее функциональность и политики фиксированы. Эта форма адаптации применяется к встроенным ОС для устройств со специфической функциональностью и более или менее стабильной деятельностью.

Динамическая адаптация от имени администратора возможна в большинстве промышленных ОС – как во время загрузки, так и во время работы. Во время загрузки ядру передаются параметры настройки и конфигурационные установки, во время работы администратор может инсталлировать и подгружать новые модули ядра, а также отслеживать параметры производительности и настраивать операционную систему.

4.1.1. Статическая адаптация, инициированная проектировщиком

В ОС этой категории все системные функции и политики определяются на этапе проектирования, а все системные сервисы встроены в ядро. Основное предназначение таких ОС – это специфическая операционная система, возможно даже для единственного приложения. Инициатором адаптации является проектировщик, функциональность и требования хорошо известны и понятны уже на этапе проектирования. Такой подход ведет к обедненным и

высокопроизводительным системам, в которых может присутствовать только строго ограниченная функциональность, а все сервисы оптимизируются статически под вполне определенное приложение. Ясно, что новая функциональность и приложения другой категории не могут поддерживаться такой системой. Это значит, что для каждого приложения должна проектироваться и реализовываться новая система, и что один тип устройств или компьютеров будет поддерживать только одно приложение (или ограниченного число приложений) [KLM93]. Появляются каркасы (framework) ОС общего назначения, которые помогают избежать проектирования каждой новой ОС с нуля.

Flux OSKit – это система, состоящая из каркаса и библиотечных модулей [FBB97]. Каркас **OSKit** представляет собой набор библиотек, из которых компонуется ядро ОС. Все компоненты состоят из модулей. Используются интегрирующие (glue) слои, через которые осуществляется взаимодействие между компонентами и сервисами. Вообще говоря, компоненты скорее обладают крупным уровнем детализации, и являются, в основном, подсистемами, такими как файловая система, стек сетевого протокола или набор драйверов устройств.

Scout – это каркас для операционных систем, обслуживающих устройства в сети [MMO95]. ОС, созданная с помощью **Scout** состоит из модулей, связи между которыми представляются в виде графа модулей. Этот граф фиксируется на этапе проектирования. Связанные модули должны обеспечивать общий интерфейс.

Термин *маршрут* (path) означает поток данных от источника ввода/вывода, через систему, к стоку ввода/вывода. Это понятие можно рассматривать как последовательность решений о маршрутизации внутри модульной системы. Маршрут состоит из двух частей – последовательности модулей, которые определяют семантику (надежность, безопасность и согласование по времени), и требований на ресурсы, необходимых для обработки и прохождения данных. Маршруты через граф модулей создаются и разрушаются в динамике. Такое понятие маршрута хорошо подходит для распределения ресурсов и оптимизации производительности, т.к. маршрут обеспечивает нелокальный контекст, который не доступен внутри отдельного модуля.

В работе Спатчека и Петерсона [SP97] определяется архитектура безопасности, которая дает возможность проектировщику устанавливать политику безопасности для операционной системы **Scout.** Эта архитектура безопасности добавляет в **Scout** также многочисленные домены защиты, которые иначе находились бы в едином адресном пространстве.

Choices – это объектно-ориентированная, настраиваемая ОС, которая для настраиваемости использует каркасную технологию и подсистемы [CRJ87, CIR93]. Основное предназначение **Choices** – обеспечить возможность пользователям легко оптимизировать и адаптировать систему в соответствии с поведением приложений и рабочей нагрузкой. Система **Choices** спроектирована как иерархия каркасов, представляющая удобную организацию операционной

системы по слоям. В **Choices** каркас состоит из набора классов, представляющих системные сущности, такие как диски, память, планировщики и т.д. Различные подсистемы ОС, такие как управление памятью, управление процессами, файловое запоминающее устройство, управление исключительными ситуациями и т.д. создаются непосредственно из объектно-ориентированных каркасов. Ресурсы системы, механизмы и политики представляются как экземпляры классов, принадлежащих некоей иерархии классов, где настройка осуществляется через использование наследственности. Таким образом, специфическая ОС создается путем конкретизации классов и реализации набора объектов, которые вместе формируют данную ОС. Интерфейсом приложения является совокупность объектов ядра, экспортируемых через уровень защиты приложение/ядро.

Система **Choices** обладает слабой формой динамической адаптации. Когда приложению нужен какой-либо сервис, происходит динамическая загрузка соответствующего класса. Таким образом, нельзя сказать, что адаптация производится непосредственно через приложение, к тому же любое возможное поведение определяется статически.

Pebble является ОС, основанной на компонентах [MBG00]. В то же время **Pebble** можно назвать как каркасом ОС общего назначения, так и микроядром для узла, исполняющего роль портала (в последнем качестве она рассматривается в разделе о портал-оринентированных ОС). Эта система обеспечивает некоторый набор абстракций операционных систем, реализуемых удостоверенными компонентами пользовательского уровня. Эти компоненты могут наращиваться, замещаться или разбиваться по слоям, что позволяет измененным абстракциям сосуществовать с прежними или полностью заместить стандартный набор. **Pebble** дает возможность создавать модульные ОС из компонент многократного использования. В отличие от подобных систем, системные сервисы не интегрируются в ядро. Они предоставляются в виде удостоверенных серверных компонент, которые выполняются в защищенных доменах на уровне пользователя.

Система **PURE** (Portable Universal Runtime Executive) является хорошо конфигурируемой системой и предоставляет средства для подбора требуемой функциональности [Beu99]. Хотя применение **PURE** и не ограничено какой-либо областью приложений, все же ее главное предназначение находится в области глубоко встроенных систем. Проектирование **PURE** основано на двух концепциях – концепции семейства программ и объектно-ориентированного подхода. Концепция семейства программ обеспечивает иерархическую структуру системы, в которой минимальный набор системных функций используется как платформа для реализационных или системных расширений. Объектно-ориентированный подход служит основой для реализации. Минимальным компонентом является класс, т.е. систему **PURE** можно рассматривать как библиотеку классов. Например, компонент, реализующий управление потоками, состоит из 45 классов, выстроенных в 14-уровневую иерархию.

Компоненты **PURE** упорядочиваются в структуру, состоящую из ядер и их расширений. Ядра, так называемые CORE (concurrent runtime executive), отвечают за реализацию минимального набора системных функций, управляющих прерываниями и потоками. Дополнительные возможности, называемые минимальными системными расширениями, добавляются в систему с помощью ядерных расширений, называемых NEXT (Nucleus Extension).

Система **PURE** хорошо конфигурируется и обладает высоким уровнем детализации настройки.

Вышеописанные подходы приводят к хорошим результатам для устройств или компьютеров, функциональность которых известна заранее.

4.1.2. Динамическая адаптация, инициированная администратором

Этот класс ОС поддерживает адаптацию во время раскрутки или выполнения, которую осуществляет администратор, а иногда и пользователь. Такая адаптация достигается двумя способами.

Первый способ – это метод *загружаемого модуля ядра* (loadable kernel module) дает возможность доверенному лицу (обычно администратору или пользователю с полномочиями root) загрузить модули в ядро, таким образом, изменяя или расширяя функциональность ОС. Преимущество данного метода заключается в его простоте – это напоминает динамическую загрузку класса обычных приложений. Недостатком является возможность нарушения механизмов безопасности системы при добавлении произвольного кода в ядро, что может вызвать крах системы. Кроме того, стабильное состояние системы может стать нестабильным после загрузки злоумышленного или просто ошибочного модуля в ядро.

Второй способ – это настройка (tuning) системы. Обычно политики ОС параметризуются, и такие параметры могут изменяться либо администратором, либо пользователем. Кроме того, существуют счетчики производительности, которые помогают оценить преимущества той или иной политики, настроенной через параметры. Преимущество этого подхода состоит в том, что здесь не дискредитируются механизмы зашиты. Недостатком можно считать тот факт, что при таком подходе ограничиваются широта применения и уровень детализации.

Динамическая адаптация от имени администратора широко применяется во всех промышленных ОС (Linux, Solaris, Windows NT). Windows 2000 имеет сотни счетчиков производительности и соответствующих системных переменных. Ядро Linux интенсивно использует загружаемые модули.

Следует отметить, что контекст и мотивы расширения и адаптируемости различны для промышленных и исследовательских ОС. В академическом сообществе для ОС главной движущей силой является производительность, часто в не удостоверенном (untrusted) контексте. Инициатор адаптации не может считаться удостоверенным и, следовательно, целостность ОС нужно защищать. В промышленных ОС расширяемость, главным образом, используется для добавления разных форм функциональности в

удостоверенном контексте, где инициатор адаптации является удостоверенным (например, администратор файловой системы). Метод загружаемого модуля ядра используется для наращивания ядра новыми драйверами устройств с поддержкой новых файловых систем с новыми способами аутентификации, а также другими видами функциональности. Модули улучшения производительности, а также механизмов защиты целостности очень редки на практике. Как следствием этого, многие исследовательские результаты по адаптируемости и наращиванию не находят применения в промышленных ОС.

4.2. Адаптация, инициированная приложением

Адаптация, инициированная приложением, может производиться только динамически, и ее применение полезно в ОС общего назначения. В то время как традиционные ОС имели большие трудности при добавлении новых возможностей, требующихся, например, для обработки мультимедийных приложений, настраиваемые ОС быстрее адаптировались в новых условиях. Приложение часто знает свои потребности и может от своего имени инструктировать ОС о необходимой настройке. Механизмы, предоставляющие возможность настройки от имени приложений во время выполнения, вводят накладные расходы по производительности. Однако цель динамически настраиваемых ОС состоит в том, чтобы достичь лучшей производительности всей системы в целом, разрешая настройки, которые приводят к преобладанию преимуществ над накладными расходами.

4.2.1. Адаптация с уровня приложения

Рассмотрим системы, которые разрешают приложениям настраивать сервисы ОС через введение кода с уровня пользователя или непривилегированного уровня. Такие ОС обычно называются микроядерными, потому что они структурируются вокруг микроядра. Истинное микроядро должно быть минимальным, и должно предполагать отсутствие в нем каких-либо сервисов или политик.

4.2.1.1. Микроядерные ОС

Микроядерный подход существует довольно давно. Однако первые поколения микроядерных ОС показали результаты, далекие от ожидаемых. На практике так и не была реализована желаемая адаптируемость из-за низкой производительности и недостаточного уровня детализации настройки в ранних микроядерных ОС. Попытки улучшить показатели привели к тому, что фундаментальные библиотеки ОС, такие как файловая система Unix, были опять интегрированы в ядро, что улучшило производительность, но совсем не помогло увеличить настраиваемость.

Новое поколение микроядерных ОС намного больше отвечает целям настраиваемости. Однако критическим остается вопрос производительности, связанный со значительным количеством переключений между доменами (как между пользовательским уровнем и ядром, так и между адресными пространствами), а также с местоположением основной памяти [Lie93].

Выбранный набор абстракций, интегрированных в ядро, существенным образом влияет на производительность и гибкость. Чем меньше абстракций, тем большая гибкость остается для приложений. В ядре должны присутствовать только те абстракции, которые необходимы для деятельности самого ядра. Это хорошо сформулировано в работах Лидке [Lie95, Lie96] о системе **L4** – в ней ядерными абстракциями являются адресные пространства, потоки, IPC и уникальные идентификаторы. С этими абстракциями **L4** поддерживает рекурсивную конструкцию адресных пространств – исходное адресное пространство включает всю память и порты ввода/вывода и принадлежит исходной подсистеме или приложению.

Результатом приближения микроядерной философии к ее логической крайности становится ОС, в которой все системные сервисы выведены за пределы ядра и реализованы в виде библиотек, а само ядро становится попросту абстракцией аппаратных ресурсов. Такой экстрим реализован в ОС **Exokernel** [EKO95]. В ядре **Exokernel** отсутствуют какие-либо абстракции ОС и весь его интерфейс сведен к надстройке над аппаратурой. Единственная функция, которая оставлена в **Exokernel** – это выделить, вернуть и мультиплексировать физические ресурсы (страницы памяти, кванты времени процессора, блоки дисков и т.п.) безопасным образом. Композиция наиболее используемых интерфейсов в **Exokernel** до сих пор остается большой проблемой [SSF99].

Систему **2K** [2K] можно отнести к микроядерным ОС. Система **2K** основана на компонентах, и ее основной задачей является обеспечение настраиваемого каркаса для поддержки адаптации в сетевом окружении. Способность к адаптации регулируется параметрами, такими как пропускная способность сети, связность, доступность памяти, протоколы взаимодействия и компоненты аппаратных средств. ОС **2K** строится над Corba и использует данные метауровня и их методы, которые предлагает уровень ORB (object request broker) для адаптации. Компонент **2K** – это динамически загружаемый программный модуль, который хранится в динамически подключаемой библиотеке (DLL). Следует отметить, что система **2K** использует крупный уровень детализации.

4.2.1.2. Портал-ориентированные системы

Портал-ориентированная система – это микроядро, которое имеет минимально возможный код и работает в самом доступном привилегированном режиме. Такие системы обладают доменами защиты, которые обеспечивают безопасность работы пользователя. Порталы используются для взаимодействия между доменами. Домен защиты состоит из набора страниц и совокупности порталов, причем порталы могут разделять и страницы, и порталы.

Примером портал-ориентированной системы служит **Kea** – портал-ориентированное микроядро, которое обеспечивает низкоуровневые концепции для конструирования высокоуровневых сервисов [VH96]. Под низкоуровневыми концепциями понимаются домены (виртуальные адресные

пространства), обращения доменов друг к другу (IPC) и порталы. Портал ассоциирован с определенным сервисом – это точка входа для обращения к домену. Каждый сервис обязан регистрировать свой идентификатор в ядре, а ядро управляет доступом к этому интерфейсу. В отличие от истинных микроядерных ОС **Kea** не дает полной свободы для проектирования и реализации сервисов. Все же **Kea** дает возможность вводить динамическую реконфигурацию. При обращении к сервису (через его портал) ядро может решать, какую реализацию выбрать. Например, при использовании файлового сервиса администратор может наложить обязательный вызов сервиса сжатия данных перед передачей их файловому сервису. Более сложная реконфигурация возникает в случае, когда приложение ассоциирует новый портал с идентификатором для некоторого сервиса. Например, при использовании менеджером виртуальной памяти портала для сервиса замещения страниц приложение может заставить его использовать для своих страниц свой собственный сервис замещения страниц. В **Kea** уровень детализации адаптации определяется реализацией сервисов. Если менеджер виртуальной памяти фиксирует политику замещения страниц (вместо использования для нее портала, как в примере), никто не может ее изменить, пока не заменит весь сервис.

В системе **SPACE** [PBK91] единственной абстракцией, присутствующей в ядре, является обобщение управления исключительными ситуациями, т.е. механизм управления прерываниями. Если бы такой обобщенный механизм управления исключительными ситуациями мог бы быть реализован на аппаратном уровне, операционную систему можно было бы считать “безъядерной”.

Система **Pebble** [MBG00], так же, как и **SPACE**, поддерживает взаимодействие через домены защиты, реализованное как обобщение управления прерываниями. Как и **Kea**, **Pebble** осуществляет взаимодействие доменов через порталы и допускает реконфигурацию порталов. Порталы реализуются как обобщенные обработчики прерываний. Ядро **Pebble** содержит только код, реализующий передачу потоков от одного домена защиты другому, и небольшое количество функций поддержки режима ядра. Как и **Exokernel**, **Pebble** отдает реализацию абстракций ресурсов на уровень пользователя, но в отличие от **Exokernel**, **Pebble** обеспечивает совокупность высокоуровневых абстракций, которые реализуются компонентами пользовательского уровня, что упоминалось в разделе о статической адаптации, инициированной проектировщиком. Каждый компонент уровня пользователя выполняется в своем собственном домене защиты, изолированном аппаратными средствами защиты памяти.

4.2.1.3. Системы мандатов (Capability Systems)

Системы **Fluke** и **EROS** являются системами мандатов, которые структурированы вокруг микроядра. Здесь под мандатом понимается пара, состоящая из идентификатора объекта и набора санкционированных операций над этим объектом (его интерфейс). Дескрипторы файлов в UNIX могут

служить примером таких мандатов. В системах мандатов каждый процесс содержит мандаты и может совершать только те операции, которые санкционированы этими мандатами. Мандаты – единственные средства инициации операций над объектами, и единственные операции, которые могут выполняться с помощью мандата — это операции, разрешенные этим мандатом. Это означает, что каждый ресурс обслуживается через посредника и полностью инкапсулирован.

Архитектура вложенных процессов системы **Fluke** сочетает микроядро с виртуальными машинами [FHL96]. Ядро обеспечивает базовые сервисы и интерфейс к ним. Виртуальные машины используют этот интерфейс и реэкспортируют его на следующий уровень. Каждый слой полностью симулирует среду для вышележащего уровня – интерфейс между слоями всегда один и тот же, что позволяет компоновать сервисы с помощью наложения (stacking) виртуальных машин. Благодаря такому наложению или многоуровневому представлению **Fluke** поддерживает вертикальную декомпозицию сервисов, в то время как микроядро обеспечивает горизонтальную декомпозицию, перенося традиционную функциональность ядра на серверы пользовательского уровня, расположенные как бы рядом (side-by-side).

Необходимость поддерживать согласованность интерфейсов между виртуальными машинами вносит большие трудности при добавлении новых методов или параметров. Несмотря на то, что архитектура **Pebble** близка к архитектуре вложенных процессов системы **Fluke**, **Pebble** дает возможность расширять систему с большим уровнем детализации благодаря механизму замещения порталов.

Система **EROS** (Extremely Reliable Operating System) состоит из ядра, которое реализует небольшой набор примитивных мандатных типов [SSF99]. Возможности, которые предлагает ядро EROS, относятся к довольно низкому уровню. Большинство системных функций реализуется приложениями уровня пользователя. Например, ядро EROS напрямую предоставляет страницы дисковой памяти, но не файловой системы. Файловая абстракция полностью строится на уровне приложений, и файловое приложение просто хранит содержимое файла в адресном пространстве, увеличивая адресное пространство по мере необходимости так, чтобы оно могло содержать весь файл. Обязанность файлового приложения состоит в том, чтобы реализовать такие операции, как чтение и запись, которые выполняются над файлом. Такой метод проектирования – создание высокоуровневых функций за счет объединения базовых примитивов операционной системы в повторно используемые компоненты – основная стратегия для создания приложений EROS. Каждый экземпляр компонента реализуется с помощью отдельного процесса, а ядро обеспечивает высокопроизводительный механизм связи между процессами, позволяющий эффективно объединять эти компоненты. Фактически, крайне редко приложения EROS работают с предлагаемыми ядром объектами напрямую. Большинство приложений повторно использует компоненты, предоставляемые системой, или реализуют новые компоненты, которые обеспечивают необходимую функцию структурированным способом.

Приложения **EROS** структурируются как защищенные, связанные мандатами компоненты. Каждый экземпляр компонента работает с индивидуально указанными мандатами, определяющими его полномочия. Мандаты защищены ядром, как и объекты, на которые они указывают. Единственные операции, которые могут выполняться с мандатом – это операции, определяемые объектом. Благодаря этому сочетанию защиты и посредничества, приложение, которое выполняет злоумышленный код, не может повредить систему в целом или нанести ущерб другим пользователям, а также не может использовать привилегии пользователя для того, чтобы повредить другие части пользовательской среды. Точно также, мандаты управляют доступом к ресурсам, не позволяя злоумышленному коду злоупотреблять ресурсами или вывести остальную часть системы из работоспособного состояния.

4.2.1.4. Операционные системы с кэшированием

Cache Kernel – это ядро операционной системы V++ [CD94]. В этой системе приложения выполняются на верхушке ядер приложений, либо в том же самом, либо в другом адресном пространстве. Ядра приложений реализуют сервисы операционной системы. Они выполняются на уровне пользователя и обеспечивают загрузку и разгрузку объектов ОС (потоков, адресных пространств и других ядер приложений) в соответствии с их собственными политиками. Собственно ядро **Cache Kernel** функционирует как кэш для таких объектов. В его интерфейс включены операции для загрузки и разгрузки системных объектов, и использует сигналы для указания о том, должен ли некоторый объект быть загружен или разгружен.

4.2.1.5. Рефлективные операционные системы

Рефлективные операционные системы вводят явную парадигму, которая дает возможность приложениям динамически настраивать свою среду выполнения. Приложение должно быть явно структурировано как совокупность объектов, а системные сервисы представляются в виде совокупности мета-объектов.

MetaOS – это объектно-ориентированная ОС с использованием языка Java [HPM98]. Архитектура этой ОС состоит из трех уровней. Объекты приложений располагаются на базовом уровне. На уровне ниже находятся мета-объекты, динамически сгруппированные в мета-просранства. Каждое мета-просранство поддерживает ряд приложений с похожими требованиями. Этот уровень носит название мета-уровня. В самом низу находится мета-мета-уровень, который сжат до единственного мета-просранства – главного (master) мета-просранства. Это мета-просранство распределяет ресурсы согласно динамически замещаемой политике. **MetaOS** использует открытую реализацию для поддержки определения и конструирования объектов, мета-объектов и их интерфейсов. Через эти интерфейсы мета-просранства могут быть адаптированы и расширены динамическим и безопасным образом. Когда приложение начинает выполняться, оно выбирает наиболее подходящее доступное мета-просранство, в котором оно может инициировать ряд изменений с большим уровнем детализации. Если этого окажется

недостаточно, приложение может клонировать мета-просранство и мигрировать в него. После этого приложение будет иметь полный контроль над мета-просранством и, таким образом, над собственной средой выполнения.

4.2.2. Адаптация на уровне ядра

Системы, допускающие адаптацию на уровне ядра, обычно называются наращиваемыми ядрами (extensible kernels). Эти системы применяют передовые технологии, которые гарантируют целостность ядра. Наращиваемые ядра принимают код пользователя и выполняют его в привилегированном безопасном режиме, динамически изменяя поведение ядра. Эта технология позволяет избавиться от переключений между режимами пользователя и ядра и между адресными пространствами. Поскольку коду, который вводится приложением в ядро, нельзя доверять, решающим фактором для наращиваемых ядер становится применяемый механизм безопасности.

Политика безопасности в ОС проводится в жизнь через верификацию или через защиту (protection). Защита пытается гарантировать корректное поведение приложения после его инсталляции, или ограничить последствия нанесенного вреда. Защита достигается как аппаратными, так и программными средствами.

Защита с помощью аппаратуры обычно осуществляется в микроядрах. В таких системах аппаратные средства гарантируют, что настройки системы, происходящие на уровне пользователя, никогда не смогут модифицировать ядро.

Верификация гарантирует корректное поведение расширения до инсталляции и развертывания. При обсуждении ОС рассматриваются два вида верификации – верификация источника и верификация поведения.

При верификации источника код считается безопасным, если он введен в систему удостоверенной стороной (например, администратором). Такая практика применяется в промышленных ОС (UNIX, Windows NT), где такой подход носит название метода загружаемого модуля ядра. Загрузка модуля может выполняться как администратором, так и приложением с правами root, или даже приложениями с правами пользователя, которому разрешена такая загрузка, если он является удостоверенной третьей стороной (например, производитель данной ОС).

При верификации поведения ОС старается проанализировать, действительно ли данный код ведет себя должным образом. Очень привлекательна автоматическая верификация поведения, но ее трудно осуществлять.

4.2.2.1. Программная защита

В системах с наращиваемыми ядрами полагаться на аппаратную защиту невозможно, потому что расширение (ненадежный код) обладает теми же полномочиями, что и ядро. Здесь необходимо применять программную защиту. Наиболее распространенными подходами к программной защите являются: программная локализация неисправностей и безопасные языки.

.4.2.2.1.1 Программная локализация неисправностей

Программная локализация неисправностей обеспечивает защиту памяти в едином адресном пространстве (например, внутри ядра) [WLA93]. Она осуществляется в два этапа. Сначала ненадежный код загружается в собственный изолированный домен (так называемый “неисправный” домен), который является областью памяти, логически разделенной с ядром. Затем код модифицируется таким образом, что из него нельзя осуществить запись или выполнить команду перехода за пределы этого изолированного домена. Одним из способов реализации этого подхода является sandboxing (механизм обеспечения безопасности подкачанных из сети или полученных по электронной почте программ, предусматривающий изоляцию на время выполнения загружаемого кода в ограниченную среду – "песочницу"). Изолированный домен состоит из сегмента кода и сегмента данных. В старших битах адреса содержится идентификатор сегмента. Перед каждой ненадежной инструкцией в сегменте кода вставляются инструкции, которые записывают в старшие биты адреса в ненадежной инструкции идентификатор изолированного сегмента, не давая ей возможности обратиться по адресу за пределы этого домена. Взаимодействие между изолированными сегментами осуществляется через RPC-интерфейс.

Система **VINO** [SS97] применяет программную локализацию неисправностей. В этой системе каждое расширение в ядре имеет свои собственные стек и область памяти. Защиту памяти гарантирует программная локализация неисправностей. Кроме того, **VINO** использует систему облегченных транзакций для управления выполнением расширений и использованием ресурсов. Наращиваемость в **VINO** можно осуществить двумя способами:

- приложение может заместить реализацию метода некоторого объекта ядра (ресурса), если это разрешено, – это позволяет изменять стандартное поведение ресурсов,
- приложение может зарегистрировать в ядре обработчик некоего события, такого как подключение в сети к конкретному порту, что позволяет инсталлировать новые сервисы в ядре.

К недостаткам метода программной локализации неисправностей можно отнести то обстоятельство, что каждый раз перед выполнением ненадежной инструкции должны выполняться накладные инструкции.

.4.2.2.1.2 Безопасные языки

Другим распространенным способом сохранения целостности ядра является наложение ограничений на абстракции языка программирования. Наиболее интересным механизмом такого типа является метод проверки типов (type checking), который подразумевает, что безопасные языки являются типизированными и обладают типовой безопасностью.

Безопасные языки, широко используемые в исследовательских проектах, – это Modula-3, Java и ML. Язык ML имеет формальную типовую систему, известную как систему Hindley-Milner, и дает возможность осуществлять

статическую проверку типов. Языки Modula-3 и Java обладают меньшим формализмом, поэтому чтобы гарантировать в них политику безопасности, необходимо выполнять многочисленные проверки типов во время выполнения. Однако ML, как декларативный язык, имеет недостаточную эффективность во время выполнения.

Система **SPIN** основана на языке Modula-3 [BSP95]. Все взаимодействия между приложением и ядром осуществляются с помощью расширений. Каждое рас-ширение связывается с событием. Расширение должно быть зарегистрировано диспетчером, который инсталлирует расширения и передает события расширениям. С отдельным событием может быть связано несколько расширений. Расширения замещаются или добавляются диспетчером. Modula-3, в основном, используется для гарантирования защиты памяти. Дополнительные ограничения накладываются диспетчером и стандартными расширениями. Динамический компоновщик гарантирует, что расширение видит только санкционированные события.

Очевидным недостатком таких систем является их жесткая привязанность к определенному языку – весь системный код и расширения должны быть написаны на этом языке. Еще один недостаток состоит в том, что политика безопасности фиксирована и определяется выбранным языком. Кроме того, многочисленные проверки во время выполнения сильно понижают производительность системы.

4.2.2.2. Автоматическая верификация

Метод автоматической верификации основан на представлении кода в определенном формате, называемом PCC (Proof-Carrying Code) [Nec97]. PCC-модуль содержит формальное доказательство соответствия кода данной политике безопасности. Истинность доказательства гарантирует безопасность кода, и программный модуль может выполняться без проверок во время выполнения. Политика безопасности ядра описывается с помощью аксиом и правил вывода в доказательствах, а также формулируется в виде предикатов логики первого порядка для каждой инструкции, в которых указывается, при каких обстоятельствах выполнения каждой инструкции будет оставаться безопасным. Приложение использует предикаты политики безопасности для вычисления предиката безопасности. Затем доказывается безопасность этого предиката по правилам логики первого порядка с использованием аксиом и правил вывода политики безопасности. Доказательство присоединяется к расширению. Ядро, в свою очередь, также вычисляет предикат безопасности и проверяет, истинность ассоциированного доказательства для этого предиката. Проверка достоверности может быть сделана через простую и эффективную проверку типов (type checking).

Несмотря на привлекательность этого подхода, применимость его пока остается проблемой. Большие трудности возникают при попытках автоматизировать генерацию доказательств и использовать доказательства для языков высокого уровня.

4.3. Автоматическая адаптация

Автоматической адаптацией является адаптация, инициированная самой ОС. Можно рассматривать переносимость, реализуемую через условную компиляцию, как статическую форму автоматической адаптации. Обнаружив, на какой платформе операционная система должна быть скомпонована, системы сама способна конфигурироваться, например, с помощью С препроцессора.

Наиболее интересную категорию составляют ОС, которые сами динамически или статически адаптируются к выполняющимся приложениям. С этой целью ОС должна быть способна отслеживать и анализировать приложения и автоматически изменять свое поведение, чтобы поддерживать приложения наилучшим образом. Промышленные ОС обычно поддерживают ограниченную форму динамической автоматической адаптации в специфических и хорошо понятных подсистемах, например, в файловой системе, которая может контролировать поведение пользовательских приложений для оптимизации своей производительности.

Создание автоматической динамически настраиваемой ОС общего назначения можно рассматривать как конечную цель исследований в области настраиваемости ОС. Однако в связи с трудностями, возникающими при компоновке таких систем, пока еще ничего не слышно об автоматических системах в полном смысле этого слова. Пока можно говорить только о нескольких, обсуждаемых далее, проектах.

Система **Synthetix** предназначена для обеспечения специализированных реализаций сервисов операционной системы, генерируемых во время выполнения, на основании частичной оценки [CBK96]. Степень детализации и широта настраиваемости ограничены – проектировщик решает, какой сервис может быть конкретизирован и выбирает параметры конкретизации. Параметры сервиса вводятся с помощью инвариантов, с которыми связаны блоки защиты (guards). После проверки корректности параметров модуль сервиса замещается реализацией с новыми параметрами. Например, системный вызов открытия файла может возвращать конкретизированный код, обеспечивающий чтение файла. Такой код мог бы иметь инварианты, такие как размер блока на диске, последовательный доступ, монопольный доступ и т.п. Когда тот же самый файл позже открывается другим приложением, инвариант монопольного доступа становится некорректным из-за нарушения блока защиты, связанного с системным вызовом открытия файла.

Автоматический подход к настраиваемости исследовался в проекте **VINO** [SS97], который уже рассматривался выше, однако этот подход в **VINO** не был реализован. По мнению авторов для осуществления автоматической адаптации поведения системы необходимо получать сведения из следующих трех источников:

- периодическая статистика от каждой подсистемы **VINO,**
- специализированный компилятор,
- трассы и журналы, которые регистрируют входящие запросы и произведенные результаты.

Вся информация собирается в реальных обстоятельствах, во время выполнения приложения. Затем эта информация анализируется с целью обнаружения чрезвычайных обстоятельств – например, ситуаций, когда потребление ресурсов превышает ожидаемую норму. В таких ситуациях адаптация проводится согласно известным эвристикам. Например, пусть некое приложение интенсивно листает страницы, тогда создаются трассы для запрашиваемых страниц. Результирующие трассы исследуются на предмет совпадения с хорошо известными моделями подкачки страниц. Если такая модель находится, инсталлируется соответствующий алгоритм. Проект **VINO** интересен тем, что в нем используются эвристики для хорошо известных случаев. В отличие от системы **Synthetix**, в которой адаптируются только функции и параметры, определенные проектировщиком, **VINO** поддерживает механизм, с помощью которого система могла бы разрабатывать и тестировать новые алгоритмы для вновь возникающих проблем.

5. Сводные таблицы характеристик свойств ОСРВ

Ниже следуют 4 таблицы.

ОСРВ	Архитектура	Предсказуемая произ-вод-ть реального времени	Что реализует микроядро, размер (мин., мах.)
VxWorks	Клиент-сервер, микроядро WIND Microkernel	Приоритетное планирование в двух вариантах, наследование приоритетов	Многозадачность, планирование, переключение контекста, взаимодействие /синхронизация задач, управление разделяемой и динамической памятью, управление прерываниями
QNX	Клиент-сервер, микроядро и взаимодействующие процессы	Приоритетное планирование с выбором методов планирования. Распределение. Наследование приоритетов	Потоки, сигналы, передачу сообщений, синхронизацию, планирование, временные сервисы
Windows CE	Модульная с ядром и необязат. компонентами	Приоритетное планирование	
pSOS	Клиент-сервер, отсутствует протокол взаимодействия на основе сообщений, вместо него используется прогр. шина	Приоритетное планирование, отсутствует наследование приоритетов	
ChorusOS	Многослойная	Приоритетное планирование, мъютексы реального времени, таймеры с высокой разр. способностью, MIPC	Многозадачность, поддержка актора, управление потоками, управление LAP, управление исключит. ситуациями, минимальное управления прерываниями
OSE	Многослойная	Приоритетное планирование, механизм предотвращения инверсии приоритетов	Приоритетное планирование, асинхронная передача сообщений, управление памятью, размер – 6К, 80К
OS-9		Приоритетное планирование, механизм предотвращения инверсии	размер – 128К, 4MB
C EXECUTIVE			размер – 5К, 22К
CMX-RTX		Приоритетное планирование, механизм предотвращения инверсии	размер – 1К, 6К
Inferno			
INTEGRITY		Приоритетное планирование, механизм предотвращения инверсии	размер –70К
INtime			
LynxOS			размер –280К, 4М
Nucleus			
RTX		Приоритетное планирование, наследование приоритетов	
CORTEX			
DeltaOS			размер – 10К

ОСРВ	Распределенная обработка	Сетевые протоколы	Файловые системы
VxWorks		TCP/IP, FTP, SMTP, NFS, PPP, RPC, Telnet, BSD 4.4 TCP/IP networking,IP, IGMP, CIDR, TCP, UDP, ARP, RIP v.1/v.2, Standard Berkeley sockets, zbufs, SLIP, CSLIP, BOOTP, DNS, DHCP, TFTP, NFS, ONC RPC, WindNet SNMP v.1/v.2c with MIB compiler - optional, WindNet OSPF	DOS-FS NFS TrueFFS
QNX	Прозрачный доступ к удаленным ресурсам. Упрощенное проектирование отказоустойч. кластеров	TCP/IP, FTP, SMTP, SNMP, NFS, PPP, ATM, ISDN, RPC, Telnet, Bootp, tiny TCP/IP	RAM, Flash, QNX, Linux, DOS, CD-ROM, DVD, NFS, CIFS
Windows CE			
PSOS			
ChorusOS	Прозрачный доступ к удаленным ресурсам	IPv4, IPv6, PPP, NTP, BFP, DHCP NFS, RPC, LDAP, FTP, Telnet	UFS, FIFOFS, NFS, MSDOSFS, ISOFS, PROCFS, PDEVFS
OSE	Прозрачный доступ к удаленным ресурсам	TCP/IP, FTP, SMTP, SNMP, PPP, ATM, ISDN, X25, Telnet, Bootp, http-server, FTP/TFTP, NTP, various routing protocols, others	FAT, VFAT, FAT32
OS-9		TCP/IP, FTP, SMTP, SNMP, NFS, PPP, ATM, ISDN, X25, RPC, Telnet, Bootp, 802.11	
C EXECUTIVE		TCP/IP, SNMP, PPP, SNMP	
CMX-RTX		TCP/IP, FTP, SMTP, SNMP, NFS, PPP, Telnet, Bootp	
Inferno		TCP/IP, FTP, PPP, Telnet, Bootp	
INTEGRITY		TCP/IP, FTP, SMTP, SNMP, NFS, PPP, ATM, X25, RPC, Telnet, Bootp, http, pop3, IGMP, UDP, ARP, RIP, sockets, zero-copy stack, tftp	
Intime		TCP/IP	
LynxOS		TCP/IP, SNMP, NFS, другие	
Nucleus		TCP/IP, SMTP, SNMP, PPP, Telnet	
RTX		TCP/IP, все протоколы, поддерж. Windows	
CORTEX		TCP/IP	
DeltaOS		TCP/IP, FTP, SMTP, PPP, WAP, HTTP, HTML, XML, OSPF2, RIP2, CORBA	

ОСРВ	POSIX	Среда разработки	Целевые платформы
VxWorks	POSIX 1003.1, .1b, .1c (включая pThreads)		x86, PowerPC, ARM, MIPS, 68K, CPU 32, ColdFire, MCORE, Pentium, i960, SH, SPARC, NEC V8xx, M32 R/D, RAD6000, ST 20, TriCore
QNX	POSIX 1003.1-2001, с потоками и расш. PB	Windows, Solaris, Self-Hosted, QNX4, Linux	ARM, MIPS, PowerPC, SH4, Strong ARM, XScale, x86
Windows CE			ARMV4, SH3, SH4, MIPS, X86
pSOS			
ChorusOS	POSIX-сигналы, сигналы реального времени, потоки, таймеры, очереди сообщений, семафоры. сокеты, разделяемая память		UltraSPARC II (CP1500 и CP20x0), Intel x86, Pentium, Motorola PowerPC 750 и 74x0 (mpc7xx), Motorola PowerQUICC I (mpc8xx) и PowerQUICC II (mpc8260)
OSE		Windows, Solaris, Linux	PowerPC, ARM, MIPS, StrongARM, Intel IXP2400, TI OMAP ARM7/C55, PowerQUICC, XScale, M-Core, Coldfire, Infineon C16x, Xc16x, E-Gold, Tricore, NEC V850, Atmel AVR, Mitusbishi M16C, Intel 8051, DSPs (TI C5/C6, Starcore, Agere 16k, LSI Logic ZSP, TigerShark, ST Micro)
OS-9		Windows	Motorola 68K, ARM/StrongARM, Intel IXP1200 Network Processor, MIPS, PowerPC, Hitachi SuperH, x86 or Intel Pentium, Intel IXC1100 XScale
C EXECUTIVE		Windows, Solaris, любая	x86, PowerPC, ARM, MIPS, 68K, i960,SH,TI
CMX-RTX		Windows	x86, PowerPC, ARM, MIPS, виртуально все 8-, 16-, 32-бит. процессоры
Inferno		Windows, Solaris, Linux	x86, PowerPC, ARM, MIPS, Sparc
INTEGRITY	POSIX 1003.1-2003	Windows, Solaris, Linux, HPUX	x86, PowerPC, ARM, MIPS, ColdFire, StrongARMXScale
INtime		Windows	x86
LynxOS	POSIX.1/.1b/.1c	Sun Solaris, SunOS, RS6000, LynxOS Native/Hosted	x86, 68k, PPC, microSPARC, microSPARC II, PA-RISC
Nucleus		Windows	x86, PowerPC, ARM, MIPS, Nios, Nios II, ColdFire, 68k, H8S, SH, DSP, OMAP, XScale, MCore
RTX		Windows	x86
CORTEX		Windows, Solaris, Linux	Hitach H8/300H, H8/S и SH-1/2/3, TI TMS320C3X, POSIX.4 (SUN SPARC)
DeltaOS		Windows, Linux	x86, PowerPC, ARM, MIPS, Dragonball

Таблица 1. Характеристики ОСРВ.

ОСРВ	Модель	Число уровней приор.	Max. число задач	Политики планирования	Состояния процесса/ потока	Механизмы синхронизации/ взаимодействия
VxWorks	Задачи имеют 1 поток, все задачи выполняются в одном адр. пр-ве без какой-либо за-щиты. КомпонентVxVMI дает возм-ть каждой задаче выпол. в собств. адр. пр-ве	256	Ограничено размером доступной памяти	POSIX и Wind планирование, каждый вариант имеет Preemptive priority и Round-robin	9	семафоры, мьютексы, условные переменные, флаги событий, POSIX-сигналы, очереди сообщений, почт. ящики
QNX	процессы/потоки	64	4095 процессов, в каждом процес-се до 32767 потоков	FIFO с приоритетами, циклическое, адаптивное, спорадическое планирование	14	передача сообщений (очереди и почт. ящики), семафоры, мьютексы, флаги событий, сигналы POSIX
Windows CE	процессы/потоки нити (fiber), неуправляемые ядром	256	32 процесса, число потоков внутри процесса ограничено доступной RAM	с приоритетами, циклическое между потоками на одном приоритетном уров-не, если квант установлен в 0, поток выполняется до завершения	5 выпол-ся (running), приос-нный (suspended), спящий (sleeping), заблокированный (blocked), завершенный (terminated)	критические секции, мьютексы, семафоры, условные переменные, события, передача сообщений (очереди, почт. ящики), сигналы POSIX
pSOS	только потоки	256	Ограничено памятью	FIFO с приоритетами, циклическое	4) создана (created), готова (ready), выполняется (running), заблокиров. (blocked)	семафоры, флаги событий, сигналы POSIX, очереди сообщений
ChorusOS	процессы/ акторы/потоки			FIFO с приоритетами, циклическое, планирование реального времени, опция одновременного выполнения различных политик планирования, возможность создания собственного планировщика		мьютексы, мьютексы реального времени, семафоры, флаги событий, LAP (Local Access Point), IPC (Inter-Process Communication) – сообщения, порты, группы портов, MIPC (почт. ящики), разделяемая память, очереди сообщений
OSE		32		FIFO с приорит.		
OS-9	процессы/потоки	65535		С приоритетами		
C EXECUTIVE		32000		FIFO с приоритет., квантование времени		

ОСРВ	Модель	Число уровней приор.	Max. число задач	Политики планирования	Состояния процесса/ потока	Механизмы синхронизации/ взаимодействия
CMX-RTX				FIFO с приоритет., циклическое с приоритет.		
INTEGRITY		255		Цикл. с приорит., ARINC 653		семафоры, мьютексы,
INtime		255		FIFO с приоритет., циклическое с приоритет.		
LynxOS		512		FIFO с приоритет., циклическое с приоритет., фикс. приоритеты, квантование времени, дин. приоритеты		
RTX		128				
CORTEX		62		FIFO с приоритет., циклическое с приоритет., разделение времени, другие		мьютексы и условия, мониторы и условия, вычислительные семафоры, события
DeltaOS		256				

Таблица 2. Характеристики многозадачной обработки.

ОСРВ	Модель защиты	Поддержка MMU	Виртуальная память	Подкачка	Вызов стр. по запросу
VxWorks	-без защиты -защита виртуальной памяти (VxVMI)	не требуется, но поддерживается для VxVMI	да (для VxVMI)	нет	нет
QNX	защита виртуальной памяти	да	да	да	нет
Windows CE	- защита виртуал. памяти - без защиты	да или нет (зависит от конфигурации)	да	да, но можно запрет.	да, но можно запрет.
pSOS	- без защиты, - защита кода, данных и пр-ва стека с пом. биб. ф-ций (2 вар.–регионы и разделы)	не требуется	нет	нет	нет
ChorusOS	-без защиты, -защищенная память, -защита виртуал. памяти	да или нет (зависит от конфигурации)	да	да	да
OSE		да			
OS-9		да			
C EXECUTIVE		нет			
CMX-RTX		да			
INTEGRITY		да			
INtime		да			
LynxOS		да			
RTX		да			

Таблица 3. Характеристики управления памятью.

ОСРВ	Управление прерываниями				Управление временем
	Прерывания	Контекст	Стек	Взаимодействие прерываний с задачами	
VxWorks	Вложенные, с приоритетами	Обработчики прерываний выполняются в отдельном контексте	Спец. стек для прерываний. Если архитектура этого не позволяет, тогда используется стек прерванной задачи	-разделяемая память и цикл. буфера -семафоры -очер. сообщений -каналы -сигналы	Часы (clock), интервальный таймер
QNX	Вложенные, с приоритетами	Прерывание выполняется в контексте потока	Прерывание имеет свой собственный стек	Сигналы и импульсы	Часы (clock), интервальный таймер
Windows CE	Вложенные, с приоритетами. IST используется для обработки большинства прерываний	ISR вып. в спец. Ко-нтексте, при этом ISR использует виртуальные адреса, стат. отоб. OEM. IST выс-пает как обычный поток приложения и имеет свой соб. контекст и приоритет.	IST выступает как обычный поток приложения и имеет свой собственный стек	Из ISR можно подать сигнал в IST только с помощью события. OEM может создать область разделяемой памяти с помощью статического отображения области памяти в адресное пространство ISR.	Часы (clock), интервальный таймер
pSOS	Вложенные, с приоритетами	Прерывание выполняется в контексте потока	Стек ядра или стек прер. в зав-ти от целевой платфор.	Через объекты взаимодействия и синхронизации	Часы (clock), интервальный таймер
ChorusOS		Обработчики прерываний выполняются в отдельном контексте		Флаги событий, MIPC	Универсальное интерв. время, вирт. таймер, унив. время. часы истинного времени, сторожевой таймер, оценочный таймер

Таблица 4. Характеристики управления прерываниями, синхронизацией и временем различных ОСРВ.

Приложение А. Перечень сокращений

API – программный интерфейс приложений.

BSP – Board Support Package – комплект конфигурационных и инициализационных модулей.

CPU – центральный процессор.

DSP – Digital Signal Processor.

EDF – Earliest Deadline First – динамические алгоритмы планирования.

FIFO – First in First Out – политика планирования обработки процессов по принципу “первым прибыл, первым обслужен”.

GUI – графический пользовательский интерфейс.

IPC – Interprocess Communication – межпроцессное взаимодействие (возможность операционной системы, позволяющая задачам или процессам обмениваться данными между собой, методы IPC включает в себя каналы, семафоры, разделение памяти, очереди, сигналы и почтовые ящики).

ISR – interrupt servicing routine – программа обработки прерывания (программа низкого уровня в ядре с ограниченными системными вызовами).

IST – interrupt servicing thread – поток обработки прерывания (поток уровня приложения, который управляет прерыванием, с доступом ко всем системным вызовам).

MMU – memory management unit – специальное аппаратное устройство для поддержки управления виртуальной памятью.

NFS – Network File System.

OLE – Object Linking and Embedding – связывание и внедрение объектов. С помощью этой технологии приложения могут обмениваться информацией с другими приложениями через стандартные интерфейсы, доступ к которым возможен из множества различных языков программирования.

OEM – original equipment manufacturer.

RAM – random access memory – память (запоминающее устройство) с произвольной выборкой; оперативное запоминающее устройство, ОЗУ.

RMS – Rate Monotonic Scheduling – статические алгоритмы планирования.

ROM – read-only memory – постоянная память, постоянное запоминающее устройство, ПЗУ.

RRS – round-robin scheduling – циклическое планирование.

RTAPI – Real-Time Application Programming Interface.

RTOS – Realtime Operating System.

RTX – Real Time Extension.

SNMP – Simple Network Management Protocol.

SRT – soft real-time.

UART – universal asynchronous receiver-transmitter, модуль асинхронной последовательной передачей данных

Приложение B. Терминология

Condition variables – переменные состояния.

Deadline – директивный срок задачи, до которого задача должна обязательно (для систем мягкого реального времени – желательно) выполниться.

Deadline-driven scheduling – политика планирования, управляемая дедлайнами.

Host – инструментальный компьютер.

Interrupt Latence Time – время задержки прерывания.

Kernel или nucleus – микроядро

Light-weight process – подпроцесс или легковесный процесс.

Paging – страничная организация памяти.

Pre-emptable OS – ОС, допускающая вытеснение.

Preemption – приоритетное прерывание обслуживания.

Scheduling – планирование задач.

Spawn – порождение нового процесса.

Target – целевой компьютер.

Thread – поток.

Time slicing – квантование времени.

Timeliness – своевременность.

Ticker – часовой механизм.

Инструментальный компьютер – host.

Квантование времени – time slicing.

Микроядро – kernel или nucleus.

Планирование задач – scheduling

Подпроцесс или легковесный процесс – light-weight process.

Политика планирования, управляемая дедлайнами – deadline-driven scheduling.

Поток – thread.

Приоритетное прерывание обслуживания – preemption.

Своевременность – timeliness.

Страничная организация памяти – paging.

Целевой компьютер – target.

Циклическое планирование – round-robin scheduling

Литература

[2K] http://choices.cs.uiuc.edu/2K/

[CEXEC] http://www.jmi.com/

[CHORUSOS] http://docs.sun.com/app/docs/prod/software#hic

[CMXRTX] http://www.cmx.com/rtx.htm

[CMXTINY] http://www.cmx.com/tiny.htm

[DEDSYS] http://www.dedicated-systems.com/

[DO178B] http://www.rtca.org/

[EPOC] http://www.symbian.com/about/symb-os.html

[INFERNO] http://www.vitanuova.com/inferno/index.html

[INTEGRITY] http://www.ghs.com/products/rtos/integrity.html

[INTIME] http://www.tenasys.com/intime.html

[ITRON] http://www.sakamura-lab.org/TRON/ITRON/home-e.html

[LynxOS] http://www.lynuxworks.com/

[MSEmb] http://msdn.microsoft.com/embedded/

[NUCLEUS] http://www.acceleratedtechnology.com/embedded/nuc_rtos.html

[OS-9] http://www.microware.com/

[OSEK] http://www.osek-vdx.org/

[OSERTOS] http://www.ose.com/

[PALMOS] http://www.palmsource.com/palmos/

[PSOS] http://www.windriver.com/products/device_technologies/os/psosystem_3/

[QNX] http://www.qnx.com/

[RTEMS] http://www.rtems.com/

[RTX] http://www.vci.com/

[VxWorks] http://www.windriver.com/products/device_technologies/os/

[Velosity] http://www.ghs.com/products/velosity.html

[VSPWorks] Wind River VSPWorks, technical brief http://www.transtech-dsp.com/datasheets/VSPWorks_TechBr.pdf

[WinCE] http://msdn.microsoft.com/embedded/windowsce/

[BSP95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chanbers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. Proc. of the 15th ACM Symposium on Operating Systems Principles, 1995.

[Beu99] D. Beuche et al. The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. Proc. 2nd IEEE Int'l Symp. Object-Oriented Real-Time Distributed Computing, IEEE Press, Piscataway, N.J., 1999.

[CC99] Common Criteria for Information Technology Security Evaluation (CC), Version 2.1, 1999. ISO/IEC 15408. http://csrc.nist.gov/cc/CC-v2.1.html.

[CIR93] R. Campbell, N. Islam, D. Raila, and P. Madany. Designing and implementing Choices: an object-oriented system in C++. Commun. ACM 36, 9, 117–126, 1993.

[CRJ87] R. Campbell, V. Russo, and G. M. Jonston. The design of a multiprocessor operating system. Proceedings of the USENIX C++ Workshop, 109–125, 1987.

[CD94] D.R. Cheriton, and K.J. Duda. A caching model of operating system kernel functionality. In Operating Systems Design and Implementation. 179–193, 1994.

[CBK96] C. Cowan, A. Black, C. Krasic, C. Pu, J. Walpole, C. Consel, and E. Volanschi. Specialization classes: An object framework for specialization. Proc. of the 5th International Workshop on Object-Orientation in Operating Systems (IWOOOS '96), 1996.

[EKO95] D. R. Engler, F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. Proc. of the 15th ACM Symposium on Operating Systems Principles. 251–266, 1995.

[DGV04] A. Dunkels, B. Grönvall, T. Voigt. Contiki – A Lightweight and Flexible Operating System for Tiny Networked Sensors. 29th Annual IEEE International Conference on Local Computer Networks (LCN'04), pp. 455-462, November 16-18, 2004, Tampa, Florida, USA.

[DoD85] Department of Defense Trusted Computer System Evaluation Criteria. – DoD 5200.28 – STD, December 26, 1985.

[DPM02] G. Denys, F. Piessens, and F. Matthijs. A survey of customizability in operating systems research. ACM Computing Surveys (CSUR), 34(4):450-468, December 2002.

[FBB97] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The flux OSKit: A substrate for kernel and language research. Proc, of the 16th ACM Symposium on Operating Systems Principles. 38-51, 1997.

[FHL96] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. Proc. of the 2nd Symposium on Operating Systems Design and Implementation. 137–151, 1996.

[HPM98] M. Horie, J. Pang, E. Manning, and G. Shoja. Using meta-interfaces to support secure dynamic system reconfiguration. Proc. of the 4th International Conference on Configurable Distributed Systems (ICCDS'98), 1998.

[HSW00] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In Proc. ASPLOS-IX, November 2000.

[Jbed98] Oberon Microsystems, Jbed Whitepaper: Component Software and Real-time Computing, tech. report, 1998; http://www.oberon.ch .

[KLM93] G. Kiczales, J. Lamping, C. Maeda, D. Keppel, and D. McNamee. The need for customizable operating systems. Proc. of the 4th Workshop on Workstation Operating Systems, 1993.

[LC02] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In Proc. ASPLOS-X, October 2002.

[LL73] C. L. Liu, J. W. Layland. Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environment. Journal of the Association for Computing Machinery 20, 1 (January 1973): 40-61.

[Lie93] J. Liedtke. Improving IPC by kernel design. Proc. of the 14th ACM Symposium on Operating System Principles (SOSP), 1993.

[Lie95] J. Liedtke. On microkernel construction. Proc. of the 15th ACM Symposium on Operating System Principles, 1995.

[Lie96] J. Liedtke. Toward real microkernels. Commun. ACM 39, 9, 70–77, 1996.

[POSIX] IEEE Std 1003.1, 2004 Edition. The Open Group Technical Standard. Base Specifications, Issue 6. Включает IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. System Interfaces.

[MBG00] K. Magoutis, J. C. Brustoloni, E. Garber, W. T. Ng, and A. Silberschatz. Building appliances out of reusable components using pebble. Proc. of the 9th ACM SIGOPS European Workshop, 2000.

[MMO95] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, and T. A. Proebsting. Scout: A communications-oriented operating system. Proc. of the 5th Workshop on Hot Topics in Operating Systems. 1995.

[Nec97] G. C. Necula. Proof-carrying code. In Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 106–119, 1997.

[PBK91] D. Probert, J. Bruno, and M. Karzaorman. Space: a new approach to operating system abstraction. Proc. of the International Workshop on Object Orientation in Operating Systems. 133–137, 1991.

[POSIXTestSuite] National Institute of Standards and Technology, PCTS: 151-2, POSIX Test Suite.

[SM99] T. Saulpaugh and C. Mirho. Inside the JavaOS Operating System. Addison Wesley, Reading, Mass., 1999.

[SS97] M. I. Seltzer, and C. Small. Self-monitoring and self-adapting operating systems. Proc. of the 6th Workshop on Hot Topics in Operating Systems, 1997.

[SSF99] J. S. Shapiro, J. M. Smith, and D. J. Fabrer. EROS: a fast capability system. Proc. 17th ACM Symp. Operating Systems Principles, ACM Press, New York, 1999.

[SP97] O. Spatscheck, and L. Peterson. Escort: a path-based os security architecture. Tech. Rep. TR97-17, Dept. of Computer Science, University of Arizona, 1997.

[VH96] A. Veitch and N. Hutchinson. Kea – a dynamically extensible and configurable operating system kernel. Proc. of the 3rd Conference on Configurable Distributed Systems, 1996.

[WLA93] R. Wahbe, S. Lucco, T. E. Andersen, and S. L. Graham,. Efficient software-based fault isolation. ACM SIGOPS Operating Systems Review 27, 5 (December), 203–216, 1993.

[YN03] W. Yuan, K. Nahrstedt. Energy-Efficient Soft Real-Time CPU Scheduling for Mobile Multimedia Systems. ACM Symposium on Operating Systems Principles (SOSP), 2003.

[ZS01] K. M. Zuberi, K. G. Shin: EMERALDS: A Small-Memory Real-Time Microkernel. IEEE Trans. Software Eng. 27(10): 909-928 (2001)

Список ОСРВ, упоминающихся в печати (и сети)

µC/OS (J.J. Labrosse, *MicroC/OS-II: The Real-Time Kernel.* Lawrence, Kans.: R&D Books (Miller Freeman, Inc.), 1999)
µITRON 53
AIX (IBM, http://www-03.ibm.com/servers/aix/)
AMX (Kadak Products Ltd, http://www.kadak.com)
Ariel (Microware Systems Corp, http://www.microware.com)
ARTOS (Locamation, http://www.locamation.com)
ASP6x (DNA Enterprises, Inc., http://www.dnaent.com)
Brainstorm Object eXecutive (Brainstorm Engineering Co., http://www.braineng.com)
Byte-BOS (Byte-BOS Integrated Systems, http://www.bytebos.com)
Cache Kernel 78
C Executive 51
Chimera (The Robotics Institute Carnegie Mellon University, http://www.ri.cmu.edu/)
Choices 70
ChorusOS 25
CMX 51
Contiki 43
CORTEX 68
CREEM (GOOFEE Systems, http://goofee.com)
CRTX (StarCom, http://www.starcom.com)

DeltaOS 69
Diamond (3L Limited, http://www.3l.com)
dSPACE (dSPACE Gmbh, http://www.dSPACE.de)
eCos (eCosCentric Limited, http://www.ecoscentric.com/)
Embedded DOS 6-XL (General Software, Inc., http://www.gensw.com)
embOS (SEGGER Microcontroller Systeme GmbH, http://www.segger.com)
EMERALDS 68
EOS (Etnoteam S.p.A., http://www.etnoteam.it)
EPCOSEK (ETAS GmbH, http://www.etas.de)
EROS 81
EspresS-VM (Mantha Software, Inc., http://www.manthasoft.com)
EUROS (Dr. Kaneff Engineering Consultants, http://www.kaneff.de)
Exokernel 76
Fluke 81
Fusion RTOS (Unicoi Systems Inc., www.unicoi.com)
Granada (Ingenieursbureau B-ware, http://www2.b-ware.nl)
GRACE OS 50
Hard Hat Linux (MontaVista Software, http://www.mvista.com)
Harmony RTOS (Institute for Information Technology, National Research Council of Canada, http://www.psti.com)
Helios (Perihelion Distributed Software, http://www.perihelion.co.uk)
HP-RT (Hewlett-Packard, http://www.hp.com)
Hyperkernel (Nematron Corporation, http://www.hyperkernel.com)
Inferno 52
INTEGRITY 46
INtime (real-time Windows NT), iRMX 32
IRIX (Silicon Graphics, Inc., http://www.sgi.com)
iRMX III (TenAsys Corporation, http://www.tenasys.com/products/irmx.php)
ITRON 53
ITS OS (In Time Systems Corporation, http://www.intimesys.com)
JavaOS 69
Jbed 67
JSCP – Software Co-Processor for Java (NSI COM, http://www.nsicom.com)
Kea 80
L4 79
LynxOS 48
MacroView (VRT, http://www.vrt.com.au/products/scada/macroview.html)
MC/OS (Mercury Computer Systems, http://www.mc.com)
MetaOS 83
MotorWorks (Wind River Systems, http://www.wrs.com)
MTEX (Telenetworks, http://www.telenetworks.com)