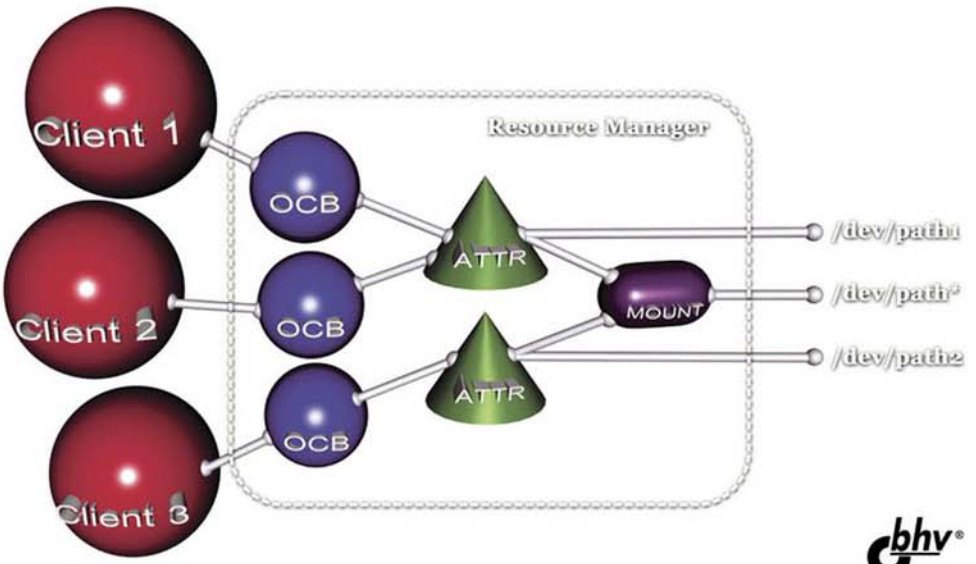
Введение

в QNX® Neutrino®

Руководство для разработчиков приложений реального времени

Роберт Кртен

3-е издание



bhv®

Роберт Кртен

Введение в QNX® Neutrino®

Руководство для разработчиков приложений реального времени

3-е издание

Санкт-Петербург
«БХВ-Петербург»
2015

УДК 004.451
ББК 32.973.26-018.2
К83

Кртен Р.

К83 Введение в QNX® Neutrino®. Руководство для разработчиков приложений реального времени: Пер. с англ. — 3-е изд., исправл. — СПб.: БХВ-Петербург, 2015. — 368 с.: ил.

ISBN 978-5-9775-3606-6

Рассмотрены основные механизмы ядра операционной системы жесткого реального времени QNX Neutrino. Описаны базовые концепции и рекомендации при работе с процессами и потоками, в том числе в распределенной среде. Показана организация периодических событий в программах с помощью таймеров. Уделено внимание администрированию ресурсов и программированию драйверов устройств. Приведены рекомендации по эффективной обработке прерываний. Материал сопровождается большим количеством примеров. В третьем издании исправлены замеченные неточности и опечатки.

Для разработчиков систем реального времени, студентов и преподавателей вузов

УДК 004.451
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Екатерина Капалыгина*
Перевод с англ.	*Александра Алексеева, Николая Горбунова*
Редактор русского перевода	*Николай Горбунов*
Редактор третьего русского издания	*Сергей Зыль*
Компьютерная верстка	*Натальи Смирновой*
Дизайн обложки	*Елены Беляевой*
Оформление обложки	*Марины Дамбиевой*

Подписано в печать 31.03.15.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 29,67.
Тираж 500 экз. Заказ №
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9 линия, 12/28

ISBN 0-9682501-1-4 (англ.)
ISBN 978-5-9775-3606-6 (рус.)

Оглавление

Предисловие к первому изданию **1**

Предисловие к первому русскому изданию **5**

Введение **7**

Немного истории 8
Для кого эта книга? 8
Что содержит эта книга? 9
Процессы и потоки 9
Обмен сообщениями 9
Часы, таймеры и периодические уведомления 10
Прерывания 10
Администраторы ресурсов 10
Переход с QNX4 на QNX Neutrino 10
Скорая помощь 11
Глоссарий 11
Другие источники информации 11
О Роберте Кртене 11
Выражение признательности 11
Типографские соглашения 12

Глава 1. Процессы и потоки **15**

Основные понятия о процессах и потоках 15
Процесс как жилой дом 15
Потоки как обитатели дома 15
Назад к процессам и потокам 16
Взаимное исключение 17
Приоритеты 17
Семафоры 18
Роль ядра 20
Одиночный процессор 20

Несколько процессоров — симметричная мультипроцессорная система (SMP) 21
Ядро в роли арбитра....21
Состояния потоков....26
Процессы и потоки....28
Почему процессы?....29
Запуск процесса....30
Запуск потока....41
Дополнительно о синхронизации....64
Блокировки чтения/записи....65
Ждущие блокировки....67
Условные переменные....71
Дополнительные сервисы Neutrino....78
Диспетчеризация и реальный мир....87
Перепланирование по аппаратному прерыванию....87
Перепланирование по системным вызовам....87
Перепланирование по исключительным ситуациям....88
Резюме....88

Глава 2. Обмен сообщениями 91

Введение в обмен сообщениями....91
Микроядро и обмен сообщениями....91
Обмен сообщениями и модель "клиент-сервер"....92
Распределенный обмен сообщениями....95
Что это означает для вас....97
Философия QNX Neutrino....97
Обмен сообщениями и многопоточность....97
Модель "сервер-субсервер"....98
Несколько примеров....100
Применение обмена сообщениями....102
Архитектура и структура....103
Клиент....103
Сервер....106
Иерархический принцип обмена (send-иерархия)....109
Идентификаторы отправителя, каналы и другие параметры....110
Составные сообщения....121
Сообщения типа "импульс" (pulse)....127
Прием импульса....128
Функция *MsgDeliverEvent()*....131
Флаги канала....132
Флаг _NTO_CHF_UNBLOCK....133
Проблема синхронизации....135
Флаг _NTO_MI_UNBLOCK_REQ....138

Обмен сообщениями в сети ... 138
Особенности обмена сообщениями в сети ... 141
Несколько замечаний о дескрипторах узлов ... 143
Наследование приоритетов ... 145
Так в чем тут хитрость? ... 147
Резюме ... 148

Глава 3. Часы, таймеры и периодические уведомления ... 149

Часы и таймеры ... 149
Периодические процессы ... 149
Источники прерывания таймера ... 151
Разрешающая способность отсчета времени ... 152
Флуктуации отсчета времени ... 153
Типы таймеров ... 154
Схема уведомления ... 155
Применение таймеров ... 159
Создание таймера ... 159
Сигнал, импульс или поток? ... 159
Какой таймер выбрать? ... 160
Сервер с периодическими импульсами ... 162
Таймеры, посылающие сигналы ... 171
Таймеры, создающие потоки ... 172
Опрос и установка часов реального времени и кое-что еще ... 172
Дополнительные возможности ... 174
Другие источники времени ... 174
Тайм-ауты ядра ... 178
Резюме ... 181

Глава 4. Прерывания ... 183

Neutrino и прерывания ... 183
Программа обработки прерывания ... 184
Активность прерываний по уровню и по фронту ... 187
Написание обработчиков прерываний ... 190
Подключение обработчиков прерывания ... 190
Теперь, когда вы подключились к прерыванию ... 191
Отключение обработчика прерывания ... 192
Параметр *flags* ... 193
Обработчик прерывания ... 194
Функции, которые может вызывать ISR ... 203
Резюме ... 205

Глава 5. Администраторы ресурсов 207

Что такое администратор ресурсов? 207
Примеры администраторов ресурсов 207
Характеристики администраторов ресурсов 208
Взгляд со стороны клиента 209
Поиск сервера 209
Поиск администратора процессов 211
Обработка каталогов 212
Объединенные файловые системы 213
Резюме о клиенте 215
Взгляд со стороны администратора ресурсов 216
Регистрация префикса 216
Обработка сообщений 217
Библиотека администратора ресурсов 218
Реально все это за вас делает библиотека 220
За кулисами библиотеки 221
Написание администратора ресурсов 223
Структуры данных 223
Структура администратора ресурсов 230
Структуры данных уровня POSIX 239
Функции-обработчики 246
Общие замечания 246
Замечания о функциях установления соединения 249
Функции установления соединения и ввода/вывода 250
Примеры 278
Базовый каркас администратора ресурсов 278
Простой пример функции *io_read()* 281
Простой пример функции *io_write()* 286
Простой пример функции *io_devctl()* 291
Пример функции *io_devctl()*, имеющей дело с данными 294
Дополнительно 298
Расширение OCB 298
Расширение атрибутной записи 300
Блокирование в пределах администратора ресурсов 301
Возврат элементов каталога 302
Резюме 313

Приложения 315

Приложение 1. Переход с QNX4 на QNX Neutrino 317

QNX4 и Neutrino 317
Сходства 317
Улучшения 318

Философия переноса программ 322
Анализ обмена сообщениями 322
Анализ прокси по идентификаторам 332
Обработчики прерывний 334
Резюме 334

Приложение 2. Скорая помощь 336

Обращайтесь за помощью к профессионалам 336
Итак, у нас проблема 336
RTFM 336
Свяжитесь с технической поддержкой 338
Обучение 341

Глоссарий 342

Перечень иллюстраций 351

Предметный указатель 353

Предисловие к первому изданию

Впервые взгянув на черновик этой книги, я подумал, что это будет трудное чтение, потому что сам много лет провел в разработке QNX Neutrino. Но я ошибался! Я нашел книгу простой, понятной и занимательной — все дело в стиле Роба, сочетающем философию QNX ("Почему все именно так, как оно есть") с полезными общими приемами, применимыми к любому проекту, связанному с задачами реального времени. Эта книга будет полезна как для читателей, никогда прежде не слышавших о Neutrino, так и для специалистов, которые активно используют ее в своих проектах.

Для тех, кто никогда не использовал Neutrino, книга представляет собой превосходное учебное пособие о том, как это делать. Поскольку Роб сам вышел из среды QNX2 и QNX4, его книга также будет очень полезна для специалистов, которые уже имели дело с QNX, поскольку ОС этого семейства имеют много общего.

Что до меня самого, то я впервые познакомился с QNX в середине 80-х прошлого века, когда работал в страховой компании. Изначально там применялся IBM-овский мэйнфрейм, но компания хотела сократить время на расчеты квот для корпоративного страхования; для этого в компании решили применить сеть из 8-мегагерцовых 80286, работающих под управлением QNX2. Было решено распределить данные в прозрачной сети QNX, обеспечив тем самым доступ к файлам данных по всем заказчикам с любой QNX-машины. Клиент-серверная идеология QNX наделила систему такой грацией, что я влюбился в эту ОС с первого взгляда.

Я был приглашен работать в QSS в начале 1991 года, когда была еще только-только выпущена QNX4. Она разрабатывалась в соответствии с только что утвержденной спецификацией POSIX 1003.1, что должно было сделать перенос общедоступных программ из UNIX проще, чем это было в QNX2, и подчинить ОС единому стандарту. Спустя несколько лет мы стали задумываться о создании операционной системы следующего поколения. Группа менее чем из 15 разработчиков стала проводить совещания, обсуждая все то, что мы хотели бы сделать иначе, а также то, что могло нам понадобиться в будущем. Мы хотели обеспечить поддержку новых спецификаций POSIX и облегчить написание драйверов. Мы также не собирались ограничиваться процессорами серии x86 и "ремонтировать то, что работает".

Фундаментальные идеи, которые Дэн Додж и Гордон Белл вложили в QNX изначально, действуют в Neutrino и по сей день — обмен сообщениями, микроядер-

ная архитектура, предсказуемое время реакции и т. д. Усложняла разработку Neutrino цель сделать ее более модульной, чем QNX4 (например, мы хотели создать полнофункциональное ядро, с которым можно было бы просто скомпоновать приложение, что позволило бы применять его в "более встраиваемых" приложениях по сравнению с QNX4). В 1994 году мы с Дэном Доджем начали работу над новой версией ядра и администратора процессов.

Те из вас, кто долго имел дело с QNX, знают, что от такой задачи как написание драйвера устройства для QNX2 волосы встают дыбом. Приходилось быть очень осторожным! В действительности большинство разработчиков просто брали поставляемый с QNX2 исходный текст драйвера спулера и аккуратно прилаживали его под свои нужды. Лишь немногие пытались писать драйверы дисковых устройств, поскольку это требовало специализированных знаний из области ассемблера. Из-за этого практически никому не удавалось довести свои драйверы для QNX2 до конца. В QNX4 написание драйверов было *значительно* упрощено сведением всех стандартных операций ввода/вывода к четко определенному интерфейсу обмена сообщениями. Когда вы вызывали *open()*, сервер получал сообщение типа "открыть ресурс". Когда вы вызывали *read()*, сервер получал сообщение типа "читать данные". Главный выигрыш механизма обмена сообщениями в QNX4 состоял в том, что он развязывал серверы от клиентуры. Помнится, когда я впервые увидел бета-версию QNX 3.99 (пре-релиз QNX4), я подумал: "Вот это да! Как изящно все сделано!" Я был настолько очарован этим, что немедленно написал драйвер файловой системы для QNX2 с использованием этого нового механизма — все вдруг стало так просто!

Администратор процессов Neutrino был разработан с учетом трех основных независимых функций: управление пространством имен путей, управление процессами и управление памятью. Он также поддерживал несколько дополнительных сервисов (**/dev/null**, **/dev/zero**, образная файловая система и т. д.), каждый из которых работал независимо, но все они разделяли общую схему обработки сообщений. Мы нашли эту схему настолько полезной, что решили выделить ее код в отдельную служебную библиотеку. Так появилась библиотека администратора ресурсов (или, как Роб любит ее называть, приводя меня в тихий ужас, "библиотека резмаггера"[1]. :-).

Мы также обнаружили, что большинство администраторов ресурсов должны предоставлять своим устройствам или файловым системам семантику POSIX, поэтому поверх библиотеки администратора ресурсов был написан еще один дополнительный уровень — семейство функций *iofunc*()*. Это позволяет любому человеку писать администраторы ресурсов, автоматически наследующие функциональность POSIX — без каких-либо дополнительных усилий. Примерно в это время Роб писал курсы по Neutrino, и ему был нужен минимальный пример администратора ресур-

[1] "Resmgr" является стандартным, но труднопроизносимым сокращением от "resource manager". Роб, очевидно, решил упростить произношение и добавить гласных — так из "администратора ресурсов" (resource manager) получился "резервный индийский крокодил" (resmugger). Аналогично Роб, кстати, в свое время поступил и со своей фамилией, сделав из "крещеного" (Krten) "занавеску" (curtain) — *прим. ред.*

сов, **/dev/null**. Его основной слайд гласил: "Все, что от вас требуется, — это написать обработчики вызовов *read()* и *write()*, и перед вами готовый **/dev/null**!" Я расценил это как вызов и убрал даже это требование — базированная на библиотеке администратора ресурсов реализация **/dev/null** теперь укладывается в примерно полдюжины вызовов. Поскольку эта библиотека поставляется с Neutrino, теперь каждый может писать POSIX-совместимые администраторы ресурсов с минимальными усилиями.

Однако при том, что концепция администратора ресурсов была значительным шагом в эволюции Neutrino и обеспечивала мощный фундамент для операционной системы, новорожденная ОС требовала большего. Файловые системы, средства коммуникации (например, TCP/IP) и типовые устройства (последовательный интерфейс, консоли) разрабатывались параллельно. В результате огромной работы в начале 1996 года вышла Neutrino 1.00. В течение последующих нескольких лет к работе над Neutrino стали привлекать все больше и больше специалистов отдела исследований и разработки (R&D) компании. Мы дополнили систему поддержкой SMP, многоплатформенностью (x86, PowerPC и MIPS) (на момент перевода также добавлена поддержка ARM и SuperH-4 — *прим. ред.*) и интерфейсом диспетчеризации (он позволяет комбинировать администраторы ресурсов и другие средства межзадачного взаимодействия) — все это описано в этой книге.

В августе 1999 года была официально выпущена Neutrino 2.00 — как раз к моменту выхода книги Роба! :-)

Я думаю, что это издание должно быть настольной книгой каждого, кто пишет программы для Neutrino.

Питер Ван Дер Вин (Peter van der Veen),
С борта самолета где-то между Оттавой и Сан-Хосе,
Сентябрь 1999 г.

Предисловие к первому русскому изданию

Вышедшее осенью 1999 года первое английское издание книги Роберта Кёртена "Getting Started With QNX Neutrino 2" имело на Западе колоссальный успех. Впрочем, такая реакция публики была абсолютно закономерна: литературы по QNX не хватало всегда, особенно если вести речь не просто о документации, а о "философских" книгах, не только рассказывающих "как", но и подробно объясняющих "почему". Успех книги, благодаря широкой популярности QNX среди разработчиков по всему миру, стал с Запада неумолимо распространяться и на Восток — буквально сразу же вышел перевод этой книги на японский язык. Оказавшись таким образом "между двух огней", мы просто не могли не задуматься о необходимости русского перевода.

Решение перевести книгу Роберта на русский язык было принято еще осенью 2000 года. Справедливости ради следует отметить, что вести проекты подобного рода нам на тот момент было уже не впервой: в 1999 году нашей компанией был выпущен комплект русской документации по QNX4, отнявший у нас почти год упорного труда. Впрочем, эта работа доставила нам также и массу удовольствия (редко по нашим временам встретишь техническую литературу с цитатами из Лаоцзы!), поэтому за перевод книги Роберта, особенно прочитав его предыдущую книгу, "Getting Started With QNX4", мы взялись с большим энтузиазмом. При этом, будучи сами разработчиками и зная на собственном опыте, что излишняя сухость придает литературе строгости, но несомненно вредит восприятию, мы решили нарушить каноны и старались во всем соблюдать непринужденный стиль автора, включая идиоматические переводы профессионального сленга, что определенно придало работе дополнительный живой колорит.

Огромное спасибо всем, кто помогал нам работать над первым изданием и "вылавливать" в нем ошибки. Отдельное спасибо самому Роберту, охотно помогавшему в редактировании перевода, поясняя тонкие места и помогая найти нужные формулировки. Мы старались, чтобы эта книга стала не просто переводом с языка на язык, но была максимально приближена к русской аудитории, чтобы как можно успешнее выполнить поставленную перед ней задачу — помочь разработчикам в кратчайший срок сделать красивую и функциональную систему, избежав многих возможных "граблей".

Первое издание русского перевода разошлось в очень короткий срок. Перед вами — третье издание книги. Надеемся, что эта книга станет верным помощником для всех, кто работает или просто интересуется QNX Neutrino. Ждем ваших комментариев и пожеланий по адресу **books.qnx@swd.ru**. Успехов!

Николай Горбунов,
редактор русского перевода[1]

[1] Николай Горбунов поскромничал, когда назвал себя редактором русского перевода. На самом деле он провел колоссальную работу, в результате который мы имеем очень точный перевод этой весьма ценной для системных программистов QNX книги. В случаях, когда оригинальный текст допускал двоякое толкование, Николай уточнял у автора, что именно он имел в виду. Стоит заметить, что Николай является известным специалистом по поэтическому переводу английских афоризмов и идиом, что, вкупе с глубоким знанием технологий QNX, позволило ему сохранить особенности языка и стиля автора. Примечания Николая Горбунова имеют отметку "прим. ред.". Примечания Сергея Зыля, работавшего над переводом изменений, внесенных специалистами компании QNX Software Systems, в английскую редакцию 2010 года, имеют отметку "прим. С. З.". — *прим. С. З.*

Введение

Спустя несколько лет после того, как я приобщился к компьютерам, вышел в продажу первый IBM PC. Я был, наверное, одним из первых в Оттаве, кто купил этот ящик. В нем было 16 Кбайт ОЗУ и отсутствовала видеокарта — неопытный продавец просто не знал, что без видеокарты машина будет абсолютно бесполезной. Впрочем, несмотря на бесполезность, на ящике было красиво написано "IBM" (а тогда такое можно было увидеть только на мэйнфреймах и им подобных) и это уже само по себе выглядело достаточно внушительно. Когда я, наконец, накопил денег на видеокарту, то смог даже запустить БЕЙСИК на телевизоре родителей. Для меня тогда все это было вершиной компьютерной технологии — особенно модем с акустической связью на 300 бод! А теперь представьте себе мою досаду, когда мне позвонил мой друг Пол Транли и сказал: "Эй, залогинься ко мне на компьютер?" Я подумал про себя: "А у *него*-то откуда VAX?" — поскольку из всех известных мне машин, на которые можно было "залогиниться", VAX была единственной, которая влезла бы в его дом. Я позвонил. Это был PC, работающий под загадочной операционной системой по имени QUNIX с номером версии меньше 1.00. Но там можно было сделать "login" — я был в шоке!

Что меня всегда поражало в операционных системах семейства QNX — это небольшой объем требуемой памяти, эффективность и абсолютная элегантность реализации. Я часто за едой развлекал (или утомлял, что более вероятно) приглашенных на ужин гостей своими баснями о программах, параллельно выполнявшихся на моей машине в подвале. Те, кто понимал что-то в компьютерах, начинали прикидывать, какой у меня огромный диск, откуда у меня такой "неограниченный" объем ОЗУ и т. п. После ужина я тащил их вниз, на мой этаж, и показывал им свой простенький PC с 8 Мбайт ОЗУ и винчестером на 70 Мбайт. На некоторых это действовало очень впечатляюще. Тем, на которых не действовало, я показывал, сколько ОЗУ и дискового пространства было еще *доступно*, при том, что бóльшую часть этого дискового пространства занимали мои собственные данные, которые я накопил за годы работы.

Прошли годы, и я имел счастье поработать во многих компаниях, большинство из которых так или иначе занимались разработкой под QNX (телекоммуникации, управление производством, драйверы устройств видеозахвата и т. д.), и где основным требованием была простота — как идеи, так и воплощения. Мне думается, что это требование вытекало из хорошего понимания идеологии QNX главными инженерами проектов — если в основе проекта лежит стройная, изящная архитектура, то велика и вероятность того, что и весь проект в целом будет стройным и изящным (если, конечно, проблема сама по себе не корявая).

В ноябре 1995 года мне выпало счастье работать непосредственно на QNX Software Systems Limited (QSSL), разрабатывая учебные материалы для двух курсов по QNX Neutrino, а затем и преподавая эти курсы в течение более чем трех последующих лет.

Именно последние 19 или около того лет моей работы дали мне вдохновение и смелость написать мою первую книгу, "Введение в QNX4: руководство по программированию приложений реального времени", которая была издана в мае 1998 года. В данной, новой книге по QNX Neutrino я надеюсь изложить ряд накопленных мной на личном опыте концепций и идей, чтобы дать вам четкое, фундаментальное восприятие того, как работает QNX Neutrino и как ее можно эффективно применять. Хочется верить, что после прочтения этой книги в вашей голове вдруг включится лампочка, и вы воскликнете: "Ага! Так вот почему они сделали это именно так!"

Немного истории

Компания QSS, разработавшая операционную систему QNX, была создана в 1980 году Дэном Доджом и Гордоном Беллом (оба — выпускники университета Ватерлоо, расположенного в Онтарио, Канада). Сначала компания называлась Quantum Software Systems Limited, а ее продукт — QUNIX (Quantum UNIX). После вежливого письма юристов компании AT&T (которой в то время принадлежала торговая марка "UNIX") имя продукта изменили на QNX. Спустя некоторое время изменили и название самой компании — на QNX Software Systems, поскольку в те дни казалось, что у всех, и у каждого, и у их собак были компании по имени "Quantum что-то" или как-нибудь в этом духе.

Первый программный продукт, получивший коммерческий успех, назывался просто QNX и работал на процессорах 8088-й серии. Затем, в начале 80-х годов прошлого века, была выпущена операционная система QNX2 (QNX, версия 2). Она до сих пор успешно применяется во многих ответственных приложениях. Примерно в 1991 году появилась новая операционная система, QNX4, с улучшенной поддержкой 32-разрядных операций и стандарта POSIX. И наконец, в 1995 году была заявлена новая модификация ОС семейства QNX, называемая Neutrino.

26 сентября 2000 года вышла новая версия (она получила номер 6) платформы реального времени QNX (включающая операционную систему QNX Neutrino, оконную систему Photon, средства разработки, компиляторы и т. д.), которая была доступна для бесплатного некоммерческого использования. В течение менее чем одного года (к июлю 2001 года) был скачан 1 млн экземпляров!

Для кого эта книга?

Данная книга подойдет любому желающему получить фундаментальное понимание ключевых особенностей QNX Neutrino и принципов ее функционирования.

Из этой книги смогут почерпнуть многое даже читатели с небольшим компьютерным образованием (хотя обсуждение в каждой главе, по мере продвижения вперед, становится все более и более техническим). Даже бывалые хакеры смогут почерпнуть из этой книги кое-какие интересные приемы, особенно касательно двух фундаментальных черт QNX Neutrino — обмена сообщениями и структурной организации драйверов.

Я попытался объяснять сложный материал в легкой для чтения "диалоговой" манере, предвидя некоторые резонные вопросы, которые могли бы возникать по ходу дела, и отвечая на них с примерами и рисунками. Поскольку книга не требует глубокого понимания языка Си, но знание его определенно даст преимущество, в тексте книги есть также и непосредственно примеры программ.

Что содержит эта книга?

Данная книга призвана рассказать читателю, что представляет собой и как работает QNX Neutrino. Главы книги содержат описание состояний процессов, потоков, алгоритмов диспетчеризации, обмена сообщениями, модульной концепции построения ОС и т. д. Если вы ранее никогда не применяли QNX Neutrino, но знакомы с операционными системами реального времени, то вам, возможно, захочется уделить особое внимание главам, посвященным обмену сообщениями и администраторам ресурсов, т. к. именно эти концепции составляют основу QNX Neutrino.

Процессы и потоки

В *главе 1* представлено описание процессов и потоков в QNX Neutrino, диспетчеризации, системы приоритетов и дано понятие о реальном времени. Вы узнаете о состояниях потоков и алгоритмах диспетчеризации, которые применяются в QNX Neutrino, а также изучите функции, применяемые для управления диспетчеризацией, создания процессов и потоков, а также изменения свойств процессов и потоков, которые уже выполняются. Вы увидите, как в QNX Neutrino реализована поддержка SMP и вытекающие из этого преимущества (и подводные камни).

В *разд. "Диспетчеризация и реальный мир" главы 1* обсуждается, как диспетчеризуются потоки в работающей системе, и что может вызвать перепланирование.

Обмен сообщениями

В *главе 2* вы ознакомитесь с наиболее яркой и фундаментальной особенностью QNX Neutrino — принципом обмена сообщениями. Вы изучите, что такое обмен сообщениями, как его применять для общения потоков между собой и как обмениваться сообщениями по сети. Также в этой главе рассмотрен ряд дополнительных

вопросов, включая извечное проклятие систем реального времени — инверсию приоритетов.

Внимание

Это одна из самых важных глав в книге!

Часы, таймеры и периодические уведомления

В *главе 3* вы изучите системные часы, таймеры, а также как заставить таймеры посылать вам сообщения. В ней много практических советов и изобилие примеров кода.

Прерывания

В *главе 4* вы научитесь писать обработчики прерываний для QNX Neutrino и узнаете, как обработчики прерываний влияют на диспетчеризацию потоков.

Администраторы ресурсов

В *главе 5* вы изучите все, что относится к администраторам ресурсов в QNX Neutrino (также известным как "драйверы устройств" и "администраторы ввода/вывода"). Перед написанием собственного администратора ресурса вам необходимо будет внимательно изучить *главу 2*. В *главе 5* также приведены исходные тексты нескольких готовых администраторов ресурсов.

Внимание

Администраторы ресурсов — еще один важный компонент любой системы на базе QNX Neutrino.

Переход с QNX4 на QNX Neutrino

В *приложении 1* представлено неоценимое руководство для всех, кто намерен переносить свои приложения из QNX4 в QNX Neutrino или писать программы для обеих платформ сразу. (QNX4 — операционная система предыдущего поколения от компании QSS, а также тема моей предыдущей книги — "Введение в QNX4".) Даже если вы разрабатываете новое приложение, у вас может быть необходимость поддерживать QNX4 и QNX Neutrino одновременно — если это так, то *приложение 1* поможет вам избежать стандартных подводных камней и написать программу так, чтобы она была переносима в обе операционные системы.

Скорая помощь

В *приложении 2* вы найдете сведения, куда обращаться, если вы зашли в тупик, нашли ошибку или когда вам просто нужен совет.

Глоссарий

Здесь дается толкование ряда используемых в книге терминов.

Другие источники информации

В дополнение к специализированному интерфейсу ядра, в QNX Neutrino также реализованы многие промышленные стандарты. Это позволяет вам "подкармливать" ваших любимых издателей, покупая литературу по стандартным функциям ANSI, POSIX, TCP/ IP и т. д.

О Роберте Кртене

Роб Кртен (да-да, именно Кртен; придуманная им самим шуточная кличка Кертен с легкой руки Николая Горбунова стала восприниматься русскоязычным читателем как фамилия Роберта. — *прим. С. З.*) выполнял (в основном контрактные) работы в области встраиваемых систем с 1986 года и занимался системным программированием с 1981 года. За период работ по трехлетнему контракту с QSS он разработал и преподавал учебные курсы "Разработка приложений реального времени для QNX Neutrino" и "Разработка администраторов ресурсов". Он также написал прототип администратора "родной" сети QNX Neutrino (Qnet), а также существенную часть учебного пособия "Построение встраиваемых систем" ("Building Embedded Systems", поставляется в комплекте электронной документации по QNX Neutrino. — *прим. ред.*).

Эта книга и предыдущая его книга, "Введение в QNX 4: Руководство по программированию приложений реального времени", были удостоены "Почетной премии" ("Award of Merit") Общества технических коммуникаций (Society for Technical Communications; **http://www.stc.org**).

Выражение признательности

Появление данной книги было бы невозможным без помощи и поддержки моих коллег, которые щедро одаривали меня своими многочисленными предложениями и комментариями. Это: Дэйв Аферсыч (Dave Athersych), Стивен Дюфрэйн (Steven

Dufrence), Томас Флетчер (Thomas Fletcher), Дэвид Гиббс (David Gibbs), Люк Базинет (Luc Bazinet), Джеймс Чанг (James Chang), Дэн Додж (Dan Dodge), Дейв Донахо (Dave Donaho), Мария Годфри (Maria Godfrey), Боб Хаббард (Bob Hubbard), Майк Хантер (Mike Hunter), Прадип Кафейл (Pradeep Kathail), Стив Марш (Steve Marsh), Дэнни Н. Прайэри (Danny N. Priarie) и Эндрю Вернон (Andrew Vernon) (прошу прощение у всех, кого я не назвал).

Особую благодарность я хотел бы выразить Брайену Стечеру (Brian Stecher), который терпеливо и внимательно рассмотрел не менее трех черновых вариантов данной книги, а также Питеру Ван Дер Вину (Peter van der Veen), который провел много ночей в моем доме (был подкуплен пивом и пиццей), выдавая мне тайны функционирования администраторов ресурсов QNX Neutrino.

Спасибо Ким Фрейзер (Kim Fraser) уже за вторую прекрасную обложку для моей книги.

Отдельное спасибо Джону Остандеру (John Ostander) за его превосходные предложения по грамматике и внимательное чтение корректуры :-).

И конечно, особую благодарность я хочу выразить моему редактору, Крису Херборту, за то, что он нашел время редактировать эту книгу, помогать мне иногда с применением мрачных SGML/LaTeX, умудряясь при этом еще делать дюжину вещей одновременно! *("Ну я же тебя просил напомнить мне, чтобы я не делал так больше!" — цитата из Криса.)*

Я также хотел бы выразить глубокую благодарность за поддержку и понимание моей жене Кристине за то, что она каждый раз терпела мое многочасовое торчание в подвале с полнейшим ее игнорированием!

Типографские соглашения

В тексте данной книги для обеспечения различимости технической терминологии используется ряд типографских соглашений. В целом примененные здесь стандарты оформления текстового материала соответствуют таковым в публикациях документов POSIX. Далее в таблице приведены образцы принятых типографских соглашений.

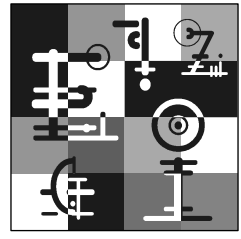
Тип текста	**Пример оформления**
Тексты программ	`if (stream == NULL)`
Опции команд	**`-lR`**
Команды	**`make`**
Переменные окружения	**PATH**
Файлы и имена путей	/dev/null
Имена функций и их параметров	*exit()*

(окончание)

Тип текста	**Пример оформления**
Комбинации клавиш	<Ctrl>+<Alt>+<Del>
Клавиатурный ввод	Текст, который вы набираете
Клавиши	<Enter>
Вывод программ	`login:`
Именованные константы	NULL
Типы данных	**unsigned short**
Литералы	0xFF, "message string"
Имена переменных	`stdin`

Этот значок указывает на что-либо важное или полезное в тексте книги.

Глава 1



Процессы и потоки

Основные понятия о процессах и потоках

Прежде чем мы начнем обсуждать потоки, процессы, кванты времени и другие замечательные "концепции диспетчеризации", давайте поговорим об аналогиях.

Сначала я хотел бы проиллюстрировать, как функционируют потоки и процессы. На мой взгляд, лучший способ (о глубинном изучении систем реального времени сейчас речь не идет) — это вообразить поведение наших потоков и процессов в некоторой привычной для нас обстановке.

Процесс как жилой дом

Давайте применим для построения аналогий о процессах и потоках объект, который мы используем повседневно — наш собственный дом.

Дом реально представляет собой контейнер с некоторыми атрибутами (общая площадь дома, число комнат и т. д.).

Если рассматривать жилой дом с этой точки зрения, он ничего не делает сам по себе. Дом — пассивный объект, в этом он аналогичен процессу. Поговорим об этом вкратце.

Потоки как обитатели дома

Люди, живущие в доме, — активные объекты: они живут в комнатах, просматривают телепрограммы, готовят пищу, принимают душ и т. д. Скоро мы поймем, что потоки функционируют аналогично.

Однопоточность

Если вы когда-либо жили в одиночестве, то знаете, каково это — вы можете делать в доме все, что пожелаете и когда пожелаете, потому что в доме больше никого нет. Если вы хотите включить стерео, принять душ, приготовить обед — вы просто идете и делаете это.

Многопоточность

Ситуация в корне изменится, если вы приведете в дом еще одного человека. Скажем, вы женитесь. Теперь у вас есть супруга, живущая в этом же доме вместе с вами. Теперь уже вы не сможете попасть в душ в любой момент времени — придется каждый раз сначала проверять, нет ли там вашей супруги.

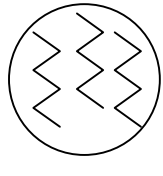
Если вы оба — взрослые и ответственные люди, о вопросах безопасности обычно можно не беспокоиться. Вы будете уверены в том, что другой совершеннолетний человек будет уважать ваши правила, принципы и жизненное пространство и не попробует тайком поджечь кухню и т. д.

А если теперь добавить в дом несколько детей — тут все станет еще интереснее.

Назад к процессам и потокам

Так же как и дом занимает некоторый участок земли в жилом массиве, так и процесс занимает некоторый объем памяти компьютера. Аналогично тому, как обитатели в доме могут свободно войти в любую комнату, в которую пожелают, потоки в процессах все вместе имеют общий доступ к этой памяти. Если поток получает доступ к некоему объекту (мама покупает игрушку), все другие потоки немедленно получают к нему доступ, потому что этот объект существует в общем адресном пространстве — в доме. Аналогично, если процесс распределяет для себя память, эта память становится доступной для всех потоков. Хитрость здесь состоит в том, что необходимо знать, должна ли эта память быть доступной для всех потоков в процессе. Если это так, то доступ потоков к ней придется синхронизировать. Если это не так, то будем считать, что эта память относится к одному конкретному потоку. В этом случае, поскольку только один поток имеет доступ к этой памяти, можно считать, что синхронизация не потребуется — не будет же этот поток сам ставить себе подножки!

Из нашего повседневного опыта мы знаем, что вещи не так просты, как кажутся... Теперь, когда мы рассмотрели основные характеристики (резюме: любой объект является разделяемым!), давайте обратимся к более интересным ситуациям и выясним, чем же они так интересны.



Процесс как контейнер потоков

На рисунке показано, как мы будем представлять потоки и процессы. Процесс здесь — это круг, отображающий "контейнерную" концепцию (адресное простран-

ство), а три ломаных линии — это потоки. Вы найдете подобные иллюстрации далее во всех разделах этой книги.

Взаимное исключение

Если вы хотите принять душ и в доме есть еще кто-то, и этот кто-то уже в ванной, вам придется подождать. Как же поток функционирует в аналогичной ситуации?

Потоки используют то, что мы называем *взаимным исключением* (mutual exclusion). Означает это в значительной степени то, о чем вы и подумали, — несколько потоков являются взаимно исключающими, когда речь идет об определенном ресурсе.

Если вы хотите принять душ, это значит, что вы хотите получить эксклюзивный доступ к ванной комнате. Для этого вы должны сначала войти в ванную, а затем закрыть ее дверь изнутри. Если при этом данной ванной комнатой попытается воспользоваться кто-либо другой, его остановит запертая дверь. После того как вы закончили свои дела в ванной, вы откроете дверь и этим позволите еще кому-либо получить доступ в душ.

Именно так и поступает поток. Поток использует объект, называемый *мьютексом* (сокращенно от MUTual EXclusion — взаимное исключение). Этот объект подобен замку в двери: как только поток заблокирует мьютекс, никакой другой поток не сможет получить доступ к мьютексу до тех пор, пока владеющий мьютексом поток его не разблокирует — иными словами, мьютекс будет удерживать другие потоки, подобно дверному замку.

Другая интересная параллель, которая проявляется как с мьютексами, так и по аналогии с дверными замками, состоит в том, что мьютекс является действительно "рекомендательной" блокировкой. Если поток не подчиняется правилам использования мьютексов, то такая защита бессмысленна. В нашей аналогии с жилым домом эта ситуация подобна тому, как кто-либо вломился бы в ванную комнату через одну из стен, игнорируя соглашение о запертой двери.

Приоритеты

А что если ванная комната в настоящее время заперта, и множество людей ожидает момента, чтобы ею воспользоваться? Очевидно, все они располагаются вне ее, ожидая, когда же тот, кто в ней находится, наконец выйдет. Закономерный вопрос: "А что произойдет, когда дверь откроется? Кто должен войти следующим?"

Можно предположить, что было бы "справедливым" позволить войти следующим тому, кто ожидает более длительное время. Или было бы "справедливо" позволить войти в ванную следующим тому, кто, например, самый старший по возрасту, или самый высокий, или самый главный. Имеется множество способов определить то, что признавать "справедливым".

Применительно к потокам мы решаем эту проблему с учетом только двух факторов: приоритета и продолжительности ожидания.

Предположим, что одновременно два человека оказываются у запертой двери в ванную комнату. Одного из них уже "поджимает" время (он опаздывает на совещание), в то время как другой тоже опаздывает, но не так уж сильно. Разве не имело бы смысл позволить тому, кого поджимает время, войти в ванную следующим? Разумеется, имело бы. Остается единственный вопрос о том, как вы принимаете решение о том, кто более "важен" в такой ситуации. Это можно сделать, например, назначив приоритет (давайте использовать номера приоритетов такие, какие приняты в Neutrino: для рассматриваемой версии Neutrino номер 1 — самый низкий, номер 255 — самый высокий). Людям в доме, которые имеют неотложные дела, следовало бы дать более высокий приоритет, а тем, у которых таких дел нет, — более низкий.

Так же дела обстоят и с потоками. Поток наследует параметры диспетчеризации у своего родительского потока, но может (если имеет соответствующие права) изменить свои дисциплину диспетчеризации и приоритет с помощью функции *pthread_setschedparam()* или только приоритет с помощью функции *pthread_setschedprio()*.

Если бы на момент разблокировки мьютекса в ожидании находилось множество потоков, мы бы отдали этот мьютекс ожидающему потоку с наивысшим приоритетом. Предположим, однако, что оба человека имеют одинаковый приоритет. Что делать? Хорошо, в этом случае было бы "справедливо" позволить человеку, который ожидал более длительное время, войти следующим. Это было бы не только "справедливо", но и аналогично тому, как это делает ядро в Neutrino. В случае, когда в ожидании находится группа потоков, мы выстраиваем их сначала по приоритету, а уже в пределах каждого приоритета — по продолжительности ожидания.

Мьютекс — конечно же, не единственное средство синхронизации из тех, которые нам доведется встретить. Давайте же рассмотрим и некоторые другие тоже.

Семафоры

Давайте переместимся из ванной комнаты на кухню, т. к. это социально адаптированное помещение для одновременного обитания более чем одного человека. На кухне вы можете не пожелать, чтобы все и каждый находились бы там одновременно. В действительности вы бы, вероятно, пожелали ограничить число людей на кухне (поваров, например).

Скажем, вы не хотите, чтобы на кухне находилось одновременно более двух человек. Смогли бы вы это реализовать с помощью мьютекса? В пределах принятого определения — нет. Почему нет? Это действительно очень интересная проблема в нашей аналогии с домом. Давайте разобьем возникшую проблему на части и проанализируем ситуацию поэтапно.

Семафор с единичным счетчиком

В ванной комнате возможна одна из двух ситуаций, каждая из которых характеризуется двумя жестко взаимосвязанными состояниями:

- дверь открыта, и в ванной комнате никого нет;
- дверь закрыта, и в помещении находится один человек.

Здесь никакая другая комбинация состояний невозможна — в пустом помещении дверь не может быть никем заперта изнутри (иначе как бы мы ее тогда открыли?) и дверь не может быть открыта кем-либо вне ванной (иначе как бы мы тогда обеспечили приватность использования?). Это и есть пример *семафора с единичным значением счетчика* — в помещении может находиться не более одного человека, или, иными словами, только один поток может использовать семафор.

Ключевым здесь (прошу прощения за каламбур) является подход к определению замка. В типовой ванной комнате вы сможете запереть и отпереть дверь только изнутри — снаружи средств для этого не предусмотрено. В действительности это означает, что блокировка мьютекса — это атомарная операция, и невозможна ситуация, в которой, пока вы находитесь в процессе блокировки мьютекса, его заблокирует некоторый другой поток, так что в результате вы оба стали бы владельцами этого мьютекса. В нашей аналогии с жилым домом это не так очевидно — хотя бы потому, что люди гораздо умнее, чем нули и единицы.

Что нам действительно потребуется на кухне, так это замок другого типа.

Семафор с неединичным счетчиком

Предположим, что мы установили в двери на кухне обычный, открываемый ключом замок. Принцип работы этого замка заключается в том, что, если у вас есть ключ, вы можете отпереть дверь и войти. Любой, кто использует этот замок, должен быть согласен с тем, что, войдя, он немедленно запрет дверь изнутри, чтобы любому, кто находится вне кухни, для входа всегда требовался бы ключ.

Ну вот, теперь управлять количеством людей, которых мы пожелали бы одновременно видеть на кухне, становится весьма легким делом — достаточно просто повесить на дверь снаружи несколько ключей. Напоминаю, что кухня должна быть всегда закрыта! Когда кто-либо пожелает попасть на кухню, он увидит, что на двери кухни висит ключ. Если это так, он возьмет этот ключ, откроет им дверь, войдет внутрь и этим же ключом закроет дверь изнутри.

Поскольку человек, входящий на кухню, должен взять ключ с собой (без этого он просто не сможет закрыть дверь изнутри), получается, что, ограничивая число висящих снаружи ключей, мы можем непосредственно управлять количеством людей, которым позволено быть на кухне в любой заданный момент времени.

При операциях с потоками подобный механизм реализуется путем применения семафоров. "Простые" семафоры работают точно так же, как и мьютексы. Вы либо являетесь владельцем мьютекса — в этом случае вы имеете доступ к ресурсу, либо нет — тогда вы не имеете доступа. Семафор, описанный выше в аналогии с досту-

пом на кухню, является *семафором со счетчиком*. Такой семафор отслеживает состояние своего внутреннего счетчика обращений (т. е. число ключей, доступных потокам).

Семафор в роли мьютекса

Мы только что задали себе вопрос: "Смогли бы мы реализовать блокировку со счетом с помощью мьютекса?" Ответ был отрицательный. А если наоборот? Смогли бы мы использовать семафор в качестве мьютекса?

Да, смогли бы. В действительности в некоторых операционных системах так все и делается — никаких мьютексов, одни семафоры! Зачем тогда вообще беспокоиться о мьютексах?

Для того чтобы ответить на этот вопрос, рассмотрим ситуацию в нашей аналогии с ванной комнатой. Как строитель вашего дома реализовал мьютекс? Я подозреваю, что в вашем доме нет ключей, которые вешались бы на двери снаружи.

Мьютексы — это семафоры "специального назначения". Если вы пожелаете, чтобы в определенном месте программы выполнялся только один поток, эффективнее всего было бы реализовать это при помощи мьютекса.

Позже мы рассмотрим и другие способы синхронизации потоков — объекты, которые называются условными переменными (condvar), барьерами (barrier) и ждущими блокировками (sleepon).

Внимание

Чтобы не возникло путаницы, необходимо также иметь в виду, что мьютекс имеет и другие свойства (например, наследование приоритетов), отличающие его от семафора.

Роль ядра

Наша аналогия с процессами в жилом доме прекрасна для объяснения концепций синхронизации, но бесполезна при анализе одной очень важной проблемы. В доме у нас было много потоков, работающих одновременно. Однако в реальной жизненной ситуации обычно имеется только один процессор, так что только один объект может реально работать в одно и то же время.

Одиночный процессор

Давайте рассмотрим, что происходит в реальном мире, и особенно в ситуации "экономии", где в системе есть только один процессор. В этом случае, поскольку имеется только один процессор, в любой заданный момент времени может выполняться только один поток. Ядро решает (с учетом ряда правил, которые мы кратко рассмотрим), какой поток должен выполняться, и запускает его.

Несколько процессоров — симметричная мультипроцессорная система (SMP)

Если вы покупаете систему, в которой имеется множество идентичных процессоров, совместно использующих одну и ту же память и устройства, это означает, что у вас есть блок SMP. (SMP расшифровывается как Symmetrical Multi-Processor — симметричный мультипроцессор; с помощью слова "симметричный" подчеркивается, что все центральные процессоры, применяемые в системе, являются идентичными.) В таком случае число потоков, которые могут работать одновременно, ограничено количеством процессоров. (Кстати, в случае с одним процессором была та же самая ситуация!) Поскольку каждый процессор может одновременно обрабатывать только один поток, в ситуации с применением множества процессоров несколько потоков могут работать одновременно. Давайте пока абстрагируемся от числа процессоров в системе — при проектировании системы бывает полезно считать, что несколько потоков могут выполняться одновременно, даже если это и не происходит в реальной ситуации. Несколько позже мы рассмотрим кое-какие неочевидные особенности симметричного мультипроцессирования.

Ядро в роли арбитра

Так кто же определяет, который из потоков должен выполняться в данный момент времени? Этим занимается ядро.

Ядро определяет, который из потоков должен использовать процессор в данный момент времени, и переключает контекст на этот поток. Давайте посмотрим, что ядро при этом делает с процессором.

Процессор имеет несколько регистров (точное их число зависит от принадлежности процессора к серии; например, сравните процессор x86 с процессором MIPS, а характерный представитель серии, например процессор 80486, — с процессором Pentium). В тот момент, когда поток выполняется, информация о нем хранится в указанных регистрах (например, данные о размещении программы в памяти).

Когда же ядро принимает решение о том, что должен выполняться другой поток, оно должно сделать следующее:

1. Сохранить текущее состояние регистров активного потока и другую контекстную информацию.
2. Записать в регистры информацию для нового потока, а также загрузить новый контекст.

Как ядро принимает решение о том, что должен выполняться другой поток? Оно анализирует, действительно ли в данный момент времени этот поток готов к использованию процессора. Когда мы обсуждали, например, мьютексы, то говорили о состояниях блокировки (это происходило в тех случаях, когда поток пытался завладеть мьютексом, уже принадлежащим другому потоку, и поэтому блокировался). Таким образом, с точки зрения ядра мы имеем один поток, который может использовать процессор, и другой поток, который не может этого делать, потому

что он заблокирован в ожидании мьютекса. В этом случае ядро предоставляет процессор потоку, который готов к работе, а другой поток заносит в свой внутренний список (чтобы можно было отслеживать запрос потока на мьютекс).

Очевидно, это не очень-то интересная ситуация. Предположим, что готовы к выполнению сразу несколько потоков. Вспомним, не мы ли делегировали доступ к мьютексу на основе приоритета и продолжительности ожидания? Ядро тоже использует подобную схему для определения того, который из потоков должен работать следующим. При этом играют роль два фактора: приоритет и дисциплина диспетчеризации. Рассмотрим их по очереди.

Концепция приоритетов

Рассмотрим два готовых к выполнению потока. Если эти потоки имеют различные приоритеты, то все просто — ядро отдает процессор потоку с высшим приоритетом. Приоритеты в Neutrino пронумерованы от единицы (самый низкий) и далее, с единичным дискретом — так же, как это было упомянуто в обсуждении получения мьютекса. Заметьте, что нулевой приоритет использовать нельзя — он зарезервирован для "холостого" (idle) потока (на профессиональном жаргоне часто называемого "холодильником" — *прим. ред.*). (Если вы захотите узнать минимальное или максимальное значение приоритета, определенное для вашей системы, используйте функции *sched_get_priority_min()* и *sched_get_priority_max()* — они описаны в **<sched.h>**. В данной книге мы будем предполагать, что приоритет 1 является самым низким, а 255 — самым высоким.)

Если другой поток с более высоким приоритетом вдруг становится готов к выполнению, ядро немедленно переключит контекст на поток с более высоким приоритетом. Это называется *вытеснением* — поток с высшим приоритетом вытесняет поток с низшим приоритетом. Когда поток с высшим приоритетом заканчивает свою работу и ядро переключает контекст обратно на поток с низшим приоритетом, который выполнялся ранее, мы называем это *возобновлением* — ядро возобновляет работу предыдущего потока.

Теперь предположим, что не один, а два потока готовы к выполнению и имеют один и тот же приоритет.

Дисциплина диспетчеризации

Предположим, что в данное время выполняется один из потоков. Рассмотрим правила, которые используются ядром при принятии решения о переключении контекста в такой ситуации. (Разумеется, все это обсуждение в действительности применимо только к потокам с одинаковыми приоритетами — как только будет готов к выполнению поток с высшим приоритетом, процессор будет отдан ему. В этом вся суть приоритетов в операционной системе реального времени.)

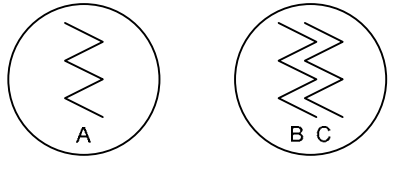
Ядро Neutrino поддерживает две основные дисциплины диспетчеризации: карусельную, она же RR (Round Robin), и FIFO (First In — First Out). (Существует также

спорадическая диспетчеризации, но она не рассматривается в этой книге; *см. разд. "Спорадическая диспетчеризация" в главе "Микроядро QNX Neutrino" руководства "Системная архитектура"*[1].)

При диспетчеризации FIFO процессор предоставляется потоку на столько времени, сколько ему необходимо. Это означает, что если один поток занят длительными вычислениями, и никакой другой поток с более высоким приоритетом не готов к выполнению, то этот поток потенциально может выполняться *вечно*. А как же потоки с тем же приоритетом? Они будут заблокированы тоже. (То, что в этот же момент потоки с более низким приоритетом будут заблокированы, должно быть очевидно.)

Если работающий поток завершает свою работу или добровольно уступает процессор, ядро анализирует состояние других потоков того же самого приоритета на готовность их к выполнению. Если таковых не имеется, то ядро анализирует потоки с более низким приоритетом, готовые к выполеннию. Заметьте, что выражение "добровольно уступить процессор" может означать одну из двух возможных ситуаций. Если поток переходит в режим ожидания, блокируется на семафоре и т. д., тогда — *да*, может выполняться поток с более низким приоритетом (как описано ранее). Но существует также специальная функция *sched_yield()*, базированная на системном вызове *SchedYield()*, по которому процессор передается *только* другому потоку с тем же самым приоритетом. Если бы был готов к выполнению поток с высшим приоритетом, у потока с низшим приоритетом все равно не было бы никаких шансов получить управление. Если поток вызывает функцию *sched_yield()*, но никакой другой поток с таким же самым приоритетом не готов к выполнению, первоначальный поток продолжает работу. В реальности функция *sched_yield()* применяется для того, чтобы дать шанс другому потоку с таким же самым приоритетом получить доступ к процессору.

На рисунке, приведенном далее, мы видим три потока, размещенных в двух различных процессах.

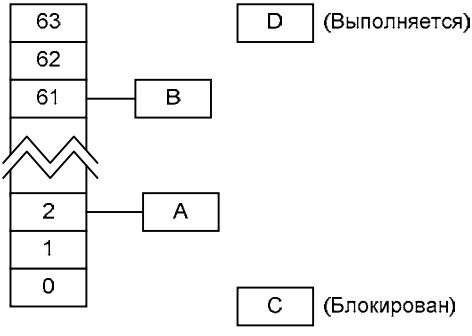


Три потока в двух различных процессах

Если мы предположим, что потоки *A* и *B* находятся в состоянии READY (готов), что поток *C* блокирован (возможно, в ожидании мьютекса), а другой поток *D*

[1] Операционная система реального времени QNX Neutrino 6.3. Системная архитектура. — СПб.: БХВ-Петербург, 2006.

(не показан) в настоящее время выполняется, то очередь готовности, которую поддерживает ядро Neutrino, будет выглядеть следующим образом:



Два потока в очереди готовности, один блокирован, один выполняется

На рисунке иллюстрируется внутренняя очередь готовности, которую использует ядро при принятии решения о том, кого запланировать на выполнение следующим. Заметьте, что поток *C* не находится в очереди готовности, потому что он блокирован, и поток *D* также не находится в этой очереди, потому что он уже выполняется.

Карусельная диспетчеризация (RR)

Дисциплина RR (карусельная диспетчеризация) аналогична дисциплине диспетчеризации FIFO, за исключением того, что поток не будет работать бесконечно, если имеется другой поток с тем же самым приоритетом. Поток будет работать только в течение предопределенного кванта времени (который фиксирован и не может быть изменен). Вы можете узнать величину кванта времени, используя функцию *sched_rr_get_interval()*. Обычно квант времени равен 4 мс, но на самом деле умноженное на 4 значение системного тика, которое вы можете узнать или задать с помощью функции *ClockPeriod()*.

Когда ядро запускает на обработку поток с дисциплиной диспетчеризации RR, оно засекает время. Если поток не блокируется в течение выделенного ему кванта времени, квант времени истечет. Тогда ядро проверяет наличие другого готового к выполнению потока с тем же самым приоритетом. Если такой поток обнаруживается, то ядро активирует его. Если такого потока нет, то ядро снова ставит на выполнение предыдущий поток (т. е. ядро выделяет потоку для работы еще один квант времени).

Постулаты

Давайте сделаем сводку правил диспетчеризации (для одиночного процессора) и отсортируем их в порядке важности:

- только один поток может выполняться в данный момент времени;

- всегда должен выполняться поток с наивысшим приоритетом;
- поток должен работать до тех пор, пока он не блокируется или не завершается;
- поток, диспетчеризуемый по дисциплине карусельного типа (RR), должен работать в течение выделенного ему кванта времени, после чего ядро обязано его перепланировать (при необходимости).

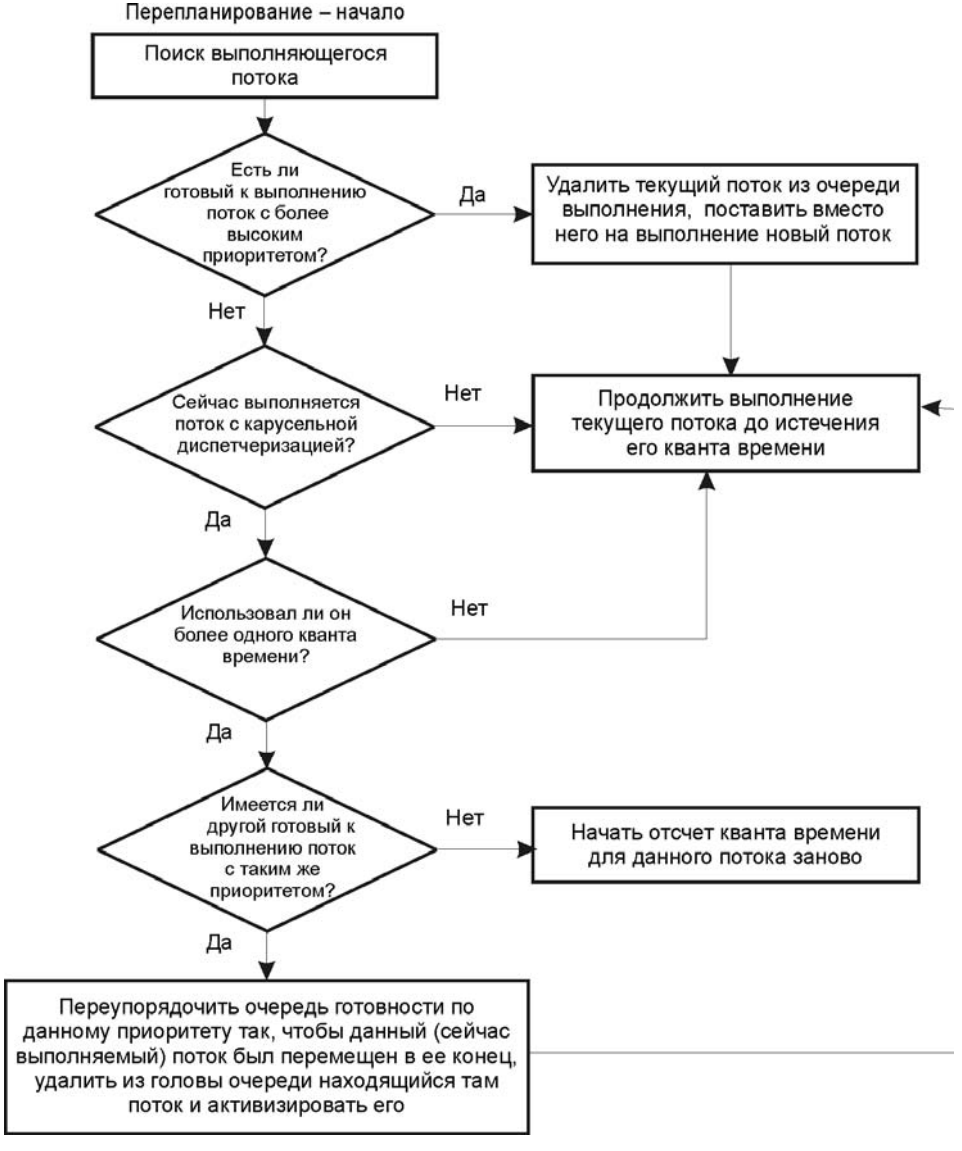


Схема алгоритма диспетчеризации

Для систем с несколькими процессорами приведенные правила остаются такими же, за исключением того, что несколько процессоров могут одновременно вы-

полнять несколько потоков. Порядок, в котором потоки выполняются (т. е. последовательность, в которой потоки ставятся на выполнение в многопроцессорной системе), определяется точно так же, как и для одиночного процессора — в любой момент времени будет выполняться готовый к выполнению поток с наивысшим приоритетом. Если существует другой готовый к выполнению поток с более высоким приоритетом, и имеется доступный процессор, то этот поток будет выполняться на следующем процессоре, и т. д. Если имеющегося числа потоков недостаточно для того, чтобы загрузить все процессоры по такому принципу, то нет проблем — "неактивные" процессоры будут выполнять "холостой" поток (его приоритет равен нулю, т. е. ниже, чем приоритет любого пользовательского потока). Если для того, чтобы обработать всю очередь, недостаточно процессоров, тогда только N потоков с наивысшим приоритетом будут выполняться, где N — число доступных процессоров. Другие потоки будут готовы к выполнению, но в действительности выполняться не будут. Отметим, что вопросы диспетчеризации потоков в симметричной мультипроцессорной системе все еще исследуются, так что вероятно, этот порядок может измениться в будущем.

Состояния потоков

Несколько раз небрежно упомянув о выполнении, готовности и блокировке, давайте теперь формализуем эти состояния потока.

Выполнение (RUNNING)

Состояние выполнения (RUNNING) в Neutrino означает, что поток активно использует ресурсы процессора. В системе SMP будет осуществляться выполнение множества потоков, а в системе с единственным процессором — выполнение одного потока.

Готовность (READY)

Состояние готовности (READY) означает, что этот поток может быть поставлен на выполнение немедленно, но не выполняется, потому что в данный момент времени активен другой поток (с таким же или более высоким приоритетом). Если бы два потока были готовы к выполнению, один из них с приоритетом 10, а другой — с приоритетом 7, то поток с приоритетом 10 был бы переведен в состояние выполнения (RUNNING), а поток с приоритетом 7 — в состояние готовности (READY).

Блокированные состояния (BLOCKED)

Что называется блокированным состоянием? Проблема здесь состоит в том, что блокированных состояний существует несколько. Реально в Neutrino имеется более десятка блокированных состояний.

Почему так много? Потому что ядро отслеживает причину, по которой поток заблокирован.

Мы уже ознакомились с двумя типами блокирующих состояний. Когда поток заблокирован в ожидании мьютекса, этот поток находится в состоянии блокировки по мьютексу (MUTEX). Когда поток заблокирован, ожидая семафор, он находится в состоянии блокировки по семафору (SEM). Эти состояния просто указывают, в очереди на какой ресурс поток заблокирован.

Если по мьютексу заблокировано несколько потоков, ядро не уделит им никакого внимания *до тех пор*, пока поток, который владеет мьютексом, не освободит его. Как только это произойдет, один из блокированных потоков будет переведен в состояние готовности (READY), и ядро при необходимости примет решение о перепланировании.

Почему "при необходимости"? У потока, который только что освободил мьютекс, вполне могут быть и другие дела, и он может иметь более высокий приоритет, чем все остальные ожидающие процессор потоки. В этом случае мы следуем второму правилу, которое гласит: "всегда должен выполняться поток с наивысшим приоритетом". Это означает, что порядок диспетчеризации не изменяется — поток с наивысшим приоритетом продолжает работать.

Полный список состояний потоков

Далее представлен полный список блокированных состояний с краткими пояснениями. Этот список, кстати, есть в заголовочном файле **<sys/Neutrino.h>**, только там эти состояния снабжены префиксом "STATE_" (например, состояние READY из данной таблицы там будет звучать как STATE_READY).

Если состояние потока	**То это значит, что**
CONDVAR	Поток ожидает соблюдения условия условной переменной
DEAD	Поток "мертв", ядро ожидает освобождения занятых им ресурсов. (В классических UNIX-системах это состояние также называют "zombie" — "зомби" — *прим. ред.*)
INTR	Поток ожидает прерывание
JOIN	Поток ожидает завершения другого потока
MUTEX	Поток ожидает захват мьютекса
NANOSLEEP	Поток "спит" (приостановлен на определенный период времени)
NET_REPLY	Поток ожидает ответ, который должен быть доставлен по сети
NET_SEND	Поток ожидает доставку через сеть своего сообщения или импульса
READY	Поток не выполняется, но готов к работе (работает один или более потоков с более высокими или равными приоритетами)

(окончание)

Если состояние потока	То это значит, что
RECEIVE	Поток ожидает сообщение от клиента
REPLY	Поток ожидает от сервера ответ на свое сообщение
RUNNING	Поток выполняется
SEM	Поток ожидает захват семафора
SEND	Поток ожидает приема своего сообщения сервером
SIGSUSPEND	Поток ожидает сигнал
SIGWAITINFO	Поток ожидает сигнал
STACK	Поток ожидает распределения дополнительного стекового пространства
STOPPED	Поток приостановлен (по сигналу SIGSTOP)
WAITCTX	Поток ожидает, когда его контекст станет доступным в регистрах (только для SMP-систем)
WAITPAGE	Поток ожидает устранения администратором процессов повреждения на странице
WAITTHREAD	Ожидание создания потока

Важно помнить о том, что, когда поток блокирован, независимо от *состояния блокировки*, он не потребляет ресурсы процессора. Наоборот, единственным состоянием, в котором поток потребляет ресурсы процессора, является состояние выполнения (RUNNING).

Мы рассмотрим блокированные состояния SEND (блокировка по передаче), RECEIVE (блокировка по приему) и REPLY (блокировка по ответу) в *главе 2*. Состояние NANOSLEEP связано с применением функций типа *sleep()*, которые мы рассмотрим в *главе 3*. Состояние INTR связано с использованием функции *InterruptWait()*, которую мы изучим в *главе 4*. Большинство всех прочих состояний обсуждается в данной главе.

Процессы и потоки

Вернемся к нашим рассуждениям о потоках и процессах, но на сей раз с точки зрения перспективы их применения в системах реального времени. Затем мы рассмотрим вызовы функций, которые применяются при работе с потоками и процессами.

Мы знаем, что процесс может содержать один или больше потоков. (Процесс с нулевым числом потоков не был бы способен что-либо *делать:* если в доме никого нет, выполнять какую-либо полезную работу просто некому.) В операционной системе Neutrino допускается один или более процессов. (Аналогично — Neutrino с нулевым количеством процессов просто не сможет ничего сделать.)

Что же делают все эти процессы и потоки? В конечном счете, они формируют систему — собрание потоков и процессов, реализующих определенную цель.

На самом высоком уровне абстракции система состоит из множества процессов. Каждый процесс ответственен за обеспечение служебных функций определенного характера, независимо от того, является ли он элементом файловой системы, драйвером дисплея, модулем сбора данных, модулем управления или чем-либо еще.

В пределах каждого процесса может быть множество потоков. Число потоков варьируется. Один разработчик ПО, используя только единственный поток, может реализовать те же функциональные возможности, что и другой, использующий пять потоков. Некоторые задачи сами по себе приводят к многопоточности и дают относительно простые решения, другие в силу своей природы являются однопоточными, и свести их к многопоточной реализации достаточно трудно.

Проблемы разработки ПО с применением потоков могли легко стать темой отдельной книги. Здесь же мы изложим только основы этой проблемы.

Почему процессы?

Почему же не взять просто один процесс с множеством потоков? В то время как некоторые операционные системы вынуждают вас программировать только в таком варианте, возникает ряд преимуществ при разделении объектов на множество процессов.

К таким преимуществам относятся:

- возможность декомпозиции задачи и модульной организации решения;
- удобство сопровождения;
- надежность.

Концепция разделения задачи на части, т. е. на несколько независимых задач, является очень мощной. И именно такая концепция лежит в основе Neutrino. Операционная система Neutrino состоит из множества независимых модулей, каждый из которых наделен некоторой зоной ответственности. Эти модули независимы и реализованы в отдельных процессах. Разработчики из QSS использовали эту удобную особенность для отдельной разработки модулей, независимых друг от друга. Единственная возможность установить зависимость этих модулей друг от друга — наладить между ними информационную связь с помощью небольшого количества строго определенных интерфейсов.

Это естественно ведет к упрощению сопровождения программных продуктов, благодаря незначительному числу взаимосвязей. Поскольку каждый модуль четко