Оглавление

Предисловие

Данное руководство по системной архитектуре посвящено операционной системе реального времени QNX Neutrino версии 6.3 и предназначено как для разработчиков приложений, так и для конечных пользователей.

В книге описана философия ОСРВ QNX Neutrino и основные архитектурные принципы, использованные для ее построения. Приводится описание структуры микроядра и служб обмена сообщениями. Также подробно описываются администратор процессов, администраторы ресурсов, графическая оболочка Photon microGUI и другие аспекты ОСРВ QNX Neutrino.

> Примечание
>
> Некоторые функции, описанные в данном руководстве, могут находиться на стадии разработки в момент выхода версии 6.3.0.
>
> Для получения последних новостей и текущей информации о любом продукте компании QNX Software Systems зайдите на веб-сайт компании по адресу www.qnx.com, где можно найти много полезных разделов, в том числе раздел для скачивания программных продуктов, раздел статей, написанных разработчиками, новостные группы, раздел технической поддержки и др.

Примечание для пользователей ОС Windows

В документации, посвященной программным продуктам QNX, левая косая черта, прямой слеш (символ "/"), используется в качестве разделителя во *всех* именах путей, включая те, которые относятся к Windows-файлам.

Кроме того, следует иметь в виду, что в большинстве случаев применяются правила, принятые для файловых систем POSIX/UNIX.

Глава 1

Философия QNX Neutrino

Целевые характеристики

Основным назначением операционной системы QNX Neutrino является реализация программного интерфейса POSIX в масштабируемой, отказоустойчивой форме, подходящей для широкого круга открытых систем, начиная от небольших встроенных систем с ограниченными ресурсами и заканчивая крупными распределенными вычислительными средами. Данная ОС поддерживает несколько семейств процессоров, в том числе x86, ARM, XScale, PowerPC, MIPS и SH-4.

Отказоустойчивая архитектура также крайне необходима для работы критически важных приложений (mission-critical applications), поэтому в ОС QNX Neutrino в полной мере используются возможности работы с MMU-оборудованием.

Конечно, описание целей само по себе не является гарантией результатов, поэтому мы предлагаем ознакомиться с данным руководством по системной архитектуре, чтобы получить представление о нашем подходе к реализации операционной системы и о тех компромиссах в разработке, к которым мы пришли, чтобы достичь поставленных целей. Как нам кажется, после прочтения данного руководства вы согласитесь с тем, что ОС QNX Neutrino является первым продуктом в своем роде, который действительно соответствует принципам открытых систем и обеспечивает высокий уровень отказоустойчивости и широкие возможности масштабируемости.

Встраиваемая операционная система стандарта POSIX?

Существует широко распространенный миф о том, что если "поскрести" операционную систему стандарта POSIX, то под ней обнаружится UNIX!

Отсюда делается вывод, будто POSIX-совместимая операционная система слишком громоздка и не подходит для встраиваемых систем.

Однако правда состоит в том, что POSIX это *не* UNIX. Хотя стандарты POSIX берут свое начало из практики использования UNIX, рабочие группы по POSIX четко определили эти стандарты как "интерфейс, а *не* реализацию".

Благодаря принятой в этих стандартах точной спецификации, а также существующим комплектам тестирования POSIX, нетрадиционные архитектуры операционных систем могут реализовывать программный интерфейс (API) POSIX без применения традиционного UNIX-ядра. Если сравнить любые две POSIX-системы, то может *показаться*, что они очень похожи — у них много одинаковых функций, утилит и т. д. Однако, когда речь идет о производительности и отказоустойчивости, они могут так же отличаться друг от друга, как день и ночь. Все дело в архитектуре.

Несмотря на то, что QNX Neutrino имеет архитектуру, которая абсолютно отличается от архитектуры UNIX, эта операционная система реализует программный интерфейс POSIX. Благодаря микроядерной архитектуре, компоновка программного интерфейса может быть легко масштабирована как "вниз", так и "вверх" по отдельным компонентам в соответствии с требованиями проектируемой встраиваемой системы реального времени.

Масштабирование операционной системы внутри семейства продуктов

Благодаря возможности легко масштабировать микроядерную ОС с помощью включения или выключения тех или иных процессов, обеспечивающих необходимую функциональность, вы можете использовать одну микроядерную ОС для значительно более широкого круга приложений, чем исполняемый модуль реального времени (realtime executive).

Разработка продуктов часто принимает форму создания целого "семейства продуктов", в котором каждая последующая модель несет все больший объем функциональности. Микроядерная архитектура избавляет от необходимости менять операционную систему с каждой новой версией продукта и дает возможность разработчикам легко масштабировать систему посредством добавления файловых систем, сетевых функций, графических пользовательских интерфейсов и других технологий.

Перечислим некоторые преимущества такого подхода:

- переносимость кода приложений (между версиями одного продукта);
- возможность использования одних и тех же инструментов для разработки всего семейства продуктов;

- "переносимость" опыта, полученного разработчиками;
- большая скорость разработки продукта.

Преимущества стандартов POSIX для встраиваемых систем

Одной из общих проблем в области разработки приложений реального времени является то, что большинство ОС реального времени поставляются со своим собственным программным интерфейсом (API). Такая ситуация нередко складывается на высококонкурентных рынках из-за отсутствия промышленных стандартов — а как показывают исследования, на рынке ОС реального времени существует множество операционных систем, изначально разработанных компаниями только для внутреннего использования. Таким образом, стандарты POSIX открывают путь для объединения этого рынка.

Среди множества стандартов POSIX наибольший интерес с точки зрения разработки встраиваемых систем представляют следующие.

- *1003.1* — данный стандарт определяет программный интерфейс для систем управления процессами, ввода/вывода данных в устройствах и файловых системах, а также общего межзадачного взаимодействия. Этот стандарт включает в себя все то, что может быть обозначено как базовая функциональность ОС UNIX, и является полезным для многих приложений. С точки зрения программирования на языке C, базовым является стандарт ANSI X3J11 C, тогда как различные аспекты управления процессами, файлами и TTY-устройствами определяются другими стандартами.
- *Расширения реального времени (Realtime Extensions)* — этот стандарт определяет набор дополнительных расширений реального времени к базовому стандарту 1003.1. Данные расширения включают в себя семафоры, планирование процессов по приоритетам, расширения реального времени к сигналам, управление таймерами высокого разрешения, дополнительные примитивы механизма межзадачного взаимодействия (IPC), систему синхронного и асинхронного ввода/вывода. Кроме того, в этом стандарте рекомендована поддержка непрерывных файлов в реальном времени.
- *Управление потоками (Threads)* — это еще одно расширение среды стандартов POSIX, оно регулирует создание и управление множеством потоков внутри заданного адресного пространства.
- *Дополнительные расширения реального времени (Additional Realtime Extensions)* — этот стандарт определяет дополнительные расширения

к стандарту реального времени, которые описывают такие функции, как подключение обработчиков прерываний.

- *Профили прикладных окружений (Application Environment Profiles)* — этот стандарт определяет несколько профилей среды POSIX (Realtime AEP, Embedded Systems AEP и др.), соответствующих разным наборам встраиваемых функций. Профили представляют встраиваемые ОС с/без файловых систем и других функций.

Замечание

Информацию о текущей документации по стандартам POSIX можно найти в отчете Комитета по стандартам переносимых приложений (Portable Applications Standards Committee) Компьютерного сообщества IEEE (IEEE Computer Society) по адресу **http://pasc.opengroup.org/standing/sdll.html**.

Стандарты POSIX важны не только как некие промышленные стандарты, необходимые для достижения разумного единообразия в индустрии встраиваемых приложений и операционных систем реального времени. Они также могут дать несколько других преимуществ.

Применимость во множестве операционных систем

Как известно, производители аппаратного оборудования неохотно используют электронные компоненты, производимые только одним изготовителем, из-за возможных рисков, связанных с неожиданным прекращением производства. По этой же причине производители не должны быть привязаны и к какой-либо ОС собственной разработки — хотя бы из-за того, что исходный код их приложения не переносится в другие приложения.

Приложения, разработанные в соответствии со стандартами POSIX, могут использоваться в различных ОС. Исходный код приложения может быть легко перенесен с одной платформы на другую и из одной ОС в другую, при условии что разработчики приложения не использовали расширения, ограниченные какой-то одной ОС.

"Переносимость" команды разработчиков

Благодаря использованию одного общего программного интерфейса для разработки встроенных систем, программисты, имеющие опыт работы с одной ОС реального времени, могут без всяких затруднений применять его и в других проектах с использованием других процессоров и операционных систем. Кроме того, программисты с опытом работы с UNIX и POSIX могут легко работать над созданием встраиваемых систем реального времени,

поскольку та часть программного интерфейса операционной системы реального времени, которая не связана непосредственно с функциями реального времени, является для них уже известной территорией.

Среда разработки: резидентная модель и кросс-платформенная модель

С помощью интерфейсного оборудования, аналогичного целевой системе среды исполнения, рабочая станция под управлением POSIX-совместимой ОС может служить в качестве функционального звена встроенной системы. В результате разработка приложения может легко выполняться на настольной системе на основе резидентной модели разработки.

Даже при использовании кросс-платформенной модели разработки программный интерфейс (API) остается принципиально тем же самым. Какая бы платформа (QNX Neutrino, Solaris, Windows и т. д.) или целевой процессор (x86, ARM, MIPS, PowerPC и др.) не применялись, программисту не нужно беспокоиться, например, о проблемах следования байтов (endian), выравнивания (alignment) или ввода/вывода данных.

Преимущества ОС QNX Neutrino для встраиваемых систем

Основной задачей операционной системы является управление ресурсами компьютеров. Все процессы внутри системы — планирование прикладных программ, запись файлов на диск, пересылка данных по компьютерной сети и т. д. — должны осуществляться с максимальной степенью цельности (seamlessly) и прозрачности.

В некоторых средах требуется более жесткое управление ресурсами и планирование процессов. Например, работа приложений реального времени зависит от способности ОС обрабатывать множество событий одновременно и реагировать на эти события в течение строго определенного периода времени. Чем выше способность ОС к реагированию, тем больше у приложения "времени" на выполнение своих функций.

ОС QNX Neutrino идеально подходит для *встраиваемых приложений реального времени*. Она может быть масштабирована до самых компактных конфигураций и способна работать в многозадачном режиме, управлять потоками, осуществлять планирование процессов по приоритетам (priority-driven preemptive scheduling) и выполнять быстрое переключение контекстов (fast context-switching). Более того, операционная система предоставляет все эти возможности посредством программного интерфейса, основанного на

стандартах POSIX. Таким образом, компактность системы достигается не в ущерб стандартам.

Кроме того, ОС QNX Neutrino обладает довольно большой гибкостью. Разработчики могут легко изменять ее конфигурацию в соответствии с требованиями создаваемых приложений. Вы можете использовать только те ресурсы, которые необходимы для конкретной задачи, изменяя систему в диапазоне от минимальной конфигурации микроядра с несколькими небольшими модулями до полнофункциональной сетевой системы, предназначенной для обслуживания сотен пользователей.

Уникальные возможности эффективности, модульности и простоты достигаются в ОС QNX Neutrino благодаря двум фундаментальным принципам:

- микроядерная архитектура;
- межзадачное взаимодействие на основе обмена сообщениями.

Микроядерная архитектура

Яркие словечки быстро входят в моду и также быстро выходят из нее. Производители программного обеспечения часто с радостью подхватывают какое-нибудь модное слово, чтобы назвать им свой продукт, даже если это слово не совсем подходит для него.

Термин "микроядерный" стал модным в последнее время. Хотя многие из новых операционных систем называются "микроядерными" (или даже "наноядерными" — nanokernel), этот термин может мало что значить без четкого его определения. Попробуем сделать это.

Микроядерная операционная система построена на основе миниатюрного ядра, обеспечивающего минимальные службы для произвольной группы взаимодействующих процессов, которые, в свою очередь, обеспечивают функциональность более высокого уровня. Само по себе микроядро не имеет файловой системы и не выполняет многих других функций, осуществляемых операционными системами. Все эти функции реализуются дополнительными процессами.

Главная цель в проектировании микроядерной операционной системы состоит не в том, чтобы просто сделать ее "компактной". Создание микроядерной операционной системы требует принципиального изменения подхода к реализации функциональности операционной системы. Первичным здесь становится модульный принцип, тогда как стремление к компактности является вторичным. Добавлять приставку "микро" к какому-либо ядру просто потому, что оно имеет небольшой размер, было бы полным заблуждением.

Поскольку механизм межзадачного взаимодействия (interprocess communication (IPC) services), действующий в микроядре, служит в качестве "связующего материала", который организует ОС как единое целое, от производительности и гибкости этого механизма зависит производительность всей ОС. Если не учитывать данный механизм, то микроядро можно сравнить, грубо говоря, с исполнительным модулем реального времени (как с точки зрения объема выполняемых им функций, так и с точки зрения скорости их выполнения).

Отличие микроядра от исполнительного модуля реального времени состоит в том, как именно используются функции межзадачного взаимодействия для того, чтобы расширить функциональность ядра посредством дополнительных рабочих процессов. Поскольку ОС реализуется как группа процессов, взаимодействующих под управлением микроядра, процессы, написанные пользователем, могут служить и как приложения, и как расширения базовой функциональности операционной системы, используемой для специализированных промышленных приложений. В результате ОС становится "открытой" и легко наращиваемой. Более того, расширения ОС, написанные пользователем, не влияют на надежность ядра ОС.

Недостатком многих исполнительных модулей реального времени, реализующих стандарт POSIX 1003.1, является то, что их среда исполнения работает на основе однопроцессной многопоточной модели, в которой адресные пространства потоков не изолированы друг от друга (unprotected memory between threads). Такая среда является всего лишь частным случаем многопроцессной модели, определяемой в POSIX, и она не поддерживает функцию `fork()`. ОС QNX Neutrino, наоборот, использует возможности работы с MMU-оборудованием для обеспечения полнофункциональной POSIX-модели в защищенной среде.

На приведенных далее схемах (рис. 1.1—1.3) видно, как истинное микроядро обеспечивает *полную защиту памяти*, причем не только для пользовательских приложений, но и для компонентов ОС (например, драйверов устройств, файловых систем и т. д.).

Рис. 1.1. Обычные исполнительные модули реального времени не обеспечивают защиту памяти

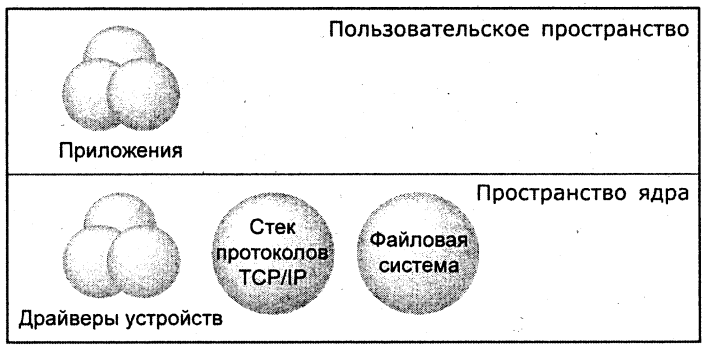


Рис. 1.2. В монолитной ОС системные процессы не защищены

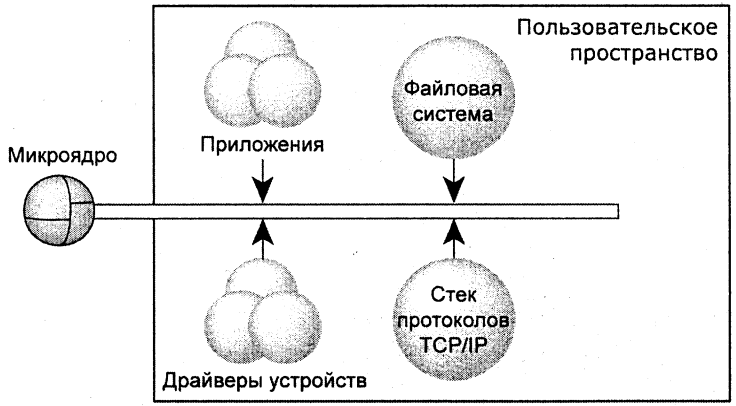


Рис. 1.3. Микроядро обеспечивает полную защиту памяти

Первая версия ОС QNX была выпущена в 1981 г. В каждой последующей версии мы воплощали тот опыт, который был получен нами при создании предыдущих версий, и поэтому ОС QNX Neutrino является на сегодняшний день самой мощной и масштабируемой операционной системой. Мы уверены, что, благодаря нашему многолетнему опыту, мы создали операционную систему, которая способна обеспечить максимальную производительность, используя минимальные ресурсы.

Операционная система как организованный набор процессов

ОС QNX Neutrino строится на основе компактного микроядра, способного управлять группой взаимодействующих процессов. Как видно на рис. 1.4, структура операционной системы больше напоминает "слаженную команду", чем иерархию, так как несколько равноправных "игроков" взаимодействуют в ней между собой посредством координирующего ядра.

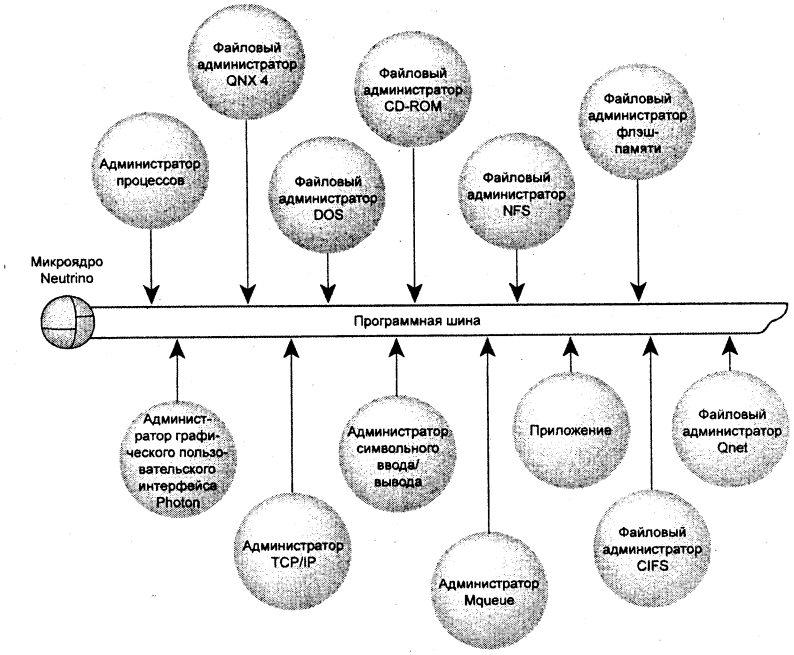


Рис. 1.4. Архитектура ОС QNX Neutrino

ОС QNX Neutrino действует как своего рода "программная шина", позволяющая динамически присоединять/отсоединять модули ОС по мере необходимости.

Истинное ядро

Ядро — это сердце любой операционной системы. В некоторых системах "ядро" несет столько функций, что оно уже само по себе является *целой операционной системой*!

Но микроядро в ОС QNX Neutrino — это истинное ядро, потому что, во-первых, как и ядро исполнительного модуля реального времени, оно очень компактно, а во-вторых, оно предназначено для выполнения только нескольких базовых функций:

- *управление потоками (thread services)* посредством POSIX-примитивов для создания потоков;
- *управление сигналами (signal services)* посредством примитивов сигналов;
- *обмен сообщениями (message-passing services)*, с помощью которого микроядро выполняет трассировку всех сообщений, пересылаемых между всеми потоками внутри системы;

- *синхронизация (synchronization services)* посредством примитивов синхронизации потоков;
- *планирование (scheduling services)*, с помощью которого микроядро осуществляет планирование потоков в реальном времени на основе различных алгоритмов;
- *управление таймерами (timer services)*, с помощью которого микроядро обеспечивает большой набор POSIX-таймеров;
- *управление процессами (process management services)*, выполняемое администратором процессов, вместе с которым ядро образует единый модуль `procnto`. Администратор процессов предназначен для управления процессами, памятью и пространством имен путей (pathname space).

В отличие от потоков, микроядро никогда не планируется на выполнение. Процессор выполняет код в микроядре только в случае явного вызова ядра, при возникновении исключения или в результате аппаратного прерывания.

Системные процессы

Все службы ОС, за исключением тех, которые выполняются обязательным модулем микроядра/администратора процессов (`procnto`), обрабатываются посредством *стандартных процессов*. Система может содержать следующие компоненты:

- администраторы файловых систем;
- администраторы устройств символьного ввода/вывода;
- графический пользовательский интерфейс (Photon);
- сетевой администратор;
- стек протоколов TCP/IP.

Системные процессы и пользовательские процессы

Системные процессы по сути никак неотличимы от пользовательских процессов — они используют те же самые унифицированные службы программного интерфейса и ядра, которые доступны для любого пользовательского процесса, имеющего соответствующие привилегии.

Именно такая архитектура дает ОС QNX Neutrino уникальные возможности расширяемости. Поскольку большинство служб ОС выполняются стандартными системными процессами, конфигурация ОС может быть легко дополнена новыми компонентами, для чего достаточно написать соответствующие программы, предназначенные для выполнения новых служб.

На самом деле граница между операционной системой и приложением может стать очень размытой. Единственное различие между системными

службами и приложениями состоит в том, что службы ОС управляют ресурсами для клиентских задач.

Допустим, вы разработали сервер базы данных. Как следует классифицировать этот процесс? Сервер базы данных, так же как и файловая система, принимает (посредством сообщений) запросы на открытие файлов и чтение или запись данных. Хотя запросы к серверу базы данных в действительности могут быть более сложными, оба сервера действуют аналогичным образом в том смысле, что они обеспечивают программный интерфейс (на основе обмена сообщениями), с помощью которого клиенты получают доступ к ресурсу. Оба являются независимыми процессами, которые могут быть написаны конечным пользователем и которые могут быть запущены или остановлены по необходимости.

Сервер базы данных можно рассматривать при одной конфигурации как системный процесс, а при другой — как приложение. *По сути, это одно и то же!* Важным здесь является то, что эти процессы могут выполняться в ОС без необходимости модифицировать стандартные компоненты самой ОС. Для разработчиков, создающих специализированные встроенные системы, это дает гибкие возможности расширения ОС в соответствии с требованиями разрабатываемых ими приложений и избавляет от необходимости обращаться к исходному коду ОС.

Драйверы устройств

Драйверы устройств позволяют ОС и прикладным программам согласованно использовать базовое оборудование компьютера (например, жесткий диск, сетевой интерфейс). Если в большинстве ОС драйверы устройств должны быть встроены в структуру ОС, в QNX Neutrino они могут запускаться и останавливаться как стандартные процессы. В результате добавление новых драйверов устройств не нарушает работу какой-либо части ОС, так как драйверы могут разрабатываться и подвергаться отладке как обычные приложения.

Межзадачное взаимодействие

Для того чтобы осуществить выполнение нескольких потоков одновременно в многозадачной операционной системе реального времени, эта ОС должна иметь механизмы обеспечения взаимодействия потоков между собой.

Межзадачное взаимодействие позволяет разрабатывать приложения в виде набора взаимодействующих процессов, каждый из которых обрабатывает только определенную часть программной задачи.

ОС QNX Neutrino обеспечивает простой, но мощный набор возможностей межзадачного взаимодействия, который значительно упрощает разработку приложений, состоящих из некоторого числа взаимодействующих процессов.

QNX Neutrino как операционная система на основе обмена сообщени ями

QNX была первой коммерческой операционной системой, в которой был применен обмен сообщениями между процессами в качестве основного метода межзадачного взаимодействия. Полная интеграция механизма межзадачного обмена сообщениями в саму архитектуру операционной системы во многом определяет ее высокую производительность, простоту и эффективность.

В ОС QNX Neutrino сообщение представляет собой группу байтов, передаваемых от одного процесса другому. Сообщение не несет никакого абсолютного значения в операционной системе. Его содержание имеет смысл только для отправителя сообщения и его получателя.

Механизм обмена сообщениями позволяет процессам не только обмениваться данными между собой, но и является средством синхронизации выполнения нескольких процессов. Процессы отправляют и получают сообщения, а также отвечают на них. При этом происходят различные "изменения состояния" процессов, от которых зависит, когда и в течение какого времени эти процессы должны выполняться. Определяя состояние и приоритет каждого из процессов, микроядро осуществляет их планирование оптимальным способом и с наиболее эффективным расходованием вычислительных ресурсов. Таким образом, механизм обмена сообщениями является основополагающим и постоянно действует на всех уровнях операционной системы.

Для приложений, работающих в режиме реального времени, и других приложений критического назначения требуется, чтобы механизм межзадачного взаимодействия имел высокую степень надежности, поскольку процессы, на основе которых работают такие приложения, очень тесно связаны между собой. Метод обмена межзадачными сообщениями как раз и позволяет достичь строгого управления и высокой надежности выполнения приложений в ОС QNX Neutrino.

Распределенные сетевые конфигурации

Локальная компьютерная сеть, грубо говоря, является механизмом, с помощью которого несколько взаимосвязанных компьютеров могут обмениваться файлами и совместно использовать периферийные устройства. ОС QNX Neutrino идет намного дальше этой простой модели и позволяет интегрировать всю компьютерную сеть в единый, общий набор ресурсов.

Любой поток на любой машине, включенной в компьютерную сеть, может напрямую использовать любой ресурс на любой другой машине. С точки зрения приложения, между локальным и удаленным ресурсом нет никакой разницы, поэтому нет необходимости встраивать в приложения специальные компоненты для использования удаленных ресурсов.

Пользователи могут обращаться к файлам в любой части компьютерной сети, а также использовать любые периферийные устройства и запускать приложения на любой машине в этой компьютерной сети (при условии что им предоставлены соответствующие права на выполнение этих действий). Процессы могут аналогичным образом взаимодействовать между собой независимо от того, в какой части сети они выполняются. Такой гибкий и прозрачный обмен данными внутри компьютерной сети возможен благодаря механизму межзадачного взаимодействия на основе обмена сообщениями.

Однокомпьютерная модель

ОС QNX Neutrino изначально разрабатывалась как сетевая операционная система. В некотором смысле, компьютерная сеть, построенная на основе QNX Neutrino, больше напоминает единую универсальную ЭВМ, чем набор индивидуальных микрокомпьютеров. В распоряжении пользователей имеется огромный набор ресурсов, которые могут быть применены в любом приложении. Однако в отличие от универсальной ЭВМ, ОС QNX Neutrino является очень гибкой средой, так как на любом ее узле может быть предоставлен необходимый объем вычислительных мощностей в соответствии с потребностями каждого пользователя.

Например, в критически важных средах приложения, предназначенные для управления устройствами ввода/вывода в реальном масштабе времени, могут потребовать от операционной системы повышенной производительности, в отличие от таких менее важных приложений, как, например, веб-браузер. Компьютерная сеть на основе ОС QNX Neutrino достаточно гибка, чтобы поддерживать оба типа приложений *одновременно*, благодаря тому, что операционная система позволяет оптимально распределять вычислительные мощности между системными устройствами, не жертвуя при этом совместимостью с приложениями рабочего стола. Более того, функции обеспечения работы в реальном масштабе времени (как, например, наследование приоритетов (priority inheritance)) интегрированы во всю компьютерную сеть на основе QNX Neutrino и действуют на всех ее уровнях независимо от того, какая среда передачи используется (волоконно-оптическая связь, последовательное соединение и т. д.).

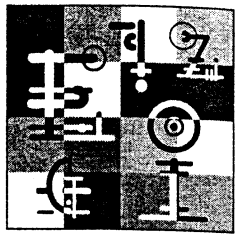
Гибкие сетевые возможности

Компьютерные сети на основе ОС QNX Neutrino могут быть объединены с помощью различного оборудования и стандартных протоколов. Благодаря их полной прозрачности для прикладных программ и пользователей, архитектура ОС может быть легко изменена в любой момент без нарушения ее работы.

Каждый узел в компьютерной сети имеет уникальное имя-идентификатор. Это имя является единственным средством, по которому можно отличить сетевую конфигурацию ОС от автономной.

Подобная степень прозрачности — это еще одно подтверждение чрезвычайно высокой эффективности системной архитектуры на основе механизма обмена межзадачными сообщениями. Во многих системах такие важные функции, как, например, сетевое взаимодействие, межзадачное взаимодействие или даже обмен межзадачными сообщениями, реализуются поверх ОС, а не интегрируются прямо в ядро системы. В результате система зачастую имеет неудобный, неэффективный интерфейс с "двойными стандартами", в котором взаимодействие между процессами происходит вне ядра, а достичь этого загадочного монолитного ядра можно только через его собственный интерфейс.

В отличие от монолитных систем, QNX Neutrino построена на основе принципа эффективного взаимодействия, который является ключом к эффективному функционированию. Таким образом, механизм обмена сообщениями — это краеугольный камень архитектуры нашего микроядра. Он увеличивает эффективность *всех* транзакций, происходящих между всеми процессами во всей системе независимо от используемой среды передачи, будь это простое прямое соединение между компьютерами или километр витой пары.



Глава 2

Микроядро ОС QNX Neutrino

Введение

Микроядро ОС QNX Neutrino реализует основные функции стандартов POSIX, используемые во встраиваемых системах реального времени, а также базовые службы обмена сообщениями. Те POSIX-функции, которые не реализованы в микроядре (например, функции файлового или аппаратного ввода/вывода), выполняются дополнительными процессами или разделяемыми библиотеками.

В каждой новой версии микроядра QNX код, предназначенный для реализации обращений к ядру, становился все более компактным. Определения объектов на самых нижних уровнях в коде ядра становились все более четкими, что позволяло увеличить возможности его повторного использования (например, выполнялось сворачивание различных форм POSIX-сигналов, сигналов реального времени и QNX-импульсов в общие структуры данных и код, предназначенный для управления этими структурами).

На самых нижних уровнях микроядра расположено всего несколько базовых объектов, а также четко отрегулированные процедуры для управления ими. Это и есть та основа, на которой строится ОС QNX Neutrino (рис. 2.1).

Некоторые разработчики считают, что компактность и высокая производительность микроядра ОС QNX Neutrino достигается благодаря его реализации в виде ассемблерного кода. В действительности микроядро ОС QNX Neutrino реализовано преимущественно на C, а компактность и высокая производительность достигаются с помощью четко отлаженных алгоритмов и структур данных, а не посредством оптимизации на уровне ассемблерного кода.

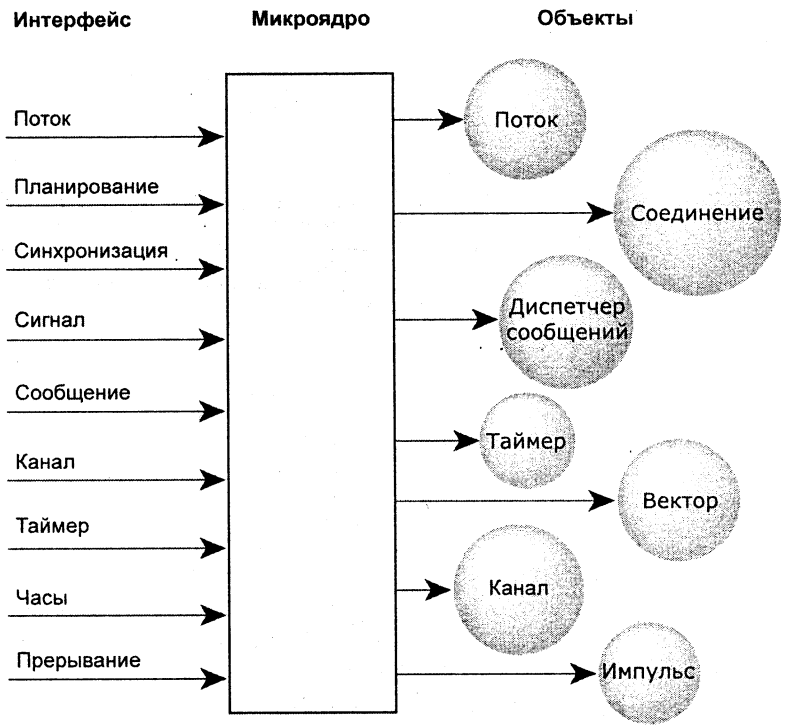


Рис. 2.1. Микроядро ОС QNX Neutrino

Реализация ОС QNX Neutrino

Исторически складывалось так, что операционные системы QNX испытывали на себе "прикладное давление" одновременно с двух сторон спектра вычислительных моделей. С одной стороны, это встраиваемые системы с ограниченными ресурсами памяти, а с другой — высокотехнологичные машины симметричной многопроцессорной обработки с гигабайтами физической памяти. Именно поэтому при разработке ОС QNX Neutrino в качестве проектных целей были приняты оба эти на первый взгляд исключающие друг друга подхода. Их выбор связан со стремлением значительно расширить функциональный диапазон операционных систем QNX за пределы возможностей других ОС.

Потоки и функции реального времени в POSIX

Поскольку в ОС QNX Neutrino большинство служб по обеспечению работы в реальном масштабе времени и по планированию потоков реализуется прямо в микроядре, эти службы могут работать даже без дополнительных модулей ОС.

Кроме того, некоторые из стандартов POSIX предполагают, что данные службы должны работать даже при отсутствии модели на основе процессов. Для достижения этого, ОС осуществляет прямую поддержку потоков с помощью администратора процессов, который позволяет управлять в том числе и процессами со множеством потоков.

Нужно отметить, что многие исполнительные модули и ядра, предназначенные для работы в реальном масштабе времени, не обеспечивают изоляции между потоками и не имеют модели на основе процессов. Однако без модели на основе процессов невозможно достичь полного соответствия стандартам POSIX.

Системные службы

В микроядре ОС QNX Neutrino существуют системные вызовы (kernel calls), которые служат для управления следующими объектами:

- потоками (threads);
- сообщениями (message passing);
- сигналами (signals);
- часами (clocks);
- таймерами (timers);
- обработчиками прерываний (interrupt handlers);
- семафорами (seamphores);
- блокировками взаимного исключения, или мутексами (mutexes);
- условными переменными (condvars);
- барьерами (barriers);

ОС QNX Neutrino целиком построена на основе таких вызовов, причем ядро ОС полностью вытесняемо (preemptable), в том числе и во время обмена сообщениями между процессами (обмен сообщениями продолжается с той точки, на которой был прерван перед вытеснением).

Благодаря простоте архитектуры микроядра, появляется возможность устанавливать ограничение на максимальный объем нереентерабельного кода в адресном пространстве ядра, что облегчает решение сложных задач многопроцессорной обработки. В микроядро были включены те службы, которые порождали наиболее короткую ветвь исполняемого кода. Операции, требующие значительных ресурсов (например, загрузка процесса), были переданы внешним процессам и потокам, в которых работа по переключению на контекст другого потока пренебрежительно мала в сравнении с работой по обработке запроса, выполняемой внутри этого потока.

Строгое применение этого подхода к разграничению функций между ядром и внешними процессами развенчивает миф о том, что микроядерной ОС свойственны более высокие накладные расходы (runtime overhead) при выполнении операций, чем операционной системе с монолитным ядром. Если оценить работу, которая выполняется между переключениями контекстов во время обмена сообщениями, и учесть высокую скорость этих переключений в минимизированном ядре, то становится очевидным, что объем времени, расходуемый на выполнение этих действий, настолько минимален, что становится пренебрежимо малым в сравнении с работой, производимой для обработки запросов от механизма обмена межзадачными сообщениями.

На рис. 2.2 показан механизм вытеснения в ядре (для процессоров с архитектурой x86) без поддержки симметричной многопроцессорной обработки (SMP).

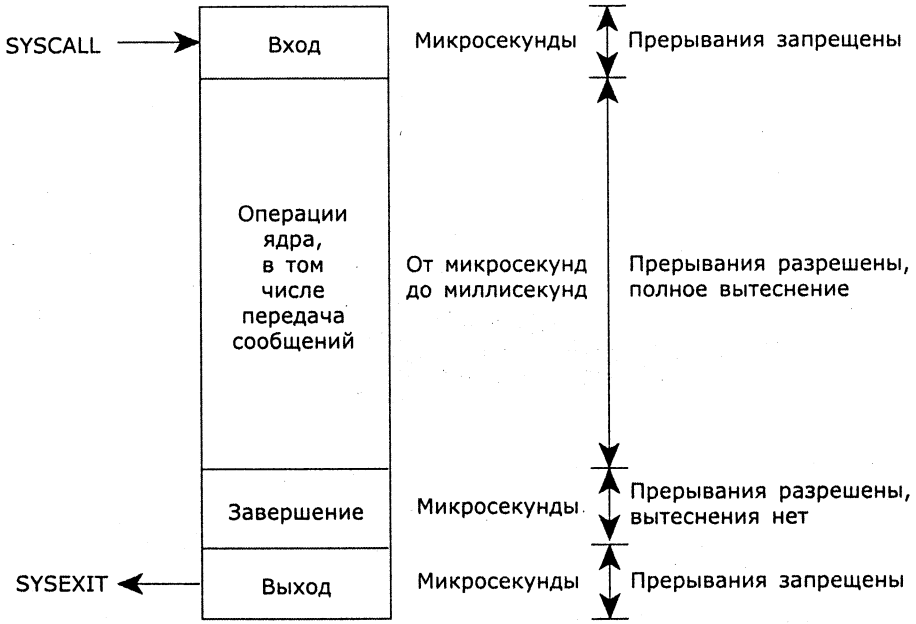


Рис. 2.2. Механизм вытеснения в ОС QNX Neutrino

Запрет прерываний или отсутствие вытеснения происходит только на короткие периоды времени (как правило, эти периоды не превышают порядка сотен наносекунд).

Потоки и процессы

При разработке какого-либо приложения (например, приложение реального времени, встраиваемое приложение, графическое приложение и т. п.) программисту может понадобиться сделать так, чтобы в нем одновременно выполнялось несколько алгоритмов. Эта одновременность может быть достигнута с помощью модели многопоточности POSIX, в которой процесс состоит из одного или нескольких потоков управления.

Поток можно рассматривать как минимальную "единицу выполнения", т. е. единицу планирования (scheduling) и выполнения в микроядре. Процесс, в свою очередь, можно рассматривать как объект, который содержит в себе эти потоки и который определяет для их выполнения свое "адресное пространство". Процесс всегда содержит по крайней мере один поток.

В зависимости от приложения, потоки могут выполняться независимо друг от друга и не требовать взаимодействия между разными алгоритмами (что является редким случаем), или же для их выполнения требуется тесная взаимосвязь с возможностью быстрого взаимодействия и строгой синхронизации. Для обеспечения такого взаимодействия и синхронизации в ОС QNX Neutrino предусмотрен целый набор специальных служб и механизмов.

В следующих далее вызовах из библиотек *pthread_** (POSIX Threads) не используются вызовы микроядра по управлению потоками:

pthread_attr_destroy()
pthread_attr_getdetachstate()
pthread_attr_inheritsched()
pthread_attr_getschedparam()
pthread_attr_getschedpolicy()
pthread_attr_getscope()
pthread_attr_getstackaddr()
pthread_attr_getstacksize()
pthread_attr_init()
pthread_attr_setdetachstate()
pthread_attr_setinheritsched()
pthread_attr_setschedparam()
pthread_attr_setschedpolicy()
pthread_attr_setscope()
pthread_attr_setstackaddr()
pthread_attr_setstacksize()
pthread_attr_getstackaddr()
pthread_cleanup_pop()
pthread_cleanup_push()
pthread_equal()

pthread_getspecific()
pthread_setspecific()
pthread_testcancel()
pthread_key_create()
pthread_key_delete()
pthread_once()
pthread_self()
pthread_setcancelstate()
pthread_setcanceltype()

В табл. 2.1 приведен список вызовов по управлению потоками POSIX и соответствующие вызовы микроядра.

Таблица 2.1. *Вызовы по управлению потоками POSIX и соответствующие вызовы микроядра*

POSIX-вызов	Вызов микроядра	Описание
pthread_create()	*ThreadCreate()*	Создать новый поток
pthread_exit()	*ThreadDestroy()*	Уничтожить поток
pthread_detach()	*ThreadDetach()*	Отсоединить поток, чтобы не ждать его завершения
pthread_join()	*ThreadJoin()*	Присоединить поток и ждать его кода завершения
pthread_cancel()	*ThreadCancel()*	Завершить поток в следующей точке завершения
отсутствует	*ThreadCtl()*	Изменить характеристики потока, специфичные для ОС QNX Neutrino
pthread_mutex_init()	*SyncTypeCreate()*	Создать мутекс
pthread_mutex_destroy()	*SyncDestroy()*	Уничтожить мутекс
pthread_mutex_lock()	*SyncMutexLock()*	Блокировать мутекс
pthread_mutex_trylock()	*SyncMutexLock()*	Условно блокировать мутекс
pthread_mutex_unlock()	*SyncMutexUnlock()*	Снять блокировку мутекса
pthread_cond_init()	*SyncTypeCreate()*	Создать условную переменную
pthread_cond_destroy()	*SyncDestroy()*	Уничтожить условную переменную

Таблица 2.1 *(окончание)*

POSIX-вызов	Вызов микроядра	Описание
pthread_cond_wait()	*SyncCondvarWait()*	Ожидать условную переменную
pthread_cond_signal()	*SyncCondvarSignal()*	Разблокировать один из потоков, блокированных на условной переменной
pthread_cond_broadcast()	*SyncCondvarSignal()*	Разблокировать все потоки, блокированные на условной переменной
pthread_getschedparam()	*SchedGet()*	Получить параметры планирования и дисциплину потока
pthread_setschedparam()	*SchedSet()*	Установить параметры планирования и дисциплину потока
pthread_sigmask()	*SignalProcMask()*	Проверить или вывести маску сигналов потока
pthread_kill()	*SignalKill()*	Отправить сигнал потоку

ОС QNX Neutrino можно конфигурировать определенным образом для реализации некоторого набора потоков и процессов (в соответствии со стандартами POSIX). Все процессы отделены друг от друга с помощью блока управления памятью (Memory Management Unit, MMU), и каждый процесс может содержать один или несколько потоков, использующих адресное пространство процесса.

От используемой среды зависят не только возможности параллельного выполнения задач приложением, но и механизмы межзадачного взаимодействия и синхронизации, которые могут понадобиться для работы этого приложения.

Примечание

Хотя термин "межзадачное взаимодействие" (InterProcess Communication, IPC) обычно относят к процессам, мы используем его для обозначения взаимодействия между *потоками* (внутри одного процесса или внутри разных процессов).

Атрибуты потока

Хотя потоки внутри процесса и используют совместно одно общее адресное пространство этого процесса, каждый из этих потоков имеет некоторые "собственные" данные. В некоторых случаях эти данные защищаются внутри ядра (например, идентификатор потока tid), в то время как другие

данные остаются незащищенными в адресном пространстве процесса (например, каждый поток имеет свой собственный стек). Перечислим некоторые из наиболее важных ресурсов, относящихся к потоку:

- идентификатор потока (tread identifier, tid) — каждый поток обозначается своим собственным целочисленным идентификатором, начиная с 1. Внутри процесса потоки имеют уникальные идентификаторы;
- набор регистров (register set) — каждый поток имеет свой указатель команд (Instruction Pointer, IP), указатель стека (Stack Pointer, SP) и другие регистры, составляющие контекст потока;
- стек (stack) — каждый поток имеет свой собственный стек, хранимый в адресном пространстве процесса, к которому поток относится;
- маска сигналов (signal mask) — каждый поток имеет свою собственную маску сигналов;
- локальная память потока (ЛПП) (Thread Local Storage, TLS) — это системная область данных потока. Локальная память потока используется для хранения информации, относящейся к каждому отдельному потоку (например, tid, pid, базовый адрес стека (stack base), errno, привязка ключей и данных, относящихся к потоку). К локальной памяти потока не нужно обращаться напрямую из пользовательского приложения. Поток может использовать содержимое ЛПП как ключ для привязывания пользовательских данных;
- обработчик завершений (cancellation handler) — содержит функции обратного вызова, выполняемые при завершении потока.

Данные, относящиеся к потоку, реализуются в библиотеке *pthread* и хранятся в локальной памяти потока. Они обеспечивают механизм, предназначенный для связывания глобального целочисленного ключа процесса (process global integer key) с уникальным значением данных для каждого потока. Для того чтобы использовать данные потока, сначала создается новый ключ, а затем с этим ключом связывается уникальное значение данных (по каждому потоку). Значение данных может, например, представлять собой целое число или являться указателем на динамическую структуру данных (dynamically allocated data structure). После этого ключ может по каждому потоку возвращать то значение данных, с которым он связан.

Типичным примером использования данных потока является их применение для функции, совместимой с механизмом многопоточности и предназначенной для формирования контекста каждого вызывающего потока (рис. 2.3).

Для создания и управления этими данными используются следующие функции — табл. 2.2.

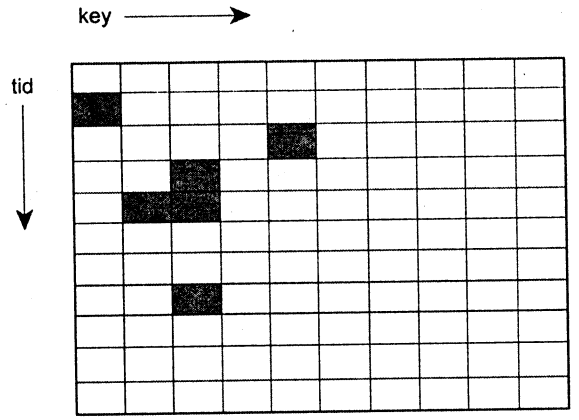


Рис. 2.3. Разреженная матрица (`tid`, `key`) отображения значений

Таблица 2.2. *Функции для управления и создания данных потока*

Функция	Описание
Pthread_key_create()	Создать ключ данных с функцией-деструктором
Pthread_key_delete()	Уничтожить ключ данных
Pthread_setspecific()	Связать значение данных с ключом данных
Pthread_getspecific()	Получить значение данных, связанное с ключом данных

Жизненный цикл потока

Потоки создаются и уничтожаются динамически, и их количество внутри процесса может изменяться в значительных пределах. Создание потока (*pthread_create()*) включает в себя выделение и инициализацию необходимых ресурсов внутри адресного пространства процесса (например, стека потока), а также запуск выполнения потока с некоторой функции в данном адресном пространстве.

Завершение потока (*pthread_exit()*, *pthread_cancel()*) включает в себя останов потока и освобождение ресурсов потока. Говоря в целом, когда поток запущен, он может находиться в одном из двух состояний: "готов" (ready) или "блокирован" (blocked). Если же говорить более детально, то поток может иметь одно из следующих состояний (рис. 2.4):

- CONDVAR — поток блокирован на условной переменной (например, при вызове функции *pthread_condvar_wait()*);
- DEAD — поток завершен и ожидает завершения другого потока;

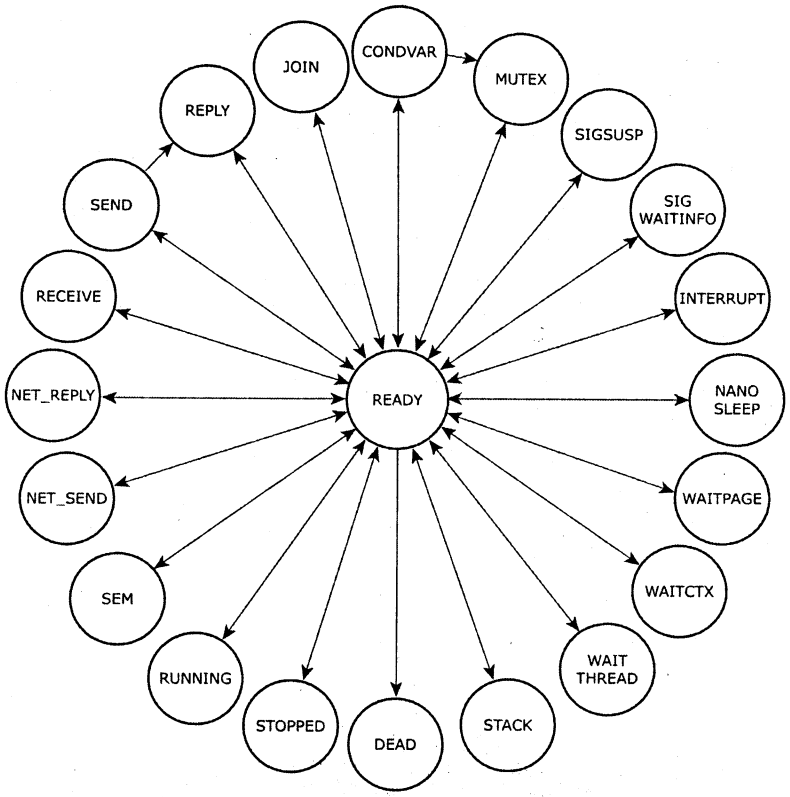


Рис. 2.4. Возможные состояния потока

- INTERRUPT — поток блокирован и ожидает прерывания (т. е. поток вызвал функцию *InterruptWait()*);
- JOIN — поток блокирован и ожидает завершения другого потока (например, при вызове функции *pthread_join()*);
- MUTEX — поток блокирован блокировкой взаимного исключения (например, при вызове функции *pthread_mutex_lock()*);
- NANOSLEEP — поток находится в режиме ожидания в течение короткого периода времени (например, при вызове функции *nanosleep()*);
- NET_REPLY — поток ожидает ответа на сообщение от другого узла сети (т.е. поток вызвал функцию *MsgReply*()*);
- NET_SEND — поток ожидает получения импульса или сигнала от другого узла сети (т. е. поток вызвал функцию *MsgSendPulse()*, *MsgDeliverEvent()* или *SignalKill()*);
- READY — поток ожидает выполнения, пока процессор занят выполнением другого потока равного или более высокого приоритета;

- RECEIVE — поток блокирован на операции получения сообщения (например, при вызове функции *MsgReceive()*);
- REPLY — поток блокирован при ответе на сообщение (т. е. при вызове функции *MsgSend()* и получении сообщения сервером);
- RUNNING — поток выполняется процессором;
- SEM — поток ожидает освобождения семафора (т. е. поток вызвал функцию *SyncSemWait()*);
- SEND — поток блокируется при отправке сообщения (т. е. после того, как поток вызвал функцию *MsgSend()*, но получения сообщения сервером еще не произошло);
- SIGSUSPEND — поток блокирован и ожидает сигнала (т. е. поток вызвал функцию *sigsuspend()*);
- SIGWAITINFO — поток блокирован и ожидает сигнала (т. е. поток вызвал функцию *sigwaitinfo()*);
- STACK — поток ожидает выделения виртуального адресного пространства для своего стека (родительский поток вызывает функцию *ThreadCreate()*);
- STOPPED — поток блокирован и ожидает сигнала SIGCONT;
- WAITCTX — поток ожидает доступности нецелочисленного контекста (например, с плавающей запятой);
- WAITPAGE — поток ожидает выделения физической памяти для виртуального адреса;
- WAITTHREAD — поток ожидает завершения создания дочернего потока (т. е. поток вызвал функцию *ThreadCreate()*).

Планирование потоков

Выполнение операций планирования

При каждом вызове ядра, исключении или аппаратном прерывании, в результате которого управление передается коду ядра, выполнение активного потока временно приостанавливается. Действие планирования совершается в момент изменения состояния любого потока независимо от того, в каком процессе поток расположен. Планирование потоков осуществляется по всем процессам сразу.

Как правило, выполнение приостановленного потока через некоторое время возобновляется. При этом планировщик (scheduler) выполняет переключение контекстов с одного потока на другой всякий раз, когда активный поток:

- блокируется (blocked);
- вытесняется (preempted);
- отдает управление (yields).

Когда поток блокируется

Активный поток блокируется, если он должен ожидать какого-либо события (например, ответа на запрос механизма обмена сообщениями, освобождения мутекса и т. д.). Блокированный поток удаляется из очереди готовности (ready queue), после чего запускается поток с наивысшим приоритетом. Когда блокированный поток разблокируется, он помещается в конец очереди готовности на соответствующий приоритетный уровень.

Когда поток вытесняется

Активный поток вытесняется, когда поток с более высоким приоритетом помещается в очередь готовности (т. е. он переходит в состояние готовности (READY) в результате снятия условия блокировки). Прерванный поток остается на соответствующем приоритетном уровне в начале очереди готовности, а поток с более высоким приоритетом начинает выполняться.

Когда поток отдает управление

Активный поток самостоятельно освобождает процессор (*sched_yield()*) и помещается в конец очереди готовности на данном уровне приоритета. После этого запускается поток с наивысшим приоритетом (в том числе им может быть поток, который только что отдал управление).

Планирование и приоритеты

Каждому потоку назначается свой приоритет. Планировщик выбирает поток для выполнения в соответствии с приоритетом каждого потока, находящегося в состоянии готовности (READY), т. е. способного использовать процессор. Таким образом выбирается поток с наивысшим приоритетом.

Рис. 2.5 схематически отображает очередь из шести потоков (A—F), находящихся в состоянии готовности (READY). Все остальные потоки (G—Z) блокированы (BLOCKED). Поток A в настоящий момент является активным. Потоки A, B и C имеют наивысший приоритет, поэтому они совместно используют процессор в соответствии с алгоритмом планирования активного потока.

Всего в ОС QNX Neutrino поддерживается до 256 уровней приоритетов. Приоритет выполнения каждого непривилегированного (nonroot) потока может изменяться в пределах от 1 до 63 (наивысший приоритет), независимо от его *дисциплины планирования (scheduling policy)*. Только привилегированные (root) потоки (т. е. потоки, действующий *uid* которых равен 0) могут иметь приоритет выше 63. Специальный поток с именем *idle* в администраторе процессов, имеет приоритет 0 и всегда готов к выполнению. По умолчанию поток наследует приоритет своего родительского потока.

Команда `procnto -p` позволяет изменить диапазон допустимых приоритетов для непривилегированного процесса:

```
procnto -p приоритет
```

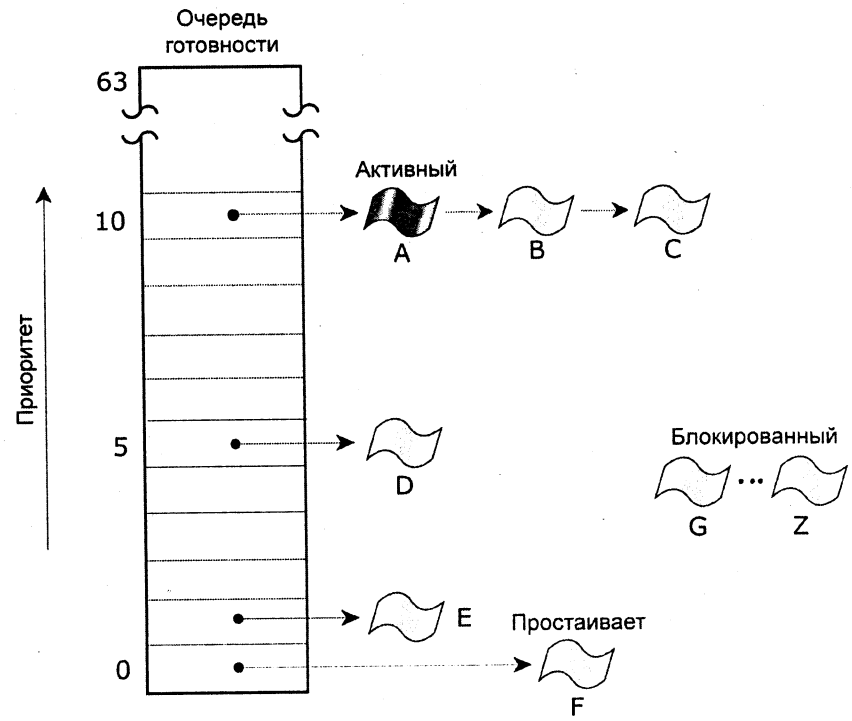



Рис. 2.5. Очередь готовности

В табл. 2.3. приводится список всех диапазонов приоритетов.

Таблица 2.3. *Диапазоны приоритетов*

Уровень приоритета	Владелец
0	поток `idle`
от 1 до *приоритет* – 1	непривилегированный или привилегированный поток
от *приоритет* до 255	привилегированный поток

Следует обратить внимание на то, что для предотвращения *инверсии приоритетов (priority inversion)* ядро может временно повышать приоритет потока.

Потоки, находящиеся в очереди, упорядочиваются по приоритету. В действительности очередь готовых потоков состоит из 256 отдельных очередей — по одной на каждый приоритет. Потоки выстраиваются в очереди по порядку поступления (FIFO) и в соответствии с их приоритетом. Для выполнения выбирается первый поток с наивысшим приоритетом.

Алгоритмы планирования

Для работы с различными приложениями в ОС QNX Neutrino используются следующие алгоритмы планирования:

- FIFO[1]-планирование;
- циклическое планирование (планирование по круговому алгоритму) (round-robin scheduling);
- спорадическое планирование (sporadic scheduling).

Каждый поток в системе может выполняться по любому из методов. Методы применяются для каждого отдельного потока, а не для всех потоков или процессов одновременно.

Следует иметь в виду, что FIFO-планирование и циклическое планирование применяются только в случаях, когда два или более потоков, имеющих *одинаковый приоритет*, находятся в состоянии готовности (READY) (т. е. в этом случае потоки напрямую конкурируют между собой за использование процессора). В спорадическом методе используется "бюджет" выполнения потока. Во всех случаях, когда поток с более высоким приоритетом переходит в состояние READY, он вытесняет (preempts) все другие потоки с более низким приоритетом.

На рис. 2.6 схематически показаны три потока с одинаковым приоритетом, находящиеся в состоянии готовности. Если поток A будет блокирован, то запустится выполнение потока B.

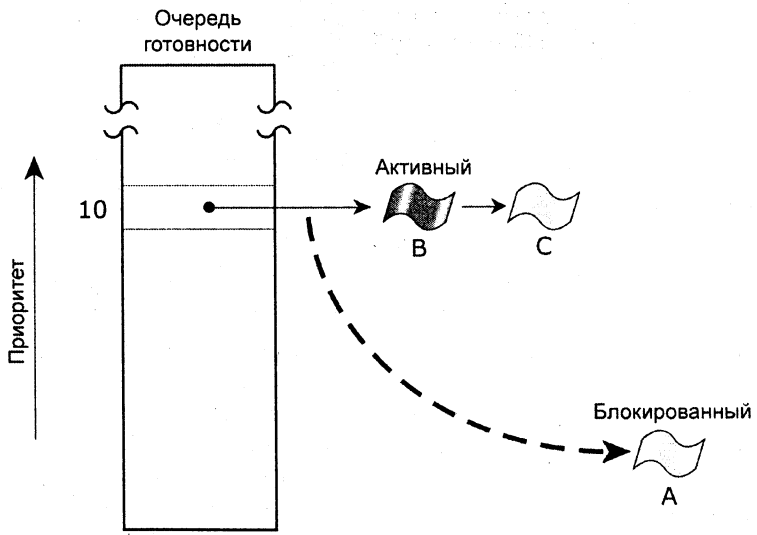


Рис. 2.6. После блокировки потока A запускается выполнение потока B

[1] От англ. First In First Out (первым пришел — первым обслужен) — *Прим. перев.*

Хотя поток наследует алгоритм планирования от своего родительского процесса, он может сделать запрос на его изменение.

FIFO-планирование

В алгоритме FIFO-планирования (рис. 2.7) поток продолжает выполняться до тех пор, пока он:

- принудительно не освободит ресурсы управления (например, при блокировке);
- не будет вытеснен потоком с более высоким приоритетом.

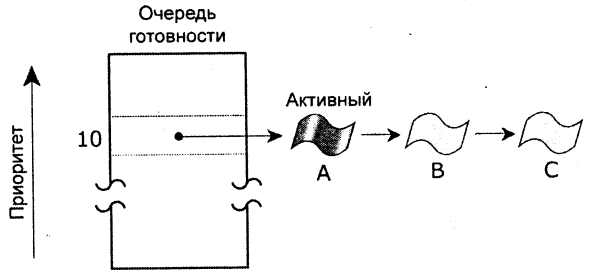


Рис. 2.7. FIFO-планирование

Циклическое планирование

В алгоритме планирования циклического типа поток продолжает выполняться до тех пор, пока он:

- принудительно не освободит ресурсы управления;
- не будет вытеснен потоком с более высоким приоритетом;
- не израсходует свой *квант времени (timeslice).*

Как показано на рис. 2.8, поток A выполняется до тех пор, пока он не израсходует свой квант времени, после чего начинается выполнение следующего потока, находящегося в состоянии готовности (поток B).

Квант времени — это единица времени, выделяемого каждому процессу. После того как поток израсходует свой квант времени, он прерывается и управление получает следующий поток, который находится в состоянии готовности и на том же приоритетном уровне. Квант времени определяется как 4 × тактовый интервал (clock period). (Более подробное определение см. в статье *ClockPeriod()* в "Справочнике по библиотекам языка Си" (Library Reference).)

Замечание

Алгоритм циклического планирования отличается от алгоритма FIFO-планирования только использованием временного квантования.

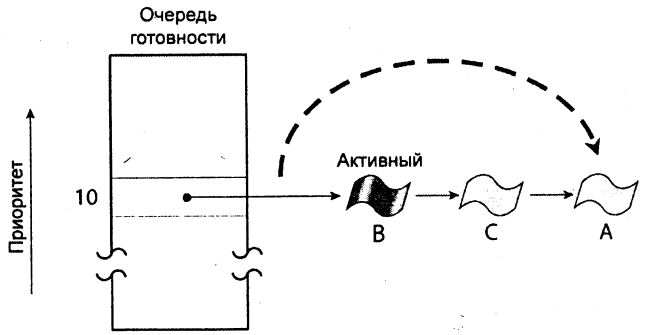


Рис. 2.8. Циклическое планирование

Спорадическое планирование

Алгоритм спорадического планирования обычно используется для задания верхнего лимита на время выполнения потока *в пределах заданного периода времени*. Этот метод необходим при выполнении монотонного частотного анализа (Rate Monotonic Analysis) системы, обслуживающей как периодические, так и апериодические события. По сути, данный алгоритм позволяет потоку обслуживать апериодические события, не препятствуя своевременному выполнению других потоков или процессов в системе.

Как и в FIFO-планировании, поток, для которого применяется спорадическое планирование, выполняется до тех пор, пока он не блокируется или прерывается потоком с более высоким приоритетом. Кроме того, так же как и в адаптивном планировании, поток, для которого применяется спорадическое планирование, получает пониженный приоритет. Однако спорадическое планирование дает значительно более точное управление потоком.

При спорадическом планировании, приоритет потока может динамически изменяться между *приоритетом переднего плана (foreground)* (или нормальным приоритетом) и *фоновым (background)* (или пониженным) приоритетом. Для управления этим спорадическим переходом используются следующие параметры:

- *начальный бюджет потока (initial budget) (C)* — количество времени, за которое поток может выполняться с нормальным приоритетом (N), перед тем как получить пониженный приоритет (L);
- *пониженный приоритет (low priority) (L)* — приоритетный уровень, до которого приоритет потока будет снижен. При пониженном приоритете (L) поток выполняется в фоновом режиме. Если же поток имеет нормальный приоритет (N), он выполняется с приоритетом переднего плана;
- *период пополнения (replenishment period) (T)* — период времени, в течение которого поток может расходовать свой бюджет выполнения (execution budget). Для планирования операций пополнения в POSIX-стандартах

это значение также используется в качестве сдвига по времени, отсчитываемого от того момента, когда поток переходит в состояние готовности (READY);

- *максимальное число текущих пополнений (max number of pending replenishments)* — это значение устанавливает ограничение на количество выполняемых операций пополнения, тем самым ограничивая объем системных ресурсов, выделяемых на дисциплину спорадического планирования.

Замечание

При неправильной настройке системы бюджет выполнения потока может быть исчерпан из-за слишком большого числа блокировок, поскольку в этом случае он не получит достаточного количества пополнений.

Как видно из рис. 2.9, алгоритм спорадического планирования устанавливает начальный бюджет выполнения потока (C), который расходуется потоком в процессе его выполнения и пополняется с периодичностью, определенной параметром T. Когда поток блокируется, израсходованная часть бюджета выполнения потока (R) пополняется через какое-то установленное время (например, через 40 мс), отсчитываемое от момента, когда поток перешел в состояние готовности.

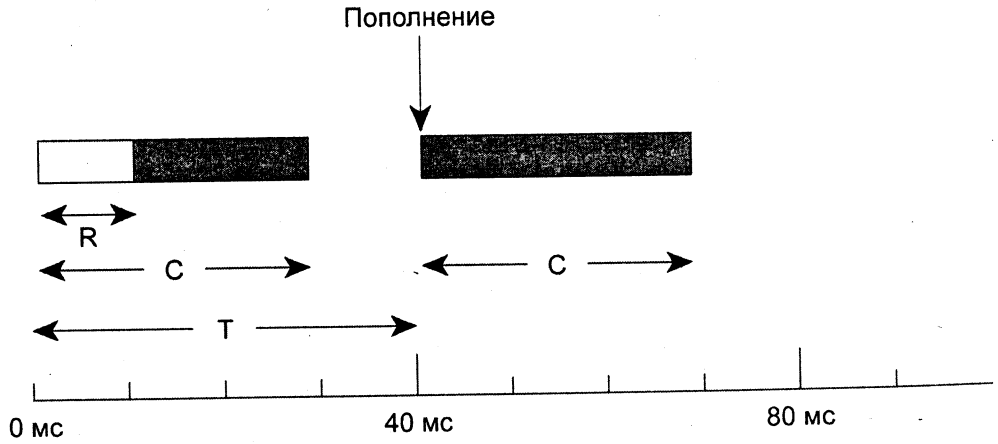


Рис. 2.9. Пополнение периода выполнения потока происходит периодически

При нормальном приоритете (N) поток выполняется в течение периода времени, установленного его начальным бюджетом выполнения (C). После того как этот период истекает, приоритет потока снижается до пониженного уровня (L) до тех пор, пока не произойдет операция пополнения.

Представим, например, систему, в которой поток никогда не блокируется и не прерывается — рис. 2.10.

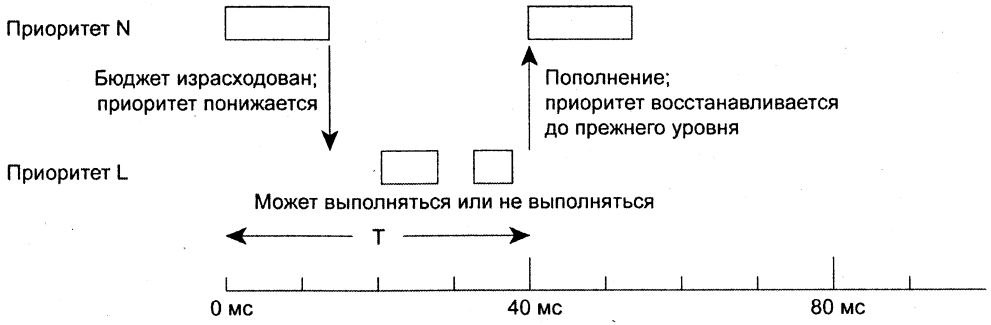


Рис. 2.10. Приоритет потока снижается до того момента, пока его бюджет выполнения не пополнится

В этом случае поток перейдет на уровень с пониженным приоритетом (фоновый режим), на котором его выполнение будет зависеть от приоритета других потоков в системе.

Как только происходит пополнение, приоритет потока повышается до начального уровня. Таким образом, в правильно настроенной системе поток выполняется *каждый период времени T* в течение максимального времени C. Это обеспечивает такой порядок, при котором каждый поток, выполняемый с приоритетом N, будет использовать только C/T процентов системных ресурсов.

Когда поток блокируется несколько раз, несколько операций пополнения могут происходить в разные моменты времени. Это может означать, что бюджет выполнения потока в пределах периода времени T дойдет до значения C; однако на протяжении этого периода бюджет может и не быть непрерывным.

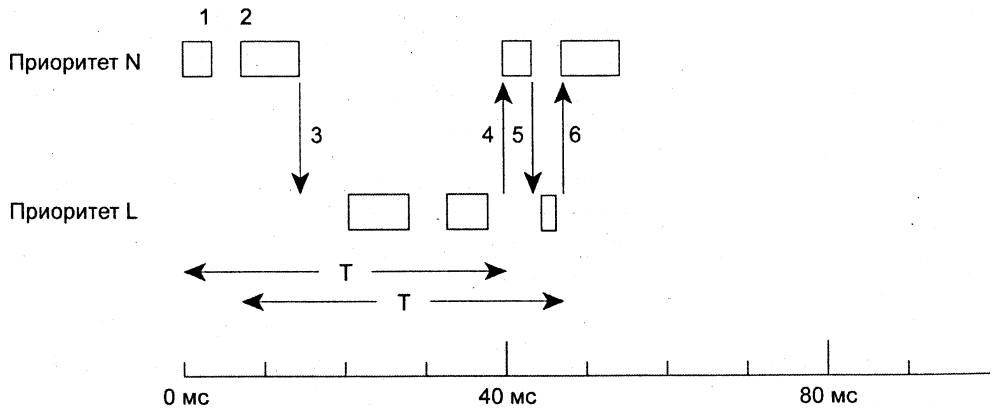


Рис. 2.11. Приоритет потока изменяется между повышенным и пониженным

На схеме рис. 2.11 видно, что в течение каждого 40-мс периода пополнения (T) бюджет выполнения потока (C) составляет 10 мс.

1. Поток блокируется через 3 мс, поэтому 3-мс операция пополнения будет запланирована к выполнению через 40 мс, т. е. на тот момент завершения первого периода пополнения.
2. Выполнение потока возобновляется на 6-й миллисекунде, и этот момент становится началом следующего периода пополнения (T). В бюджете выполнения потока еще остается запас в 7 мс.
3. Поток выполняется без блокировки в течение 7 мс, в результате чего бюджет выполнения потока исчерпывается, и приоритет потока снижается до уровня L, на котором он сможет или не сможет получить управление. Пополнение в объеме 7 мс запланировано произойти на 46-й миллисекунде (40 + 6), т. е. по истечении периода T.
4. На 40-й миллисекунде бюджет потока пополняется на 3 мс (см. шаг 1 на схеме), в результате чего приоритет потока поднимается до нормального.
5. Поток расходует 3 мс своего бюджета и затем снова переходит на пониженный приоритет.
6. На 46-й миллисекунде бюджет потока пополняется на 7 мс (см. шаг 3), и поток снова получает нормальный приоритет.

И так далее. Таким образом, перемещаясь между двумя уровнями приоритета, поток обслуживает апериодические события в системе предсказуемо и управляемо.

Управление приоритетами и алгоритмами планирования

Во время выполнения потока его приоритет может изменяться либо в результате прямого действия самого потока, либо в результате вмешательства ядра при получении сообщения от какого-либо потока с более высоким приоритетом.

Изменяться может не только приоритет, но и алгоритм планирования, применяемый ядром для выполнения потока. В табл. 2.4 приводится список POSIX-вызовов, используемых для управления потоком, а также соответствующие вызовы микроядра, используемые этими библиотечными функциями.

Таблица 2.4. *POSIX-вызовы для управления потоком и вызовы микроядра*

POSIX-вызов	Вызов микроядра	Описание
sched_getparam()	*SchedGet()*	Получить приоритет.
sched_setparam()	*SchedSet()*	Установить приоритет.
sched_getscheduler()	*SchedGet()*	Получить дисциплину планирования.
sched_setscheduler()	*SchedSet()*	Установить дисциплину планирования.

Механизм межзадачного взаимодействия (IPC)

Поскольку все потоки внутри процесса имеют свободный доступ к общему пространству данных, не является ли это "тривиальным" решением всех проблем межзадачного взаимодействия? Разве нельзя использовать разделяемую память для обмена данными и обойтись без всех других механизмов?

Если бы все было так просто!

Проблема заключается в том, что доступ отдельных потоков к общим данным должен быть *синхронизован*. Если один поток пытается считать данные в тот момент, когда другой поток изменяет их, это может привести к катастрофическому отказу системы. Например, если один поток выполняет обновление связного списка, другие потоки не должны читать или изменять этот список, пока поток не закончит свою операцию. Отрывок программного кода, который должен выполняться "последовательно" (serially), т. е. не более чем одним потоком одновременно, называется "критической секцией программного кода" (critical section). Без механизма синхронизации, обеспечивающего последовательный доступ к данным, в программе произошел бы сбой из-за полного нарушения связей.

Мутексы, семафоры и условные переменные — примеры инструментов синхронизации, которые предназначены для решения этой проблемы. Описание этих инструментов будет дано далее в этом разделе.

Хотя службы синхронизации используются для управления взаимодействием между потоками, разделяемая память сама по себе не может решить ряда вопросов межзадачного взаимодействия. Например, хотя потоки и могут взаимодействовать посредством общего пространства данных, это взаимодействие возможно только при условии, что эти потоки находятся внутри одного процесса. Как же быть, если приложению требуется сделать запрос серверу базы данных? Ведь в этом случае данные запроса должны быть посланы на сервер, однако поток, с которым требуется установить взаимодействие, находится *внутри* процесса, выполняемого на сервере базы данных, а адресное пространство этого сервера для приложения недоступно.

Механизм обмена сообщениями используется операционной системой как локально, так и удаленно, и позволяет обращаться ко всем системным службам. Сообщения могут иметь строго определенный и, как правило, очень небольшой размер (например, сообщение может быть отчетом об ошибках по запросу на запись или небольшим запросом на чтение), поэтому механизм обмена сообщениями обеспечивает значительно меньший объем передаваемых данных, чем распределенная разделяемая память, при использовании которой память копируется целыми страницами.

Алгоритмическая сложность потоков

Хотя потоки очень подходят для некоторых системных архитектур, нельзя забывать о ящике Пандоры, в котором скрыто множество сложностей, связанных с их использованием. Нужно отметить, что, как ни странно, несмотря на большую распространенность многозадачности с MMU-защитой, в компьютерную моду вошло использование множества потоков в незащищенном адресном пространстве. Это не только усложняет отладку, но и препятствует созданию надежного программного кода.

Изначально потоки возникли в UNIX-системах как "легковесный" механизм управления параллелизмом, предназначенный для решения проблемы слишком медленного переключения контекстов между "тяжеловесными" процессами. Конечно, сам по себе такой механизм весьма полезен, но возникает очевидный вопрос: почему переключения контекстов между процессами происходят так медленно?

С точки зрения архитектуры, ОС QNX Neutrino в первую очередь решает вопрос производительности переключения контекстов. В действительности переключения контекстов между потоками и между процессами происходят с почти одинаковой скоростью. Однако в QNX Neutrino скорость переключения межзадачных контекстов выше, чем скорость переключения контекстов между потоками в UNIX. В результате в ОС QNX Neutrino отпадает необходимость использовать потоки для решения проблем производительности межзадачного взаимодействия. Вместо этого они служат в качестве инструментов для достижения более высокой степени параллелизма внутри прикладных и серверных процессов.

Без помощи потоков быстрое переключение контекстов между процессами позволяет структурировать приложение как группу взаимодействующих процессов, совместно использующих определенную область памяти. Таким образом, работа приложения подвержена ошибкам только в той мере, в какой они влияют на содержание данной области памяти. Локальное адресное пространство (private memory) процесса остается защищенным от других процессов. В модели, целиком основанной на использовании потоков, локальные данные всех потоков (в том числе их стеки) остаются полностью доступными и поэтому подвержены ошибкам из-за некорректных указателей в любом из потоков процесса.

Кроме того, потоки могут дать такие возможности параллельной обработки, которыми не обладает модель на основе процессов. Например, серверный процесс файловой системы (filesystem server process), предназначенный для обработки запросов, посылаемых множеством клиентов (при этом каждый запрос требует значительного времени для завершения), стал бы значительно более эффективным, если бы состоял из множества потоков. Если один

клиентский процесс запрашивает блок данных с диска, а другой клиент запрашивает блок данных, который уже находится в кэше, процесс файловой системы может использовать группу потоков для одновременного обслуживания клиентских запросов вместо того, чтобы оставаться "занятым" во время считывания блока данных по первому запросу.

По мере поступления запросов каждый поток может отвечать на них, обращаясь напрямую к кэшу, или блокировать их и ждать ввода/вывода данных с диска, не увеличивая время реакции (response latency), как это происходит в других клиентских процессах. Сервер файловой системы может заранее создавать ("precreate") группу потоков, готовых последовательно реагировать на клиентские запросы по мере их поступления. Хотя такая модель усложняет архитектуру администратора файловой системы, она значительно увеличивает возможности параллельной обработки.

Службы синхронизации

В ОС QNX Neutrino используются POSIX-примитивы для синхронизации на уровне потоков. Некоторые из этих примитивов могут применяться для потоков в разных процессах. К службам синхронизации относятся по крайней мере следующие объекты — табл. 2.5.

Таблица 2.5. *Службы синхронизации*

Служба синхронизации	Межзадачная поддержка	Сетевая поддержка
Мутекс	Да	Нет
Условная переменная	Да	Нет
Барьер	Нет	Нет
Ждущая блокировка	Нет	Нет
Блокировка чтения/записи	Да	Нет
Семафор	Да	Да (только для именованных)
FIFO-планирование	Да	Нет
Отправка/получение/ответ	Да	Да
Атомарная операция	Да	Нет

Замечание

Приведенные ранее примитивы синхронизации реализуются непосредственно ядром, за исключением:

- барьеров, ждущих блокировок (sleepon locks) и блокировок чтения/записи (которые основаны на мутексах и условных переменных);
- атомарных операций (которые либо реализуются непосредственно процессором, либо эмулируются ядром).

Блокировки взаимного исключения (мутексы)

Наиболее простыми из служб синхронизации являются мутексы. Мутекс (от англ. mutex — mutual exclusion lock) служит для обеспечения монопольного доступа к данным, которые совместно используются несколькими потоками. Операциями захвата мутекса (с помощью функции *pthread_mutex_lock()*) и освобождения мутекса (с помощью функции *pthread_mutex_unlock()*) обычно обрамляются участки кода, который обращается к совместно используемым данным (обычно это критическая секция кода).

В каждый момент времени мутекс может быть захвачен только одним потоком. Потоки, которые пытаются захватить уже захваченный мутекс, блокируются до тех пор, пока этот мутекс не освобождается. Когда поток освобождает мутекс, поток с наивысшим приоритетом, который ожидает возможности захватить мутекс, будет разблокирован и станет новым владельцем мутекса. Таким образом, потоки обрабатывают критическую секцию кода в порядке своих приоритетов.

В большинстве процессоров захват мутекса не требует обращения к ядру. Это достигается благодаря операции "сравнить и переставить" (compare-and-swap opcode) в семействе x86, а также посредством условных инструкций "загрузить/сохранить" на большинстве процессоров семейства RISC.

При захвате мутекса передача управления коду ядра происходит только при условии, что мутекс уже захвачен и поток может быть включен в список блокированных потоков. Управление передается коду ядра также при освобождении мутекса, если другие потоки ожидают разблокирования на этом мутексе. Это позволяет выполнять захват и освобождение критических секций с очень высокой скоростью. Таким образом, действие ОС сводится всего лишь к управлению приоритетами.

Для определения текущего состояния мутекса используется специальная неблокирующая функция *pthread_mutex_trylock()*. Для повышения производительности системы время выполнения критической секции кода должно быть небольшим и ограниченным. Если поток может блокироваться во время выполнения критической секции, то должна использоваться условная переменная.

Наследование приоритетов

Если поток, имеющий более высокий приоритет, чем владелец мутекса, пытается захватить мутекс, то действующий приоритет текущего владельца устанавливается равным приоритету блокированного потока, ожидающего мутекс. Владелец вернется к своему исходному приоритету после того, как он освободит мутекс. Такая схема не только обеспечивает минимальное время ожидания мутекса, но и решает классическую проблему инверсии приоритетов.

Функция *pthread_mutex_setrecursive()* позволяет изменять атрибуты мутекса таким образом, чтобы один и тот же поток мог выполнять рекурсивный захват мутекса. Это дает потоку возможность вызывать процедуры для выполнения повторного захвата мутекса, который к этому моменту уже захвачен данным потоком.

Замечание

Рекурсивные мутексы *не* входят в стандарты POSIX и *не* работают с условными переменными.

Условные переменные

Условная переменная (condvar — сокр. от condition variable) используется для блокировки потока по какому-либо условию во время выполнения критической секции кода. Условие может быть сколь угодно сложным и не зависит от условной переменной. Однако условная переменная всегда должна использоваться совместно с мутексом для проверки условия.

Условные переменные поддерживают следующие функции:

- ожидание условной переменной (wait) (*pthread_cond_wait()*);
- единичная разблокировка потока (signal) (*pthread_cond_signal()*);
- множественная разблокировка потока (broadcast) (*pthread_cond_broadcast()*).

Замечание

Следует иметь в виду, что единичная разблокировка потока, обозначаемая термином "signal", никак не связана с понятием сигнала в стандартах POSIX.

Приведем пример типичного использования условной переменной:

```
pthread_mutex_lock( &m );
. . .
while (!arbitrary condition) {
pthread_cond_wait( &cv, &m );
}
. . .
pthread_mutex_unlock( &m );
```


В этом примере захват мутекса происходит до проверки условия. Таким образом, проверяемое условие применяется только к текущему потоку. Пока данное условие является истинным, эта секция кода блокируется на вызове ожидания до тех пор, пока какой-либо другой поток не выполнит операцию единичной или множественной разблокировки потока по условной переменной.

Цикл `while` в приведенном примере требуется по двум причинам. Во-первых, стандарты POSIX не гарантируют отсутствие ложных пробуждений (например, в многопроцессорных системах). Во-вторых, если другой поток изменяет условие, необходимо заново выполнить его проверку, чтобы убедиться, что изменение соответствует принятым критериям. При блокировании ожидающего потока связанный с условной переменной мутекс атомарно освобождается функцией *pthread_cond_wait()* для того, чтобы другой поток мог войти в критическую секцию программного кода.

Поток, который выполняет единичную разблокировку потока, разблокирует поток с наивысшим приоритетом, который стоит в очереди на условной переменной. Операция множественной разблокировки потока разблокирует все потоки, стоящие в очереди на условной переменной. Связанный с условной переменной мутекс освобождается атомарно разблокированным потоком с наивысшим приоритетом. После обработки критической секции кода этот поток должен освободить мутекс.

Другой вид операции ожидания условной переменной (*pthread_cond_timedwait()*) позволяет установить таймаут. По окончании этого периода ожидающий поток может быть разблокирован.

Барьеры

Барьер — это механизм синхронизации, который позволяет скоординировать работу нескольких взаимодействующих потоков (например, при матричных вычислениях) таким образом, чтобы каждый из них остановился в заданной точке в ожидании остальных потоков, прежде чем продолжить свою работу.

В отличие от функции *pthread_join()*, при которой поток ожидает завершения другого потока, барьер заставляет потоки *встретиться* в определенной точке. После того как заданное количество потоков достигает установленного барьера, *все* эти потоки разблокируются и продолжат свою работу.

Барьер создается с помощью функции *pthread_barrier_init()*:

```
#include <pthread.h>
int
pthread_barrier_init (pthread_barrier_t *barrier,
const pthread_barrierattr_t *attr,
unsigned int count);
```

В результате выполнения этого кода создается барьер по заданному адресу (указатель на барьер находится в аргументе *barrier*) и с атрибутами, установленными аргументом *attr*. Аргумент *count* задает количество потоков, которые должны вызвать функцию *pthread_barrier_wait()*.

После создания барьера каждый поток вызывает функцию `pthread_barrier_wait()`, тем самым сообщая о завершения этого действия:

```
#include <pthread.h>
int pthread_barrier_wait (pthread_barrier_t *barrier);
```

Когда поток вызывает функцию *pthread_barrier_wait()*, он блокируется до тех пор, пока то число потоков, которое было задано функцией *pthread_barrier_init()*, не вызовет функцию *pthread_barrier_wait()* и, соответственно, не заблокируется. После того как заданное количество потоков вызовет функцию *pthread_barrier_wait()*, все они разблокируются *одновременно*.

Листинг 2.1

```
/*
* barrier1.c
*/
#include <stdio.h>
#include <time.h>
#include <pthread.h>
#include <sys/neutrino.h>
pthread_barrier_t barrier; // объект синхронизации типа "барьер"
main () // игнорировать аргументы
{
time_t now;
// создать барьер со значением счетчика 3

pthread_barrier_init (&barrier, NULL, 3);
// стартовать два потока - thread1 и thread2
pthread_create (NULL, NULL, thread1, NULL);
pthread_create (NULL, NULL, thread2, NULL);
// потоки thread1 и thread2 выполняются
// ожидание завершения
time (&now);
printf ("main() waiting for barrier at %s", ctime (&now));
pthread_barrier_wait (&barrier);
// после этого момента все три потока завершены
time (&now);
printf ("barrier in main() done at %s", ctime (&now));
}
void *
thread1 (void *not used)
{
time_t now;
time (&now);
printf ("thread1 starting at %s", ctime (&now));
// выполнение вычислений
// пауза
sleep (20);
pthread_barrier_wait (&barrier);
// после этого момента все три потока завершены
time (&now);
printf ("barrier in thread1() done at %s", ctime (&now));
}
void *
thread2 (void *not_used)
{
time_t now;
time (&now);
printf ("thread2 starting at %s", ctime (&now));
// выполнение вычислений
// пауза
sleep (40);
pthread_barrier_wait (&barrier);
// после этого момента все три потока завершены
time (&now);
printf ("barrier in thread2() done at %s", ctime (&now));
}
```


В примере из листинга 2.1 основной поток создает барьер, после запуска которого начинается подсчет количества потоков, заблокированных на барьере для синхронизации. В данном случае количество синхронизируемых потоков задается равным 3: поток *main()*, поток *thread1()* и поток *thread2()*.

Запускаются потоки *thread1()* и *thread2()*. Для наглядности в потоке задается пауза, чтобы имитировать процесс вычислений. Для выполнения синхронизации основной поток блокируется на барьере и ожидает разблокировки,

которая произойдет после того, как остальные два потока не присоединятся к нему на этом барьере.

В данную версию ОС QNX Neutrino включены следующие функции работы с барьерами — табл. 2.6.

Таблица 2.6. *Функции работы с барьерами*

Функция	Описание
pthread_barrierattr_getpshared()	Получить значение атрибута совместного использования для заданного барьера
pthread_barrierattr_destroy()	Уничтожить атрибутную запись барьера
pthread_barrierattr_init()	Инициализировать атрибуты объекта
pthread_barrierattr_setpshared()	Установить значение атрибута совместного использования для заданного барьера
pthread_barrier_destroy()	Уничтожить барьер
pthread_barrier_init()	Инициализировать барьер
pthread_barrier_wait()	Синхронизировать потоки на барьере

Ждущие блокировки

Ждущие блокировки (sleepon locks) работают аналогично условным переменным, за исключением некоторых деталей. Как и условные переменные, ждущие блокировки (*pthread_sleepon_lock()*) могут использоваться для блокировки потока до тех пор, пока условие не станет истинным (аналогично изменению значения ячейки памяти). Но в отличие от условных переменных (которые должны существовать для каждого проверяемого условия), ждущие блокировки применяются к одному мутексу и динамически создаваемой условной переменной независимо от количества проверяемых условий. Максимальное число условных переменных в конечном итоге равно максимальному числу блокированных потоков. Этот вид блокировок аналогичен тем, которые применяются в ядре UNIX.

Блокировки по чтению/записи

Блокировки по чтению/записи (reader/writer locks) (или более точное название "блокировки на множественное чтение и однократную запись") используются в тех случаях, когда доступ к структуре данных должен определяться по следующей схеме: чтение данных выполняет множество потоков,

запись — не более одного потока. Хотя этот вид блокировок несет больше накладных расходов, чем мутексы, он позволяет организовывать описанную схему доступа к данным.

Блокировка по чтению/записи (*pthread_rwlock_rdlock()*) предоставляет доступ по чтению всем потокам, которые его запрашивают. Однако если поток запрашивает блокировку по записи (*pthread_rwlock_wrlock()*), запрос отклоняется до тех пор, пока все потоки, выполняющие чтение, не снимут свои блокировки по чтению (*pthread_rwlock_unlock()*).

Множество потоков, выполняющих запись, выстраиваются в очередь (в порядке своих приоритетов), ожидая возможности выполнить операцию записи в защищенную структуру данных. Все блокированные потоки, выполняющие запись, запускаются до того, как читающие потоки снова получат разрешение на доступ к данным. Приоритеты читающих потоков не учитываются.

Существуют специальные вызовы, которые позволяют потоку тестировать возможность доступа к необходимой блокировке, оставаясь в активном состоянии (*pthread_rwlock_tryrdlock()* и *pthread_rwlock_trywrlock()*). Эти вызовы возвращают код завершения, сообщающий о возможности или невозможности установки блокировки.

Реализация блокировок по чтению/записи происходит не в ядре, а посредством мутексов и условных переменных, предоставляемых ядром.

Семафоры

Еще одним средством синхронизации являются семафоры (semaphores), которые позволяют потокам увеличивать (с помощью функции *sem_post()*) или уменьшать (с помощью функции *sem_wait()*) значение счетчика на семафоре для управления блокировкой потока (операции "post" и "wait" соответственно).

При вызове функции *sem_wait()* поток не блокируется, если счетчик имел положительное значение. При неположительном значении счетчика поток блокируется до тех пор, пока какой-либо другой поток не увеличит значение счетчика. Увеличение значения счетчика может выполняться несколько раз подряд, что позволяет уменьшать значение счетчика, не вызывая блокировки потоков.

Существенным отличием семафоров от других примитивов синхронизации является то, что семафоры безопасны для применения в асинхронной среде ("async safe") и могут управляться обработчиками сигналов. Семафоры как раз подходят для тех случаев, когда требуется пробудить поток с помощью обработчика сигнала.

Замечание

Как правило, мутексы работают быстрее, чем семафоры (которые всегда требуют обращения к коду ядра).

Другим полезным свойством семафоров является то, что они были определены для работы между процессами. Хотя мутексы в QNX Neutrino тоже работают между процессами, стандарты POSIX рассматривают эту возможность как дополнительную функцию, которая может оказаться не переносимой между системами. Что касается синхронизации между потоками внутри одного процесса, то здесь мутексы более эффективны, чем семафоры.

Полезной разновидностью семафоров является служба *именованных* семафоров (named semaphore service). Эта служба использует администратор ресурсов и позволяет применять семафоры между процессами, выполняемыми на разных машинах внутри сети.

Замечание

Именованные семафоры работают *медленнее*, чем неименованные.

Поскольку семафоры, как и условные переменные, могут в штатном порядке возвращать ненулевое значение из-за ложного пробуждения, для их корректной работы требуется использование цикла:

```
while (sem wait(&s) && errno == EINTR) { do_nothing(); }
do_critical_region();/* Значение семафора уменьшилось. */
```

Синхронизация с помощью алгоритма планирования

Применение алгоритма FIFO-планирования стандарта POSIX в системе без симметричной многопроцессорной обработки предотвращает выполнение критической секции кода одновременно несколькими потоками с одинаковым приоритетом. Алгоритм FIFO-планирования предписывает, что все потоки, запланированные к выполнению по этому алгоритму и имеющие одинаковый приоритет, выполняются до тех пор, пока они самостоятельно не освободят процессор для другого потока.

Такое "освобождение" процессора также может произойти в случае, когда поток блокируется в результате обращения к другому процессу за сервисом или при получении сигнала. *Поэтому критическая секция кода должна быть тщательно проработана и документирована для того, чтобы последующее обслуживание этого кода не приводило к нарушению данного условия.*

Кроме того, потоки с более высоким приоритетом в том (или любом другом) процессе все же могут вытеснять потоки, выполняемые по алгоритму FIFO-планирования. Поэтому все потоки, которые могут "столкнуться" между собой внутри критической секции кода, должны планироваться по алгоритму FIFO с *одинаковым* приоритетом. При таком условии потоки могут обращаться к этой разделяемой памяти без необходимости делать предварительный явный запрос на синхронизацию.

Внимание!

Метод монопольного доступа неприменим в многопроцессорных системах, поскольку в таких системах несколько процессоров могут одновременно исполнить код, который в однопроцессорной машине был бы запланирован на последовательное исполнение.

Синхронизация с помощью механизма обмена сообщениями

Службы обмена сообщениями (Send/Receive/Reply), используемые в ОС QNX Neutrino (см. описание далее), осуществляют неявную синхронизацию посредством блокировок. Во многих случаях они могут заменить собой другие службы синхронизации. Кроме того, службы обмена сообщениями — единственные примитивы синхронизации и межзадачного взаимодействия (кроме именованных семафоров, основанных на механизме обмена сообщениями), которые могут работать в сети.

Синхронизация с помощью атомарных операций

В некоторых случаях необходимо выполнить какую-либо небольшую операцию (например, увеличить значение переменной) *атомарно*, т. е. с гарантией того, что она не будет вытеснена другим потоком или каким-либо обработчиком прерываний (Interrupt Service Routine, ISR).

В ОС QNX Neutrino атомарные операции применяются для:

- сложения;
- вычитания;
- обнуления битов;
- установки битов;
- переключения (дополнения) битов.

Атомарные операции можно задействовать, подключив заголовочный файл <atomic.h>.

Использоваться атомарные операции могут где угодно, но особенно они полезны в следующих двух случаях:

- при взаимодействии между обработчиком прерываний (ISR) и потоком;
- при взаимодействии между двумя потоками (как при многопроцессорной, так и при однопроцессорной обработке).

Поскольку обработчик прерываний может вытеснять поток в любой точке, единственным способом защиты потока от прерываний остается *запрещение прерываний*. Поскольку запрещение прерываний неприемлемо для системы реального времени, рекомендуется применение атомарных операций, которые предусмотрены в QNX Neutrino.

В многопроцессорных системах множество потоков *выполняются одновременно*, поэтому здесь ситуация аналогичная — чтобы избежать запрещения прерываний, следует там, где это возможно, использовать атомарные операции.

Реализация служб синхронизации

В табл. 2.7 приведены различные вызовы микроядра и соответствующие POSIX-вызовы более высокого уровня.

Таблица 2.7. *Вызовы микроядра и соответствующие POSIX-вызовы*

Вызов микроядра	POSIX-вызов	Описание
SyncTypeCreate()	*pthread_mutex_init(), pthread_cond_init(), sem_init()*	Создать объект для мутекса, условной переменной или семафора
SyncDestroy()	*pthread_mutex_destroy(), pthread_cond_destroy(), sem_destroy*	Уничтожить объект синхронизации
SyncCondvarWait()	*pthread_cond_wait(), pthread_cond_timedwait()*	Блокировать поток на условной переменной
SyncCondvarSignal()	*pthread_cond_broadcast(), pthread_cond_signal()*	Пробудить потоки, блокированные на условной переменной
SyncMutexLock()	*pthread_mutex_lock(), pthread_mutex_trylock()*	Захватить мутекс
SyncMutexUnlock()	*pthread_mutex_unlock()*	Освободить мутекс
SyncSemPost()	*sem_post()*	Увеличить значение счетчика на семафоре
SyncSemWait()	*sem_wait(), sem_trywait()*	Уменьшить значение счетчика на семафоре

Межзадачное взаимодействие в ОС QNX Neutrino

Механизм межзадачного взаимодействия играет фундаментальную роль в ОС QNX Neutrino, превращая ее из встроенного ядра реального времени в полнофункциональную POSIX-совместимую операционную систему. Механизм межзадачного взаимодействия действует в качестве "клея", который соединяет между собой различные компоненты, добавляемые в микроядро для обеспечения системных служб, и объединяет их в единое целое.

Кроме основной формы межзадачного взаимодействия — обмена сообщениями, в QNX Neutrino используется и несколько других форм. В большинстве случаев, все другие формы межзадачного взаимодействия надстраиваются поверх механизма обмена сообщениями ОС QNX Neutrino. Основная стратегия здесь состоит в том, чтобы создать простую и надежную службу межзадачного взаимодействия, которую можно оптимизировать для максимальной производительности, используя минимум кода ядра. На основе этой службы потом можно создать более сложные механизмы межзадачного взаимодействия.

Тестирования, в которых сравнивались высокоуровневые службы межзадачного взаимодействия (например, неименованные и именованные программные каналы, реализованные на основе механизма обмена сообщениями QNX Neutrino) с аналогичными службами монолитного ядра, выявили приблизительно одинаковые характеристики производительности.

В ОС QNX Neutrino предусмотрены следующие формы межзадачного взаимодействия — табл. 2.8.

Таблица 2.8. *Формы межзадачного взаимодействия*

Служба	Область реализации
Обмен сообщениями	Ядро
Сигналы	Ядро
Очереди сообщений POSIX	Внешний процесс
Разделяемая память	Администратор процессов
Неименованные программные каналы (pipes)	Внешний процесс
Именованные программные каналы (FIFOs)	Внешний процесс

Указанные в таблице службы разработчик может выбирать в различных сочетаниях в зависимости от требований пропускной способности, необходимости формирования очередей, прозрачности сети и т. д. Выбирать необходимые службы можно в различных сочетаниях, что дает полезные возможности гибкости.

При разработке микроядра ОС QNX Neutrino акцент на механизм обмена сообщениями как на основополагающий примитив межзадачного взаимодействия был сделан намеренно. Будучи одной из форм межзадачного взаимодействия, механизм обмена сообщениями, реализованный посредством функций *MsgSend()*, *MsgReceive()* и *MsgReply()*, является синхронным и осуществляет копирование данных. Рассмотрим подробнее эти две характеристики.

Синхронный обмен сообщениями

Поток, который выполняет передачу сообщения (посредством функции *MsgSend()*) другому потоку (который может относиться к другому процессу), блокируется до тех пор, пока поток-получатель не выполнит прием сообщения (*MsgReceive()*) и его обработку, а также не отправит ответное сообщение (*MsgReply()*). Если поток выполняет функцию *MsgReceive()*, хотя никаких сообщений до этого ему не отправлялось, он блокируется до тех пор, пока какой-либо другой поток не выполнит функцию *MsgSend()* (рис. 2.12).

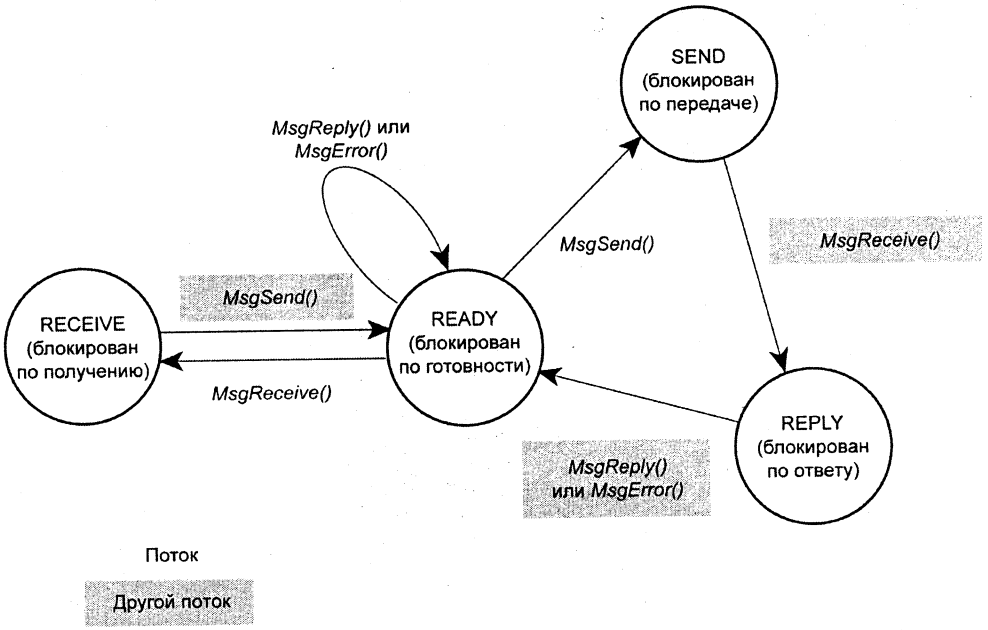


Рис. 2.12. Изменения состояния потока при выполнении транзакции Send/Receive/Reply

Такое автоматическое блокирование обеспечивает синхронизацию отправки и приема сообщений, поскольку запрос на отправку данных также вызывает блокировку потока-отправителя и планирование потока-получателя на выполнение. Это происходит без прямого вмешательства ядра с целью определения, какой поток должен быть выполнен следующим, что имело бы место в большинстве других механизмов межзадачного взаимодействия. Таким образом, очередность выполнения и данные перемещаются непосредственно из одного контекста в другой.

Формирование очередей данных не используется в этих примитивах обмена сообщениями, поскольку управление очередью может при необходимости осуществляться внутри потока-получателя. Поток-отправитель во многих

случаях уже готов к приему ответа, поэтому отпадает необходимость в управлении очередью, тем более что оно замедляет работу, даже когда формирование очереди не требуется. В результате потоку-отправителю не нужно делать непосредственный запрос на блокирование, чтобы дожидаться ответа, как это потребовалось бы при использовании какой-либо другой формы межзадачного взаимодействия.

Операции отправки и получения сообщений являются блокирующими и синхронизующими, тогда как операции отправки ответа и отправки кода ошибки (функция *MsgReply()* и *MsgError()* соответственно) — не блокирующие. Поскольку поток-клиент уже заблокирован для ожидания ответа, дополнительная синхронизация не требуется, поэтому функция *MsgReply()* не вызывает блокировки. Это позволяет серверу ответить клиенту и продолжить выполнение своего процесса в то время, как ядро и/или код, отвечающий за сетевое взаимодействие, асинхронно передает ответ потоку-отправителю и отмечает его готовность на выполнение. Большинству серверов, как правило, необходимо выполнить некоторые действия для подготовки к приему следующего запроса (и, соответственно, очередной блокировке), поэтому такой механизм полезен.

Замечание

Передача ответа по сети может проходить не так быстро, как при локальной передаче. Более подробную информацию об обмене сообщениями по сети см. в *главе 11*.

MsgReply() и *MsgError()*

Функция *MsgReply()* возвращает клиенту код завершения операции, а также ноль или более байтов. Функция *MsgError()* возвращает *только* код завершения. Обе функции снимают блокировку клиента, установленную функцией *MsgSend()*.

Копирование сообщений

Поскольку службы обмена сообщениями в ОС QNX Neutrino копируют сообщение прямо из адресного пространства одного потока в адресное пространство другого потока без помощи буфера, скорость обмена сообщениями определяется производительностью памяти в используемом оборудовании. Содержание сообщения для ядра не имеет особого смысла. Данные в сообщении имеют смысл только для отправителя и получателя. Однако существуют еще такие типы сообщений, с помощью которых могут взаимодействовать и пользовательские процессы или потоки, применяемые для модификации или замены каких-либо системных служб.

Примитивы обмена сообщениями поддерживают составную (multipart) передачу данных, поэтому буфер не обязательно должен быть сплошным. Вместо этого оба потока (поток-отправитель и поток-получатель) могут задавать таблицу-вектор (vector table), которая указывает на место расположения фрагментов сообщений. Следует иметь в виду, что размер части сообщения может быть разным для отправителя и получателя.

Составная передача позволяет пересылать сообщения, в которых заголовок отделен от тела, без затрат на копирование данных для получения сплошного сообщения. Кроме того, если исходная структура данных построена на основе кольцевого буфера, формирование трехчастного сообщения позволяет передавать заголовок и две части, размещенные в кольцевом буфере, в виде одного атомарного сообщения. В качестве аппаратного аналога такой схемы можно было бы назвать DMA-устройство с распределением/сбором данных.

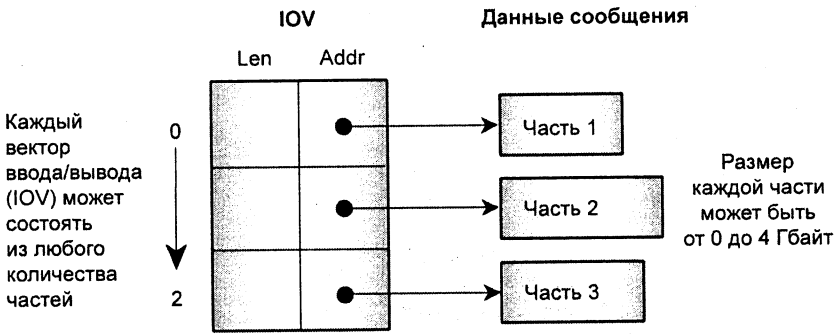


Рис. 2.13. Составная передача

Составная передача (рис. 2.13) также часто используется в файловых системах. При чтении данных выполняется их копирование напрямую из кэша файловой системы в приложение с помощью сообщения, состоящего из *n* частей. Каждая часть ссылается на кэш, и таким образом решается проблема, связанная с тем, что блоки кэша располагаются в памяти не последовательно, а точка начала или окончания чтения данных может находиться внутри блока.

Например, если размер блока кэша равен 512 байтам, чтение данных размером 1454 байт может быть выполнено посредством 5-частного сообщения (рис. 2.14).

Поскольку данные сообщения копируются из одного адресного пространства в другое напрямую, а не с помощью операций с таблицей страниц, сообщения можно легко формировать в стеке, а не брать память из специального пула памяти со страничным выравниванием для "переключения страниц" (MMU "page flipping"). В результате в исполнения многих библиотечных подпрограмм, предназначенных для реализации программного интерфейса между процессом-клиентом и процессом-сервером, можно обойтись без сложных вызовов распределения памяти, специфичных для механизмов межзадачного взаимодействия.

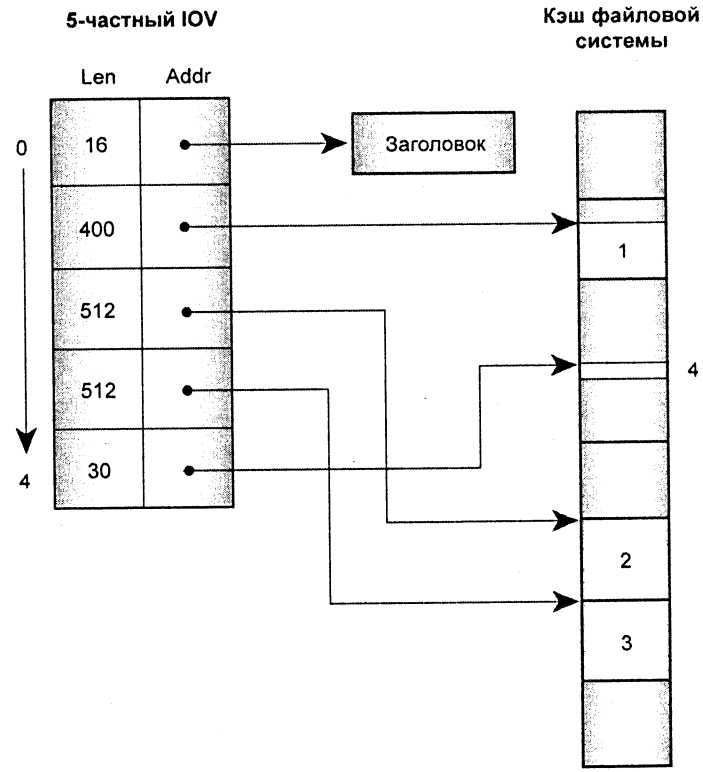


Рис. 2.14. Распределение/сбор данных при чтении блока данных размером 1454 байт

Например, с помощью следующего кода поток-клиент может сделать запрос к администратору файловой системы для выполнения `lseek`:

```
#include <unistd.h>
#include <errno.h>
#include <sys/iomsg.h>
off64_t lseek64(int fd, off64_t offset, int whence) {
io_lseek_t msg;
off64_t off;
msg.i.type = _IO_LSEEK;
msg.i.combine_len = sizeof msg.i;
```

```
msg.i.offset = offset;
msg.i.whence = whence;
msg.i.zero = 0;
if(MsgSend(fd, &msg.i, sizeof msg.i, &off, sizeof off) == -1) {
return -1;
}
return off;
}
off64_t tell64(int fd) {
return lseek64(fd, 0, SEEK_CUR);
}
off_t lseek(int fd, off_t offset, int whence) {
return lseek64(fd, offset, whence);
}
off_t tell(int fd) {
return lseek64(fd, 0, SEEK_CUR);
}
```

В результате выполнения этого программного кода, в стеке строится структура сообщения, которая затем заполняется различными константами и параметрами, переданными от вызывающего потока, и пересылается администратору файловой системы, связанному с файловым дескриптором (*fd*). Ответ сообщает об успешном или неуспешном выполнении операции.

Замечание

Тем не менее, описанная ранее схема не исключает для ядра возможности передавать большие сообщения с помощью "переключения страниц". Поскольку большинство сообщений имеют небольшой размер, копирование этих сообщений выполняется быстрее, чем управление таблицами страниц в страничном диспетчере памяти. Для передачи больших объемов данных также может применяться разделяемая память между процессами, при этом для пересылки уведомлений могут служить примитивы передачи сообщений или другие примитивы синхронизации.

Простые сообщения

Для передачи простых 1-частных сообщений в ОС QNX Neutrino используются функции, которым передается указатель непосредственно на буфер без помощи вектора ввода/вывода (IOV). В этом случае количество частей заменяется на размер сообщения. Например, в примитиве *передачи сообщений* (для которого требуется буфер передачи и буфер ответа), используются следующие четыре функции — табл. 2.9.

Таблица 2.9. *Функции примитива передачи сообщений*

Функция	Передача	Прием
MsgSend()	Указатель на буфер	Указатель на буфер
MsgSendsv()	Указатель на буфер	IOV
MsgSendvs()	IOV	Указатель на буфер
MsgSendv()	IOV	IOV

В названиях других примитивов передачи сообщений, которые принимают указатель на буфер, последняя буква "v" опускается — табл. 2.10.

Таблица 2.10. *Примитивы передачи сообщений, принимающие указатель на буфер*

IOV	Указатель на буфер
MsgReceivev()	*MsgReceive()*
MsgReceivePulsev()	*MsgReceivePulse()*
MsgReplyv()	*MsgReply()*
MsgReadv()	*MsgRead()*
MsgWritev()	*MsgWrite()*

Каналы и соединения

В ОС QNX Neutrino сообщения передаются в направлении каналов (channel) и соединений (connection), а не напрямую от потока к потоку. Поток, которому необходимо получить сообщение, сначала создает канал, а другой поток, которому необходимо передать сообщение этому потоку, должен сначала установить соединение, "подключившись" к этому каналу.

Каналы требуются для вызовов микроядра, предназначенных для обмена сообщениями (message kernel calls), и используются серверами для приема сообщений с помощью функции *MsgReceive()*. Соединения создаются потоками-клиентами для того, чтобы "присоединиться" к каналам, открытым серверами. После того как соединения установлены, клиенты могут передавать по ним сообщения с помощью функции *MsgSend()*. Если несколько потоков процесса подключается к одному и тому же каналу, тогда для повышения эффективности все эти соединения отображаются в один объект ядра. Каналы и соединения, созданные процессом, обозначаются небольшими целочисленными идентификаторами. Клиентские соединения отображаются непосредственно в дескрипторы файлов (рис. 2.15).

С точки зрения архитектуры это имеет ключевое значение. Благодаря тому, что клиентские соединения отображаются непосредственно в дескриптор файла, на одну операцию преобразования становится меньше. Отпадает необходимость "выяснять" из дескриптора файла (например, с помощью вызова `read(fd)`), куда именно сообщение должно быть передано. Вместо этого сообщение передается непосредственно в "дескриптор файла" (т. е. идентификатор соединения).

Приведем функции для каналов и соединений (табл. 2.11).

Таблица 2.11. *Функции для каналов и соединений*

Функция	Описание
ChannelCreate()	Создать канал для получения сообщений
ChannelDestroy()	Уничтожить канал
ConnectAttach()	Создать соединение для передачи сообщений
ConnectDetach()	Закрыть соединение

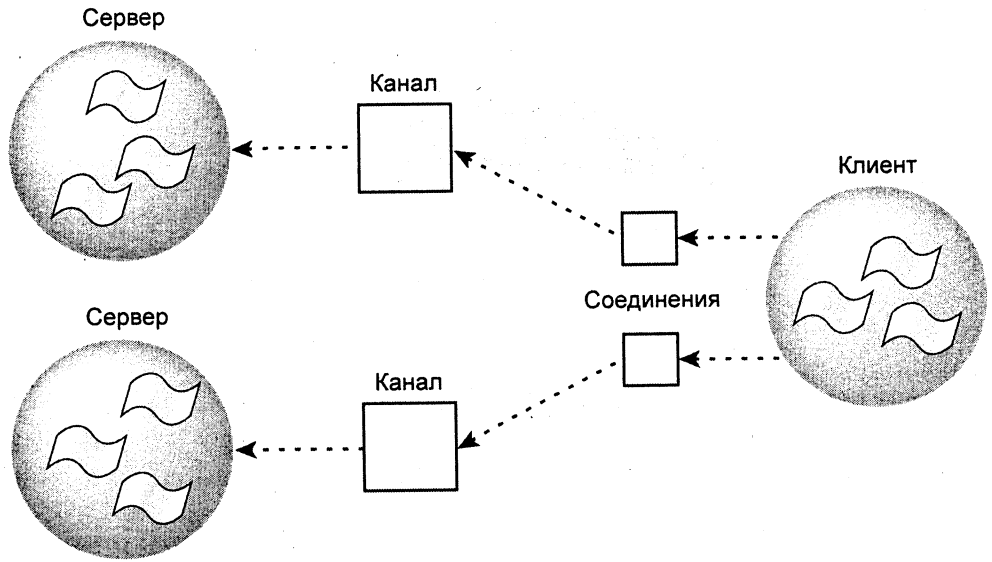


Рис. 2.15. Соединения отображаются в дескрипторы файлов

Процесс, выполняемый как сервер, может содержать событийный цикл для получения и обработки сообщений. Например:

```
chid = ChannelCreate(flags);
SETIOV(&iov, &msg, sizeof(msg));
```



```
for(;;) {
rcv_id = MsgReceivev( chid, &iov, parts, &info );
switch( msg.type ) {
/* Обработка сообщения. */
}
MsgReplyv( rcv_d, &iov, rparts );
}
```

Этот цикл позволяет потоку получать сообщения от любого другого потока, установившего соединение с данным каналом.

С этим каналом связаны три очереди:

- очередь потоков, ожидающих сообщения;
- очередь потоков, которые отправили сообщения, но эти сообщения еще не получены;
- очередь потоков, которые отправили сообщения, и эти сообщения были получены, но ответы на них еще не отправлены.

Ожидающий поток блокируется, если оказывается в любой из этих очередей (т. е. блокируется в состоянии RECEIVE, SEND или REPLY) — рис. 2.16.

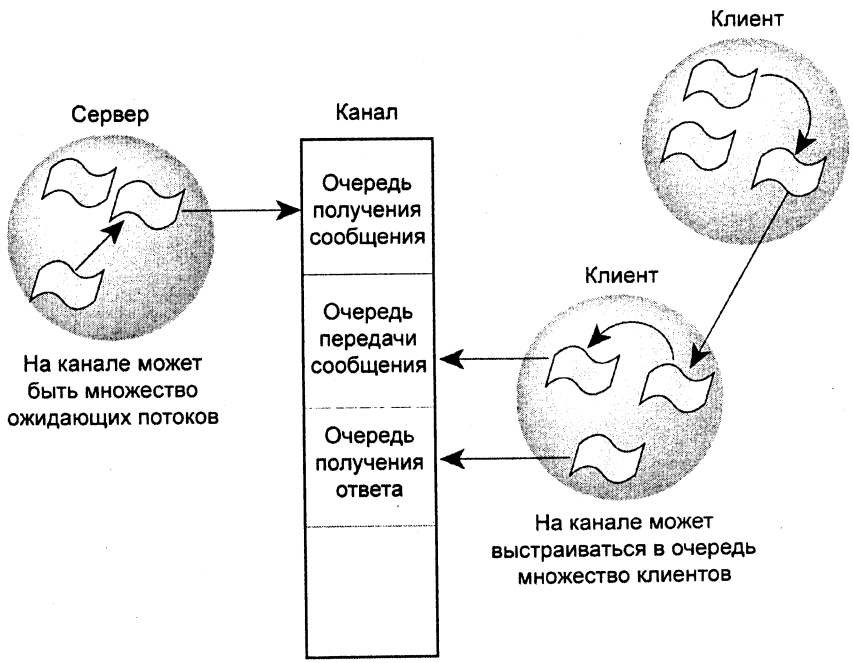


Рис. 2.16. Блокирование потоков, находящихся в очереди на канале

Импульсы

Кроме служб синхронизации Send/Receive/Reply, в ОС QNX Neutrino используются неблокирующие сообщения фиксированного размера. Эти сообщения называются *импульсами (pulses)* и имеют небольшую длину (4 байта данных и 1 байт кода).

Таким образом, импульсы несут в себе всего лишь 8 битов кода и 32 бита данных (рис. 2.17). Этот вид сообщений часто используется в качестве механизма уведомления внутри обработчиков прерываний. Импульсы также позволяют серверам передавать сообщения о событиях, не блокируясь.

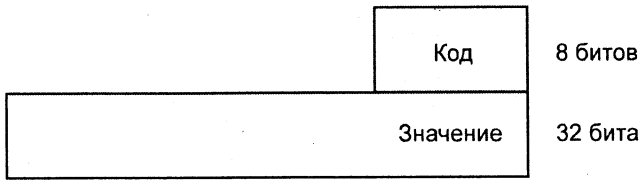


Рис. 2.17. Импульсы имеют небольшой размер

Наследование приоритетов

Серверный процесс получает сообщения в порядке приоритетов. После получения запроса потоки внутри сервера наследуют приоритет потока-отправителя (но не алгоритм планирования). В результате относительные приоритеты потоков, осуществляющих запрос к серверу, сохраняются прежними, а сервер работает с соответствующим приоритетом. Таким образом, механизм наследования приоритетов на основе передачи сообщений позволяет избежать проблемы инверсии приоритетов.

Программный интерфейс механизма обмена сообщениями

Программный интерфейс (API) механизма обмена сообщениями состоит из следующих функций — табл. 2.12.

Таблица 2.12. *Функции программного интерфейса механизма обмена сообщениями*

Функция	Описание
MsgSend()	Отправить сообщение и блокировать поток до получения ответа
MsgReceive()	Ожидать сообщения
MsgReceivePulse()	Ожидать короткого блокирующего сообщения (импульса)

Таблица 2.12 *(окончание)*

Функция	Описание
MsgReply()	Ответить на сообщение
MsgError()	Переслать ответ, содержащий статус потока. Никакая другая информация, за исключением статуса, не передается
MsgRead()	Прочитать дополнительные данные из полученного сообщения
MsgWrite()	Записать дополнительные данные в ответное сообщение
MsgInfo()	Получить информацию о полученном сообщении
MsgSendPulse()	Передать короткое неблокирующее сообщение (импульс)
MsgDeliverEvent()	Передать событие клиенту
MsgKeyData()	Снабдить сообщение ключом безопасности

Отказоустойчивая архитектура на основе механизма Send/Receive/Reply

Для управления приложениями, архитектура которых в QNX Neutrino строится как набор потоков и процессов, взаимодействующих посредством механизма Send/Receive/Reply, используются *синхронные уведомления*. Таким образом, межзадачное взаимодействие (IPC) осуществляется в системе в строго определенные моменты, а не асинхронно.

Одна из главных проблем асинхронных систем заключается в том, что для уведомления о событиях требуются обработчики сигналов. Асинхронное межзадачное взаимодействие может затруднить тщательное тестирование работы системы и не дать гарантии, что, когда бы ни был вызван обработчик сигнала, работа продолжится, как планировалось. Во многих случаях приложения не предусматривают такого режима работы и используют вместо него специальное "временное окно", которое явно открывается и закрывается на какой-то период, в течение которого может происходить обмен сигналами.

Благодаря системной архитектуре без очередей и с синхронным межзадачным взаимодействием, построенным на примитивах обмена сообщениями Send/Receive/Reply, приложения могут иметь надежную и простую архитектуру.

Другой сложной проблемой в приложениях, построенных на основе межзадачного взаимодействия, механизма организации очередей, разделяемой памяти и различных примитивов синхронизации, являются ситуации взаимной блокировки (deadlock). Например, допустим поток A освобождает мутекс 1 только после того, как поток B освобождает мутекс 2. К сожалению, если поток B находится в состоянии, при котором он может освободить мутекс 2 только после того, как поток A освободит мутекс 1, возникает состояние необратимой блокировки (standoff). Для выявления ситуации взаимной блокировки часто используются инструменты моделирования.

Примитивы обмена сообщениями (Send/Receive/Reply), которые существуют в механизме межзадачного взаимодействия, позволяют создавать системы, в которых ситуации взаимной блокировки исключены благодаря соблюдению следующих простых правил:

1. Никакие два потока не должны передавать сообщения одновременно друг другу.
2. Потоки должны быть организованы иерархически, причем сообщения должны передаваться только вверх по дереву иерархии.

Как видно, первое правило предназначено для исключения состояний необратимой блокировки. Что касается второго правила, то оно требует некоторых объяснений. Группа взаимодействующих потоков и процессов организуется так, как показано на рис. 2.18.

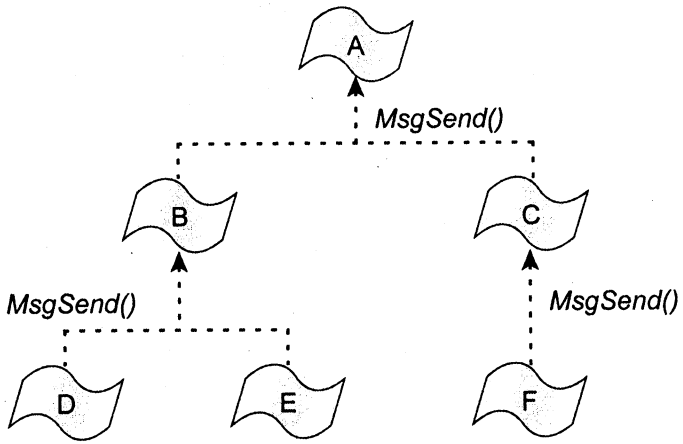


Рис. 2.18. Потоки всегда должны пересылать сообщения только вышестоящим потокам

Как видно из рис. 2.18, на любом уровне иерархии потоки не могут пересылать сообщения друг другу, но только "вверх", на более высокий уровень.

Например, клиентское приложение передает сообщение серверному процессу базы данных, который в свою очередь передает сообщение процессу файловой системы. Поскольку поток-отправитель блокируется в ожидании ответа потока-получателя, а поток-получатель не блокируется по передаче от потока-отправителя, ситуация взаимной блокировки не может произойти.

Каким же образом вышестоящий поток может уведомить нижестоящий поток о том, что он содержит результаты выполнения ранее запрошенной операции? (Допустим, что нижестоящий поток не ожидает получения результатов после передачи своего последнего сообщения.)

ОС QNX Neutrino обеспечивает очень гибкую архитектуру, в которой предусмотрен вызов ядра *MsgDeliverEvent()*, предназначенный для передачи неблокирующих событий. Все известные асинхронные службы могут реализовываться с использованием этой функции.

Например, серверная часть вызова *select()* представляет собой программный интерфейс, с помощью которого приложение может заставить поток ожидать завершения операции ввода/вывода на наборе файловых дескрипторов. Кроме асинхронного механизма уведомления, который служит в качестве "обратного канала", используемого вышестоящими потоками для передачи сообщений нижестоящим потокам, мы можем построить надежную систему уведомления для таймеров, аппаратных прерываний и других аналогичных источников событий (рис. 2.19).

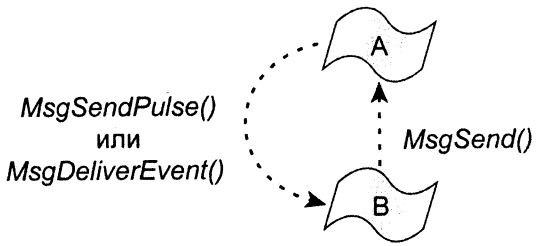


Рис. 2.19. Вышестоящий поток может "отправить" импульсное событие

Другая проблема касается того, каким образом вышестоящий поток может запросить выполнение нижестоящего потока без передачи ему сообщения, которое может привести к ситуации взаимной блокировки. Для вышестоящего потока нижестоящий поток служит только в качестве подчиненного, т. е. выполняет работу по запросу вышестоящего потока. Нижестоящий поток передает сообщение вышестоящему потоку для того, чтобы "уведомить его о своей работе", но вышестоящий поток не ответит на это уведомление до тех пор, пока не выполнит своей работы. В его ответе (передача которого является неблокирующей операцией) будут

содержаться данные, описывающие выполненную им работу. Таким образом, именно ответ служит в качестве инициатора работы, а не передача начального сообщения, в результате чего обеспечивается соблюдение вышеописанного правила 1.

События

Важным новшеством в архитектуре микроядра ОС QNX Neutrino является подсистема обработки событий. Стандарты POSIX и их расширения реального времени определяют несколько асинхронных методов уведомления — например, UNIX-сигналы (не выстраивают данные в очередь и не пересылают их), POSIX-сигналы реального времени (могут выстраивать данные в очередь и пересылать их) и т. д.

Кроме того, ядро ОС QNX Neutrino содержит такие особые механизмы уведомления, как импульсы. Реализация всех этих механизмов обработки событий могла бы потребовать значительного объема кода, поэтому стратегия проектирования ОС QNX Neutrino заключалась в том, чтобы построить все эти методы уведомления в виде одной комплексной подсистемы обработки событий.

Преимуществом такого подхода является то, что возможности одних механизмов уведомления становятся доступными для других механизмов. Например, одни и те же службы организации очередей могут применяться как для сигналов реального времени POSIX, так и для UNIX-сигналов, что позволяет упростить реализацию обработчиков сигналов в приложении.

Существует три источника событий для выполняемого потока:

- вызов микроядра *MsgDeliverEvent()*;
- обработчик сигналов;
- истечение срока действия таймера.

События бывают самых различных типов: импульсы QNX Neutrino, прерывания, различные формы сигналов, события принудительной "разблокировки". "Разблокировка" является средством, с помощью которого поток может быть освобожден от принудительной блокировки без передачи какого-либо явного события.

Из-за такого разнообразия типов событий, а также необходимости того, чтобы приложение выбирало наиболее подходящий асинхронный метод уведомления, было бы нерационально переложить исполнение программного кода, реализующего все эти функции, на серверные процессы (т. е. вышестоящие потоки, о которых шла речь в предыдущем подразделе).

Вместо этого клиентский поток может передать серверу некоторую структуру данных (или т. н. "жетон" — cookie) для последующего взаимодействия (рис. 2.20). Когда серверу необходимо послать уведомление клиентскому потоку, он производит вызов *MsgDeliverEvent()*, после чего микроядро устанавливает на данный клиентский поток указанный в "жетоне" тип события.

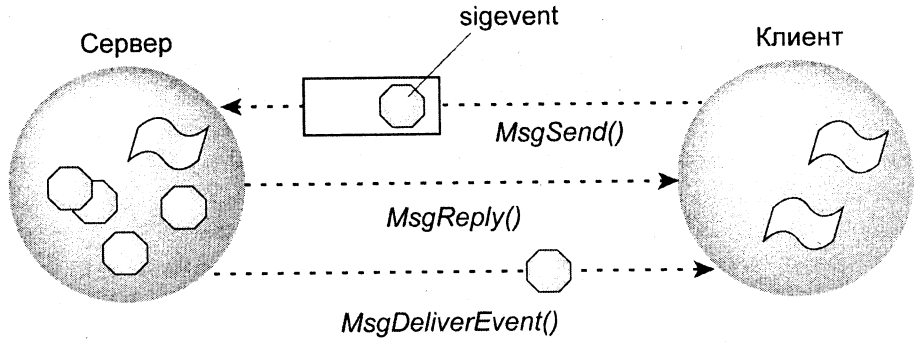


Рис. 2.20. Клиент передает `sigevent` серверу

Уведомления ввода/вывода

Функция *ionotify()* — средство, с помощью которого клиентский поток может запрашивать асинхронную передачу события. На основе этой функции реализуются многие асинхронные службы POSIX (например, *mq_notify()* и клиентская часть функции *select()*). При выполнении операции ввода/вывода на файловый дескриптор (*fd*) поток может ожидать завершения события ввода/вывода (для функции *write()*) или приема данных (для функции *read()*). Вместо блокировки потока на процессе администратора ресурсов, обслуживающем запрос чтения/записи, функция *ionotify()* позволяет клиентскому потоку сообщить администратору ресурсов о том, какого именно события он ожидает при возникновении заданных условий ввода/вывода. Такое ожидание позволяет потоку реагировать на другие источники событий, кроме заданного запроса ввода/вывода, при этом продолжая свое выполнение.

Вызов *select()* реализуется с помощью уведомления ввода/вывода и позволяет потоку блокироваться в ожидании группы событий ввода/вывода на множестве файловых дескрипторов, сохраняя возможность реагировать на другие формы межзадачного взаимодействия.

Далее приводится список условий, при которых запрошенное событие может быть доставлено:

- _NOTIFY_COND_OUTPUT — в буфере вывода имеется свободное пространство;

- _NOTIFY_COND_INPUT — объем данных, указанный администратором ресурсов, доступен для чтения;
- _NOTIFY_OUT_OF_BAND — имеются данные "вне полосы пропускания". Объем данных задан администратором ресурсов.

Сигналы

ОС QNX Neutrino поддерживает как 32 стандартных сигнала стандарта POSIX (аналогичные UNIX-сигналам), так и сигналы реального времени стандарта POSIX. Нумерация обоих наборов сигналов организована на основе набора из 64 однотипных сигналов, реализованных в ядре. Хотя сигналы реального времени в стандарте POSIX отличаются от UNIX-сигналов (во-первых, тем, что они могут содержать 4 байта данных и 1 байт кода, и, во-вторых, тем, что их можно ставить в очередь на передачу), каждый из них можно использовать по отдельности, а комбинированный набор сигналов всегда будет соответствовать стандарту POSIX.

Отметим, что по запросу приложения UNIX-сигналы могут использовать механизмы постановки в очередь для сигналов реального времени стандарта POSIX. Кроме того, ОС QNX Neutrino расширяет механизмы передачи сигналов стандарта POSIX благодаря тому, что позволяет направлять сигналы отдельным потокам, а не всему процессу, содержащему их. Поскольку сигналы — это асинхронные события, они также реализуются посредством механизмов передачи событий (табл. 2.13).

Таблица 2.13. *Вызовы микроядра и соответствующие POSIX-вызовы*

Вызов микроядра	POSIX-вызов	Описание
SignalKill()	*kill(), pthread_kill(), raise(), sigqueue()*	Установить сигнал на группе процессов, процессе или потоке
SignalAction()	*sigaction()*	Определить действие, выполняемое при получении сигнала
SignalProcmask()	*sigprocmask(), pthread_sigmask()*	Изменить маску сигналов потока
SignalSuspend()	*sigsuspend(), pause()*	Блокироваться до тех пор, пока сигнал не вызовет обработчик сигнала
SignalWaitinfo()	*sigwaitinfo()*	Ожидать сигнала. После получения сигнала вернуть информацию о нем

Изначальная спецификация стандарта POSIX предусматривала применение сигналов только для процессов. Для многопоточных процессов устанавливаются определенные правила.

- Операции с сигналами выполняются на уровне процессов. Если поток игнорирует или перехватывает сигнал, то он применяется ко *всем* потокам внутри процесса.
- Маска сигналов применяется на уровне потоков. Если поток маскирует сигнал, то это маскирование относится только к этому потоку.
- Непроигнорированный (un-ignored) сигнал, направленный некоторому потоку, передается только этому потоку.
- Непроигнорированный сигнал, направленный некоторому процессу, передается первому потоку, в котором этот сигнал немаскирован. Если все потоки маскировали этот сигнал, он будет поставлен в очередь на этот процесс до тех пор, пока какой-либо поток не игнорирует или демаскирует его. Если сигнал проигнорирован, он снимается с процесса. Если сигнал демаскирован, он перемещается с процесса на поток, который демаскировал его.

Когда сигнал направляется некоторому процессу, содержащему большое количество потоков, должно быть выполнено сканирование таблицы потоков для поиска потока, у которого данный сигнал не маскирован. Для большинства многопоточных процессов стандартной практикой является маскирование сигналов по всем потокам за исключением одного, который предназначен для их обработки. Для повышения эффективности передачи сигналов процессам, ядро выполняет кеширование последнего потока, принявшего сигнал, и впоследствии всегда передает сигнал этому потоку в первую очередь (рис. 2.21).

В стандарте POSIX существует принцип очередности сигналов реального времени. В ОС QNX Neutrino любые сигналы, не только сигналы реального времени, могут ставиться в очередь. Внутри процесса постановка в очередь может выполняться по отдельным сигналам. С каждым сигналом может быть связано 8 битов кода и 32-битовое значение.

В этом отношении сигналы очень похожи на импульсы, описанные ранее. В ядре это сходство используется оптимальным образом посредством применения общего кода для управления как сигналами, так и импульсами. Номер сигнала преобразуется в приоритет импульса с помощью _SIGMAX – *signo*. В результате сигналы передаются в порядке своих приоритетов — чем *меньше* номер сигнала, тем *выше* приоритет. Такой метод соответствует стандарту POSIX, который утверждает, что существующие сигналы имеют более высокий приоритет, чем новые сигналы реального времени.

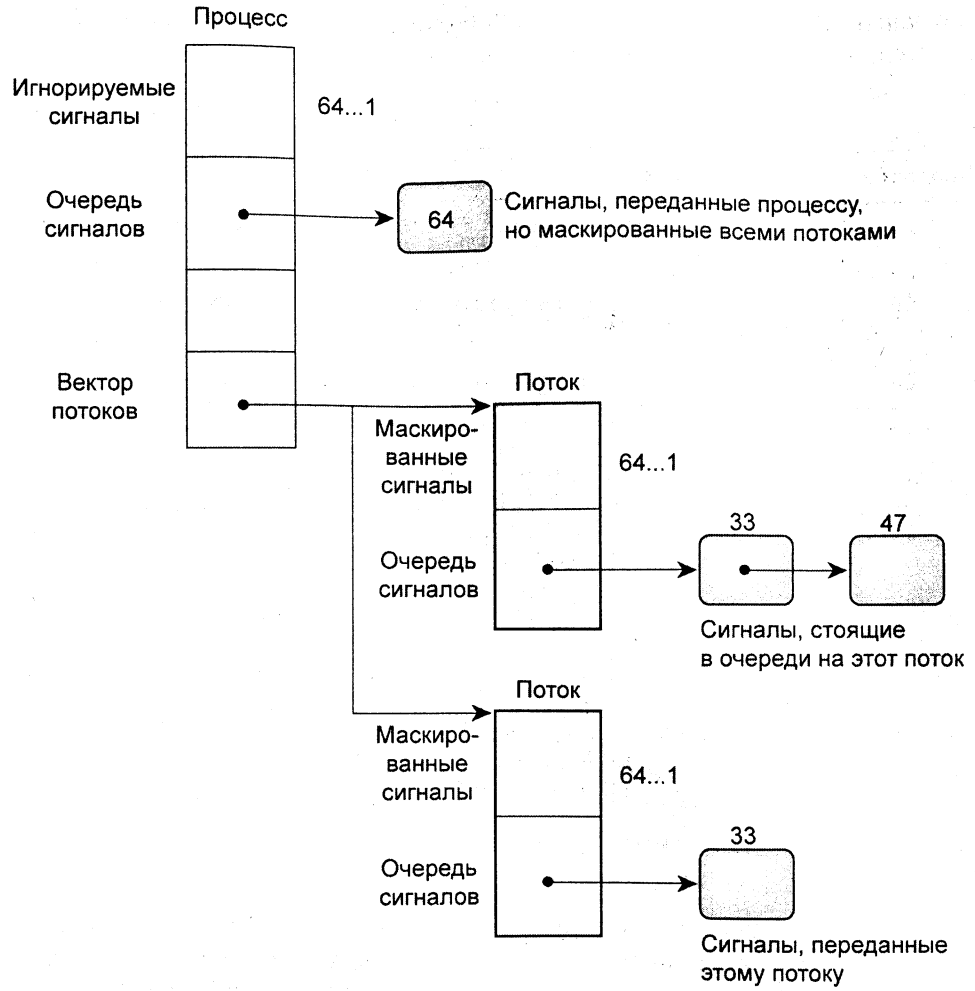


Рис. 2.21. Передача сигналов

Специальные сигналы

Как было сказано ранее, в ОС QNX Neutrino определено 64 сигнала, которые распределяются в следующих диапазонах — табл. 2.14.

Указанные восемь специальных сигналов не могут быть игнорированы или перехвачены. Попытка вызвать функцию *signal()* или *sigaction()* или выполнить вызов ядра *SignalAction()* для того, чтобы изменить эти специальные сигналы, будет приводить к ошибке EINVAL.

Кроме того, данные сигналы всегда маскированы и для них включена организация очередей (queuing). Попытка демаскировать эти сигналы

с помощью функции *sigprocmask()* или вызова ядра *SignalProcmask()* игнорируется.

Таблица 2.14. *Диапазоны сигналов*

Диапазон	Описание
1 ... 57	57 сигналов стандарта POSIX (включая традиционные UNIX-сигналы)
41 ... 56	16 сигналов реального времени стандарта POSIX (от SIGRTMIN до SIGRTMAX)
57 ... 64	8 специальных сигналов ОС QNX Neutrino

Обычный сигнал может быть сделан специальным с помощью следующих стандартных вызовов сигналов. Специальные сигналы избавляют программиста от необходимости писать для этого код и защищать сигнал от случайных изменений.

```
sigset_t *set;
struct sigaction action;
sigemptyset(&set);
sigaddset(&set, signo);
sigprocmask(SIG_BLOCK, &set, NULL);
action.sa_handler = SIG_DFL;
action.sa_flags = SA_SIGINFO;
sigaction(signo, &action, NULL);
```

Такая конфигурация делает специальные сигналы подходящими для синхронного уведомления посредством функции *sigwaitinfo()* или вызова ядра *SignalWaitinfo()*. Следующий пример кода блокируется до получения специального восьмого сигнала:

```
sigset_t *set;
siginfo_t info;
sigemptyset(&set);
sigaddset(&set, SIGRTMAX + 8);
sigwaitinfo(&set, &info);
printf("Received signal %d with code %d and value %d\n",
info.si_signo,
info.si_code,
info.si_value.sival_int);
```

Поскольку специальные сигналы всегда маскируются, программа не может быть прервана или остановлена в тех случаях, когда специальный сигнал получен вне тела функции *sigwaitinfo()*. Поскольку очередность сигналов работает всегда, сигналы не теряются — они выстраиваются в очередь для следующего вызова *sigwaitinfo()*.

Специальные сигналы были разработаны для решения общей ситуации межзадачного взаимодействия, когда серверу необходимо уведомить клиента о том, что он может передать ему данные. Для уведомления клиента сервер использует вызов *MsgDeliverEvent()*. Для передачи этого события внутри уведомления предусмотрено два объекта: импульсы и сигналы.

Импульсы больше подходят для клиентов, которые тоже могут быть серверами для других клиентов. В этом случае клиент создает канал для приема сообщений, который позволяет получить импульсы.

Для большинства простых клиентов дело обстоит иначе. Чтобы получить импульс, простому клиенту придется создать канал специально для этой цели. Если сигнал синхронный (т. е. маскирован) и для него включен механизм очередности, то он может использоваться в качестве импульса. Таким образом, клиент произведет замену вызова *MsgReceive()*, используемого для ожидания импульса на канале, на простой вызов *sigwaitinfo()*, служащий для ожидания сигнала.

Этот механизм обработки сигналов используется в графической оболочке Photon для ожидания событий. Кроме того, он применяется функцией *select()* для ожидания операций ввода/вывода, вызываемых множеством серверов. Из восьми специальных сигналов первые два имеют специальные имена.

```
#define SIGSELECT (SIGRTMAX + 1)
#define SIGPHOTON (SIGRTMAX + 2)
```

Краткое описание сигналов

Приведем краткое описание сигналов.

- SIGABRT — сигнал аварийного завершения (например, при вызове функции *abort()*).
- SIGALRM — сигнал таймаута (например, при вызове функции *alarm()*).
- SIGBUS — сообщает об ошибке контроля по четности при обращении к памяти (только для ОС QNX). Если эта ошибка возникла повторно во время того, как процесс уже находится в обработчике сигналов из-за аналогичной ошибки, то процесс завершается.
- SIGCHLD — сигнал о завершении порожденного процесса. По умолчанию игнорируется.

- SIGCONT — продолжить выполнение, если процесс находится в состоянии HELD (Задержан). По умолчанию игнорируется, если процесс не находится в состоянии HELD (Задержан).
- SIGDEADLK — сигнал о взаимной блокировке на мутексе. Если не было вызова функции *SyncMutexEvent()* и если возникли условия, при которых ядро должно передать данное событие, тогда ядро передает сигнал SIGDEADLK.
- SIGEMT — машинная команда EMT.
- SIGFPE — некорректная арифметическая операция (с целочисленными или вещественными числами), например, операция деления на ноль или операция, приводящая к переполнению. Если эта ошибка возникла повторно во время того, как процесс уже находится в обработчике сигналов из-за аналогичной ошибки, то процесс завершается.
- SIGHUP — завершение лидера сессии или разрыв связи с терминалом.
- SIGILL — некорректная команда. Если эта ошибка возникла повторно во время того, как процесс уже находится в обработчике сигналов из-за аналогичной ошибки, то процесс завершается.
- SIGINT — интерактивный предупредительный сигнал (`Break`).
- SIGIOT — машинная команда IOT (на платформе x86 не генерируется).
- SIGKILL — сигнал завершения. Должен использоваться только в аварийных ситуациях. *Этот сигнал не может быть перехвачен или игнорирован.*
- SIGPIPE — попытка записи в программный канал без зарегистрированных читателей.
- SIGPOLL — событие, требующее программного опроса (pollable event).
- SIGQUIT — интерактивный сигнал завершения.
- SIGSEGV — некорректное обращение к памяти. Если эта ошибка возникла повторно во время того, как процесс уже находится в обработчике сигналов из-за аналогичной ошибки, то процесс завершается.
- SIGSTOP — остановка процесса (по умолчанию). *Этот сигнал не может быть перехвачен или проигнорирован.*
- SIGSYS — некорректный параметр системного вызова.
- SIGTERM — сигнал завершения.
- SIGTRAP — неподдерживаемое программное прерывание.
- SIGTSTP — сигнал остановки, генерированный с клавиатуры.
- SIGTTIN — попытка чтения с управляющего терминала фоновым процессом.
- SIGTTOU — попытка записи на управляющий терминал фоновым процессом.

- SIGURG — исключительная ситуация на сокете.
- SIGUSR1 — зарезервирован как определяемый приложением сигнал 1.
- SIGUSR2 — зарезервирован как определяемый приложением сигнал 2.
- SIGWINCH — изменение размера окна.

Очереди сообщений в стандарте POSIX

В стандарте POSIX предусмотрен набор неблокирующих механизмов обмена сообщениями в виде очередей. Так же, как и программные каналы, очереди сообщений являются именованными объектами, которые взаимодействуют с читателями ("readers") и писателями ("writers")[1]. Очередь сообщений действует на основе приоритетов, присваиваемых каждому сообщению, и поэтому имеет более сложную структуру, чем каналы, тем самым позволяя приложениям более гибко управлять взаимодействием.

Замечание

Для работы с очередями сообщений стандарта POSIX в ОС QNX Neutrino должен быть запущен администратор ресурсов очередей `mqueue` (message queue resource manager).

В ОС QNX Neutrino очереди сообщений стандарта POSIX реализуются посредством *необязательного* администратора ресурсов `mqueue`. (Более подробные сведения см. в *главе 7*.)

В отличие от примитивов обмена сообщениями, используемых в QNX Neutrino, очереди сообщений стандарта POSIX располагаются *вне* ядра.

Преимущества очередей сообщений стандарта POSIX

Очереди сообщений стандарта POSIX обеспечивают интерфейс, который знаком многим разработчикам приложений реального времени, поскольку они аналогичны т. н. "почтовым ящикам" ("mailboxes"), используемым во многих исполняемых модулях реального времени.

Однако есть одно существенное различие между сообщениями, используемыми в QNX Neutrino, и очередями сообщений стандарта POSIX. Сообщения

[1] Здесь и далее в этом параграфе имеются в виду *именованные* программные каналы, обозначаемые в англоязычной терминологии терминами "FIFO" и "named pipe". Термин же "pipe", приведенный в данном предложении в оригинале книги и переведенный здесь как просто "программный канал", обычно используется для обозначения *неименованных* программных каналов. Подробнее о программных каналах можно прочитать в литературе по UNIX. — *Прим. ред.*

в QNX Neutrino являются блокирующими, поскольку они напрямую копируют данные из адресного пространства одного процесса в адресное пространство другого процесса. Очереди сообщений стандарта POSIX, наоборот, действуют по методу передачи с промежуточным хранением, благодаря которому отправитель сообщений может не блокироваться, а выстраивать свои сообщения в очередь. Очереди сообщений стандарта POSIX существуют независимо от процессов, использующих их. В результате именованные очереди сообщений могут использоваться одновременно множеством процессов.

С точки зрения производительности, очереди сообщений стандарта POSIX работают *медленнее*, чем сообщения, используемые в QNX Neutrino для передачи данных. Однако гибкость механизма очередности сообщений может вполне оправдывать такое снижение производительности.

Интерфейс, аналогичный файлам

Очереди сообщений похожи на файлы, по крайней мере с точки зрения интерфейса. Очередь сообщений открывается с помощью функции *mq_open()*, закрывается с помощью функции *mq_close()*, а уничтожается — *mq_unlink()*. Для того чтобы записать ("write") данные в очередь сообщений или прочитать ("read") данные из нее, используются функции *mq_send()* и *mq_receive()* соответственно.

Для строгого соответствия стандарту POSIX имена очередей сообщений должны начинаться с символа косой черты (/) и больше не содержать других косых черт. Однако в QNX Neutrino данное ограничение снято, и имена путей могут содержать множество косых черт. Это расширяет возможности стандарта POSIX — например, компания может размещать все очереди сообщений под своим именем и таким образом быть более уверенной в том, что имена ее очередей *не* будут конфликтовать с именами очередей других компаний.

В ОС QNX Neutrino все создаваемые очереди сообщений размещаются в пространстве файловых имен в каталоге /dev/mqueue (табл. 2.15).

Таблица 2.15. *Соответствие имен очередей*

Имя, задаваемое в функции *mq_open()*	**Путевое имя очереди сообщений**
/data	/dev/mqueue/data
/acme/data	/dev/mqueue/acme/data
/qnx/data	/dev/mqueue/qnx/data

С помощью команды `ls` можно отобразить все очереди сообщений в системе, например:

```
ls -Rl /dev/mqueue
```

Размер, отображаемый при выполнении этой команды, означает количество ожидающих сообщений.

Функции управления очередями сообщений

Управление очередями сообщений стандарта POSIX выполняется с помощью функций, отраженных в табл. 2.16.

***Таблица 2.16.** Функции управления очередями сообщений*

Функция	Описание
mq_open()	Открыть очередь сообщений
mq_close()	Закрыть очередь сообщений
mq_unlink()	Удалить очередь сообщений
mq_send()	Добавить сообщение в очередь сообщений
mq_receive()	Прочитать сообщение из очереди сообщений
mq_notify()	Сообщить вызывающему процессу о том, что в очереди имеется сообщение
mq_setattr()	Установить атрибуты очереди
mq_getattr()	Получить атрибуты очереди

Разделяемая память

Разделяемая память (shared memory) обеспечивает максимальную пропускную способность механизма межзадачного взаимодействия. После создания некоторого объекта в разделяемой памяти процессы, имеющие доступ к этому объекту, могут использовать указатели (pointers) для непосредственного чтения или записи данных. Это означает, что доступ к разделяемой памяти, по сути, является *несинхронным*. Если процесс обновляет содержание некоторой области разделяемой памяти, другой процесс не должен в этот момент обращаться к этой области для чтения или записи данных в нее. Даже в случае простого чтения данных такое действие может быть некорректным, так как процесс может получить неверную или испорченную информацию.

Для решения этой проблемы разделяемая память часто используется в сочетании с одним из примитивов синхронизации, что позволяет

процессам обновлять данные атомарно. Однако если размер обновляемого блока данных будет невелик, то само использование примитивов синхронизации сильно ограничит пропускную способность. Таким образом, наибольшая производительность разделяемой памяти достигается в том случае, когда блок обновляемых данных имеет достаточно большой размер.

В качестве примитивов синхронизации для разделяемой памяти подходят как семафоры, так и мутексы. В стандарте реального времени POSIX семафоры были определены как средство для межзадачной синхронизации, в то время как мутексы — для межпотоковой синхронизации. Мутексы могут также использоваться для синхронизации потоков в разных процессах. В стандарте POSIX эта возможность определяется как необязательная. Тем не менее, в ОС QNX Neutrino она имеется. В целом, мутексы работают более эффективно, чем семафоры.

Разделяемая память с механизмом обмена сообщениями

Сочетание разделяемой памяти с механизмом обмена сообщениями обеспечивает следующие возможности межзадачного взаимодействия:

- очень высокая производительность (разделяемая память);
- синхронизация (обмен сообщениями);
- сетевая прозрачность (обмен сообщениями).

С помощью механизма обмена сообщениями клиент отправляет запрос серверу и блокируется. Сервер получает от клиентов сообщения в соответствии с их приоритетами, обрабатывает их и отправляет ответ в том случае, если может выполнить запрос. Сама операция передачи сообщения служит естественным механизмом синхронизации между клиентом и сервером. Вместо копирования всех данных посредством обмена сообщениями передаваемое сообщение может содержать ссылку на область в разделяемой памяти. Благодаря этой ссылке сервер может прочитать или записать данные самостоятельно. Проиллюстрируем это следующим примером.

Допустим, у нас есть графический сервер, который принимает от клиентов запросы на создание графических изображений и помещает их в буфер кадров на графической карте. Используя только механизм обмена сообщениями, клиент отправит серверу сообщение, содержащее данные этого изображения, в результате чего данные будут скопированы из адресного пространства клиента в адресное пространство сервера. После этого сервер выполнит обработку изображения и отправит клиенту короткое ответное сообщение.

Если бы клиент отправил внутри сообщения не данные изображения, а ссылку на область разделяемой памяти, в которой эти данные изображения содержатся, то сервер смог бы получить клиентские данные *непосредственно*.

Поскольку клиент блокируется на сервере после передачи ему сообщения, сервер знает, что данные в разделяемой памяти стабильны и не изменятся до тех пор, пока сервер не отправит ответ. Такая комбинация механизма обмена сообщениями и разделяемой памяти обеспечивает естественную синхронизацию и очень высокую производительность.

Описанная модель может работать и в обратном направлении, т. е. сервер может генерировать данные и передавать их клиенту. Например, допустим, клиент передает сообщение серверу, в соответствии с которым сервер должен прочитать видеоданные непосредственно с устройства CD-ROM в приготовленный клиентом буфер разделяемой памяти. В течение того времени, пока содержание разделяемой памяти изменяется, клиент блокируется на сервере. После получения ответа от сервера, клиент возобновляет работу и снова может обращаться к разделяемой памяти. Одновременное использование множества областей разделяемой памяти позволяет организовать конвейерную обработку.

Простая разделяемая память не может использоваться между процессами на разных компьютерах, соединенных через сеть. С другой стороны, нужно отметить, что механизм обмена сообщениями прозрачен для сети. Сервер может использовать разделяемую память для локальных клиентов, а механизм обмена сообщениями для удаленных клиентов. Это позволяет получить высокопроизводительный сервер, который к тому же имеет сетевую прозрачность.

На практике примитивы обмена сообщениями сами по себе почти всегда обеспечивают более чем достаточную скорость работы межзадачного взаимодействия. Комбинированная модель нужна только в особых случаях, когда требуется получить очень высокую пропускную способность.

Создание объектов разделяемой памяти

Потоки внутри процесса совместно используют память, выделенную этому процессу. Для совместного использования памяти разными процессами необходимо создать область разделяемой памяти и затем отобразить эту область в адресное пространство процесса. Для создания и управления областями разделяемой памяти используются вызовы, описанные в табл. 2.17.

Таблица 2.17. *Функции создания и управления областями разделяемой памяти*

Функция	Описание
shm_open()	Открыть (или создать) область разделяемой памяти
close()	Закрыть область разделяемой памяти
mmap()	Отобразить область разделяемой памяти в адресное пространство процесса

Таблица 2.17 *(окончание)*

Функция	Описание
munmap()	Отсоединить область разделяемой памяти от адресного пространства процесса
mprotect()	Изменить атрибуты защиты для заданной области разделяемой памяти
msync()	Синхронизировать содержимое памяти с физической памятью
shm_ctl()	Назначить специальные атрибуты для объекта разделяемой памяти
shm_unlink()	Удалить область разделяемой памяти

Разделяемая память стандарта POSIX реализуется в ОС QNX Neutrino посредством администратора процессов (procnto). Описанные ранее вызовы реализуются в виде сообщений, передаваемых procnto *(см. главу 5)*.

Функция *shm_open()* принимает те же аргументы, что и функция *open()*, и возвращает объекту дескриптор файла. Эта функция позволяет создавать новые и открывать существующие объекты разделяемой памяти, как обычные файлы.

При создании нового объекта разделяемой памяти его размер устанавливается равным нулю. Для установки размера используется функция *ftruncate()* или функция *shm_ctl()*. Отметим, что это те же самые функции, которые используются для установки размеров *файлов*.

mmap()

Если объект разделяемой памяти имеет дескриптор файла, то с помощью функции *mmap()* этот объект (или его часть) можно отобразить в адресное пространство процесса. Функция *mmap()* является одним из главнейших инструментов управления памятью в ОС QNX Neutrino, поэтому опишем ее подробнее.

Функция *mmap()* определяется следующим образом:

```
void * mmap(void *where_I_want it, size_t_length,
int memory_protections, int mapping_flags, int fd,
off_t offset_within_shared_memory);
```

Говоря простым языком, это означает: "Отобразить в адресное пространство текущего процесса *length* байтов по смещению *offset_within_shared_memory* из объекта разделяемой памяти, связанного с дескриптором файла *fd*".

Функция *mmap()* отображает содержимое памяти по адресу *where_i_want_it* в адресном пространстве. Эта область памяти получит атрибуты защиты, заданные как *memory_protections*, а отображение памяти будет выполнено с учетом флагов *mapping_flags*.

Аргументы *fd*, *offset_within_shared_memory* и *length* определяют параметры части объекта разделяемой памяти, которая должна быть отображена в адресное пространство. Часто отображается весь объект целиком; в этом случае смещение (*offset_within_shared_memory*) задается равным нулю, а размер (*length*) — равным размеру объекта в байтах. На процессорах Intel этот параметр определяется числом, кратным размеру страницы, который составляет 4096 байтов (рис. 2.22).

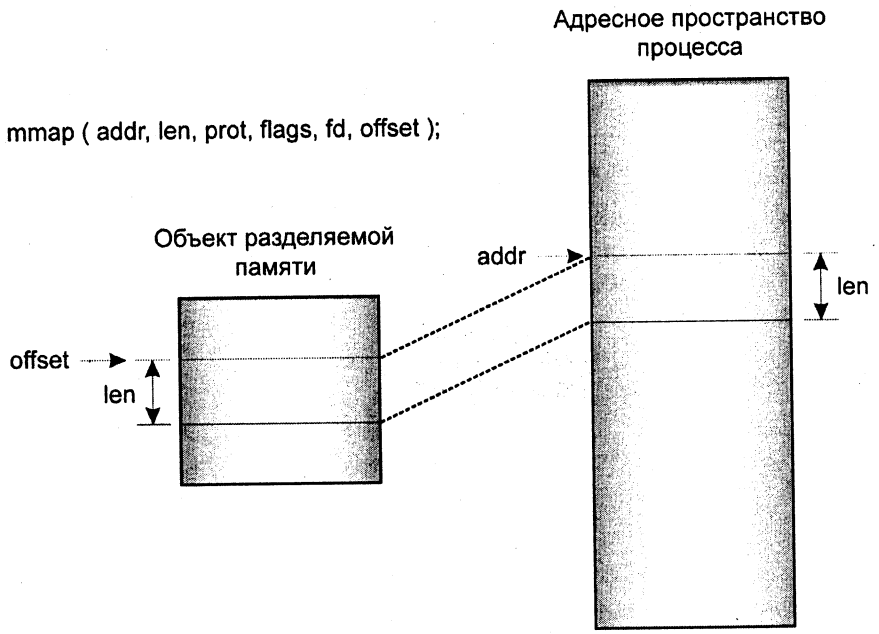


Рис. 2.22. Аргументы функции *mmap()*

Функция *mmap()* возвращает адрес, по которому объект был отображен. Аргумент *where_i_want_it* используется системой в качестве указания на то, куда именно объект следует поместить. Если это возможно, объект будет помещен по заданному адресу. Большинство приложений задают адрес равным нулю, что дает системе возможность самостоятельно определять адрес для размещения объекта.

В качестве атрибута защиты *memory_protections* могут быть заданы следующие типы — табл. 2.18.

Таблица 2.18. *Типы атрибута защиты*

Декларация	Описание
PROT_EXEC	Содержимое памяти может быть исполнено
PROT_NOCACHE	Отключить кеширование памяти
PROT_NONE	Доступ запрещен
PROT_READ	Доступ на чтение
PROT_WRITE	Доступ на запись

Декларация PROT_NOCACHE используется в тех случаях, когда область разделяемой памяти служит для доступа к двухпортовой памяти с аппаратным управлением (например, буфер видеокадров, или плата сетевого адаптера, или системы связи с отображением памяти). Без этой декларации процессор может возвращать "устаревшие" данные, которые ранее были кешированы.

Флаги, задаваемые в *mapping_flags*, определяют способ отображения памяти. Эти флаги разбиваются на две части. Первая часть означает тип флага (табл. 2.19).

Таблица 2.19. *Типы отображения памяти*

Тип отображения памяти	Описание
MAP_SHARED	Отображение совместно используется вызывающими процессами
MAP_PRIVATE	Отображение используется только вызывающим процессом. Этот тип выделяет системную память и создает копию объекта
MAP_ANON	Аналогичен типу MAP_PRIVATE, за исключением того, что параметр *fd* не используется (должен быть установлен в значение NOFD), а выделяемая память заполняется нулями

Тип MAP_SHARED используется для создания разделяемой памяти между процессами. Другие типы имеют более специальное назначение. Например, тип отображения MAP_ANON может использоваться как основа для механизма постраничного выделения памяти (page-level memory allocator).

Некоторые флаги могут быть установлены (с помощью логической операции ИЛИ) в указанный ранее тип для более точного определения метода отображения памяти. Эти флаги подробно описаны в функции *mmap()*

в "Справочнике по библиотекам языка Си" (Library Reference). В табл. 2.20 дано описание нескольких наиболее интересных флагов.

Таблица 2.20. *Модификатор типа отображения памяти*

Модификатор типа отображения памяти	Описание
MAP_FIXED	Отобразить объект по адресу, заданному параметром *where_i_want_it*. Если область разделяемой памяти содержит указатели, тогда эту область, возможно, потребуется расположить по одному и тому же адресу в адресных пространствах всех процессов, отображающих ее. Этого можно избежать, используя смещение внутри области разделяемой памяти вместо прямых указателей
MAP_PHYS	Данный флаг задает работу с физической памятью. Параметр *fd* должен быть установлен в значение NOFD. В сочетании с MAP_SHARED параметр *offset_within_shared_memory* задает точный адрес (например, для буфера видеокадров). В сочетании с MAP_ANON происходит выделение физически сплошной области памяти (например, для DMA-буфера). Флаги MAP_NOX64K и MAP_BELOW16M служат для дальнейшего определения метода выделения памяти типа MAP_ANON и ограничений адресов, которые существуют в некоторых формах DMA
MAP_NOX64K	(Только для семейства процессоров x86.) Используется вместе с MAP_PHYS | MAP_ANON. Выделенная область памяти не может превышать 64 Кбайт. Этот флаг необходим для старых 16-битных контроллеров DMA
MAP_BELOW16M	(Только для семейства процессоров x86.) Используется вместе с MAP_PHYS | MAP_ANON. Выделенная область памяти не может занимать в физической памяти более 16 Мбайт. Это требуется для работы DMA вместе с устройствами на шине ISA

С помощью описанных ранее флагов отображения процесс может легко управлять совместным с другими процессами использованием памяти:

```
/* Отобразить область разделяемой памяти. */
fd = shm_open("datapoints", O_RDWR);
addr = mmap(0, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
```

Или совместным с оборудованием использованием памяти, например, видеопамятью:

```
/* Отобразить видеопамять VGA. */
addr = mmap(0, 65536, PROT_READ|PROT_WRITE,
MAP_PHYS|MAP_SHARED, NOFD, 0xa0000);
```


Или выделять буфер DMA-памяти для сетевой карты PCI:

```
/* Выделить физически сплошной буфер. */
addr = mmap(0, 262144, PROT_READ|PROT_WRITE|PROT_NOCACHE,
MAP_PHYS|MAP_ANON, NOFD, 0);
```

С помощью функции *munmap()* объект разделяемой памяти можно полностью или частично отсоединить (unmap) от адресного пространства. Применение этого примитива не ограничивается отсоединением разделяемой памяти. Он также может использоваться для отсоединения любой области памяти, занятой в каком-либо процессе. В сочетании с флагом MAP_ANON функция *munmap()* позволяет реализовать индивидуальный механизм постраничного выделения/отсоединения памяти (private page-level allocator/deallocator).

С помощью функции *mprotect()* можно изменить атрибуты защиты отображенной области памяти. Так же, как и функция *munmap()*, функция *mprotect()* не ограничивается областями разделяемой памяти и может применяться для изменения атрибутов защиты любой области памяти, занятой в каком-либо процессе.

Неименованные и именованные каналы

Примечание

Для использования неименованных каналов (pipes) и именованных каналов (FIFOs) необходимо, чтобы был запущен `pipe` — администратор программных каналов (pipe resource manager).

Неименованные каналы

Неименованный канал (pipe) — это неименованный файл, который служит в качестве канала ввода/вывода между двумя и более взаимодействующими процессами. Один процесс записывает данные в канал, другой процесс читает эти данные из канала. Администратор программных каналов `pipe` предназначен действовать в роли буфера данных. Размер буфера определяется как PIPE_BUF в файле <limits.h>. Канал уничтожается после закрытия его сторон. Функция *pathconf()* возвращает значение размера буфера.

Каналы обычно используются для организации параллельной работы двух процессов. При этом данные передаются по каналу от одного процесса другому только в одном направлении. (Если необходимо установить двунаправленное взаимодействие, вместо каналов должен использоваться обмен сообщениями.)

Типичный способ применения канала — соединение выхода одной программы со входом другой программы. Такое соединение часто устанавливается с помощью командного интерпретатора. Например:

```
ls | more
```

В результате выполнения этой операции выходные данные утилиты `ls` будут направлены по каналу на вход утилиты `more`.

Чтобы	Используйте
создать канал из командного интерпретатора	символ канала (" | ")
создать канал из программы	функцию *pipe()* или *popen()*

Именованные каналы

Именованные каналы (FIFOs) — по сути то же самое, что и неименованные каналы, но они представляют собой именованные постоянные файлы, которые хранятся в каталогах файловой системы.

Чтобы	Используйте
создать именованный канал из командного интерпретатора	утилиту `mkfifo`
создать именованный канал из программы	функцию *mkfifo()*
создать именованный канал из командного интерпретатора	утилиту `rm`
удалить именованный канал из программы	функцию *remove()* или *unlink()*

Службы управления часами и таймерами

Службы управления часами (clock services) используются для отсчета системного времени, которое необходимо ядру для обеспечения работы интервальных таймеров, реализуемых посредством соответствующих системных вызовов.

Замечание

В ОС QNX Neutrino значение даты допустимо в диапазоне от января 1970 года до января 2554 года, хотя в соответствии со стандартами POSIX программный код не может содержать значения даты больше 2038 года. Внутреннее представление даты и времени не может превышать максимального значения 2554. Если система должна оперировать значениями больше 2554, но возможность провести модификацию системы отсутствует, следует проявлять особое внимание при работе с системными датами (при необходимости, пожалуйста, обратитесь по этому вопросу в компанию QNX Software Systems).

Вызов ядра *ClockTime()* позволяет установить системные часы с идентификатором ID (CLOCK_REALTIME) или получить их значение. После установки значение системного времени увеличивается на некоторое число наносекунд в зависимости от разрешения системных часов. Получить или установить значение этого разрешения можно с помощью вызова *ClockPeriod()*.

Системная страница (system page), которая служит в качестве ОЗУ-резидентной структуры данных, содержит 64-битное поле (*nsec*), которое отображает количество наносекунд, прошедшее с момента начальной загрузки системы. Поле *nsec* всегда увеличивается монотонным образом и никогда не зависит от текущего времени, установленного функцией *ClockTime()* или *ClockAdjust()*.

Функция *ClockCycles()* возвращает текущее значение автономно работающего 64-битного счетчика циклов. Это обеспечивает на любом процессоре высокопроизводительный механизм для отсчета коротких интервалов времени. Например, на процессорах Intel x86 выполняется операция чтения счетчика "тиков" времени. На процессорах Pentium такой счетчик увеличивает свое значение на каждом такте. Таким образом, на процессоре Pentium с частотой 100 МГц один цикл будет длиться 1/100 000 000 секунды (10 наносекунд). В других процессорных архитектурах используются аналогичные механизмы.

В некоторых процессорах (например, в 386-ом) данный механизм не реализуется аппаратными средствами, а эмулируется ядром. Это позволяет получить более высокое разрешение времени (838,095345 наносекунд в системе типа IBM PC), чем при использовании машинной команды.

Во всех случаях поле `SYSPAGE_ENTRY(qtime)->cycles_per_sec` отображает значение приращения счетчика *ClockCycles()* в одну секунду.

Функция *ClockPeriod()* позволяет потоку установить разрешающую способность системного таймера в какое-либо значение, кратное наносекундам. Ядро ОС QNX Neutrino использует все аппаратные возможности для достижения заданной точности.

Выбранный интервал времени всегда округляется до целого значения, близкого к разрешению таймера системного оборудования. Естественно, если выбрать слишком низкое значение, это может привести к значительным расходам производительности на обработку прерываний по таймерам.

В табл. 2.21 приведены функции, используемые при работе со временем.

Таблица 2.21. *Вызовы микроядра и соответствующие POSIX-вызовы*

Вызов микроядра	POSIX-вызов	Описание
ClockTime()	*clock_gettime()*, *clock_settime()*	Получить или установить время и дату (используя 64-битное значение в наносекундах в диапазоне от 1979 до 2554)
ClockAdjust()	*отсутствует*	Применить тонкую корректировку времени для синхронизации часов
ClockCycles()	*отсутствует*	Прочитать значение 64-битного высокоточного автономного счетчика

Таблица 2.21 (окончание)

Вызов микроядра	POSIX-вызов	Описание
ClockPeriod()	*clock_getres()*	Получить или установить размер периода часов
ClockId()	*clock_getcpuclockid()*, *pthread_getcpuclockid()*	Получить целое значение, переданное функции *ClockTime()* как `clockid_t`

Корректировка времени

Для того чтобы обеспечить корректировку системного времени без "скачков" (или даже "возвратов"), вызов *ClockAdjust()* предусматривает возможность задания интервала времени, в течение которого корректировка должна быть выполнена. В результате применения этой опции системное время может ускоряться или замедляться в течение заданного интервала, пока система не произведет синхронизацию с указанным текущим временем. Эта служба может использоваться для выполнения синхронизации времени между множеством сетевых узлов.

Таймеры

ОС QNX Neutrino обеспечивает полный набор таймеров стандарта POSIX. Они являются очень удобным инструментом ядра, поскольку их можно быстро создать, и ими легко управлять.

Модель таймеров стандарта POSIX имеет весьма широкие возможности. Срок действия таймера может определяться следующими параметрами:

- абсолютной датой;
- относительной датой (например, "*n* наносекунд от настоящего момента");
- циклическим периодом (например, "каждые *n* наносекунд").

Циклический режим имеет очень большое значение, так как наиболее часто таймер используется в качестве периодического источника событий — он помогает "пробудить" поток, чтобы он выполнил какой-то цикл вычислений, и затем снова "усыпить" его до возникновения следующего события. Необходимость перепрограммировать таймер на каждое следующее событие могла бы привести к сбоям отсчета времени, за исключением случаев, когда программирование выполняется по абсолютному времени. Если бы запуску потока по таймеру могло препятствовать вытеснение данного потока потоком с более высоким приоритетом, то следующая дата, на которую установлен таймер, могла бы быть пропущена!

Циклический режим позволяет избежать этих проблем, поскольку при таком режиме поток должен устанавливать таймер один раз и затем только реагировать на созданный источник периодических событий.

Так как таймеры являются разновидностью источников событий в ОС, они также используют систему передачи событий. Это дает возможность приложению потребовать, чтобы все события, реализуемые в QNX Neutrino, доставлялись при возникновении таймаута.

В качестве службы управления таймаутами (timeout service), в которой часто возникает необходимость, используется возможность задания максимального времени, в течение которого приложение готово ожидать завершения указанного вызова ядра или запроса. Однако проблема, связанная с использованием службы таймеров в ОС реального времени с вытесняющей многозадачностью (preemptive realtime OS), состоит в том, что в течение периода, протекающего между заданием таймаута и запросом службы, может быть запущен процесс с более высоким приоритетом, который будет выполняться дольше заданного таймаута, и поэтому запрос службы даже не произойдет. В результате приложение не будет делать попытки запроса службы из-за истекшего таймаута (т. е. из-за отсутствия таймаута). Такое временное окно может приводить к "зависанию" процессов, непонятным задержкам при передаче данных и другим проблемам.

```
alarm(...);
.
.       ← Подача сигнала.
.
blocking_call();
```

Для решения этой проблемы в ОС QNX Neutrino используется запрос таймаута, который является атомарным по отношению к самому запросу службы. Другим решением могло бы быть применение необязательного параметра таймаута в каждом запросе службы, однако это чрезмерно усложнило бы структуру запроса, в то время как параметр использовался бы далеко не всегда.

В ОС QNX Neutrino используется вызов ядра *TimerTimeout()*, позволяющий приложению задать список состояний блокировки, при которых должен быть запущен соответствующий таймаут. В результате, когда приложение делает запрос ядру, ядро автоматически включает установленный таймаут в том случае, если приложение может быть блокировано по какому-либо из заданных состояний.

Поскольку в ОС QNX Neutrino предусмотрено только очень небольшое количество состояний блокировки, этот механизм работает очень быстро. По завершении либо запроса, либо таймаута, таймер будет отключен и управление отдано обратно приложению.

```
TimerTimeout(...);
.
.
.
blocking_call();
.
.       ← Таймер атомарно включается в ядре.
.
```

В табл. 2.22 приведены функции, используемые при работе с таймерами.

Таблица 2.22. *Вызовы микроядра и соответствующие POSIX-вызовы*

Вызов микроядра	POSIX-вызов	Описание
TimerAlarm()	*alarm()*	Установить для процесса "будильник"
TimerCreate()	*timer_create()*	Создать интервальный таймер
TimerDestroy()	*timer_delete()*	Уничтожить интервальный таймер
TimerGettime()	*timer_gettime()*	Получить остаток времени в интервальном таймере
TimerGetoverrun()	*timer_getoverrun()*	Получить количество переполнений интервального таймера
TimerSettime()	*timer_settime()*	Запустить интервальный таймер
TimerTimeout()	*sleep(), nanosleep(), sigtimedwait(), pthread_cond_timedwait(), pthread_mutex_trylock(), intr_timed_wait()*	Включить таймаут ядра для какого-либо состояния блокировки

Обработка прерываний

Компьютеры не могут работать с бесконечной скоростью, как бы сильно мы этого ни хотели. Поэтому для системы реального времени крайне важно, чтобы вычислительные циклы процессора не тратились впустую. Также очень важно минимизировать время между возникновением внешнего

события и выполнением программного кода внутри потока, предназначенного для обработки этого события. Это время называется *задержкой* (latency).

Наиболее важными формами задержки являются следующие: задержка обработки прерывания (interrupt latency) и задержка планирования (scheduling latency).

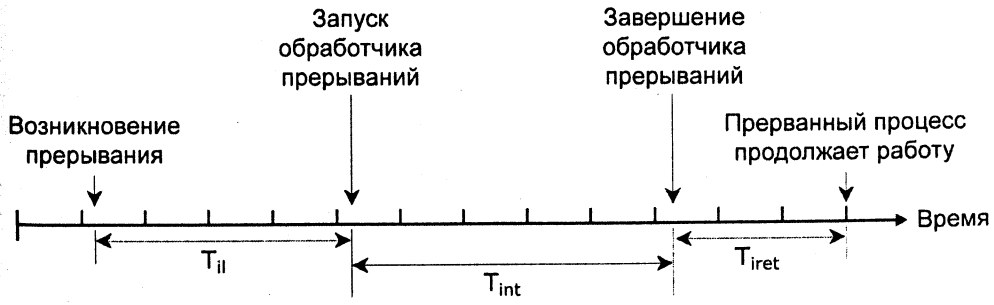
Замечание

Время задержки может быть очень различным в зависимости от производительности процессора и других факторов. Более подробную информацию можно найти на веб-сайте компании QNX Software Systems (**www.qnx.com**).

Задержка обработки прерывания

Задержка обработки прерывания (interrupt latency) — это время, прошедшее от момента возникновения аппаратного прерывания до выполнения первой команды обработчиком прерывания в драйвере устройства. В ОС QNX Neutrino прерывания почти всегда остаются разрешенными, поэтому задержка обработки прерывания обычно незначительная. Однако некоторые критические секции программного кода требуют временного запрета прерываний. Обычно максимальное время такого запрета и определяет наибольшую задержку обработки прерывания, и в QNX Neutrino оно составляет очень небольшое значение.

На рис. 2.23 показано, как происходит обработка аппаратного прерывания обработчиком прерываний. Обработчик прерываний либо просто возвращается, либо возвращается и запускает событие.



T_{il} - задержка обработки прерывания
T_{int} - время обработки прерывания
T_{iret} - время завершения прерывания

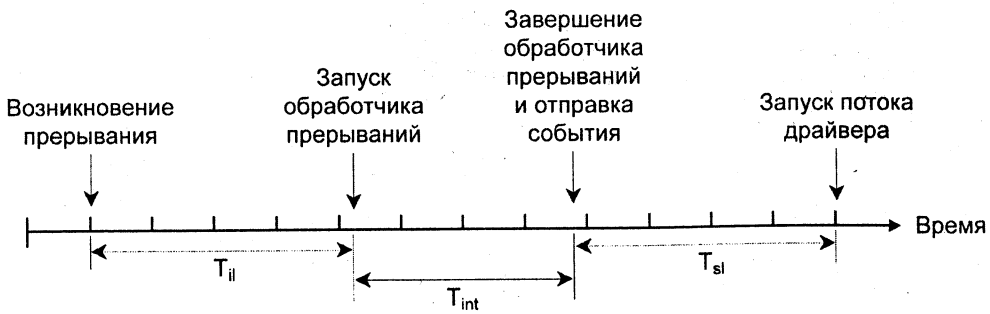
Рис. 2.23. Простое завершение обработчика прерываний

На рис. 2.23 задержка обработки прерывания (T_{il}) обозначает *минимальную* задержку, которая возникает при условии, что прерывания были разрешены в тот момент, когда произошло данное прерывание. Максимальное время задержки обработки прерывания будет определяться как *сумма* минимальной задержки и наибольшего времени, за которое ОС (или активный системный процесс) запрещает аппаратные прерывания.

Задержка планирования

В некоторых случаях низкоуровневый обработчик аппаратных прерываний должен запланировать запуск обработчика прерываний более высокого уровня (потока). При таком сценарии обработчик прерываний возвращается и сообщает о необходимости передачи события. Здесь возникает вторая форма задержек — *задержка планирования*.

Задержка планирования (рис. 2.24) — это время между последней командой обработчика прерываний и выполнением первой команды потока драйвера. Как правило, это означает время, за которое выполняется сохранение контекста текущего потока и загрузка контекста требуемого потока драйвера. Хотя задержка планирования происходит дольше, чем задержка обработки прерывания, в ОС QNX Neutrino этот период времени также имеет довольно небольшое значение.



T_{il} - задержка обработки прерывания

T_{int} - время обработки прерывания

T_{sl} - задержка планирования

Рис. 2.24. Обработчик прерываний завершает работу, возвращая событие

Важно отметить, что *большинство* прерываний завершается без передачи события. Как правило, обработчик прерываний может самостоятельно решить все вопросы аппаратного уровня. Передача события для пробуждения потока *драйвера* верхнего уровня осуществляется только в том случае, когда

происходит значительное событие. Например, во время взаимодействия с драйвером последовательных портов обработчик прерываний может передавать оборудованию один байт данных при получении каждого прерывания передачи данных и запускать поток более высокого уровня (т. е. поток модуля `devc-ser*`) только в том случае, когда выходной буфер почти пуст.

Вложенные прерывания

Этот механизм полностью поддерживается в ОС QNX Neutrino. В предыдущих сценариях описывалась самая простая и наиболее распространенная ситуация, когда возникает только одно прерывание. Однако для получения максимального значения задержки в отсутствие немаскированных прерываний следует рассматривать время с учетом сразу всех текущих прерываний, так как немаскированное прерывание с более высоким приоритетом будет вытеснять текущее прерывание.

На рис. 2.25 показан выполняемый Поток А. Прерывание IRQx запускает обработчик прерываний Intx, который вытесняется прерыванием IRQy и обработчиком прерываний Inty. Обработчик прерываний Inty возвращает событие, которое запускает Поток В, а обработчик прерываний Intx возвращает событие, которое запускает Поток С.

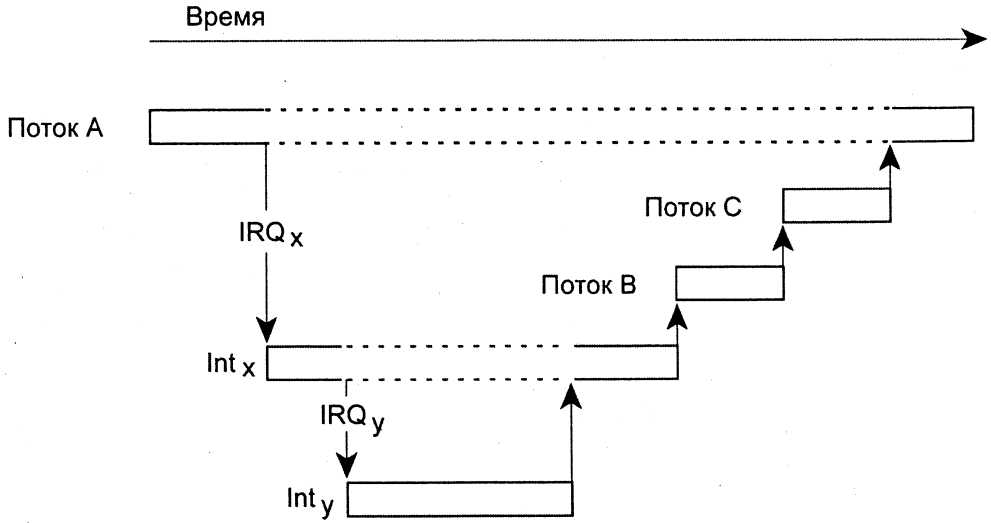


Рис. 2.25. Пакет прерываний

Вызовы, связанные с прерываниями

Программный интерфейс обработки прерываний включает в себя следующие вызовы ядра — табл. 2.23.

Таблица 2.23. *Функции вызовов ядра*

Функция	Описание
InterruptAttach()	Присоединить локальную функцию к вектору прерываний
InterruptAttachEvent()	Генерировать при возникновении прерывания событие, которое переведет поток в активное состояние. Пользовательский обработчик прерываний при этом не запускается. Этот вызов является предпочтительным
InterruptDetach()	Отсоединиться от прерывания, используя идентификатор, возвращенный функцией *InterruptAttach()* или *InterruptAttachEvent()*
InterruptWait()	Ожидать прерывание
InterruptEnable()	Включить аппаратные прерывания
InterruptDisable()	Выключить аппаратные прерывания
InterruptMask()	Маскировать аппаратное прерывание
InterruptUnmask()	Снять маску с аппаратного прерывания
InterruptLock()	Защитить критическую секцию кода между обработчиком прерываний и потоком. Для того чтобы обеспечить возможность применения этого кода в SMP-системах, используется блокировка активного ожидания (spinlock) . Эта функция является расширением функции *InterruptDisable()* и должна использоваться вместо нее
InterruptUnlock()	Снять защиту критической секции программного кода

Посредством этого программного интерфейса поток с соответствующими пользовательскими привилегиями может вызывать функцию *InterruptAttach()* или *InterruptAttachEvent()*, передавая номер аппаратного прерывания и адрес функции в адресном пространстве потока, которая должна быть вызвана при возникновении прерывания. ОС QNX Neutrino позволяет с каждым номером аппаратного прерывания связывать множество обработчиков прерываний (ISR). Немаскированные прерывания могут обрабатываться во время выполнения уже запущенных обработчиков прерываний.

Замечание

Более подробную информацию о функциях *InterruptLock()* и *InterruptUnlock()* можно найти в *разделе "Критические секции программного кода" главы 4.*

Далее приводится пример программного кода, с помощью которого обработчик прерываний (ISR) присоединяется к аппаратному прерыванию таймера (которое ОС также использует в качестве системных часов). Обработчик прерываний таймера, имеющийся в ядре, самостоятельно выполняет очистку источников прерываний, поэтому он только увеличивает значение счетчика в пространстве данных потока и затем передает управление ядру.

```
#include <stdio.h>
#include <sys/neutrino.h>
#include <sys/syspage.h>
struct sigevent event;
volatile unsigned counter;
const struct sigevent *handler( void *area, int id ) {
// Пробуждение потока на каждом 100-ом прерывании.
if ( ++counter == 100 ) {
counter = 0;
return( &event );
}
else
return( NULL );
}
int main() {
int i;
int id;
// Запросить допуск ввода/вывода.
ThreadCtl(_NTO_TCTL_IO, 0 );
// Инициализировать структуру событий.
event.sigev_notify = SIGEV_INTR;
// Подключить вектор обработчика прерываний.
id=InterruptAttach( SYSPAGE_ENTRY(qtime)->intr, &handler,
NULL, 0, 0 );
for( i = 0; i < 10; ++i ) {
// Ожидать вектор обработчика прерываний для пробуждения.
InterruptWait( 0, NULL );
printf( "100 events\n" );
}
```

```
// Отключить обработчик прерываний.
InterruptDetach(id);
return 0;
}
```

В результате пользовательские потоки, имеющие соответствующие привилегии, могут динамически присоединять (и отсоединять) обработчики прерываний к (от) векторам аппаратных прерываний в процессе их выполнения. Эти потоки могут быть отлажены на уровне исходного текста с помощью обычных отладочных средств, а сам обработчик прерываний может быть отлажен посредством вызова этого обработчика на уровне потока и пошагового выполнения на уровне исходного текста или посредством вызова функции *InterruptAttachEvent()*.

Когда происходит аппаратное прерывание, процессор вызывает модуль первичной обработки прерываний (interrupt redirector) микроядра. Данный модуль записывает на стек регистровый контекст выполняемого потока в соответствующий элемент таблицы потоков и затем устанавливает такой контекст, чтобы у обработчика прерываний был доступ к программному коду и данным потока, в котором содержится этот обработчик прерываний. Таким образом, обработчик прерываний получает возможность использовать буферы и код в пользовательском потоке для определения источника возникшего прерывания и, в случае если поток требует выполнения работы более высокого уровня, сгенерировать событие для того потока, в который входит данный обработчик прерываний, после чего этот поток может обработать данные, которые обработчик прерываний поместил в принадлежащий ему буфер.

Поскольку обработчик прерываний отображается в контекст содержащего его потока, он может непосредственно оперировать устройствами, отображенными в адресном пространстве потока, или самостоятельно выполнять команды ввода/вывода. В результате это избавляет от необходимости связывать драйверы устройств с ядром.

Модуль первичной обработки прерываний, который содержится в микроядре, вызывает все обработчики, связанные с данным аппаратным прерыванием. Если возвращенное значение указывает на то, что процессу должно быть передано некоторое событие, ядро ставит это событие в очередь. После того как последний обработчик прерываний вызывается для обработки данного вектора прерываний, обработчик прерываний ядра завершает работу с устройством управления прерываниями и "возвращается из прерывания".

Этот возврат из прерывания не обязательно должен приводить в контекст прерванного потока. Если событие, поставленное в очередь, заставило поток с более высоким приоритетом перейти в состояние готовности (READY), то микроядро выполнит возврат из прерывания в контекст потока, который является активным в текущий момент.

Таким образом устанавливается период (т. н. *задержка обработки прерывания*) между возникновением прерывания и выполнением первой инструкции обработчика прерываний, а также период (т. н. *задержка планирования*) между последней инструкцией обработчика прерываний и первой инструкцией потока, приведенного обработчиком прерываний в состояние готовности.

Период максимальной задержки обработки прерывания строго определяется благодаря тому, что ОС запрещает прерывания только на время выполнения нескольких инструкций в нескольких критических секциях программного кода. Периоды, в течение которых прерывания запрещены, продолжаются строго определенное время и не имеют зависимости от данных.

Модуль первичной обработки прерываний, имеющийся в микроядре, выполняет несколько инструкций перед тем, как вызвать обработчик прерываний. В результате вытеснение процессов по аппаратным прерываниям или вызовам ядра происходит одинаково быстро и посредством одной и той же цепи инструкций.

Во время своей работы обработчик прерываний имеет полный доступ к оборудованию (поскольку является частью привилегированного потока), но другие вызовы ядра выполнять не может. Обработчик прерываний должен реагировать на аппаратное прерывание с максимально возможной скоростью, выполнять минимальный объем работы для обслуживания прерывания (чтение байта из универсального асинхронного приемопередатчика (UART) и т. д.) и, при необходимости, производить планирование потока с некоторым приоритетом для выполнения дальнейшей работы.

Максимальная задержка обработки прерывания для заданного аппаратного приоритета может быть непосредственно вычислена, исходя из значения задержки обработки прерывания, вносимой ядром, и максимального времени работы обработчика прерываний по каждому прерыванию, имеющему более высокий аппаратный приоритет, чем заданный обработчик. Поскольку приоритеты аппаратных прерываний могут переназначаться, самому важному прерыванию в системе можно присвоить наивысший приоритет.

Следует отметить, что вызов функции *InterruptAttachEvent()* не приводит к запуску обработчика прерываний. Вместо этого по каждому прерыванию генерируется заданное пользователем событие. Как правило, это событие приводит к планированию ожидающего потока, чтобы он мог выполнить основную работу. Прерывание автоматически маскируется после генерации события и должно в нужный момент явно демаскироваться потоком, обслуживающим устройство.

Замечание

Обе функции *InterruptMask()* и *InterruptUnmask()* являются *счетными*. Например, если функция *InterruptMask()* вызвана 10 раз, то *InterruptUnmask()* также должна быть вызвана 10 раз.