ОПЕРАЦИОННЫЕ СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ

Горошко Егор, к.т.н., ИРЭ НАН Украины, г. Харьков. Украина. egdrago@hotbox.ru

Редакция 1.05 от 24.12.2003

1. Введение

Развитие технологий производства promPC (промышленных ПК) привели к ужесточению конкуренции на рынке систем автоматизации производства между аппаратными решениями на базе микроконтроллеров и программными комплексами, работающими под управлением специализированных операционных систем (ОС) на интегрированных promPC платформах. Темой данной работы является обзор функциональным требований к таким ОС, а также попытка уточнения терминологии, применяющейся в контексте программных комплексов автоматизации производства.

2. Определение терминологии

Одним из наиболее спорных и сложных терминов систем автоматизации является английское выражение «realtime» и соответствующее ему в русском языке понятие – «реальное время». Понятие это применяется в различных научно–технических областях и подразумевает некие действия, продолжительность которых определяется внешними процессами.

В этой работе о реальном времени мы будем говорить в контексте т.н. «систем реального времени» (СРВ). В литературе встречается достаточно большое количество определений этого понятия, но главные черты СРВ могут быть определены как комбинация следующих двух определений:

1. *Система называется системой реального времени, если правильность ее функционирования зависит не только от логической корректности вычислений, но и от времени, за которое эти вычисления производятся. То есть для событий, происходящих в такой системе, то, КОГДА эти события происходят, так же важно, как логическая корректность самих событий* [2, 3].

2. Реальное время (программное обеспечение) (IEEE 610.12 – 1990): *Относится к системе или режиму работы, в котором вычисления проводятся в течение времени, определяемого внешним процессом, с целью управления или мониторинга внешнего процесса по результатам этих вычислений*.

Оба эти выражения подчеркивают главное **требование** к СРВ – эти системы должны выполнять свои операции вовремя. Каким же образом разработчик может обеспечить выполнение этого требования? Ответом на этот вопрос является определение, данное Мартином Тиммерманом (Martin Timmerman):

Системы реального времени – это системы, которые предсказуемо (в смысле времени реакции) реагируют на не предсказуемые (по времени появления) внешние события [1].

Из этого определения вытекает главное **свойство** систем реального времени: *предсказуемость* или *детерминированность*[9]. Только благодаря этому свойству разработчик может гарантировать функциональность и корректность спроектированной системы. При этом собственно скорость реакции системы важна только относительно скорости протекания внешних процессов, за которыми СРВ должна следить, или которыми должна управлять.

Из приведенных определений следует, что СРВ призваны решать задачи, в которых важны не только правильность решения, но и сроки, в которые эти решения принимаются. В зарубежной литературе срок, в пределах которого должно быть принято решение называется *deadline*, что может быть переведено как *критический срок обслуживания*. Если невыполнение задачи в критический срок обслуживания означает, что она вообще не была выполнена, то такие задачи называют *задачами жесткого реального времени*. В большинстве случаев о задачах жесткого реального времени говорят тогда, когда нарушение сроков критического обслуживания может нанести значительный материальный или физический ущерб. К *задачам мягкого реального времени* относят случаи, когда нарушение критического времени обслуживания ведет к неприятным, но допустимым последствиям (например, требует дополнительной обработки).

СРВ в большинстве случаев решают комбинацию задач жесткого и мягкого реального времени, а также задач, не имеющих критического срока обслуживания. Иногда задача может переходить из статуса мягкого реального времени при пропуске некоторого срока обслуживания в статус задачи жесткого реального времени назначением критического срока обслуживания.

Если СРВ строится как программный комплекс, то, в общем виде, она может быть представлена как комбинация трех компонент (таблица 1): прикладное программное обеспечение, операционная система реального времени (ОСРВ) и аппаратное обеспечение. При разработке СРВ необходим тщательный анализ соответствия характеристик этих трех компонент требованиям внешнего объекта, для управления или

мониторинга которым эта СРВ предназначена. Как уже говорилось, проведение такого анализа требует, что бы временные характеристики всех компонент системы были хорошо прогнозируемыми.

В целом ряде задач автоматизации программные комплексы должны работать как составная часть более крупных автоматических систем без непосредственного участия человека. В таких случаях СРВ называют встраиваемыми. Вот одно из определений таких систем:

Встраиваемые системы (Embedded systems) можно определить как программное и аппаратное обеспечение, составляющее компоненты другой, большей системы и работающее без вмешательства человека [1].

Таблица 1. *Компоненты системы реального времени.*

Прикладное программное обеспечение	Потоки	Диспетчеризация	
		Меж–потоковое взаимодействие	
Операционная система реального времени	API	Обработка прерывания	
	Защита от инверсии приоритетов	Управление потоками	
	I/O	Управление памятью	
Аппаратное обеспечение	CPU	Кэш	Устройства

Аппаратную часть СРВ, на которой исполняется ОСРВ и прикладное программное обеспечение, принято называть *целевой* (target) *платформой*. В святи с возможной специфичностью целевой платформы, особенно в случае встраиваемых систем, разработка прикладных программ может проводится на другой аппаратуре и даже, в некоторых случаях, на другой ОС, а отладка конечных программ производится либо удаленно с помощью специальных инструментальных средств, либо с помощью эмуляции работы целевой ОС.

Дальнейшие материалы данной работы посвящены в первую очередь второй компоненте системы реального времени, а именно операционным системам реального времени.

3. Обзор архитектур ОСРВ

За свою историю архитектура операционных систем претерпела значительное развитие. Один из первых принципов построения, т.н. *монолитные* ОС (рисунок 1), заключался в представлении ОС как набора модулей, взаимодействующих между собой различным образом внутри ядра системы и предоставляющих прикладным программам входные интерфейсы для обращений к аппаратуре. Главным недостатком такой архитектуры является плохая предсказуемость ее поведения, вызванная сложным взаимодействием модулей системы между собой.

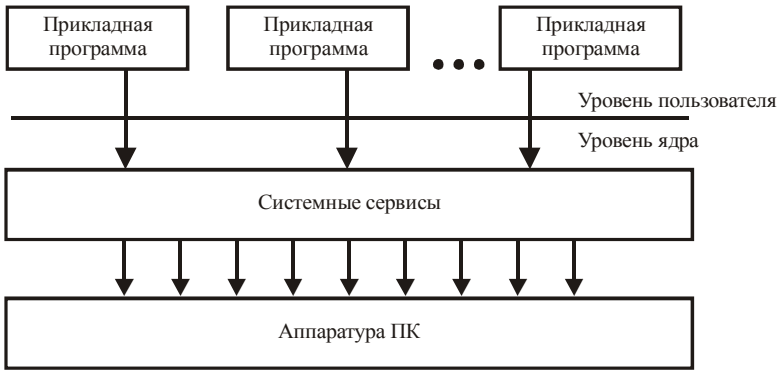

Рисунок 1. Архитектура монолитной ОС

Однако большинство современных ОС, как реального времени, так и общего назначения, строятся именно по этому принципу.

В задачах автоматизации широкое распространение в качестве ОСРВ получили *уровневые* ОС (рисунок 2). Примером такой ОС является хорошо известная система MS-DOS. В системах этого класса прикладные приложения могли получить доступ к аппаратуре не только посредством ядра системы или ее резидентных сервисов, но и непосредственно. По такому принципу строились ОСРВ в течение многих лет. По сравнению с монолитными ОС такая архитектура обеспечивает значительно большую степень предсказуемости реакций системы, а также позволяет осуществлять быстрый доступ прикладных приложений к аппаратуре. Недостатком

таких систем является отсутствие в них многозадачности. В рамках такой архитектуры проблема обработки асинхронных событий сводилась к буферизации сообщений, а затем последовательному опросу буферов и обработке. При этом соблюдение критических сроков обслуживания обеспечивалось высоким быстродействием вычислительного комплекса по сравнению со скоростью протекания внешних процессов.

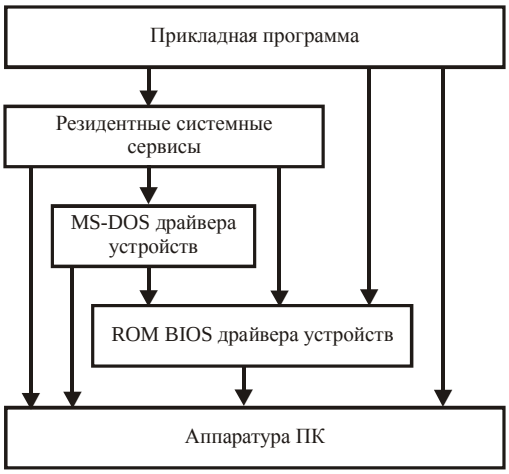

Рисунок 2. Архитектура уровневой ОС

Одной из наиболее эффективных архитектур для построения операционных систем реального времени считается архитектура клиент – сервер. Общая схема ОС работающей по этой технологии представлена на рисунке 3. Основным принципом такой архитектуры является вынесение сервисов ОС в виде серверов на уровень пользователя, а микроядро выполняет функции диспетчера сообщений между клиентскими пользовательскими программами и серверами – системными сервисами. Такая архитектура дает массу плюсов с точки зрения требований к ОСРВ и встраиваемым системам. Среди этих преимуществ можно отметить:

1. Повышается надежность ОС, т.к. каждый сервис является, по сути, самостоятельным приложением и его легче отладить и отследить ошибки.
2. Такая система лучше масштабируется, поскольку ненужные сервисы могут быть исключены из системы без ущерба к ее работоспособности.
3. Повышается отказоустойчивость системы, т.к. «зависший» сервис может быть перезапущен без перезагрузки системы.

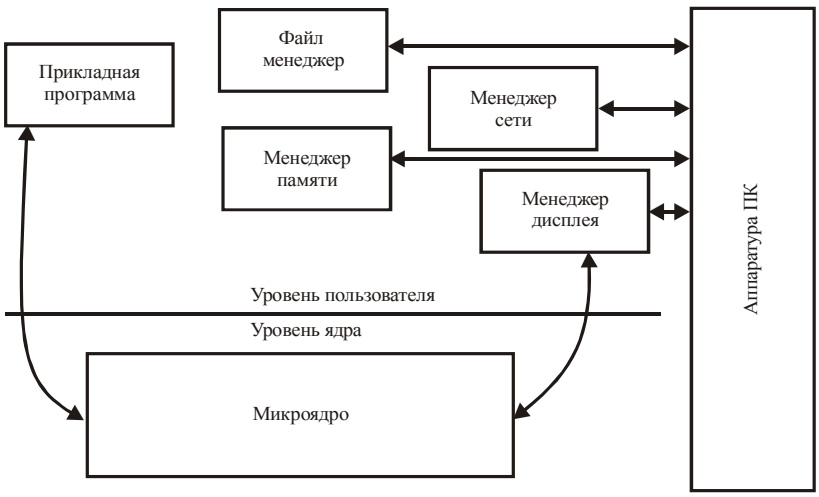

Рисунок 3. Построение ОС с использованием архитектуры клиент-сервер

К сожалению на сегодняшний день не так много ОС реализуется по принципу клиент-сервер. Среди известных ОСРВ реализующих архитектуру микроядра можно отметить OS9 и QNX.

4. Функциональные требования к ОСРВ

Расширение области применения СРВ привело к повышению требований к этим системам. В настоящее время обязательным условием, предъявляемым к ОС, претендующей на применение в задачах реального времени,

является реализация в ней механизмов многозадачности. Та же тенденция присутствует и в ОС общего назначения. Но для СРВ к реализации механизмов многозадачности предъявляется ряд дополнительных, по сравнению с системами общего назначения, требований. Определяются же эти требования тем обязательным свойством СРВ, о котором уже говорилось – *предсказуемостью*.

Многозадачность подразумевает параллельное выполнение нескольких действий, однако практическая реализация параллельной работы упирается в проблему совместного использования ресурсов вычислительной системы. И главным ресурсом, распределение которого между несколькими задачами называется *диспетчеризацией* (*scheduling*), является процессор. Поэтому в однопроцессорной системе по настоящему параллельное выполнение нескольких задач невозможно. Существует достаточно большое количество различных методов диспетчеризации, и основные среди них будут рассмотрены далее.

В многопроцессорных системах проблема разделения ресурсов также является актуальной, поскольку несколько процессоров вынуждены разделять между собой одну общую шину. Поэтому при построении СРВ нуждающейся в одновременном решении нескольких задач применяют группы вычислительных комплексов, объединенных общим управлением. Возможность работы с несколькими процессорами в пределах одного вычислительного комплекса и максимально прозрачное взаимодействие между несколькими вычислительными комплексами в пределах, скажем локальной сети, является важной чертой ОСРВ, значительно расширяющей возможности ее применения.

Под понятием *задачи* в терминах ОС и программных комплексов могут пониматься две разные вещи: *процессы* и *потоки*. *Процесс* является более крупномасштабным представлением задачи, поскольку обозначает независимый модуль программы или весь исполняемый файл целиком с его адресным пространством, состоянием регистров процессора, счетчиком команд, кодом процедур и функций. *Поток* же является составной частью процесса и обозначает последовательность исполняемого кода. Каждый процесс содержит как минимум один поток, при этом максимальное количество потоков в пределах одного процесса в большинстве ОС ограниченно только объемом оперативной памяти вычислительного комплекса. Потоки, принадлежащие одному процессу, разделяют его адресное пространство, поэтому они могут легко обмениваться данными, также время переключения между такими потоками (то есть время, за которое процессор переход от выполнения команд одного потока к выполнению команд другого) оказывается значительно меньшим, чем время переключение между процессами. В связи с этим в задачах реального времени параллельно выполняемые задачи стараются максимально компоновать в виде потоков, исполняющихся в пределах одного процесса.

Каждый поток имеет важное свойство, на основании которого ОС принимает решение о том, когда предоставить ему время процессора. Это свойство называется *приоритетом потока* и выражается целочисленным значением. Количество приоритетов (или уровней приоритетов) определяется функциональными возможностями ОС, при этом самое низкое значение (0) закрепляется за потоком idle ОС, который предназначен для корректной работы системы, когда ей «ничего не надо делать».

Поток может находиться в одном из следующих *состояний:*

1. Активный поток – это тот поток, который в данный момент выполняется системой.
2. Поток в состоянии готовности – поток, который может выполняться и ждет своей очереди.
3. Блокированный поток – поток, который не может выполняться по некоторым причинам (например, ожидание события или освобождения нужного ресурса).

Далее рассматриваются функциональные требования, предъявляемые на данном этапе к ОС применяющимся в СРВ.

4.1 Диспетчеризация потоков

Методы диспетчеризации, т.е. предоставления разным потокам доступа к процессору, в общем случае могут быть разделены на две группы. К первой относятся случаи, когда все потоки, которые разделяют процессор, имеют одинаковый приоритет, т.е. их важность с точки зрения системы одинакова:

1. *FIFO* (*First In First Out*) – *Первый Вошел Первый Вышел*. Первой выполняется задача, первой вошедшая в очередь, при этом она выполняется до тех пор, пока не закончит свою работу или не будет заблокирована в ожидании освобождения некоторого ресурса или события. После этого управление передается следующей в очереди задаче.
2. *Карусельная многозадачностью* (*round robin*). При этом методе диспетчеризации в системе определяется специализированная константа, определяющая продолжительность непрерывного выполнения потока, т.н. квант времени выполнения (*time slice*). Таким образом, выполнение потока может быть прервано либо окончанием его работы, либо блокированием в ожидании ресурса или события, либо завершением кванта времени (того самого time slice). После этого управление передается следующему в очередности потоку. После окончания времени последнего потока управление передается первому потоку, находящемуся в

состоянии готовности. Таким образом, выполнение каждого потока разбито на последовательность временных циклов.

Появление второй группы методов диспетчеризации связано с необходимостью распределения времени процессора между потоками, имеющими разную важность, т.е. разный приоритет. В таких случаях для потоков с равным приоритетом используется один из указанных выше методов диспетчеризации, а передача управления между разно приоритетными потоками осуществляется одним из следующих методов:

3. В наиболее простом случае если в состояние готовности переходят два потока с разными приоритетами, то процессорное время передается тому, у которого более высокий приоритет. Такой метод называется *приоритетной многозадачностью*, но его использование в таком виде связано с рядом сложностей. При наличии в системе одной группы потоков с одним приоритетом и другой группы с другим, более низким приоритетом, то при карусельной диспетчеризации каждой группы в системе с приоритетной многозадачностью потоки низкоприоритетной группы могут вообще не получить доступа к процессору.
4. Одним из решений проблем приоритетной многозадачности стала т.н. *адаптивная многозадачность*, широко применяющаяся в интерфейсных системах. Суть этого метода заключается в том, что приоритет потока, не выполняющегося какой–то период времени повышается на единицу. Восстановление исходного приоритета происходит после выполнения потока в течении одного кванта времени или при блокировке потока. Таким образом, при карусельной многозадачности, очередь (или «карусель») более приоритетных потоков не может полностью заблокировать выполнение очереди менее приоритетных потоков.
5. В задачах реального времени предъявляются специфические требования к методам диспетчеризации, поскольку передача управления потоку должно определяться критическим сроком его обслуживания (т.н. deadline-driven scheduling). В наибольшей степени этому требованию соответствует *вытесняющая приоритетная многозадачность*. Суть этого метода заключается в том, что как только поток с более высоким, чем у активного потока, приоритетом переходит в состояние готовности, активный поток *вытесняется* (т.е. из активного состояния принудительно переходит в состояние готовности) и управление передается более приоритетному потоку.

На практике широко применяются как комбинации описанных методов, так и различные их модификации. В СРВ, в контексте задачи диспетчеризации нескольких разноприоритеных потоков, очень важной является проблема распределения приоритетов таким образом, чтобы каждый поток уложился в свой срок критического обслуживания. Если все потоки системы укладываются в свои сроки критического обслуживания, то говорят, что система *диспетчируема* (*schedulable*).

Для СРВ, применяющихся в обработке периодических событий, в 1970 году Лиу и Лейленд [4] предложили математический аппарат, позволяющий определить, является ли система диспетчируемой. Этот аппарат называется «Частотно монотонный анализ» (ЧМА) (Rate Monotonic Analyzing) [3]. Эффективность этого математического аппарата привела к тому, что ЧМА был принят в качестве стандарта такими организациями как USA Department of Defense, Boeing, General Dynamics, Magnavox, Mitre, NASA, Panamax, и др. [7, 12]. Также среди организаций, установивших ЧМА в качестве стандартного средства анализа и разработки систем жесткого реального времени можно отметить IBM Federal Sector Division, US Navy и European Space Agency [8].

Подобная позиция ведущих производителей привела к тому, что разработчикам ОСРВ пришлось учитывать требования по применению ЧМА при разработке своих систем. Возможность применения ЧМА ограничена рядом условий, первым из которых является диспетчеризация потоков методом вытесняющей приоритетной многозадачности.

На основании всего выше сказанного можно сформировать первое требование к ОСРВ: *ОСРВ должна реализовывать возможность многозадачности, причем с поддержкой вытесняющей приоритетной методики диспетчеризации*.

4.2 Уровни приоритетов

Как уже говорилось, для организации параллельного выполнения нескольких потоков зачастую необходимо разделение этих потоков по степени важности (или критическому сроку обслуживания). Среди совокупности параллельно выполняющихся задач выделяются потоки жесткого реального времени, потоки мягкого реального времени и потоки, не критичные ко времени обслуживания. Каждая из указанных групп должна иметь свой уровень приоритетов, к тому же потоки жесткого реального времени, в ряде случаев, должны иметь индивидуальные значения приоритетов. Практический опыт разработки систем реального времени говорит, что увеличение количества разно приоритетных потоков приводит к непрозрачности и непредсказуемости системы. Однако в ряде случаев это не так.

Как уже говорилось, на сегодняшний день существует ряд инструментов математического анализа, позволяющих распределить приоритеты между несколькими потоками так, что бы они гарантировано выполняли свои критические сроки обслуживания. Если же для данного набора потоков это невозможно, то результаты математического анализа покажут, какие именно потоки имеют критическое отношение срока обслуживания ко времени выполнения.

Упомянутый ранее аппарат ЧМА позволяет провести такое исследование для случая периодических критических времен обслуживания [5]. Однако для его применения анализируемые потоки должны иметь уникальные значения приоритетов, определяемые периодом каждого потока. В связи с этим требованием разработчики ОСРВ закладывают в своих системах достаточно большое количество приоритетов. Для примера в QNX 6.x их 64, а в Windows CE и VxWorks – 256.

Таким образом, можно сформировать второе функциональное требование ОСРВ: *ОС должна иметь достаточно большое (определяется масштабом задачи) количество приоритетов*. Рекомендуемым значением является 128 уровней. Естественно, что в прикладных задачах необходимо крайне осторожно использовать потоки с разными приоритетами и, по возможности стремиться к минимизации их количества. Большое количество разно приоритетных потоков может привести не только к потере предсказуемости системы, но и к проблемам синхронизации на доступе к разделяемым ресурсам.

4.3 Механизмы синхронизации

Помимо процессорного времени разные потоки могут иметь и другие ресурсы, которые им приходится разделять между собой. Это могут быть переменные в памяти, буферы устройств и т.д. Для защиты от искажения, вызванного одновременным редактированием одних и тех же данных разными потоками, используются специфические переменные, называемые объектами синхронизации. К таким объектам относятся мютексы, семафоры, события и т.д..

Третьим функциональным требованием к ОСРВ является *наличие в ОС механизмов синхронизации доступа к разделяемым ресурсам*. В принципе механизмы синхронизации присутствуют в любых многозадачных системах, поскольку без них нельзя обеспечить корректную работу нескольких потоков с одним ресурсом (например, буфером устройства или некоторой общей переменной). Однако в задачах реального времени к объектам синхронизации предъявляются специфические требования. Связано это с тем, что именно на объектах синхронизации возможны значительные задержки выполнения потоков, поскольку назначением этих объектов является фактически блокирование доступа к некоторому разделяемому ресурсу. Одной из наиболее серьезных проблем, возможных при блокировании ресурса является *инверсия приоритетов*.

4.4 Защита от инверсии приоритетов

Итак, проблема инверсии приоритетов оказалась настолько важной для ОСРВ, что реализацию в системе механизмов защиты от этой проблемы вынесли в отдельное функциональное требование к операционным системам реального времени [1]. Давайте разберемся, что это такое? Инверсия приоритетов возникает, когда два потока, высоко приоритетный (*B*) и низкоприоритетный (*H*) разделяют некий общий ресурс (*P*). Предположим, также что в системе присутствует третий поток, приоритет которого находится между приоритетами *B* и *H*. Назовем его средним (*C*). Если поток B переходит в состояние готовности когда активен поток *H* и *H* заблокировал ресурс *P*, то поток *B* вытеснит поток *H* и *P* останется заблокирован. Когда *B* понадобится ресурс *P*, то он сам перейдет в заблокированное состояние. Если в состоянии готовности находится только поток *H*, то ничего страшного не произойдет, *H* освободит заблокированный ресурс и будет вытеснен потоком *B*. Но если на момент блокирования потока *B*, в состоянии готовности находится поток *C*, приоритет которого выше чем у *H*, то активным станет именно он, а *H* опять будет вытеснен, и получит управление только после того, как *C* закончит свою работу. Подобная задержка вполне может привести к тому, что критическое время обслуживания потока *B* будет пропущено. Если *B* это поток жесткого реального времени, то подобная ситуация недопустима.

Какие же механизмы защиты от этой проблемы используют разработчики операционных систем реального времени? Наиболее широко распространенный и проверенный механизм – это *наследование приоритетов*.

Суть этого метода заключается в наследовании низкоприоритетным потоком, захватившим ресурс, приоритета от высокоприоритетного потока, которому этот ресурс нужен. В описанном примере это означает следующее. Если *H* блокировал ресурс *P*, который нужен *B*, то при блокировании *B* его приоритет присваивается потоку *H*, и, таким образом, он не может быть вытеснен потоком, с меньшим чем у *B* приоритетом. После того, как поток *H* разблокирует ресурс *P*, его приоритет понижается до исходного значения и он вытесняется потоком *B*.

Механизм наследования приоритетов, к сожалению, не всегда может решить проблемы, связанные с блокирование высокоприоритетного потока на заблокированном ресурсе. В случае, когда несколько средне– и низко–приоритетных потоков разделяют некоторые ресурсы с высокоприоритетным потоком возможна ситуация, когда высокоприоритетному потоку придется слишком долго ждать пока каждый из младших потоков не освободит свой ресурс и критический срок обслуживания будет потерян. Однако такие ситуации (разделения ресурсов высокоприоритетного потока) должны отслеживаться разработчиками прикладной системы. В принципе наследование приоритетов является наиболее распространенным механизмом защиты от проблемы инверсии приоритетов.

Другой, несколько менее распространенный метод, называется Протокол Предельного Приоритета (Priority Ceiling Protocol) [6]. Метод этот заключается в добавлении к стандартным свойствам объектов синхронизации параметра, определяемого максимальным приоритетом потока, которые к этому объекту обращаются. Если

этот параметр установлен, то приоритет любого потока, обращающегося к этому объекту синхронизации, будет увеличен до указанного уровня, и, таким образом, не сможет быть вытеснен никаким потоком, который может нуждаться в заблокированном им ресурсе. После разблокирования ресурса, приоритет потока понижается до начального уровня. Таким образом, получается нечто вроде предварительного наследования приоритетов. Однако этот метод имеет ряд серьезных недостатков. В первую очередь, на разработчика ложиться работа по «обучению» объектов синхронизации их уровню приоритетов. Во вторых, возможны задержки в запуске высокоприоритетных потоков на время отработки низкоприоритетных потоков. В целом максимально эффективно этот механизм может быть использован в случае, когда имеется один поток жесткого реального времени и несколько менее приоритетных потоков, разделяющих с ним ресурсы.

4.5 Временные характеристики ОС

Общепринято отделяет ОСРВ от операционных систем общего назначения следующее условие: *Время реакции операционной системы при любых вариантах загрузки должно оставаться постоянным*. На практике это означает высокую стабильность таких характеристик системы как латенция прерываний (т.е. время от момента инициации прерывания до первой команды программного обработчика), время переключения контекстов процессов и потоков, и т.д. Также для ОСРВ очень важны времена разрешения конфликтов, таких как приход низкоприоритетного и высокоприоритетного прерываний подряд в указанном порядке с небольшим временным разрывом. Стабильно малое время, за которое управление будет передано обработчику высокоприоритетного прерывания, является хорошей характеристикой ОСРВ [10]. Однако тут необходимо отметить такой важный факт, что само по себе время реакции системы не играет особой роли, временные характеристики должны рассматриваться в контексте параметров внешнего процесса. Необходимо помнить, что в системах реального времени ключевыми являются не статистические (средние) оценки, а **максимальные** значения, поскольку превышение времени реакции даже в одном случае из миллиона в задачах жесткого реального времени может привести к катастрофическим последствиям.

Еще одна важная особенность операционных систем реального времени, отделяющая их от систем общего назначения заключается в независимости поведения системы и ее времен реакций от количества текущих задач. В большинстве систем общего назначения такие параметры как время переключения контекста потока прямо зависит от количества потоков в системе, в системах же реального времени такой зависимости быть не должно.

5. Современные ОСРВ

5.1 VxWorks AE 1.1

Операционная система VxWorks построена по принципам монолитной операционной системы. Она реализует достаточно богатый набор функций API и поддерживает приоритетную вытесняющую многозадачность в комбинации с карусельной многозадачностью. Система VxWorks имеет мощные средства разработки и отладки приложений и в течении многих лет считается одним из лидеров среди ОСРВ.

Новшеством, появившимся в версии AE, стали *защищенные домены* (*protected domains*) представляющие собой некие контейнеры, со своим адресным пространством и, в зависимости от настройки, видимые или не видимые друг для друга. Появление защищенных доменов позволило осуществить более высокую защищенность данных и кода прикладных приложений по сравнению с предыдущей версией системы (5.x) в которой было единое адресное пространство для системы и прикладных задач.

Другой положительной чертой защищенных доменов, по сравнению с классическими процессами, является возможность установки диапазона приоритетов, которые будут наследоваться потоками этого домена. Таким образом, компоненты системы, не требующие реального времени, могут быт легко перенесены в область приоритетов, где они не смогут вызвать конфликтов с потоками реального времени.

5.2 Windows CE.NET

Windows CE достаточно недавно начала завоевывать рынок ОСРВ и делает это с определенными успехами. Архитектура этой системы также соответствует монолитной модели архитектуры ОС, однако для повышения масштабируемости часть сервисов системы оформлены как отдельные модули, взаимодействующие с ядром по технологии COM. Подобный подход позволил получить минимальный объем полнофункциональной системы порядка 200 Кб. Система поддерживает вытесняющую приоритетную многозадачность в комбинации с карусельной и FIFO многозадачностями. В управлении памятью система Windows CE реализует виртуальную модель, когда каждый процесс имеет индивидуальное адресное пространство, что обеспечивает высокую степень защищенности данных и кода.

Поскольку ОС Windows CE является Win32 совместимой, разработка СРВ на базе этой ОС проводится с использованием богатого набора инструментальных средств. Также компания Microsoft предоставляет специализированные средства разработки приложений для Windows CE.

5.3 QNX 6.21

Операционная система QNX канадской компании QSSL имеет более чем 20 летнюю историю. Эта система строится на базе микроядра с организованными по технологии клиент – сервер сервисами, вынесенными на уровень пользовательских приложений. Микроядро системы выступает в качестве диспетчера сообщений, переадресовывая системные вызовы прикладных программ клиентов к соответствующим сервисам серверам и обратно. Как уже говорилось, такое построение является одним из наиболее оптимальных решений в ОСРВ и обеспечивает высокую надежность и масштабируемость системы.

В системе QNX только микроядро исполняется на уровне привилегий 0 процессора Intel, системные сервисы (менеджеры) запускаются на уровне привилегий 1, драйвера устройств – 2 и пользовательские приложения на 3 уровне привилегий [11]. Подобное разделение приводит к более высокой надежности и отказоустойчивости системы, т.к. при «зависании» отдельных драйверов или сервисов, они могут быть перезапущены без перезагрузки системы. Также в ОС QNX реализована модель виртуальной памяти для каждого процесса, что обеспечивает высокую степень защищенности данных и кода прикладных приложений и системы.

Однако, за высокую надежность, обеспечиваемую разделением уровней приоритетов и индивидуальным адресным пространством процессов, приходится платить более длительным временем переключения контекстов прикладных программ, ядра и системных сервисов.

В системе QNX реализовано управление памятью на основе виртуального адресного пространства, что обеспечивает защиту данных и кода приложений, ядра и системных сервисов.

Важной особенностью ОСРВ QNX является очень высокая степень POSIX совместимости, что обеспечивает легкую переносимость приложений написанных в среде POSIX совместимых систем. Это приводит к тому, что в разработчик в QNX может воспользоваться всем богатством инструментальных средств, предоставляемых POSIX сообществом.

6. Заключение

В заключении необходимо еще раз отметить ключевые элементы, определяющие отличие операционных систем общего назначения от операционных систем реального времени.

Важнейшим свойством систем реального времени является предсказуемость временных реакций системы на внешние события. Только исходя из этого свойства можно говорить о состоятельности и обоснованности решений, заложенных в конкретной СРВ. И именно в свете временной предсказуемости необходимо рассматривать возможности выбора конкретной операционной системы под конкретную задачу реального времени.

Также необходимо отметить, что при анализе систем реального времени важным является выбор модели диспетчеризации потоков в рамках конкретной задачи. Определение метода распределения приоритетов, обоснование диспетчиризируемости всех потоков жесткого реального времени, все это является важнейшими действиями при проектировании систем реального времени. Дополнительные сложности создает отсутствие строгих математических методов в оценке диспетчирезируемости непереодических, динамических потоков.

В связи с этими требованиями выбор операционной системы для реализации конкретной системы жесткого реального времени является ответственным шагом, могущим определить успех или не успех разработки системы в целом.

Литература

1. Martin Timmerman, Bart Van Beneden, Lourent Uhres. RTOS Evaluation Kick Off! // Real-Time Magazine. – 1998. – N3 pp 6 – 10
2. Сергей Сорокин. Системы Реального Времени. // СТА. – 1997. - №2. – С 22-29.
3. Эрик Верхалст. Задача разработки ОСРВ для цифровой обработки сигналов// Мир компьютерной автоматизации. – 1997. – №4.
4. C. L. Liu, James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment //Journal of the Association for Computing Machinery, Vol. 20, No. I, January 1973. - P 46-61.
5. J.Zalewski. What Every Engineer Needs To Know About Rate-Monotonic Scheduling: A Tutorial// Real-Time Magazine. – 1995. – N 1. – P 6-24.
6. N.J. Keeling. Missed it! - How Priority Inversion messes up real-time performance and how the Priority Ceiling Protocol puts it right// Real-Time Magazine. – 1999. – N4. – P. 46-50.
7. Obenza, R., "Rate Monotonic analysis for real-time systems". *IEEE Computer,* 26, 3, 1993, pp. 73-74.

8. Lui Sha, Mark H. Klein, John B. Goodenough. Rate Monotonic Analysis for Real-Time Systems// Technical Report CMU/SEI-91-TR-6, March 1991, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213
9. А. А. Блискавицкий, С. В. Кабаев. Операционные системы реального времени (обзор)// Средства и системы компьютерной автоматизации. http://www.asutp.ru.
10. Жданов А.А. Операционные системы реального времени// "PCWeek", N 8, 1999.
11. Dedicated Systems Experts, QNX® RTOS 6.1// Dedicated Systems (www.dedicated-systems.com) . – 2002. – 97 p.
12. R. Cayssials, J. Santos, J. Orozco and E. Ferro. Fixed Priority scheduling in hard real-time multiprogramming environments: Liu & Layland revisited // Universidad Nacional del Sur/CONICET, 8000 Bahia Blanca, Argentina. – 4 p.