****

Министерство образования Российской Федерации

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

*Государственное образовательное учреждение высшего профессионального образования*

*КАЗАНСКИЙ ГОСУДАРСТВЕННЫЙ ЭНЕРГЕТИЧЕСКИЙ УНИВЕРСИТЕТ*

Кафедра ПЭ

Реферат

Тема:

«Операционные системы реального времени.»

Выполнил:

студент группы ПЭ-06

Закиев А.Р.

Проверил:

Ахметвалеева Л.В

Казань 2010

Содержание

Введение . 3

1. Особенности операционных систем реального времени. 5

1.1Процессы, потоки, задачи. 5

1.2 Планирование, приоритеты. 5

1.3 Память. 7

1.4. Прерывания. 8

1.5. Часы и таймеры . 9

1.6. Стандарты ОСРВ. 9

1.7. Стандарты безопасности. 10

2. Настраиваемость операционных систем. 12

2.1. Адаптация, осуществляемая человеком. 13

2.1.1. Статическая адаптация, инициированная проектировщиком. 13

2.1.2. Динамическая адаптация, инициированная администратором. 16

2.2. Адаптация, инициированная приложением. 17

2.2.1. Адаптация с уровня приложения. 17

2.2.2. Адаптация на уровне ядра. 22

2.3. Автоматическая адаптация. 25

Заключение. 27

Список литературы. 28Введение

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. От встроенной системы не всегда 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 (в англоязычной литературе – deadline), до истечения 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 (preemptable),

- 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 во всех сценариях рабочей нагрузки системы должно быть определено максимальное время отклика.

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 построено 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. Особенности операционных систем реального времени

1.1. Процессы, потоки, задачи

Концепция многозадачности (псевдопараллелизм) 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 вопросы безопасности не принимаются во внимание. 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 в соответствии с приоритетами. 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 у EDF имеются 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, это требуется 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, контекст процесса сохраняется, и он ставится в конец очереди. Во многих ОСРВ для планирования задач на одном уровне используется RRS. 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 от системы не требуется никакого управления памятью, 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, то 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. У каждого процесса имеются свои собственные сегменты и, следовательно, своя таблица 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 к 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 для процессов реального времени техники вызова страниц по запросу (подкачка с диска). 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. В службах времени ОСРВ используются 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”, то 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 времени.

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 компания 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. ОСРВ ITRON. Этот стандарт является очень распространенным в Японии.

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.

1.7. 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).

В ряде других стран были разработаны аналогичные критерии, на основе которых был создан 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:

А1 – верифицированная разработка. Этот уровень требует, чтобы защиту секретной и другой критичной информации средствами управления безопасностью гарантировали методы формальной верификации.

В3 – домены безопасности. Этот уровень предназначен для защиты систем от опытных программистов.

В2 – структурированная защита. В систему с этим уровнем защиты нельзя допустить проникновение хакеров.

В1 – мандатный контроль доступа. Защиту этого уровня, возможно, удастся преодолеть опытному хакеру, но никак не рядовым пользователям.

С2 – дискреционный контроль доступа. Уровень С2 обеспечивает защиту процедур входа, позволяет производить контроль за событиями, имеющими отношение к безопасности, а также изолировать ресурсы.

С1 – 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 полуформальной 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, чем на уровне 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, описание процедур инсталляции и поставки, руководств администратора и пользователя, функциональное и независимое тестирование, оценку 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 С2, определявшийся в соответствии с американским стандартом TCSEC. В соответствии с оценочными уровнями доверия сертификация на соответствие более высокому уровню 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.

2. 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, допускающая 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.

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).

2.1. 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.

2.1.1Статическая 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 через систему от источника ввода/вывода 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, в которой для настраиваемости используются каркасная технология и подсистемы [CRJ87, CIR93]. Основное предназначение Choices – обеспечить пользователям возможность простой оптимизации и адаптации системы в соответствии с поведением приложений и рабочей нагрузкой. Система Choices спроектирована как иерархия каркасов, поддерживающих 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 наследования. 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-ориенентированных 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 существующими абстрациями или полностью заместить их стандартный набор. Pebble дает возможность создавать модульные ОС из компонентов многократного использования. В отличие от подобных систем, системные сервисы не интегрируются в ядро. Они предоставляются в виде удостоверенных серверных компонентов, 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 – 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.

2.1.2. 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 – 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 в ядро злоумышленного или просто ошибочного модуля.

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 и т.д.). 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 следствие этого, многие исследовательские результаты по адаптируемости и наращиванию не находят применения в промышленных ОС.

2.2. 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.

2.2.1. 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 следует предполагать отсутствие в нем каких-либо сервисов или политик.

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. На основе этих абстракциями в L4 поддерживается рекурсивная конструкция 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 представляет собой попросту абстракцию 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 выделение, возврат и мультиплексирование физических ресурсов (страниц памяти, квантов времени процессора, блоков дисков и т.п.) безопасным образом. Композиция наиболее используемых интерфейсов в Exokernel до сих пор остается большой проблемой [SSF99].

* Многоядерные ОС.

К микроядерным ОС можно отнести систему 2K [2K]. 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 основывается на технологии Corba, и в ней для адаптации используются данные метауровня и методы, которые предлагает уровень ORB (object request broker). Компонент 2K – это динамически загружаемый программный модуль, который хранится в динамически подключаемой библиотеке (DLL). Следует отметить, что в системе 2K используется крупный уровень детализации.

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 в ядре свой идентификатор, а ядро управляет доступом к интерфейсу этого сервиса. В отличие от истинных микроядерных ОС Kea не дает полной свободы для проектирования и реализации сервисов. Но все же в Kea обеспечивается возможность динамической реконфигурации. 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 некоторого сервиса. Например, при использовании менеджером виртуальной памяти портала для сервиса замещения страниц приложение может заставить его использовать для своих страниц 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.

* 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). Примером таких мандатов могут служить дескрипторы файлов в UNIX. 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 так же, мандаты управляют доступом к ресурсам, не позволяя злоумышленному коду злоупотреблять ресурсами или вывести остальную часть системы из работоспособного состояния.

* Операционные системы с кэшированием

Cache Kernel – это ядро операционной системы V++ [CD94]. В этой системе приложения выполняются поверх 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. Для указания того, должен ли некоторый объект быть загружен или разгружен, используются сигналы.

* Рефлективные операционные системы:

В рефлективных операционных системах вводится явная парадигма, 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, основанная на использовании 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-пространства. Каждое мета-пространство поддерживает ряд приложений с похожими требованиями. Этот уровень носит название мета-уровня. В самом низу находится мета-мета-уровень, который сжат до единственного мета-пространства – главного (master) мета-пространства. Это мета-пространство распределяет ресурсы согласно динамически замещаемой политике. В MetaOS используется открытая реализация для поддержки определения и конструирования объектов, мета-объектов и их интерфейсов. Через эти интерфейсы мета-пространства могут быть адаптированы и расширены динамическим и безопасным образом. Когда приложение начинает выполняться, оно выбирает наиболее подходящее доступное мета-пространство, в котором оно может инициировать ряд изменений с большим уровнем детализации. Если этого окажется недостаточно, приложение может клонировать мета-пространство и мигрировать в него. После этого приложение будет иметь полный контроль над мета-пространством и, таким образом, над собственной средой выполнения.

2.2.2. Адаптация на уровне ядра

Системы, допускающие адаптацию на уровне ядра, обычно называются наращиваемыми ядрами (extensible kernels). В этих системах применяются 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.

* 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.

* 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 (remote procedure calls).

В системе VINO [SS97] также применяется программная локализация 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 заменить 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 в ядре новые сервисы.

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

* 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, обеспечивает недостаточную эффективность во время выполнения.

Система SPIN основана на языке Modula-3 [BSP95]. Все взаимодействия между приложением и ядром осуществляются с помощью расширений. Каждое расширение 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.

* 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 выполнение 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.

2.3. 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, система сама способна конфигурироваться, например, с помощью C 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 проектах, обсуждаемых далее.

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.

Заключение

Задачи реального времени составляют одну из сложнейших и крайне важных областей применения вычислительной техники. Как правило, они связаны с контролем и управлением процессами, являющимися неотъемлемой частью современной жизни. Управление прокатными станами, роботами, движение на автомагистралях, контроль за состоянием окружающей среды, управление атомными и космическими станциями и многое другое - область задач реального времени. Эти задачи предъявляют такие требования к аппаратному и программному обеспечению, как надежность, высокая пропускная способность передающей среды в распределенных системах, своевременная реакция на внешние события и т.д. Для выполнения этих требований и создаются системы реального времени, аппаратное и программное обеспечение.

Список литературы

1. С. Ф. Баррет, Д. Дж. Пак. Встраиваемые системы. Проектирование приложений на микроконтроллерах семейства 68HC12/HCS12 с применением языка C. ДМК Пресс M. 2007г.
2. CIT FORUM - http://citforum.ru/operating\_systems/rtos/
3. СВД Встраиваемые системы. - http://www.kpda.ru/Publications/TagAll
4. Средства и системы компьютерной автоматизации. - http://www.asutp.ru/