Архитектура ОС QNX. Особенности программирования. Процессы и потоки

Лекция

Ревизия: 0.1

Последняя модификация: Vlad Kovtun

© NRJETIX 2000 - 2010

Дата последней модификации: 24.11.2010 15:47:00

История изменений

14.11.2010 – Версия 0.1.   Первичный документ. Ковтун В.Ю.

Содержание

История изменений 2
Содержание 3
Лекция 15. Архитектура ОС QNX. Особенности программирования. Процессы и потоки 4
  Вопросы 4
  Введение 4
    Основные понятия 4
  Обзор архитектур ОСРВ 5
    Функциональные требования к ОСРВ 7
    Диспетчеризация потоков 8
    Уровни приоритетов 9
    Механизмы синхронизации 10
    Защита от инверсии приоритетов 10
    Временные характеристики ОС 11
    Современные ОСРВ 11
  Заключение 12
Инверсия приоритетов и реальное время 13
  Как это выглядит ... in nature...? 13
Служба времени 15
  Модель временной шкалы QNX 15
  Измерение временных характеристик 16
Литература 17

Лекция 15. Архитектура ОС QNX. Особенности программирования. Процессы и потоки

Вопросы

1. Введение.
2. Обзор архитектур.
3. Инверсия приоритетов.
4. Служба времени.

Введение

Основные понятия

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 курсе лекций, о реальном времени будем говорить в контексте так называемых «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 событии.**

2. **Реальное время** (программное обеспечение) (IEEE 610Л2 — 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?):

Системы реального времени — это системы, которые предсказуемо (в смысле времени реакции) реагируют на непредсказуемые (по времени появления) внешние события.

Из этого определения вытекает **главное свойство систем реального времени**: **предсказуемость или детерминированность**. 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 **критический срок обслуживания** (deadline). 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 компонентов (таблица 1):

- прикладное программное обеспечение,
- операционная система реального времени (ОСРВ),
- аппаратное обеспечение.

При разработке СРВ необходим тщательный анализ соответствия характеристик этих трех компонентов требованиям внешнего объекта, для управления или мониторинга которым эта СРВ предназначена. Как уже говорилось, проведение такого анализа требует, чтобы временные характеристики всех компонентов 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**.

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

Таблица 1. Компоненты СРВ

<table>
<tr><td rowspan="2">Прикладное ПО</td><td rowspan="2">Потоки</td><td colspan="2">Диспетчеризация</td></tr>
<tr><td colspan="2">Межпотоковое взаимодействие</td></tr>
<tr><td rowspan="3">ОС реального времени</td><td>API</td><td colspan="2">Обработка прерывания</td></tr>
<tr><td>Защита от инверсии приоритетов</td><td colspan="2">Управление потоками</td></tr>
<tr><td>I/O</td><td colspan="2">Управление памятью</td></tr>
<tr><td>Аппаратное обеспечение</td><td>CPU</td><td>Кэш</td><td>Устройства</td></tr>
</table>

Аппаратную часть СРВ, на которой исполняется ОСРВ и прикладное программное обеспечение, принято называть **целевой (target) платформой**. В связи с возможной специфичностью целевой платформы, особенно в случае встраиваемых сметем, разработка прикладных программ может проводиться 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.

Дальнейшее изложение посвящено, главным образом, второй компоненте системы реального времени, а именно - ОС реального времени.

Обзор архитектур ОСРВ

За свою историю архитектура ОС претерпела значительное развитие. Один из первых принципов построения, так называемых **монолитных ОС**, рис. 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.

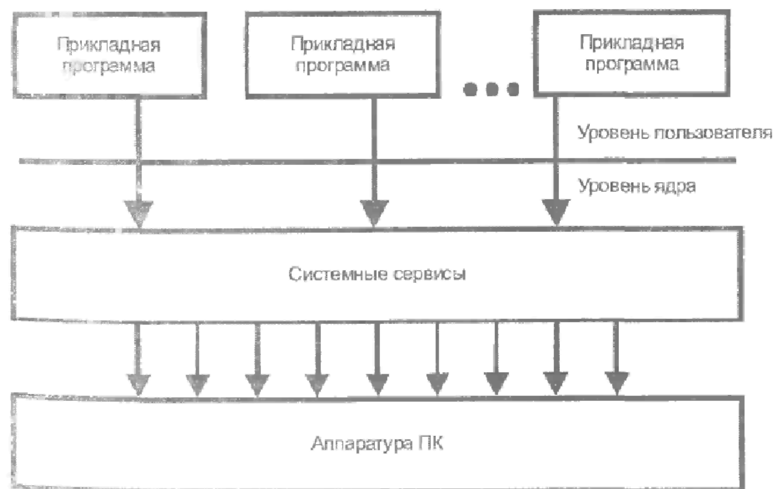

Рис. 1. Архитектура монолитной ОС

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**, рис. 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 этих 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.

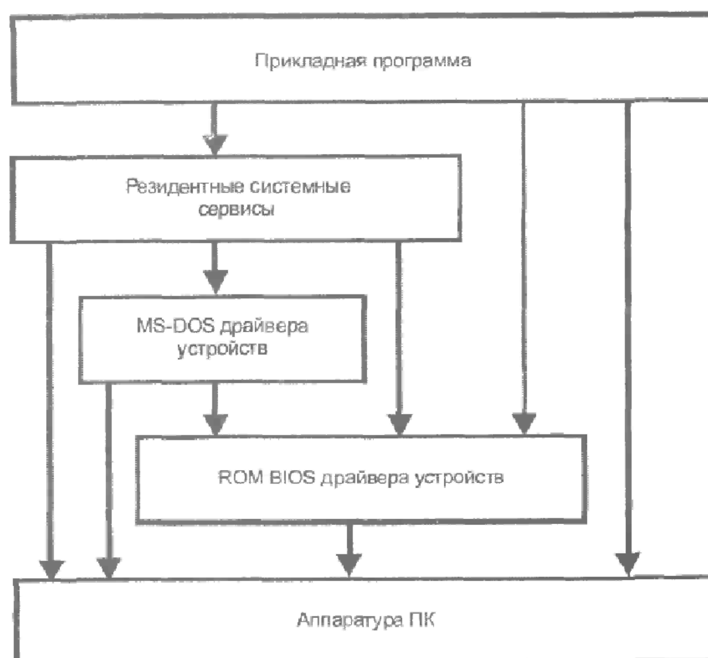

Рис. 2. Архитектура уровневой ОС

Одной из наиболее эффективных архитектур для построения ОС реального времени считается архитектура **клиент-сервер**. Общая схема ОС работающей по этой технологии представлена на рис. 3.

Рис. 3. Построение ОС с использованием архитектуры клиент-сервер

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. Данная архитектура дает массу плюсов с точки зрения требований к ОСРВ и встраиваемым системам. Среди этих преимуществ можно отметить следующие:

1. **Повышается надежность ОС**, т.к. каждый сервис является, по сути, самостоятельным приложением, и его легче отладить и отследить ошибки.

2. **Система лучше масштабируется**, поскольку ненужные сервисы могут быть исключены из системы без ущерба для ее работоспособности.

3. **Повышается отказоустойчивость системы**, т.к. «зависший» сервис может быть перезапущен без перезагрузки системы.

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

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 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 CPU. Поэтому в single CPU системе по-настоящему параллельное выполнение нескольких задач невозможно. Существует достаточно большое количество различных методов диспетчеризации, и основные среди них будут рассмотрены далее.

В multi CPU системах проблема разделения ресурсов также является актуальной, поскольку несколько CPU 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 CPU 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 масштабным 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 ограничено только объемом оперативной памяти вычислительного комплекса. Потоки, принадлежащие одному процессу, разделяют его адресное пространство, поэтому они могут легко обмениваться данными, а время переключения между такими потоками (то есть время, за которое процессор переходит 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 CPU. 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.

Диспетчеризация потоков

Методы диспетчеризации, т.е. предоставления разным потоки доступа к CPU, в общем случае могут быть разделены на две группы. К первой относятся случаи, когда все потоки, которые разделяют CPU, 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 многозадачность (round robin)**. При этом методе диспетчеризации в системе задается специализированная константа, определяющая продолжительность непрерывного выполнения потока, так называемый **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. По окончании 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 CPU 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 потоками с разным приоритетом осуществляется одним из следующих методов:

1. В наиболее простом случае если в состояние готовности переходят два потока с разными приоритетами, то время CPU передается тому, у которого более высокий приоритет. Данный 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, при карусельной диспетчеризации каждой группы в системе с приоритетной многозадачностью потоки низкоприоритетной группы могут вообще не получить доступа к процессору.

2. Одним из решений проблем приоритетной многозадачности стала так называемая **адаптивной многозадачности** широко применяющаяся в интерфейсных системах. Суть метода заключается в том, что приоритет потока, не выполняющегося какой—то период времени, повышается на единицу. Восстановление исходного приоритета происходит после выполнения потока в течение 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.

3. В задачах реального времени предъявляются специфические требования к методам диспетчеризации, поскольку передача управления потоку должна 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 потоков с разным приоритетом, очень важной является проблема распределения приоритетов таким образом, чтобы каждый поток уложился в свой критический срок обслуживания. Если все потоки системы укладываются в свои критические сроки обслуживания, то говорят, что система **диспетчиризуема** (schedulable).

Для СРВ, применяющихся в обработке периодических событий, в 1970 году Лиу и Лейленд предложили математический аппарат, позволяющий определить, является ли система диспетчиризуемой. Этот аппарат называется **Частотно монотонный анализ** (ЧМА) (Rate Monotonic Analyzing). Эффективность данного 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. Среди организаций, установивших ЧМА в качестве стандартного средства анализа и разработки систем жесткого реального времени можно также отметить IBM Federal Sector Division, US Navy и European Space Agency.

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**.

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 СРВ говорит, что увеличение количества разно приоритетных потоков приводит к непрозрачности и непредсказуемости системы. Однако не всегда это выглядит именно так.

Как уже говорилось, на сегодняшний день существует ряд инструментов математического анализа, позволяющих распределить приоритеты между несколькими потоками таким образом, чтобы они гарантировано выполняли свои критические сроки обслуживания. Если же для данного набора потоков реализовать 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. 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 большой (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 потоков с разным приоритетом может привести не только к потере предсказуемости системы, но и к проблемам синхронизации на доступе к разделяемым ресурсам.

Механизмы синхронизации

Помимо времени CPU, 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**.

Защита от инверсии приоритетов

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

Какие же механизмы защиты от этой проблемы используют разработчики ОСРВ? 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 Н блокировал ресурс Р, который нужен В, то при блокировании В его приоритет присваивается потоку Н, и, таким образом, он не может быть вытеснен потоком, с меньшим чем у В приоритетом. После того, как поток Н разблокирует ресурс Р, его приоритет понижается до исходного значения и он вытесняется потоком В.

Механизм наследования приоритетов, к сожалению, не всегда может решить проблемы, связанные с блокированием 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). Метод заключается в добавлении к стандартным свойствам объектов синхронизации параметра, определяемого максимальным приоритетом потока, которые к объекту обращаются. Если такой параметр установлен, то приоритет любого потока, обращающегося к данному 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 ложится 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.

Временные характеристики ОС

Общепринятым условием, отделяющим ОСРВ от ОС общего назначения, является следующее: **Время реакции ОС 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. Однако здесь необходимо отметить важные момент: **само по себе время реакции системы не играет особой роли, временные характеристики должны рассматриваться в контексте параметров внешнего процесса**. Следует помнить, что в СРВ 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 ОСРВ**, 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 этой зависимости быть не должно.

Современные ОСРВ

VxWorks AE 1.1

ОС VxWorks построена по принципам монолитной ОС. 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 течение 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 невидимые 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 существовало 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 быть легко перенесены в область приоритетов, где они не смогут вызвать конфликтов с потоками реального времени.

Windows CE.NET

Windows CE не очень давно начала завоевывать рынок 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. При этом компания Microsoft предоставляет специализированные средства разработки приложений для Windows CE.

QNX 6.21

ОС 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 соответствующим сервисам серверам и обратно. Как уже говорилось, такое построение является одним из 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 — на 2-м уровне, а пользовательские приложения на 3-м уровне привилегий. Подобное разделение приводит к более высокой надежности и отказоустойчивости системы, т.к. при «зависании» отельных драйверов или сервисов, они могут быть перезапущены без перезагрузки системы. В ОС QNX реализована также **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 зашиту 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 разработчик в QNX может воспользоваться всем богатством инструментальных средств, предоставляемых POSIX сообществом.

Заключение

В заключении необходимо еще раз отметить ключевые элементы, определяющие отличие ОС общего назначения от ОСРВ.

Важнейшим свойством СРВ 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 СРВ важным является выбор модели диспетчеризации потоков в рамках конкретной задачи. Определение метода распределения приоритетов, обоснование диспетчеризируемости всех потоков жесткого реального времени, все это является важнейшими действиями при проектировании СРВ. Дополнительные сложности создает отсутствие строгих математических методов в оценке диспетчеризируемости непериодических, динамических потоков.

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

Инверсия приоритетов и реальное время

При оценке степени принадлежности той или иной программной системы (будь то прикладная система «в целом», или отдельно взятая ОС, в среде которой должно функционировав приложение) к категории realtime, возникает необходимость проверки системы на выполнение ряда критериев, известных как «требования realtime». Не будем затрагивать вопросы различных определений realtime и анализа требований, которым должна удовлетворять realtime система - это слишком неохватный предмет, имеющий столь же расширенный диапазон мнений и толкований. Остановимся только на 1-м частном требовании (иногда его называют **«последним в ряду»**), которое выдвигают к системам, в частности ОС для того, чтобы ее можно было отнести к realtime OS (RTOS) в противоположность OS общего назначения (general OS - GOS), а именно — инверсии приоритетов.

Инверсия приоритетов - это ситуация, при которой в результате взаимных синхронизаций, управление получает не ветвь исполнения, которая должна была бы получить из соображений приоритетности, а другая, с более низким приоритетом. Механизмы, порождающие это явление, могут быть разно; образны, но, независимо от деталей происходящего, результат один — управление получает не та ветвь, которая должна, а в результате нарушается детерминированность и прогнозируемость; системы.

Как это выглядит ... in nature...?

Предположим, что в системе присутствуют три потока управления: T1, T2, T3, причем их приоритеты: T1 < T2 < T3. Допустим, что потоки активируются (стартуют или «пробуждаются» по некоторым событиям) во временной последовательности так: T1 - T2 — T3. Если теперь T1 захватывает некоторый монопольный ресурс, который требуется так же и T3, и не успевает освободить его до активации T2, то происходит

- T1 не выполняется, потому что он вытеснен T2 (в приоритетности);
- T3 не выполняется, потому что ожидает ресурс, захваченный T1;
- T2 выполняется вплоть до бесконечно долго!

Но T2 в описываемой ситуации не является самым высокоприоритетным потоком, подлежащим немедленному исполнению.

Все приехали: если T3 - реакция на citical mission событие, оно не только не обработается в гарантированное время, но и вообще гарантировано не обработается.

Ситуация возникновения инверсии приоритетов не только экстремальная (аварийная) с точки зрения последствий для функционирования системы, она еще и крайне тяжела для диагностики, локализации и отладки. Дело в том, что факт возникновения или не возникновения инверсии зависит от взаимной временной расстановки точек активизации T1, T2, T3. Стоит T2 стартовать «чуть попозже», когда T1 уже освободит захваченный ресурс, и приложение благополучно проходит эту точку. Стоит T3 стартовать «чуть раньше», когда T1 еще не захватил ресурс, и тот же результат, но с совершенно другой расстановкой последовательности прохождения потоками критического участка!. В результате, при тестировании реального приложения, инверсия приоритетов может фиксироваться в 1-м случае из 1 миллиона тестовых прогонов, ... или из миллиарда. Это ситуация, которая способствует тому, что даже проявившись, случай с инверсией приоритетов будет списан на артефакты («грязная посуда») и даже не будет зафиксирован в протоколах тестирования... Со всеми вытекающими последствиями.

Иногда, такую ситуацию с потоками T1, T2, T3 относят к «неправильно спроектированной архитектуре приложения», но такой подход — это желание отмахнуться от проблемы, а не разрешить ее. Посмотрим на ту же ситуацию (то же приложение), но с потребительской точки зрения, как на изделие.

Ситуация: проектируется система управления каким-нибудь... паровым катком (с потенциально взрывоопасными наклонностями). При этом программная система строится из 2-х потоков/ процессов:

- T3 — **критическая реакция на перегрев парового котла**. Для повышения динамики этой задачи, она использует собственный allocator памяти для многих промежуточных структур данных в большой, предварительно резервированной буферной области. Естественно (время критично!), что размешать-то он размещает, а вот удаления и уборки памяти в буферной области он не делает — не до того. Выполняется этот поток на критически высоком приоритете 60 (используются численные значения из OS QNX, где приоритет потока может быть 1...63).
- T1 - программа **(поток, процесс, задача) сборки мусор в буферной области и удаление размешенных T3 структур данных – «сборка мусора»**. Этот поток может работать по принципу «когда нечем больше заняться» далеко на заднем плане, поэтому запускается этот поток с приоритетом 5. Естественно, на очень короткий интервал времен! Он обязан монопольно блокировать структуру буфера, во избежание ее разрушения вообще. Ничего особенного и крамольного. Нормальное приложение проектировщики добротно поработали... Даже если блокирование буфера T1 приходится точно на редко возникающее событие активации T3, то ничего критического не может произойти интервал времени, в течении которого T3 полностью «перекраивает» структуру буфера, на несколько порядков меньше критического времени срабатывания T3, T3 выжидает эту «досадную» блокировку и добросовестно выполняет свою работу. Все осмыслено и оттестировано миллион раз. Изделие передается на опытную эксплуатацию...

Начальник стенда опытной эксплуатации тоже «классный специалист» — он, попутно с выполнением своих обязанностей написал на BASIC программу расчета будущей неимоверной экономической эффективности предстоящего серийного выпуска изделия. И гоняет ее тоскливыми ночами на стенде. А у него там вычисляется константа PI с 135-ю значащими цифрами. Ну вот, считает он экономические эффекты «вкруговую», и твердо убежден, что излишняя точность никому еще не навредила. А какой спрос с человека, пишущего на BASIC. То, что он гоняем свои расчеты с консоли работающей установки его, естественно, нисколько не смущает. Во-первых, от разработчиков он слышал о приоритете 60 для критической секции; во-вторых, он проделывает это уже в 1000-ную ночь на протяжении 12-ти часов ежедневно... Естественно также, что его задача T2 запускается под приоритетом by default 10 (хотя бы потому, что про команду nice ему просто ничего не известно). Но в 1001-ю ночь ему не везет — запуск на выполнение T2 приходится на тот 1 мс интервал, когда T1 блокирует memory allocator для T3. T1 заблокирована вычислением PI, и возникает критический перегрев, задача T3 пытается активироваться... но не тут-то было. И на вычислении 72-го знака в записи константы наш паровой каток взлетает в воздух вместе с незадачливым расчетчиком!

Такая вот сказка с грустным концом, которая вполне может иметь место в реальной жизни... Какие способы предлагаются для борьбы с инверсией приоритетов? Первое, что, возможно, напрашивается, это найти **способ отобрать критический ресурс у T1, как только он понадобится** (просто по Шарикову: «а что тут спорить - взять и разделить...»). Но этого категорически нельзя делать ни в коем случае: если T1 блокирует ресурс, то он делает это для каких-то манипуляций с ним. Отобрав ресурс до завершения критической секции работы с ним у T1, мы почти гарантировано получим ресурс в некотором промежуточном, нестационарном и непригодном для дальнейшего использования состоянии. Разработчиками OS предлагается простое и относительно логичное решение: нужно **динамически повысить** приоритет потока T1 до значения максимального приоритета всех тех потоков, которые ожидают освобождения заблокированного ресурса. А после его освобождения тут же динамически вернуть приоритет T1 к его исходному статическому уровню. При этом «задерживающий всех» поток T1 быстро проскочит критическую секцию на максимальном приоритете тех, кого он «задерживает». Такое поведение получило наименование **наследование приоритетов.**

Но это гораздо проще сформулировать, чем сделать! А именно:

- объекты синхронизации, как правило, — это объекты ядра OS. И для того, чтобы в OS осуществить наследование приоритетов на объектах синхронизации (если это не было изначально заложено в архитектуру OS), нужно подвергнуть ревизии и перекроить весь код ядра OS;
- ревизии подлежит реализация наследования приоритетов для каждого вида синхронизирующих примитивов IPC отдельно, а это многократно умножает трудоемкость предстоящих изменений;
- результат такой титанической перекройки заметят и оценят только крайне малая часть пользователей OS — те, кому нужны ее realtime характеристики;
- зато интегральные рыночные характеристики OS после такой перекройки, скорее всего, упадут — все операции синхронизации будут более «тяжеловесными», OS станет субъективно более медленной и менее предпочтительной для подавляющего большинства потребителей на рынке подобных систем. Есть над чем задуматься...

Служба времени

Для более или менее отчетливого понимания функционирования служб времени в QNX требуется внимательно рассмотреть:

- Задание любых временных интервалов с использованием функций sleep(), usleep(), nanosleep(), delay(), alarm(), select(), nanospin(), sigaction();
- Все механизмы и функции, связанные с созданием и использованием интервальных таймеров: все функции группы timer_*();
- Измерение временных характеристик выполнения (хронометрирование) критических участков программного кода — clock(), time(), times() и др.

Модель временной шкалы QNX

Модель и поведение службы времени QNX наиболее полно, изложена Р.Кертеном, но степень детализации ее определенно недостаточна. так, что мы имеем "в предпосылках":

- Микроядро QNX "живет" в дискретной "сетке" времени; Каждый единичный момент времени для микроядра — это такт или "тик" системного времени (timeslice);
- Какие-либо изменения состояний времени фиксируются микроядром только в узлах этой дискретной шкалы времени;

Микроядро «не различает» (не разрешает) два временных события, если они происходят между соседними «тиками» системного времени с интервалом, естественно, меньше чем интервал системного тика.

Собственно, все достаточно естественно и разумно: прикладывая линейку для измерения длины, мы имеем «дискретную сетку» с разрешением в 1 мм, и в утверждениях типа «длина составляет 13.55 мм» уточнение «.55» — это «от лукавого», и мы должны говорить либо о 13-ти, либо о 14-ти мм.

Начальная (после загрузки системы) длительность тика определяется соглашениями, принятыми для той или иной версии QNX. Из документации следует, что для QNX RTP 6.1 при частоте CPU >40МГц тик составляет 1 мс, и 10 мс - при меньших частотах CPU. Значение тика для QNX RTP программно может быть переустановлено вызовом ClockPeriod(), но не точнее 500 мкс. Для QNX Momentics 6.2.1 (последняя стабильная версия): значение системного тика по умолчанию - 1 мс, а минимальное значение системного тика может быть установлено в 10 мкс.

В архитектуре PC x86 часы реального времени работают от задающего кварцевого генератора, сигнал от которого далее поступает на делитель. Из-за различий в 5-6 знаке (коэффициент деления должен быть целым числом) реально период тика чуть меньше 1мсек для значения, установленного по умолчанию или чуть меньше переустанавливаемого значения. Точное значение времени тика можно получить так:

```
clockperiod clcold;

ClockPeriod(CLOCK_REALTIME, NULL, clcold, 0);

cout << "ClockPeriod=" << clcold.nsec << endl;
```

Измерение временных характеристик

Вот теперь, более или менее разобравшись с моделью временной шкалы QNX и методологией задания временных интервалов, мы можем перейти к вопросу об измерении временных характеристик в QNX. Из сказанного выше уже понятно, что:

- любые временные интервалы могут быть установлены и измерены только в единицах целых значений установленного в системе тика;
- любые численные значения, в которых фигурирует избыточное число значащих цифр или которые указывают значения, не являющиеся целым числом системных тиков, игнорируются системой.

Теперь сосредоточимся на рассмотрении того, «как» и «каким инструментом» можно измерять временные интервалы в системе:

Способ 1. О нем уже было достаточно сказано выше — используем аппаратный счетчик циклов CPU:

```
#include <inttypes.h>

finclude <sys/pages,h>

uint64_t cps = SYSPAGE_ENTRY (qtime)->cycles_per_sec;

uint64_t bg = ClockCycles();

//  ... делаем что-то

uint64_t fn = ClockCycles();

cout<< "Draft time in seconds - " << (float)(fn - bg)/cps << endl;
```

Этим способом мы можем замерять только «грязное» хронометрическое время без привязки к выполняемому процессу.

Способ 2. Использование time_t clock(void). Дословно из документации: "The clock() function returns the number of clock ticks of processor time used by the program since it started executing. You can convert number of tics in seconds by dividing by the value CLOCK_PER_SEC". Это чистое процессорное время, диспетчерезированное потоку.

```
clock_t bq, fn, bg;

bg = clock();

// ... делаем что-то

fn = clock();

cout << "Time in seconds = " << (long)((fn - bg) / CLOCK_PER_SEC) << endl;
```

Посмотрим значение константы `CLOCK_PER_SEC`, установленное в системе: `CLOCK_PER_SEC = 1000000.`

Способ 3. Получим сведения о времени выполнения процесса непосредственно из системной информации исполняемого процесса (аналогичным способом можно «достать» и временные интервалы, относящиеся к отдельным потокам процесса):

```
#include <sys/procfs.h>

#include  <fcntl.h>

// .....

pid_t nProc = getpid();      // pid текущего процесса

char sBuff  [100];

sprintf(sBuf, "/proc/%d/as",  nProc );

int fd = open(sBuf, O_RDONLY );

proc_info info; // читаем proc_info

If(devctl(fd, DCMD_PROC_INFO, &info, sizeof(info)) != EOK)

{

     //  ...  ERROR  ...

}

uint64_t put1, put2, pst1, pst2;

put1 = info.utime;
```

```
pst1 = info.stime;
// … делаем что-то
// снова читаем proc info
If(devctl(fd, DCMD_PROC_INFO, &info, sizeof(info)) != EOK )
{
     //  ...  ERROR  ...
}
put2 = info.utime;
pst2 = info.stime;
cout << "User time: " (put2 - put1) << " System time: " << (pst2 - pst1) << endl;
```

Литература

1. Э. Таненбаум. Современные операционные системы. 2-ое изд. –СПб.: Питер, 2002. –1040 с.

2. Э. Таненбаум, А. Вудхалл. Операционные системы: разработка и реализация. Классика CS. –СПб.: Питер, 2006. –576 с.

3. Д. Алексеев, Е. Ведричев, А. Волков и др. Практика работы с QNX. –М: Издательский дом «КомБук», 2004. -432 с.

4. Роб Кертен. Введение в QNX/Neutrino 2. Руководство по программированию приложений реального времени в QNX Realtime Platform. –СПб.: Издательство «Петрополис», 2001. -480 с.

5. QNX official web-site URL: http://qnx.com